Fallbasiertes Schließen zur Unterstützung der Wiederverwendung objektorientierter Software: Eine Fallstudie[1]

Ralph Bergmann[*] und Ulrich Eisenecker[+]
[*]Universität Kaiserslautern, Fachbereich Informatik,
Postfach 3049, D-67653 Kaiserslautern
[+]Daimler-Benz AG, Forschung und Technik
Postfach 2360, D-89013 Ulm

Zusammenfassung

Bei der Erstellung komplexer Software spielt die *Wiederverwendung* vorhandener Programmbestandteile eine besonders große Rolle, da hierdurch sowohl die Software-Qualität gesteigert, als auch der gesamte Erstellungs- und Wartungsaufwand erheblich reduziert werden kann. In jüngster Zeit gewinnen *objektorientierte* Programmiersprachen zunehmend an Bedeutung, da die Wiederverwendung hierbei bereits durch Sprachkonzepte wie z.B. Vererbung und Polymorphie unterstützt wird. Weiterhin besteht jedoch das Problem, zur Wiederverwendung geeignete Programmbestandteile aufzufinden. Ziel dieser Arbeit ist es herauszufinden, inwieweit fallbasiertes Schließen nach dem aktuellen Stand der Kunst die Wiederverwendung objektorientierter Software unterstützen kann. Hierzu wurde eine entsprechende Anwendung prototypisch auf der Basis des INRECA-Systems entwickelt. Durch ausgewählte Testsituationen wurden Erfahrungen mit diesem Prototyp gesammelt und systematisch ausgewertet.

1 Einleitung

Die objektorientierte Software-Entwicklung (z.B. Jacobson et al., 1992) betrachtet das *Objekt* als gemeinsame Einheit aller Software-Entwicklungsphasen. Objekte mit gemeinsamen Eigenschaften werden zu einer Klasse zusammengefaßt. Die Schnittstelle

1 Dieses Projekt wurde gefördert von der Daimler-Benz AG, Forschung und Technik, Ulm. Die Realisierung dieses Ansatzes erfolgt mit dem fallbasierten System INRECA, das von der Europäischen Union als ESPRIT-Projekt P-6322 gefördert wird. Das INRECA-Konsortium besteht aus AcknoSoft (Paris), IMS (Dublin), tecInno GmbH (Kaiserslautern) und der Universität Kaiserslautern.

einer Klasse nach Außen besteht aus den *Methoden*. *Vererbung* ist eine Relation zwischen Klassen, die eine Implementierung einer Klasse basierend auf einer anderen existierenden Klasse erlaubt.

Die bei der objektorientierten Software-Entwicklung angestrebte Wiederverwendung bereits existierender Komponenten führt zu zahlreichen Vorteilen. Insbesondere beeinflußt die Wiederverwendbarkeit alle wesentlichen Aspekte von Software-Qualität (Meyer, 1990). Wenn Wiederverwendung stattfindet, wird weniger Aufwand für die Neuentwicklung von Software-Bestandteilen erbracht. Dadurch kann bei gleichen oder - eine entsprechende Häufigkeit der Wiederverwendung vorausgesetzt - sogar bei sinkenden Gesamtkosten mehr Aufwand bezüglich Korrektheit, Robustheit usw. bei der Entwicklung der wiederverwendbaren Komponenten getrieben werden.

Diesen Vorteilen der Wiederverwendung stehen jedoch mehrere Schwierigkeiten gegenüber: Es werden geeignete Dokumentationsformen zur Beschreibung wiederverwendbarer Komponenten benötigt. Die Information über verfügbare Komponenten muß von ihren Produzenten potentiellen Kunden zugänglich gemacht werden. Zur Verwaltung großer Informationsbestände bezüglich wiederverwendbarer Komponenten werden geeignete Verfahren und Werkzeuge benötigt, wozu in diesem Papier ein wesentlicher Beitrag vorgestellt wird.

Fallbasiertes Schließen (Kolodner 1980; Bartsch-Spörl, 1987; Althoff & Wess 1992; Kolodner, 1993) bedeutet in einer allgemeinen Form die Lösung eines neuen Problems durch Erinnerung an eine vergangene ähnliche Erfahrung und die Verwendung der Information und des Wissens dieser Situation (Aamodt & Plaza, 1994). Jede vergangene Erfahrung wird traditionell als *Fall* bezeichnet und die Menge mehrerer solcher Erfahrungen ergibt eine *Fallbasis*. Desweiteren ist die Fähigkeit, sich aufgrund *wesentlicher Merkmale* des vorliegenden Problems in angemessener Zeit an eine ähnliche Situation zu erinnern, eine wichtige Voraussetzung des fallbasierten Schließens. Gegebenenfalls muß der erinnerte Fall auf die aktuelle Situation übertragen werden. Dazu bedarf es unter Umständen einer *Modifikation* dieses Falls derart, daß den Unterschieden zwischen der alten Situation und der neuen Problematik Rechnung getragen wird (Koton, 1988; Hammond 1989; Veloso & Carbonell 1993; Bergmann, Pews & Wilke, 1994). Schließlich sollte die so entstandene erfolgreiche neue Lösung geprüft und in der Fallbasis gespeichert werden. Diese vier Phasen des fallbasierten Schließens werden als *Retrieve, Reuse, Revise* und *Retain* (Aamodt & Plaza, 1994) bezeichnet.

Die allgemeine Grundidee der wiederholten Verwendung der Lösungen bereits bearbeiteter Probleme ist der Wiederverwendung von Software und dem fallbasierten Schließen gemein. Betrachtet man sowohl die Wiederverwendung von Software als auch das fallbasierte Schließen auf dieser sehr allgemeinen Ebene, so könnte man die These vertreten, daß jede Form der Wiederverwendung von Software auch eine Art des fallbasierten Schließens darstellt. Bei dieser abstrakten Betrachtungsweise wird aber nicht berücksichtigt, daß jede allgemeine Strategie konkretisiert werden muß, um zur Lösung eines spezifischen Problems beizutragen. Hierzu leistet diese Papier einen Beitrag.

Zur Zeit sind bereits einige Ansätze und Systeme zur Wiederverwendung von Software mittels fallbasierten Schließen entwickelt worden, wie etwa GTE (Gish, Huff & Thomson, 1994), Deja Vu (Smyth & Cunningham, 1992), APU (Bhansali & Harandi, 1993), CAESAR (Fouque & Matwin, 1993), oder AIRS(Ostertag, Hendler et al.,1992). Alle diese Systeme sind in (Reinartz, Althoff & Bergmann 1994; Reinartz, 1994) ausführlich untersucht und verglichen worden. Dabei hat sich gezeigt, daß jedes dieser Systeme eine Fallbasis und einen Retrieval-Mechanismus in der ein oder anderen Form besitzt, jedoch keines der Systeme die *objektorientierte Wiederverwendung* auch nur ansatzweise unterstützt.

In der nachfolgend beschriebenen Untersuchung zur Wiederverwendung objektorientierter Software mittels fallbasiertem Schließen wurden hauptsächlich Smalltalk-80-Klassen aus zwei Domänen (Collection-Klassen und User-Interface-Klassen) zu Grunde gelegt. Die dabei gewonnenen Ergebnisse und Erkenntnisse lassen sich sicherlich innerhalb bestimmter Grenzen auch auf die Problematik der Wiederverwendung bei anderen objektorientierten Sprachen wie z.B. C++ übertragen.

2 Wiederverwendung objektorientierter Software

Als wesentliche Merkmale objektorientierter Programmierung können die Kapselung von Objektzustand und Methoden, Vererbung und Polymorphie genannt werden. Ein wichtiger Unterschied mit erheblichen Auswirkungen besteht darin, ob eine objektorientierte Programmiersprache streng typisiert oder typfrei ist. Diese Unterscheidung geht in der Regel mit einer unterschiedlichen Realisierung der Polymorphie einher, nämlich vererbungsgebundenem vs. signatur-gebundenem Polymorphismus (Eisenecker, 1993). Weitere Unterschiede ergeben sich daraus, ob multiple Vererbung und Überladen zulässig sind und Generizität realisiert wird. Alle diese Besonderheiten, die z.B. wesentlich zwischen Smalltalk und C++ differenzieren, haben zum Teil erhebliche Auswirkungen auf die Wiederverwendbarkeit.

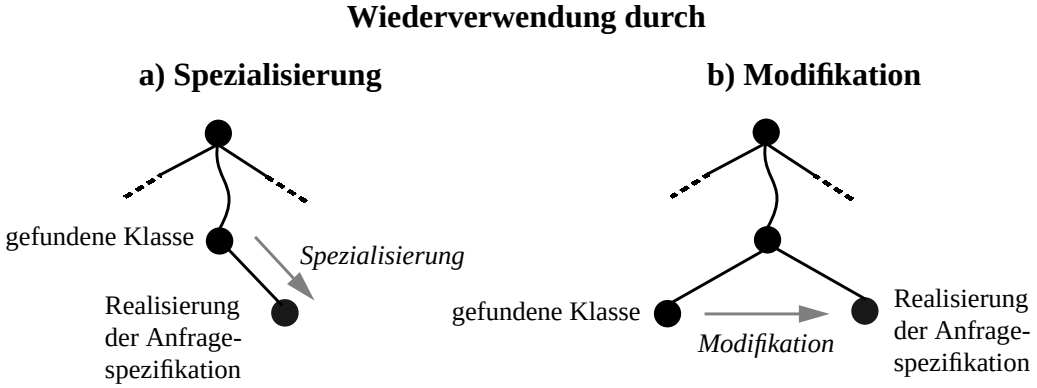

Abbildung 1: Zwei Wiederverwendungsmöglichkeiten

Die wichtigste Grundlage für die Wiederverwendung ist das Abstraktionsmittel der Vererbung, wodurch auch gleichzeitig die Klasse in den Fokus der Wiederverwendung gerückt wird. Wird eine neue Klasse benötigt, so muß diese nicht von Grund auf neu beschrieben werden. Vielmehr genügt es, auf eine bereits bestehende Klasse in Form einer Vererbungsbeziehung hinzuweisen, und dann bestehende Unterschiede und Erweiterungen zu spezifizieren. Diese Form der Wiederverwendung wird im folgenden *Spezialisierung* genannt (siehe Abb. 1a). Eine Besonderheit der Spezialisierung ergibt sich in typisierten Sprachen, wenn abgeleitete Klassen so entworfen werden, daß ihre Semantik gegenüber der Vorfahr-Klasse nicht verändert wird (Liskov, 1988).

Eine weitere Form objektorientierter Wiederverwendung besteht darin, eine vorhandene Implementierung zu *modifizieren*, um eine neue Funktionalität zu realisieren (siehe Abb. 1b). Dies ist dann erforderlich, wenn es nicht sinnvoll ist, die bestehende Klasse weiter zu spezialisieren, obwohl diese funktionale Bestandteile enthält, die auch für die aktuellen Erfordernisse geeignet sind. In diesem Fall muß die neue Implementierung als “Geschwisterklasse” der bestehenden Klasse realisiert werden. Hierzu können bestimmte Methoden der bestehenden Klasse übernommen bzw. geeignet modifiziert werden. Diese Art der Modifikation ist im Prinzip auch eine Spezialisierung, und zwar eine Spezialisierung der Oberklasse der wiedergefundenen Klasse. Jedoch enthält in diesem Fall die wiedergefundene Klasse zusätzliche wiederverwendbare Bestandteile und ist somit der Oberklasse vorzuziehen. Diese Art der Wiederverwendung birgt allerdings erhebliche Risiken im Bezug auf die Wartbarkeit. Da gewissermaßen mit "Copy & Paste" gearbeitet wird, ist im Grunde anschließend nur noch die Vererbungsbeziehung zur Elternklasse stabil. Veränderungen in der Elternklasse wirken sich in beiden abgeleiteten Klassen noch aus. Wird hingegen in der Geschwisterklasse ein Implementierungsdetail geändert, welches auch für die durch "Copy & Paste" gewonnene Klasse relevant ist, existiert kein Mechanismus mehr, der diese Veränderung automatisch nachzieht oder auch nur diesbezüglich eine Warnung generiert (Verletzung des Single-Source-Prinzips). Beide Arten objektorientierter Wiederverwendung sind nützlich und sollten daher fallbasiert unterstützt werden. Daher werden im folgenden Ähnlichkeitsmaße definiert, mit denen zumindest ansatzweise versucht werden soll, die Wiederverwendungskosten abzuschätzen.

Im Rahmen des Software-Engineerings sind bereits einige Methoden und Systeme entwickelt worden, die die Wiederverwendung von Software unterstützen. Die einfachsten Systeme beruhen auf klassischen Freitextrecherchen und Stichwortsuchverfahren, wobei zusätzlich kontrollierte Vokabulare und Thesauri eingesetzt werden können. Darauf beruhend wurden leistungsfähigere Suchverfahren wie etwa "concept matching" und facettenbasierte Verfahren definiert. Z.B. werden im System CLIS ein hierarchisch aufgebauter Thesaurus, ein Begriffshierarchie-Graph (hierarchical-concept graph) und eine Distanzfunktion für das Retrieval verwendet (Cho, Kim & Kim, 1990). Im System AIRS werden facettenbasierte Verfahren zusammen mit auf semantischen Netzen beruhenden Verfahren, die eine mehrdimensionale Klassifikation erlauben, eingesetzt (Ostertag, Hendler, Prieto-Diaz & Braun, 1992). Neben diesen rein klassifikationsbasierten Verfahren werden auch Methoden der formalen Spezifikation, die vor allem auch semantische

Aspekte besser abdecken können, für das Retrieval wiederverwendbarer Software-Komponenten genutzt. Ein solches Verfahren, welches auf der Verfeinerung von Spezifikationen, die in einem gerichteten azyklischen Graph repräsentiert werden, wird von (Mili, Mili & Mittermeir, 1994) beschrieben. In diesem Ansatz wird die Software-Komponente zur Wiederverwendung vorgeschlagen, die in einer Halbordnung spezifizierter Merkmale die Spezifikation in einer Suchanfrage am detailliertesten erfüllt. Wird eine solche Komponente nicht gefunden, liefert eine Ähnlichkeitssuche die Software-Bausteine, welche der Suchspezifikation am nähesten kommen. Drei auf der Verwendung von deskriptiver Logik beruhende Systeme, nämlich LaSSIE, COMET und KITSS, werden in (Devanbu & Jones, 1994) beschrieben, wobei z.B. durch die das Modul-Design unterstützende Funktion von COMET auch Wiederverwendungsaspekte mitabgedeckt werden. Die Ansätze der genannten Systeme wurden nicht für das Retrieval objektorientierter Software genutzt und teilweise auch nur anhand eng umgrenzter Beispiele evaluiert. Keines der in den Systemen verwendeten Verfahren zur Ähnlichkeitsbestimmung bzw. zur Ermittlung des aufgrund der formalen Spezifikation am besten geeigneten Kandidaten berücksichtigt sowohl syntaktische als auch funktionale Merkmale gleichermaßen angemessen. Desweiteren wird häufig auch nicht überprüft, ob die zur Wiederverwendung vorgeschlagenen Lösungen tatsächlich nutzbar sind. Eine detaillierte Untersuchung dieser und weiterer Systeme, der in ihnen verwendeten Ähnlichkeitsmaße und der zugrundegelegten Bibliothekskonzepte findet sich in (Reinartz, Althoff & Bergmann, 1994).

3 Fallbasiertes Schließen zur Wiederverwendung objektorientierter Software

Das fallbasierte Schließen (Kolodner, 1980; Bartsch-Spörl, 1987; Althoff & Wess, 1992; Aamodt & Plaza, 1994; Kolodner, 1993) beschäftigt sich mit Methoden zum Retrieval von Erfahrungen und der Adaption der gespeicherten Lösungen auf das neue Problem. Um die Methode des fallbasierten Schließens zur Wiederverwendung objektorientierter Software einsetzen zu können, muß zunächst die Frage erörtert werden, was überhaupt als *Fall* betrachtet werden soll. Die naheliegende und einfach zu realisierende Möglichkeit besteht darin, die *Programmobjekte*, die wiederverwendet werden sollen, als Fall zu repräsentieren. Hierbei wird eine *Spezifikation* der Programmobjekte als *Problembeschreibung* des Falles angesehen und die zugehörige Implementierung (der konkrete Code) als *Lösung* betrachtet. Bei einer solchen Realisierung besteht eine Anfrage an das fallbasierte System (im folgenden als *Anfragefall* bezeichnet) aus einer Programmspezifikation ohne zugehörige Lösung. Ziel des Retrievals ist es dann, aus der Fallbasis einen Fall auszuwählen, der eine *ähnliche* Spezifikation aufweist wie die aktuelle Spezifikation im Anfragefall. Der Grad der Ähnlichkeit soll dabei den Aufwand, der für die Wiederverwendung erforderlich ist, widerspiegeln. Somit muß sämtliche Information über objektorientierte Wiederverwendungsmöglichkeiten in das Ähnlichkeitsmaß eingebracht werden.

3.1 Merkmale zur Fallbeschreibung

Im folgenden soll die Frage erörtert werden, auf welche Art objektorientierte Programme beschrieben werden können. Damit eng verbunden ist die Frage nach dem Aufwand zur Erhebung dieser Beschreibungen. Im allgemeinen kann erwartet werden, daß eine reichhaltigere Programmbeschreibung adäquatere Informationen für die Ähnlichkeitsbestimmung zur Verfügung stellt und ein besseres Gesamtergebnis erwarten läßt. Dies wird jedoch meist durch einen erhöhten Aufwand zur Erstellung der Fallbeschreibung erkauft, insbesondere wenn die erforderlichen Merkmale nicht automatisch aus den vorliegenden Programmen extrahiert werden können. Deshalb sollte es das Ziel sein, die automatisch extrahierbaren Informationen so gut wie möglich auszunutzen. Bei Beschreibungsmerkmalen, die durch den Benutzer manuell vorgegeben werden müssen, sollte hingegen eine erhöhte Wiederfindungsgenauigkeit gegen den Erhebungsaufwand abgewogen werden.

Im folgenden werden zwei verschiedene Arten von Merkmalen unterschieden: *syntaktische* Merkmale zur Beschreibung der Programmschnittstelle (z.B. Namengebung, Typinformation), sowie *funktionale* Merkmale, welche die Funktion des Programms abstrakt spezifizieren.

3.1.1 Syntaktische Merkmale

Im objektorientierten Paradigma besteht die Schnittstelle zu Programmen (Objekten) aus einer Ansammlung von Methodenaufrufen. Jeder Methode können beim Aufruf Argumente übergeben werden. Da die genaue Schnittstelle bei der Implementierung der Klassen deklariert werden muß, liegt es nahe, diese Information in die Programmbeschreibung aufzunehmen. Für jede Klasse können somit die Namen der Methoden und für jede Methode, die Anzahl und die Klasse der Aktualparameter und des Rückgabewertes berücksichtigt werden. Methoden, die von der Oberklasse geerbt werden, gehören ebenfalls zu dieser Beschreibung. An dieser Stelle soll noch einmal ausdrücklich darauf hingewiesen werden, daß der Verwendung syntaktischer Merkmale zur Auswahl wiederverwendbarer Programme die implizite Annahme zugrunde liegt, daß syntaktische Ähnlichkeiten (z.B. ähnlich bezeichnete Methoden oder Kategorien) von Programmen auch auf funktionale Ähnlichkeiten hindeuten. Eine weitere implizite Annahme ist, daß gleiche Funktionalitäten auch auf Methoden mit dem gleichen Namen abgebildet wird. Dies ist in der Praxis nicht unbedingt der Fall (Meyer, 1990).

3.1.2 Funktionale Merkmale

Mit Hilfe von funktionalen Merkmale soll die Funktion der wiederzuverwendenden Klasse beschrieben werden. Für einfache und gut untersuchte Bereiche ist es vereinzelt möglich, formale Spezifikationen zu erstellen. Wir erachten diese Art von Spezifikationen jedoch für die praktische Anwendung als unrealistisch, da sie einen extrem hohen Erstellungs- und Axiomatisierungsaufwand erfordern und auch nur für wenige eingeschränkte Bereiche tragfähig sind. Praktikabler erscheint jedoch die Funktionsbeschrei-

bung mit Hilfe abstrakter, bereichsspezifischer Merkmale zu sein. Diese Merkmale müssen, wie für jede Art der Wiederverwendung von Software, als Ergebnis einer entsprechenden *Domänenanalyse* (Prieto-Diaz, 1990) für jeden Anwendungsbereich separat erhoben werden. Wir wollen im folgenden funktionale Merkmale für die Collection-Klassen der Smalltalk-80-Klassenbibliothek bestimmen. Diese sind sehr gut verstanden, so daß abstrakte Beschreibungen bereits in der Literatur zu finden sind (Goldberg, 1983; Cook, 1992). Collection-Klassen können wie folgt beschrieben werden:

- *ElementOrdnung*: beschreibt, ob die Elemente einer Collection *ungeordnet* (z.B. Dictionary), *extern geordnet* (z.B. Array) oder *intern geordnet* (z.B. SortedCollection) sind.
- *ElementArt*: nennt die Smalltalk-Klasse, deren Instanzen als Elemente der Collection verwaltet werden können. Dies ist bei vielen Collections (z.B. OrderedCollection) die Klasse *Object* und bei einigen spezielleren Klassen (z.B. String) die Klasse *Character.*
- *ElementDuplikate*: gibt an, ob Elemente *mehrfach* in einer Instanz vorkommen können (z.B. Klasse Bag) oder ob jedes Element nur genau *einmal* auftreten darf (z.B. Klasse Set).
- *ElementZugriff*: bezeichnet, auf welche Art auf die Elemente einer Collection zugegriffen werden kann. Dies kann ein Zugriff über einen Schlüssel sein, der auf Gleichheit getestet wird (*SchlüsselZugriffGleichheit*; z.B. Klasse Dictionary), ein Zugriff über einen Schlüssel, der auf Identität getestet wird (*SchlüsselZugriffIdentität*; z.B. Klasse IdentityDictionary) oder ein beliebiger anderer Zugriff ohne Schlüssel (*KeinSchlüsselZugriff*; z.B. Klasse Set).

3.2 Fallrepräsentation

Um Techniken des fallbasierten Schließens einsetzen zu können, müssen Fälle erfaßt und repräsentiert werden. Wir setzen hierzu die Fallrepräsentationssprache CASUEL (Common Case Representation Language) (Manago, Bergmann et al. 1993) ein, die im Rahmen des ESPRIT-Projektes INRECA (Althoff, Wess, Bergmann et. al., 1993; Manago, Conruyt & Althoff, et. al., 1993) für Klassifikations- und Diagnoseaufgaben entwickelt wurde. CASUEL erlaubt *objektorientierte Fallstrukturen* mit *Multi-Value Slots*. In objektorientierten Fallstrukturen besteht ein Fall aus einer Ansammlung von Objekten. Ein Objekt wird durch Slots (Attribute) beschrieben, als deren Werte wiederum neue Objekte eingetragen werden können. Bei Multi-Value Slots ist es dabei zusätzlich erlaubt, daß mehrere Werte oder Objekte in einen einzelnen Sollt eingetragen werden. Mit dieser Ausdrucksstärke repräsentiert CASUEL den Stand der Forschung im Bereich der Fallrepräsentationssprachen für Klassifikationsaufgaben.

Für den hier betrachteten Bereich wurde ein Fall (eine Smalltalk-80-Klasse) aus mehreren Objekten aufgebaut. Hierbei beschreibt genau ein Objekt den Fall selbst. Dieses Objekt besitzt Attribute, in denen wiederum Objekte für die Methodenbeschreibung enthalten sind, sowie ein Objekt für die funktionalen Merkmale. Zusätzlich enthält dieses Objekte noch einen Verweis auf den zugehörigen Code der Smalltalk-Klasse. Die Attribute für Instanzvariablen und Methoden sind hierbei Multi-Value Slots, da eine

Klasse in der Regel durch mehrere Instanzvariablen und Methoden beschrieben ist. Die Objekte für die Methoden und Instanzvariablen enthalten nun die Attribute, die in Abschnitt 3.1.1 als syntaktische Merkmale eingeführt wurden. Das Objekt zur funktionalen Beschreibung enthält die funktionalen Merkmale, die in Abschnitt 3.1.2 erklärt wurden.

3.3 Retrieval von Fällen und Ähnlichkeitsmaße

Die zentrale Frage beim *Retrieval* von Fällen ist die nach einer adäquaten Bestimmung der Ähnlichkeit zweier Fallbeschreibungen. Der Anfragefall ist zu einem Fall in der Fallbasis dann ähnlich, wenn der Fall in der Fallbasis zur Implementierung der Spezifikation, die im Anfragefall angegeben ist, verwendet werden kann. Der Grad der Ähnlichkeit sollte dabei einen Hinweis geben, wieviel Aufwand die Wiederverwendung des Falls in der Fallbasis zur Lösung des aktuellen Problems erfordert. Der Aufwand der Wiederverwendung läßt sich jedoch im Allgemeinen nicht a priori bestimmen. Erst wenn die aktuelle Programmspezifikation durch Wiederverwendung des vorgeschlagenen Programms bereits vollständig implementiert ist, kann der Aufwand, den diese Vorgehensweise erfordert hat, genau bestimmt werden. Die Aufgabe der Ähnlichkeitsbestimmung ist es also, eine gute a priori-Abschätzung eines a posteriori-Kriteriums zu liefern.

Die Bestimmung der Ähnlichkeit erfolgt für die definierte Fallrepräsentation, d.h. aufgrund der syntaktischen und funktionalen Merkmale. Zur Berechnung werden hierbei zunächst verschiedene lokale Ähnlichkeitsmaße für die einzelnen Merkmale definiert. Diese Merkmalsähnlichkeiten werden dann zu einer Ähnlichkeit für die Objekte der Fallrepräsentation (Instanzvariablen, Methoden und Funktionsbeschreibung) kombiniert. Die Kombination mehrerer lokaler Ähnlichkeitsmaße zu einer globalen Ähnlichkeitsbeurteilung erfolgt in der Regel durch eine gewichtete Mittelwertbildung. Hierbei besteht durch die Angabe der Gewichte die Möglichkeit, bestimmte Aspekte mehr oder weniger stark zur Ähnlichkeitsbetrachtung heranzuziehen. Im folgenden werden einzelne Ähnlichkeitsmaße genauer diskutiert.

3.3.1 *Ähnlichkeit von Methoden*

Bei der Ähnlichkeitsbestimmung von Methoden stellt sich das Problem, daß diese als Multi-Value Slots modelliert sind, da in der Regel für jede Klasse mehrere Methoden und Instanzvariablen angegeben werden müssen. Dies bedeutet aber, daß jeder Methode im Anfragefall eine entsprechende Methode eines Falls in der Fallbasis zugeordnet werden muß. Auf der Grundlage einer solchen Zuordnung wird erst die Ähnlichkeit der *einzelnen Methoden* bestimmt und dann zu einer Gesamtähnlichkeit über alle Methoden hinweg gemittelt. Um die Ähnlichkeit zweier Multi-Value Slots zu bestimmen, ist es erforderlich, eine *optimale* Zuordnung von Objekten der beiden Slots zu finden. Diese Zuordnung zeichnet sich durch eine maximale gemittelte Gesamtähnlichkeit aller zugeordneten Paare aus. Eine solche optimale Zuordnung kann jedoch nur dadurch gefunden werden, daß alle möglichen Zuordnungen bestimmt und bezüglich der Gesamtähnlich-

keit beurteilt werden. Die kombinatorische Komplexität dieses Vergleichsverfahren liegt jedoch bei $O(n_F! / (n_F - n_Q)!)$ (n_F = Anzahl der Objekte im Multi-Value Slot der Fallbasis, n_Q = Anzahl der Objekte im Multi-Value Slot des Anfragefalls, $n_Q < n_F$) und ist somit nicht praktikabel. Daher wird im folgenden ein quadratischer *Hillclimbing*-Algorithmus skizziert (Komplexität: $O(n_F * n_Q)$), der die optimale Zuordnung approximiert und die Ähnlichkeit für einen Multi-Value Slot bestimmt. Hierbei ist die Ähnlichkeit sim_S, zum Vergleich zweier Objekte, hier der einzelnen Methoden, vorausgesetzt.

```
Algorithmus: MVS(sim_S)

Sei MVS_Q die Menge der Objekte im Anfragefall
Sei MVS_F die Menge der Objekte im Fall aus der Fallbasis
S:= 0
Für alle Objekte O aus MVS_Q:
   Falls MVS_F nicht leer ist:
      Bestimme O' aus MVS_F so daß sim_S(O,O') maximal ist.
      S:= S + sim_S(O,O')
      Entferne O' aus MVS_F
sim_MVS:= S / |MVS_Q|
```

Bei diesem Algorithmus ist zu beachten, daß er die maximale Gesamtähnlichkeit nur annähert und nicht exakt bestimmt. Die Objekte in der Anfrage werden sequentiell dem *lokal* ähnlichsten Objekt der Fallbasis zugeordnet. Bei den Methoden erfolgt diese lokale Ähnlichkeitsbestimmung aufgrund der Ähnlichkeit von Methodenname, Methodenkategorie, Argumentanzahl, Argumentklassen und Rückgabeklasse (siehe auch 3.3.2). Diese lokal optimale Zuordnung garantiert jedoch in keiner Weise auch eine globale Optimierung der Gesamtähnlichkeit. Dieses wirkt sich beim Retrieval so aus, daß nicht mehr garantiert werden kann, daß der ähnlichste Fall auch gefunden wird. Es ist jedoch zu erwarten, daß sich dieser theoretische "Fehler" in praktischen Situationen nicht auswirkt. Auch wären die Kosten für eine absolut korrekte Ähnlichkeitsbestimmung nicht akzeptabel. Selbst die Komplexität des skizzierten Algorithmus ist im Vergleich zu den Ähnlichkeitsbestimmungen für konventionelle Slots nicht unkritisch.

3.3.2 *Ähnlichkeit von Argument- und Rückgabeklassen*

Die Argumentklassen einer Methode sind als Taxonomie repräsentiert, die die Smalltalk-Klassenhierarchie wiederspiegelt. Über der Taxonomie läßt sich ein Ähnlichkeitsmaß definieren, wobei im folgenden zwischen der Anfrageklasse (*Q*) und der Klasse in der Fallbasis (*F*) unterschieden wird. Die Anfrageklasse und die Klasse aus der Fallbasis besitzen in der Taxonomie einen speziellsten gemeinsamen Vorgänger (*V*). Die Länge des Pfades von diesem gemeinsamen Vorgänger *V* zur Klasse *Q* wird im folgenden mit d_Q bezeichnet und analog wird die Länge des Pfades von *V* zu *F* mit d_F bezeichnet (siehe Abb. 2).

Betrachten wir nun die Situation in der d_F=0 und d_Q>0 ist, d.h. der Fall in der Fallbasis bezeichnet eine Klasse als Argument einer Methode, die eine Oberklasse der Klasse ist, die in der Anfrage gefordert wird. In dieser Konstellation kann die Methode wiederverwendet werden, da die zur Verfügung stehende Methode sogar allgemeiner ist als die geforderte. Jedoch kann diese unnötige Allgemeinheit dazu führen, daß die Methodenimplementierung nicht so effizient ist, wie sie sein könnte. Deshalb ist die Ähnlichkeit in Abhängigkeit von d_Q etwas abzustufen.

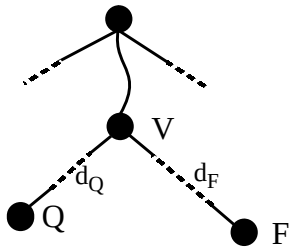

Abbildung 2: Ähnlichkeit von Klassen

Im umgekehrten Fall, bei dem der Fall in der Fallbasis eine speziellere Klasse bietet als in der Anfrage gefordert wird (d_F>0 und d_Q=0), kann jedoch die vorhandene Implementierung nicht direkt wiederverwendet werden. Sie muß verallgemeinert werden. Daher muß in diesem Fall die Ähnlichkeitsbewertung (in Abhängigkeit von d_F) niedrigere Werte liefern als im zuvor betrachteten Fall. Dies wird mit der folgenden Ähnlichkeitsfunktion realisiert (*H* bezeichnet die Tiefe des Klassenbaumes):

$$sim = 1 - d_F\left(\frac{1}{H+1}\right) - d_Q\left(\frac{1}{H(H+1)}\right)$$

3.4 Fallretrieval

Nachdem nun einige Ansätze zur Definition von Ähnlichkeit zwischen einem Anfragefall und einem Fall in der Fallbasis vorgestellt worden sind, soll jetzt die Frage beantwortet werden, wie der ähnlichste Fall aus der Fallbasis aufgefunden werden kann.

3.4.1 Lineare Suche

Die einfachste Art des Zugriffs auf die Fallbasis besteht in einem sequentiellen Durchsuchen. Hierbei muß der Anfragefall mit jedem Fall der Fallbasis verglichen und die Ähnlichkeit bestimmt werden. Mit diesem Verfahren wird natürlich garantiert, daß der ähnlichste Fall gefunden wird, jedoch nur mit erheblichem Suchaufwand (*O(n)*, *n*=Anzahl der Fälle). Dies ist bei kleinen Fallbasen unkritisch, kann jedoch bei großen Fallbasen leicht zum Problem werden. Aber gerade bei großen Bibliotheken ist eine fallbasierte Wiederverwendung von Software von Interesse.

3.4.2 Vorselektion mittels kd-Baum

Eine Möglichkeit die Fallbasis effizienter zu durchsuchen besteht darin, zuvor eine Indexstruktur aufzubauen. Eine Möglichkeit hierzu bietet der *kd-Baum,* ein multidimensionaler binärer Suchbaum (Bentley, 1975; Friedman et al., 1977; Broder, 1990), der es ermöglicht, das Retrieval im fallbasierten Schließen deutlich zu beschleunigen (Öchsner & Wess, 1992; Wess, Althoff et al. 1993). Die Idee hierbei ist es, einen Baum aufzubauen, der den Suchraum in einzelne Bereiche aufteilt, die eine bestimmte Anzahl zueinander ähnlicher Fälle abdecken. Jeder Knoten im kd-Baum repräsentiert dabei eine bestimmte Teilmenge von Fällen der Fallbasis. Jeder innere Knoten partitioniert die ihm zugeordnete Fallbasis in zwei disjunkte Teilmengen. Die Blätter des Baumes, genannt *Buckets*, enthalten eine geringe Anzahl zueinander ähnlicher Fälle, die nicht weiter aufgeteilt werden. Die Partitionierung eines Knotens im Baum in seine Teilknoten erfolgt durch die Auswahl eines Attributes und eines Wertes aus dessen Wertebereich. Die Fälle werden dann danach aufgeteilt, ob die Ausprägung des gewählten Attributes größer oder kleiner als der ausgewählte Wert ist. Der Aufbau eines solchen Baumes kann nun nach verschiedenen Varianten erfolgen, je nachdem, auf welche Weise Attribute und Werte ausgewählt werden (siehe Wess, Althoff et al. 1993). Im Mittel erlaubt der kd-Baum das Wiederfinden des ähnlichsten Falls mit einem Aufwand von $O(log_2\ n)$ (n=Anzahl der Fälle).

Ein Problem beim Einsatz dieses Verfahrens für die beschriebene Art der Wiederverwendung von objektorientierter Software besteht in den Anforderungen an die Fallrepräsentation. So ist der kd-Baum nicht in der Lage, mit Multi-Value Slots umzugehen. Diese sind jedoch, wie gezeigt, zur Modellierung von Methoden und Instanzvariablen erforderlich. Eine Möglichkeit, dennoch vom Performanzgewinn dieses Ansatzes zu profitieren, besteht darin, den kd-Baum als erste Filterstufe lediglich auf den funktionalen Merkmalen einzusetzen, die keine Multi-Value Slots erfordern, um eine gewisse Anzahl von funktional ähnlichen Fällen effizient auszuwählen. Die so ausgefilterte Fallmenge kann dann in einer zweiten Stufe linear durchsucht werden, um den ähnlichsten Fall in Bezug auf die volle Fallbeschreibung auszuwählen. Dadurch sollte es möglich sein, auch bei relativ großen Fallbasen zu erträglichen Retrieval-Zeiten zu kommen.

4 Praktische Erfahrungen mit den Beispielanwendungen

Im folgenden sollen einige praktischen Erfahrungen skizziert werden, die mit einer protoypischen Realisierung des dargestellten Ansatzes gesammelt wurden. Diese Implementierung umfaßt neben dem bislang eingeführten Beispielbereich der Collection-Klassen zusätzlich einen zweiten Bereich, nämlich die Klassen des INRECA-Systems, welche die Benutzungsoberfläche von INRECA realisieren. Diese sind keine Standard-Klassen einer Bibliothek, sondern sind sehr anwendungs-spezifisch. Für diesen Bereich wurden, ähnlich wie für die Collection-Klassen, eigene funktionale Merkmale festgelegt.

Die Fallbeschreibungen wurden für beide Bereiche soweit wie möglich automatisch aus dem Smalltalk-System generiert. Die beschriebenen funktionalen Merkmale wurden manuell hinzugefügt. Für jeden der beiden Beispielbereiche wurde eine eigene Fallbasis aufgebaut.

4.1 Testsituationen

Ziel der Tests war es, einen Eindruck vom Klassifikationsverhalten des Systems unter verschiedenen Bedingungen zu bekommen. Hierzu wurden bestimmte Anfragesituationen für jede der beiden Fallbasen festgelegt. Jede Anfragesituation entspricht dabei im wesentlichen einem bereits in der Fallbasis enthaltenen Fall, wobei jedoch die spezifizierten Methoden auf einige wenige beschränkt wurden. Für die Collection-Klassen wurden als Anfragesituationen alle Klassen ausgewählt, die Blätter in der Vererbungshierarchie darstellen (insgesamt 17 verschiedene Anfragen). Von diesen wurden dann auch nur die Methoden aus den Kategorien *adding*, *accessing* und *removing* in die Anfrage aufgenommen. Bei der Methodenauswahl liegt die Idee zugrunde, daß es unrealistisch ist anzunehmen, daß ein Anwender alle Methoden in der Anfrage spezifiziert. Er wird immer nur die für ihn am meisten relevanten Methoden angeben. Die Beurteilung der Wiederverwendbarkeit (Spezialisierung oder Modifikation) eines Falls richtet sich bei dieser Untersuchung nur nach der Position innerhalb der Klassenhierarchie. Hierbei wird implizit angenommen, daß die bestehende Smalltalk-80 Klassenhierarchie besonders gut durchdacht ist und möglichst wenige redundante Implementierungen enthält.

Ein weiteres Ziel dieser Untersuchung war es, einen Eindruck von der Nützlichkeit des beschriebenen fallbasierten Ansatzes in Abhängigkeit von den verschiedenen Merkmalsgruppen zu bekommen. Deshalb wurden für jede der beiden Fallbasen Retrievaltests mit syntaktischen und funktionalen Merkmalen, nur den syntaktischen Merkmalen und nur den funktionalen Merkmalen durchgeführt.

4.2 Ergebnisse

Im folgenden werden die daraus gewonnenen Ergebnisse systematisch zusammengefaßt und bewertet. Dies ist jedoch nicht als methodisch abgesicherte Untersuchung anzusehen, sondern eher als Sammlung von Einzelbeobachtungen, die auf einer recht geringen Anzahl von Tests beruhen (17 bzw. 24 Anfragen aus 30 bzw. 49 Fällen).

4.2.1 Beobachtung 1: Ähnlichkeit zum ursprünglichen Fall

Da jeder Anfragefall im wesentlichen einem Fall aus der Fallbasis entspricht, ist beim Retrieval zu erwarten, daß dieser Fall auch als ähnlichster Fall gefunden wird. Diese Erwartung wurde bei beiden Fallbasen unter jeder Bedingung bestätigt. Falls ein Programm, das ohne Modifikation oder Spezialisierung eingesetzt werden kann, bereits in

der Fallbasis enthalten ist, so wird dieses auch als ähnlichstes Programm gefunden. Jedoch kann es sein, daß einige andere Programme in der Fallbasis ebenfalls dieselbe Ähnlichkeit besitzen wie der ursprüngliche Fall.

4.2.2 Beobachtung 2: Menge der Fälle mit gleicher Ähnlichkeit

Im folgenden möchten wir davon ausgehen, daß sich der Anfragefall selbst nicht in der Fallbasis befindet. Wir betrachten deshalb die Menge aller übrigen Fälle, die die höchste Ähnlichkeit zum Anfragefall besitzen. Hierbei fällt auf, daß es je nach Anfrage und Ähnlichkeitsvariante verschieden große Gruppen von Fällen gibt, die dieselbe (höchste) Ähnlichkeit zum Anfragefall besitzen. Zwischen diesen Fällen kann nicht weiter differenziert werden. Sie werden vom fallbasierten System als gleich gut zur objektorientierten Wiederverwendung angesehen. In der folgenden Tabelle ist die durchschnittliche Anzahl der Fälle mit gleicher Ähnlichkeit in Abhängigkeit der Fallbasis und der Ähnlichkeitsvariante dargestellt.

TABELLE 1. Durchschnittliche Anzahl der Fälle/Prozentsatz der Fallbasis mit max. Ähnlichkeit

Fallbasis	**nur funktionale Merkmale**	**nur syntaktische Merkmale**	**funktionale und syntaktische Merkmale**
Collection-Klassen	4.3 Fälle (14.1%)	3.9 Fälle (13.1%)	1.65 Fälle (5.5%)
Benutzungsoberflächen-Klassen	4.2 Fälle (8.5%)	1.54 Fälle (3.1%)	1.46 Fälle (3.0%)

Man erkennt, daß bei den Collection-Klassen die syntaktischen und funktionalen Merkmale für sich genommen jeweils relativ große Fallmengen auswählen. Die Kombination beider Merkmalsklassen erlaubt es jedoch, die Menge der relevanten Fälle drastisch zu reduzieren. Offensichtlich sind diese beiden Merkmalgruppen unabhängig voneinander. Bei der Benutzungsoberfläche zeigen sich ähnliche Ergebnisse. Jedoch scheint es, daß sich hier die Fälle syntaktisch stärker unterscheiden als die Fälle der Collection-Klassen. Dies ist vor allem darauf zurückzuführen, daß bei den Collection-Klassen bereits eine starke Angleichung der Namengebung durch die Smalltalk-Entwickler stattgefunden hat.

4.2.3 Beobachtung 3: Korrektheit des Retrievals

Die entscheidende Frage für die Nützlichkeit des gewählten Ansatzes für die Wiederverwendung von Software ist jedoch die Frage nach der Korrektheit des Retrievals. Daher wurde im folgenden untersucht, bei wievielen Anfragen mindestens *ein* geeigneter Wiederverwendungskandidat innerhalb der Gruppe der Fälle mit höchster Ähnlichkeit vertreten war. Die Ergebnisse sind in der folgenden Tabelle dargestellt:

TABELLE 2. Anteil der Anfragen mit mindestens einem nützlichen Fall höchster Ähnlichkeit

Fallbasis	nur funktionale Merkmale	nur syntaktische Merkmale	funktionale und syntaktische Merkmale
Collection-Klassen	100%	59%	71%
Benutzungsoberflächen-Klassen	100%	88%	100%

Werden nur die funktionalen Merkmale zur Ähnlichkeitsbestimmung herangezogen, so ist bei jeder Anfrage mindestens ein für die Wiederverwendung nützlicher Fall in der Gruppe der ähnlichsten Fälle enthalten. Jedoch ist diese Gruppe, wie Tabelle 1 zeigt, im Mittel aller Anfragen sehr groß. Daher relativiert sich dieses scheinbar optimale Klassifikationsergebnis.

Werden nur die syntaktischen Merkmale betrachtet, so reduziert sich die Trefferquote erheblich. Insbesondere bei den Collection-Klassen zeigt sich ein schlechtes Klassifikationsergebnis trotz einer großen Gruppe von Fällen mit höchster Ähnlichkeit. Werden funktionale und syntaktische Merkmale kombiniert, so ergibt sich eine mittlere bis gute Trefferquote bei kleiner Gruppengröße.

4.2.4 Beobachtung 4: Art der Wiederverwendung

Wie bereits erläutert wurde, werden sowohl die Oberklasse der im Anfragefall spezifizierten Klasse als auch eine bestimmte Menge der Geschwister als nützliche Wiederverwendungskandidaten betrachtet. Eine Oberklasse muß zur Wiederverwendung spezialisiert werden, wohingegen eine Geschwisterklasse modifiziert werden muß. Jedoch ist es einem ausgewählten Fall nicht a priori anzusehen, für welche Art der Wiederverwendung er geeignet ist. Folglich muß diese Entscheidung dem Benutzer überlassen bleiben. Daher soll im folgenden untersucht werden, welche Art der Wiederverwendung durch das Ähnlichkeitsmaß bevorzugt wird. In Tabelle 3 sind die gefundenen Fälle mit höchster Ähnlichkeit nach der Art der erforderlichen Wiederverwendung aufgeschlüsselt.

TABELLE 3. Durchschnittlicher Prozentsatz wiederverwendbarer Fälle innerhalb der Gruppe mut höchster Ähnlichkeit, aufgeteilt nach der Art der Wiederverwendung

Fallbasis	nur funktionale Merkmale	nur syntaktische Merkmale	funktionale und syntaktische Merkmale
Collection-Klassen	Spez.: 27% Mod.: 39%	Spez.: 24% Mod.: 13%	Spez.: 29% Mod.: 31%
Benutzungsoberflächen-Klassen	Spez.: 17% Mod.: 83%	Spez.: 33% Mod.: 55%	Spez.: 25% Mod.: 75%

Im allgemeinen werden mehr Klassen gefunden, die zur Wiederverwendung eine Modifikation erfordern, als solche, für die eine Spezialisierung notwendig ist. Dabei scheinen

tendenziell die syntaktische Merkmale spezialisierbare Klassen etwas stärker zu bevorzugen als die funktionalen Merkmale. In jedem Fall müssen allerdings immer beide Arten der Wiederverwendung erwogen werden.

Generell kann der Grund für einen größeren Anteil modifizierbarer Klassen auch darin bestehen, daß für eine Anfrage in der Regel mehr modifizierbare Klassen in der Fallbasis vorhanden sind als spezialisierbare Klassen. Dieser Unterschied ist vor allem bei den Benutzungsoberflächen-Klassen ausgeprägt. Bei diesen zeigt sich auch die Bevorzugung zu modifizierender Klassen besonders deutlich.

5 Resümee

Fallbasiertes Schließen läßt sich zur Unterstützung der Wiederverwendung objektorientierter Software prinzipiell einsetzen. Dies bedarf jedoch einiger Voraussetzungen, die zum Teil vor dem Hintergrund der Wahl von Smalltalk als Programmiersprache für die ausgewählten Klassen zu sehen sind:

- Der erfolgreiche Einsatz erfordert einen gewissen Beschreibungsaufwand für die wiederzuverwendenden Programme. Hierbei haben sich insbesondere funktionale Beschreibungen als sehr nützlich herausgestellt. Jedoch müssen diese in der Regel manuell erhoben werden.
- Syntaktische Merkmale, die zwar ggfs. automatisch erhoben werden können, sind nur von begrenztem Nutzen. Ihre Verwendbarkeit hängt im wesentlichen davon ab, ob der Programmierer syntaktisch ähnlichen Bezeichnungen auch eine ähnliche Funktionalität gegeben hat.
- Der Benutzer muß in der Lage sein, eine Anfrage durch die gewählten syntaktischen und funktionalen Merkmale zu beschreiben. D.h. er muß mit der Beschreibungsmethodik des Bereichs, aus dem die wiederzuverwendenden Programme stammen, prinzipiell vertraut sein. Dies ist für funktionale Merkmale als wesentlich einfacher anzusehen als für die syntaktischen Merkmale.

Unter diesen Voraussetzungen kann ein fallbasiertes System folgendes leisten:

- Wenn sich ein Programm in der Fallbasis befindet, das direkt, also ohne Spezialisierung oder Modifikation, verwendbar ist, so wird dieses in jedem Fall gefunden.
- Es werden in der Regel immer mehrere Fälle mit höchster Ähnlichkeit zur Anfrage aufgefunden. Die Anzahl dieser Fälle hängt stark von den verwendeten Merkmalen ab. Meistens sind aber nur einige dieser Fälle tatsächlich zur Wiederverwendung geeignet.
- Die Verwendung funktionaler Merkmale und geeigneter Ähnlichkeitsmaße liefert eine gute Vorhersagekraft für die Auswahl eines wiederverwendbaren Programmes, bedingt aber ggfs. eine relativ große Menge gleich ähnlicher Fälle.
- Die Kombination von syntaktischen und funktionalen Merkmalen kann es ermöglichen, die Anzahl der aufgefundenen Fälle mit gleicher Ähnlichkeit zu reduzieren, ohne den Anteil wiederverwendbarer Programme deutlich zu vermindern.

Dennoch ergeben sich einige Probleme mit diesem Ansatz:

- Bei dem beschriebenen Ansatz wird lediglich die Programmspezifikation zur Ähnlichkeitsbestimmung verwendet, nicht jedoch eine Implementierungsbeschreibung. Deshalb ist nicht möglich, zur Beurteilung des Wiederverwendungsspektrums eines Programms die Implementierung der Methoden zu berücksichtigen.
- Das Retrieval ist im allgemeinen sehr aufwendig. Für große Fallbasen ist der lineare Algorithmus nicht praktikabel. Indexierungsmethoden, wie beispielsweise ein kd-Baum, arbeiten nicht mit Multi-Value Slot-Repräsentationen, die aber zur adäquaten Beschreibung von syntaktischen und funktionalen Merkmalen erforderlich sind. Um ein kombiniertes Retrievalverfahren (linear und kd-Baum) einsetzen zu können, müssen möglichst wenig relevante funktionale Informationen in Multi-Value Slots repräsentiert werden.

Darüber hinaus verdienen zwei Aspekte noch eine genauere Untersuchung:

- Es wäre wünschenswert, den beschriebenen Ansatz auch auf eine Auswahl von Klassen einer typgebundenen Programmiersprache wie z.B. C++ oder Eiffel anzuwenden, wobei gegebenenfalls auch geprüft werden könnte, inwieweit möglicherweise anteilig vorhandene formale Spezifikationen ausgenutzt werden können. Sicherlich wäre es auch lohnenswert, die Bedeutung bestimmter Aspekte einzelner objektorientierter Programmiersprachen wie etwa Templates (Schablonen) und multiple Vererbung im Hinblick auf den Einsatz fallbasierter Methoden zur Wiederverwendung gezielt zu untersuchen.
- Es sind Auswertungen in der Praxis, auch im Anschluß an den Einsatz fallbasierter Retrieval-Werkzeuge des vorgehend beschriebenen Typs, erforderlich, die dazu dienen herauszufinden, auf welcher Ebene (Methoden, Klassen, Subsysteme) und in welcher Reihenfolge der tatsächliche Wiederverwendungsbedarf besteht.
- Neben der Wiederverwendung von Code sind auch Analyse- und Design-Spezifikationen sowohl textueller als auch grafischer Art potentielle Wiederverwendungskandidaten. Analyse- und Design-Dokumente weisen grundsätzlich den Vorteil einer gewissen Unabhängigkeit von bestimmten Programmiersprachen oder Entwicklungsumgebungen auf. Von daher ist die Suche nach ähnlichen Problemen in der Analyse-Phase oder nach bereits vorhandenen Entwürfen für ähnliche Probleme in der Design-Phase grundsätzlich interessant. Es gibt jedoch mehrere Schwierigkeiten, die diesem Ansatz derzeit noch im Wege stehen. Hierbei ist insbesondere die große Zahl der verwendeten Notationen und Prinzipien zu nennen (Stein, 94). Außerdem werden vorhandene Analyse- und Designwerkzeuge in der Praxis häufig nicht umfassend eingesetzt.

6 Literatur

Aamodt, A. & Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, **7(1)**, pp. 39-59.

Althoff, K.-D. , Wess, S. , Bergmann, R. , Maurer, F. , Manago, M. , Auriol, E., Conruyt, N. ,Traphöner, R. , Bräuer, M. & Dittrich, S. (1993). Induction and case-based reasoning for classification tasks. *17th Anual Conference of the German Society for Classification.*

Althoff, K.-D. & Wess, S. (1992). Case-Based Reasoning and Expert System Development. In: Schmalhofer, F., Strube, G., & Wettter, T. (Hrsg.), *Contemporary Knowledge Engineering and Cognition*, Heidelberg: Springer Verlag, pp. 146-158.

Bartsch-Spörl, B. (1987). Ansätze zur Behandlung von fallorientiertem Erfahrungswissen in Expertensystemen. *Künstliche Intelligenz*, 1987.

Bentley, J.L. (1975). Multi-dimensional binary search trees used for associative retrieval tasks. *CACM* **Vol**. 18, pp. 509-517.

Bergmann, R. , Pews, G. & Wilke, W. (1994). Explanation-based similarity: A unifying approach for integrating domain knowledge into case-based reasoning for diagnosis and planning. *Topics in Case-Based Reasoning*. Lecture Notes in AI, Vol. 837, pp. 182-196, Springer.

Bhansali, S. & Harandi, M. T. (1993). Synthesis of UNIX programs using derivational analogy. *Machine Learning*, 10, pp. 7-55.

Broder, A. J. (1990). Strategies for efficient incremental nearest neighbor search. *Pattern Recognition* **Vol**. 23, pp. 171-178.

Cho, W. G. , Kim, Y. W. & Kim, J. H. (1990). CLIS: A Software Reuse Library System with A Knowledge Based Information Retrieval Model. In: H. Tanaka (Ed.), Artificial Intelligence in the Pacific Rim: *Proceedings of the Pacific Rim International Conference on Artificial Intelligence*, pp. 402-407

Cook, W. R. (1992). Interfaces and Specifications for the Smalltalk-80 Collection Classes. *OOPSLA '92*, pp. 1-15.

Devanbu, T. & Jones, M.A. (1994). The Use of Description Logics in KBSE systems. Experience Report. *Proceedings of the 16th International Conference on Software Engineering*. LosAlamitos, California: IEEE Computer Society Press, pp. 23-35

Eisenecker, U. W. (1993). Objektorientierung und Wiederverwendbarkeit. *unix/mail*, **11**(**6**), pp. 420-429.

Fouque, G. & Matwin, S. (1993). A case-based approach to software reuse. *Journal of Intelligent Information Systems*, 1, pp. 165-197.

Friedman, J. H. , Bentley, J. L. & Finkel, R. A. (1977). An algorithm for finding best matches in logarithmic expected time, *ACM Transactions Mathematical Software* **Vol. 3** pp. 209-226.

Gish, J. W., Huff, K. E. & Thomason, R. (1994). The GTE environment - Supporting understanding and adaptation in software reuse. In: Schäfer, W., Prieto-Diaz, R., & Matsumoto, M., *Software reusability*, pp. 140-149.

Goldberg, A. & Robson, D. (1983). *Smalltalk-80 - The language and its implementation*. Addison-Wesley.

Hammond, K.(1989). *Case-Based Planning*. London: Academic Press.

Jacobson, I. , Christerson, M., Jonsson, P. & Övergaard, G. (1992). *Object-Oriented Software Engineering*. Menlo Park, CA: Addison-Wesley.

Kolodner, J. (1980). Retrieval and organizational strategies in conceptual memory: A computer model. Ph. D. Thesis, Yale University.

Kolodner, J. (1993b). *Case-based reasoning*. San Mateo, CA: Morgan Kaufmann.

Koton, P. (1988). Using experience in learning and problem solving. MIT, Laboratory of Computer Science, Ph. D. Dissertation, October 1988, MIT/LCS/TR-441.

Liskov, B. (1988). Data Abstraction and Hierarchy. *SIGPLAN Notices* 23, 5.

Manago, M., Bergmann, R. , Conruyt, N. , Traphöner, R., Paisley, J. , Renard, J. L. , Maurer, F., Wess, S., Althoff, K.-D. & Dumont, S. (1993). CASUEL: A Common Case Representation Language. *Deliverable D1, Esprit Project 6322 INRECA*, Universität Kaiserslautern.

Manago, M. , Conruyt, N. , Althoff, K.-D. , Maurer, F. , Wess, S. & Traphöner, R. (1993). Induction and reasoning from cases. *First European Workshop on Case-Based Reasoning* (EWCBR-93).

Meyer, B. (1990). Lesson from the design of the EIFFEL libraries. *Communications of the ACM*, Vol. **33**, Np. **9**, pp. 68-88.

Mili, A., Mili, R. & Mittermeir, R. (1994). Storing and Retrieving Software Components: A Refinement Based System. *16th International Conference on Software Engineering*. Los Alamitos, California: IEEE Computer Society Press, pp. 91-100.

Öchsner, H. & Wess, S. (1992). Ähnlichkeitsbasiertes Retrieval von Fallbeispielen durch assoziative Suche in einem mehrdimensionalen Datenraum. In: Althoff, K.-D. , Wess, St., Bartsch-Spörl, B., Janetzko, D. (Eds.) *Ähnlichkeit von Fällen in Systemen des fallbasierten Schließens*, SEKI Working Paper, Universität Kaiserslautern.

Ostertag, E., Hendler, J. , Prieto-Diaz, R. & Braun, Chr. (1992). Computing Similarity in a Reuse Library System: An AI-Based Approach. ACM Transactions on Software Engineering and Methodology. Vol 1, No. 3, pp. 205-228.

Prieto-Diaz, R. (1990). Domain analysis: An introduction. *ACM SIGSOFT Software Engineering Notes*, **Vol. 18**, **No. 4**, pp. 75-82.

Reinartz, T., Althoff, K.-D. & Bergmann, R. (1994). Studie: Objektorientierte Wiederverwendung und Fallbasiertes Schließen; Teil 2: Literaturüberblick. *Interner Bericht: Universität Kaiserslautern.*

Reinartz, T. (1994). Fallbasiertes Schließen zur Wiederverwendung von Ausschnitten der Smalltalkklassenbibliothek. *Diplomarbeit Universität Kaiserslautern*.

Smyth, B. & Cunningham, P. (1992). Deja Vu: A hierarchical case-based reasoning system for software design. *Proceedings of the 10th European Conference on Artificial Intelligence*, pp. 587-589.

Stein, W. (1994). *Objektorientierte Analysemethoden. Vergleich, Bewertung, Auswahl*. B.I. Wissenschaftsverlag.

Veloso, M. & Carbonell, J. (1993). Towards scaling up machine learning: A case study with derivational analogy in PRODIGY. In: *Machine Learning Methods for Planning*. Morgan Kaufmann.

Wess, S. , Althoff, K.-D. & Derwand, G. (1993).Improving the Retrieval Step in Case-Based Reasoning; *EWCBR '93*, pp. 83-88, Universität Kaiserslautern.