Der *ReUse Class Browser*

Integration deskriptiver und navigierender Suche in objektorientierten Klassenbibliotheken

Bernhard Convent
IBM Wissenschaftliches Zentrum Heidelberg
Vangerowstraße 18
D-6900 Heidelberg

Thomas Kohaut, Stefan Kuhlins*
Universität Mannheim
Schloß
D-6800 Mannheim 1

Workshop „Objektorientierte Technologien – Möglichkeiten und Grenzen“,
Bayerisches Forschungszentrum für wissensbasierte Systeme,
FORWISS REPORT FR–1993–011

Zusammenfassung

Von der Software-Wiederverwendung erhofft man sich bei der Software-Entwicklung eine erhöhte Produktivität bei gleichzeitig besserer Qualität. Momentan scheint dazu das objektorientierte Paradigma am besten geeignet zu sein, weil Konzepte für die Wiederverwendung bereits in Analyse- und Design-Methoden sowie Programmiersprachen enthalten sind. In solchen Entwicklungsumgebungen sind Klassen die wiederverwendbaren Einheiten, und somit werden leistungsfähige Class Browser benötigt, die die Software-Entwickler bei der Suche in großen und unbekannten Klassenbibliotheken unterstützen.

In diesem Bericht wird erläutert, warum zur Unterstützung der Wiederverwendung Class Browser sowohl deskriptive als auch navigierende Suchfunktionalität benötigen. Aufbauend auf einer Darstellung der theoretischen Grundlagen beider Sucharten, stellen wir mit dem *ReUse Class Browser* eine prototypische Integration beider Konzepte für die Unterstützung der Software-Wiederverwendung in C++-Klassenbibliotheken vor.

1 Motivation

Die von vornherein geplante Entwicklung und Nutzung *wiederverwendbarer Software-„Bausteine“* verspricht eine erhebliche Produktivitäts- und Qualitätssteigerung bei der Anwendungsentwicklung [End88, BP89a, BP89b, Kru92]. Auch wenn diese Vision des Entwurfs neuer Software-Systeme auf der Grundlage von qualitativ hochwertigen, wiederverwendbaren Bausteinen bereits 25 Jahre alt ist

*email: kuhlins@wifo.uni-mannheim.de

[McI68], so unterstützt erst die *objektorientierte Software-Entwicklung* diesen Ansatz in einem umfassenden Maße. Objektorientierte Analyse- und Designmethoden und zugehörige Programmiersprachen basieren direkt auf einer Vielzahl von mächtigen, die Wiederverwendung unterstützenden Konzepten wie z.B. Einkapselung, Vererbung, Polymorphismus, Überschreiben und spätes Binden. Als wiederverwendbare Einheiten werden dabei im wesentlichen die Klassen angesehen, die in der Art eines abstrakten Datentyps Daten und zugehörige Operationen zusammenfassen und nur über definierte Schnittstellen nach außen zur Verfügung stellen.

Mit dem Ziel der Wiederverwendung werden Klassen (häufig getrennt nach speziellen Anwendungsbereichen) in umfangreichen *Klassenbibliotheken* gesammelt und verwaltet. Diese enthalten neben bekannten Klassen aus der engeren Programmiersprachensystemumgebung oder aus selbst durchgeführten Projekten in der Regel auch eine Vielzahl von für den einzelnen Software-Entwickler völlig unbekannten Klassen. Diese stammen z.B. aus fremden Entwicklungsprojekten oder aus Spezialanwendungen wie etwa die Realisierung von Benutzungsoberflächen, Parser oder die persistente Verwaltung von Daten mittels einer Datenbank. Für eine erfolgreiche Nutzung solcher Klassenbibliotheken (und damit für den Erfolg der Wiederverwendung überhaupt) ist ein Werkzeug zur Suchunterstützung von größter Bedeutung. Diese Aufgabe übernimmt ein sogenannter *Class Browser*.

Welche Funktionalität erwartet man von einem solchen Class Browser? Beim Entwurf eines Software-Systems beschreibt der Entwickler die benötigte Funktionalität in einer ersten Phase zunächst nur sehr grob und noch ungenau, um diese Beschreibung als Suchanfrage an die Klassenbibliothek zu geben. Als Ergebnis dieser *deskriptiven Suche* erhält der Entwickler eine Menge von möglicherweise verwendbaren Klassen (vgl. auch "class selection" [GTC$^+$90]). In einer zweiten Phase geht es nun darum, die Kandidatenklassen genauer zu verstehen, um die Verwendbarkeit entscheiden bzw. den möglicherweise notwendigen Modifikationsaufwand abschätzen zu können (vgl. auch "class exploration" [GTC$^+$90]). Dabei nutzt man unter anderem aus, daß zwischen den Klassen einer Bibliothek eine Vielzahl von semantischen Beziehungen (z.B. der Vererbung, der Spezialisierung oder Generalisierung, der Benutzung, der Versionierung) bestehen. Diese Beziehungen „vernetzen" die Klassen untereinander, und indem man diesen Beziehungen folgt, ist es möglich, das Umfeld einer Klasse zu erkunden, um so das Verständnis erheblich zu erleichtern. Diese Art der Suche wird *navigierende Suche* genannt.

Ein Class Browser muß somit zur Unterstützung der Wiederverwendung eine *zweiphasige Suche* mit deskriptivem und navigierendem Anteil zur Verfügung stellen [CGGW92]. Heute verfügbare Browser konzentrieren sich fast ausschließlich auf die navigierende Suche und bieten nur äußerst rudimentäre Unterstützung für eine initiale, deskriptive Suche. Statt dessen wird eigentlich implizit angenommen, daß die grundsätzliche Struktur der Klassenbibliothek bekannt ist, so daß eine navigierende Suche prinzipiell ausreicht. Dies ist eine mögliche Erklärung für die verhältnismäßig lange Lernphase beim Programmieren in einer Entwicklungsumgebung für eine objektorientierte Programmiersprache, da der Entwickler aufgrund der fehlenden Unterstützung der deskriptiven Suche die wichtigen Systemklassenbibliotheken zunächst sehr genau kennenlernen muß. Im folgenden soll unter dem Begriff *Class Browser* immer die gleichzeitige Bereitstellung deskriptiver und navigierender Suchfunktionalität verstanden werden.

In diesem Bericht beschreiben wir die Grundlagen und die prototypische Realisierung des *ReUse Class Browsers*, der für die Programmiersprache C++ deskriptive und navigierende Suchmethoden integriert. Er ist im Rahmen des größeren Projektes *SoDoM* [CGGW92] am Wissenschaftlichen Zentrum der IBM entstanden. Im Abschnitt 2 und 3 werden jeweils die Grundlagen für die deskriptive und navigierende Suche dargestellt und anhand von Beispielen erläutert. Im Abschnitt 4 wird dann auf die Architektur und die Besonderheiten des erstellten Prototypen eingegangen. Im letzten Abschnitt geben wir eine kurze Zusammenfassung und einen Ausblick auf zukünftige Aktivitäten.

2 Deskriptive Suche

Angenommen, aus den der Implementierung vorgeschalteten Phasen liegt die Spezifikation einer benötigten Klasse vor. Das Ziel der deskriptiven Suche ist das Finden von Klassen aus einer Klassenbibliothek, die diese Spezifikation möglichst vollständig erfüllen. Dazu muß die Spezifikation in eine Anfrage an den Class Browser umgesetzt werden. Dabei muß der Suchende nicht wissen (und kann es oft auch gar nicht), welche Klassen sich in der Bibliothek befinden.

Die Suche in einer Bibliothek setzt die Klassifikation ihres Inhalts voraus. Gängige Verfahren sind z.B. die *hierarchische Indexstruktur* und *Schlüsselwortmengen*. Ersteres ist mit der Querverweisproblematik belastet [PD91] und letzteres entspricht dem einfachen Spezialfall einer einzelnen Facette aus dem *Facettenansatz*. Wir verwenden in unserem Prototyp den allgemeinen Facettenansatz, den wir in den folgenden Abschnitten beschreiben werden.

2.1 Facettenklassifikation

Bei der Facettenklassifikation werden als erstes die sogenannten *Facetten* festgelegt. Diese können als disjunkte Aspekte oder auch Dimensionen des Problembereichs (bzw. der Klassenbibliothek) aufgefaßt werden. Jeder Facette ist eine *Termliste* zugeordnet, in der die einzelnen *Terme* als mögliche Beschreibungswerte aufgeführt werden. Diese werden hierarchisch strukturiert, um Ähnlichkeitsbeziehungen zwischen den Termen innerhalb einer Facette auszudrücken. [PD91, Beu89]

Danach werden *Bereiche* (“domains”) gebildet, die jeweils eine Gruppe von Facetten umfassen. Die einem Bereich zugeordneten Klassen werden über diese Facetten beschrieben. Der Vorteil der Bereichsbildung liegt in einer exakteren Facettierung und Begriffsbildung.

Bei der Klassifikation, also der Aufnahme einer Klasse in die Bibliothek, wird zunächst ein passender Bereich ausgewählt und anschließend für jede Facette dieses Bereichs der am besten passende Term selektiert. Sollte kein geeigneter Term vordefiniert sein, so kann die Termliste um einen neuen Term erweitert werden. (Puristen werden danach alle bereits klassifizierten Klassen daraufhin untersuchen, ob der neue Term besser paßt als ihr bisheriger.) Während Erweiterungen dieser Art vergleichsweise wenig Aufwand erfordern, ist dagegen das Anlegen einer neuen Facette sehr aufwendig, weil im Hinblick auf gute Suchergebnisse für alle Klassen der Bibliothek diese neue Facette mit einem Term belegt werden sollte.

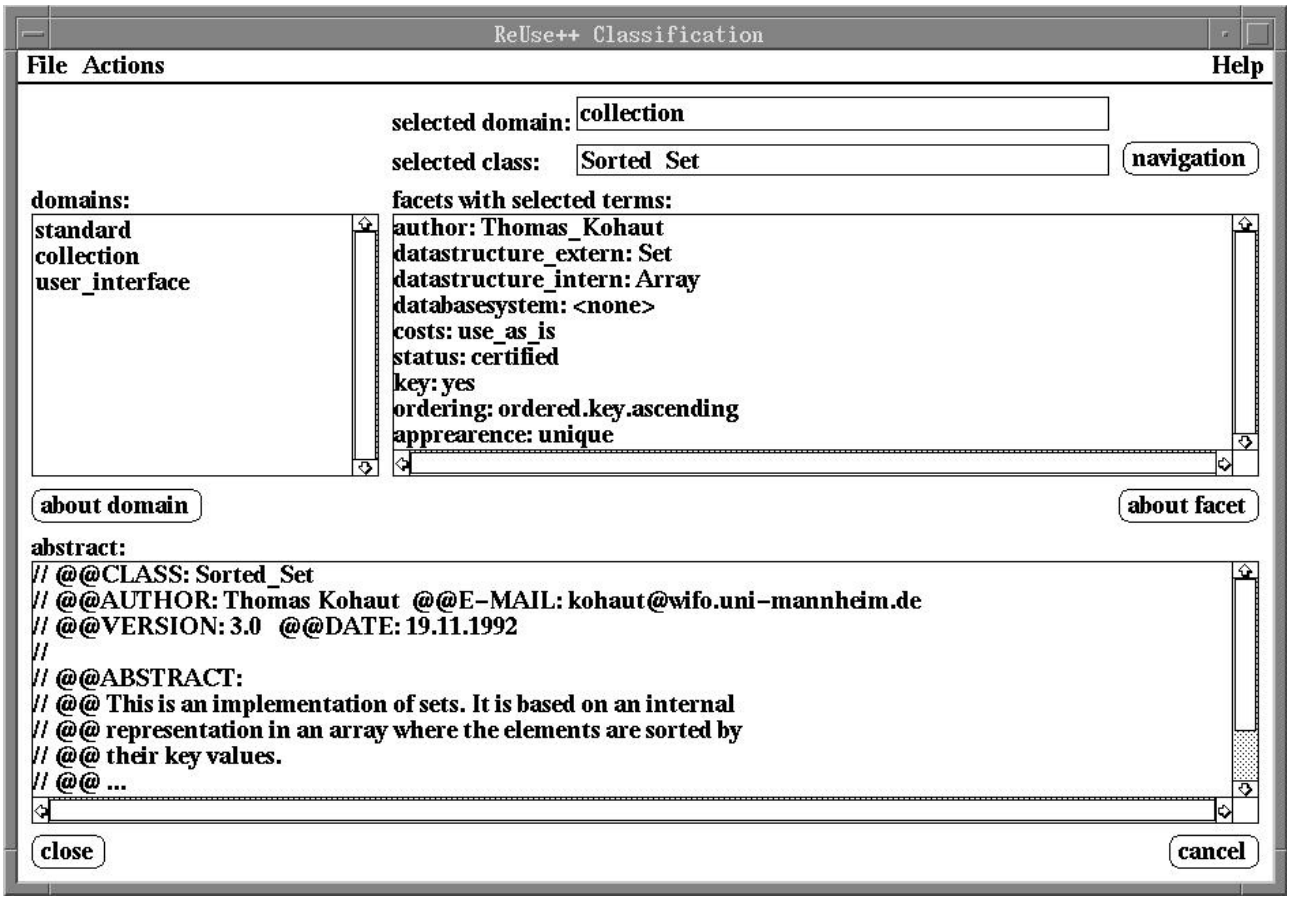

Abbildung 1: Beispielklassifikation

Ein Beispiel für eine Klassifikation ist in Abbildung 1 dargestellt. Dort wird eine Klasse für einen abstrakten Datentyp `Sorted Set` dem Bereich `collection` zugeordnet, wobei die zum Bereich gehörenden Facetten mit den ausgewählten Termen angezeigt werden. So wird unter der Facette `datastructure_extern` z.B. angegeben, daß als externe Sicht der Klasse der Datentyp `Set` geliefert wird, während der Facettenwert zu `datastructure_intern` besagt, daß die Implementierung mittels `Arrays` erfolgt. Eine Besonderheit ist der hierarchische Term `ordered.key.ascending` zur Facette `ordering`. Er besagt, daß die Elemente der Menge zunächst einmal intern sortiert gehalten werden. Als Spezialisierung wird dann auf der nächsten Hierarchiestufe angegeben, daß sich die Sortierung auf die Schlüsselwerte bezieht, wobei auf einer weiteren Hierarchiestufe auch noch die Art der Sortierung angegeben wird. Neben der Facettenklassifikation wird in unserem Prototypen in einem weiteren Teilfenster auch noch eine kurze umgangssprachliche Beschreibung der Klasse angezeigt.

2.2 Suchanfragen

Analog zur Klassifikation, die einen sogenannten *Istdeskriptor* liefert, wird bei der Suche der *Solldeskriptor* definiert, indem für relevante Facetten ein Term aus deren Termliste ausgewählt wird (siehe

Abbildung 2). Anschließend werden die Istdeskriptoren der Klassen in der Bibliothek mit dem Solldeskriptor verglichen, um deren Ähnlichkeit zu ermitteln.

Abbildung 2: Anfragebeispiel

2.3 Ähnlichkeit

Mit Hilfe einer Ähnlichkeitsfunktion wird die Ähnlichkeit der Deskriptoren quantifiziert [Beu89]. Dazu bestimmen wir zunächst die Ähnlichkeit zwischen einzelnen Termen (siehe Funktion SF), um diese dann auf ganze Deskriptoren zu erweitern (siehe Funktion S). Die Ähnlichkeitsberechnung basiert auf der hierarchischen Struktur der zur Beschreibung benutzten Terme, die als Merkmalsmengen aufgefaßt werden. Die einzelnen Hierarchiestufen sind die Elemente dieser Menge. Ein Term $a.b.c$ hat somit die Merkmalsmenge $\{a, a.b, a.b.c\}$. Im folgenden verwenden wir für einen Term und seine Merkmalsmenge denselben Großbuchstaben, wobei die jeweilige Bedeutung aus dem Kontext ersichtlich ist.

Die Facettenähnlichkeit SF eines Sollterms A zu einem Istterm B, ergibt sich aus der Kardinalität der relativen Überlappung der Merkmalsmengen von A und B, normiert mit der Kardinalität von A:

$$SF(A, B) := \frac{|A \cap B|}{|A|}$$

Man beachte, daß die so definierte Ähnlichkeit nicht symmetrisch ist, da sie sich am Sollterm orientiert. Wird z.B. nach einem Sollterm $A := a.b.c.d$ gesucht, und existieren die Istterme $B := a.b$ und $C := a.b.c.e$, so liegen die folgenden Merkmalsmengen vor:

$$A = \{a, a.b, a.b.c, a.b.c.d\}$$
$$B = \{a, a.b\}$$
$$C = \{a, a.b, a.b.c, a.b.c.e\}$$

Für die Ähnlichkeiten berechnet man $SF(A,B) = \frac{2}{4}$ und $SF(A,C) = \frac{3}{4}$. Nimmt man statt dessen B als Sollterm, so erhält man $SF(B,A) = \frac{2}{2}$.

Zur Berechnung der Ähnlichkeit von Klassen wird ein als geeignet erachteter Bereich ausgewählt und über dessen Facetten (oder eine Teilmenge davon) die gewünschte Klasse beschrieben. Der Solldeskriptor wird somit folgendermaßen definiert:

$$D_{Soll} := (M_1, M_2, \ldots, M_n)$$

M_1 bis M_n sind die den gewählten Termen zugeordneten Merkmalsmengen. Der Istdeskriptor einer Klasse ergibt sich parallel dazu aus den Merkmalsmengen der Istterme der Facetten des Bereichs der Klasse:

$$D_{Ist} := (M'_1, M'_2, \ldots, M'_m)$$

Die Ähnlichkeit S der beiden Deskriptoren wird über die Facettenähnlichkeit ihrer gemeinsamen Facetten von Soll- und Istdeskriptor berechnet. Nicht gemeinsame Facetten der Deskriptoren werden nicht berücksichtigt. (Fehlende Informationen werden also zugunsten der Ähnlichkeit ausgelegt.) Das Ergebnis der Klassenähnlichkeit wird auf das Intervall $[0,1]$ normiert. Falls die Anzahl der gemeinsamen Facetten 0 ist, ist die Klassenähnlichkeit ebenfalls als 0 definiert, ansonsten gilt:

$$S(D_{Soll}, D_{Ist}) := \frac{\sum_{i=1}^{n} \lambda(i)\ SF(M_i, M'_{f(i)})}{\sum_{i=1}^{n} \lambda(i)}, \text{ wobei}$$

$$\lambda(i) := \begin{cases} 1 & : \quad i \text{ ist gemeinsame Facette bzgl. } D_{Soll} \text{ und } D_{Ist} \\ 0 & : \quad \text{sonst} \end{cases}$$

und die Funktion f die Zuordnung der gemeinsamen Facetten übernimmt.

Entsprechend ihrer Bedeutung können die einzelnen Facetten zusätzlich noch gewichtet werden.

Das Ergebnis der Suche ist die nach Ähnlichkeit geordnete Kandidatenliste (siehe Abbildung 3). Wie viele Klassen diese Liste enthält, ist davon abhängig, wie hoch der Schwellwert bzw. die maximale Anzahl angesetzt ist. Wenn zu viele Klassen in der Liste sind, wird im nächsten Schritt eine Reduktion durchgeführt.

2.4 Reduktion

Die Reduktion ist eine einfache und zugleich effiziente Methode zur Reduzierung der durch die deskriptive Suche gefundenen Klassen. Alle Terme der Klassifikationen der durch die deskriptive Suche

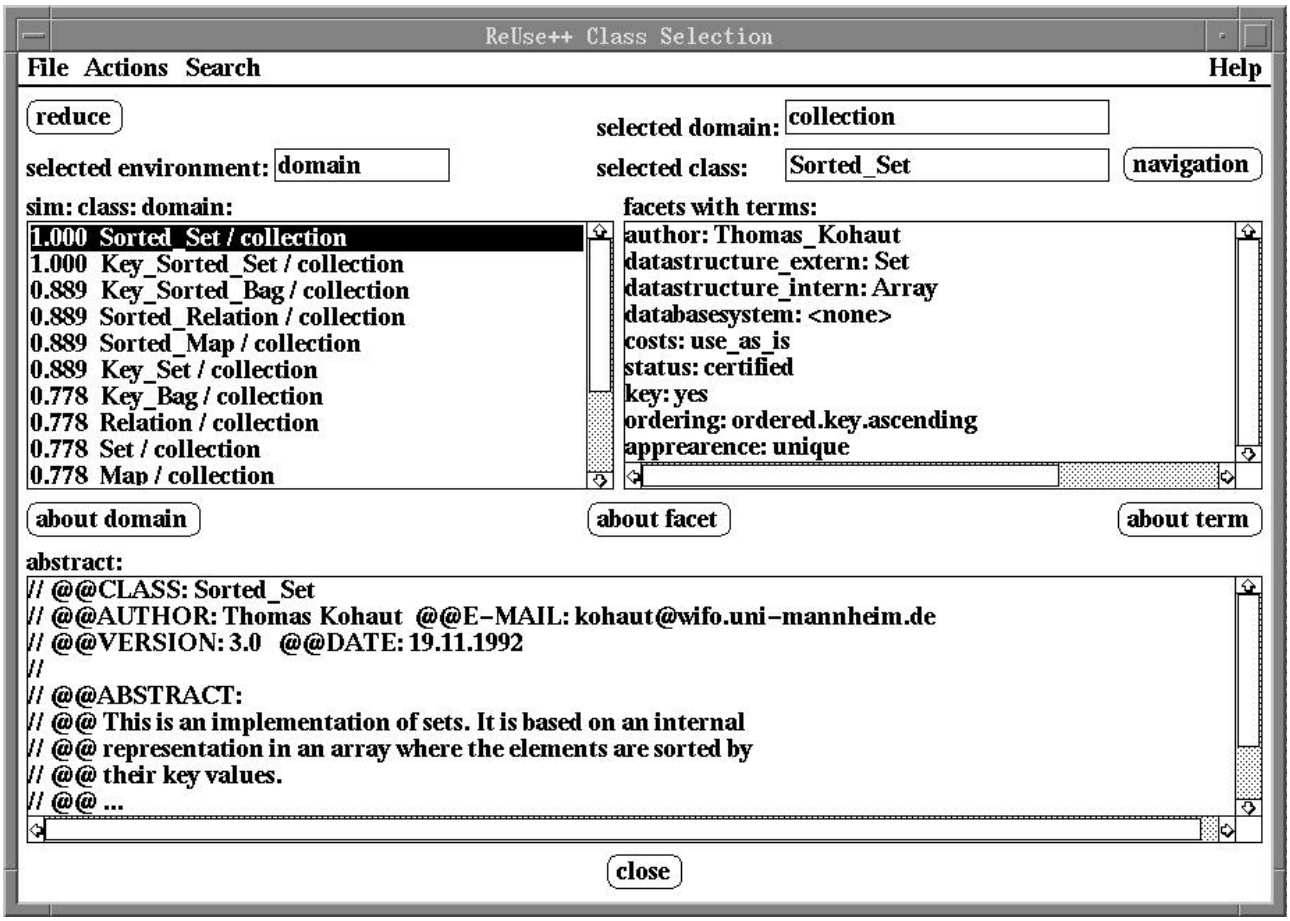

Abbildung 3: Ergebnis des Anfragebeispiels

gefundenen Klassen, also nicht nur die des Solldeskriptors, werden dazu in einer einzigen Liste zusammengefaßt und nach Facetten geordnet präsentiert. Aus dieser wird dann eine Negativauswahl der ungeeigneten Terme getroffen und die Kandidatenliste durch diese sukzessive reduziert. (Siehe Abbildung 4.)

3 Navigierende Suche

Ziel der navigierenden Suche ist das Finden einer Klasse, die die Spezifikation der gewünschten Klasse möglichst gut erfüllt bzw. die eine gute Basis für eine davon abzuleitende Klasse darstellt. Zu diesem Zweck soll die Erforschung des Klassenumfeldes ermöglicht werden, um so die Klassen besser verstehen und ihre Wiederverwendbarkeit abschätzen zu können.

Das Navigieren durch eine Menge von Klassen ist vergleichbar mit dem Wandern durch einen Graphen, bei dem die Knoten den Klassen entsprechen und die Kanten die Beziehungen zwischen den Klassen darstellen. Als Beziehungen kommen dabei neben der Vererbung auch Methodenaufrufe, Zeigerverbindungen u.ä. in Frage. Die Darstellung erfolgt graphisch in einem Fenster.

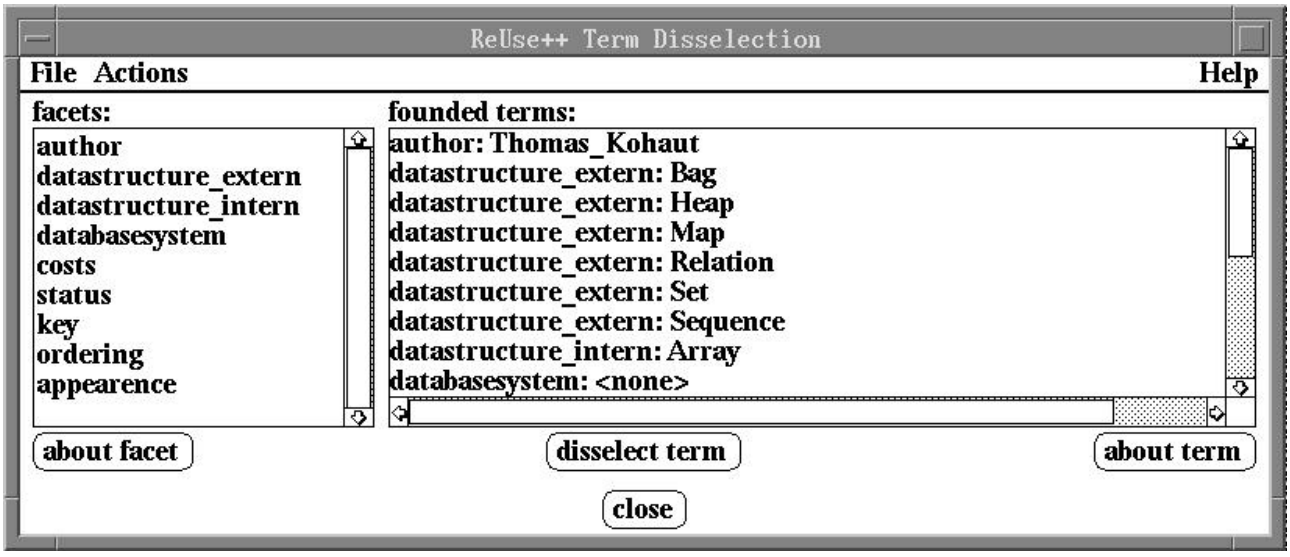

Abbildung 4: Reduktionsbeispiel

In diesem Fenster setzt der Benutzer den *Fokus* – eine Markierung, die anzeigt welche Klasse bzw. welcher Ausschnitt gerade untersucht wird – und erhält daraufhin in anderen Fenstern weitergehende Informationen hierzu.

Zahlreiche Filter ermöglichen das Einrichten benutzerdefinierter Sichten ("views"), damit nur die relevanten Informationen angezeigt werden. Zum Beispiel sind meist nur die `public` Elemente von Interesse; erst beim „Beerben" benötigt man auch die `protected` Elemente. Die `private` Implementierungsdetails sollten dagegen nur in Ausnahmefällen eingesehen werden.

Neben den bereits aufgeführten Funktionen sollte ein Browser auch über die folgenden verfügen [Mey93]:

- Adäquate graphische Darstellung der Vererbungshierarchie inklusive multipler Vererbung
- Unterschiedliche Darstellung der verschiedenen Vererbungsarten (`public`, `private`, `protected`)
- Zoom- und Scroll-Möglichkeiten
- Anzeige aller Beziehungen (nicht nur der Vererbungsbeziehungen)
- Aufführen aller Elementfunktionen und Datenelemente inklusive der geerbten

Der Browser stützt sich dabei auf Informationen, die vom Compiler generiert werden. Abbildung 5 zeigt den *AIX XL C++ Source Code Browser* [IBM92b], den wir mit unserem Prototypen gekoppelt haben, in voller Aktion.

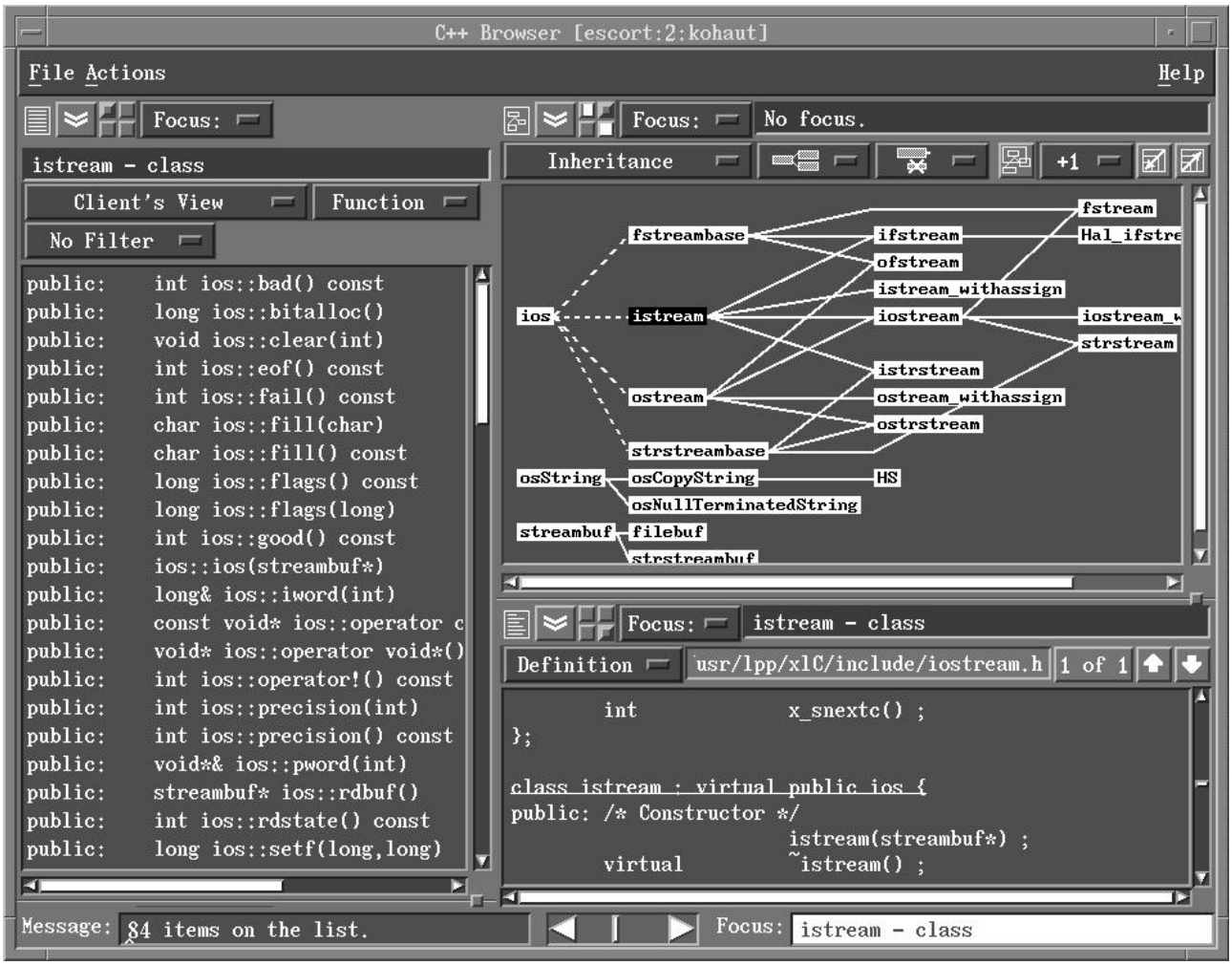

Abbildung 5: Navigationsbeispiel

4 Integration der deskriptiven und der navigierenden Suche

Die navigierende Suche in einer umfangreichen Klassenbibliothek gleicht der sprichwörtlichen Suche nach der Nadel im Heuhaufen. Erst die Vorselektion durch die deskriptive Suche grenzt die möglichen Kandidaten so ein, daß die navigierende Suche auch in kurzer Zeit zu guten Ergebnissen führt. Dabei unterstützt die navigierende Suche das Finden und Verstehen der am besten geeigneten Klasse. Bisherige Browser legen das Hauptgewicht auf die navigierende Suche und vernachlässigen die deskriptive Suche i.d.R. völlig. Dies ist dadurch zu erklären, daß sie bisher hauptsächlich die Fehlersuche in bekannten Klassen und nicht deren Wiederverwendung unterstützen sollten.

In unserem Prototypen haben wir die deskriptive Suche, wie oben beschrieben, selbst implementiert. Für die Kandidaten wird anschließend der vorhandene Source Code Browser [IBM92b] aufgerufen, der die navigierende Suche übernimmt. Die Kopplung ist noch verhältnismäßig lose.

Im deskriptiven Teil besteht sowohl die Möglichkeit zur Klassifizierung von neuen Klassen als auch

zur Suche nach vorhandenen Klassen. Diese kann auf einen Bereich beschränkt werden oder sich auf die ganze Klassenbibliothek beziehen. Die sich ergebende Kandidatenliste (vgl. Abbildung 3) kann anschließend sukzessive durch die in 2.4 beschriebene Reduktion eingeschränkt werden.

Zur Untersuchung der einzelnen Klassen werden diese aus der Kandidatenliste heraus ausgewählt. Zunächst werden die vollständige Klassifikation und die Kurzbeschreibung der ausgewählten Klasse angezeigt. Soll sie daraufhin näher untersucht werden, wird zur navigierenden Suche der Source Code Browser für sie und ihr Vererbungsumfeld aufgerufen. Dabei werden standardmäßig drei Aspekte der fokussierten Klasse angezeigt: die benutzerorientierte Sicht (`public` Elemente), die Vererbungsstruktur und die Klassendefinition (ggf. mit Kommentaren). Anschließend können die Klassen mit der vollen Mächtigkeit des Source Code Browsers untersucht werden.

Um dem Wiederverwender das Verständnis der Klassen und ihrer Methoden zu erleichtern, wurde eine einheitliche Dokumentation für die Klassen der Wiederverwendungsbibliothek eingeführt. Diese enthält auf Klassenebene u.a. Informationen über Copyright, Restriktionen, vorhandene Portierungen, Template-Parameter und eine Kurzbeschreibung, die das Verständnis der Klassen wesentlich unterstützt. Die Dokumentation der Methoden umfaßt die Vor- und Nachbedingungen und eine ausführliche Beschreibung der Parameter.

Die Realisierung unseres *ReUse Class Browsers* erfolgte unter C++ und dem Betriebssystem AIX. Zur Programmierung der Benutzungsoberfläche unter dem *X Window System* wurde die Klassenbibliothek *InterViews 3.01* benutzt. Die Datenbankfunktionen zur Ablage aller Klassifikationsdaten wurden im Hauptspeicher mittels eines eigenen „Daten-Servers“ realisiert. Die Integration des Source Code Browsers erfolgte über den „Message-Server“ der *AIX SDE WorkBench/6000* [IBM92a]. Dieser stellt eine Reihe von Kommunikationsmöglichkeiten zur Verfügung, die eine nachrichtengesteuerte Interaktion mit dem Source Code Browser erlauben.

5 Ausblick

In diesem Bericht haben wir die wesentlichen Grundlagen für eine Integration von deskriptiver und navigierender Suche in Class Browsern objektorientierter Programmiersprachen dargestellt. Weiterhin zeigt unser *ReUse Class Browser*, wie dieser Ansatz für die Programmiersprache C++ prototypisch implementiert werden kann. Die ersten Erfahrungen mit dem Prototypen sind sehr zufriedenstellend, wozu insbesondere auch die Reduktion beigetragen hat. Die Erfahrungen geben aber auch Anlaß, um über zukünftige Erweiterungen nachzudenken.

Zunächst haben wir aufgrund der Erweiterung eines kommerziell verfügbaren Class Browsers um eine deskriptive Klassifikations- und Suchkomponente nur eine sehr lose Kopplung realisiert. Ein “common look and feel” der beiden Bereiche ist nur in Ansätzen erreicht. Hier scheint eine von vornherein geplante Integration erfolgversprechender.

Der Class Browser unterstützt in seiner jetzigen Form nur die Wiederverwendung von C++-Klassen. Allerdings kann er leicht auf andere objektorientierte Sprachen übertragen werden. Wie-

derverwendung macht aber besonders dann Sinn, wenn man schon in früheren Entwicklungsphasen damit beginnt und auch schon Analyse- und Designergebnisse wiederverwenden kann. Eine solche Verallgemeinerung des hier dargestellten Ansatzes wird im SoDoM-Projekt [CGGW92] verfolgt.

Sicherlich ist auch eine Integration unseres Werkzeuges in eine allgemeine Software-Entwicklungsumgebung wünschenswert (vgl. auch [PD91]). Dazu muß der Prototyp um eine von außen benutzbare Schnittstelle (etwa über Nachrichtenaustausch) erweitert werden.

Nicht zuletzt werden wir die Funktionalität unseres *ReUse Class Browsers* mit einer wirklich großen Klassenbibliothek erproben und überprüfen, um so unseren Ansatz und seine Unterstützungsmöglichkeiten für die Wiederverwendung von Klassen fundiert bewerten zu können.

Literatur

[Beu89] Klaus Beutler. Auswertung von quantitativen Ähnlichkeitsmaßen bei der Suche nach wiederverwendbarer Software. In Wolf R. Haas, Herausgeber, *Softwaretechnik in Automatisierung und Kommunikation: Wiederverwendbarkeit von Software*, Seiten 173–184. vde-Verlag, Berlin, Offenbach, November 1989.

[BP89a] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability – Concepts and Models*, volume I. ACM Press, 1989.

[BP89b] Ted J. Biggerstaff and Alan J. Perlis, editors. *Software Reusability – Applications and Experience*, volume II. ACM Press, 1989.

[CGGW92] Bernhard Convent, Rainer Gimnich, Jürgen Günauer, and Wolfgang Wernecke. Software documents management for reuse. Technical Report 75.92.25, IBM Germany Scientific Center Heidelberg, December 1992.

[End88] Albert Endres. Software-Wiederverwendung: Ziele, Wege und Erfahrungen. *Informatik Spektrum*, 11(2):85–95, 1988.

[GTC+90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.

[IBM92a] IBM, International Business Machines Corporation. *AIX SDE WorkBench/6000, User's Guide and Reference*, 1992. Publication No. SC09-1453-01.

[IBM92b] IBM, International Business Machines Corporation. *AIX XL C++ Compiler/6000, Source Code Browser User's Guide*, 1992. Publication No. SC09-1538-00.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–183, June 1992.

[McI68] M.D. McIlroy. Mass-produced software components. In *Software Engeneering*, pages 138–150. NATO Science Committee, 1968.

[Mey93] Scott Meyers. Using C++ effectively – examining development tools. *C++ Report*, 5(1):30–35, January 1993.

[PD91] Rubén Prieto-Díaz. Implementing faceted classification for software reuse. *Communications of the ACM*, 34(5):89–97, May 1991.