Studienarbeit

Ein Browser für die objektorientierte Programmierprache Eiffel 3

vorgelegt von

Jan Willamowius
Rückertstr. 27
22089 Hamburg
jan@janhh.shnet.org
Matrikel-Nr. 4175391

November 1995

Universität Hamburg

Fachbereich Informatik

Arbeitsbereich Softwaretechnik

Danksagung:

Ich bedanke mich bei allen, die mich unterstützt haben und durch ihre Kritik und Anregungen zum Gelingen dieser Arbei beigetragen haben, insbesondere:
Walter Bischofberger für die zur Verfügung gestellte Dokumentation,
Neil Wilson für die zur Verfügung gestellten Testprogramme,
Wolfgang Strunk für konstruktive Kritik und viele Anregungen.

Inhaltsverzeichnis

1. Einleitung 1

2. Die Terminologie von Eiffel 3
2.1. Sprachkonzepte 3
2.2. Strukturierungskonzepte 4

3. Umgang mit Eiffel 7
3.1. Eine Beispiel-Situation 8
3.1.1. Szenario 1 : Bewältigen der Aufgabe mit traditionellen Eiffel-Werkzeugen 8
3.1.2. Szenario 2: Bewältigen der Aufgabe mit ISE Eiffel 3 9
3.1.3. Bewertung der Szenarien 10
3.1.4. Vision: Bewältigen der Aufgabe mit einem neuen Eiffel-Browser 11

4. Einsatzarten von Browsern 13
4.1. Top-Down Browsing 13
4.2. Bottom-Up Browsing 14

5. Existierende Browser 15
5.1. Traditionelle Werkzeuge für Eiffel 15
5.2. Die Entwicklungsumgebung von ISE Eiffel 3 18
5.3. Browser für andere objektorientierte Sprachen 21
5.3.1. Der Browser von Visual C++ 21
5.3.2. Sniff+ 23
5.4. Funktionalität von Browsern 27
5.5. Anforderungen an einen Eiffel-Browser 28

6. Möglichkeiten zur Erstellung der Datenbasis für den Browser 29
6.1. Aus dem Laufzeitsystem 29
6.2. Aus der Datenbasis des Compilers 29
6.3. Aus dem Quelltext 30
6.3.1. Kurzeinführung Compilerbau 30
6.3.2. Traditionelle Parser 33
6.3.3. Fuzzy-Parser 34

7. Implementation eines Fuzzy-Parsers 35
7.1. Architektur 36
7.2. Error-Recovery 37
7.3. Test des Fuzzy-Parsers 38

8. Technische Integration in Sniff+ 39
8.1. Die Architektur von Sniff+ 39
8.2. Abbilden der Eigenschaften von Eiffel auf C++ 41
8.3. Probleme bei der Integration 42

9. Umgang mit Eiffel in Sniff+ 45
9.1. Der Editor 45
9.2. Der Hierarchie-Browser 46
9.3. Der Symbol-Browser 47
9.4. Der Class-Browser 48
9.5. Der Projekt-Editor 49

10. Ausblick 51

11. Anhang A Literatur 53

12. Anhang B Installationsbeschreibung 57

1. Einleitung

Um produktiv arbeiten zu können, benötigt ein Programmierer eine gute Werkzeugunterstützung - am besten eine integrierte Entwicklungsumgebung mit Werkzeugen, die es ihm ermöglichen, auf einem hohen Abstraktionsniveau zu arbeiten.

Ein wesentlicher Teil so einer Entwicklungsumgebung ist der Browser. Ein Browser ist ein Werkzeug zum schnellen und überblicksartigen Betrachten von Quelltexten. Meist dient er nur zum Betrachten am Bildschirm und nicht zum Verändern. Er soll es erleichtern, eigene und fremde Quelltexte zu analysieren und zu verstehen.

Im Englischen heißt "to browse" soviel wie schmökern. Ein Werkzeug, das diese Tätigkeit unterstützen soll, sollte es dem Benutzer leichtmachen, an Informationen heranzukommen und flexibel neue Fragen zu stellen und beantwortet zu bekommen. Es sollte förmlich dazu verleiten zu schmökern.

Um die Versprechen der objektorientierten Programmierung von Wiederverwendbarkeit und Qualitätssteigerung einlösen zu können, ist es notwendig, existierende Bibliotheken zu verstehen und zu benutzen. Im objektorientierten Bereich ist hierzu eine Werkzeugunterstützung noch wichtiger als bei prozeduraler Programmierung, da der Kontrollfluß weniger linear ist.

Programmierer sind bei der Arbeit mit Eiffel bisher meist auf die zu ihrem Compiler mitgelieferten Werkzeuge beschränkt. Diese sind oft unzureichend in ihrer Funktionalität und mangelhaft integriert.

Das praktische Ergebnis dieser Arbeit ist die Anpassung der C++-Entwicklungsumgebung Sniff+, so daß man mit ihr auch Eiffel-Programme bearbeiten und alle ihre Werkzeuge auch auf Eiffel anwenden kann.

Zur Einführung in die Programmiersprache stelle ich in Kapitel 2 zunächst die Terminologie von Eiffel vor. Danach motiviere ich die Notwendigkeit eines neuen Browsers mit der Darstellung des bisherigen Umgangs mit Eiffel in Kapitel 3. Daraus abgeleitet stelle ich in Kapitel 4 die prinzipiellen Einsatzarten von Browsern dar.

In Kapitel 5 beschreibe ich einige existierende Browser, um daraus die Anforderungen für einen neuen Eiffel-Browser abzuleiten.

Die für die Implementation des Browsers sehr wichtige Frage der Erstellung seiner Datenbasis diskutiere ich in Kapitel 6. Es wird sich dort zeigen, daß die m.E. beste Möglichkeit hierzu ein Fuzzy-Parser ist.

Die Implementation des Fuzzy-Parsers wird in Kapitel 7 und dessen Integration in einen existierenden Browser in Kapitel 8 beschrieben.

Der Umgang mit Eiffel im neuen Browser, der das Ergebnis dieser Arbeit ist, wird dann in Kapitel 9 vorgestellt.

2. Die Terminologie von Eiffel

Eiffel verwendet z.T. eine etwas andere Terminologie als andere objektorientierte Sprachen. Insbesondere wird es in Kapitel 8.2 um den Unterschied zur Terminologie von C++ gehen. Daher soll hier zunächst die Terminologie von Eiffel eingeführt werden.

Als Referenz für die Terminologie von Eiffel verwende ich [Meyer 91]. Die Standardwerke zu C++ sind [Stroustrup 92] und [EllisStrou 90].

2.1. Sprachkonzepte

Ein Eiffel-Programm setzt sich aus einzelnen *Klassen* (engl. *classes*) zusammen. Die Klassen beschreiben jeweils einen Datentyp. Zur Laufzeit werden Exemplare der Klasse erzeugt, die dann *Objekte* (engl. *objects*) genannt werden.

In Eiffel werden die Klassen durch sog. *Merkmale* (engl. *features*) beschrieben. Dabei werden die Merkmale unterteilt in *Attribute* (engl. *attributes*), die den Zustand beschreiben, und *Routinen* (engl. *routines*), die die Operationen beschreiben.

Klassen können auf zwei Arten miteinander in Beziehung stehen: durch Vererbung und durch Benutzung. In den Diagrammen werden zwei unterschiedliche Pfeilarten zur Kennzeichnung dieser Beziehungen verwendet.

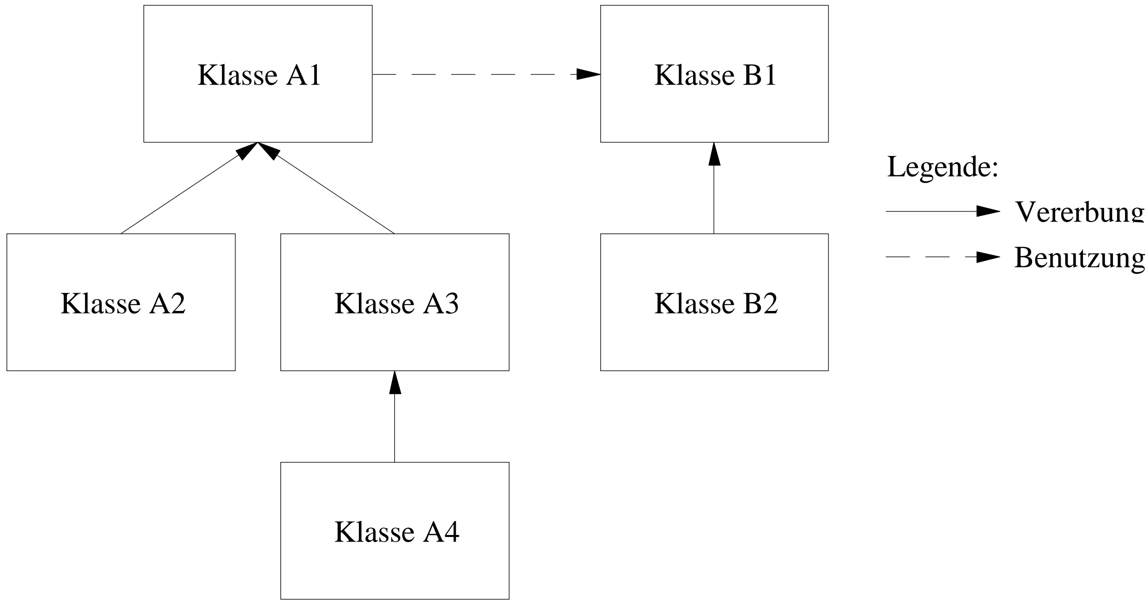


Abb. 1 Beziehungen zwischen Klassen

Im Beispiel (Abb. 1) erben also die Klassen A2 und A3 von der Klasse A1, und die Klasse B2 erbt von B1, während die Klasse A1 die Klasse B1 benutzt.

Nach der Eiffel-Terminologie ist A1 *Elternteil* (engl. *parent*) von A2 und A3, da A2 und A3 direkt von A1 erben. Andersherum betrachtet sind die Klassen A2 und A3 *Erben* (engl. *heir*) von der Klasse A1.[1]

Eine Klasse, die direkt oder indirekt von einer Klasse erbt, wird als ihr *Nachkomme* (engl. *descendant*) bezeichnet. Im Beispiel sind A2, A3 und A4 Nachkommen von A1. Die Klasse A1 wird dann als *Vorfahre* (engl. *ancestor*) dieser Klassen bezeichnet. Der Unterschied zu den vorher vorgestellten Begriffen Vorfahre und Elternteile liegt darin, daß keine direkte Vererbung vorliegen muß, um eine Klasse als Nachkomme bzw. Vorfahre zu bezeichnen. So kann z.B. auch die Beziehung von A1 und A4 miterfaßt werden.

Synonym für die Begriffe Nachkomme und Vorfahre verwende ich auch die Begriffe *Unterklasse* und *Oberklasse*.

Im Beispiel (Abb. 1) ist Klasse A1 *Kunde* (engl. *client*) von B2. Dies entspricht der Benutzungs-Beziehung (gestrichelter Pfeil). Da die Klasse B2 einen Dienst erbringt wird sie auch als *Lieferant* (engl. *supplier*) bezeichnet.

Wie andere objektorientierte Sprachen auch bietet Eiffel die Möglichkeit, Klassen oder Merkmale als *aufgeschoben* (engl. *deferred*) zu deklarieren. Dabei wird lediglich die Schnittstelle, nicht aber die Implementierung angegeben. Eine Klasse ist aufgeschoben, wenn sie als aufgeschoben deklariert ist oder wenn sie ein aufgeschobenes Merkmal besitzt. Von aufgeschobenen Klassen können keine Exemplare erzeugt werden, sehr wohl aber von Klassen, die von ihnen erben und alle aufgeschobenen Merkmale konkretisiert haben.

Da Eiffel grundsätzlich mit Referenzsemantik arbeitet, dies aber z.B. bei Basisdatentypen nicht erwünscht ist, besteht die Möglichkeit, Klassen als *expandiert* (engl. *expanded*) zu deklarieren, so daß für diese Klassen bei Zuweisungen und Vergleichen immer Wertsemantik verwendet wird (ohne daß auf die Routinen clone und equal zurückgegriffen werden muß). Diese Eigenschaft einer Klasse wird vererbt, ohne daß die Erben sie in ihrer Deklaration erwähnen müssen.

2.2. Strukturierungskonzepte

Oberhalb von Klassen gibt es noch weitere Strukturierungsmittel in Eiffel, die aber nicht mehr in Eiffel selbst ausgedrückt werden können. Sie dienen dazu, entweder fachlich oder technisch zusammengehörige Klassen zu Projekten bzw. Bibliotheken zu gruppieren. Dies erhöht die Übersichtlichkeit für den Programmierer und teilt den verwendeten Werkzeugen mit, welche Klassen bearbeitet (z.B. übersetzt) werden müssen.[2]

Die Zusammenstellung einer oder mehrerer Klassen, die ein ausführbares Programm bilden (sollen), wird in Eiffel-Terminologie als *System* bezeichnet. Als *Cluster* bezeichnet man eine Menge von zusammengehörigen Klassen. Diese können z.B. eine Bibliothek oder aber auch

---

[1] Die Klasse A1 ist nicht Elternteil von A4, da keine direkte Beziehung vorliegt.

[2] Ein Teil dieser Aufgaben würde in C++ vom Makefile übernommen werden.

nur ein einzelnes Designmuster verkörpern. Die Menge aller Cluster, die für einen speziellen Entwurf verwendet werden, wird als *Universum* (engl. *universe*) bezeichnet.

Meyer schlägt in [Meyer 91] eine Beschreibungssprache namens *LACE* (Language for Assembling Classes in Eiffel) vor, um die Strukturierung und Montage eines Systems zu beschreiben. Zur Beschreibung von Clustern wird die Verwendung von Verzeichnissen des Dateisystems vorgeschlagen, in denen die zugehörigen Klassen abgelegt werden. Die Abgrenzung des Universums ergibt sich aus den in LACE referenzierten Clustern.

Da LACE nicht Bestandteil der Sprache Eiffel ist, haben die verschiedenen Hersteller von Eiffel-Compilern z.T. sehr abweichende Implementationen vorgenommen.[3]

[3] SiG hat für Eiffel/S sogar eine ganz eigene Sprache Namens PDL (Program Description Language) entwickelt (vergl. [SiG 92]).

3. Umgang mit Eiffel

Gerade Eiffel unterstützt den Ansatz, Software aus wiederverwendbaren Komponenten zusammenzusetzen. Bertrand Meyer schreibt in einem Artikel über die Erfahrungen mit der Eiffel-Bibliothek ([Meyer 90]):

> "A common problem in the component-based approach to software development is determining how to enable client programmers to find out about available components and retrieve them easily. Obviously, the seriousness of this problem grows with the number of components."

Bei der Implementation in Eiffel hat der Programmierer das Bedürfnis, Klassen in Bibliotheken (die zum Compiler mitgelieferten oder anderen) zu finden, die er für seine Zwecke benutzen kann.

Solange er mit dem Aufbau der Bibliotheken und den Funktionen der einzelnen Klassen nicht vertraut ist (und bei dem Umfang der meisten Bibliotheken ist dies eine sehr lange Zeit, wenn nicht immer der Fall), muß er sich auf der Suche nach einer passenden Klasse die Beschreibung und das Interface vieler Klassen ansehen, bis er eine passende gefunden hat.

Der Programmierer kann natürlich alle Klassen mit einem Texteditor o.ä. ansehen (Eiffel-Bibliotheken werden üblicherweise mit Quelltext ausgeliefert), bis er fündig wird. Die Beziehungen der Klassen untereinander müssen aber von ihm selbst (geistig) hergestellt werden. Hier ist eine Werkzeugunterstützung unbedingt nötig, sobald mit größeren Projekten gearbeitet wird.

Das gleiche Problem, die Struktur und den Aufbau der Bibliotheken zu verstehen, taucht auch bei der Einarbeitung in Programme anderer Programmierer auf. Konnte man bei Bibliotheken dem Problem z.T. noch mit einer guten Dokumentation beikommen, so ist hier jedoch oft keinerlei Dokumentation vorhanden und der Autor für Fragen oft nicht zu erreichen. Eine gute Werkzeugunterstützung ist hier fast lebenswichtig.

Einige Beispiele für Fragen eines Programmierers:

- Wo finde ich eine Behälterklasse für ca. 50 Objekte, auf die ich wahlfrei und schnell zugreifen kann ?
- Gibt es eine Behälterklasse, die ca. 20.000 "Dauerauftrag"-Objekte platzsparend speichert ?
- Welche Interaktionstypen für eine 1-aus-n Selektion gibt es ?
- Gibt es in unserem Projekt noch Klassen, die von der Klasse SPEICHERBAR erben ?
- Ich möchte Merkmalsnamen einheitlich vergeben. Gibt es schon andere Klassen, die den Merkmalsnamen "Index" verwenden ?

Dies sind nur einige Beispiele für Arbeitssituationen, in denen ein Programmierer eine Werkzeugunterstützug zum Betrachten von Quelltexten braucht, die ihm auch hilft die Zusammenarbeit der einzelnen Komponenten zu verstehen. Um zu zeigen, wie bisher mit diesen Fragen umgegangen wird, folgen zwei Szenarien, die die Arbeit mit den bisherigen Hilfsmitteln exemplarisch verdeutlichen, und eine Vision, wie es sein könnte.

3.1. Eine Beispiel-Situation

Der Programmier soll die Klasse UEBERWEISUNGSAUFTRAG anpassen, damit sie in einer Behälterklasse der neuen ConlibEiffel-Bibliothek gespeichert werden kann.

3.1.1. Szenario 1 : Bewältigen der Aufgabe mit traditionellen Eiffel-Werkzeugen

Da der Programmierer die Bibliothek ConlibEiffel noch nicht kennt und sich die beigefügte Dokumentation nur auf die C++ Version bezieht, sucht er nach einem Beispiel, das die Verwendung der Bibliothek demonstriert.

Im Verzeichnis /usr/src/conlib findet er die Datei testlib.e. Der Programmierer schaut sich das Beispiel mit dem Unix-Befehl "`short testlib`" an. Er kann an der Ausgabe erkennen, daß die Klasse Testlib heißt und eine Creation-Feature make und drei andere parameterlose Routinen InitTest, InsertObjs und ListObjs hat.

Da die Klasse von keiner anderen erbt, schaut er sich nun mit dem Unix-Befehl "`more`" die Datei an. Im Beispiel werden INTEGER-Objekte in eine Liste (SORTED_LIST) eingefügt und danach der Reihe nach wieder ausgegeben. Ein Testlauf des Beispiels zeigt, daß es einwandfrei funktioniert.

Da offenbar keine Modifikationen an der Klasse INTEGER nötig waren, ändert er das Beispiel, so daß nun Überweisungsaufträge in die Liste eingefügt werden. Er ist ein wenig verwundert, daß er keine Möglichkeit sieht, anzugeben, wie die Überweisungsaufträge sortiert werden sollen.

Ein Blick auf die Ausgabe von "`short SORTED_LIST`" zeigt, daß SORTED_LIST das Merkmal AddAt redefiniert und sonst keine Änderungen an der geerbten Klasse LIST vornimmt. Da dieses Feature im Beispiel aber nicht verwendet wird, kann der Programmierer mit dieser Information nichts weiter anfangen. Da die Ausgabe von "`flat SORTED_LIST | short`" ca. 15 Bildschirmseiten lang ist, entscheidet sich der Programmierer für einen Testlauf des eben modifizierten Testprogramms.

Vom Compiler erhält er die Fehlermeldung, daß das Einfügen von UEBERWEISUNGS-AUFTRAG mit Insert nicht typverträglich ist. Der Programmierer schaut sich mit "`flat SORTED_LIST | short | more`" nun erneut die volle Klassenschnittstelle an und sucht nach dem Insert Feature. Nachdem er es gefunden hat, stellt er fest, daß nur Objekte vom Typ COMPARABLE eingefügt werden dürfen.

Ein "`short COMPARABLE`" zeigt ihm, daß die Klasse aufgeschoben (deferred) ist. Aus den Kommentaren ist ersichtlich, daß mindestens der Operator < von Unterklassen implementiert werden muß.

Mit diesem Wissen ausgerüstet, modifiziert er seine Klasse UEBERWEISUNGSAUFTRAG derart, daß sie nun zusätzlich von COMPARABLE erbt und den Operator < implementiert (abhängig von der Kontonummer des Absenders - oder der laufenden Nummer des Überweisungsauftrags).

Das Testbeispiel funktioniert nun auch mit Überweisungsaufträgen, und der Programmierer gibt die modifizierte Klasse für die restlichen Entwickler frei.

3.1.2. Szenario 2: Bewältigen der Aufgabe mit ISE Eiffel 3[4]

Da der Programmierer die Bibliothek ConlibEiffel noch nicht kennt und sich die beigefügte Dokumentation nur auf die C++ Version bezieht, sucht er nach einem Beispiel, das die Verwendung der Bibliothek demonstriert.

Im Verzeichnis /usr/src/conlib findet er die Datei testlib.e, die er in das Class Tool lädt. Im Beispiel werden INTEGER-Objekte in eine Liste (SORTED_LIST) eingefügt und danach der Reihe nach wieder ausgegeben. Ein Testlauf des Beispiels zeigt, daß es einwandfrei funktioniert.

Da offenbar keine Modifikationen an der Klasse INTEGER nötig waren, ändert er das Beispiel, so daß nun Überweisungsaufträge in die Liste eingefügt werden. Er ist ein wenig verwundert, daß er keine Möglichkeit sieht, anzugeben, wie die Überweisungsaufträge sortiert werden sollen.

Er stellt den Darstellungsmodus auf "clickable" um, klickt mit der rechten Maustaste auf den Klassennamen SORTED_LIST und zieht ihn auf das Class Tool-Icon links oben im Fenster. Er sieht nun den Klassentext der Klasse SORTED_LIST. Da er an den Details der Implementierung nicht interessiert ist, schaltet er mit einem Klick auf das Short-Icon am unteren Fensterrand auf die Short-Darstellung um. Er sieht, daß das Feature AddAt redefiniert wird (das im Beispiel aber nicht verwendet wird) und sonst keine Änderungen an der geerbten Klasse LIST vorgenommen werden.

Mit einem Klick auf das Flatshort-Icon am unteren Fensterrand schaltet er auf die Flatshort-Form um (was ihm Gelegenheit gibt, seinen Kaffee auszutrinken). Danach hat er eine ca. 15 Bildschirmseiten lange Klassenschnittstelle im Editor. Da er nicht genau weiß, wonach er sucht, entscheidet sich der Programmierer für einen Testlauf des eben modifizierten Testprogramms.

Vom Compiler erhält er die Fehlermeldung, daß das Einfügen von UEBERWEISUNGS-AUFTRAG mit Insert nicht typverträglich ist. Der Programmierer sucht nun nach dem Insert Feature. Nachdem er es gefunden hat, stellt er fest, daß nur Objekte vom Typ COMPARABLE eingefügt werden dürfen.

Er stellt den Darstellungsmodus auf "clickable" um, klickt mit der rechten Maustaste auf den Klassennamen COMPARABLE und zieht ihn auf das Class Tool-Icon links oben im Fenster. Im Class Tool hat er nun die Klasse COMPARABLE in Flatshort-Form vor sich. Nachdem er die Darstellung auf Short-Form umgeschaltet hat, sieht er, daß die Klasse aufgeschoben (deferred) ist. Aus dem Kommentaren ist ersichtlich, daß mindestens der Operator < von Unterklassen implementiert werden muß.

---

[4] Eine detailiertere Beschreibung der Entwicklungsumgebung von ISE Eiffel 3 findet sich in Kapitel 5.2.

Mit diesem Wissen ausgerüstet, modifiziert er seine Klasse UEBERWEISUNGSAUFTRAG derart, daß sie nun zusätzlich von COMPARABLE erbt und den Operator < implementiert (abhängig von der Kontonummer des Absenders - oder der laufenden Nummer des Überweisungsauftrags).

Das Testbeispiel funktioniert nun auch mit Überweisungsaufträgen.

Mit einem Klick auf das Ancestors-Icon am unteren Bildschirmrand sieht er sich die neue Vererbungshierarchie (Oberklassen) an. Ihm fällt auf, daß die Klasse UEBERWEISUNGS-AUFTRAG noch von SPEICHERBAR erbt. Nach dem Durchsehen des Klassentextes stellt er fest, daß dies nur für die alte Behälterbibliothek nötig war, und entfernt SPEICHERBAR aus der Liste der Oberklassen.

Er compiliert die Klasse UEBERWEISUNGSAUFTRAG erneut, und der Programmierer gibt die modifizierte Klasse für die restlichen Entwickler frei.

3.1.3. Bewertung der Szenarien

Das zweite Szenario macht ansatzweise deutlich, wie ein Browser dem Programmierer helfen kann, seine Probleme zu lösen. Insoweit ist der Einsatz von ISE Eiffel 3 sicher ein Schritt in die richtige Richtung.

Bei einem guten Browser wird der Programmierer oft auch auf Dinge stoßen, nach denen er nicht explizit sucht, die aber für seine Arbeit doch hilfreich sind. Im Beispiel wird bei der Arbeit mit traditionellen Werkzeugen beispielsweise überhaupt nicht bemerkt, daß eine Klasse überflüssig geworden ist. Solange die Informationsbeschaffung schwierig und langwierig ist, wird sich der Programmierer auch kaum bemühen, solchen Details nachzugehen.

Es gibt dabei jedoch auch Probleme beim Einsatz von ISE Eiffel 3, die nur zum Teil aus dem Szenario zu erkennen sind. Dies beginnt mit der sehr gewöhnungsbedürftigen Oberfläche, die bei mir zu keinem Zeitpunkt das Gefühl von *direkter Manipulation* aufkommen ließ. Zum einen liegt dies an der Handhabung, die nicht dem Styleguide der jeweiligen Plattform entspricht. Zum anderen treten aber auch oft störende Wartezeiten bei der Arbeit mit größeren Informationsmengen auf, da die monolithische Umgebung kaum skalierbar ist.

Auch geht die Umsetzung der Darstellungsmethoden auf eine graphische Oberfläche nicht weit genug. War es bei den traditionellen Werkzeugen **short** und **flat** noch sinnvoll, immer die Schnittstelle der gesamten Klasse anzuzeigen, so wird man unter der graphischen Oberfläche z.T. von der Informationsmenge erschlagen und benötigt eine weitere Filterbarkeit der Informationen.

Im Szenario nicht sichtbar ist auch die explizite Beachtung, die der Programmierer bei ISE Eiffel 3 der Aktualität der angezeigten Informationen widmen muß. Viele Darstellungen sind nur dann korrekt, wenn sich der Quelltext in einer vorkompilierten Form befindet (melted). Auch werden offene Fenster nicht automatisch aktualisiert, wenn sich der Quelltext geändert hat.

3.1.4. Vision: Bewältigen der Aufgabe mit einem neuen Eiffel-Browser

Da der Programmierer die Bibliothek ConlibEiffel noch nicht kennt und sich die beigefügte Dokumentation nur auf die C++ Version bezieht, sucht er nach einem Beispiel, das die Verwendung der Bibliothek demonstriert.

Im Unterprojekt ConlibEiffel sieht er die Datei testlib.e, die er mit einem Doppelklick in den Editor lädt. Im Beispiel werden INTEGER-Objekte in eine Liste (SORTED_LIST) eingefügt und danach der Reihe nach wieder ausgegeben. Ein Testlauf des Beispiels zeigt, daß es einwandfrei funktioniert.

Da offenbar keine Modifikationen an der Klasse INTEGER nötig waren, ändert er das Beispiel, so daß nun Überweisungsaufträge in die Liste eingefügt werden. Er ist ein wenig verwundert, daß er keine Möglichkeit sieht, anzugeben, wie die Überweisungsaufträge sortiert werden sollen.

Er öffnet den Symbol-Browser und lädt SORTED_LIST mit einem Doppelklick auf den Klassennamen in den Editor. Am rechten Rand sieht er, daß nur das Feature AddAt in dieser Klasse definiert wird (das im Beispiel aber nicht verwendet wird).

Er wählt den Menüpunkt "Show SORTED_LIST in Hierarchie" aus und sieht, daß SORTED_LIST nur von LIST erbt. Ihm fällt auf, daß im gleichen Projekt andere zu speichernde Klassen von COMPARABLE erben. Mit einem Doppelklick lädt er LIST in den Editor.

In der Feature-Liste am rechten Rand des Editors sieht er die ca. 20 in dieser Klasse definierten Features. Da er nicht genau weiß, wonach er sucht, entscheidet sich der Programmierer für einen Testlauf des eben modifizierten Testprogramms.

Vom Compiler erhält er die Fehlermeldung, daß das Einfügen von UEBERWEISUNGS-AUFTRAG mit Insert nicht typverträglich ist. Im Editor klickt der Programmierer nun rechts in der Liste auf Insert, und der Editor springt auf die Implementation von Insert. Hier sieht er, daß nur Objekte vom Typ COMPARABLE eingefügt werden dürfen.

Im Symbol-Browser klickt er nun auf COMPARABLE, um es in den Editor zu laden. Dort sieht er, daß die Klasse aufgeschoben (deferred) ist. Aus dem Kommentaren ist ersichtlich, daß mindestens der Operator < von Unterklassen implementiert werden muß.

Mit diesem Wissen ausgerüstet, modifiziert er seine Klasse UEBERWEISUNGSAUFTRAG derart, daß sie nun zusätzlich von COMPARABLE erbt und den Operator < implementiert (abhängig von der Kontonummer des Absenders - oder der laufenden Nummer des Überweisungsauftrags).

Nachdem er die Änderungen im Editor gespeichert hat, ändert sich die Darstellung im Hierarchie-Browser (noch bevor er den Compiler gestartet hat). Ihm fällt auf, daß die Klasse UEBERWEISUNGSAUFTRAG noch von SPEICHERBAR erbt. Nach dem Durchsehen des Klassentextes stellt er fest, daß dies nur für die alte Behälterbibliothek nötig war, und entfernt SPEICHERBAR aus der Liste der Oberklassen.

Da SPEICHERBAR in der nächsten Woche überarbeitet werden sollte, schickt er eine EMail an den Rest des Entwicklerteams mit der Anregung, diese ohnehin recht problematische Klasse völlig aus dem Projekt zu entfernen.

Er compiliert das Testbeispiel nun. Es funktioniert nun auch mit Überweisungsaufträgen, und der Programmierer gibt die modifizierte Klasse für die restlichen Entwickler frei.

4. Einsatzarten von Browsern

Browser dienen dazu, dem Entwickler Einblick in seinen zu bearbeitenden Quelltext zu verschaffen, dessen Struktur zu visualisieren und ein strukturiertes Suchen zu ermöglichen.

Da der Browser die Struktur der verwendeten Sprache kennt, kann er viel mehr Unterstützung bieten als z.B. ein Texteditor. In einem Browser könnte man z.B. die Suche von Zeichenketten auf Klassennamen, Methodennamen o.ä. beschränken.

Nach [McConnell 93] ist dem Entwickler meist sehr genau bewußt *was* er sucht und beim Einsatz eines Browser geht es hauptsächlich darum die Information möglichst schnell aufzufinden.

Zwei unterschiedliche Arten von Quelltexten müssen vom Entwickler betrachtet werden. Der quasi statische Quelltext der verwendeten Bibliotheken und der im ständigen Wandel befindliche Quelltext der Anwendung, die gerade entwickelt wird.

Bei der Einarbeitung in Bibliotheken ist ein Browser sehr wichtig, denn die Wiederverwendung der dort schon vorhandenen Funktionalität ist nur bei ausreichendem Verständnis der Bibliothek möglich. Ein Browser sollte nach [Gamma 92] dabei das "interaktive und explorative Untersuchen" der Quelltexte in den Vordergrund stellen.

Beim Entwickeln von eigenen bzw. neuen Klassen wird (gerade in der objekt-orientierten Programmierung) Bezug genommen auf Bibliotheksklassen, die dann spezialisiert werden. Hier sind die Anforderungen an den Browser andere als bei statischem Quelltext.

Der Browser sollte auch die Reorganisation des Quelltextes unterstützen, da dies eine relativ häufige und gleichzeitig auch komplexe Aufgabe ist. Diese Forderung impliziert, daß er auch mit syntaktisch nicht korrektem Quelltext arbeiten können muß.

In [Bischofberger 94a] werden zwei Arten, Quelltexte zu browsen, unterschieden: Top-Down Browsing und Bottom-Up Browsing.

4.1. Top-Down Browsing

Um mehr über den Aufbau und die Struktur eines existierenden Softwaresystems zu lernen, wird Top-Down Browsing verwendet. Der Entwickler betrachtet die Klassen des Systems, ihre Schnittstellen und ihre Beziehungen zueinander (hauptsächlich Benutzung und Vererbung).

Alle dazu nötigen Informationen muß der Browser für ihn sammeln und visuell aufbereiten. Es ist wichtig, gute Navigationshilfen zur Verfügung zu stellen, da der Kontrollfluß meist nicht-linear ist. Bei objektorientierten Frameworks spricht man oft sogar von *invertiertem Kontrollfluß*, bei dem die eigentliche Ablaufsteuerung innerhalb des Frameworks liegt und im Quelltext der Anwendung überhaupt nicht sichtbar ist (vergl. [Gamma 95]).

Nicht nur die Struktur des Softwaresystems, sondern auch die Implementation der betrachteten Klassen sollte beim Top-Down Browsing eingesehen und evtl. verändert werden können.

Ein Beispiel für das Top-Down Browsing ist die Arbeit eines Programmierers, der zu verstehen versucht, wie eine bestimmte Klasse bzw. Bibliothek arbeitet bzw. benutzt werden kann. Dies entspricht der Grundaufgabe des Programmierers in der in Kapitel 3 behandelten Beispielsituation, wo es ihm um das grundsätzliche Verständnis der Bibliothek ging.

4.2. Bottom-Up Browsing

Beim Untersuchen eines bestimmten Teils der Implementation treten dagegen meist andere Fragestellungen auf. Hier ist z.B. wichtig zu erfahren, welchen Typ eine bestimmte Variable hat und wo dieser implementiert ist. Der Programmierer geht also von einer vorliegenden Verwendung aus und versucht auf den Kontext zu schließen.

Ein Beispiel für Bottom-Up Browsing ist die Arbeit eines Programmierers, der auf eine Variable WindowHandle stößt und sich bemüht, den Zusammenhang zu verstehen, in dem sie verwendet wird.

Oft sind diese beiden Arten des Browsing sehr eng miteinander verbunden, indem z.B. das Verständnis eines Teils einer Bibliothek durch diverse Bottom-Up Untersuchungen vertieft wird, bevor andere Teile Top-Down untersucht werden. Gerade dieser flexible Umgang mit dem zu untersuchenden Quelltext macht die Überlegenheit eines Browsers z.B. gegenüber einem Cross-Referencer aus.

5. Existierende Browser

5.1. Traditionelle Werkzeuge für Eiffel

Bertrand Meyer hat in [Meyer 88] drei Werkzeuge beschrieben, die den Umgang mit Eiffel erleichtern sollten: **short**, **flat** und **good**. Diese wurden auch z.B. mit ISE Eiffel 2.x ausgeliefert.

short gibt zu einer als Parameter angegebenen Klasse das Interface aus. Es sind nur die in dieser Klassen definierten Merkmale sichtbar und nicht die geerbten. Das Werkzeug ist "batchorientiert", d.h. eine interaktive Arbeit (abgesehen von einem erneuten Aufruf mit anderen Parametern) wird nicht unterstützt.

(Anmerkung: Ich verwende hier einen etwas anderen Werkzeugbegriff als WAM, wo Interaktivität bereits in der Definition eines Werkzeuges enthalten ist.)

Durch den Aufruf von "`short Klassenname`" erhält man die Klassenschnittstelle der angegebenen Klasse. Die Merkmale der Klasse erscheinen grundsätzlich in der Reihenfolge ihres Auftretens im Quelltext. Als zusätzliche Dokumentation erscheinen lediglich die Kommentare im Quelltext der Klasse. Diese Art der Darstellung wird als *Short-Form* bezeichnet.

flat erweitert eine Klasse um die Eigenschaften, die sie von ihren Oberklassen erbt. Man erhält so die volle Schnittstelle einer Klasse, wie sie von ihren Benutzern (Clients) verwendet werden kann, die sog. *Flat-Form*. Diese Erweiterung ist wichtig, weil sehr häufig nur wenige Operationen in einer Klasse spezialisiert werden, während eine größere Zahl von Operationen unverändert übernommen wird.

Um in Verbindung mit den eben beschriebenen **short** die volle Schnittstelle einer Klasse inklusive ihrer geerbten Operationen, die sog. *Flatshort-Form*, zu erhalten, ist das Unix Kommando "`flat Klassenname | short`" nötig. **flat** ist ebenfalls nicht interaktiv.

Die eigentliche Intention von **short** war es, keine separate Dokumentation zu den erstellten Klassen pflegen zu müssen und alle Informationen im Klassentext unterzubringen und dann mit **short** zu extrahieren. In diesem Sinne ist **short** in der Tat ein brauchbarer Ersatz für ein Online-Manual zu den einzelnen Klassen. Ein Vorteil ist sicher, daß die Dokumentation immer auf dem gleichen Stand wie die Implementation ist. Auch wenn Meyer behauptet, man würde durch dieses Vorgehen die Dokumentation "essentially for free" erhalten ([Meyer 88], Seite 346), so ist es doch notwendig, eine gute, strukturierte Dokumentation (als Kommentar) in den Quelltext zu schreiben, damit diese dann mit **short** extrahiert werden kann.

Als einen Browser kann man dieses Gespann sicherlich nicht bezeichnen. Auch eine klassenübergreifende Dokumentation, um die Verwendung der Klassen zu verdeutlichen, kann durch sie nicht ersetzt werden. (Trotzdem liefert ISE fast ausschließlich mit **short** produzierte Dokumentation zur Eiffel-Bibliothek aus.)

Im Umgang mit **short** und **flat** stellt sich die mangelnde Interaktivität als besonders hinderlich heraus. Bei jedem Verweis auf andere Klassen muß zu diesen gesondert die Schnittstelle

angefordert werden. Oft ist auch das Volumen der zu einer Klasse gelieferten Informationen so groß, daß sich kaum ein Vorteil gegenüber dem direkten Betrachten des Quelltextes ergibt.

Es ist nicht sichtbar, in welchem Zusammenhang eine Klasse verwendet wird. Man kann zwar die Oberklassen sehen, nicht aber, ob es bereits andere Klassen gibt, die eine bestimmte Klasse benutzen oder von ihr erben.

Auch fehlt es völlig an Möglichkeiten, die Informationsflut zu filtern. Immer werden alle Informationen zu einer Klasse präsentiert. So ist die Ausgabe von **short** für eine durchschnittliche Bibliotheksklasse oft mehrere Bildschirmseiten lang.

Als abschreckendes Beispiel zeigt Abb. 2 die keinesfalls kurze Ausgabe von **short** zur Klasse LINKED_LIST aus ISE Eiffel 3. Das Beispiel ist bereits um ca. 2/3 gekürzt und zeigt, wie unübersichtlich die Ausgabe sein kann. Die Flatshort-Form (also die volle Klassenschnittstelle) ist ca. dreifach so lang. Ein weiteres Filtern der Informationsflut ist nicht möglich. Die Kommentierung und Strukturierung ist direkt aus dem Quelltext übernommen und sieht bei weniger gut formatierten Quelltexten entsprechend unübersichtlicher aus.

```
indexing
	description: "Sequential, one-way linked lists"
	status: "See notice at end of class"
	names: linked_list, sequence
	representation: linked
	access: index, cursor, membership
	contents: generic
	date: "$Date: 94/01/04 12:25:59 $"
	revision: "$Revision: 1.13 $"

class interface LINKED_LIST [G]

creation
	make

feature  --  Access

	cursor: CURSOR
		--  Current cursor position

	first: like item
		--  Item at first position

	index: INTEGER
		--  Index of current position

	item: G
		--  Current item

	last: like item
		--  Item at last position

feature  --  Cursor movement

	back
		--  Move to previous item.

	finish
		--  Move cursor to last position.
		--  (Go before if empty)
	ensure
		empty_convention: empty implies before

	forth
		--  Move cursor to next position.

	go_i_th (i: INTEGER)
```

```
               --  Move cursor to `i'-th position.

       go_to (p: CURSOR)
               --  Move cursor to position `p'.

       move (i: INTEGER)
               --  Move cursor `i' positions. The cursor
               --  may end up `off' if the offset is too big.
       ensure
               moved_if_inbounds: ((old index + i) >= 0 and (old index + i) <= (count + 1))
implies index = (old index + i);
               before_set: (old index + i) <= 0 implies before;
               after_set: (old index + i) >= (count + 1) implies after

       start
               --  Move cursor to first position.
       ensure
               empty_convention: empty implies after


[...]


invariant
       prunable: prunable;

end -- class LINKED_LIST
```

Abb. 2: Short-Form der Klasse LINKED_LIST (gekürzt)

Der graphische Browser **good** unterstützt das interaktive Untersuchen des Systems und stellt die Beziehungen der Klassen graphisch dar. Prinzipiell ist **good** dazu gedacht, das Zusammenspiel aller Klassen eines Eiffel-Universums darzustellen. Dies gelingt aber nur sehr bedingt, da "die Arbeitsfläche beschränkt ist und die Funktionen zum Filtern der Informationsmenge nicht ausreichen" ([Strunk 92]). Auch ist kein Zugriff auf den Quelltext der dargestellten Klassen vorgesehen ([GamWeiMar 89]), und die Bedienung weist diverse Probleme auf. Statt auf **good** weiter einzugehen, beschreibe ich im nächsten Abschnitt die neue Entwicklungsumgebung von ISE, die **good** ersetzt hat (siehe [Strunk 92] für eine ausführlichere Kritik an **good**).

Die hier beschriebenen Werkzeuge wurden zusammen mit ISE Eiffel 2.x ausgeliefert. Bei anderen Compilern sind z.T. ähnliche, aber z.T. auch überhaupt keine Werkzeuge enthalten (so z.B. SiG Eiffel/S), so daß man bei ihnen auf das Handbuch und das Studium der Quelltexte für die Bibliotheken angewiesen ist. Bei der Arbeit mit nicht vom Compiler-Hersteller gelieferten Quelltexten ist man ausschließlich auf die direkte Arbeit mit dem Quelltext angewiesen.

5.2. Die Entwicklungsumgebung von ISE Eiffel 3

In der neusten Version von ISE Eiffel 3 hat man einige der Probleme der alten Werkzeuge erkannt. Die Entwicklungsumgebung ist erweitert worden [Meyer 94], und es steht innerhalb der Entwicklungsumgebung **ebench** ein Klassen-Browser, das sog. *Class Tool,* zur Verfügung.

Das Class Tool stellt per default den Quelltext der Klasse dar. Durch Anklicken eines Icons am unteren Rand des Fensters kann man stattdessen die bereits beschriebene Short-Form, Flat-Form oder Flatshort-Form darstellen lassen. Insoweit sind die bisherigen Umgangsformen nur auf eine graphische Oberfläche umgesetzt worden.

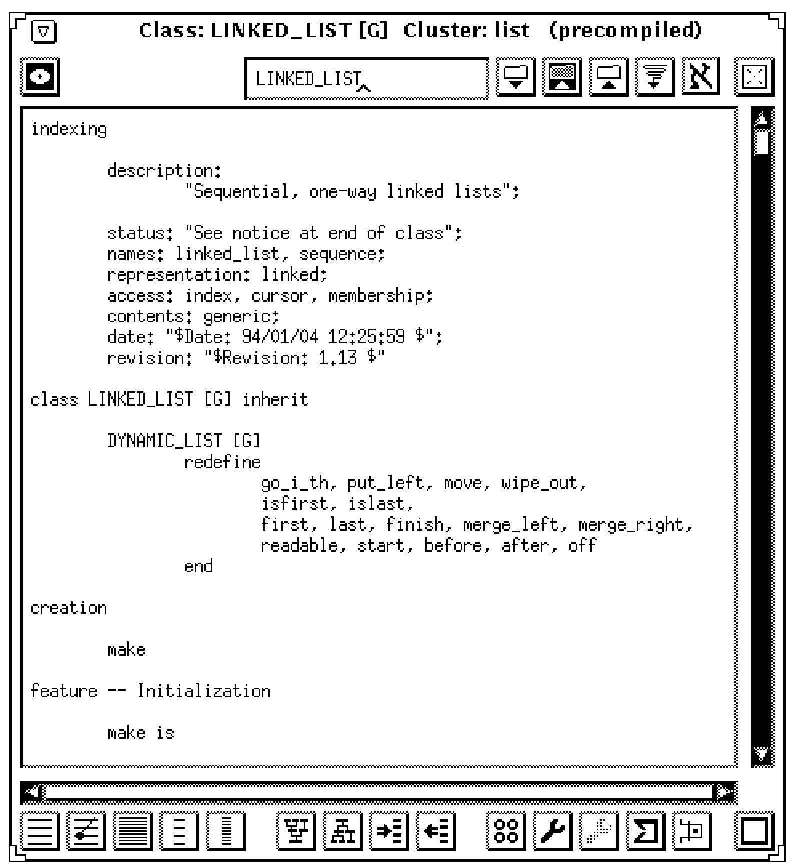


Abb. 3: Das Class Tool mit der Klasse LINKED_LIST

Alternativ können die Ober- oder Unterklassen in textueller Darstellung als Baum angezeigt werden. Weiterhin kann man sich statt des Klassentextes eine Liste der Benutzer (Clients) der Klasse, der benutzten Klassen (Supplier), der Attribute der Klasse, der Routinen der Klasse, der aufgeschobenen (deferred) Routinen, der einmalig (once) ausgeführten Routinen oder der externen Routinen anzeigen lassen.

Da man beliebig viele Fenster mit dem Class Tool öffnen kann, stehen alle Darstellungsformen auch parallel zur Verfügung. Änderungen können aber nur in der Quelltext-Darstellung vorgenommen werden. Da das Class Tool jedoch hauptsächlich als Browser ausgelegt ist, sind die Editor-Funktionen dementsprechend nur wenig mächtig. Die Darstellung in den anderen Fenstern wird bei Änderungen nicht aktualisiert.

Die Funktionen um nur Attribute, nur Routinen o.ä. darzustellen (siehe Abb. 4) beziehen sich immer nur auf die aktuell bearbeitete Klasse, so daß ein gezieltes, klassenübergreifendes Suchen nach bestimmten Merkmalen nicht möglich ist.

	Quelltext der Klasse
	anklickbarer Quelltext
	Flat-Form
	Short-Form
	Flatshort-Form
	Oberklassen
	Unterklassen
	Kunden
	Lieferanten
	Attribute
	Routinen
	Aufgeschobene Merkmale
	einmalig ausgeführte Routinen
	externe Routinen

Abb. 4: Die Icons des Class Tool

Die alten Umgangsformen sind fast 1:1 auf eine graphische Oberfläche übertragen worden. Es ist nun zwar leichter, eine Short-Darstellung zu bekommen, aber die grundsätzliche Kritik, daß weitere Einschränkungen der Informationsmenge möglich sein sollten, bleibt bestehen.

Es können immer entweder nur die Oberklassen oder nur die Unterklassen dargestellt werden und nicht die volle Vererbungshierarchie. Auch ist die textuelle Darstellung, die Vererbung durch Einrückungen darstellt, nur bedingt in der Lage, Mehrfachvererbung sinnvoll darzustellen. Da sie baumartig strukturiert ist, kann es vorkommen, daß bei Mehrfachvererbung eine Klasse auch mehrfach in der Vererbungshierarchie auftaucht, ohne daß zu erkennen wäre, daß sie anderen Orts noch einmal auftaucht.

Ein Beispiel: In Abb. 5 taucht die Klasse LINKED_TREE sowohl als Unterklasse von DYNAMIC_TREE als auch als Unterklasse von LINKED_LIST auf. Der geneigte Leser möge selbst beurteilen, ob es ihm aufgefallen wäre.

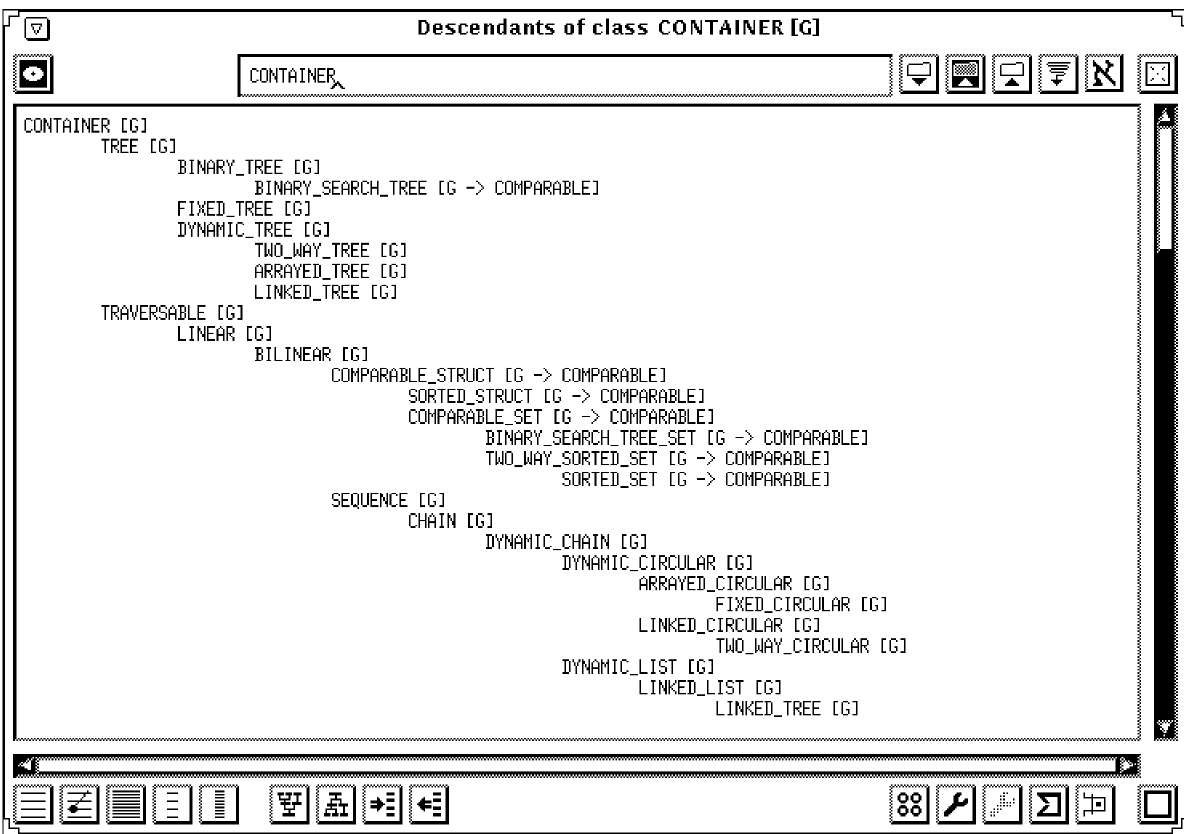


Abb. 5: Darstellung von Vererbung im Class Tool

Die Bedienung des Browsers ist gewöhnungsbedürftig und entspricht nicht unbedingt dem Styleguide der jeweiligen Plattform. Andererseits ist die Bedienung von ISE Eiffel 3 auf allen unterstützten Plattformen gleich, was man auch als Vorteil werten kann.

Ein Beispiel für die ungewöhnliche Bedienung: Um im Class Tool die Implementation einer verwendeten Klasse anzeigen zu lassen muß zunächst auf die anklickbare Darstellung umgeschaltet und dann der Klassenname mit der rechten Maustaste auf das Icon des Class Tools gezogen werden.

Die z.T. ins Feld geführte mangelnde Stabilität des Browsers (vergl. [Shipman 94]) ist bei mir nicht aufgetreten.

5.3. Browser für andere objektorientierte Sprachen

Da mir keine weiteren Browser für Eiffel zur Verfügung standen, habe ich auch einige Browser für andere objektorientierte Sprache betrachtet, um ihre Funktionalität mit der von Eiffel-Browsern zu vergleichen.

5.3.1. Der Browser von Visual C++

Visual C++ von Microsoft stellt eine integrierte Entwicklungsumgebung zur Verfügung, die unter anderem auch einen Browser enthält (vergl. [Microsoft 93]).

Ein Compilerschalter sorgt dafür, daß während des Compilierens zum jeweiligen Projekt eine Datenbank für den Browser aufgebaut wird. Diese Datenbank steht dann nach dem Linken des Projektes zur Verfügung.

Der Browser stellt eine Liste der Namen aller Klassen, Methoden etc. des Projektes dar. Zu jedem dieser Einträge kann man sich eine Liste mit den Definitionen des Namens und allen Verwendungen anzeigen lassen. Die dargestellten Einträge können auf Funktionen, Variablen, Typen, Makros oder Klassen eingeschränkt werden. Zusätzlich können sie noch durch Wildcards in begrenztem Maße eingeschränkt werden.

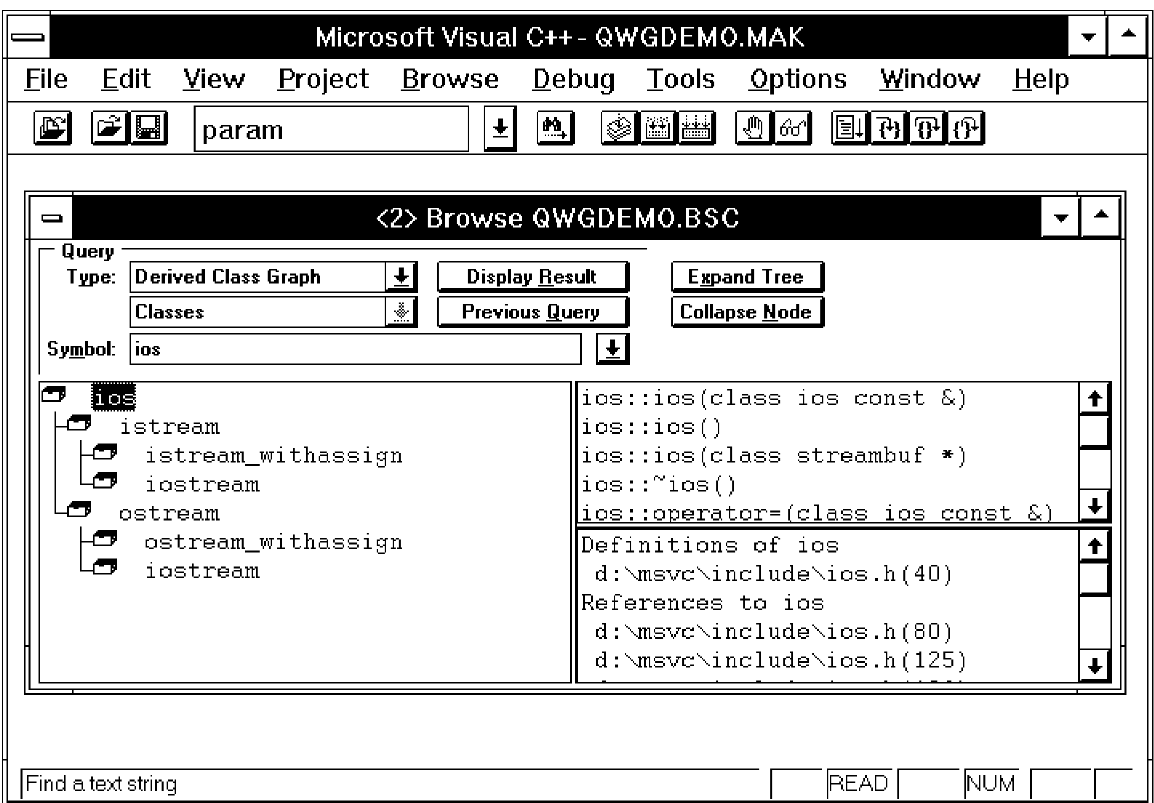


Abb. 6: Der Browser von Visual C++

Über den Call Graph bzw. den Caller Graph kann dargestellt werden, welche Funktion von einer anderen aufgerufen wird. Benutzt-Beziehungen auf Klassenebene können nicht betrachtet werden.

Um die Vererbungshierarchie darstellen zu lassen, kann man sich entweder den Graph der Oberklassen oder den Graph der Unterklassen anzeigen lassen. Um die Menge der angezeigten Informationen zu reduzieren, kann man einen Untergraph zu einem Knoten zusammenfassen lassen ("Colapse Node"). Eine Kombination der beiden Graphen ist nicht möglich, bzw. man müßte alle Unterklassen der Wurzelklasse anzeigen lassen. Durch die baumartige Anordnung läßt sich Mehrfachvererbung nicht sinnvoll darstellen.

Bei allen Darstellungen kann man durch einen Doppelklick auf das Symbol zu seiner Definition gelangen.

Da der Compiler die Informationen für die Browser-Datenbank sammelt, können nur compilierbare Quelltexte überhaupt untersucht werden. Sogar korrekte Klassen können nicht betrachtet werden, wenn es noch inkorrekte Klassen an anderen Stellen im Projekt gibt.

Eine zusätzliche Problematik ist die Konsistenz der Browser-Informationen: Nach Änderungen im Quelltext sind die Informationen im Browser inkonsistent, bis das Projekt erneut compiliert und gelinkt wird. Der Benutzer wird nicht über den inkonsistenten Zustand informiert.

Selbst bei kleinen Projekten wird der (ohnehin schon recht langsame) Compiler durch das Sammeln von Browser-Informationen erheblich verlangsamt, so daß man den Browser nur aktiviert, wenn man meint, ihn dringend zu brauchen.

Auch die Bedienung des Browsers ist etwas umständlich, da jede Veränderung der Such-einstellungen mit einem Klick auf "Display Result" bestätigt werden muß.

5.3.2. Sniff+

Sniff+ ist eine auf C und C++ ausgerichtete Entwicklungsumgebung, die von der Firma TakeFive, Salzburg vertrieben wird. Ein Ziel von Sniff+ ist es, die Entwicklung von großen Softwareprojekten[5] zu unterstützen. Eine ganze Reihe von Werkzeugen ist eng miteinander integriert, um die Bearbeitung der Quelltexte zu erleichtern ([Bischofberger 92]).

Die zu bearbeitenden Dateien werden mit dem **Projekt-Editor** in Projekte und Unterprojekte zusammengefaßt. Auf diese Weise ist es möglich, große Projekte in Module und Bibliotheken zu strukturieren und somit überschaubarer zu machen. Zusätzlich steigert dieses Vorgehen die Effizienz von Sniff+, da für alle Benutzer die Informationen über die verwalteten Quelltexte zentral vorgehalten werden können.

Der **Editor** erlaubt ein hypertext-artiges Navigieren zwischen den im Quelltext vorkommenden Methoden. Mit einem Doppelklick auf ein an der Seite aufgelistetes Symbol gelangt man wahlweise zu dessen Definition bzw. Implementation. (Anmerkung: In C++ ist es üblich, die Definition und Implementation von Methoden zu trennen, meist sogar in unterschiedliche Dateien.)

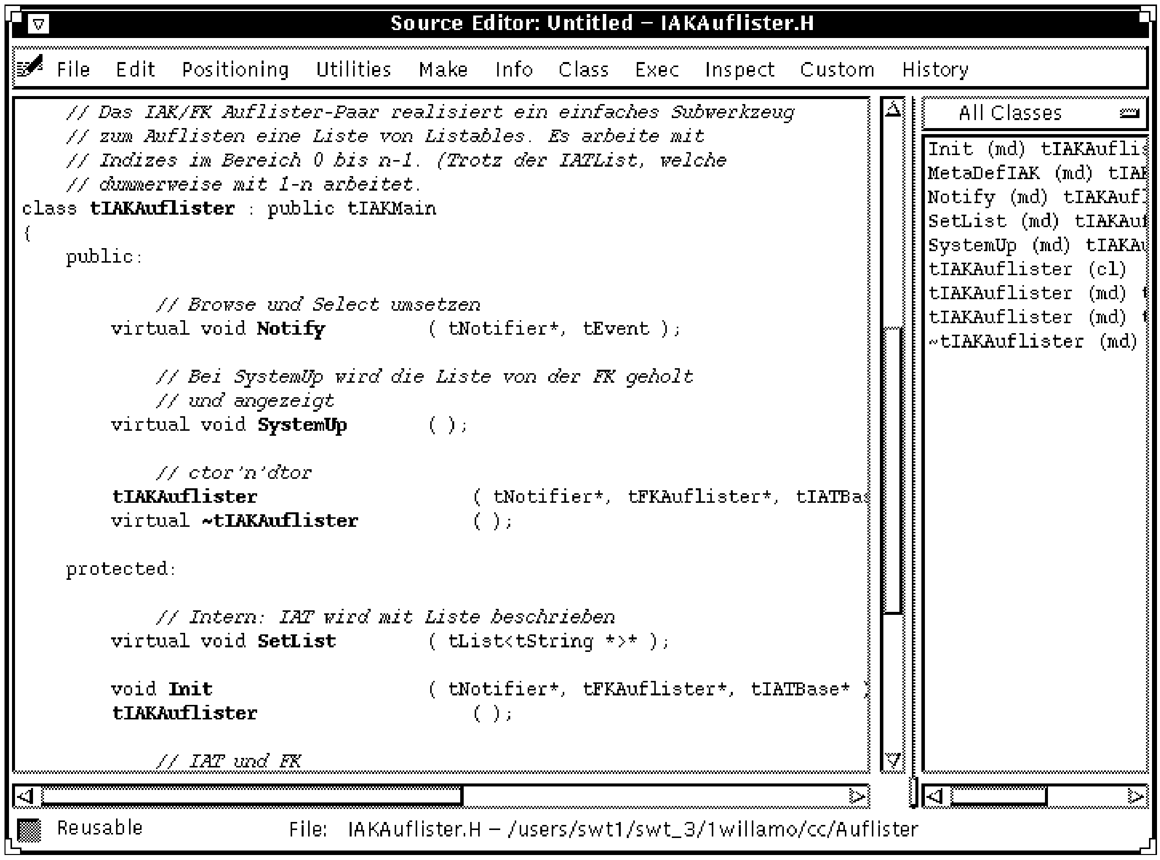


Abb. 7: Der Editor von Sniff+

[5] Laut [TakeFive 95] ist Sniff+ in der Lage, mehrere Millionen Zeilen Quelltext effizient zu verarbeiten.

Weiterhin markiert der Editor Klassen-, Methoden- und Variablennamen sowie Kommentare und abstrakte Klassen farblich bzw. mit einem anderen Font. Der Programmierer kann so auf einer höheren Ebene als der einer Textdatei arbeiten. Er ist jedoch nicht, wie bei der Arbeit mit vielen syntaxgesteuerten Editoren, in seiner Freiheit eingeschränkt, den Quelltext frei zu formatieren oder auch über längere Zeit hinweg syntaktisch inkorrekten Quelltext zu bearbeiten.

Sobald Änderungen des Quelltextes im Editor gespeichert werden, werden die Symbol-definitionen für den Editor und alle anderen Browser-Werkzeuge aktualisiert. Auch die Anzeige aller geöffneten Fenster wird sofort automatisch aktualisiert.

Der **Hierarchie-Browser** stellt die Vererbungshierarchien graphisch dar. Mit einem Doppel-klick auf eine Klasse gelangt man zu ihrer Definition bzw. Implementation oder über das Menü in eine beliebige andere Komponente von Sniff+.

Um die Zahl der angezeigten Klassen zu reduzieren, stehen mehrere Möglichkeiten zur Verfügung: Man kann eine Reihe von Klasse markieren und alle anderen ausblenden lassen, oder man kann die dargestellte Hierarchie auf die Ober- und Unterklassen einer bestimmten Klasse reduzieren lassen, an der man gerade interessiert ist. Zusätzlich besteht die Möglichkeit, die Anzeige auf bestimmte Unterprojekte zu reduzieren.

Auch im Hierarchie-Browser sind abstrakte Klassen durch kursive Schrift hervorgehoben, da sie oft abstrakte Protokolle implementieren, über die Klassen miteinander interagieren.

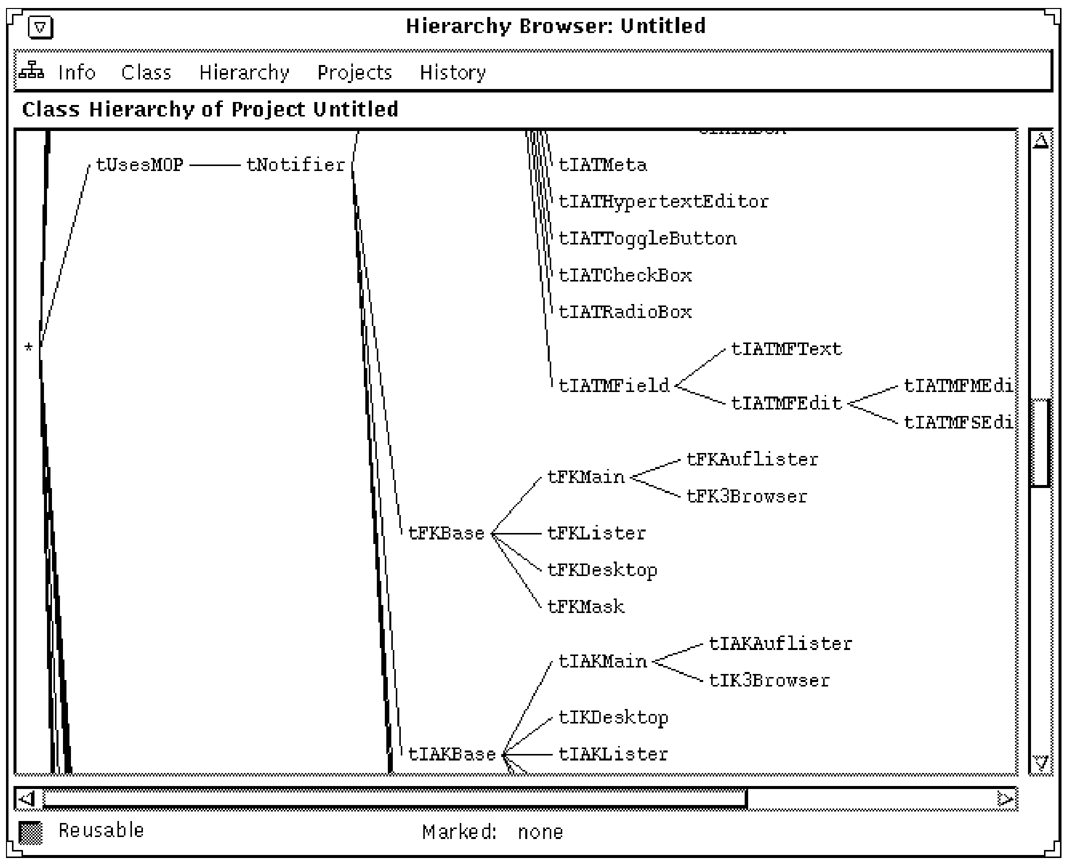


Abb. 8: Der Hierarchie-Browser von Sniff+

Der **Class-Browser** stellt alle Merkmale einer Klasse dar. Dabei können wahlweise alle Merkmale oder nur die in der Klasse selbst definierten Eigenschaften dargestellt werden. Wahlweise können auch nur ganz bestimmte Oberklassen aus dem (mit angezeigten) Vererbungsgraph ausgeblendet werden. Zu jedem dargestellten Merkmal wird die Klasse angezeigt, in der es definiert wurde.

Weiterhin können die angezeigten Merkmale nach verschiedenen Kategorien (Methoden, Variablen, Konstanten etc.) ein- und ausgeblendet und mit regulären Ausdrücken weiter eingeschränkt werden.

Einige Eigenschaften der angezeigten Merkmale werden visuell hervorgehoben (z.B. Sichtbarkeit, abstrakte Methode). Wahlweise werden auch die von dieser Klasse überladenen Eigenschaften der Oberklassen mit angezeigt.

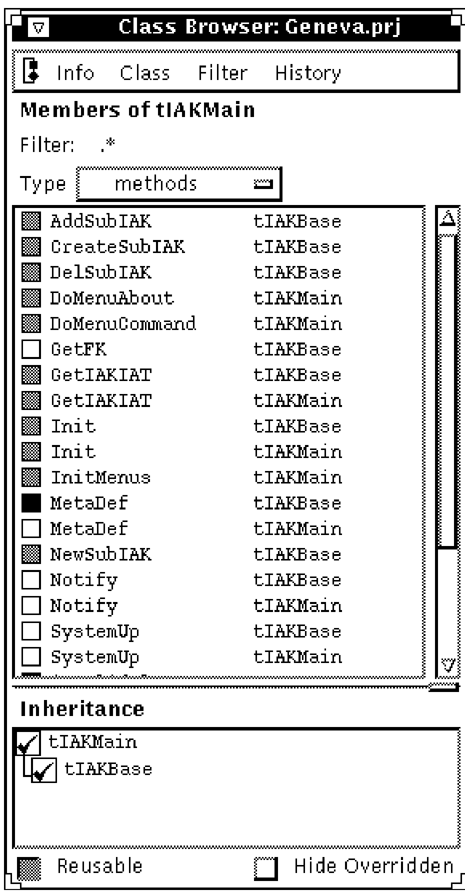


Abb. 9: Der Class-Browser von Sniff+

Der **Symbol-Browser** erlaubt es, nach den Definitionen bestimmter Symbole (Namen) über ganze Projekte hinweg zu suchen. Da der Symbol-Browser die Syntax von C bzw. C++ kennt, wird aber nicht auf Text-Ebene, sondern bereits auf der Ebene von Klassen- und Methodennamen etc. gearbeitet. So kann der Symbol-Browser beispielsweise alle Methodennamen eines Unterprojekts darstellen, die mit dem Präfix "Create" beginnen.

Um die Verwendung von Namen innerhalb des Quelltextes finden zu können, gibt es den **Retriever**. Er erlaubt die Suche nach regulären Ausdrücken über ein ganzes (Teil-)Projekt

hinweg. Er ist dem Unix-Werkzeug grep jedoch dadurch überlegen, daß er die Syntax von C/C++ kennt und die Suche auf bestimmte syntaktische Konstrukte einschränken kann. Er wäre beispielsweise in der Lage, alle Zuweisungen auf mit dem Präfix "Window" beginnende Variablen zu finden.

Die Extraktion der Informationen erfolgt mit einem sog. **Fuzzy-Parser**[6], so daß es nicht nötig ist, die Quelltexte zu compilieren, um sie browsen zu können. Mehr noch müssen die Quelltexte überhaupt nicht compilierbar sein, da der Fuzzy-Parser sich bemüht, auch aus Quelltexten mit syntaktischen Fehlern die Informationen über deren Struktur zu extrahieren. Auf diese Weise kann Sniff+ auch mit einer großen Zahl von Compilern eingesetzt werden, deren Syntax sich oft im Detail unterscheidet.

Normalerweise wird das Fenster des jeweiligen Werkzeugtyps mit den neuen Informationen gefüllt, sobald man andere Informationen selektiert. Es besteht zusätzlich aber auch die Möglichkeit, Fenster als "frozen" zu markieren, so daß für weitere Sichten zusätzliche Fenster geöffnet werden.

Die Stärke von Sniff+ besteht darin, daß man auf der Ebene von symbolischen Informationen und Konstrukten der Programmiersprache arbeiten kann, da alle Werkzeuge die Syntax der Programmiersprache kennen und die Arbeit mit ihr unterstützen.

[6] Eine detailiertere Beschreibung von Fuzzy-Parsern befindet sich in Kapitel 6.3.3 und in Kapitel 7.

5.4. Funktionalität von Browsern

Die folgende Tabelle zeigt die Funktionalität der verschiedenen Browser im Überblick:

	short/flat	**ISE Eiffel 3**	**Visual C++**	**Sniff+**
Vererbungs-Graph	nein	ja, baumartig (nur Ober- oder nur Unterklassen)	ja, baumartig (nur Ober- oder nur Unterklassen)	ja
Darstellung von wiederholter Vererbung	-	nein (Klassenname taucht mehrfach auf)	nein (Klassenname taucht mehrfach auf)	ja
Vererbungs-Graph filterbar	-	nein	ja	ja
Benutzung	nein	ja	nur für Funktionen und Methoden, nicht auf Klassen-Ebene	ja (Retriever, Cross-Referencer geplant)
Benutzung berücksichtigt Polymorphie	nein	nein	nein	nein
Symbole	nein	nur zur aktuellen Klasse	ja	ja
Symbole im Editor markiert	nein	nein	ja	ja
Symbole filterbar nach Kategorien	nein	ja	ja	ja
Symbole filterbar nach Wildcards	nein	nein	ja (eingeschränkt, keine echten regulären Ausdrücke)	ja
muß Quelltext compilierbar (fehlerfrei) sein	ja (implementations-abhängig)	ja	ja, muß sogar linkbar sein	nein
Aktualisierung der Datenbasis	manuell (explizite Anforderung)	automatisch, nach Speicherung der Änderungen	beim nächsten Compilieren <u>und</u> Linken des Projekts (bis dahin inkonsistent)	automatisch, nach Speicherung der Änderungen
verschiedene Sichten gleichzeitig	ja (mehrere Fenster bzw. Ausdrucke nötig)	ja (sofern keine ungespeicherten Änderungen)	nein	ja
Aktualisierung der verschiedenen Sichten	manuell (explizite Anforderung)	nein	-	automatisch beim Speichern der Änderung
GUI	nein	ja	ja	ja
Verfügbarkeit	im Lieferumfang des Compilers	im Lieferumfang des Compilers	im Lieferumfang des Compilers	muß extra erworben werden
Projekt-Konzept	nein	ja	ja	ja
untersuchte Version	ISE Eiffel 2 (nach [Meyer 88])	ISE Eiffel 3	Visual C++ 1.0	Sniff+ 1.1

5.5. Anforderungen an einen Eiffel-Browser

Ein Eiffel-Browser sollte es dem Entwickler erlauben, auf einer möglichst hohen Abstraktionsebene zu arbeiten. Wichtig ist die Visualisierung des Software-Designs und nicht die Implementierungsdetails in einer speziellen Sprache (vergl. [KilGryZül 93], S. 69).

Im Sinne des Arbeitens auf möglichst hohen Niveaus wäre es wünschenswert, wenn der Browser auch die verwendeten Design-Pattern (vergl. [Gamma 92] und [Gamma 95]) darstellen könnte.

Um auch größere Projekte bearbeiten zu können, ist es unbedingt notwendig, die dargestellte Informationsmenge filtern zu können. Hier bietet sich ein Filtern nach Kategorien (Klassen, Methoden, Methoden mit speziellen Eigenschaften) an. Weiterhin sollte es aber auch möglich sein, die dargestellten Symbole mit Wildcards zu filtern bzw. explizit anzugeben, welche Klassen z.B. in einer Vererbungshierarchie angezeigt werden sollen.

Eins der wichtigsten Strukturierungsmittel in der objektorientierten Programmierung ist die Vererbung (auch wenn sie von verschiedenen Schulen unterschiedlich eingesetzt wird). Daher muß ein Browser für Eiffel die Vererbungs-Hierarchie möglichst flexibel darstellen können.

Wünschenswert wäre auch die Darstellung, welche Klassen einander benutzen. Eine solche Darstellung ist aber nur aussagekräftig, wenn Polymorphie berücksichtigt wird. Da es aber erst zur Laufzeit feststellbar ist, welche Klassen wirklich benutzt werden, müßten alle möglichen Benutzungs-Beziehungen vom Browser dargestellt werden, was zu einer unübersichtlichen und damit entwerteten Darstellung führt (vergl. [Bischofberger 94a]).

Es sollte möglich sein, mehrere Sichten auf den Quelltext gleichzeitig zu haben. Wichtig ist aber auch, daß diese bei Änderungen im Quelltext sofort (und automatisch !) aktualisiert werden.

Auch ein noch so guter Browser wird nicht alle Fragen über den Quelltext auf einer abstrakten Ebene beantworten können. Daher sollte auch der Rückgriff auf den Quelltext mit einem integrierten Editor immer möglich sein. Aber auch im Editor sollte die bearbeitete Sprache z.B. durch Syntax-Highlighting (hervorgehobene Anzeige verschiedener syntaktischer Strukturen, z.B. Methodennamen, Kommentare etc.) unterstützt werden.

Um den Browser von speziellen Eiffel-Implementationen unabhängig zu halten, muß er auch ein eigenes Projekt-Konzept haben. Zwar wurde von Meyer in der Sprachreferenz eine Beschreibungssprache für Eiffel-Universums vorgeschlagen (LACE), aber alle vorhandenen Implementationen haben eine eigene inkompatible Variante implementiert. Um aber die verwendeten Bibliotheksklassen finden zu können, muß der Browser den Umfang des Eiffel-Universums kennen, d.h. der Programmierer muß die Möglichkeit haben, alle zum Universum gehörigen Teile dem Projekt hinzuzufügen.

Prinzipiell sollte eine möglichst flexible Handhabung möglich sein, denn die hier zu unterstützende Tätigkeit ist sehr anspruchsvoll, und der Entwickler einer Browsers wird niemals alle möglichen Umgangsformen mit dem von ihm entwickelten Browser vorhersehen können. Der Benutzer sollte also dynamisch die dargestellten Informationen filtern und durchsuchen können.

6. Möglichkeiten zur Erstellung der Datenbasis für den Browser

Es wäre wünschenswert, wenn der Browser die entsprechenden Informationen vom jeweils verwendeten Eiffel-Compiler bereitgestellt bekommen würde. Man würde so das Wissen über den syntaktischen Aufbau der Sprache mit all seinen Details in einem Werkzeug kapseln. Es wird sich jedoch zeigen, daß die existierenden Compiler aufgrund der unterschiedlichen Anforderungen von Browsern und Compilieren dieser Anforderung nicht gerecht werden können.

Es gibt mehrere Wege, um an die für den Browser notwendigen Informationen über die Struktur des zu untersuchenden Programms zu kommen:
- aus dem Laufzeitsystem,
- aus der Datenbank des Compilers,
- aus dem Quelltext.

6.1. Aus dem Laufzeitsystem

Die erste Möglichkeit besteht darin, die Informationen vom Laufzeitsystem zu gewinnen. Da Eiffel das Vorhandensein von Typinformationen zur Laufzeit voraussetzt, stellt das Laufzeitsystem einiger Eiffel-Umgebungen diese Informationen auch dem Anwendungsprogrammierer zur Verfügung. Die Typinformationen werden vom Laufzeitsystem für Typprüfungen, z.B. beim "reverse assignment attempt", benötigt, bei dem versucht wird, einer Variablen der Unterklasse eine Variable der Oberklasse zuzuweisen.

Die zur Verfügung stehenden Informationen umfassen aber meist wenig mehr als die Vererbungshierarchie und die Methoden der einzelnen Klassen. Man könnte mit diesen Informationen also wenig mehr machen, als einen Vererbungsgraph aufzuzeigen. Schon die Zuordnung der Klassen zu ihren Quelltexten ist nicht immer feststellbar.

Auch ist die Schnittstelle, um diese Informationen zu bekommen, stark von der verwendeten Eiffel-Implementation abhängig, so daß diese Informationsquelle für einen vom Compiler unabhängigen Browser nicht brauchbar ist.

Zudem wäre es nur möglich, Programme zu browsen, die tatsächlich lauffähig sind. Da in der Entwicklung befindliche Programme die meiste Zeit nicht ablauffähig sind (zumindest in den Momenten, in denen sie gerade geändert werden und der Browser am nötigsten gebraucht wird), scheidet diese Möglichkeit vollends aus.

6.2. Aus der Datenbasis des Compilers

Eine andere Möglichkeit, an die Informationen zu kommen, wird ebenfalls von vielen Eiffel-Implementationen angeboten: Nachdem in der ersten Stufe des Eiffel-Compilers der Quelltext analysiert wurde, werden die gewonnenen Informationen in einer lokalen Datenbasis abgelegt. Der Eiffel-Compiler benötigt sie für seine weitere Arbeit. Die Struktur dieser Datenbasis wird bei einigen Compilern offengelegt, so daß ein Browser darauf zugreifen könnte.

Die in der Datenbasis des Compilers verfügbaren Informationen reichen für die meisten Belange eines Browsers aus. Daher greifen die von den Compiler-Herstellern mitgelieferten

Browser, z.B. bei ISE Eiffel 3, Eon/Eiffel (vergl. [Eon 94]), meist auf diese Datenbasis zu. Bei dieser Methode müßte das zu untersuchende Programm nicht ablauffähig sein, sondern nur syntaktisch korrekt. Bereits dies stellt aber eine starke und für das Browsen unnötige Einschränkung dar.

Das Problem, daß die Schnittstelle zu den Daten stark herstellerabhängig ist, besteht hier allerdings ebenfalls, so daß einem von einer speziellen Eiffel-Implementation unabhängigen Browser der Zugriff auf die Datenbasis des Compilers nicht offensteht (oder er zumindest eine Anpassung für jede Eiffel-Implementation benötigen würde). In [Strunk 92] wird ein Browser beschrieben, der die Datenbasis von ISE Eiffel 2 benutzt.

6.3. Aus dem Quelltext

Als letzte Informationsquelle steht der zu untersuchende Quelltext selbst zur Verfügung, den der Browser natürlich auch direkt analysieren kann. Dies macht ihn von jedem verwendeten Eiffel-Compiler unabhängig. Der nötige Zusatzaufwand bei der Erstellung macht jedoch auch den Einsatz spezieller Analyse-Techniken möglich, die mit syntaktisch nicht korrekten Programmen umgehen können.

Im Gegensatz zum Compiler benötigt der Browser sehr viel weniger Informationen aus dem Quelltext. Für ihn ist nur die grobe Struktur wichtig, die sich meist auch aus einem unkorrekten Quelltext ablesen läßt.

Der Eiffel-Compiler dagegen muß auf der korrekten Syntax (wie definiert in [Meyer 91]) bestehen. Insbesondere in den Feature-Implementationen (dort, wo ausführbarer Code erzeugt werden muß) kann z.B. das Fehlen eines Semikolon zu einer anderen Interpretation des Programmtextes führen. Da der Programmierer aber nicht in syntaktischen Konstrukten, sondern in höheren Abstraktionen von Klassen und deren Umgangsformen denkt, sind solche Fehler recht häufig. Nötig ist also eine Analyse-Methode, die, soweit möglich, die Struktur des Quelltextes analysiert, aber die anderen Teile des Quelltextes (und die möglicherweise darin enthaltenen syntaktischen Fehler) ignoriert.

6.3.1. Kurzeinführung Compilerbau

Der Compilerbau (die Erstellung von Compilern) ist eine der älteste Sparten der Informatik. Basierend auf der Automatentheorie wurden Methoden entwickelt, um *formale Sprachen* (z.B. mit Hilfe von Grammatiken) exakt zu beschreiben.

Eines der klassischen Probleme der Automatentheorie ist das sog. *Wortproblem*. Es untersucht, ob eine Zeichenkette zu einer formalen Sprache gehört (und somit ein *Wort* der Sprache ist) oder nicht. Man kann die formalen Sprachen anhand der Struktur ihrer Grammatiken in Klassen unterteilen, bei denen das Wortproblem jeweils mit einer bestimmten Zeitkomplexität gelöst werden kann.

Übertragen auf Programmiersprachen stellt ein Programm (bzw. eine einzeln compilierbare Einheit) ein *Wort* der Sprache dar. Für dieses Programm ist zu prüfen, ob es Element der Sprache (d.h. syntaktisch korrekt) ist oder nicht. Um diese Aufgabe zu bewältigen, kann man aus der Grammatik einen sog. *akzeptierenden Automaten* konstruieren, der das zu unter-

suchende Wort (sprich Programm) als Eingabe erhält und als Ergebnis die Korrektheit des Programms bestätigt oder eben nicht.

Die Entwicklung ist schon seit langem so weit, daß solche (sogar minimale) akzeptierenden Automaten nach festen Regeln aus der Grammatik erzeugt werden können. (Dies gilt nur für eine bestimmte Klasse von Sprachen (die sog. *kontextfreien Sprachen*). Da die in diesem Zusammenhang wichtigen Programmiersprachen aber praktisch alle so entworfen sind, daß sie dieser Klasse angehören, vernachlässige ich die Diskussion anderer Sprachklassen.)

Die *Grammatik*, die eine Sprache beschreibt, besteht u.a. aus einem *Startsymbol*, Ableitungsregeln (sog. *Produktionen*) und einem *Alphabet*, aus dem die Worte der Sprache gebildet werden können.

Zur vollständigen Beschreibung von *Programmiersprachen* reichen Grammatiken und formale Sprachen jedoch nicht aus. Zum einen muß zusätzlich noch die Sematik (Bedeutung) der Konstrukte beschrieben werden, und zum anderen werden häufig über die Grammatik hinaus noch zusätzliche Regeln für die Syntax verwendet[7].

Bei Programmiersprachen besteht ein *Zeichen* des Alphabets jedoch, der menschlichen Lesbarkeit wegen, meist aus mehreren (ASCII-)Zeichen. Im Compilerbau ist daher die erste Stufe die sog. *lexikalische Analyse*. Sie fügt die (ASCII-)Zeichen des Programmquelltextes zu *Zeichen* der Sprache zusammen. Dieser Programmteil wird *Scanner* genannt. Die erkannten lexikalischen Einheiten werden in sog. *Token* (üblicherweise in Aufzählungstypen) kodiert, die die lexikalischen Symbole (Schlüsselworte, Bezeichner, Operatoren etc.) darstellen.

Das Beschreibungsmittel, um festzulegen, welche (ASCII-)Zeichen zu einem Zeichen der Sprache zusammengefügt werden dürften, sind *reguläre Ausdrücke*. Die Erzeugung von Scannern aus regulären Ausdrücken kann automatisch erfolgen. Das bekannteste Werkzeug hierzu ist Lex (siehe [LevMasBro 92]).

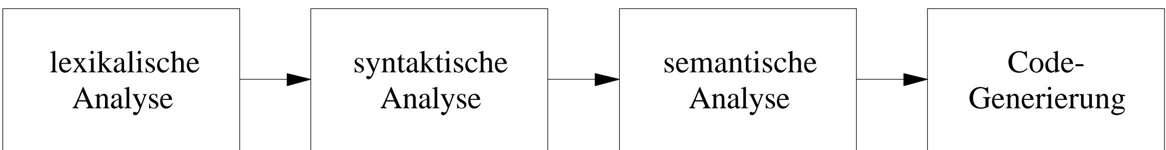


Abb. 10: Die Phasen eines Compilers

Die zweite Stufe eines Compilers ist die sog. *syntaktische Analyse*. Sie entspricht dem akzeptierenden Automaten und stellt fest, ob ein Programm syntaktische Fehler beinhaltet. Dieser Programmteil wird *Parser* genannt. Entsprechend der automatischen Konstruktion der akzeptierenden Automaten können auch Parser automatisch aus einer Grammatik generiert werden.

[7] Der Hauptgrund herfür ist, die Grammatik möglichst einfach in einen Compiler übersetzen zu können.

Während der syntaktischen Analyse wird eine sog. *Symboltabelle* aufgebaut, die alle verwendeten Symbole (Funktionsnamen, Variablen etc.) sowie ihren Typ (und diverse Zusatzinformationen) speichert.

Der Parser arbeitet ausschließlich auf dem Strom von sog. *Token*, die der Scanner liefert. (Zusätzlich kann er Angaben über den Ort des Auftretens und andere Details zum aktuellen Token vom Scanner erfragen.)

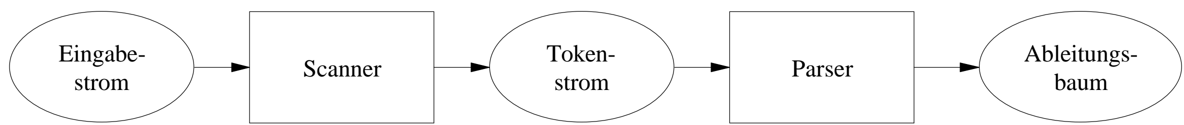


Abb. 11: Aufgabenteilung zwischen Scanner und Parser

Die beiden wichtigsten Methoden zur Implementierung eines Parsers sind der *rekursive Abstieg* (engl. recursive descent), der Top-Down vom Startsymbol aus vorgeht, und die *Shift-Reduce* Methode, die Bottom-Up arbeitet.

Eine sehr kompakte Anleitung, wie man Parser nach der Methode des rekursiven Abstiegs von Hand aus den Syntaxgraphen einer Sprache erstellen kann, findet sich in [Wirth 77].

Bisher nicht erwähnt habe ich, daß die syntaktische Prüfung ja kein Selbstzweck ist, sondern der Erzeugung eines ablauffähigen Programms dienen soll. Hierzu wird vom Parser üblicherweise ein sog. *Ableitungsbaum* erzeugt, der beschreibt, welche Produktionen der Grammatik angewendet wurden. Die Produktionen tragen die Semantik der Programmiersprache. Die folgenden Stufen des Compilers heißen daher *semantische Analyse* und *Code-Generierung,* d.h. aus dem Ableitungsbaum wird ein ausführbares Programm erzeugt. (Da dieser Vorgang für mich im weiteren keine Rolle spielt, gehe ich nicht weiter darauf ein.)

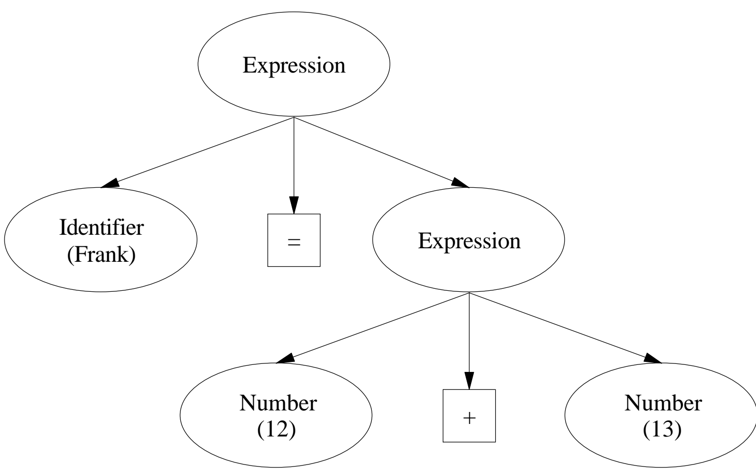


Abb. 12: Beispiel für einen Ableitungsbaum (vergl. [LevMasBro 92])

Ebenso wie die Scanner können auch die Parser weitgehend automatisch aus der Grammatik erzeugt werden. Der bekannteste Parser-Generator ist YACC (Yet Another Compiler-Compiler, siehe [LevMasBro 92]). Ein weiteres recht verbreitetes Werkzeug, das sowohl einen

Scanner-Generator als auch einen Parser-Generator enthält, ist das Cocktail-Toolkit (siehe [Grosch 92]).

Das übliche Vorgehen beim Compilerbau ist es, die Grammatik der Programmiersprache mit einer Grammatik (und regulären Ausdrücken) zu beschreiben und dann automatisch einen Scanner und einen Parser generieren zu lassen. Meist wird nur die Code-Generierung "von Hand" programmiert. Die so erstellten Compiler sind aufgrund der ausgereiften Automatentheorie recht effizient und vor allen recht einfach zu erzeugen.

6.3.2. Traditionelle Parser

Im ersten Anlauf zu dieser Studienarbeit habe ich mich bemüht, einen frei verfügbaren Parser für Eiffel 3 zu finden, um mir die Arbeit zu ersparen, einen eigenen schreiben zu müssen. Ich hatte die Hoffnung, diesen Parser dann nur geringfügig modifizieren zu müssen, um ihn fehlertolerant zu machen.

Von der Freien Universität Berlin stammt der frei verfügbare Eiffel 3 Parser **ep** [Groeber 92a]. Er wurde mit dem Cocktail-Toolkit erstellt und erzeugt als Ausgabe einen abstrakten Syntaxbaum (AST - abstract syntax tree), der eine Mischung aus Ableitungsbaum und Symboltabelle darstellt. (Da der Shareware Eiffel-Compiler Eon/Eiffel ebenfalls mit dem Cocktail-Toolkit erstellt worden ist (vergl. [Leonard 94]), wäre es sogar denkbar gewesen, später diesen Compiler und meinen Browser auf einer gemeinsamen Datenbasis arbeiten zu lassen.)

ep ist mit ca. 4000 Zeilen / Sek. (laut [Groeber 92b]) recht schnell und entspricht exakt der Eiffel 3 Syntax aus der Sprachdefinition [Meyer 91]. Meine Hoffung, ihn als Unterbau für einen Browser verwenden zu können, hat sich aber trotzdem nicht erfüllt.

Bei der Verwendung von **ep** wurde sehr schnell die Schwäche von traditionell erstellten Parsern als Grundlage für einen Browser deutlich: Jede kleine syntaktische Unkorrektheit führt zu einer Fehlermeldung (und bei **ep** sogar zu einem Abbruch) des Parsers. Bei Versuchen mit **ep** wurde deutlich, wie weit sich die wenigen verfügbaren Eiffel-Implementationen vom Sprachstandard ([Meyer 91]) entfernt haben. Nicht einmal 50% der nach Ansicht der jeweiligen Compiler korrekten (!) Beispielprogramme der Compiler von SiG bzw. ISE konnten vollständig geparst werden. In der Entwicklung befindliche Programme haben sicherlich kaum eine Chance, mit diesem Parser erfolgreich bearbeitet zu werden.

Die Änderungen am Quelltext der Beispielprogramme, um sie an den Parser **ep** anzupassen, waren zwar minimal, für einen Praxiseinsatz ist es jedoch nicht zu vertreten, daß man sich beim Schreiben des Quelltextes an die Schnittmenge zweier unterschiedlicher Compiler-Syntaxen halten muß. Damit hatte sich **ep** als Unterbau für einen Browser als unbrauchbar erwiesen.

Dieses Problem liegt nicht bei **ep**, sondern in der Art, wie die automatisch generierten Parser üblicherweise arbeiten. Zwar gibt es Ansätze zum Wiederaufsetzen nach einem Fehler (zur Fortsetzung der Analyse), aber ihre Hauptaufgabe liegt im Finden von syntaktischen Fehlern und <u>nicht</u> im Erkennen der Programmstruktur.

Ein Lösungsansatz, die Probleme traditioneller Parser im Hinblick auf ihre Fehlertoleranz zu beheben, wäre, die Grammatiken der von ihnen akzeptieren Sprachen zu modifizieren (sog. *Fehlerproduktionen* hinzuzufügen). So können einige Probleme (z.B. fehlende oder überzählige Semikolons etc.) beseitigt werden. Auf diese Weise kann man aber nicht alle möglichen Fehler antizipieren. Problematisch werden dabei auch die in der Grammatik auftretenden Mehrdeutigkeiten. Dieses Vorgehen ist meiner Ansicht nach nur bei Sprachen mit sehr einfacher Grammatik praktikabel.

6.3.3. Fuzzy-Parser

Ein Lösungsansatz wird in [Bischofberger 92] vorgeschlagen: Fuzzy-Parser. Bischofberger schreibt

> "...we have developed a fuzzy recursive descent parser which has only partial understanding of C++, which can deal with incomplete software systems containing errors, and extracts information about where and how the symbols (e.g., classes, members, variables) of a software system are declared and where they are defined. This way only declarations and the headers of (member) functions (i.e., return code, name and argument list) have to be parsed while the executable code in the bodies of (member) functions can be ignored."

Unter einem Fuzzy-Parser versteht man also einen Parser, der versucht, die Struktur des Quelltextes trotz syntaktischer Fehler zu analysieren. Er ist speziell auf die Bedürfnisse eines Browsers zugeschnitten und bemüht sich, nur diese Informationen zu sammeln, so daß ihn Fehler in anderen Bereichen nicht stören können.

Ein Fuzzy-Parser ermöglicht es auch, den Browser mit verschiedenen Eiffel-Dialekten zu verwenden, die eine leicht abweichende Syntax haben (siehe z.B. [Wilson 94a], [SiG 92]).

Da Fuzzy-Parser völlig andere Zielsetzungen haben als traditionelle Parser, sind die Konstruktionsmethoden auch nur bedingt übertragbar. Bei der Beschreibung der Implementation des Fuzzy-Parsers für Eiffel werde ich hierauf noch weiter eingehen.

Ein Beispiel für die abweichende Syntax verschiedener Eiffel-Compiler war die Zulässigkeit bzw. Verpflichtung, an bestimmten Stellen Semikolons zu setzen. Ein Fuzzy-Parser erkennt die Struktur trotzdem, während ein Parser, der auf die korrekte Syntax Wert legt, einen Fehler meldet.

Da ein Fuzzy-Parser nur an der Struktur des Quelltextes interessiert ist, ist er von allen Änderungen, die den ausführbaren Teil des Quelltextes betreffen, vollkommen unberührt.

7. Implementation eines Fuzzy-Parsers

Ein Fuzzy-Parser kann nur bedingt die Methoden und Toolkits des klassischen Compilerbaus [AhoSethiUllmann 88] nutzen. Dies liegt darin begründet, daß beim klassischen Compilerbau alle (auch noch so geringfügigen) Abweichungen von der Syntax gefunden werden sollen. Für die erfolgreiche Arbeit eines Fuzzy-Parsers ist nur insoweit eine syntaktische Korrektheit erforderlich, als einige wenige Schlüsselelemente erkannt werden müssen. Mehr sollte aus obigen Gründen nicht vorausgesetzt werden.

Die lexikalische Analyse ist beiden Methoden gemeinsam. Hier könnte ein automatisch generierter Scanner verwendet werden. Da der Scanner aber der kleinere Teil des Scanner/Parser-Gespanns ist, habe ich ihn zunächst ebenfalls von Hand geschrieben. (Da die meisten Generatoren, z.B. Lex, ohnehin nur C-Code generieren, müßte man ihn wohl ohnehin in einer Klasse kapseln.) Bei der Syntaxprüfung gelten andere Ziele: Es sollen nicht die vorhandenen Fehler gefunden, sondern trotz möglicher Fehler so viel wie möglich Informationen extrahiert werden. Auf Typprüfungen o.ä. wird gänzlich verzichtet.

Zur Syntaxanalyse bei Fuzzy-Parsern eignen sich Top Down-Verfahren wie die Methode des rekursiven Abstiegs besser als Bottom Up-Verfahren (z.B. Shift-Reduce Methode), da sie nicht so eng an den Produktionen (Regeln) hängen und es erlauben, bestimmte Teile des Token-Stream zu überlesen, da zu jedem Zeitpunkt klar ist, in welchem semantischen Zusammenhang man sich befindet. Compiler-Generatoren (z.B. YACC, Cocktail) erzeugen meist Shift-Reduce Parser, da diese schneller arbeiten und mächtigere Grammatiken verarbeiten können.

Der von mir erstellte Fuzzy-Parser sammelt deutlich weniger Informationen über den Quelltext als ein traditioneller Parser, wie beispielsweise der bereits beschriebene **ep**.

Ein Nachteil eines von Hand gebauten Parsers ist, daß er auf eine spezielle Aufgabe (hier: als Back-End eines Browsers) zugeschnitten ist und die auszuführenden Operationen im Quelltext nicht von der Syntax und dem Prozeß der Informationsgewinnung getrennt sind. Bei generierten Parsern sind das eigentliche Parsen und die Verwendung der gewonnenen Operationen konzeptionell besser getrennt und werden erst vom Generator in den Parser integriert (siehe z.B. Action-Teil eine YACC-Regel). Diesem Problem kann man begegnen, indem man den Parser als Framework realisiert (siehe Abschnitt über die von mir gewählte Architektur).

Dieser Nachteil ist gleichzeitig auch ein Vorteil: Da zu jedem Zeitpunkt genau bekannt ist, welche Informationen noch benötigt werden, wird nicht zuviel Quelltext exakt geparst, sondern kann weitläufig überlesen werden, was der Fuzzy-Eigenschaft (d.h. der Fehlertoleranz) sehr entgegenkommt.

7.1. Architektur

Die klassische Aufteilung in Scanner und Parser kann und sollte auch bei Fuzzy-Parsern beibehalten werden. Die Anforderungen an den Scanner eines Fuzzy-Parsers unterscheiden sich kaum von denen an einen traditionellen Scanner (mit Ausnahme der Tatsache, daß Fehler nicht zu einem Abbruch führen dürfen, was bei Scannern aber ohnehin unüblich ist.)

Scanner und Parser sind bei mir in der Art eines Frameworks (nach [Gamma 92] "Framework im kleinen") zur Verfeinerung durch den Klienten konstruiert. Ich habe also einen (in Grenzen) allgemeinen Eiffel-Parser implementiert, der dann vom Parser für Sniff+ verfeinert wird. Auf diese Weise wird eine strikte Trennung erreicht zwischen dem Extrahieren (Beschaffen) der Informationen über den Quelltext und der Verwendung dieser Informationen.

Die Verfeinerung erfolgt durch Konkretisierung (d.h. Implementierung) einiger speziell dafür vorgesehener Methoden. Diese Methoden werden an passenden Stellen (z.B. beim Auffinden einer Klasse) aufgerufen. Im allgemeinen Eiffel-Parser sind diese Methoden leer. (Man kann / sollte sie sich als abstrakt vorstellen, da es kaum Sinn macht, den Parser ohne jegliche Verwendung der gewonnenen Informationen überhaupt zu benutzen. Da oft jedoch nur einige der Methoden benötigt werden, habe ich alle im allgemeinen Parser als leer implementiert, so daß sie in den Verfeinerungen nicht beachtet werden müssen.)

Alle zur Konkretisierung vorgesehenen Methoden in *EiffelScanner* und *EiffelParser* heißen *ActOn...()* (z.B. *ActOnClassname()*, *ActOnFeature()* etc.).

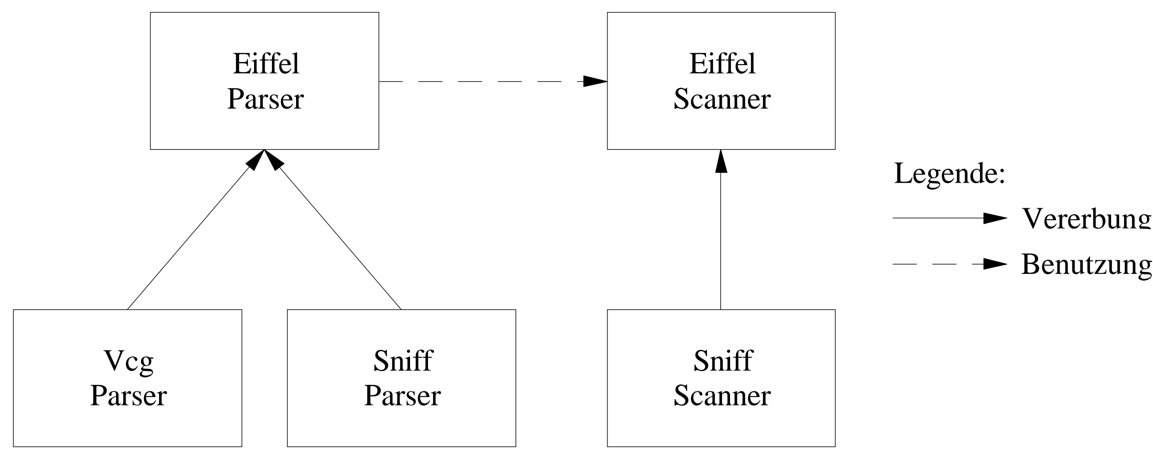


Abb. 13: Klassendiagramm

In den Klassen *EiffelScanner* und *EiffelParser* wird durch die eben beschriebene Konstruktion der zum Teil recht komplexe Algorithmus zum Scannen bzw. Parsen gekapselt. Nach [Gamma 92] bezeichnet man dieses Muster als "Verfahrensklassen mit einer White-Box-Schnittstelle". Die Kapselung erleichtert die eventuelle Auswechslung des verwendeten Algorithmus des rekursiven Abstiegs gegen einen anderen Algorithmus (z.B. Shift-Reduce Algorithmus).

Im Mittelpunkt der Verfahrensklasse steht eine *Run()*-Methode, die den Algorithmus nach seiner Parametrisierung ablaufen läßt. Da nur zu den wenigsten vom Scanner generierten Token Zusatzinformationen benötigt werden, läuft der EiffelScanner nicht ohne Unterbrechung

durch, sondern wird jeweils aufgefordert, einen weiteren Token zu lesen. Somit entfällt die Notwendigkeit, die Zusatzinformationen längerfristig vorzuhalten. Sie sind immer nur zum aktuellen Token verfügbar. Der in Abschnitt 5.1 beschriebene Tokenstrom vom Scanner zum Parser ist also nur konzeptionell vorhanden.

Im Nachhinein habe ich festgestellt, daß der in der Klassenbibliothek ET++ für die dort integrierte Programmierumgebung ET++PE verwendete (C++-) CodeAnalyser exakt die gleiche Struktur aufweist. (Ich betrachte dies als zusätzliche Bestätigung meines Designs.) Da dort wesentlich weniger Informationen extrahiert werden, ist dort jedoch auf die Trennung von Scanner und Parser verzichtet worden.

Neben dem für Sniff+ spezialisierten Parser habe ich als weitere Verfeinerungen des Eiffel-Parsers eine Ausgabe der Vererbungshierarchie als Eingabe für ein Graphen-Layout Programm (VCG) realisiert bzw. eine Variante mit Textausgabe (für das Debugging des Parsers). (In dieser Art könnte man beispielsweise auch das traditionelle Eiffel-Werkzeug **short** mit meinem Parser realisieren.)

Eine Schwachstelle dieses Design ist, daß die Zahl und Art der zur Konkretisierung vorgesehenen Methoden zu Anfang nur auf (eher vagen) Annahmen beruhen, welche Informationen die abgeleiteten Klassen benötigen werden. Wie bei den meisten Frameworks sind mehrere Redesign-Zyklen nötig, bis sich diese Schnittstelle stabilisiert. Bei Änderungen an den Parametern dieser Methoden sind Änderungen in allen abgeleiteten Klassen notwendig. (Der Änderungsbedarf wird allerdings vom Compiler erkannt, so daß die Änderungen nicht vergessen werden können.) Da die Methoden leer und nicht abstrakt implementiert sind, bereitet hingegen das Hinzufügen neuer ActOn...()-Methoden keine Probleme.

7.2. Error-Recovery

Error-Recovery dient dazu, den Parser nach einem Syntaxfehler wieder mit der syntaktischen Struktur seiner Eingabedaten zu synchronisieren. Bei traditionellen Parsern wird Error-Recovery dazu verwendet, um mehr als einen Syntaxfehler je Parserlauf finden zu können. Je nachdem, wie gut die eingesetzte Error-Recovery-Strategie ist, kann der Parser seine Arbeit (evtl. nach einigen Folgefehlern) fortsetzen.

Bei einem Fuzzy-Parser ist es ebenfalls notwendig, Error-Recovery einzusetzen. Obwohl keine Fehler gesucht werden, kann es leicht vorkommen, daß ein Syntaxfehler erkannt wird. Um trotzdem noch maximal viele Informationen über die Struktur des Quelltextes sammeln zu können, ist die Synchronisation mit den Eingabedaten auch hier sehr wichtig.

Um die Synchronisation mit den Eingabedaten zu erreichen, gibt es (nach [Watson 89]) zwei Möglichkeiten: Entweder werden zusätzliche Symbole in den Eingabestrom hinzugefügt (sog. *Konstruktiv-orientierte Recovery*), oder es werden solange Eingabesymbole übergangen, bis für den Parser erkennbar ist, welches syntaktische Konstrukt im Eingabestrom gerade beginnt (sog. *Panische Recovery*).

Die Strategie des Hinzufügens von Symbolen ist nur sehr eingeschränkt einsetzbar, da hierzu mit hinlänglicher Sicherheit festgestellt werden muß, was der Programmierer eigentlich wollte bzw. welches Symbol er vergessen hat. Sie hat (wenn sie gelingt) den Vorteil, daß keinerlei

Informationen durch den Fehler verloren gehen. In meinem Fuzzy-Parser wird diese Strategie für fehlende Kommas und Semikolons eingesetzt.

Das Ignorieren von Eingabedaten ist potentiell mit Informationsverlust verbunden. Daher sollten so wenig wie möglich Daten ignoriert werden. Die Zustände, in denen der Parser mit (relativer) Sicherheit erkennen kann, welches syntaktische Konstrukt in seiner Eingabe vorliegt, nennt man *Wiederaufsetzpunkte*.

Als Wiederaufsetzpunkt nach Fehlern in Eiffel-Programmen kann spätestens das Dateiende (was auf jeden Fall mit dem Klassenende zusammenfällt) verwendet werden. So geht maximal die Information über den Teil der Klasse zwischen Fehler und Klassenende verloren, sofern dazwischen nicht noch ein anderer Wiederaufsetzpunkt liegt, der den Verlust weiter minimiert.

Innerhalb des Klassenheaders dienen die Schlüselworte INDEXING, DEFERRED, EXPANDED, CLASS, OBSOLETE, INHERIT und CREATION als Wiederaufsetzpunkte; d.h. ein Syntaxfehler in einer dieser Deklarationen führt nicht dazu, daß der Rest der Headers falsch interpretiert wird.

Auf ein Wiederaufsetzen innerhalb der Feature-Deklarationen wurde mangels geeigneter Wiederaufsetzpunkte verzichtet.

7.3. Test des Fuzzy-Parsers

Da der Fuzzy-Parser von Hand erstellt wurde, muß man damit rechnen, daß er mehr Fehler enthält als ein automatisch generierter Parser. Ein großes Problem beim Test des Fuzzy-Parsers war, daß sich seine Struktur nur grob an den einzelnen Produktionen der Grammatik orientiert und oft mehrere Produktionen zusammengefaßt sind. Weiterhin werden z.T. Fehler antizipiert (z.B. Verwechslung von Komma und Semikolon), so daß sich die Struktur des Fuzzy-Parsers noch weiter von der Grammatik entfernt.

Um möglichst ausgiebig zu testen, wurde der Fuzzy-Parser auf die mir zur Verfügung stehenden Quelltexte der Standardbibliothek von SiG Eiffel/S angewendet. Bei den enthaltenen ca. 10.000 Zeilen Quelltext ist kein Fehler aufgetreten.

Weiterhin wurde der Parser getestet mit dem von Neil Wilson[8] entwickelten Eiffel Torture Test (siehe [Wilson 94b], [Wilson 94c] und [Wilson 94d]). Dieser Test bemüht sich, möglichst viele syntaktischen Varianten zu enthalten / zu testen. Es handelt sich allerdings nicht um eine vollständige Validations-Suite, wie sie beispielsweise für COBOL verfügbar ist. Immerhin wird aber eine sehr große Zahl der syntaktischen Varianten abgedeckt.

Durch den Einsatz des Eiffel Torture Test habe ich während der Entwicklung des Parsers eine ganze Reihe von Fehlern gefunden, die mir beim Testen mit eher zufällg ausgewählten Beispielprogrammen sicher entgangen wären.

[8] Neil Wilson ist Mitglied des NICE Eiffel Language Commitee.

8. Technische Integration in Sniff+

Sniff+ Version 1.x ist ausschließlich auf die Bearbeitung von C++ (und C) Programmen ausgelegt. (In der Version 2.0 existiert ein Konzept von verschiedenen Dateitypen, die unterschiedlich behandelt werden. Dieses Konzept ist zum jetzigen Zeitpunkt jedoch noch nicht vollständig implementiert.)

Um zu verstehen, wie es möglich ist, in Sniff+ die Kenntnis über die Eiffel-Syntax zu integrieren, muß man die Architektur von Sniff+ betrachten.

8.1. Die Architektur von Sniff+

Sniff+ besteht aus einem Kern, der alle Werkzeuge mit graphischer Benutzerschnittstelle beinhaltet (Editor, Class Browser, etc.). Ebenso enthält dieser Kern eine zentrale Datenbasis (Symboltabelle) mit allen Informationen über die bearbeiteten Quelltexte. Alle im Kern enthaltenen Werkzeuge beziehen ihr C++-spezifisches Wissen aus dieser zentralen Datenbasis (zumindest sollte dies so sein, ich erwähne später noch einige Probleme).

Die Datenbasis von Sniff+ wird gefüllt und aktualisiert von einem sog. *Information Extractor* (auch als *Sniffserver* bezeichnet). Dieser läuft in einem externen Prozeß und wird bei Änderungen an Quelltexten oder neu hinzugekommenen Quelldateien aufgefordert, diese zu analysieren und die gewonnenen Informationen in der Datenbasis abzulegen. Der Information Extractor enthält als seinen Hauptbestandteil einen Fuzzy-Parser für C und C++.

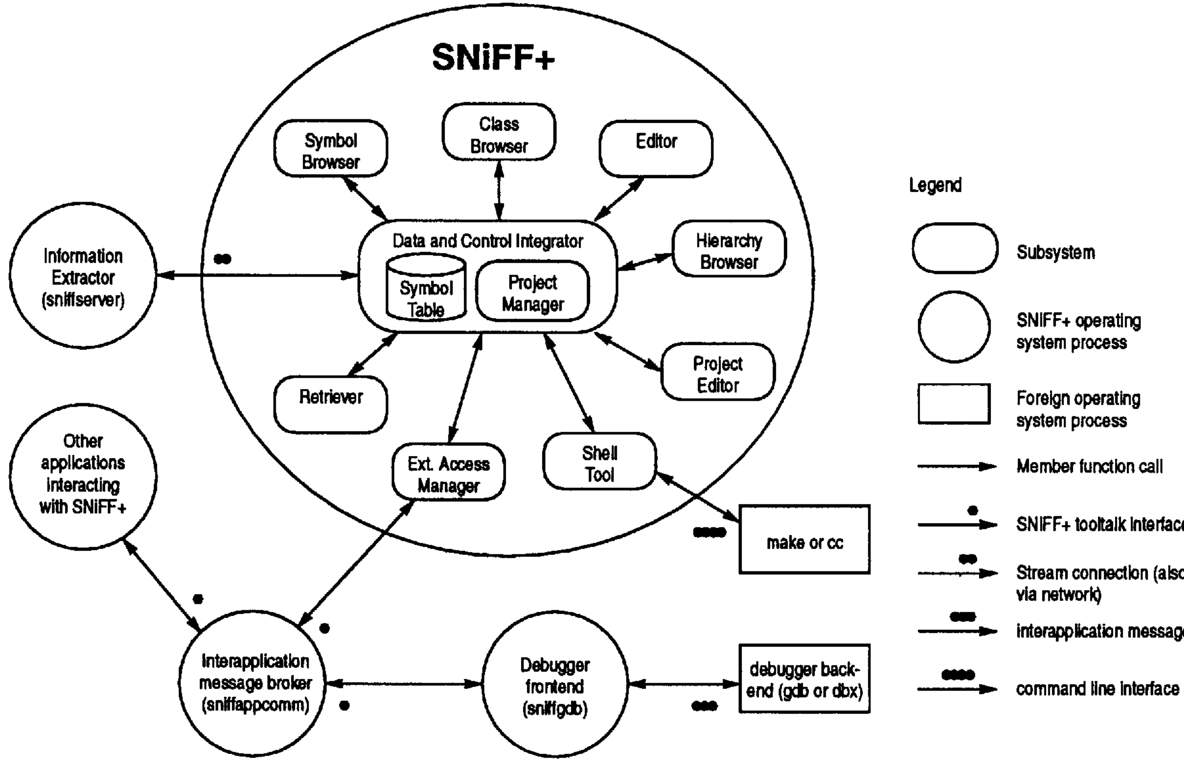


Abb. 14: Die Architektur von Sniff+ (aus [TakeFive 93])

Da mir das Kommunikationsprotokoll zwischen dem Sniff+-Kern und dem Information Extractor (aus [Bischofberger 95][9]) bekannt, ist konnte ich einen Information Extractor implementieren (basierend auf meinem Fuzzy-Parser für Eiffel), der die aus den Eiffel-Quelltexten gewonnenen Informationen auf C++-Eigenschaften abbildet und diese in der Datenbasis von Sniff+ ablegt. An Sniff+ habe ich keine Änderungen vorgenommen (und auch mangels Zugang zum Quelltext nicht vornehmen können).

Der Information-Extractor wird von Sniff+ gestartet, sobald die Symbolinformationen in der Datenbasis aktualisiert werden sollen, falls er zu dem Zeitpunkt noch nicht gestartet ist..

Die Kommunikation zwischen Sniff+ und dem Information Extractor findet auf einer TCP/IP-Socketverbindung statt. (Für eine einführende Erklärung in TCP/IP-Socketverbindungen siehe [ComerNarten 89].) Durch dieses Kommunikationsmedium ist bereits die Möglichkeit geschaffen, den Information Extractor auf einer anderen Maschine im Netz laufen zu lassen. Gerade wenn Sniff+ gestartet bzw. ein neues Projekt angelegt wird, müssen eine ganze Reihe von Dateien analysiert werden, so daß ein ganz erheblicher Rechenaufwand nötig ist, der auf eine andere, leistungsfähigere Maschine verlagert werden kann.

Über die Socketverbindung teilt Sniff+ dem Information Extractor mit, welche Quelldatei dieser jeweils analysieren soll. Bei einer verteilten Ausführung müssen die Dateisysteme der beteiligten Maschinen (z.B. durch eine Vernetzung mit dem Network File System, NFS) einen gemeinsamen Zugriff auf die Quelltexte zulassen.

Der Parser im Information Extractor muß keine eigene Symboltabelle führen, sondern teilt dem Sniff+-Kern via eines binären Protokolls mit, wenn er ein Symbol (z.B. Klassen- oder Methodendeklaration) gefunden hat. Zusätzlich zum Namen des Symbols werden der Symboltyp (Klassenname, Methodenname etc.) sowie die Position des Symbols im Quelltext[10] übertragen. Diese Positionsangabe erlaubt es Sniff+, bestimmte Symbole im Editor farbig zu markieren oder z.B. zwischen Deklaration und Implementation einer Methode hin und her zu wechseln. (Zusätzlich werden je nach Symboltyp einige Zusatzinformationen wie z.B. Zugriffsrechte o.ä. übermittelt.)

Für die Kommunikation ist ein einheitliches, genau festgelegtes Kommunikationsprotokoll nötig, damit der Client (hier: Sniff+) nicht auf Informationen wartet, die der Server (hier: der Information Extractor) nicht sendet.

Vor der Übertragung der Daten werden diese von dem lokal auf der jeweiligen Maschine verwendeten Datenformat (Binärkodierung) in ein vom TCP/IP-Protokoll festgelegtes Netzwerk-Datenformat gewandelt und vom Empfänger wieder in sein lokales Datenformat zurück gewandelt. So wird sichergestellt, daß die Daten nicht (z.B. durch unterschiedliche Byteanordnung (Big Endian / Little Endian)) vom Empfänger anders interpretiert werden als vom Sender und trotzdem jeder Sender bzw. Empfänger nur ein Paar Konvertierungsroutinen benötigt.

---

[9] inoffizielle Entwickler-Dokumemtation

[10] Um eine effiziente Verarbeitung zu ermöglichen wird die Byte-Position relativ zum Dateianfang übertragen.

8.2. Abbilden der Eigenschaften von Eiffel auf C++

Da an Sniff+ keinerlei Änderungen vorgenommen werden (können), müssen die für das Browsen wichtigen Eigenschaften von Eiffel auf Eigenschaften von C++ abgebildet werden. Da beide Sprachen ähnliche Konzepte (trotz unterschiedlicher Terminologie) haben, ist dies größtenteils möglich. Einige spezielle Eigenschaften von Eiffel sind leider nicht auf C++ abbildbar und können deshalb vom Browser nicht unterstützt werden.

Bezeichnung in Eiffel	Bezeichnung in C++	Bezeichnung in Sniff+
Klasse	Klasse	classes
Routine	Methode	methods
Attribut	Instanzvariable	instance variable
Elternteil	Basisklasse	base class
Erbe	abgeleitete Klasse	derived class
aufgeschoben	abstrakt	abstract
dynamisches Binden (immer verwendet)	virtuelle Methode	virtual method

Abb. 15: Eigenschaften, die sich direkt abbilden lassen

Die Vererbung ist ohne Probleme abzubilden; Probleme treten aber bei der Redefinition bzw. dem Überladen von Merkmalen auf. Mehr dazu im nächsten Abschnitt.

Bezeichnung in Eiffel	Bezeichnung in C++	Bezeichnung in Sniff+
Creation-Routine	Konstruktor	(nicht verwendet)
Zugriffsrechte auf Merkmale	public, protected, private	public, protected, private

Abb. 16: Eigenschaften, die sich teilweise abbilden lassen

In Eiffel ist es möglich, den Namen der Routine (oder mehrerer alternativer Routinen) frei zu wählen, die bei der Initialisierung eines Objektes aufgerufen wird. Dies ist die sog. CREATION-Methode. In C++ ist der Name des Konstruktors auf den Klassennamen festgelegt, kann aber unterschiedliche Ausprägungen mit anderen Parametern haben. Da Konstruktoren von Sniff+ nicht weiter beachtet werden, erübrigt sich der Versuch einer Abbildung.

Die Zugriffsrechte auf Merkmale lassen sich nur indirekt abbilden, da es in Eiffel die Möglichkeit gibt, bestimmte Merkmale nur für eine spezielle Klasse freizugeben.

Die Friend-Methoden in C++ können dies nicht abbilden, da ihnen immer der Zugriff auf alle Merkmale einer Klasse gewährt wird.

Ich habe mir damit geholfen, daß ich das Eiffel-Zugriffsrecht {ANY) auf das C++-PUBLIC und das Eiffel {NONE} auf das C++-PRIVATE und alle differenzierteren Zugriffsrechte auf das C++-PROTECTED abgebildet habe. Dies entspricht nicht exakt der Bedeutung in C++,

bietet aber dem Eiffel-Programmierer die bestmögliche Abstufung unter diesen Rahmenbedingungen.

Es gibt aber auch eine ganze Reihe von Eigenschaften von Eiffel, die sich überhaupt nicht in C++ abbilden lassen.

Bezeichnung in Eiffel
Indexing
Vor- und Nach-Bedingungen
Invarianten
expandierte (Expanded) Klassen
veraltete (Obsolete) Klassen, Methoden
einmalige (Once) Ausführung von Methoden
Umbenennung (Rename) von geerbten Merkmalen
Aufschieben (Undefine) von geerbten Merkmalen

Abb. 17: Eigenschaften von Eiffel, die kein C++-Gegenstück haben

Bei einigen Eigenschaften (z.B. expandiert, Obsolete, Once) würde eine farbliche Markierung o.ä. wie bei aufgeschobenen Merkmalen ausreichen. Bei anderen, z.B. Indexing, wäre eine umfassendere Unterstützung durch den Browser wünschenswert.

In [EllisStrou 90] findet sich ein Vorschlag für das ANSI-Normungs Komitee, C++ um ein syntaktisches Konstrukt zur Umbenennung geerbter Methoden zu erweitern. Soweit mir bekannt ist, ist dieser Vorschlag bisher jedoch nicht in den Sprachstandard übernommen worden. Von Sniff+ ist hierfür ebenfalls bisher keine Unterstützung vorhanden.

Das Aufschieben von geerbten Merkmalen ist in Eiffel zwar zugelassen, widerspricht aber guter Programmierpraxis, da es die Subtyp-Beziehung der Klassen stört.[11] Von dieser Möglichkeit in Eiffel sollte ein Programmierer daher keinen Gebrauch machen. Trotzdem wäre es sehr wichtig, daß der Browser deutliche Warn-Hinweise gibt, falls geerbte Merkmale aufgeschoben wurden.

8.3. Probleme bei der Integration

Bei der Integration des neuen Information Extractors sind einige Stellen sichtbar geworden, an denen Sniff+ implizite Annahmen über die zu bearbeitende Sprache (C++) macht. Die Struktur der Datenbasis ist z.T. stark auf C++ zugeschnitten. Daher kann ein neuer Information Extractor nur begrenzt die Eigenschaften einer neuen Sprache einbringen.

Zum einen betrifft dies die Reihenfolge, in der Information Extractor mitteilen darf, welche Informationen er gefunden hat.

[11] Bei Verwendung von Polymorphie ist dann erst zur Laufzeit feststellbar, ob ein Objekt überhaupt ein bestimmtes Merkmal besitzt.

Ein Beispiel für implizite Annahmen über die Reihenfolge von Informationen: Da C++ kein direktes Ausdrucksmittel hat, um eine aufgeschobene Klasse zu markieren[12], wird dies in ET++ durch Makros nachgebildet und so auch von Sniff+ unterstützt. Da diese Makros aber nur außerhalb der Klassendefinitionen stehen dürfen, macht Sniff+ die Annahme, daß die Information über eine gefundene aufgeschobene Klasse nur außerhalb einer Klassendefinition gemeldet wird, und hängt sich andernfalls auf.

Zum anderen ist im Class Browser fest verdrahtet, wie das Redefinieren von Methoden (in C++) funktioniert. So werden Methoden mit gleichem Namen und gleichen Parametern von der neuen Definition überladen, und alle anderen werden additiv hinzugefügt. In Eiffel spielen die Parameter dagegen keine Rolle. (Um dieses Problem zu umgehen, meldet mein Information Extractor zu keiner Routine die Parameter.)

Die Möglichkeiten, in Eiffel geerbte Merkmale umzubenennen oder aufzuschieben, lassen sich überhaupt nicht abbilden.

---

[12] Die Betonung liegt auf "direkt": Es können sehr wohl Methoden als aufgeschoben (in C++ Terminologie "pure virtual") markiert werden. Klassen, die aufgeschobene Methoden besitzen, sind auch in C++ aufgeschoben. Da es aber auch möglich ist, eine Klasse durch das Weglassen einer Methoden-Implementation aufzuschieben, ist es der Klassendeklaration nicht immer anzusehen, ob die Klasse aufgeschoben ist oder nicht.

9. Umgang mit Eiffel in Sniff+

Durch die Integration des neuen Information Extractors können nun Eiffel-Quelltexte mit Sniff+ bearbeitet werden. Alle in Sniff+ integrierten Werkzeuge können verwendet werden.

Da auch ein externer Eiffel-Compiler integriert werden kann, stellt Sniff+ eine vollständige Eiffel-Entwicklungsumgebung dar. Lediglich die Bezeichnungen in den Menüs verwenden noch die C++-Terminologie. Hier ist eine gewisse Umgewöhnung für den Eiffel-Programmierer notwendig, sofern er nicht ohnehin schon mit dieser Terminologie vertraut ist.

9.1. Der Editor

Im Editor von Sniff+ ist ein hypertext-artiges Navigieren zwischen den einzelnen Merkmalen möglich. Durch Anklicken des Methodennamens in der Scroll-Liste am rechten Rand gelangt man direkt zur Implementation.

Die Namen von Klassen und Routinen werden farblich oder durch einen speziellen Font im Quelltext hervorgehoben. Die Namen von Attributen werden bei ihrer Deklaration hervorgehoben.

Weiterhin wird angezeigt, ob bereits Änderungen am Quelltext vorgenommen wurden.

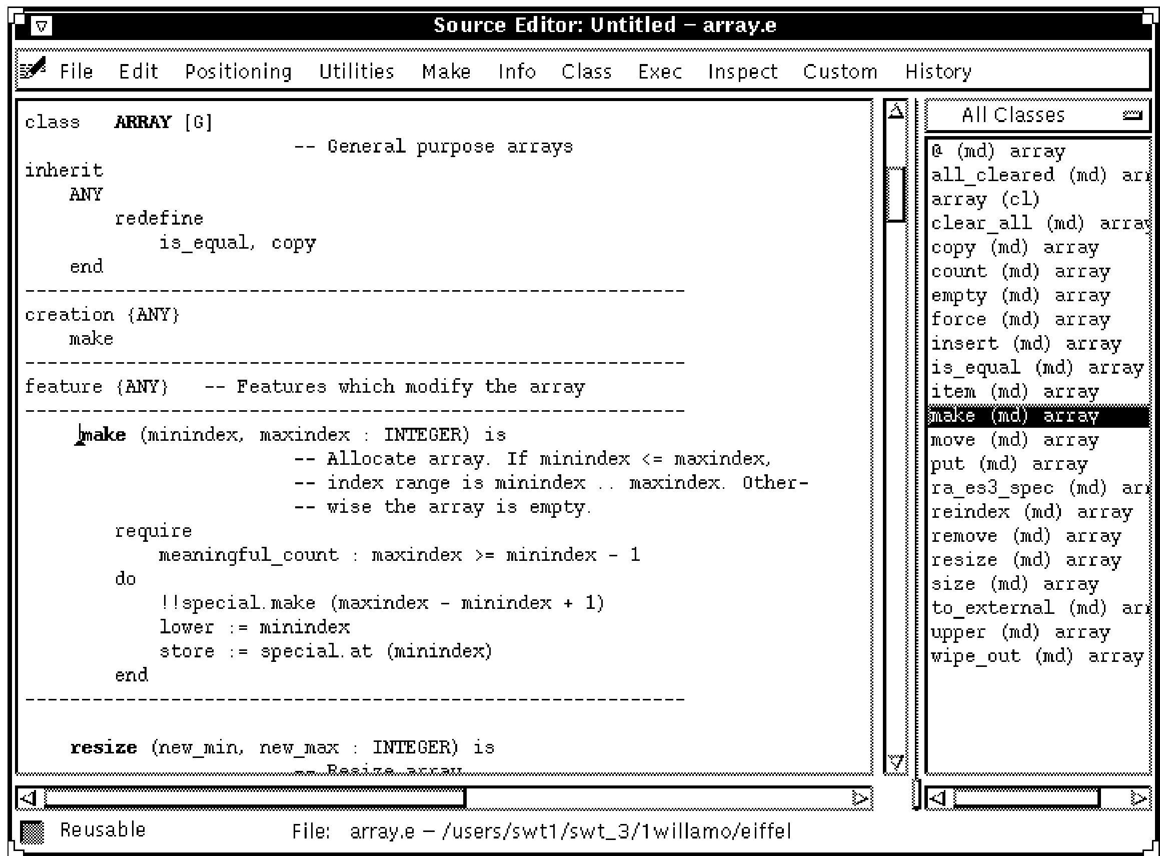


Abb. 18: Der Editor mit Eiffel-Quelltext

Bei Änderungen an Eiffel-Quelltexten wird die Darstellung in den anderen Werkzeugen von Sniff+ automatisch aktualisiert, sobald die Änderungen gespeichert wurden, ohne daß ein zeitaufwendiges Neu-Kompilieren des Projektes nötig ist.

9.2. Der Hierarchie-Browser

Im Hierarchie-Browser wird die Vererbungstruktur des Projekts graphisch dargestellt. Die graphische Darstellung macht auch die Verwendung von Mehrfachvererbung deutlich, wozu die rein textuelle Darstellung z.B. von ISE Eiffel 3 nicht in der Lage war.

Die Ausgabe kann nach verschiedenen Kriterien begrenzt werden, so daß z.B. nur die Vererbungshierarchie einer bestimmten Klasse dargestellt wird. Zusätzlich kann die Darstellung auch auf ein bzw. mehrere Unterprojekte beschränkt werden. Auf diese Weise können z.B. Bibliotheken von der Darstellung ausgenommen und nur der von ihnen abgeleitete anwendungsspezifische Teil der Vererbungshierarchie dargestellt werden.

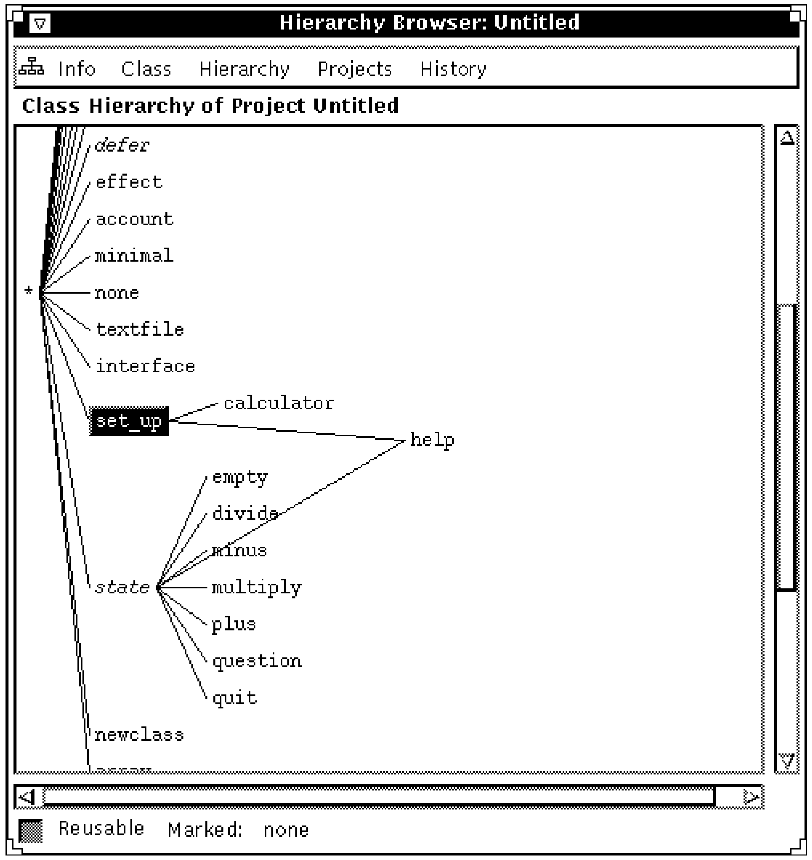


Abb. 19: Der Hierarchie-Browser

Aufgeschobene Klassen (deferred) sind durch kursive Schrift gekennzeichnet. Sie werden oft zur Definition von Protokollen verwendet und sind so leicht zu erkennen (wie beispielsweise die Klasse "state" in der Abbildung).

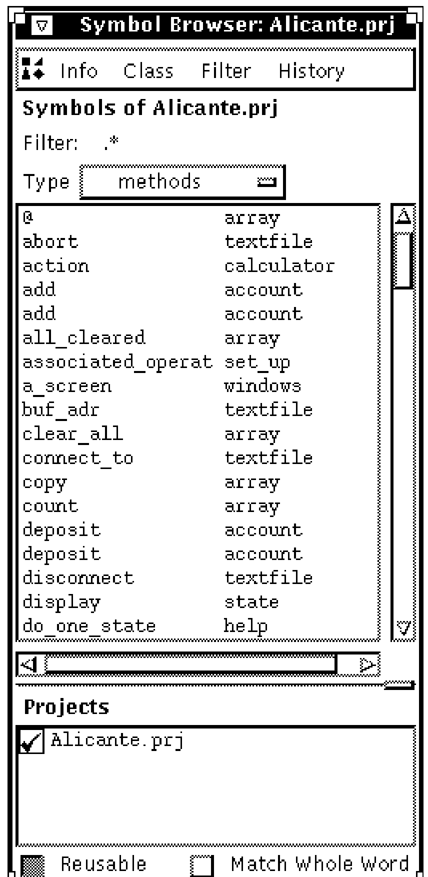
Durch einen Doppelklick auf einen Klassennamen werden der Editor gestartet und die betreffende Klasse geladen. Über das Menü können auch andere Werkzeuge, z.B. der Class Browser, auf die markierte Klasse angewendet werden. Dies ist ein Beispiel, das die enge Integration der einzelnen Werkzeuge in Sniff+ verdeutlicht.

9.3. Der Symbol-Browser

Der Symbol-Browser dient zur Suche nach Namen von Klassen, Routinen oder Attributen. Mit dem Type-Menü wird festgelegt, welche Namen angezeigt werden sollen (nur Klassennamen, nur Routinen oder nur Attribute). Zur weiteren Einschränkung kann ein regulärer Ausdruck angegeben werden, um die Anzahl der angezeigten Namen weiter zu reduzieren (z.B. alle die mit Window... beginnen).

Auf diese Weise ist es beispielsweise leicht, alle Klassen zu finden, die eine Methode mit einem bestimmten Namen definieren.

Auch hier kann man durch Doppelklick direkt in den Editor gelangen. Der Cursor wird sogar direkt auf die Deklaration des entsprechenden Namens gesetzt.



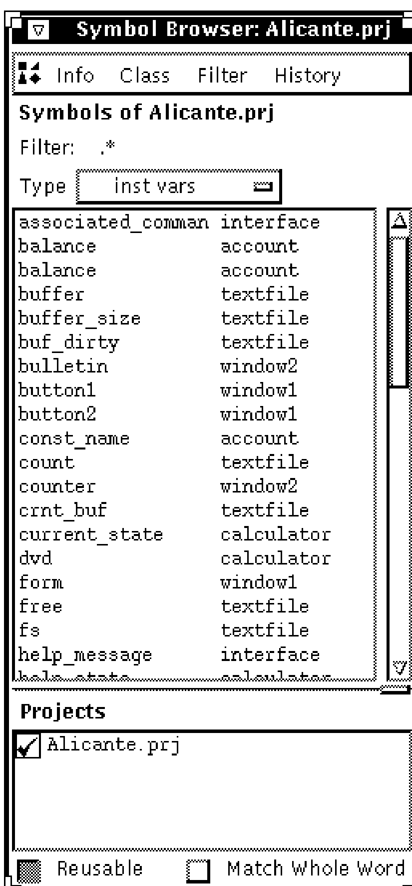


Abb. 20: Zwei Exemplare des Symbol-Browsers

Wie alle anderen Werkzeuge von Sniff+ auch, kann man auch mehre Exemplare des Symbol-Browser aufrufen, so daß man die Namen aus verschiedenen Kategorien auch nebeneinander betrachten kann.

Die Darstellung in einem geöffneten Fenster wird aktualisiert, sobald ein neuer Name in einem Editorfenster deklariert (und gespeichert) worden ist.

Die Projekthierarchie wird im unteren Teil des Fensters dargestellt, und die Anzeige kann durch Anklicken auf bestimmte Unterprojekte begrenzt werden.

9.4. Der Class-Browser

Der Class-Browser stellt alle Merkmale (Attribute oder Routinen) einer Klasse dar. Wahlweise können nur die in dieser Klasse implementierten Merkmale oder auch die geerbten Merkmale dargestellt werden (alle oder nur von selektierten Oberklassen).

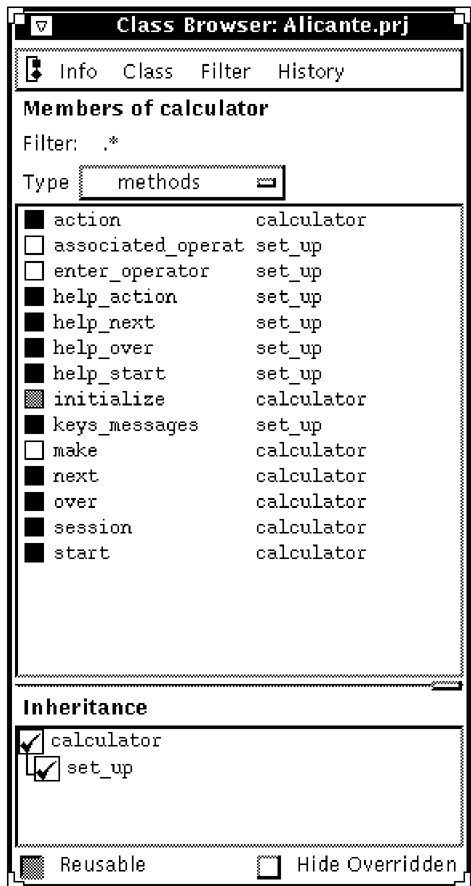
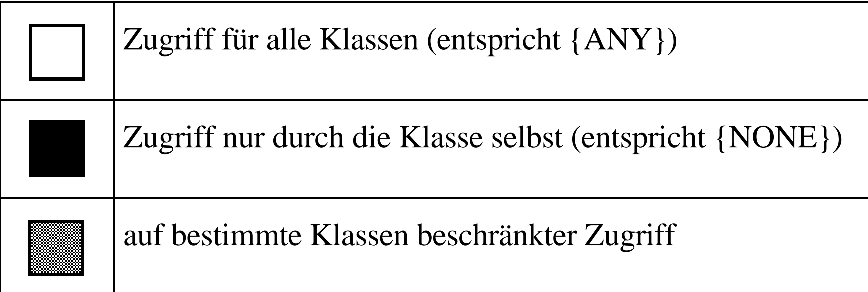


Abb. 21: Der Class-Browser

Im unteren Teil des Fensters ist die Vererbungshierarchie der Klasse dargestellt. Grundlegende Informationen über die Vererbungsstruktur sind so auch ohne Verwendung des Hierarchie-Browser sichtbar.

Aufgeschobene (deferred) Merkmale sind durch kursive Schrift markiert. Zu jedem Merkmal wird auch die (Ober-)Klasse angezeigt, in der es implementiert ist.

Die Zugriffsrechte auf die Merkmale sind durch den Kasten vor dem Namen symbolisiert.



	Zugriff für alle Klassen (entspricht {ANY})
	Zugriff nur durch die Klasse selbst (entspricht {NONE})
	auf bestimmte Klassen beschränkter Zugriff

Abb. 22: Zugriffsrechte

Die im Class-Browser dargestellten Merkmale spiegeln jedoch nicht hundertprozentig die Schnittstelle der Eiffel-Klasse wider, da einige Eigenschaften von Eiffel bezüglich der Vererbung nicht auf C++ abgebildet werden können (vergl. Implementationsprobleme bei der Umbenennung und dem Aufschieben von geerbten Merkmalen).

9.5. Der Projekt-Editor

Ein compilerunabhängiges Projekt-Konzept zur Strukturierung der bearbeiteten Eiffel-Klassen steht ebenfalls zur Verfügung. Die Zugehörigkeit der Klassen zu Projekten wird mit dem Projekt-Editor verwaltet.

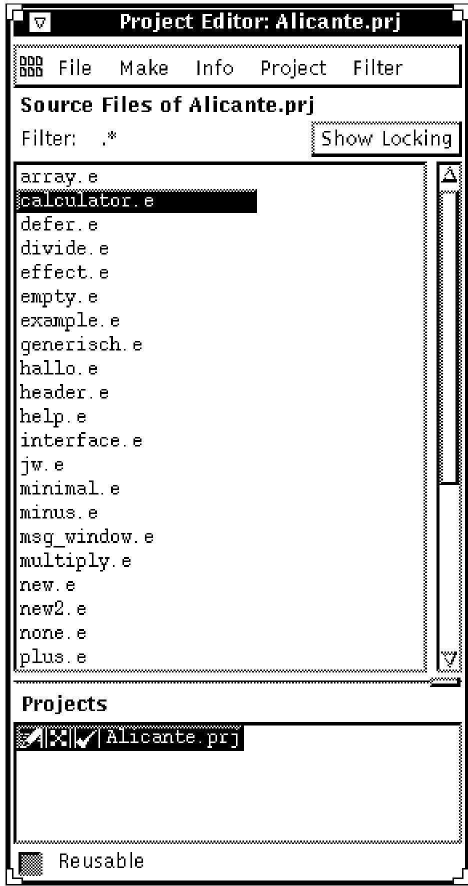


Abb. 23: Der Projekt-Editor

Jedes Projekt kann Unterprojekte enthalten, so daß nicht jeder Benutzer z.B. für die Standard-Eiffel-Bibliotheken ein eigenes Projekt anlegen muß. Da die Symbolinformationen für die Projekte gespeichert werden, steigert dieses Vorgehen auch die Effizienz, da für die keinen Veränderungen unterworfenen Bibliotheken sofort alle Symbole zur Verfügung stehen.

Die von Meyer vorgeschlagenen Strukturierungskonzepte (vergl. Kapitel 2.2) lassen sich auf dieses Projekt-Konzept abbilden, so daß das Hauptprojekt dem Eiffel-*System* entspricht und die *Cluster* durch Unterprojekte dargestellt werden.

Es ist auch möglich, mit dem Projekt-Editor neue Dateien zu erzeugen, für die Sniff+ dann den Rahmen einer Eiffel-Klasse vorgibt, der dann nur noch ausgefüllt werden muß.

Ebenfalls integriert ist der Anschluß einer Versionsverwaltung für die im Projekt enthaltenen Dateien. Dabei können wahlweise die Versionsverwaltungssysteme RCS oder SCCS verwendet werden.

10. Ausblick

Eine ganze Reihe von Eigenschaften von Eiffel konnten nicht auf C++ abgebildet werden und werden daher nicht von Sniff+ unterstützt. Im Zusammenhang damit wurde deutlich, wieviel implizite Annahmen über die Sprache C++ in Sniff+ enthalten sind. Es wäre wünschenswert, wenn Sniff+ derart überarbeitet würde, daß alle Eigenschaften von Eiffel unterstützt werden können.

Eine dynamische Erweiterbarkeit der Symboltabelle um die Eigenschaften einer anderen Sprache könnte einige dieser Probleme beheben. Zusätzlich wäre es aber auch dringend notwendig, die Menüs von Sniff+ an die gerade bearbeitete Sprache anzupassen. Bei relativ ähnlichen Sprachen wie C++ und Eiffel ist es gerade noch vertretbar, mit den Begriffen der jeweils anderen zu arbeiten (sofern es eine direkte Abbildung zwischen den beiden gibt).

Es gibt jedoch noch viel tiefer sitzende Annahmen über die Art, wie gleichnamige Merkmale vererbt werden bzw. ob es noch andere Einflüsse (z.B. Renaming, Redefining, Undefining) bei der Vererbung von Merkmalen gibt. Hier ist es sicher schwer, eine allgemeine Methode zu definieren. Constraints o.ä. für jede verwendete Sprache wären eine Möglichkeit. Auch die Zugriffsrechte auf einzelne Merkmale sind nicht direkt abbildbar, und flexiblere Darstellungen als die von C++ wären wünschenswert.

An der Schnittstelle von Sniff+ wird von TakeFive bereits gearbeitet, um Sniff+ noch besser für die Integration anderer Programmiersprachen zu öffnen. Die Entwicklung hier sollte genau verfolgt werden.

Die Erweiterung der darstellbaren Eigenschaften könnte dann auch auf Designmuster ausgedehnt werden. Wie dies geschehen könnte ist ein noch ungeklärtes Problem.

Der hier entwickelte Browser betrachtet nur die statische Struktur des Eiffel-Systems, also das, was sich direkt aus dem Quelltext ableiten läßt. In viele Fällen kann es jedoch auch hilfreich sein, sich die dynamische Struktur der Objekte eines Systems zur Laufzeit anzuschauen. In [Gamma 92] und [GamWeiMar 89] wird dargestellt, wie man eine Programmierumgebung in ein Applikation-Framework integrieren kann, so daß es auch zur Laufzeit zur Verfügung steht und die dynamische Struktur darstellen kann. Eine derartige Erweiterung wäre sicher auch für Eiffel sehr wertvoll.

11. Anhang A Literatur

[AhoSethiUllmann 88] Aho, A., Sethi, R., Ullman, J., Compilerbau, Teil 1, Addison-Wesley, 1988

[Bischofberger 92] Bischofberger, W. R., Sniff - A Pragmatic Approach to a C++ Programming Environment, Proceedings of the USENIX C++ Conference, Portland, Oregon, August 1992

[Bischofberger 94a] Bischofberger, W. R., Kofler, T., Schäffer B: Object-Oriented Programming Environments: Requirements and Approaches. In Software - Concepts and Tools, Vol. 15, No. 2, Springer-Verlag, 1994

[Bischofberger 94b] Bischofberger, W. R., Frei, H. P. (Editors): Computer Science Research at UBILAB: strategy and projects, Proceedings of the UBILAB Conference '94, Zürich, Univ-Verl. Konstanz, 1994

[Bischofberger 95] Bischofberger, W. R., EMail vom 27. April 1995

[Blach 92] Blach, R., Ein LL(1)-Parser für Eiffel. In [Hoffmann 92]

[ComerNarten 89] Comer, Douglas E., Narten, Thomas, TCP/IP. In [KochanWood 90]

[EllisStrou 90] Ellis, M. A., Stroustrup, B., The Anotated C++ Reference Manual, Addison-Wesely, 1990

[EngelsSchäfer 89] Engels, G., Schäfer, W., Programmentwicklungsumgebungen, Teubner, 1989

[Eon 94] Eon/Eiffel, Installation Guide, 1994

[Gamma 92] Gamma, E., Objektorientierte Software-Entwicklung am Beispiel von ET++, Design-Muster, Klassenbibliothek, Werkzeuge, Springer-Verlag, 1992

[Gamma 95] Gamma, E. et al., Design Patterns, Addison-Wesley, 1995

[GamWeiMar 89] Gamma, E., Weinand, A., Marty, R., Integration of a Programming Environment into ET++, A Case Study. In ECOOP 89, Proceedings of the Third European Conference on Object-Oriented Programming (Nottingham, UK), S. Cook (ed), Cambridge University Press, Cambridge 1989, pp. 283-297

[Grass 90] Grass, J. E., Chen, Y. F., The C++ Information Abstractor. USENIX C++ Conference Proceedings, San Francisco, CA, April 1990.

[Groeber 92a] Groeber, B., Langmack, O., A Compiler Front-End for Eiffel-3, Report B-92-25, Institut für Informatik, Freie Universität Berlin, Dezember 1992

[Groeber 92b] Groeber, B., Langmack, O., A Compiler Front-End for Eiffel 3, Eiffel Outlook, Vol 2, #5, July 1992

[Grosch 92] Grosch, J., Compiler Construction Tool Box, Dokumentation zu Cocktail, GMD, 1992

[Hoffmann 92] Hoffmann, H.-J.,, editor. Eiffel - Fachtagung des German Chapter of the ACM, Vol. 35 of Berichte des German Chapter of the ACM, Teubner, 1992

[KilGryZül 93] Kilbert, K., Gryczan, G., Züllighoven, H., Anwendungsorientierte Softwareentwicklung, Vieweg, 1993

[KochanWood 90] Kochan, Stephen G., Wood, Patrick H., editors, Unix Networking, Hayden Books, 1989

[Leonard 94] Leonard, Ian, Eon/Eiffel - A New Eiffel Compiler, Eiffel Outlook, November 1994

[LevMasBro 92] Levine, J., Mason, T., Brown, D., lex & yacc, O'Reilly, 1992

[McConnell 93] McConnell, Steve C., Code Complete, Microsoft Press 1993

[Meyer 88] Meyer, B., Object-oriented Software Construction, Prentice Hall, 1988

[Meyer 90] Meyer, B., Lessons from the design of the Eiffel libraries, in: Communications of the ACM, Vol. 33, No. 9, September 1990

[Meyer 91] Meyer, B., Eiffel: The Language, Prentice-Hall, 1991

[Meyer 94] Meyer, B., ISE Eiffel 3, The Environment, ISE Technical Report TR-EI-39/IE, Version 3.2.2, Januar 1994

[Microsoft 93] Visual Workbench User's Guide, Microsoft Corporation, 1993

[Ravenhagen 87] Raghavan, R., Ramakrishnan, N., and Strater, S., A C++ Browser. In USENIX Proceedings and Additional Papers C++ Workshop, USENIX Assoc., 1987

[Shipman 94] Shipman, Anthony, Evaluation Report on ISE Eiffel, comp.lang.eiffel, 15. Aug. 1994

[SiG 92] The Eiffel/S Compiler and Runtime System, SiG Computer GmbH, 1992

[Stroustrup 92] Stroustrup, B., Die C++ Programmiersprache, 2. Auflage, Addison-Wesely, 1992

[Strunk 92] Strunk, W., Entwurf und prototypische Implementierung eines Klassenbrowsers für Eiffel, in [Hoffmann 92].

[Switzer 93] Switzer, R., Eiffel: an introduction, Prentice Hall, 1993

[TakeFive 93] Sniff+ Version 1.1, Reference Guide, Product-Number: SNiFF-REF-001, TakeFive Software GesmbH, 1993

[TakeFive 95] Sniff+, Werbebroschüre, TakeFive Software GesmbH, 1995

[Traub 94] Traub, H.-P., ConLib Benutzerdokumentation, Anhang zur Diplomarbeit "Objektorientierte Behälterklassen - Konzepte, Entwurf und Implementation", Universität Hamburg, 1994

[Watson 89] Watson, Des, High-level languages and their compilers, Addison-Wesley, 1989

[Wilson 94a] Wilson, Neil, Eiffel The Language, Second Printing, Eiffel Outlook, July 1994

[Wilson 94b] Wilson, Neil, Putting Eiffel Compilers to the Test, Eiffel Outlook, July 1994

[Wilson 94c] Wilson, Neil, Putting Eiffel Compilers to the Test (Part 2), Eiffel Outlook, September 1994

[Wilson 94d] Wilson, Neil, Putting Eiffel Compilers to the Test (Part 3), Eiffel Outlook, November 1994

[Wirth 77] Wirth, Niklaus, Compilerbau, Teubner, 1977

12. Anhang B Installationsbeschreibung

Auf der beigefügten Diskette befindet sich der von mir erstellte Information Extractor (Sniffserver) für SunOS 4.x. Da Sniff+ 1.1 nur mit einem einzigen Information Extractor zusammenarbeiten kann, muß eine Kopie der Sniff+ 1.1 Installation (im Verzeichnis /local/swt1/Sniff+) in einem neuen Verzeichnis angelegt werden.[13] Die Environment-Variable $SNIFF_DIR muß vom Benutzer auf dieses neue Verzeichnis eingestellt werden.

Nun kann die Datei "sniffserver" im Verzeichnis $SNIFF_DIR/bin/sunos durch die von mir erstellte Version ersetzt werden. Die bin-Verzeichnisse für andere Architekturen sollten gelöscht werden, da diese noch den C++ Information Extractor enthalten.

Weiterhin sollte im Verzeichnis $SNIFF_DIR/bin die beigefügte Datei eiffelsniff abgelegt werden.

Wenn Sniff+ das erstemal mit dem Befehl "eiffelsniff &" gestartet wird, muß noch in den "Preferences" die Dateiendung für Eiffel-Quelltexte im Feld "Source Suffixes" in "e" geändert werden. Das Feld "Include Suffixes" sollte leer sein.

Als letztes sollte im Verzeichnis $SNIFF_DIR/config eine Datei mit dem Namen "template.e" angelegt werden, die ein Gerüst für neue Eiffel-Dateien enthält. Sie wird immer dann verwendet, wenn Sniff+ eine Eiffel-Datei neu anlegt.

```
-- generated by Sniff+
class NEW

end
```

Abb. 24: Beispiel für template.e

[13] Statt einer Kopie der gesamten Installation können auch nur die Verzeichnisse angelegt und für jede Datei ein symbolischer Link auf die entsprechende Datei der Original-Installation angelegt werden. Lediglich die Lizensdatei license.dat muß als echte Kopie vorliegen.