

Berichte des German Chapter of the ACM

Im Auftrag des German Chapter
of the ACM herausgegeben durch den Vorstand

Chairman
Hans-Joachim Habermann, Neuer Wall 32, 2000 Hamburg 36

Vice Chairman
Prof. Dr. Gerhard Barth, Erwin-Schrödinger-Straße 57, 6750 Kaiserslautern

Treasurer
Eckhard Jaus, Gernsenweg 12, 7250 Leonberg

Secretary
Prof. Dr. Peter Gorny, Ammerländer Heerstraße 114–118, 2900 Oldenburg

Band 35

Die Reihe dient der schnellen und weiten Verbreitung neuer, für die Praxis relevanter Entwicklungen in der Informatik. Hierbei sollen alle Gebiete der Informatik sowie ihre Anwendungen angemessen berücksichtigt werden.

Bevorzugt werden in dieser Reihe die Tagungsberichte der vom German Chapter allein oder gemeinsam mit anderen Gesellschaften veranstalteten Tagungen veröffentlicht. Darüber hinaus sollen wichtige Forschungs- und Übersichtsberichte in dieser Reihe aufgenommen werden.

Aktualität und Qualität sind entscheidend für die Veröffentlichung. Die Herausgeber nehmen Manuskripte in deutscher und englischer Sprache entgegen.

Eiffel

Fachtagung des German Chapter
of the ACM e. V. in Zusammenarbeit mit der
Gesellschaft für Informatik e. V., FA 2.1,
am 25. und 26. Mai 1992 in Darmstadt

Herausgegeben von

Prof. Dr. Hans-Jürgen Hoffmann
Technische Hochschule Darmstadt



B. G. Teubner Stuttgart 1992

Vorwort

Die Programmiersprache Eiffel, 1988 von Bertrand Meyer in seinem Buch *Object-oriented Software Construction* [MEY 88] der breiten Fachwelt vorgestellt, stößt auf zunehmendes Interesse.

Die Fachtagung, die am 25. und 26. Mai 1992 in Darmstadt veranstaltet wird, soll Gelegenheit bieten, Interessenten an Eiffel zusammenzuführen, aktuelle Themen dazu zu behandeln und eine breite Diskussion darüber zu ermöglichen. Die Veranstaltung ist, so hoffen die Veranstalter, als eine Initialzündung zum wissenschaftlichen Betrachten dieser objektorientierten Programmiersprache und einem Auseinandersetzen mit Erreichtem und möglichen Weiterentwicklungen zu sehen.

Mitten in die Planung und das Zusammenstellen des Tagungsprogramms fiel die Ankündigung von Eiffel 3 [MEY 91]. Beitragende haben versucht, ihren Aufsatz an diese neue Sprachversion anzupassen. In der kurzen Zeitspanne bis zur Drucklegung mußte diese Anpassung zum Teil rudimentär bleiben; man möge hier Nachsicht üben.

Womit könnte ich mein Interesse an Eiffel begründen?

Bei den Software-Ingenieuren und Programmierern gab es von Beginn an die Diskussion über die richtige Methodik des Programmierens und über das richtige Mittel, Realisierungsideen für eine Anwendungsaufgabe in einer Programmiersprache auszudrücken; seit Aufkommen der persönlichen Rechner am Arbeitsplatz spielt das unterstützende Angebot einer dazu passenden Programmierumgebung zunehmend eine wichtige Rolle.

Objektorientierte Ansätze, so alt wie sie sind, als Grundlage der Aufgabenanalyse, des Umsetzens in Entwurfskonzepte, der Programmentwicklung, der Qualitätskontrolle und des Umgangs mit schließlich entstandenen Programmen sind Stand der Methodendiskussion. Mit Eiffel besteht m.E. die Chance, Datenkapselung und Datenabstraktion als bewährte Ansätze mit qualitätssichernden Maßnahmen, die von den Vorstellungen über Programmverifikation und der auch im täglichen Leben gewohnten vertraglichen Bindung von kooperierenden Partnern geprägt sind, in Verbindung zu bringen und damit zu aller Nutzen einen weiteren Schritt im Übergang der Software-Entwicklungsmethodik von Kunst über Wissenschaft zu Technik zu vollziehen. Ich sehe hier eine Verpflichtung.

Mittel zum Zweck der Programmrealisierung ist bei all dem immer die Programmiersprache. Hier steht Eiffel in Konkurrenz zu anderen objektorientierten Programmiersprachen. Ist es da wie bei der Muttersprache, die uns im Kindesalter zukommt; sie beherrschen wir unser ganzes Leben lang am besten. Nur an wenigen Orten beginnt die Programmierausbildung mit Eiffel. So geht es um sprachliche Nähe zu typischen Ausbildungssprachen. Eiffel steht da besser da als das methodisch vergleichbare Smalltalk, ohne Zweifel. Wie ist es im Vergleich mit C++? Eleganz, Beitrag zur Qualitätssicherung, Ausgrenzung von überkommenem Ballast sind einige Plus-Punkte für Eiffel; Effizienz und Akzeptanz müssen noch erreicht werden. Ist es wert, sich dafür einzusetzen, insbesondere als Professor an einer Hochschule?

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Eiffel:

Fachtagung des German Chapter of the ACM e.V. in Zusammenarbeit mit Gesellschaft für Informatik e.V., FA 2.1, am 25. und 26. Mai 1992 in Darmstadt / hrsg. von Hans-Jürgen Hoffmann. – Stuttgart : Teubner, 1992

ISBN 3-519-02676-7

NE: Hoffmann, Hans-Jürgen [Hrsg.]; Association for Computing Machinery / German Chapter

Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

© B. G. Teubner Stuttgart 1992

Printed in Germany

Gesamtherstellung: Präzis-Druck GmbH, Karlsruhe

Einband: P.P.K.S-Konzepte, Tabca Koch, Ostfildern/Stgt.

Interaktives Programmieren, heute in zahlreichen und vielfältigen Programmierumgebungen realisiert, stößt auf mein Interesse. Die Nähe von Eiffel zu anderen, verbreiteten Sprachen erlaubt die Integration auch von Eiffel in eine moderne Arbeitsumgebung des Programmierers. Hier ist eine Vereinheitlichung unter einem der offenen Benutzungsoberflächensysteme anzustreben, die gleichzeitigen Umgang mit verschiedenen Systemen erleichtert. Eiffel bietet sich als eine Basis dazu an.

Tagungsbeiträge können einen Arbeitsbereich nicht vollständig abdecken, in dem Innovation und Umsetzen im Detail so gefordert sind wie in dem Bereich der Programmiermethodik, der Programmiersprachen und der Programmierumgebungen. Einiges sollte ein Teilnehmer der Tagung bzw. ein Leser dieses Buchs doch mit nach Hause nehmen bzw. beim genauen Studium herausfinden können. Ich freue mich, die Gelegenheit gehabt zu haben, die Tagung vorzubereiten und den Tagungsband herauszugeben.

Man erlaube mir aber noch eine Bemerkung:

Ein Herausgeber eines Tagungsbands, der Beiträge in der Form enthält, wie sie Autoren abgegeben haben, steht heutzutage immer vor der Frage, ob man Tipp-, Rechtschreib- und Stilfehler in den Texten durchgehen läßt oder nicht. Soll man sich als Lektor betätigen oder nicht? Kritischen Lesern sei weitergegeben, daß wir Autoren auf derartige Mängel nach der Begutachtung hingewiesen und um gründliche Überarbeitung gebeten haben. Leider sind in den endgültigen Druckvorlagen nicht alle Mängel behoben gewesen. Die kritischen Leser bitte ich, dies nicht dem Herausgeber anzulasten.

Zu drei Begriffen, die naheliegenderweise in den Beiträgen immer wieder vorkommen, möchte ich bei dieser Gelegenheit meine Meinung weitergeben. Ich tue das angeregt durch die Diskussion über den kürzlich im Informatikspektrum erschienenen Artikel von Prof. Rechenberg aus Linz [REC 91]:

- Nach meinem Verständnis der Regeln der deutschen Rechtschreibung heißt es "objektorientiert", und nicht "objekt-orientiert".
- "Browser" empfinde ich als einen unnötigen, vermeidbaren Amerikanismus. Mein Vorschlag ist, im Deutschen dafür den Begriff "Stöberer" zu verwenden. Der Wortstamm eignet sich gut zum Bilden von Verbformen wie "stöbern", "gestöbert haben" u.ä. (man stelle dem einmal das Wort "gebrowst" gegenüber, das sicherlich schon jemand in Deutschland gebraucht hat).
- "Instanz" hat bei uns eine eingeführte Bedeutung. Statt "Instanz einer Klasse", was mit dieser Bedeutung nicht in Einklang zu bringen ist, schlage ich "Exemplar einer Klasse" oder "Ausprägung einer Klasse" vor.

Ich danke dem Vorstand des German Chapter of the ACM e.V. dafür, daß meine Initiative zum Veranstellen einer Tagung über Eiffel so bereitwillig aufgegriffen und tatkräftig unterstützt wurde. Dem Fachausschuß *Software-Engineering* der Gesellschaft für Informatik e.V. danke ich ebenfalls für die gerne wahrgenommene Zusammenarbeit. Den Mitgliedern des Programmkomitees muß für die engagierte fachliche Unterstützung und den Mitarbeitern an meinem Lehrstuhl an der Techn. Hochschule Darmstadt für die willige organisatorische Unterstützung besonderer Dank gesagt werden. Mein

Dank geht an den Verlag für das unkomplizierte Herstellen des Tagungsbands.

Schließlich muß für das Angebot von Beiträgen gedankt werden, aus denen das Programmkomitee das Tagungsprogramm zusammenstellen konnte. Herrn Dr. Bertrand Meyer und Herrn Dr. Kim Walden ist für die beiden Vorträge zu Beginn und am Ende der Veranstaltung zu danken. Den Vortragenden der acht Fachbeiträge dazwischen und den Koordinatoren der Arbeitskreise gilt ebenso mein Dank.

Darmstadt, im Februar 1992

Univ.-Prof.Dr. Hans-Jürgen Hoffmann

- [Mey 88] Bertrand Meyer: Object-oriented Software Construction; Prentice-Hall, 1988
Bertrand Meyer: Objektorientierte Softwareentwicklung; (übersetzt von W. Simonsmeier) Carl-Hanser Verlag/Prentice-Hall Intl., 1990
- [Mey 91] Bertrand Meyer: Eiffel, the language; Interactive Software Engineering, Santa Barbara and Société des Outils du Logiciel, Paris, 1991
- [Rec 91] Peter Rechenberg: Übersetzungen von Informatik-Literatur bekümmert betrachtet; Informatik-Spektrum, Band 14, Heft 1, Februar 1991, Seiten 28 - 33

Inhalt

Vorwort	1
Bertrand Meyer Eiffel: Version 3 and beyond	5
Martin Nagler, Univ. Erlangen-Nürnberg Erweiterung von Eiffel um persistente Konzepte	7
Ruth Breu, TU München/Michael Breu, SNI München Ein Konzept der Modul- und Typvererbung	23
Tibor Németh, PROMATIS Informatik GmbH & Co. KG, Straubenhardt Methoden und Werkzeuge für den konstruktiven Entwurf von Objektsystemen	39
Rainer Fischbach, Starzach <i>Programming by Contract</i> - Erfüllt Eiffel das Ideal? -	55
Rüdiger Blach, Humboldt-Univ. Berlin Ein LL(1)-Parser für Eiffel	69
Wolfgang Strunk, Techn. Univ. Berlin Entwurf und prototypische Implementierung eines Klassenbrowsers für Eiffel	79
Dirk Bäumer, RWG Stuttgart/Horst Lichter, Univ. Stuttgart Ein <i>User Interface Management System</i> für Eiffel	91
Reinhard Budde, Marie-Luise Christ-Neumann, Karl-Heinz Sylla, Heinz Züllighoven, GMD Birlinghoven Erfahrungen beim objektorientierten Entwerfen und Analysieren	105
Koordinatoren der Arbeitskreise	107
Mitglieder des Programmkomitees	108

Eiffel: Version 3 and beyond

Bertrand Meyer

Interactive Software Engineering
Santa Barbara, Calif., USA

bertrand@EIFFEL.COM

Abstract

With version 3 of ISE's Eiffel implementation the various components of the Eiffel approach - method, language, environments, libraries - have been put together. This presentation will introduce the components of the brand new implementation, emphasizing the more novel tools, the support for object-oriented analysis and design, and the mechanisms allowing database access, graphics, and user interface construction.

The second part of the presentation will outline a program of action for the development of Eiffel in the years to come.

Programming by Contract

- Erfüllt Eiffel das Ideal? -

Rainer Fischbach

Hirtenbrunnle 25
7245 Starzach

rf@GARFIELD.T-INFORMATIK.BA-STUTTGART.DE

Zusammenfassung

Programmieren durch Vertrag ist ein auf Abstraktionen beruhendes Modell der Softwareentwicklung, das Nutzen und Verpflichtungen auf die Verwender und Lieferanten dieser Abstraktionen verteilt. Spezifikationen geben Verträgen eine verifizierbare Form. Unterstützen die Zusicherungen, die Bestandteil der Programmiersprache Eiffel sind, das Vertragsmodell der Softwareentwicklung in angemessener Weise, wie dies von ihrem Erfinder nahegelegt wird? Die folgende Argumentation weist diesen Vorschlag zurück, indem sie vor allem aufzeigt, daß die Zusicherungen von Eiffel das dazu erforderliche Abstraktionsniveau nicht erreichen.

Abstract

Programming by contract as a model of software development based on abstractions assigns benefits and obligations to users and providers of those abstractions. Specifications endow contracts with a verifiable form. Do the assertions that are part of the programming language Eiffel give adequate support to programming by contract, as suggested by its inventor? The following argumentation refutes this proposal, mainly by proving Eiffel assertions not to meet the required level of abstraction.

Problemstellung und Überblick

Den Proponenten der verschiedenen objektorientierten Programmiersprachen ist gemeinsam, daß sie sich von deren Einsatz wesentliche Produktivitätsgewinne in der Softwareentwicklung versprechen. Diese Gewinne sollen darauf beruhen, daß die für den objektorientierten Ansatz typische Abstraktionsform die Komplexität von Softwaresystemen reduziert, indem sie deren Code in lokal verständliche Einheiten – Klassen, die jeweils eine Menge gleichartiger Objekte beschreiben – zerlegt und dabei das sichtbare Verhalten dieser Objekte strikt von seiner verborgenen Implementation scheidet. Die Klasse ist als Codemodul eine syntaktische und als Typkonstruktor auch eine semantische Einheit.

Während eine Klasse einen statischen Namensraum bildet, aus dem einzelne Elemente kontrolliert exportiert werden können, repräsentiert ein Objekt eine dynamische Umgebung, d. h. eine Variable, deren konkreter Wert eine Menge von Bindungen der lokalen Namen seiner erzeugenden Klasse ist. Die Merkmale, welche die Abstraktion des externen Objektverhaltens konstituieren, entsprechen den exportierten Namen der lokalen Umgebung. Diese Namen können für Attribute oder Routinen stehen. Die Anwendung eines Merkmals auf ein Objekt impliziert die Ausführung der korrespondierenden, d. h. in der von dem Objekt repräsentierten Umgebung an seinen Namen gebundenen Routine in dieser Umgebung oder im Falle eines Attributs dessen Auswertung in derselben. Die Anwendung eines exportierten Merkmals ist außerhalb der das Objektverhalten definierenden Klassen die einzig zulässige Form des Zugriffs auf ein Objekt.

Zu den positiven Folgen dieses Ansatzes sollen die lokale Verständlichkeit des Objektverhaltens bzw. der Codemodule, die es beschreiben, deren erhöhte Wartbarkeit, Änderbarkeit und Erweiterbarkeit, sowie schließlich auch deren Wiederverwendbarkeit gehören. Die hier vertretene Sicht der objektorientierten Programmierung hält die Existenz lokaler Objekte im Unterschied zum Konzept eines zusammenhängenden, globalen Speichers für deren wesentliches Merkmal – ohne den hohen Wert von mehrfacher Vererbung und dynamischer Bindung von Merkmalen in Frage stellen zu wollen. [1]

Die Versprechen des objektorientierten Ansatzes einzulösen, verlangt jedoch, daß eine Voraussetzung erfüllt ist, deren Bestehen oft nur unterstellt wird: Die Abstraktion von der Implementation bedarf der Spezifikation des Verhaltens der Objekte, wenn sie nicht unverbindlich bleiben soll:

'Abstraction provides the two key benefits of locality and modifiability. Both are based on the distinction between an abstraction and its implementations. Locality means that each implementation can be understood in isolation. An abstraction can be used without our having to understand how it is implemented, and it can be implemented without our having to understand how it is used. Modifiability means that one implementation can be substituted for another without disturbing the using programs.

To obtain these benefits, we must have a description of the abstraction that is distinct from any implementation. [...] Users can assume the behavior described by the specification, and implementers must provide this behavior. Thus the specification serves as a contract between users and implementers.' [2]

Von der Implementation kann nur abstrahiert werden, wenn das implementierte bzw. zu implementierende Verhalten auch implementationsunabhängig beschrieben wird. Dies ist die Aufgabe der Spezifikation. Die Spezifikation ist ein verbindlicher Bestandteil der Dokumentation einer Klassenschnittstelle, welche das abstrakte Verhalten einer Menge von Objekten zur Verfügung stellt. Der Aufbau korrekter Systeme aus wiederverwendbaren Komponenten ist nur auf der Basis solcher durch präzise Spezifikationen gebundener Abstraktionen möglich. Ohne präzise Spezifikation gibt es weder eine sichere Verwendung noch eine überprüfbare Implementation von Abstraktionen.

Hinter der Klasse X mit den unten angeführten Features könnte sich ein Stapel, doch ebenso gut auch eine Schlange oder eine Struktur mit völlig undurchsichtigem Verhalten verbergen. Die Signatur einer Klasse sagt nicht genug über das Verhalten der zu ihr gehörenden Objekte aus, um spezifische Erwartungen zu rechtfertigen. Sie ist zwar ein notwendiger, doch kein hinreichender Faktor einer Klassenspezifikation. [3]

```
class X [T]
feature
  item: T is ... end
  put (x: T) is ... end
  remove is ... end
```

```
count: INTEGER is ... end
end -- class X
```

Die umgangssprachliche Beschreibung des Objektverhaltens wird sicher immer die Basis des Verständnisses einer Klasse bilden. Formale Spezifikation ist nicht ihr Ersatz, sondern ihre Ergänzung. Eine formale Beschreibung setzt informale Präzision voraus, verfügt jedoch – zumindest wenn sie gewisse, noch zu diskutierende Kriterien erfüllt – über den Vorzug konzis, präzise und eindeutig zu sein. Einer formalen Spezifikation kommt deshalb die Rolle der letzten Referenz zu, vor der sich jede Interpretation umgangssprachlicher Beschreibungen rechtfertigen muß.

Eine Spezifikation verpflichtet zunächst diejenigen, die eine Abstraktion implementieren, und garantiert deren Verwendern ein bestimmtes Verhalten. Oft wird dieses Verhalten jedoch nicht bedingungslos gewährt: Die Nutzer müssen bestimmte Vorleistungen erbringen, um in den Genuß der zugesagten Leistungen zu kommen. So muß z. B. eine indizierbare Tabelle sortiert sein, um eine binäre Suche zuzulassen. Diese Vorbedingung wird andererseits als Nachbedingung einer Sortierprozedur zugesichert. Die binäre Suche ist also immer zulässig, wenn die Tabelle zuvor sortiert wurde und erbringt dann auch das zugesicherte Ergebnis, etwa einen ganzzahligen Index, der den Platz des gesuchten Elementes angibt, wenn er innerhalb des belegten Indexbereichs liegt, und im anderen Fall anzeigt, daß dieses Element in der Tabelle nicht vorhanden ist.

Die Korrektheit einer Anweisungssequenz ergibt sich also aus der Implikation der Vorbedingungen der Folgeanweisungen durch die Nachbedingungen der vorausgehenden. Diese und entsprechende Regeln für weitere Formen der algorithmischen Konstruktion wie Schleifen, Auswahl, etc. erlauben es, Programme aus Spezifikationen zu entwickeln bzw. zu beweisen, daß ein gegebener Code eine Spezifikation erfüllt.

Die juristische Metapher von der Spezifikation als Vertrag benennt nicht nur die formalen Konstitutiva korrekter Software, sondern schlägt zugleich ein Modell der Arbeitsteilung zwischen den Lieferanten und Kunden von Softwarebausteinen vor. Mit Eiffel verbindet sich oft die Erwartung, hier liege ein Medium vor, in dem sich dieses Modell der Softwareentwicklung durchsetzen lasse. [4]

Eiffel kennt Zusicherungen in der Gestalt von Ausdrücken mit Typ BOOLEAN. Diese Zusicherungen können als Klasseninvarianten sowie als Vor- und Nachbedingungen von Routinen auftreten, in welcher Form sie den Vertrag zwischen den Kunden einer Klasse und denjenigen formulieren sollen, die diese Klasse zur Verfügung stellen, weiterhin als Invarianten von Schleifen und in unspezifischen check-Klauseln, die der systematischen Konstruktion und Dokumentation von Implementationen dienen sollen.

Ist der Anspruch, damit sei das Vertragsmodell der Softwareentwicklung auf eine tragfähige Grundlage gestellt, auch begründet? Dagegen existiert eine Reihe von Einwänden. Hier die Auflistung der Einwände in der Reihenfolge vom Grundsätzlichen und Allgemeinen zum Besonderen:

1. Spezifikationen müssen unabhängig von der Implementation sein. Programmiersprachliche Ausdrücke wie die Eiffel-Assertions, deren Werte vom konkreten Zustand eines in der Ausführung begriffenen Programms abhängen, können *per definitionem* nicht implementationsunabhängig sein. Sie bedürften vielmehr selbst erst der Spezifikation und eines Beweises ihrer Korrektheit. Sie repräsentieren deshalb eine methodologische *Petitio Principii*.
2. Spezifikationen sollten so abstrakt wie möglich bleiben und keine unnötigen Festlegungen enthalten. Werden Prädikate in der Implementationssprache abgefaßt, wird diese Anforderung in der Regel nicht erfüllt und vor allem ein Anlaß zum Durchbrechen der Schranke zwischen Abstraktion und Implementation gegeben.

3. Programmiersprachliche Ausdrücke können der doppelten Anforderung an eine Spezifikation, nämlich abstrakt und präzise zu sein, nicht genügen. Sie bleiben, sofern in ihnen von den einzigen hier zur Verfügung stehenden Formen der Abstraktion – der durch Abbreviation und Parameterisierung – Gebrauch gemacht wird, eben unpräzise und unspezifisch und vermögen Präzision und Besonderheit nur zu erlangen, indem sie auf Abstraktion gänzlich verzichten.
4. Spezifikationen, die optional auswertbare Ausdrücke sind, bilden vielmehr Bestandteile der Software und können – zumal in einer Sprache, die referentielle Transparenz nicht garantiert – deren Verhalten beeinflussen. Spezifikationen müssen dagegen Aussagen über das Verhalten von Software und nicht Teile derselben sein.
5. Ausdrücke der Programmiersprache sind inadäquate sprachliche Mittel: Die deklarative Semantik der Spezifikation wird durch die prozedurale Semantik der Programmiersprache beschränkt.
6. Zum Wesen des Vertrags gehört es, nicht einseitig abänderbar zu sein. Da Spezifikationen in Eiffel jedoch nicht unabhängig von der Implementation formulierbar sind, ist es den Implementierenden mittels polymorpher Merkmale jederzeit möglich, den Vertrag neu zu schreiben.
7. Der Status von Spezifikationen als programmiersprachlichen Ausdrücken, die optional auch evaluiert werden können, macht sie zwar zu brauchbaren Testhilfen, fördert jedoch das Mißverständnis, Spezifikationen seien dazu da, zur Laufzeit geprüft zu werden, und nicht, um die Konstruktion von Software zu ermöglichen, deren Korrektheit einer argumentativen Begründung fähig ist.

Ein kritisches Überdenken der Eiffel-Assertions ist jedoch noch aus einem weiteren Grund angezeigt: Das Typsystem von Eiffel, das einerseits Zuweisbarkeit an Variable bzw. Substituierbarkeit für formale Argumente auf Vererbung gründet, andererseits jedoch einem Klassenerben erlaubt, den Exportstatus von Merkmalen einzuschränken sowie den Typ von Attributen und von formalen Routinenargumenten kovariant zu verfeinern, verlangt danach, den Begriff der Typkonformität neu zu überdenken. Soll darüber hinaus das Konzept der abstrakten Datentypen ernst genommen werden, so ist ohnehin angezeigt, unter der Konformität von Datentypen mehr zu verstehen als nur die Verträglichkeit von Signaturen gemäß der Kontravarianzregel.

Ein semantischer Begriff der Typkonformität, der nicht auf der syntaktischen Beziehung des Erbens basiert und die Verträglichkeit der Signaturen zwar als notwendiges, doch nicht als alleiniges Kriterium akzeptiert, kann sich nur auf eine formale Spezifikation des Objektverhaltens stützen. Dadurch vermag er auch unverwandte Klassen als Implementationen desselben abstrakten Typs auszuweisen. Abstrakte Typen können als Familien von nicht notwendigerweise verträglichen konkreten Typen begriffen werden. [5]

Letzteres erscheint vor allem aus Klientensicht attraktiv zu sein: Die potentiellen Verwender von Softwarebausteinen müssen ja von einer Spezifikation des Verhaltens ausgehen, das sie benötigen, und dann anhand der Spezifikation eines Kandidaten feststellen, ob er ihre Anforderungen erfüllt.

Abstrakte versus konkrete Spezifikation

Der offensichtlichste Einwand gegen die Zusicherungen von Eiffel betrifft die mangelnde Ausdrucksmächtigkeit der Subsprache der booleschen Ausdrücke. In der Literatur finden sich hierzu Stellungnahmen, die diesen Sachverhalt zwar prinzipiell zugestehen, jedoch in der Praxis für kompensierbar halten:

'The assertion sublanguage of Eiffel is not a full-fledged formal specification language but is limited to boolean expressions, with a view extension. Purely applicative expressions are usually sufficient to cover the most important semantic properties of routines and classes; more advanced properties are captured by functions.' [6]

Eine neuere Äußerung verstärkt diese Aussage:

'As part of its specification, a class contains assertions, which formally express the precise properties of its operations.' [7]

Die Annahme, die Ausdrucksfähigkeit der Zusicherungen in Eiffel sei hinreichend, um die wichtigsten semantischen Eigenschaften von Klassen zu beschreiben, ist schwer verständlich und die vorgeschlagene Abhilfe nicht ohne Risiko. Ein Beispiel wird dies erläutern. Ich betrachte den bereits oben skizzierten Fall einer indizierbaren Tabelle:

```
class TABLE [T->ORDER]
creation make
feature
  add (x: T) is
    do count := count + 1
      if count > a.size then a.resize (1, 2 * a.size) end
      a.put (x, count)
    ensure count = old count + 1 end
  item (n: INTEGER): T is
    require 1 <= n; n <= count
    do Result := a.item (n)
    end
  count: INTEGER
  sort is
    do the_sorter.sort (a, 1, count)
    ensure sorted; count = old count end
  sorted: BOOLEAN is
    -- are the items in Current sorted in
    -- nondecreasing order?
    local i, j: INTEGER
    do
      from i := 1 j := 2
      until j > count or else item (j) < item (i)
      loop i := j j := j + 1 end
      Result := j > count
    end
  index (x: T): INTEGER is
    -- search for item 'x' and return its index, if it
    -- is present, return 0 otherwise
    require sorted
    local l, u, m: INTEGER
    do
      from l := 1 u := count m := (l + u) div 2
      until u < l or else x.is_equal (a.item (m))
      loop
        if x < a.item (m) then u := m - 1
        else l := m + 1 end
        m := (l + u) div 2
      end
    end
end
```

```

end
  if l <= u then Result := m end
ensure 1 <= Result and Result <= count implies
  x.is_equal (item (Result))
end
make is do a.make (1, 100)
  ensure count = 0 end
feature {NONE}
  a: ARRAY [T]
  the_sorter: SORTER [T] is
    once !!Result end
invariant count >= 0
end -- class TABLE

```

Das Feature `index` hat zur Vorbedingung, daß die Tabelle sortiert ist, da ein effizienter Suchalgorithmus wie die binäre Suche sonst nicht anwendbar wäre. Dies müßte als Nachbedingung des Features `sort` zugesichert werden. Mit den Ausdrucksmitteln der Prädikatenlogik läßt sich dieser Sachverhalt implementationsunabhängig formulieren:

$$x.\text{SORTED} = \forall i, j: \text{INTEGER} \bullet 1 \leq i \wedge i < j \wedge j \leq x.\text{COUNT} \implies x.\text{ITEM}(i) \leq x.\text{ITEM}(j)$$

Die Sortiertheit in nichtfallender Folge ist als Vorbedingung der binären Suche ausreichend, als Nachbedingung eines Sortieralgorithmus jedoch nicht, da hier zusätzlich verlangt werden muß, daß die sortierte Tabelle eine Permutation der ursprünglichen Tabelle ist, d. h. es muß eine Bijektion $p: [1, x.\text{COUNT}_{\text{pre}}] \rightarrow [1, x.\text{COUNT}_{\text{post}}]$ existieren, so daß

$$\forall k: \text{INTEGER} \bullet k \in [1, x.\text{COUNT}_{\text{pre}}] \implies x.\text{ITEM}_{\text{pre}}(k) = x.\text{ITEM}_{\text{post}}(p(k))$$

Natürlich impliziert deren Existenz, daß $x.\text{COUNT}_{\text{pre}} = x.\text{COUNT}_{\text{post}}$ ist. Sortieren ist zudem idempotent: $x.\text{SORT.SORT} = x.\text{SORT}$. Die Bedingung, daß sich die Anzahl der Einträge in der Tabelle durch das Sortieren nicht ändert, läßt sich noch durch die Eiffel-Zusicherung `count = old count` ausdrücken und die Sortiertheit läßt sich durch eine Eiffel-Funktion bestätigen, doch ist nicht erkennbar, wie sich die Forderung, daß Sortieren die Einträge nur permutiert und nicht etwa durch andere ersetzt, in diese Form umsetzen ließe.

Die tiefgestellten Indizes *pre* und *post* in den obigen Formeln verweisen auf den Zustand vor bzw. nach Ausführung der spezifizierten Routine. Die großgeschriebenen Namen wie `COUNT` und `ITEM` gehören der Sprache an, in welcher die abstrakten Werte spezifiziert werden, die Objekte der Klasse `TABLE` annehmen können. Sie sind deshalb auch keine programmiersprachlichen Notationen: Vielmehr bezeichnen sie mathematische Funktionen, deren Vor- und Nachbereiche Mengen abstrakter Werte sind. Sie sind nicht mit den entsprechenden, kleingeschriebenen konkreten Features identisch.

`ITEM` ist ein Funktional, also eine Funktion, die eine Funktion liefert, und zwar ist es von der Form $\text{ITEM}: \text{TABLE}[T] \rightarrow \text{INTEGER} \rightarrow T$. Sein Wert $x.\text{ITEM}$ für ein Objekt x ist eine partielle Funktion, die ein Anfangssegment der positiven ganzen Zahlen in die Menge T abbildet. `SORT` ist eine Funktion, welche die Menge der Werte von `TABLE` $[T]$ in sich selbst abbildet, also $\text{SORT}: \text{TABLE}[T] \rightarrow \text{TABLE}[T]$. T ist eine gebundene Sortenvariable, welche die Theorie einer totalen Ordnung einführt. Eine Spezifikation, welche von einer solchen gebundenen Variablen abhängt, legt eine ganze Familie von Datentypen fest bzw. faßt die Menge der Spezifikationen zusammen, welche durch die zulässigen Substitutionen der gebundenen Variablen bestimmt ist.

Den abstrakten Werten entsprechen Äquivalenzklassen von konkreten Werten, die aus der Sicht der Klienten ununterscheidbar sind. Der Wert von $x.\text{ITEM}$ – das ist eine Funktion, welche den ganzzahligen Bereich $[1, x.\text{COUNT}]$ in T abbildet – charakterisiert den abstrakten Wert des Objekts x der Klasse `TABLE` vollständig. Zwei Objekte t_1 und t_2 dieser Klasse sind gleich – nicht notwendigerweise identisch –, wenn die Funktion `ITEM` für beide Objekte identisch ist: $t_1.\text{ITEM} = t_2.\text{ITEM}$: Genau dies müßte die Bedeutung von `t1.is_equal(t2)` sein, die sich innerhalb von Eiffel wie auch von anderen Programmiersprachen nicht formulieren läßt, da die Gleichheit von Funktionen keine operative Bedeutung hat! In welchem Verhältnis stehen nun die Zusicherungen der angeführten Eiffel-Klasse, die dem vorgeschlagenen Schema entspricht, zu den obigen Prädikaten?

Zunächst ist nicht ohne weiteres klar, ob der Körper der Funktion `sorted` genau dann den Wert `true` liefert, wenn das angeführte Prädikat von dem abstrakten Wert eines Objektes ausgesagt werden kann. Der Beweis dieses Sachverhalts ist umständlich und muß von bestimmten Eigenschaften einer Ordnungsrelation, wie der Transitivität, Gebrauch machen. Er setzt also eine Theorie der Operatoren '`<`' bzw. '`<=`' voraus, die dartut, daß selbige eine Ordnungsrelation repräsentieren. Der Mechanismus der eingeschränkten generischen Parameter in Eiffel gewährleistet lediglich, daß diese Operatoren existieren, nicht jedoch, daß sie die Bedeutung einer totalen Ordnungsrelation auf den Typen haben, an die der formale Parameter gebunden sein kann! Letzteres könnte nur – hier funktioniert die Generizität von Klassen als Abstraktionsmechanismus nicht – aus ihrer jeweiligen Implementation hervorgehen.

Um sich davon zu überzeugen, daß das konkrete Feature `sorted` die gewünschte Eigenschaft hat, ist es notwendig, die Schwelle zwischen Abstraktion und Implementation zu durchbrechen und den Code zu inspizieren. Das besagte Verfahren spezifiziert also keine abstrakten Datentypen. Zudem ist ein solcher Nachweis nur sinnvoll, wenn zuvor ein unabhängiger und klarer Begriff von Sortiertheit gewonnen wurde.

Im vorliegenden Fall macht die Routine `sorted` nur von öffentlichen Features Gebrauch. Nicht selten müßten vergleichbare Funktionen jedoch auf verborgene Attribute zugreifen. Die Gleichheit von abstrakten Werten läßt sich entweder durch die Gleichheit zwischen Termen ausdrücken, die mit den Funktionszeichen der Signatur aufgebaut werden, oder über die Referenzobjekte, die ein explizites mathematisches Modell liefert. Die in Eiffel vorhandenen Mittel der Spezifikation versagen vor dieser Aufgabe. Eigenschaften, wie die von `QUEUE`, daß

$$q.\text{ADD}(x).\text{REMOVE} = \begin{cases} q, & \text{if } q.\text{EMPTY} \\ q.\text{REMOVE}.\text{ADD}(x), & \text{otherwise.} \end{cases}$$

sind mit Hilfe der Zusicherungen von Eiffel nicht formulierbar. [8] Auch die wesentlichen Eigenschaften von Mengen, Bags, Graphen etc. lassen sich durch Eiffel-Zusicherungen nicht ausdrücken. [9] Eiffel-Funktionen liefern für eine Spezifikation gleichzeitig zu viel *und* zu wenig:

- Zu viel, weil eine Spezifikation nur eine universelle Aussage sein soll und keine Methode zur empirischen Verifikation ihrer singulären Instanzen;
- zu wenig, weil die deklarative Bedeutung des ausführbaren Codes alles andere als offensichtlich und klar ist.

Die Denotation einer wirklich unabhängigen, abstrakten Spezifikation ist die Äquivalenzklasse der sie erfüllenden Implementationen. Doch genau diese Klasse bleibt bei programmiersprachlichen Spezifikationen unbestimmt, weil nicht klar ist, *wovon* abstrahiert werden, d. h. worin Äquivalenz bestehen soll.

Aussagen, welche eine universelle Quantifikation oder die Negation einer Existenzbehauptung enthalten, sind nur sehr umständlich prozedural zu fassen. Der Rückgabewert

0 impliziert beim Feature `index`, daß in der Tabelle kein Wert existiert, der mit dem Argument x identisch ist.

$$t.\text{INDEX}(x) = 0 \implies \neg \exists k: \text{INTEGER} \bullet t.\text{ITEM}(k) = x$$

Eine konkrete prozedurale Spezifikation dieses Sachverhalts würde, wie übrigens auch der Versuch, die Schleifeninvariante der binären Suche innerhalb von Eiffel zu explizieren, auf nichts anderes hinauslaufen, als eine Suchroutine – also etwas von derselben Art, wie das, was gerade spezifiziert werden soll – zu kodieren... Hier die Schleifeninvariante:

$$\begin{aligned} (\exists m: \text{INTEGER} \bullet c.\text{ITEM}(m) = x) \implies \\ \exists n: \text{INTEGER} \bullet l \leq n \wedge n \leq u \wedge c.\text{ITEM}(n) = x \end{aligned}$$

wobei *Current* durch c abgekürzt ist. Aus $u < l$ folgt natürlich sofort, daß ein solcher Index nicht existiert und die Tabelle deshalb den gesuchten Wert nicht enthält.

Die Spezifikation ist nicht dazu da, bei jeder Applikation eines Features auf eine Instanz der Klasse überprüft zu werden – nur dazu wäre der ausführbare Code gut –, sondern um als Grundlage einer strengen Argumentation zu dienen, welche die Korrektheit sowohl einer Implementation als auch einer bestimmten Verwendung nachweist. Selbst als Testhilfen wären solche Funktionen, wie sie als Komponenten von Zusicherungen oft vorgeschlagen werden, strenggenommen nur dann zulässig, wenn sie ihrerseits spezifiziert und bewiesen wären!

Circulus vitiosus

Prädikate, in deren Formulierung Quantoren vorkommen müßten, durch Eiffel-Funktionen zu ersetzen, kann, wie einige weitere Überlegungen zeigen, nämlich nur bedeuten, ein ohnehin schon zirkuläres Verfahren eine Umdrehung weiter zu treiben. Die logische Zulässigkeit eines solchen Kunstgriffes würde also die formale Spezifikation und den Korrektheitsbeweis der Routine requirieren. Ein solches Vorgehen ist bodenlos und kann die Beweislast der Lieferanten nicht verringern, sondern nur erhöhen. Dem Kunden wird dabei zugemutet, noch mehr glauben zu müssen, als wenn nur platt die Korrektheit der ursprünglichen Routine behauptet worden wäre. Darüber hinaus wird es jedem Erben ermöglicht, durch schlichte Umdefinition der betreffenden Funktionen sich scheinbar aller Lasten zu entledigen:

```
class BLACK_TABLE [T->ORDER]
inherit TABLE [T] redefine sorted, index end
feature
  sorted: BOOLEAN is true
  index (x: T): INTEGER is do end
end -- class BLACK_TABLE
```

Auf diese Problematik wurde bereits von Gary T. Leavens und William E. Weihl hingewiesen:

'However, assertions for Eiffel specifications are written using a type's operations. A subclass in Eiffel can redefine the operations of a superclass, so that while the implications among the pre- and post-conditions may be valid, the behavior of instances of the subtype may be surprising. The extreme of this problem occurs for deferred types: types for which one or more of the operations are not implemented (i.e. their implementation is deferred to a subclass). Consider a class D where all the operations are deferred. The pre- and post-conditions of the operations of D are

written using the operations of D. But the operations of D are not implemented, so the assertions that are used to define these operations are meaningless.' [10]

Die beiden Autoren vergessen an dieser Stelle natürlich nicht, darauf hinzuweisen, wie sich eine solche Situation vermeiden läßt: Durch den Gebrauch einer unabhängigen Form der Spezifikation.

Im Kontext einer objektorientierten Sprache läßt sich die referentielle Transparenz von Ausdrücken, die Funktionsaufrufe enthalten, nicht garantieren. Selbst von Seiteneffekten und verdeckten dynamischen Abhängigkeiten einmal abgesehen, nimmt die Möglichkeit, Funktionen in einer Nachkommenklasse neu zu implementieren, der Sprache der Ausdrücke jene Implementationsunabhängigkeit, Verbindlichkeit und Transparenz, die von einer Spezifikationssprache zu fordern ist. Die Formeln der Prädikatenlogik sind z. B. referentiell transparent. Ihre Bedeutung läßt sich aus dem lokalen Kontext ermitteln, sie hängt nicht von unsichtbaren Einflüssen ab und vor allem kann sie nicht durch Manipulationen an einem anderen Ort verändert werden.

Die einzelnen Vorwürfe an die gegenwärtige Gestalt der Zusicherungen in Eiffel sind nicht logisch unabhängig. Der allen anderen zugrunde liegende ist Einwand Nummer eins: Eine Programmiersprache kann nicht der Spezifikation dienen, zumindest nicht, solange sie dafür keine angemessenen Konstrukte mit entsprechender Semantik enthält.

Die Ausnahmen davon wären Programmiersprachen, in denen Programme selbst den Charakter von ausführbaren Spezifikationen annehmen, [11] etwa in Form von definitorischen Gleichungen wie in ML oder von Regeln wie in PROLOG. Doch selbst dort trifft dies nur bedingt zu. Bei den meisten PROLOG-Interpretern hat etwa die Reihenfolge, in der die Klauseln aufgeführt sind, Einfluß auf die operative Bedeutung eines Logikprogramms und es gibt intransparente prozedurale Elemente wie den Cut-Operator, mit dessen Hilfe sich solche Abhängigkeiten ausnutzen lassen. Am ehesten kommt unter den geläufigen Sprachen vielleicht noch Miranda in die Nähe des Ideals eines transparenten, spezifikatorischen Stils. [12]

Eine derartige Sprache, die es erlaubte, Spezifikationen auszuführen, müßte mit einer transparenten definitorischen Semantik ausgestattet sein, um garantieren zu können, daß die Spezifikationen eine klare und wohldefinierte operative Bedeutung haben. Die heute populären objektorientierten Sprachen – Eiffel eingeschlossen – bieten dafür keine Anhaltspunkte. Objektorientierte Programme erzielen ihre Resultate in der Regel durch Seiteneffekte und nutzen oft Aliasbeziehungen zwischen Namen aus. Zudem können Routinen ja redefiniert werden. Diese Kombination ist mit referentieller Transparenz nicht verträglich. Eiffel-Ausdrücke vom Typ `BOOLEAN` haben keine eindeutige definitorische Bedeutung, sondern eben nur eine implementationsabhängige und intransparente operative. Sie in Zusicherungen zu verwenden, bedeutet deshalb, eine Spezifikation nicht zu liefern, sondern den Anschein einer solchen zu erschleichen.

Die Bedeutung einer Spezifikation muß unabhängig von der Implementation sein. Schon das in der Spezifikation von abstrakten Datentypen unerläßliche Gleichheitsprädikat ist jedoch in Eiffel unabhängig von der Implementation nicht formulierbar, wenn es tatsächlich die Gleichheit von abstrakten Werten, d. h. die gemeinsame Mitgliedschaft zweier Objekte in einer Äquivalenzklasse verhaltensmäßig austauschbarer Repräsentationen meint und nicht etwa nur die paarweise Übereinstimmung aller ihrer Attributwerte.

Eine Ausnahme davon stellen lediglich besonders konstruierte Klassen dar, deren Schnittstelle in geeigneter Weise erweitert wurde – und zwar um ein explizites Modell der abstrakten Werte! Dasselbe gilt für praktisch alle interessanten Prädikate und Observatoren von abstrakten Datentypen.

Ein Beispiel soll dies verdeutlichen. Einen zirkulären FIFO-Puffer wird man in Eiffel

etwa folgendermaßen implementieren:

```
class FIFO [T]
  creation make
  feature
    item: T is require count > 0
      do Result := b.item (first) end
    add (x: T) is require count < size
      do count := count + 1 b.put (x, free)
        free := (free + 1) mod size
      ensure count = old count + 1 end
    remove is require count > 0
      do count := count - 1 first := (first + 1) mod size
        ensure count = old count - 1 end
    count: INTEGER;
    size: INTEGER is 10
    make is do b.make (0, size - 1) end
  feature {NONE}
    b: ARRAY[T]
    first, free: INTEGER
  end -- class FIFO
```

Wann sind zwei Objekte des Typs FIFO [INTEGER] gleich? Die Übereinstimmung der konkreten Repräsentationen des abstrakten Datentyps, d. h. aller Attribute ist dafür zwar eine hinreichende, doch keine notwendige Voraussetzung. Z. B. repräsentiert die folgende Menge

```
{FIRST ↦ {() ↦ 1}, FREE ↦ {() ↦ 5}, COUNT ↦ {() ↦ 4}, SIZE ↦ {() ↦ 10},
B.ITEM ↦ {0 ↦ 7, 1 ↦ 6, 2 ↦ 8, 3 ↦ 0, 4 ↦ 9, 5 ↦ 99, 6 ↦ 69, 7 ↦ 33, 8 ↦ 88,
9 ↦ 11}...}
```

von Bindungen den selben abstrakten Wert wie

```
{FIRST ↦ {() ↦ 8}, FREE ↦ {() ↦ 2}, COUNT ↦ {() ↦ 4}, SIZE ↦ {() ↦ 10},
B.ITEM ↦ {0 ↦ 0, 1 ↦ 9, 2 ↦ 88, 3 ↦ 33, 4 ↦ 69, 5 ↦ 19, 6 ↦ 609, 7 ↦ 13, 8 ↦ 6,
9 ↦ 8}...}
```

Ein Feature `is_equal`, das diesem Sachverhalt Rechnung trüge, müßte dann auf Implementationsdetails zurückgreifen. In die Implementation einer Abstraktion muß das Wissen darüber eingehen, *wovon* abstrahiert wird:

```
class FIFO_EQ [T]
  creation make
  inherit FIFO [T] redefine is_equal
  export {FIFO_EQ} b, first, free
  end
  feature
    is_equal (other: FIFO_EQ [T]): BOOLEAN is
      local i, j: INTEGER
      do
        if count = other.count then
          from i := first j := other.first
```

```
        until i = free or else
          not equal (b.item (i), other.b.item (j))
        loop i := (i + 1) mod size j := (j + 1) mod size end
        Result := i = free
      end
    end
  end -- class FIFO_EQ
```

Ein etwas transparenterer Ansatz würde dagegen ein explizites Modell der abstrakten Werte in die Klassenschnittstelle aufnehmen:

```
class FIFO_VAL [T]
  creation make
  inherit FIFO [T] redefine is_equal end
  feature
    val (n: INTEGER): T is
      require n > 0; n <= count
      do Result := b.item ((first + n - 1) mod size) end
    is_equal (other: FIFO_VAL [T]): BOOLEAN is
      local i: INTEGER
      do
        if count = other.count then
          from i := first
          until i > count or else
            not equal (val (i), other.val (i))
          loop i := i + 1 end
          Result := i > count
        end
      end
    end -- class FIFO_VAL
```

In diesem Fall würde den beiden oben angeführten konkreten Repräsentationen der gemeinsame abstrakte Wert

```
{COUNT ↦ {() ↦ 4}, SIZE ↦ {() ↦ 10},
VAL ↦ {1 ↦ 6, 2 ↦ 8, 3 ↦ 0, 4 ↦ 9}...}
```

entsprechen. Jedoch setzt dieser Kunstgriff die korrekte Implementierung von `is_equal` bzw. `val` voraus. Die beiden Klassen `FIFO_EQ` und `FIFO_VAL` sind – nebenbei gesagt – wunderbare Beispiele für die in Version 3 durch eine Lösung, die ad hoc entstanden zu sein scheint, eher kompensierte als wirklich bereinigte Problematik des Typsystems von Eiffel, da beide nicht typkonform zu ihrer Vorfahrenklasse `FIFO` sind! Dies resultiert hier darin, daß `is_equal` nicht einmal symmetrisch ist, da die Vertauschung der Argumente zu einem Typfehler führt.

Schon bei einem so fundamentalen Feature wie `is_equal` kann also nicht von einer transparenten deklarativen Semantik ausgegangen werden. Vielmehr lauern überall Fallstricke, die von der prozeduralen Semantik der Programmiersprache sowie der in der Regel unsichtbaren und nicht eindeutig festlegbaren konkreten Implementation dieses Merkmals herrühren. Zusicherungen, die derartiges enthalten, können deshalb nicht eindeutig sein!

In Spezifikationen, die den Namen verdienen, wird die Bedeutung der Gleichheit sowie anderer Prädikate durch Aussageschemata konstituiert, die angeben, wann zwei Terme gleich sind oder ein Prädikat auf einen Term bzw. ein Tupel von Termen zutrifft. Dadurch

ist es möglich, irreduzible Terme als kanonische Repräsentanten von Äquivalenzklassen auszuzeichnen. Das Gleichheitszeichen in einer algebraischen Spezifikation oder der definitorische Gleichheitsoperator von Miranda haben eine völlig andere Bedeutung als der Gleichheitsoperator bzw. die Funktion `is_equal` in Eiffel! In einer Sprache wie Eiffel konstituiert die Verwendung des Gleichheitsoperators oder einer anderen Funktion vom Typ `BOOLEAN` niemals eine präskriptive Aussage, sondern immer nur einen evaluierbaren Ausdruck, der selbst spezifikationsbedürftig ist. In Abwesenheit einer Spezifikation und Verifikation desselben kann sich seine Bedeutung nur aus seiner Implementation ergeben. Doch genau dann erfüllt die Zusicherung, in der er vorkommt, ihre Aufgabe als Medium der Abstraktion nicht mehr!

Zusicherungen von Eiffel sind als vermeintliche Spezifikationen entweder haltlos, weil zirkulär und unterdeterminiert, oder abstraktiv impotent. Ein programmiersprachlicher Ausdruck vom Typ `BOOLEAN` ist keine Proposition! Seine Bedeutung ist vielmehr entweder ein Wert aus der Menge `{false, true}` oder undefiniert. Solche Ausdrücke gehören der Objektsprache an und nicht der Metasprache, in der abstrakte Spezifikationen anzusiedeln wären. Spezifikationen sind Aussagen über Programme und nicht Bestandteile von Programmen. Überdies sollte die Wahl der Implementationssprache durch eine Spezifikationsprache allerhöchstens nahegelegt, nicht jedoch präjudiziert werden. Eine Spezifikation kann ihre formale Funktion wie auch ihre soziale Aufgabe mit einer symmetrischen Bindungswirkung für beide Seiten des Vertrags nicht erfüllen, solange sie die Kriterien der Unabhängigkeit, Transparenz und logischen Konsistenz nicht erfüllt.

Perspektiven

Daß die Eiffel-Assertions unbefriedigend sind, dürfte nunmehr deutlich geworden sein. Doch was wäre an ihre Stelle zu setzen?

Eiffel besitzt als Programmiersprache bereits viele attraktive Merkmale, weshalb sich eine Apologie, die mit wenig überzeugenden Argumenten ihre Schwächen als Spezifikationsprache rechtfertigt, eigentlich erübrigt. [13] Spezifikationen müssen nicht in ausführbarem Code resultieren, und deshalb auch keine Effizienzprobleme verursachen. Entsprechende Erweiterungen sollten orthogonal zur Syntax der ausführbaren Anweisungen sein und deshalb auch kein Hindernis für alle diejenigen darstellen, die Eiffel nur als Programmiersprache lernen und anwenden wollen. Dies muß nicht bedeuten, daß einer Erweiterung nicht auch Werkzeuge folgen sollten, die Spezifikationen ausführbar machen oder die Verifikation von Klassen unterstützen. Der konventionelle Kern der Programmiersprache muß davon nicht berührt werden.

An welchen Vorbildern könnte sich eine solche Erweiterung orientieren? Die formale Spezifikation von Software ist eine Disziplin, die bisher nur geringe praktische Bedeutung hat. Ausgesprochen unausgereift sind die Versuche, den objektorientierten Ansatz in ihr zum Tragen zu bringen. Die modellbasierten Methoden wie VDM [14] und Z [15] vertragen sich in ihrer gegenwärtigen Form schlecht mit den Prinzipien der objektorientierten Programmierung. Modularität und Datenabstraktion lassen sich mit ihrer Hilfe nicht erzwingen. Dies hat seinen Grund zum einen darin, daß es in beiden Sprachen nur primitive oder konkrete Datentypen gibt, also alle nichtprimitiven Typen explizit modelliert, d.h. in der Sprache der Mengenlehre implementiert werden müssen, und zum anderen in der Abwesenheit eines syntaktischen Konstrukts, das die Sichtbarkeit von Namen begrenzen würde. Objektorientierte Varianten modellbasierter Spezifikationsprachen wie Object-Z [16] werden gegenwärtig diskutiert, doch ist noch nicht klar, ob aus diesen

Bemühungen ein signifikanter Beitrag zur Ergänzung von Eiffel resultieren wird.

Algebraische Techniken scheinen sich mit den Prinzipien der objektorientierten Programmierung besser zu vertragen. Die *Traits* von Clu/LARCH stellen das einzige mir bekannte Beispiel einer gelungenen Ergänzung einer Programmiersprache durch eine Notation zur Spezifikation von abstrakten Datentypen dar. [17] Traits charakterisieren Datenabstraktionen, indem sie Signaturen einführen und die Einschränkungen, denen die abstrakten Werte unterliegen, mittels Gleichungen zwischen Termschemata formulieren. Eine Interface-Spezifikation verbindet ein Trait mit einem Cluster, das die Abstraktion implementiert.

In diesem Zusammenhang ist es angezeigt, ein weiteres Tabu anzutasten: die Werkzeuge wie *short* zugrunde liegende Idee, daß es möglich und sinnvoll sei, Klassenschnittstellen bzw. Dokumentationen aus den Klassentexten zu destillieren. Auch dieser Ansatz stellt – wie die Zusicherungen – das angezeigte Verhältnis von Spezifikation und Implementation auf den Kopf. Zur Unabhängigkeit der Spezifikation gehört natürlich auch, daß sie als Text *vor* bzw. *getrennt* vom Klassentext vorliegen sollte. Die Idee, die Dokumentation parallel zur Spezifikation bzw. zum Klassentext zu entwickeln ist prinzipiell gut und sinnvoll, doch bräuchte es dazu mächtigerer Werkzeuge als *short*, etwa einer abgewandelten Form von WEB. [18]

Anmerkungen

- [1] Ich schließe mich hier der Position von JOSEPH GOGUEN und JOSÉ MESEGUER an. Vgl. JOSEPH GOGUEN, JOSÉ MESEGUER: 'Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics', in: BRUCE SHRIVER, PETER WEGNER (Hrsg.): *Research Directions in Object-Oriented Programming*, Cambridge, MA: MIT-Press, 1987, S. 417–477.
- [2] BARBARA LISKOV, JOHN GUTTAG: *Abstraction and Specification in Program Development*, Cambridge, MA: MIT-Press; New York: McGraw-Hill, 1986, S. 53 f.
- [3] Vgl. PIERRE AMERICA: 'A Behavioural Approach to Subtyping in Object-oriented Programming Languages', in: MAURIZIO LENZERINI, DANIELE NARDI, MARIA SIMI (Hrsg.): *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Chichester: Wiley, 1991, S. 173–190.
- [4] Vgl. BERTRAND MEYER: 'Lessons from the Design of the Eiffel Libraries', in: *Communications of the ACM*, September 1990 (Bd. 33, Nr. 9), S. 75 f.
- [5] Vgl. RAINER FISCHBACH: 'Type and Class', in: JENS PALSBERG, MICHAEL I. SCHWARTZBACH (Hrsg.): *Types, Inheritance and Assignments – A Collection of Papers from the ECOOP'91 Workshop W5*, Geneva, Switzerland, July 1991, Aarhus: University, Computer Science Department, Juni 1991 (DAIMI PB; 357), S. 19 f.
- [6] BERTRAND MEYER: 'Lessons from the Design of the Eiffel Libraries', a. a. O., S. 72
- [7] BERTRAND MEYER: 'From the bubbles to the objects', in: *Object Magazine*, November/Dezember 1991 (Bd. 1; Nr. 4), S. 38.
- [8] Vgl. BERTRAND MEYER: *Object-Oriented Software Construction*, London: Prentice Hall, 1988, S. 155 f.
- [9] Vgl. RAINER FISCHBACH: 'Implementing a Graph ADT in Eiffel', in: *TOOLS'89*, Proceedings, Paris 1989, S. 449–454, wo auf prädikatenlogische Formeln in Kommentarzeilen zurückgegriffen wurde.
- [10] GARY T. LEAVENS, WILLIAM E. WEIHL: 'Reasoning about Object-Oriented Programs that use Subtypes', in: *ECOOP/OOPSLA '90*, Proceedings, New York: ACM-Press, 1990, S. 220 f.
- [11] Ausführbare Spezifikation heißt hier natürlich, daß ein Softwarewerkzeug aus der Spezifikation eine Implementation ableiten kann und nicht, daß Assertions zur Laufzeit überprüft werden!
- [12] Vgl. DAVID A. TURNER: 'Functional programs as executable specifications', in: C. A. R. HOARE, J. C. SHEPERDSON (Hrsg.): *Mathematical Logic and Programming Languages*, Englewood Cliffs, NJ: Prentice-Hall, 1985, S. 29–54.

- [13] MEYER, *Object-Oriented Software Construction*, a. a. O., S. 156.
- [14] CLIFF B. JONES: *Systematic Software Development using VDM*, 2. Aufl., London: Prentice Hall, 1990.
- [15] J. M. SPIVEY: *The Z Notation: A Reference Manual*, London: Prentice Hall, 1989.
- [16] ROGER DUKE, PAUL KING, GORDON ROSE, GRAEME SMITH: *The Object-Z Specification Language, Version 1*, Software Verification Centre, Department of Computer Science, The University of Queensland, 1991 (Technical Report; 91-1).
- [17] Vgl. LISKOV und GUTTAG, a. a. O., S. 187 ff.
- [18] WAYNE SEWELL: *Weaving a Program: Literate Programming in WEB*, New York: Van Nostrand Reinhold, 1989.

Ein LL(1)-Parser für Eiffel

Rüdiger Blach

Humboldt-Universität zu Berlin
 Fachbereich 16 - Informatik
 Institut für Softwaretechnik

Postfach 1297
 D-O-1086 Berlin

blach@hubinf.de

Zusammenfassung

Aus der von Meyer angegebenen Syntaxbeschreibung für Eiffel wird eine Syntax in EBNF konstruiert. Diese läßt sich so umformen, daß sie den Bedingungen an LL(1)-Grammatiken genügt. Mittels einer für die Ausgabe von Quelltexten in C++ modifizierten Variante des Compilergenerators Coco/R kann nun ein lauffzeiteffizienter, nach der Methode des rekursiven Abstiegs arbeitender Eiffel-Parser erzeugt werden.

Abstract

Starting with Meyers syntax description for Eiffel an EBNF syntax was constructed. This syntax was modified to a grammar which fulfils LL(1)-conditions. After that a modified version of the compiler generator Coco/R was used to generate a runtime efficient, recursive descendent parser for Eiffel in C++.

1 Motivation

Der von ISE gelieferte Compiler für Eiffel 2.3 [7] übersetzt in vier sequentiell nacheinander ablaufenden Pässen nach C. Häufig wird er wegen zu langer Übersetzungszeiten kritisiert.

Ursachen dafür lassen sich zum Teil in der Sprache selbst finden. Schließlich bietet Eiffel eine Reihe von Konzepten an, die insbesondere zur Entwicklung großer Softwareprodukte in hoher Qualität gedacht und in vielen anderen Sprachen (z.B. C) nicht vorhanden sind. Eiffel ermöglicht die Anwendung moderner softwaretechnologischer Methoden,