Universität Kaiserslautern
Fachbereich Elektrotechnik
Lehrstuhl für Digitale Systeme
Prof. Dr.-Ing. S. Wendt

Diplomarbeit

Bereitstellung einer Experimentierumgebung zur Benutzung eines Softwarebusses und Untersuchung der objektorientierten Programmiersprache Eiffel

Eberhard Iglhaut
Oktober 1996

Betreuer: Prof. Dr.-Ing. S. Wendt

Bearbeiter: Eberhard Iglhaut
Obere Hauptstr. 23
67551 Worms

Erklärung:

Hiermit erkläre ich, die vorliegende Diplomarbeit unter Verwendung der angegebenen Quellen und ohne fremde Hilfe angefertigt zu haben.

Walldorf, den 11. Oktober 1996

(Eberhard Iglhaut)

Danksagung

An dieser Stelle danke ich allen, die zum Gelingen dieser Arbeit beigetragen haben.

Herrn Prof. Dr.-Ing. Siegfried Wendt danke ich für die zahlreichen richtungsweisenden Gespräche und das große mir entgegengebrachte Vertrauen. Sowohl der mir eingeräumte Entscheidungsspielraum als auch Prof. Wendts Bereitschaft, seine Zeit zur Verfügung zu stellen, waren außergewöhnlich groß.

Der Firma SAP möchte ich dafür danken, daß sie die Arbeit finanziell und räumlich ermöglicht hat. An vielen Stellen standen Mitarbeiter der SAP meinen zwei Kommilitonen und mir hilfsbereit und wohlwollend zur Seite.

Der Firma TIBCO danke ich für die kostenlose Bereitstellung des Rendezvous-Software-Busses und die prompte Beantwortung meiner Fragen.

Meinen Kommilitonen Bernhard Gröne und Johannes Otto danke ich für die stets gute Arbeitsathmosphäre, die vielen hilfreichen Diskussionen und ihre geduldige Hilfsbereitschaft bei meinen zahllosen Kämpfen mit den Tücken des Computers.

Eberhard Iglhaut

Inhaltsverzeichnis

1 Einleitung

Seit einigen Jahren gibt es eine Zusammenarbeit zwischen dem Lehrstuhl für digitale Systeme der Universität Kaiserslautern und der Firma SAP AG, Walldorf.

Im Rahmen dieser Zusammenarbeit ist bei der SAP die Abteilung Basismodellierung entstanden. Aufgabe dieser Abteilung ist es insbesondere, Teile des SAP R/3-Systems mit den von Prof. Dr. Siegfried Wendt entwickelten Techniken zu beschreiben.

Im Laufe der Zeit entstand auch bei SAP der Wunsch, über die reine Beschreibung vorhandener Systeme hinaus die Architektur noch zu entwickelnder Systeme zu verbessern. Zu diesem Zweck bildete sich eine Gruppe mit der Bezeichnung „Architekturplanungsteam“ unter der Leitung von Prof. Wendt. Das Architekturplanungsteam gibt es inzwischen zwar nicht mehr in dieser Form, seine Ziele werden aber weiterverfolgt.

Selbstverständlich ist es auch bei der Planung von Software-Architektur sinnvoll, über den eigenen Tellerrand hinauszublicken und zu beobachten, was andere tun. Häufig sind interessante Ideen auch schon in Produkten verwirklicht, so daß es sich lohnt, diese Produkte kennenzulernen.

„Software is *everything* we use

to tell *machines*

how to do what we want,

and to tell *each other*

what we have decided the machines should do.“[1]

Dieser Satz betont eine Zweiteilung der Funktion von Software. Die erste Funktion ist die, Programm für Maschinen zu sein. In dieser Funktion ist Software reines *Produkt*, eine Komponente eines technischen Systems, nicht anders als es eine Schraube auch ist.

Ungewohnt ist es, Software auch die anschließend genannte Funktion zuzuordnen. Mit der steigenden Komplexität eines Produkts wird die Kommunikation von Menschen über das Produkt immer wichtiger. Und dies bezieht sich nicht nur auf das Endprodukt selbst, dessen Bestandteil Software in der zweiten Funktion beispielsweise in der Form einer Gebrauchsanleitung sein kann. Die Verständigung über von uns getroffene Entscheidungen ist ganz besonders im stark arbeitsteiligen *Prozeß* der Herstellung eines komplexen Produkts von Bedeutung

Wenn man den Begriff Software dermaßen umfassend versteht – was leider unüblich ist – dann fällt auf, daß die Softwareindustrie im ersten Bereich (nämlich dem Erstellen von maschinenlesbaren Programmen) zwar beeindruckende Erfolge erzielt hat, daß es aber an den für Menschen lesbaren Beschreibungen deutlich hapert.

Um bestimmte Produkte kennenzulernen, kann man sich auf schriftliche Beschreibungen allein in der Regel nicht verlassen. Es ist nötig, eine Zeitlang mit den Produkten zu arbeiten. Für eine tiefergehende Kenntnis ist dies auch unabhängig von der Qualität der Beschreibungen notwendig.

Im Jahr 1995 hat SAP verschiedene Produkte direkt bei den Herstellern evaluiert. Um in Walldorf selbst Produkte über einen längeren Zeitraum zur Verfügung zu haben, wurde von Prof. Wendt ein Software-Technologie-Labor eingerichtet. Die Einrichtung des Labors geschah im Rahmen dreier Diplomarbeiten[2].

Neben der Beschaffung und Inbetriebsetzung der Hardware war eine Aufgabe die Installation und Untersuchung bestimmter Software-Produkte. Bei den untersuchten Produkten handelt es sich um Datenbanken, Programmiersprachen, Entwicklungswerkzeuge sowie einen Softwarebus.[3] Der Softwarebus dient zur Kommunikation zwischen Akteuren

---

[1] Zitiert aus [Rockwell]

[2] [Gröne], [Otto] und die vorliegende Arbeit.

[3] Eine Aufstellung der eingesetzten Hard- und Softwareprodukte findet sich in [Gröne].

eines über mehrere Rechner verteilten programmierten Systems. Er setzt auf dem TCP/IP-Protokoll auf.

Die Auswahl der Softwareprodukte und der Aufbau des Labors berücksichtigt die Tatsache, daß Erkenntnisse gewonnen werden sollen, die im Zusammenhang mit der Architektur des SAP R/3 Systems interessant sein können. Eigenschaften und Bedürfnisse des R/3-Systems sollten sich also im Labor widerspiegeln. Konkret sind hier die Drei-Schichten-Architektur sowie der hohe Vernetzungsgrad und die Plattformunabhängigkeit des R/3-Systems zu nennen.

Es wäre nicht sinnvoll gewesen, die einzelnen Produkte nur für sich allein zu betrachten. Aus diesem Grund wurde ein Beispielsystem entworfen. Die Untersuchung der Produkte erfolgte anhand der Realisierung von Teilen dieses Beispielsystems im Rahmen der genannten Diplomarbeiten.

Bei dem Beispielsystem handelt es sich um ein System zur Verwaltung von Videotheken. Eine Videothekenverwaltung eignet sich gut als Beispielsystem, weil sich damit einerseits die Forderungen nach einem heterogenen und verteilten System mit Anschluß an unterschiedliche Datenbanken gut abdecken lassen und andererseits der Umfang der bereitzustellenden Funktionalität überschaubar bleibt. Außerdem sollte das Beispiel aus einem allgemein bekannten Bereich stammen.[1]

Ein grober Entwurf des Beispielsystems liegt vor. Die Implementierung ist allerdings noch weit von der Fertigstellung entfernt. Das Beispielsystem hat sich als Rahmen und Zielvorstellung beim Experimentieren mit den Produkten jedoch als sinnvoll und nützlich erwiesen. Die bereits fertigen Elemente können als Basis für eine Fortsetzung der Implementierung der Vidothekenverwaltung verwendet werden. Zum anderen können sie als Vorlage und Beispiel für ganz andere experimentelle Systeme dienen. Viele Anfangsschwierigkeiten werden die gleichen sein und lassen sich am einfachsten durch die Untersuchung eines Beispiels überwinden.

Für eine Beschreibung der grundsätzlichen Architektur des Beispielsystems sei hier auf [Otto, Kapitel 3 bis Abschnitt 3.3.4 einschl.] verwiesen. In [Otto] wird auch der Rendezvous-Software-Bus vorgestellt, der eine zentrale Rolle im Labor spielt. In der vorliegenden Arbeit wird die grundsätzliche Funktionsweise des Busses nicht eigens erklärt, so daß die Lektüre zumindest des genannten Abschnitts bzw. von [Auer] empfohlen wird.

Es sind bisher noch keine Komponenten realisiert, die die Verbindung des Beipielsystems zu Datenbanken herstellen. Sie sind jedoch vorgesehen und die nötigen technischen Voraussetzungen sind gegeben.[2] In [Gröne] werden unter anderem mehrere objektorientierte Datenbanken untersucht und miteinander verglichen.

---

[1] Eine Bibliotheksverwaltung oder andere ähnliche Systeme wären genauso gut in Frage gekommen.

[2] Verschiedene Datenbanken wurden im Labor installiert, Softwarekomponenten für den Anschluß der Datenbanken an Programmiersprachen sind vorhanden. Im Fall von Eiffel ist das die Bibliothek *Eiffel Store*.

Während der Schwerpunkt von [Gröne] auf den Datenbanken liegt, beschäftigt sich [Otto] mit der Erstellung einer Experimentierumgebung auf der Basis der Programmiersprache Smalltalk.

Der Schwerpunkt der vorliegenden Arbeit ist die Programmiersprache Eiffel. Die Charakteristika dieser Sprache und einige ihrer Konzepte werden in Kapitel 2 beschrieben. Der bereits genannte Softwarebus mußte an die Sprache Eiffel angeschlossen werden, wozu eine Untersuchung der C-Sprachschnittstelle von Eiffel Voraussetzung war. Der Beschreibung dieser Schnittstelle und des Busanschlusses ist Kapitel 3 gewidmet.

Nicht zuletzt wegen der starken Gewichtung von Objektorientierung, Heterogenität und Verteilung im Softwarelabor ist die Sprache Java von Interesse. Sie wird im vierten Kapitel vorgestellt.

2 Eiffel

Eiffel ist eine objektorientierte Programmiersprache, die in den vergangenen Jahren von Bertrand Meyer entwickelt worden ist. Es handelt sich um eine zu übersetzende Sprache, die prozedural ist und nebenläufige Programmabwicklung noch nicht unterstützt. Es gibt mehrere Hersteller von Entwicklungsumgebungen für Eiffel und ein Konsortium[1], das den Sprachstandard definiert.

2.1 Warum Eiffel?

Es gibt eine ganze Anzahl von objektorientierten Sprachen, die teilweise weit verbreitet sind. Damit stellt sich die Frage nach der Berechtigung einer weiteren, neuen Sprache. Sind die vorhandenen Sprachen nicht genug oder nicht gut genug?

Zumindest Bertrand Meyer würde den letzten Teil der Frage mit „Nein“ beantworten. Er hat mit dem Entwurf der Sprache das Anliegen verfolgt, ein Werkzeug zu entwickeln, das ein hohes Potential zur Verbesserung von Software-Qualität bietet.

Objektorientierung wird als ein geeignetes Mittel für eine bessere Modularisierung und leichtere Wartbarkeit von Software sowie für Mehrfachverwendung von Software-komponenten betrachtet.

Die heute wohl am häufigsten industriell eingesetzte übersetzte Sprache dürfte C++ sein. Diese Sprache hat jedoch insbesondere den Nachteil, daß sie eine Erweiterung von C ist und objektorientierte Konzepte nicht konsequent umgesetzt sind.

In Eiffel konnten diese Konzepte ohne Rücksicht auf die Vergangenheit umgesetzt werden. Dies trifft auf objektorientiertes Pascal, C++ und auch auf Java[2] nicht zu.

---

[1] Nonprofit International Consortium for Eiffel (NICE)

[2] aufgrund der (absichtlichen) Anlehnung an C und C++

Smalltalk dagegen ist ebenfalls eine Sprache ohne direkte Vorgänger. Smalltalk und Eiffel unterscheiden sich aber insbesondere darin, daß es sich bei Smalltalk um eine interpretierte Sprache mit entsprechenden Geschwindigkeitsnachteilen handelt. Eiffel wird von Bertrand Meyer in Bezug auf die Geschwindigkeit ausdrücklich in Konkurrenz zu C gesehen.

Die Sprache Eiffel sollte unter anderem folgende Kriterien erfüllen:

- Die Sprache soll eine einfache Syntax haben und mit wenigen Sprachmitteln auskommen.

- Die Sprache soll schnell sein und sich für den industriellen Einsatz eignen.

- Die Sprache muß objektorientiert sein.

- Es soll durch die Technik des Mehrfacherbens erleichtert werden, Software aus vorgefertigten Einzelteilen zusammenzubauen. Meyer begründet, warum herkömmliche Modularisierungstechniken, beispielsweise die Verwendung von Unterprogramm-Bibliotheken, ungenügend sind, und warum Mehrfacherben ein geeignetes Modularisierungskonzept ist.[1]

- Mit Hilfe der Sprache sollen auch Spezifikationen ausdrückbar sein. Das heißt, es soll Softwareelemente geben können, die Vorgaben für später zu implementierende Elemente machen, sich aber nicht in abwickelbaren Code übersetzen lassen. Man beachte den Zusammmenhang mit der doppelten Funktion von Software: Es handelt sich hierbei um Elemente, die in erster Linie der Kommunikation zwischen Menschen dienen.

- Das Laufzeitsystem der Sprache soll unbenutzte Objekte automatisch freigeben. Ein Kennzeichen aller objektorientierten Systeme ist ein hohes Maß an Strukturvarianz. Beim Betrieb des Systems werden laufend neue Objekte erzeugt. Nicht mehr benötigte Objekte beanspruchen Speicherplatz, der wieder freigegeben werden sollte. Die Speicherbereinigung ist daher noch problematischer als bei nicht objektorientierten Sprachen. Mit dem Wegfall der Notwendigkeit, sich um die Speicherfreigabe zu kümmern, fallen natürlich die in diesem Bereich reichlich vorhandenen Quellen für Programmierfehler weg. Da die Problematik der Speicherfreigabe bei fast allen Programmen auftritt, ist die Entscheidung, sie ein für alle mal zu lösen, auch ökonomisch sinnvoll.

- Soweit möglich, sollen Fehler bereits zum Zeitpunkt der Übersetzung entdeckt werden. Durch eine strenge Typbindung stellt Eiffel insbesondere sicher, daß es unmöglich ist, eine Methode eines Objekts aufzurufen, die dieses Objekt gar nicht besitzt.

---

[1] vgl. [Meyer, insbesondere Kapitel 4.9]

Schmale Schnittstellen

Eine der größten Schwierigkeiten bei der Programmierung großer Systeme ist die Beherrschung von Komplexität. Eine Lösung besteht darin, Systeme in Subsysteme zu zerlegen, die für sich verständlich sind. Dies macht aber nur Sinn, wenn die Schnittstellen zwischen den Subsystemen schmal sind. Man muß ein Subsystem als Black Box beschreiben können. Diese Beschreibung als Black Box ist gleichbedeutend mit der Beschreibung der Schnittstelle dieses Subsystems mit der Außenwelt. Eine solche Beschreibung stellt ein Verhaltensmodell[1] dar. Benutzt man zur Beschreibung der Box dagegen die Struktur des Innenlebens, so handelt es sich um ein Aufbaumodell. Wenn nun die Beschreibung des Verhaltens einer Box genauso kompliziert ist wie die Beschreibung ihres Aufbaus, dann hat man nichts gewonnen. Zur Komplexitätsreduktion trägt eine Zerlegung in Subsysteme also nur dann bei, wenn die Schnittstellen einfach zu beschreiben, also „schmal“ sind.

Dem Ziel, schmale Schnittstellen zu benutzen, dienen mehrere Konstrukte in Eiffel:

- Wie bei anderen Programmiersprachen auch werden die Daten eines Objekts gekapselt, d.h. nur das Objekt selbst hat Zugriff auf diese Daten. Ein Zugang von außen ist nur über die durch die Zugriffsroutinen[2] des Objekts gegebene Schnittstelle möglich. Die Benutzungsberechtigung von Merkmalen[3] kann darüber hinaus auf Objekte bestimmter Klassen begrenzt werden.

- Neu bei Eiffel ist das Vertrags-Paradigma[4]. Die konsequente Verwendung dieses Paradigmas zwingt die Benutzer und Programmierer von Softwareelementen dazu, einen sie gegenseitig bindenden Vertrag einzuhalten. Auf dieses Thema wird in Abschnitt 2.4 noch näher eingegangen.

Sprachbeschreibung

Eiffel bietet in Bezug auf die Beschreibung der Sprache eine außergewöhnliche Qualität. Mit den Büchern [Meyer], [ETL] und [EiffelLib] liegt eine Beschreibung der der Sprache zugrundeliegenden Konzepte und derer Umsetzung vor, die auf diesem Niveau selten zu finden ist. Die Sprachkonstrukte erscheinen nicht als vom Himmel gefallen, sondern sie werden gründlich und ausführlich begründet. Ein Eindruck, der durchaus nicht von allen Sprachen erweckt wird.

---

[1] vgl. [Wendt]

[2] In Eiffel ist auch der direkte *lesende* Zugriff auf Attribute möglich.

[3] Zu der Bedeutung von „Merkmal“ im Eiffel-Kontext siehe Abschnitt 2.3.4.

[4] engl.: Programming by Contract, siehe [Meyer]

Resümee

Zusammenfassend läßt sich sagen, daß mit Eiffel einige neue Ideen verbunden sind, deren Verfolgung gerade auch im Zusammenhang mit Problemen bei der Programmierung so komplexer Systeme wie dem R/3-System sehr interessant sein können.

Eiffel legt ein Programmiermodell nahe, das zumindest teilweise auch bei der Verwendung anderer Programmiersprachen verwendbar ist. Damit ist Eiffel meines Ermessens auch dann interessant, wenn eine Programmierung in Eiffel nicht in Frage kommt.

2.2 Weitere Kennzeichen von Eiffel

Nicht nur der Vollständigkeit halber, sondern auch um die Sprache einordnen zu können, werde ich im folgenden einige weitere Kennzeichen von Eiffel vorstellen:

- Häufig wird „prozedural“ und „objektorientiert“ als ein Gegensatzpaar benutzt. Dies ist so jedoch nicht richtig, da es sich bei der Objektorientierung um ein Modularisierungskonzept handelt. So wie es prozedurale und funktionale Sprachen gibt, die nicht objektorientiert sind, gibt es auch unter den objektorientierten Sprachen prozedurale und funktionale Sprachen. CLOS (objektorientiertes Lisp) ist ein Beispiel für eine objektorientierte funktionale Sprache.

  In diesem Sinn ist Eiffel eine objektorientierte prozedurale Sprache. Die Methoden in Eiffel sind nichts anderes als die Funktionen und Prozeduren eines rein prozeduralen Programms.

- Eiffel ist eine Sprache, die (noch) keine nebenläufige Programmabwicklung zuläßt[1]. Dies führt dann zu Problemen, wenn auf Ereignisse aus unabhängigen Quellen reagiert werden soll. Die nebenläufige Programmabwicklung war in Eiffel von vornherein beabsichtigt, sie ist bisher nur weder in der Sprachbeschreibung noch in der Implementierung realisiert worden. Die Neuauflage von [Meyer] wird sich mit diesem Thema beschäftigen. Es ist damit zu rechnen, daß auch die Definition und die Implementierung der Sprache in den kommenden Jahren nachziehen werden.

- Wie bereits erwähnt, müssen Eiffel-Programme vor ihrer Abwicklung übersetzt werden. In einem Zwischenschritt wird Eiffel dabei in die Sprache C übersetzt, um die Portierung von in Eiffel geschriebenen Programmen auf unterschiedliche Rechner zu ermöglichen. C-Übersetzer stehen für eine große Zahl von Rechnern zur Verfügung.

[1] Das heißt, es gibt für jedes Eiffel-System genau einen virtuellen Eiffel-Abwickler ohne Multiplexbetrieb. Eiffel wird übersetzt und nicht interpretiert, dennoch ist die Vorstellung eines virtuellen Abwicklers sinnvoll, vgl. Abschnitt 2.3.8.

2.3 Eiffel-Grundlagen, Begriffe

Die Kenntnis der grundsätzlichen Prinzipien der Objektorientierung wird hier vorausgesetzt. Da es aber große Unterschiede gibt in dem, was unter Objektorientierung verstanden wird, und auch die Terminologie nicht immer dieselbe ist, will ich in beschränktem Umfang die der Sprache Eiffel und dieser Arbeit zugrundeliegende Sicht der Dinge vorstellen.

2.3.1 Begriffe im natürlichen und im Eiffel-Kontext

Insbesondere dort, wo Softwareelemente Dinge aus der realen Welt simulieren, kann es Verwirrung um den Begriff der Klasse geben.

Als Beispiel betrachte man ein Programm zur Verwaltung einer Videothek. Ein naheliegender Gedanke ist, im Programm für jede Videokassette ein Objekt vorzusehen. Daraus könnte sich die Frage ergeben, ob mit der „Klasse der Videokassetten“ die Klasse der Videokassetten im Regal der Videothek oder die Klasse der Objekte im Speicher eines Rechners, die diese Kassetten repräsentieren, gemeint ist. In der Regel ist es dann sinnvoll, die Objekte im Speicher als „Videokassettenverwalter“ zu bezeichnen.

Daß diese Unterscheidung hilfreich sein kann, wird deutlich, wenn man an solche Funktionen wie „drucken“ denkt. Für eine druckende Videokassette braucht es schon eine lebhafte Phantasie. Ein Videokassettenverwalter, der einen Ausdruck der Kassettendaten erstellt, ist wesentlich realistischer.

Der Gegenstand dieses Kapitels ist die Programmiersprache Eiffel. Wenn in diesem Zusammenhang Aussagen über Objekte, Klassen, Merkmale usw. gemacht werden, so sind in der Regel Entitäten aus der Eiffel-Welt gemeint. Naturgemäß haben die Begriffe in diesem Kontext eine eingeschränkte Bedeutung im Vergleich zur realen Welt. Es wäre falsch zu behaupten: „Alle Attribute sind Speicherplätze“ ohne diese Behauptung einzuschränken auf die spezielle Bedeutung des Begriffs Attribut im Eiffel-Kontext. Man sollte sich dieses Unterschiedes bewußt bleiben, auch wenn kein neuer Begriff (oder eine Abkürzung) für „Attribut im Eiffel-Sinn“ gesucht wurde.

Dies gilt insbesondere auch für Objekte. Wenn im folgenden von Objekten geredet wird, dann sind in der Regel „Eiffel-Objekte“ gemeint. „Eiffel-Objekte“ sind eine Unterklasse der allgemeinen Objekte. Diese Unterklasse ist dadurch gekennzeichnet, daß ihre Exemplare Komponenten eines in Eiffel programmierten Systems sind oder sein könnten.

2.3.2 Programm und System

Die Redeweise „ein Programm tut dies oder jenes“ ist weitverbreitet. Genaugenommen ist aber ein Programm niemals ein zu einer Handlung fähiger Akteur, sondern nur eine Verhaltensbeschreibung für den Programmabwickler.

Erst durch das Zusammenfügen eines Programms mit einem zu dessen Abwicklung fähigen Rechner entsteht ein System. Dieses System ist dann in der Lage, beobachtbares Verhalten zu zeigen.

Zerlegt man ein System gedanklich in Akteure, dann lassen sich manchmal Akteure bestimmten Programmkomponenten zuordnen. In diesem Fall wird hier stellenweise die Bezeichnung der Programmkomponente auch für den Akteur benutzt.

In einem Programm ist häufig eine Beschreibung des Abwicklers und damit eine Beschreibung der Hardware impliziert. Mann kann dann ein Programm auffassen als Systembeschreibung.

2.3.3 Objekte und Klassen

Ein objektorientiert programmiertes System stelle man sich vor als eine Menge von Objekten, die untereinander Informationen austauschen und sich gegenseitig Aufträge erteilen. Die Objekte sind geordnet durch ihre Zugehörigkeit zu Klassen. Siehe hierzu und zu den folgenden Abschnitten Bild 2.1. Dieses Bild zeigt die Entitäten eines Eiffel-Systems und ihre Beziehungen untereinander.

Im Bild unten links sieht man die Entität „Objekt“. Jedes Objekt läßt sich eindeutig einer bestimmten Klasse zuordnen, deren direktes[1] Exemplar es ist. Es gibt eine Eins-zu-eins-Zuordnung zwischen Klassen und ihren Beschreibungen. Ein Eiffel Programm besteht aus den rechts im Bild gezeigten Klassenbeschreibungen und der in der Mitte liegenden Systembeschreibungsdatei. Auf diese Datei wird in Abschnitt 2.3.8 noch eingegangen, zunächst soll das Augenmerk auf den oben gezeigten Merkmalen liegen.

Ein Objekt zeichnet sich aus durch eine Menge von sog. Merkmalen. Alle direkten Exemplare einer Klasse zeichnen sich dadurch aus, daß sie die gleiche Menge von Merkmalen haben. Jedes Merkmal kann eindeutig einer Klasse zugeordnet werden. Deshalb ist auch keine Relation zwischen Merkmalen und Objekten eingezeichnet worden. Eine Klasse kann eine beliebige Zahl von Merkmalen haben, es kann auch Klassen ohne Merkmale geben.

Die Klassen stehen untereinander in einer Vererbungs- und einer Kundenrelation. Diese beiden Relationen werden im Abschnitt 2.3.6 noch erläutert werden.

---

[1] Zu „*direktes* Exemplar“: siehe Abschnitt 2.3.6 über Typmehrdeutigkeit.

In einem Eiffel-Programm sind nicht Objekte, sondern Klassen beschrieben. Es hat sich die Sprechweise eingebürgert, „Klassen zu schreiben“. Dies ist genaugenommen falsch, da man beim „Klassen schreiben“ eigentlich Baupläne für die Exemplare einer Klasse erstellt. Solange man sich dieser Tatsache bewußt bleibt, sind Mißverständnisse allerdings unwahrscheinlich. Aus Bequemlichkeitsgründen werde ich mich gelegentlich der Redeweise anschließen und vom Editieren von Klassen reden.

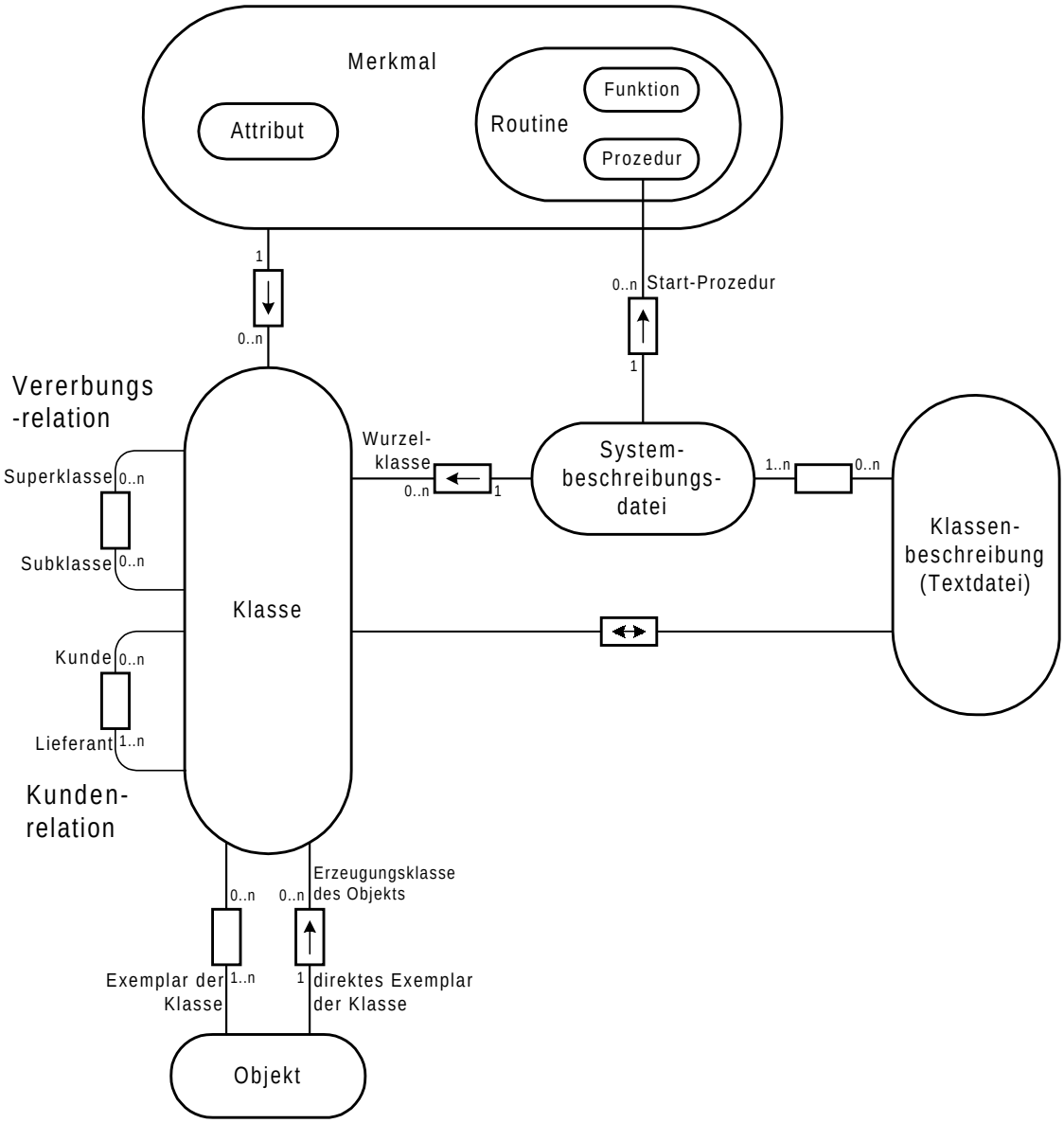


Bild 2.1 Entitäten eines Eiffel-Systems

2.3.4 Merkmale

2.3.4.1 Klassifikation aus Sicht des Implementierers

Man betrachte Bild 2.2.

Die Gesamtheit der Merkmale läßt sich partitionieren in Attribute und Routinen. Die Attribute sind Speicher für die Daten eines Objekts. Verändernd darf auf sie nur das Objekt selbst zugreifen. Routinen sind dadurch gekennzeichnet, daß sie sich abwickeln lassen.

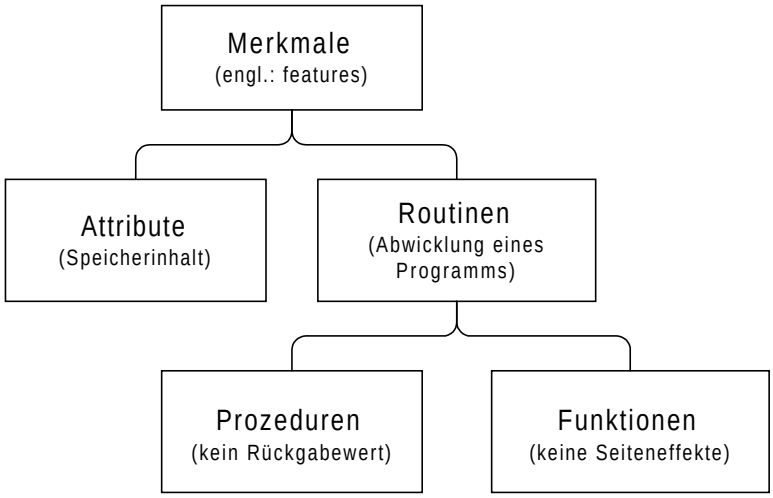


Bild 2.2 Klassifikation von Merkmalen aus Sicht des Implementierers

Attribute

Wenn oben gesagt wurde, daß alle Objekte einer Klasse die gleichen Attribute haben, dann bezog sich das nicht auf den Inhalt der Attribute. Dieser kann sich von Objekt zu Objekt unterscheiden. Es ist notwendig, zwischen einem Attribut und seinem Wert zu unterscheiden. So wie der Begriff Attribut hier gebraucht wird, kann man der Klasse der Spielfilme das Attribut „Dauer“ zuordnen. Der Inhalt oder Wert dieses Attributs, also zum Beispiel 157 min, ist dagegen nur einzelnen Objekten zugeordnet. Es gibt neben den normalen Attributen sog. Klassenattribute, das sind Attribute, die für alle Objekte einer Klasse immer den gleichen Wert haben. Beispielsweise könnte man das Attribut „Farbe“ für die Klasse der Schimmel als Klassenattribut auffassen. Auf Klassenattribute können alle Objekte der Klasse gleichberechtigt zugreifen.

Routinen

Die Gesamtheit der Routinen läßt sich partitionieren in Prozeduren und Funktionen. Prozeduren haben keinen Rückgabewert, sie verändern nur den Zustand eines Objekts. Funktionen dagegen liefern ein Ergebnis, dürfen aber prinzipiell keine Seiteneffekte haben.

Die Benutzung einer Funktion darf allerdings dann einen Seiteneffekt haben, wenn sich dieser nicht auf den von außen feststellbaren Zustand des Objekts auswirkt oder wenn die Zustandsänderung semantisch irrelevant ist.

Als Beispiel hierfür betrachte man eine Klasse komplexer Zahlen, die für die interne Darstellung umschaltbar entweder die kartesische oder die Polarform verwendet. An der Schnittstelle des Objekts nach außen wäre nicht sichtbar, welche interne Repräsentation gerade vorliegt. Die Benutzung einer die Polarform liefernden Zugriffsfunktion könnte als Seiteneffekt eine Umschaltung auf die interne Polarformdarstellung bewirken. Seiteneffekte dieser Art können sinnvoll sein und sollten nicht verboten werden. Der Eiffel-Übersetzer erzwingt daher nicht den Verzicht auf Seiteneffekte.

Merkmale sind zunächst Objekten zugeordnet. Dadurch, daß alle Exemplare einer Klasse die gleichen Merkmale haben, entsteht eine enge Bindung zwischen Merkmalen und Klassen. So ist es in Eiffel auch nicht möglich, einzelne Objekte selbst zu beschreiben. Aus diesem Grund wurde in Bild 2.1 auf die Darstellung einer direkten Relation zwischen Objekten und Merkmalen verzichtet. Auch wenn man nur ein einzelnes Objekt beschreiben will, ist man gezwungen, dafür eine eigene Klasse einzuführen. Das Objekt ist dann einziges Exemplar seiner Klasse. Handelt es sich bei dem Objekt um einen Akteur, so wird dieser Monopolakteur genannt.

Es wäre unsinnig, für jedes Objekt einer Klasse eine eigene Routinenimplementierung im Speicher des Rechners vorzusehen. Statt dessen wird eine einzige, der Klasse zugeordnete Implementierung benutzt, die aber immer im Kontext eines bestimmten Objekts ausgeführt wird. Grundsätzlich kann man an der Sicht festhalten, daß jedes Objekt *seine* Routinen hat. Es ist aber gelegentlich wichtig, das Wissen um die Implementierung der Routinen im Hinterkopf zu haben.

2.3.4.2 Klassifikation aus Sicht des Benutzers

Eine andere Klassifikation ergibt sich aus der Perspektive des Kunden (d.h. Benutzers) eines Objekts. Man betrachte dazu Bild 2.3. Merkmale lassen sich einteilen in solche, die einen Wert liefern und in solche, die keinen Wert liefern. Eine weitere Unterteilung erfolgt danach, ob ein Merkmal Parameter hat oder nicht.

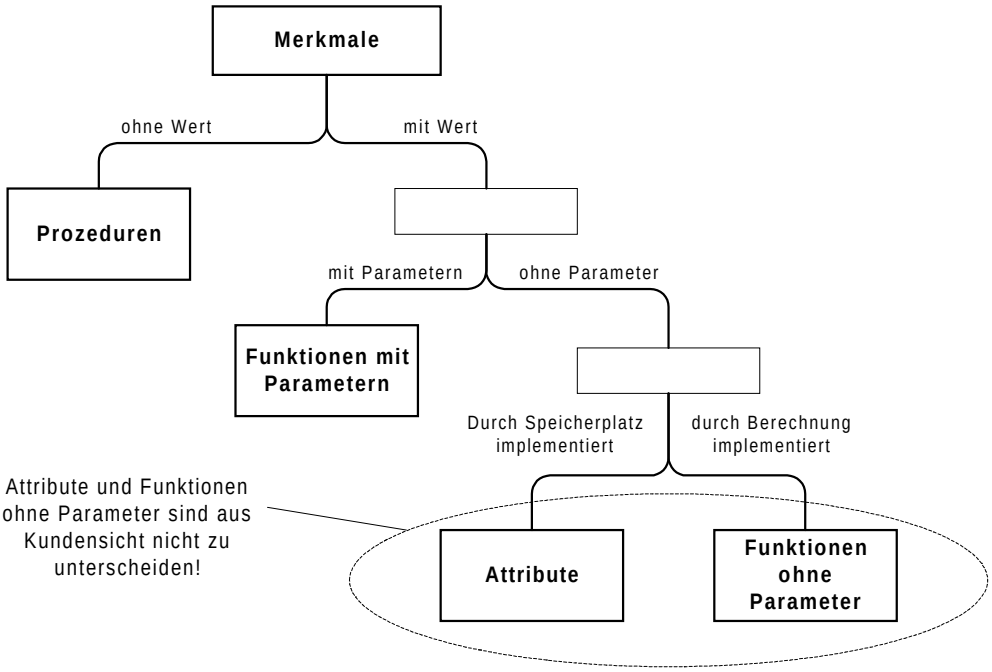


Bild 2.3 Klassifikation von Merkmalen aus Benutzersicht

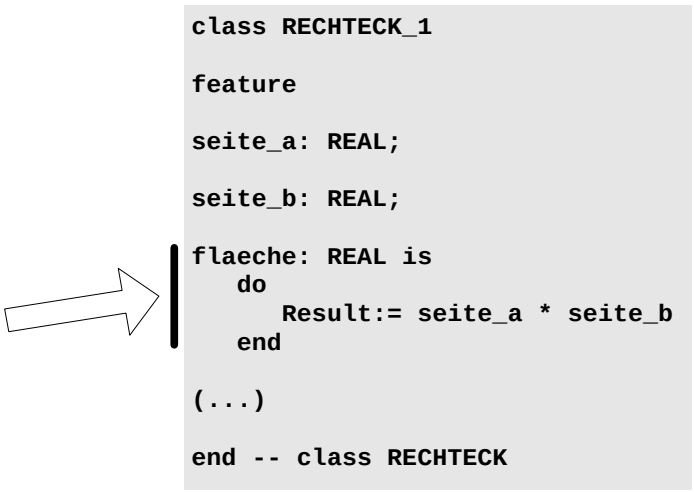
Es gibt offensichtlich zwei Arten von Merkmalen, die einen Wert, aber keine Parameter haben: Attribute und parameterlose Funktionen. Für den Benutzer eines Objekts sind beide syntaktisch nicht voneinander zu unterscheiden[1]. Bild 2.4 veranschaulicht diesen Sachverhalt an einem Beispiel.

2.3.5 Klassenakteure

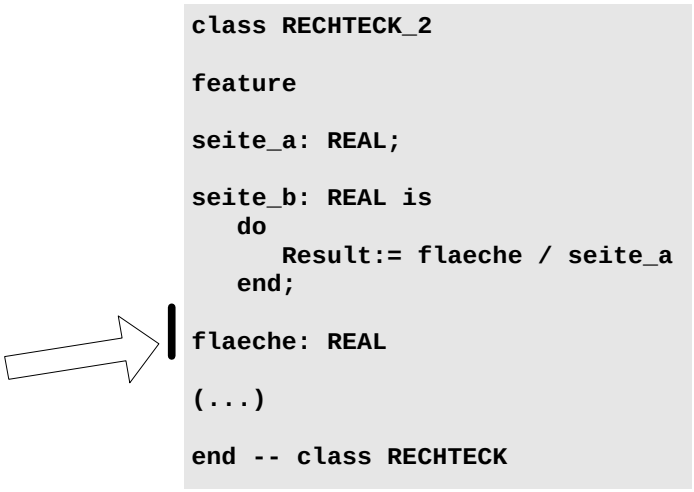
Im Zusammenhang mit objektorientierter Programmierung wird häufig nicht unterschieden zwischen Klassen und Klassenakteuren. Eine Klasse ist eine Gruppe von Objekten, die aufgrund gleicher Merkmale der Objekte gebildet wird. Eine Klasse ist damit niemals ein Akteur. Ein Klassenakteur ist dagegen ein Akteur eines objektorientiert programmierten Systems, der Aufgaben übernimmt, die im Zusammenhang mit einer ihm zugeordneten Klasse stehen.

In einigen Programmiersprachen werden automatisch für alle zum System gehörenden Klassen Klassenakteure erzeugt. Eiffel kennt im Unterschied dazu von sich aus keine Klassenakteure. Es gibt jedoch einen eleganten Weg, Monopolakteure zu erzeugen, die die Funktion eines Klassenakteurs wahrnehmen, dort, wo dies notwendig ist. Hierauf werde ich im Abschnitt 2.3.9 über Klassenattribute zurückkommen.

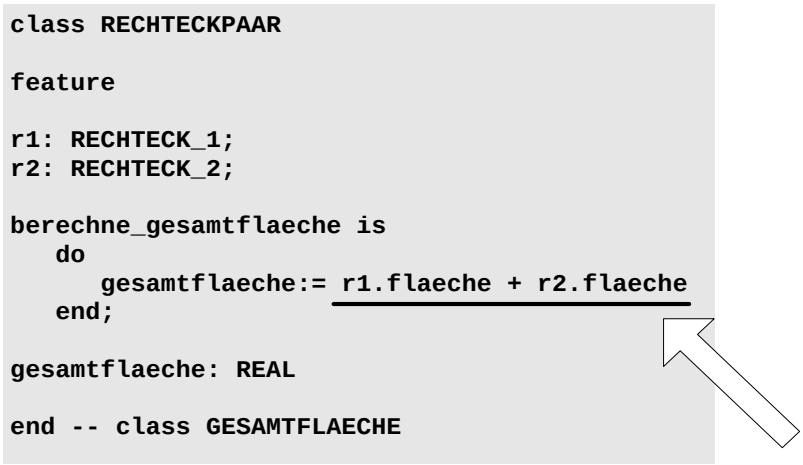
[1] Eine parameterlose Funktion kann deswegen durch ein Attribut redefiniert werden.



```
class RECHTECK_1

feature

seite_a: REAL;

seite_b: REAL;

flaeche: REAL is
   do
      Result:= seite_a * seite_b
   end

(...)

end -- class RECHTECK
```



```
class RECHTECK_2

feature

seite_a: REAL;

seite_b: REAL is
   do
      Result:= flaeche / seite_a
   end;

flaeche: REAL

(...)

end -- class RECHTECK
```



```
class RECHTECKPAAR

feature

r1: RECHTECK_1;
r2: RECHTECK_2;

berechne_gesamtflaeche is
   do
      gesamtflaeche:= r1.flaeche + r2.flaeche
   end;

gesamtflaeche: REAL

end -- class GESAMTFLAECHE
```

Bild 2.4 Der Aufruf einer Funktion ohne Parameter läßt sich syntaktisch nicht unterscheiden vom Lesen eines Attributs.

2.3.6 Vererbungsrelation und Typbindung

Nicht nur im Eiffel-Kontext erhält man durch die Klassifikation von Objekten eine Partialordnung der Klassen. Es handelt sich dabei um eine Abstraktionshierarchie. Die Klasse der Lebewesen ist eine Superklasse der Klasse der Säugetiere und damit abstrakter als diese. Alle Exemplare einer Klasse sind gleichzeitig auch Exemplare aller Superklassen dieser Klasse. Ein Elefant ist sowohl ein Säugetier als auch ein Lebewesen.

Im Bereich der Objektorientierung nennt man diese Relation über der Menge der Klassen Vererbungsrelation. Eine Klasse erbt von ihren Superklassen und wird von ihren Subklassen beerbt.

Für die Klassifikation der Eiffel-Objekte sind ihre Merkmale maßgeblich. Jede Klasse hat bestimmte Merkmale. Alle Objekte, die mindestens diese Merkmale besitzen, sind Exemplare der Klasse. Man betrachte zwei Klassen A und B. B sei Subklasse von A. Dann müssen die Exemplare von B mindestens alle Merkmale von A haben. Daraus folgt, daß eine erbende Klasse alle Merkmale ihrer Superklassen erbt und selbst nur neue Merkmale hinzufügen, aber keine entfernen kann.

Typmehrdeutigkeit

Es gibt eine 1:1-Abbildung zwischen Klassen und Typen[1]. Es ist äquivalent, zu sagen „Das Objekt O ist Exemplar der Klasse X.“ oder zu sagen „Das Objekt O ist vom Typ X“. Bei einer hierarchischen Klassifikation ist ein Objekt nicht eindeutig einer Klasse zuzuordnen. Ein Elefant ist sowohl ein Säugetier als auch ein Lebewesen, s.o. Diesen Sachverhalt bezeichnet man als Typmehrdeutigkeit[2]. Im Fall von Eiffel ist jedoch einem Objekt eindeutig ein konkretester Typ zugeordnet. Dieser ist bestimmt durch die Klasse, die bei der Erzeugung des Objekts angegeben wurde. Jedem Objekt ist damit eindeutig seine Erzeugungsklasse zugeordnet. Ein Objekt ist „direktes Exemplar“ seiner Erzeugungsklasse. Wenn im folgenden einfach von *„der* Klasse eines Objekts“ oder von *„dem* Typ eines Objekts“ geredet wird, dann ist damit die Erzeugungsklasse bzw. der konkreteste Typ gemeint.

Ein Objekt kann in Eiffel nicht migrieren, d.h. seine Klasse wechseln.

Strenge Typbindung

Um erklären zu können, was mit strenger Typbindung gemeint ist, will ich zunächst auf die Verwendung des Wortes „Größe“ im Eiffel-Kontext eingehen. „Größe“ ist dort ein

---

[1] Für den Gebrauch der Begriffe Typ und Klasse in Eiffel trifft das genaugenommen nicht zu. Einer generischen Klasse sind je nach Belegung der formalen Typparameter mehrere Typen zugeordnet. Weiterhin wird zwischen expandierten und nicht expandierten Typen einer Klasse unterschieden. Für die vorliegende Betrachtung kann das aber vernachlässigt werden.

[2] vgl. [Wiegert]

Sammelbegriff für lokale Variablen einer Routine, formale Argumente von Routinen und Attribute von Objekten[1]. Größen zeichnen sich dadurch aus, daß man ihnen einen Wert zuweisen kann.

Häufig ist ein solcher Wert ein Verweis auf ein Objekt. Eiffel ist streng getypt, und das heißt, daß jeder Größe ein Typ fest zugeordnet ist. Diese Zuordnung geschieht durch die Definition der Größe und liegt damit schon bei der Übersetzung des Programms fest. Die Größe darf nur einen Verweis auf ein Objekt ihres Typs oder auf ein Objekt eines Subtyps dieses Typs enthalten; der Typ einer Größe bestimmt den Wertebereich des Inhalts der Größe.

Die strenge Typbindung gilt nicht nur für Größen, sondern entsprechend auch für den Ergebnistyp von Ausdrücken. Dieser liegt ebenfalls schon bei der Übersetzung fest.

In der strengen Typbindung besteht ein deutlicher Unterschied zu typfreien Sprachen wie Smalltalk. Neben dem Ziel, Programmfehler bereits durch den Übersetzer zu finden, dient die Typbindung auch einer besseren Verständlichkeit des Programms. Jeder Größe ist direkt anzusehen, welche Klasse von Objekten sie enthalten kann.

Die Kundenrelation

In Bild 2.1 auf Seite 19 ist neben der Vererbungsrelation noch die Kundenrelation auf der Menge der Klassen zu sehen. Eine Klasse ist genau dann „Kunde“ einer anderen, „Lieferant“ genannten Klasse, wenn in ihrer Beschreibung Größen oder Ausdrücke mit dem Typ des Lieferanten vorkommen. Zu beachten ist, daß die Kundenrelation statisch, d.h. bereits bei der Übersetzung, bestimmbar ist.

Aufgeschobene Klassen

Die Kennzeichnung einer Klasse als „aufgeschoben“ bezieht sich auf die Implementierung von Merkmalen einer Klasse. Es ist möglich, Merkmale einer Klasse zu spezifizieren, ihre Implementierung aber Subklassen zu überlassen. Eine Klasse heißt dann aufgeschoben, wenn sie mindestens ein solches „aufgeschobenes“ Merkmal besitzt. Von aufgeschobenen Klassen können keine direkten Exemplare erzeugt werden, dies bleibt den Unterklassen vorbehalten.

Die Identifikation von Merkmalen

Man betrachte Bild 2.5. Am Bildrand rechts sind Merkmale dargestellt. Merkmale sind Klassen zugeordnet, hier sei an Bild 2.1 auf Seite 19 erinnert. In Bild 2.5 sind die Merkmale durch zwei Relationen mit den Klassen verbunden. Wenn eine Klasse ein bestimmtes Merkmal hat, dann haben auch alle Subklassen der Klasse dieses Merkmal. Das Merkmal ist

---

[1] Auch die vordefinierte lokale Variable *Result* zählt zu den Größen.

dann nicht nur einer, sondern mehreren Klassen zugeordnet, die in der Vererbungs-Partialordnung übereinander liegen.

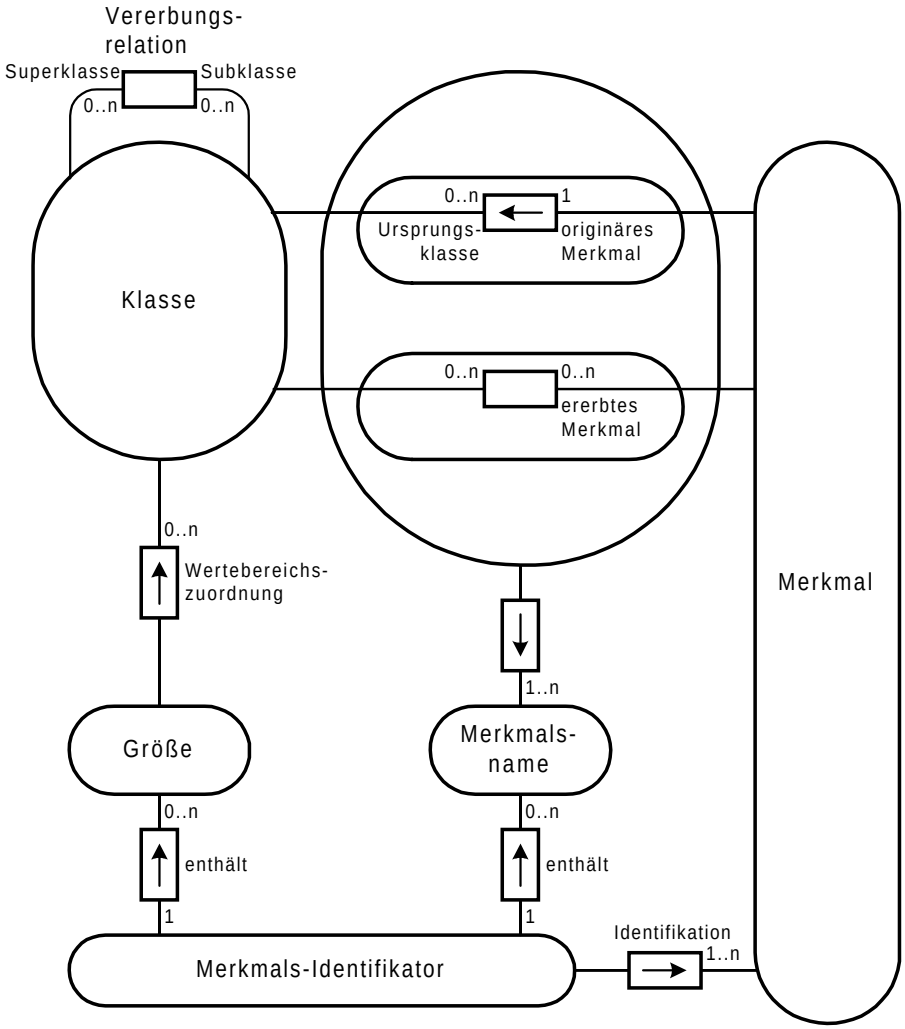


Bild 2.5 Identifikation von Merkmalen

Für jedes Merkmal läßt sich aber die abstrakteste der zugeordneten Klasse angeben. Diese Klasse wird „Ursprungsklasse des Merkmals“ genannt. Für sie ist das Merkmal ein „originäres Merkmal“. Diese Beziehung zwischen Merkmalen und Klassen wird durch die obere der beiden Relationen gezeigt. Alle Subklassen der Ursprungsklasse erben das Merkmal, für sie ist das Merkmal daher ein „ererbtes Merkmal“. Die untere Relation entspricht dieser Beziehung.

Beide Relationen drücken aus, daß das Merkmal ein Merkmal der Klasse ist, bzw. daß die Klasse das Merkmal enthält.

Das Merkmal einer Klasse ist für diese Klasse immer entweder eine originäres oder ein ererbtes Merkmal. Die Menge aller Paare aus Merkmal und das Merkmal enthaltender Klasse läßt sich daher in zwei Teilmengen partitionieren. Objektifiziert man die beiden

besprochenen Relationen, so wie dies im Bild getan wurde, dann erhält man genau diese disjunkten Teilmengen. Die Vereinigungsmenge der beiden objektifizierten Relationen ist genau die Menge aller Paare aus Merkmal und das Merkmal enthaltender Klasse. Das große Oval, das die beiden Relationen umschließt, stellt diese Vereinigungsmenge dar.

In Eiffel ist es möglich, jedem Merkmal *für eine Klasse* einen frei wählbaren Namen zuzuordnen. Die einzige Bedingung ist, daß nicht zwei Merkmale *einer Klasse* den gleichen Namen haben. Ein Merkmal ist in der Regel Merkmal mehrerer Klassen, dem Merkmal kann in jeder dieser Klassen ein anderer Name zugeordnet werden. Auch kann der gleiche Merkmalsname in mehreren Klassen für unterschiedliche Merkmale verwendet werden.

Daher ist im Bild ein Merkmalsname nicht direkt einem Merkmal zugeordnet, sondern es wird jedem der Klasse-Merkmals-Paare des großen Ovals eindeutig der darunter gezeigte Merkmalsname gegeben.

Für den Programmierer ist es notwendig, ein Merkmal identifizieren zu können. Ein typischer Fall ist der folgende: Eine Größe enthält ein Objekt, und der Programmierer möchte eine Prozedur des Objekts aufrufen. Dazu gibt der Programmierer die Bezeichnung der Größe an (beispielsweise „`bell`“) und durch einen Punkt getrennt einen Merkmalsnamen (beispielsweise „`ring`“, zusammen also: „`bell.ring`“).

Die Merkmalsidentifikatoren finden sich im ER-Diagramm am unteren Rand. Ein Identifikator besteht aus einem Paar aus Größe (genaugenommen: Größenbezeichner) und Merkmalsnamen. Das wird durch die beiden nach oben führenden Relationen ausgedrückt.

Wir haben gesehen, daß ein Merkmalsname allein nicht in der Lage ist, ein Merkmal zu identifizieren. Mit einem Paar aus Größe und Merkmalsname ist das dagegen möglich:

Jeder Größe ist per Definition eine Klasse und damit ein Typ zugeordnet. Die Größe kann Objekte von diesem Typ enthalten. Diese Wertebereichszuordnung ist im Bild links zwischen den Entitäten „Größe“ und „Klasse“ gezeigt.

Innerhalb einer Klasse müssen Merkmalsnamen eindeutig sein; das ist die oben bereits erwähnte Bedingung für die Wahl von Merkmalsnamen. Zusammen mit der Klasse ist damit die Identifizierbarkeit eines Merkmal gegeben.

Von den Merkmalsidentifikatoren führt eine Relation nach rechts, die zeigt, daß man von jedem Identifikator eindeutig auf ein Merkmal schließen kann.

Durch die im Merkmalsidentifikator enthaltene Größe läßt sich jedem Identifikator eine Klasse zuordnen. Das ist die Klasse, die den Wertebereich der Größe bestimmt.

Die freie Zuordnung von Merkmalsnamen ist durchaus sinnvoll. Denn der Benutzer einer Klasse braucht grundsätzlich nichts über andere Klassen zu wissen, er braucht nur die Merkmale seiner Klasse zu kennen. Für ihn spielt die Namenswahl in anderen Klassen keine Rolle.

Der Umbenennungsmechanismus

Die Zuordnung von Namen zu Merkmalen für eine Klasse erfolgt bei originären Merkmalen ganz trivial durch Benennung. Ererbte Merkmale behalten den Namen, den sie in der Superklasse haben, wenn sie nicht umbenannt werden.

Eine Umbenennung ist bei Namenskonflikten zwingend erforderlich. Namenskonflikte können zwischen ererbten und originären Merkmalen auftreten sowie zwischen mehreren ererbten Merkmalen beim Mehrfacherben. Ein Namenskonflikt ist dann und nur dann gegeben, wenn die Regel verletzt wird, daß zwei Merkmale einer Klasse nicht den gleichen Namen haben dürfen.

Der Umbenennungsmechanismus kann aber auch freiwillig benutzt werden, um aussagekräftigere oder passendere Namen zu wählen.

Das Enthaltensein von Objekten in Größen

In Bild 2.6 ist die Enthaltenseins-Relation zwischen Größen und Objekten gezeigt. Eine

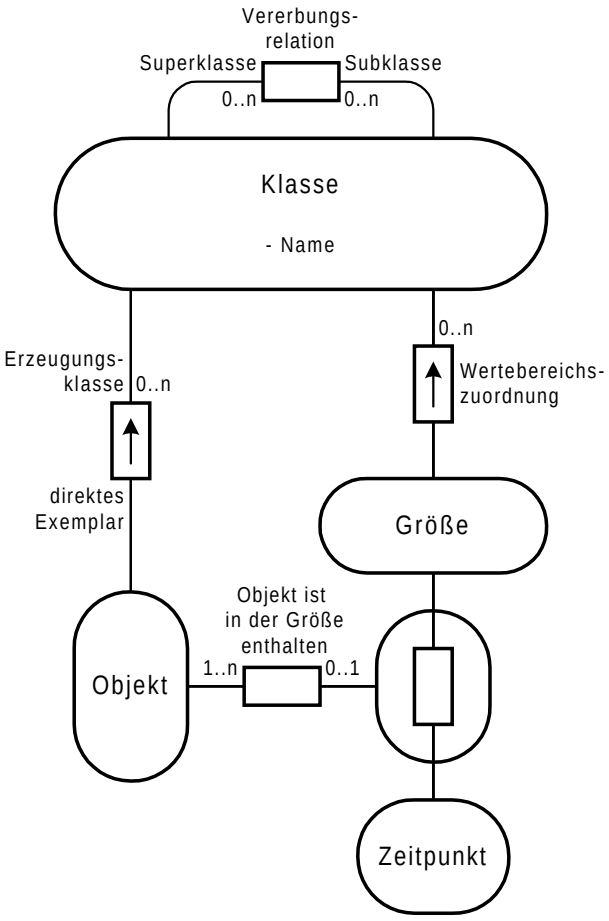


Bild 2.6 Das Enthaltensein von Objekten in Größen

Größe kann zu einem bestimmten Zeitpunkt während der Abwicklung eines Programms ein Objekt enthalten oder leer sein.

Genaugenommen enthalten Größen nicht Objekte, sondern nur Verweise auf Objekte. Dieser Unterschied ist aber im vorliegenden Kontext von untergeordneter Bedeutung.

Zusätzlich ist in Bild 2.6 gezeigt, daß jedem Objekt eindeutig seine Erzeugungsklasse zugeordnet ist. Das Objekt muß zum Wertebereich der Größe gehören.

Merkmale und ihre Implementierungen

Im Zusammenhang mit aufgeschobenen Klassen wurde schon darauf hingewiesen, daß ein Merkmal spezifiziert, aber noch nicht implementiert sein kann. Es wird daher unterschieden zwischen Merkmalen und ihren Implementierungen. Bild 2.7 zeigt einen Ausschnitt aus dem bereits bekannten ER-Diagramm. Zusätzlich ist rechts unten die Entität „Implementierung“ zu finden.

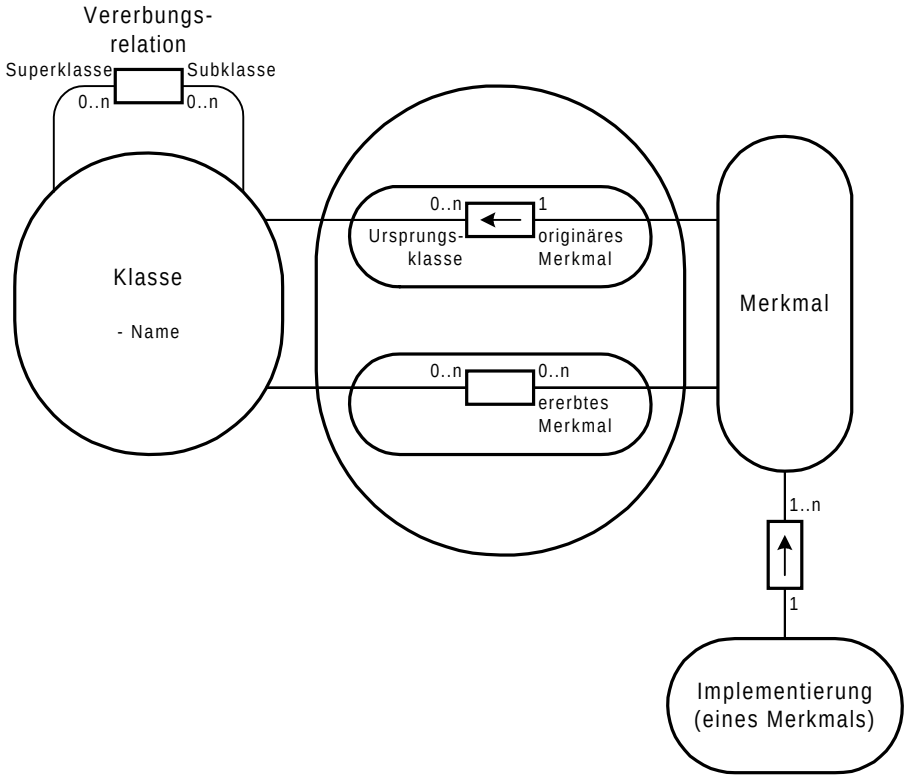


Bild 2.7 Merkmale und ihre Implementierungen

Bei der Deklaration eines aufgeschobenen Merkmals werden dem Kunden einer Klasse alle Informationen gegeben, um das Merkmal benutzen zu können. Mit der Implementierung wird dafür gesorgt, daß es für den Abwickler auch tatsächlich möglich ist, das zu tun, was der Kunde aufgrund dieser Spezifikation verlangt.

Will man ein tatsächlich abwickelbares System erhalten, dann muß jedem Merkmal irgendwann einmal eine Implementierung gegeben werden. Die obere Kardinalität der Relation zwischen Implementierungen und Merkmalen ist daher *eins* bis n.

Zu einer Spezifikation sind unterschiedliche Implementierungen denkbar. Dementsprechend können einem Merkmal mehrere Implementierungen zugeordnet werden. In Bild 2.7 ist dies gezeigt. Wir haben bereits gesehen, wie es für einen Programmierer möglich ist, ein *Merkmal* zu identifizieren. Er identifiziert damit eine Spezifikation, aber noch nicht die *Implementierung* eines Merkmals.

Es stellt sich somit die Frage, wie bei der Programmabwicklung die zu benutzende Implementierung bestimmt wird. Tatsächlich erfolgt die Auswahl erst zur Laufzeit des Programms.

Man betrachte dazu Bild 2.8. In diesem Bild wurden die bereits vorgestellten Bilder 2.5 bis 2.7 zusammengesetzt. Neu ist lediglich die Entität „Merkmalsbenutzung“ unten in der Mitte.

Mit Merkmalsbenutzung ist die Verarbeitung eines Merkmals durch den Abwickler gemeint. Bei Merkmalen mit Wert ist das die Auswertung, bei Prozeduren ist es die Abwicklung der Prozedur.

Die Merkmalsbenutzung findet zu einem bestimmten Zeitpunkt statt, was durch obere der beiden nach links führenden Relationen beschrieben ist. Die Merkmalsbenutzung findet statt, wenn der Abwickler auf einen Merkmalsidentifikator stößt, deshalb kommt man über die nach oben führende Relation eindeutig auf einen Merkmalsidentifikator. Zu diesem gehört eine Größe, so daß der Merkmalsbenutzung genau eine Größe zugeordnet werden kann.

Da die Merkmalsbenutzung zu einer bestimmten Zeit erfolgt, liegt auch das Objekt fest, das die Größe zu diesem Zeitpunkt enthält. Daher ist jeder Merkmalsbenutzung eindeutig ein Objekt zugeordnet. Eine Merkmalsbenutzung ist nicht möglich, wenn die Größe *kein* Objekt enthält. Wird sie dennoch versucht, so scheitert sie und es wird eine Ausnahme ausgelöst. Die rechte Kardinalität der unten links eingezeichneten Relation zwischen Merkmalsbenutzung und Objekt ist daher eins. Die Merkmalsbenutzung erfolgt im *Kontext* des Objekts, oder besser, der Semantik dieses Vorgangs entsprechend: Es wird ein Merkmal *des Objekts* benutzt.

Wir wissen noch immer nicht, wie bei der Merkmalsbenutzung die zu verwendende Implementierung auszuwählen ist. *Daß* diese Auswahl erfolgen muß, ist aber offensichtlich und wird durch die Relation zwischen den Entitäten „Merkmalsbenutzung“ und „Implementierung“ ausgedrückt. Der Vorgang der Zuordnung erfolgt erst zum Zeitpunkt der Merkmalsbenutzung, und zwar abhängig vom Objekt, dessen Merkmal benutzt wird.

Das Objekt hat mit Sicherheit das benutzte Merkmal. Denn es ist in der Größe enthalten, und ist daher vom Typ der Größe. Die Erzeugungsklasse des Objekts ist die Klasse der

Größe oder eine Subklasse davon. Sie muß daher auch alle die Merkmale haben, die die Klasse der Größe hat.

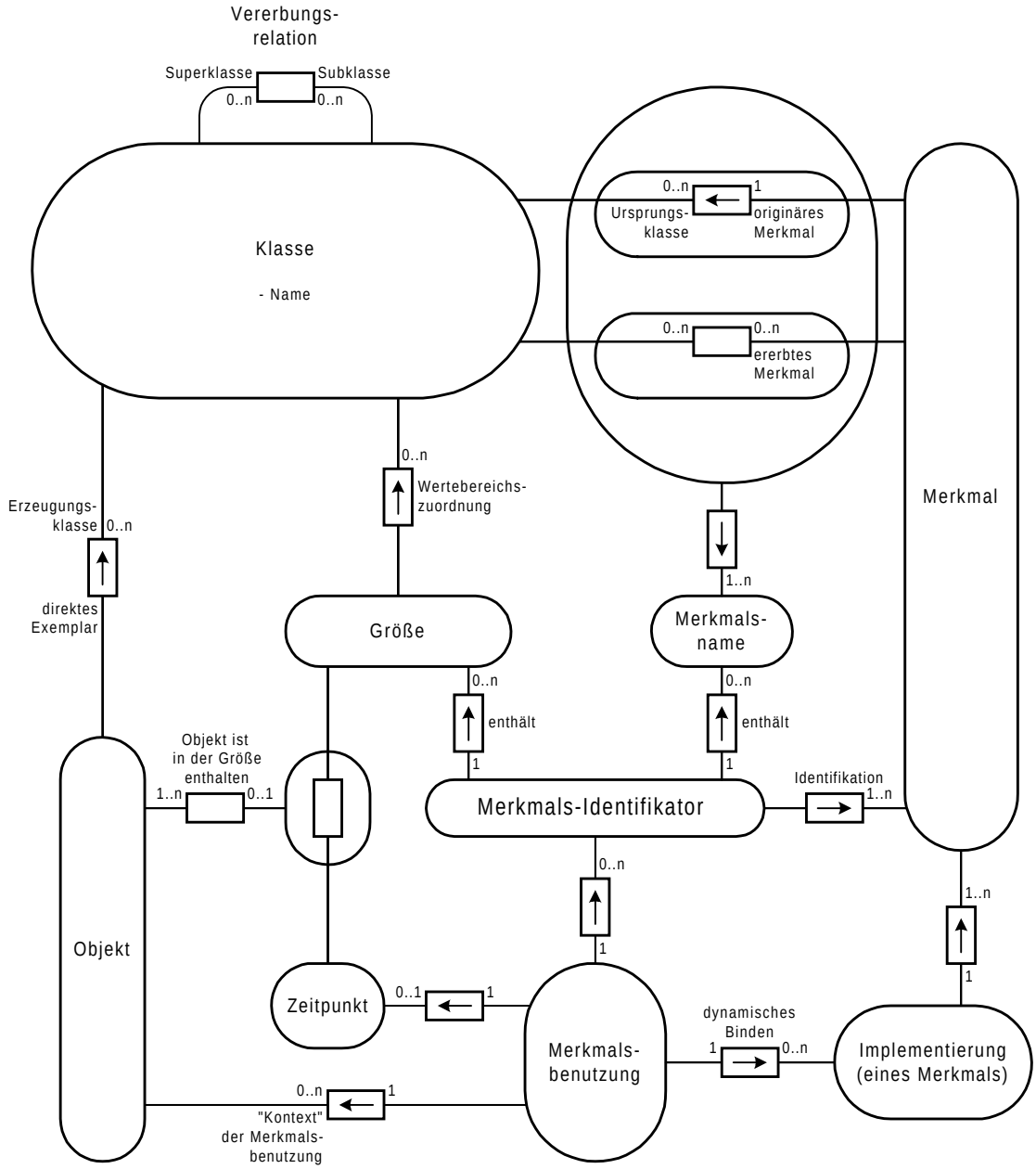


Bild 2.8 Merkmalsbenutzung

Jede Subklasse der Klasse der Größe kann für ein Merkmal eine eigene Implementierung vorsehen. Eine Implementierung ist immer genau einer Klasse zugeordnet. Und Klassenbeschreibungen enthalten immer die Information, welche Implementierung den Merkmalen der Klasse zuzuordnen ist.

Bei der Merkmalsbenutzung bestimmt also die Klasse des Objekts, dessen Merkmal benutzt wird, die zu benutzende Implementierung.

Da eine Größe im Laufe der Abwicklung eines Programms Objekte unterschiedlicher Klassen enthalten kann, ist es erst zur Laufzeit möglich, dem vom Programmierer geschriebenen Merkmalsidentifikator eine Implementierung zuzuordnen. Den Vorgang der Zuordnung nennt man *dynamisches Binden.*

2.3.7 Generische Klassen

Klassen können in Eiffel generisch sein. Bei „generischen Klassen“ handelt es sich um parametrisierbare Klassenbeschreibungen. Das heißt, daß man eine ganze Gruppe von Klassen durch Vorgabe eines Klassenmusters unter Verwendung formaler Parameter beschreiben kann. Damit ist eine Beschreibung einer Klasse von Klassen gegeben. Eine generische Klasse kann daher als Metaklasse aufgefaßt werden. Durch Ersetzen des formalen Parameters mit einer aktuellen Typ-Belegung erhält man dann die eigentliche Klasse. Dies dürfte am ehesten anhand eines Beispiels deutlich werden:

`LIST[T]` sei eine generische Klasse mit dem formalen Parameter `T`. `T` kann durch einen beliebigen Typ ersetzt werden, beispielsweise durch `INTEGER` oder `BUCHUNGSPOSTEN`. `LIST[INTEGER]` und `LIST[BUCHUNGSPOSTEN]` sind dann zwei eigenständige Klassen bzw. Typen, die wie ganz gewöhnliche Klassen oder Typen verwendet werden.

2.3.8 Abwicklung eines Programms

Bisher blieb die Frage offen, wie die Abwicklung eines Eiffel-Programms erfolgt. Ein Eiffel-Programm wird, wie bereits beschrieben, in die Maschinensprache des jeweiligen Rechners übersetzt. Dennoch ist die Vorstellung eines (virtuellen) Eiffel-Prozessors sinnvoll. Denn um Eiffel-Quelltext zu verstehen oder zu schreiben, muß man sich gleichsam in die Rolle dieses Abwicklers hineinversetzen[1].

Bei den durch den Programmierer geschriebenen Klassen handelt es sich lediglich um Baupläne von Objekten, sie führen für sich allein nicht zur Erzeugung irgendeines Objekts. Zu der Beschreibung eines Eiffel-Systems gehört neben den Klassen noch eine Systembeschreibungsdatei, in der unter anderem eine Wurzelklasse angegeben und eine Prozedur dieser Wurzelklasse als Startprozedur ausgewählt wird. Die Abwicklung beginnt mit der Erzeugung eines Exemplars der Wurzelklasse und dem Aufruf der angegebenen Prozedur dieses Objekts. Die Beendigung der Abwicklung dieser Prozedur ist gleichzeitig

[1] Dies trifft grundsätzlich auf alle übersetzten Sprachen zu.

die Beendigung der Abwicklung des Programms. Siehe hierzu nochmals Bild 2.1 auf Seite 19.

2.3.9 Klassenattribute

Klassenattribute sind Attribute, die einer Klasse als Gesamtheit zugeordnet werden, sie beziehen sich nicht auf einzelne Exemplare einer Klasse[1]. Eiffel kennt keine Klassenakteure, denen diese Attribute zugeordnet werden könnten. Zur Implementierung von Klassenattributen gibt es ein besonderes Konstrukt, die sog. Einmalfunktion[2]. Eine derartige Funktion ist formal und syntaktisch (bis auf das Schlüsselwort „once“) nichts anderes als eine normale Funktion, siehe Bild 2.9.

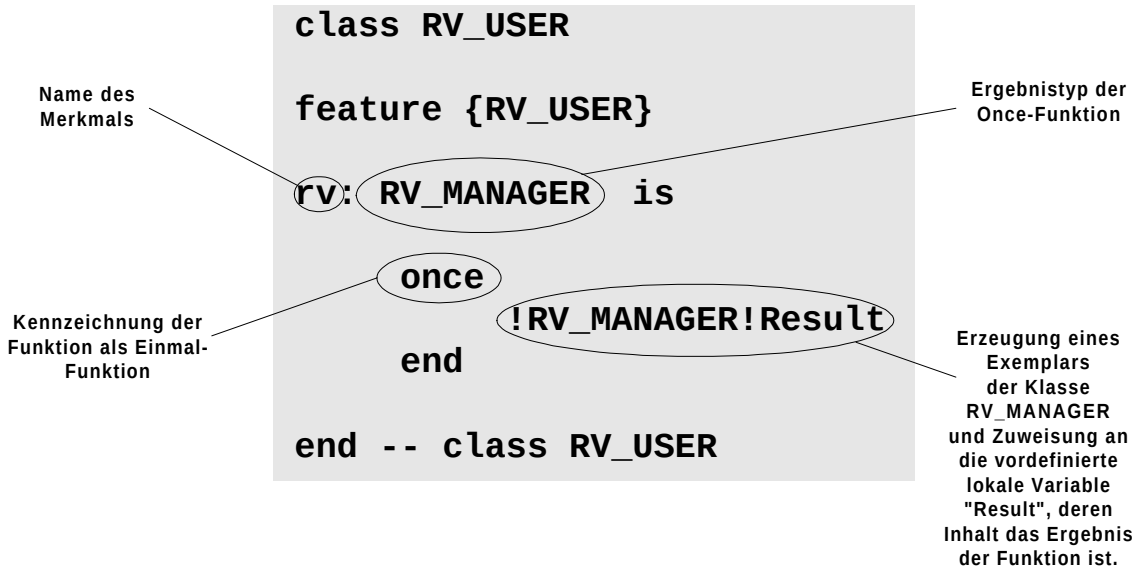


Bild 2.9 Syntaxbeispiel für eine Einmalfunktion

Die Implementierung einer Einmalfunktion wird jedoch nur maximal ein einziges Mal während der Laufzeit des Systems abgewickelt. Das Ergebnis dieser ersten Abwicklung wird gespeichert. Jede weitere Benutzung der Implementierung – auch im Kontext eines anderen Exemplars derselben Klasse – führt zur Rückgabe des gespeicherten Ergebnisses. Obwohl der *Zugriff* auf die Klassenattribute immer über einzelne Exemplare der Klasse erfolgt, sind es doch Attribute der gesamten Klasse. Unabhängig vom für den Zugriff gewählten Objekt liefert eine Einmalfunktion das gleiche, konstante Ergebnis.

---

[1] siehe den Abschnitt über Attribute auf Seite 20.

[2] engl.: once-function

Möglicherweise ist es aufgefallen, daß ich hier von *Implementierungen* von Funktionen gesprochen habe. Es ist bekannt, daß einem Merkmal durch Redefinition unterschiedliche Implementierungen zugeordnet sein können. Für Einmalfunktionen bedeutet dies, daß die Abwicklung nicht höchstens einmal für jede Funktion, sondern höchstens einmal für jede Implementierung der Funktion erfolgt. Unterschiedliche Implementierungen können unterschiedliche Ergebnisse liefern, auch wenn das leider im Widerspruch zu dem oben über Klassenattribute Gesagten steht.

Ein schreibender Zugriff ist auf Klassenattribute in Eiffel nicht möglich. Dies ist insofern konsequent, als schreibenden Zugriff auf Attribute stets nur ein einziger, für diese Daten zuständiger Akteur haben sollte. A priori kennt Eiffel keine Klassenakteure, womit sich kein Akteur für die Aufgabe des schreibenden Zugriffs anbietet.

Einmalfunktionen können dazu benutzt werden, Klassenakteure zu implementieren. Das Ergebnis einer Einmalfunktion kann ein Verweis auf ein Objekt sein. Dieses Objekt ist dann allen Exemplaren der Klasse bekannt und kann die Aufgaben eines Klassenakteurs übernehmen. Es kann selbst Attribute haben, auf die es dann natürlich auch schreibend zugreifen kann.

Bild 2.9 zeigt den Text der Klasse `RV_USER`. In der Kapselung des Rendezvous-Software-Busses ist der „Busverwalter“ als einziges Exemplar dieser Klasse vorgesehen[1]. Verschiedene Klassen nehmen Dienste dieses Busverwalters in Anspruch. Alle diese Klassen beerben die Klasse `RV_USER`. Sie erhalten dadurch das Merkmal „`rv`“[2], welches eine Einmalfunktion ist, die als Ergebnis einen Verweis auf den Busverwalter liefert.

2.3.10 Selektive Bereitstellung von Konstanten

In Eiffel gibt es keine globalen Variablen oder Konstanten. Wird der Zugriff auf bestimmte Dinge wie Monopolakteure oder Konstanten von einer Vielzahl von Klassen aus benötigt, möchte man sie dennoch an zentraler Stelle bereitstellen. Dadurch, daß in Eiffel das Mehrfacherben möglich ist, bietet es sich an, den Vererbungsmechanismus für die Lösung dieses Problems zu benutzen.

Es ist dann sogar möglich, Konstanten selektiv nur genau den Klassen zur Verfügung gestellt, die sie benötigen. Globale Konstanten sind so überflüssig, eventuell auftretende Namenskonflikte lassen sich mit dem Umbenennungsmechanismus lösen.

Bei der Kapselung des RV-APIs wurden verschiedene Konstanten, wie z.B. die maximal zulässige Länge von bestimmten Namen, in einer Klasse `RV_OBJECT` als Merkmal definiert. Alle Klassen, die diese Konstanten benötigen, erben von `RV_OBJECT`. Im Bild 2.10 läßt sich

---

[1] vgl. Abschnitt 3.3.4

[2] Für den Bus wird die Abkürzung „RV“ benutzt.

erkennen, daß die Menge der Unterklassen von RV_OBJECT nicht deckungsgleich mit der Menge der Unterklassen von RV_USER ist.

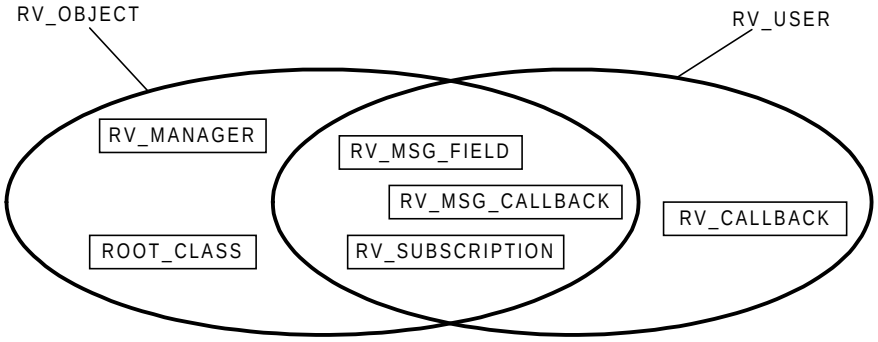


Bild 2.10 Selektive Bereitstellung von Konstanten und Monopolakteuren

2.4 Das Vertragsparadigma

Was haben Verträge mit Software zu tun? Beim „Programmieren mit Verträgen“ handelt es sich auch nicht um Verträge zwischen Software-Elementen, sondern letztlich um gewöhnliche Verträge zwischen Menschen. Das bedarf einer Erläuterung.

Die Mehrfachverwendung von Software-Komponenten ist ein wichtiges Ziel, wenn man die Herstellungsweise von Software verbessern will. Wieso sollte es nicht auch bei der Herstellung von Software möglich sein, Bausteine mehrfach zu verwenden?

Dahinter steht der Wunsch, Software ähnlich zu entwickeln, wie ein Hardware-Ingenieur, der bei der Konstruktion seines Produkts eingekaufte Bauteile kombiniert, statt das Rad stets neu zu erfinden. Bei einem derartigen Vorgehen spielt die Spezifikation der Bauteile eine wesentlich wichtigere Rolle als bei der Eigenfertigung. Denn bei der Fertigung aus Komponenten wird ein Teil der Entscheidungsmacht an den Komponentenlieferanten abgegeben. Innerhalb der Spezifikation des Bauteils darf dieser tun und lassen, was er will. Dies schafft einen Spielraum für den Lieferanten, der notwendig ist, um Verbesserungen im Herstellungsverfahren der Komponente zu erreichen.

Wichtig ist, zu erkennen, daß eine Spezifikation nicht in erster Linie die Beschreibung eines tatsächlich vorliegenden Bauteils, sondern Bestandteil eines Vertrags ist.

Ein Beispiel: Man stelle sich vor, daß der Zulieferer eines Automobilherstellers bisher nur Kurbelwellen mit einer Abweichung der Maße vom Sollwert von weniger als 3 µm geliefert hat. Es wäre jedoch ein grober Fehler, sich bei der Auswahl der Lager für die Wellen auf *diesen* Wert zu verlassen. Solange der Vertrag mit dem Lieferanten eine Toleranz von 10 µm zuläßt, ist der Lieferant ohne weiteres berechtigt, das Produktionsverfahren umzustellen, und zukünftig weniger präzis gefertigte Wellen zu liefern. Die tatsächliche Beschaffenheit der Wellen ist für die Wahl der Lager uninteressant, wichtig ist die Spezifikation der Wellen im Vertrag zwischen Hersteller und Lieferant.

Dieser Gedankengang mag trivial klingen, ich habe aber den Eindruck, daß er in der Softwarebranche besonders ungewohnt ist. Wenn es um Schnittstellenbeschreibungen geht, wird in Ermangelung einer hinreichenden Spezifikation häufig das fertige Produkt als Referenz mißbraucht. Dabei werden gerade bei Softwareprodukten häufig Änderungen vorgenommen.

Die Freiheit, die die Spezifikation einem Komponentenlieferanten läßt, ist wichtig für den Fortschritt im Herstellungsverfahren der Komponente. Würde man sich auf die tatsächliche Beschaffenheit eines Produkts verlassen, dann könnte ein Lieferant das Herstellungsverfahren nur nach einer vorherigen Absprache und Verhandlungen mit *allen* Kunden ändern. In der Praxis führte dies vermutlich dazu, daß Produktionsverfahren selten geändert würden und lieber parallel unterschiedliche Versionen eines Produktes hergestellt würden. Man würde in der Situation enden, in der sich heute die Softwarebranche befindet.

Das Ziel, Freiheitsgrade für die Komponentenerstellung zu bewahren, kann durch das Konzept der Datenkapselung in Objekten verfolgt werden. Das allein reicht aber nicht aus.

In der Kurbelwellen-Analogie bedeutete dies, daß der Hersteller das Verfahren zur Herstellung der Welle frei bestimmen darf, daß aber die Welle exakt so beschaffen sein muß, wie ein bestimmter Prototyp. Erst durch eine von Musterexemplaren unabhängige Spezifikation des Produkts erhält der Produzent die Freiheit, auch das Produkt selbst in bestimmten Grenzen zu verändern.

Zurück zu Eiffel. Eiffel bietet die Möglichkeit, Verträge zwischen den Herstellern von Softwarekomponenten und ihren Kunden mit Sprachmitteln auszudrücken. Dabei wird eine bestimmte Form von Verträgen unterstützt, was mit der Bereitstellung eines Rahmenvertrags vergleichbar ist. Eine Art allgemeiner Geschäftsbedingung für die Benutzung von Softwarekomponenten. Die Komponenten, über die Verträge abgeschlossen werden, sind Routinen und Objekte.

Zunächst zu den Verträgen über Routinen: Einer Routine kann eine „Vorbedingung“ und eine „Nachbedingung“ zugeordnet werden. Durch Angabe der Vor- und Nachbedingung kommt ein Vertrag zustande zwischen dem Programmierer der Routine und einem Benutzer der Routine, der folgendes besagt:

§1 Wenn der Benutzer sicherstellt, daß bei Aufruf der Routine die Vorbedingung erfüllt ist, dann hat der Programmierer der Routine dafür zu sorgen, daß nach Abwicklung der Routine die Nachbedingung erfüllt wird.

§2 Für den Fall, daß die Vorbedingung verletzt wird, ist der Routinenlieferant von jeder Verpflichtung frei.

Paragraph 2 hat weitreichende Konsequenzen. Denn grundsätzlich darf eine Routine, deren Vorbedingung verletzt wird, auch die Festplatte formatieren. Das wäre zwar vermutlich sittenwidriges Verhalten, der Vertrag an sich verbietet es aber nicht. Der Benutzer der Routine hat ohne Wenn und Aber dafür Sorge zu tragen, daß die Vorbedingungen erfüllt werden. Eine derartige Regelung hat durchaus Vorteile. Denn warum sollte der Kunde weniger dazu in der Lage sein, Vorbedingungen sicherzustellen, als der Routinenlieferant dazu, auf jede falsche Eingabe sinnvoll zu reagieren? Um Zuverlässigkeit zu erreichen, ist es notwendig, Verantwortlichkeiten klar zuzuordnen. Mißverständnisse der Art „Ich dachte, Du kümmerst Dich darum!“ – „Nein, wieso, das hast Du doch bisher immer getan!“ sind damit genauso vermeidbar, wie die unnötige doppelte Prüfung von Vorbedingungen.

Wenn man das Ziel verfolgt, die Qualität von Software zu verbessern, dann reicht es nicht aus, Fehler nur zu beheben. Man wird um eine wie auch immer geartete Sanktionierung von Programmierfehlern nicht umhinkommen. Auch dies ist in der Fertigung von Hardware-Produkten eine selbstverständliche Aufgabe der Qualitätssicherung. Voraussetzung dafür ist aber, daß Fehler bestimmten Programmkomponenten (und damit ihren Produzenten) eindeutig zugeordnet werden können. Dies ist nur dann möglich, wenn bereits die Aufgaben klar spezifiziert sind.

Oben wurde bereits erwähnt, daß sich Verträge auch auf Objekte beziehen können. Durch Angabe einer sog. Klasseninvariante ist es in Eiffel möglich, dem Benutzer einer Klasse

Eigenschaften zuzusichern, die die Objekte einer Klasse stets erfüllen werden. Ein Beispiel für die Angabe einer Klasseninvariante zeigt Bild 2.11.

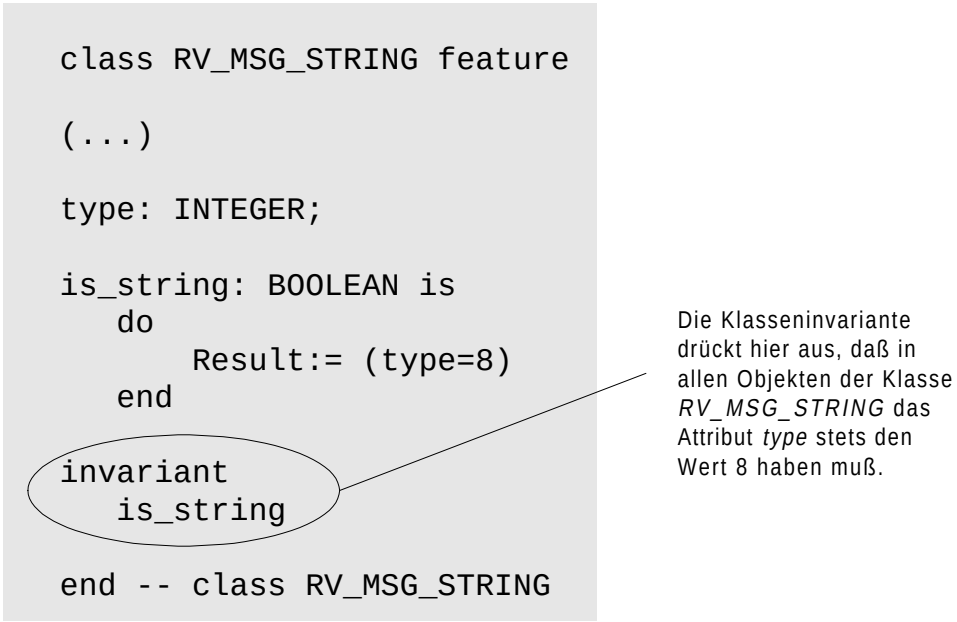


Bild 2.11 Beispiel für die Angabe einer Klasseninvariante

Genaugenommen brauchen diese Eigenschaften jedoch nicht zu jedem Zeitpunkt erfüllt sein. Der Aufruf der Prozedur eines Objekts erfolgt ja in der Regel, um den Zustand des Objekts zu ändern. Während einer solchen Änderung, das heißt innerhalb eines Übergangsintervalls von einem erlaubten Zustand in einen anderen, darf die Klasseninvariante verletzt werden. Dies führt zu einem Problem, das in Abschnitt 2.5[1] über Ausnahmen aufgegriffen wird.

Eiffel beschränkt sich nicht darauf, eine Form für die Formulierung von Verträgen vorzugeben. Es ist möglich, die Einhaltung von Verträgen zur Laufzeit überwachen zu lassen. Dies ist selbstverständlich nur möglich für den Teil der Zusicherungen, der als ein durch den Abwickler auswertbarer boole'scher Ausdruck formuliert ist. Zusicherungen können daneben auch nur durch Menschen interpretierbare Texte[2] enthalten, sie sind also nicht zwangsläufig formal.

Die Überwachung von Zusicherungen hat keinerlei funktionelle Bedeutung für die Abwicklung eines Programms. Sie wird allein durch die Erkenntnis gerechtfertigt, daß es wohl niemals möglich sein wird, fehlerfreie Software herzustellen. Die Überwachung ist ein gutes Werkzeug für das Aufspüren von Programmfehlern in Tests. Als Grundlage des Ausnahmebehandlungsmechanismus, der in Abschnitt 2.5 behandelt wird, kann die

[1] Exkurs „Gleichzeitige Routinenabwicklung auch ohne nebenläufige Programmabwicklung"

[2] Hier wird bewußt das Wort „Kommentar" vermieden, denn eine nichtformale Zusicherung kommentiert nichts. Sie ist ein selbständiges Softwareelement.

Überwachung die Schwere der Auswirkungen von Fehlern vermindern und ein Wiederaufsetzen nach einer Ausnahme ermöglichen.

Die Überwachung von Zusicherungen kann spürbare Geschwindigkeitseinbußen bewirken. Nach der Testphase, sobald man von der Beseitigung der meisten Fehler überzeugt ist, wird man die Überwachung daher in der Regel ganz oder teilweise abschalten. In der Systembeschreibungsdatei läßt sich angeben, ob bzw. an welchen Stellen eine Überprüfung von Zusicherungen erfolgen soll.

Neben den vorgestellten Zusicherungen, die im Rahmen des Vertragsparadigmas eine Rolle spielen, gibt es weitere Zusicherungsarten, die in erster Linie für die Fehlerbeseitigung von Bedeutung sind:

Die Check-Anweisung

Die Check-Anweisung enthält eine Liste boole'scher Ausdrücke, die alle wahr sein müssen, wenn der Abwickler auf die Anweisung trifft.

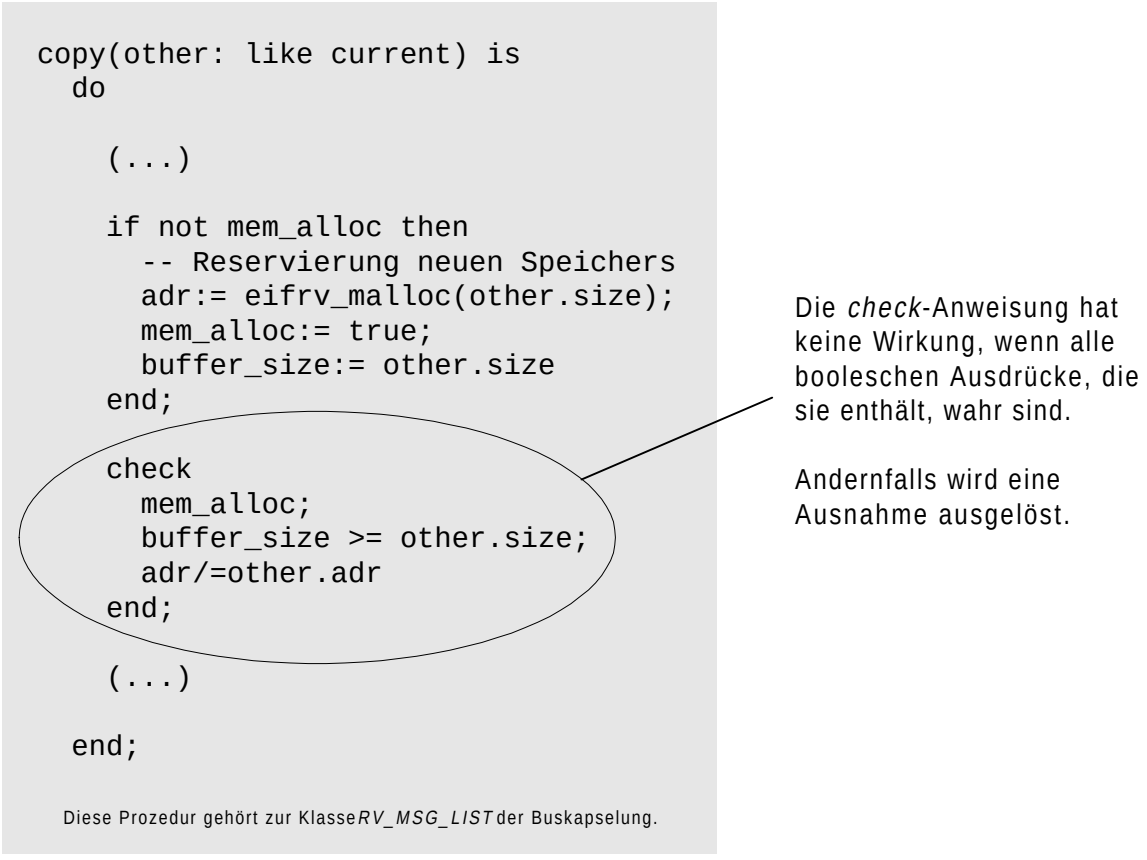


Bild 2.12 Syntaxbeispiel für die Check-Anweisung

Die Check-Anweisung kann an beliebige Stellen in den Rumpf von Routinen eingefügt werden. Mit ihr kann ein Programmierer ausdrücken, von welchen Voraussetzungen er an

einer bestimmten Stelle der Routinenabwicklung ausgeht. Sie ist nicht nur ein Hilfsmittel zur Fehlersuche im Zusammenhang mit dem Ausnahmemechanismus (siehe Abschnitt 2.5), sondern dient auch zur Programmdokumentation. Bild 2.12 zeigt ein Syntaxbeispiel für die Check-Anweisung.

Schleifen-Invarianten und Schleifen-Varianten

Diese Zusicherungen helfen bei der Entdeckung von Fehlern in Schleifen. Die Schleifenvariante verhindert Endlosschleifen, die Schleifeninvariante drückt Eigenschaften aus, die sich durch die Iteration nicht ändern dürfen.

2.5 Der Ausnahmemechanismus

Um den Eiffel-Ausnahmemechanismus vorstellen zu können, möchte ich zunächst einige Begriffe klären:

- Eine ***Operation*** ist die Abwicklung einer Routine oder die Ausführung einer elementaren Anweisung des Abwicklers.
- Eine Operation ***scheitert***, wenn es nicht möglich ist, sie so zu beenden, daß ihr Zweck erfüllt wird.
- Eine ***Ausnahme*** ist der Abbruch einer Operation aufgrund ihres Scheiterns oder aufgrund eines anderen abnormen Ereignisses.
- Ein ***Fehler*** ist die Anwesenheit eines Elements in der Software, das seiner Spezifikation nicht genügt.

Eine elementare Anweisung des Abwicklers – im vorliegenden Fall des virtuellen Eiffel-Abwicklers – ist beispielsweise die Zuweisung (:=). Insbesondere handelt es sich aber auch bei „externen“ C-Routinen um elementare Anweisungen. Diese C-Routinen entsprechen den Mikroprogrammroutinen eines Hardware-Prozessors[1].

Als Beispiel für das Scheitern einer Operation sei der Versuch der Division durch Null genannt. Eine Ausnahme ist dadurch gekennzeichnet, daß der Abwickler nicht an der Stelle fortfährt, die das Programm vorschreibt, sondern zu einer speziellen Ausnahmebehandlungsprozedur springt.

Eine Ausnahme wird ausgelöst, wenn das Scheitern einer Operation festgestellt wird. Das kann sowohl innerhalb einer elementaren Anweisung des Eiffel-Abwicklers als auch in einer in Eiffel geschriebenen Routine[2] geschehen.

Daneben gibt es auch den Fall, daß die Ausnahme durch ein externes Ereignis ausgelöst wird, das auf einem eigenen Ereigniskanal direkt dem Abwickler gemeldet wird. Im Fall eines Hardwareprozessors wäre ein Interrupt-Signal ein solches Ereignis. Im Fall des Eiffel-Prozessor kann dies eine Ereignismeldung vom Betriebssystem sein. Diese Art von Ereignissen ist in obiger Definition mit „anderen abnormen Ereignissen“ gemeint.

So wie hier Fehler definiert ist, ist damit nicht das Fehl*verhalten* eines Systems gemeint, sondern die statische Anwesenheit des Fehlers im Programm, ganz unabhängig von seinem Auftreten. Ein Fehler ist auch dann vorhanden, wenn der Benutzer sich zufällig so verhält, daß er kein Fehlverhalten des Systems bemerkt. Ausschlaggebend ist die Spezifikation, nicht die tatsächliche Benutzung.

---

[1] vgl. [Otto], Abschnitt 3.4.3: „Das Eventloop-Problem und seine Lösung.“

[2] durch die Prozedur „raise“ der Klasse „Exception“, siehe [EiffelLib].

Im Zusammenhang mit dem Vertragsparadigma wurde bereits gesagt, daß es möglich ist, die Einhaltung von Zusicherungen zur Laufzeit zu überprüfen. Dort blieb die Frage offen, was passiert, wenn eine Zusicherungsverletzung entdeckt wird. Die Zusicherungsüberwachung erfolgt durch den Eiffel-Prozessor. Dieser löst eine Ausnahme aus, wenn er eine Zusicherungsverletzung feststellt, d.h. er springt zu einer Ausnahmebehandlungsprozedur. Eine solche Ausnahmebehandlungsprozedur ist vom Programmierer vorgebbar. Sie wird eindeutig einer Routine[1] zugeordnet und ist zuständig für alle Ausnahmen, die während der Abwicklung der Routine auftreten.

Der Ausdruck „während der Abwicklung der Routine“ ist nicht eindeutig. Dies bedarf einer Erläuterung, der der folgende Exkurs dient:

Exkurs: Gleichzeitige Routinenabwicklung auch ohne nebenläufige Programmabwicklung

Ein Eiffel-Programm wird immer nur von einem virtuellen Eiffel-Prozessor abgewickelt. Dennoch kann man auch bei Einprozessorbetrieb von der gleichzeitigen Abwicklung von Routinen sprechen:

Man betrachte eine Routine A, die zur Erfüllung bestimmter Aufträge dient. Dazu wird in A eine andere Routine B aufgerufen, die zur Erledigung eines Unterauftrags in Anspruch genommen wird. Während der Abwicklung von B ist die Abwicklung von A noch nicht beendet. Man kann also mit Recht behaupten, daß zu diesem Zeitpunkt der Abwickler *gleichzeitig* mit der Abwicklung der Routinen A und B beschäftigt ist.

Andererseits kann man behaupten, daß der Abwickler *im Moment* doch nur B abwickelt. Aber auch die Abwicklung von B besteht wieder aus Aufrufen von Unterprogrammen...

Die Mehrdeutigkeit, die hier auftritt, existiert auch im wirklichen Leben. Wenn ich behaupte, daß ich gerade an meiner Diplomarbeit schreibe, könnte ein Beobachter einwenden, daß dies gar nicht stimme, da ich doch gerade Kaffee tränke. Dabei habe ich doch meine Schreibtätigkeit nur für einen Schluck Kaffee vorübergehend unterbrochen.

Die Möglichkeit einer gleichzeitigen Routinenabwicklung impliziert in objektorientierten Programmiersprachen eine besondere Fehlerquelle bei rekursiven Routinenaufrufen. Bei klassischer prozeduraler Programmierung werden die lokalen Variablen einer Routine auf den Stack gerettet, wenn ein Unterprogramm aufgerufen wird. Wird die Routine rekursiv erneut aufgerufen, bevor die erste Abwicklung beendet wurde, wird der Kontext der ersten Abwicklung nicht beschädigt.

[1] genaugenommen der *Implementierung* einer Routine

> Eine Routine eines objektorientierten Programms wird dagegen nicht nur im Kontext ihrer eigenen lokalen Variablen aufgerufen, sondern auch und in erster Linie im Kontext eines Objekts. Eine Routine muß so programmiert sein, daß sie nach ihrer Abwicklung das Objekt in einem konsistenten Zustand hinterläßt, d.h. die Klasseninvariante muß erfüllt sein. Während der Abwicklung braucht der Objektzustand dagegen die Klasseninvariante nicht zu erfüllen. Wird nun das Objekt benutzt, bevor die Routinenabwicklung beendet ist, so muß der Benutzer des Objekts damit rechnen, das Objekt möglicherweise in einem inkonsistenten Zustand vorzufinden. Dieses Problems sollte sich ein Programmierer bewußt sein. Eine Standardlösung dafür bietet Eiffel nicht an.

Zurück zur Ausnahmebehandlung. Eine Ausnahme ist eine Aktion des Abwicklers. Die Mehrdeutigkeit bei der Zuordnung einer Ausnahme zu einer Routine kann nun beseitigt werden: Die Ausnahme wird derjenigen gerade abgewickelten Eiffel-Routine zugeordnet, die als letzte aufgerufen und nicht für die Abwicklung anderer Eiffel-Routinen unterbrochen wurde[1]. „Zuordnung“ einer Ausnahme heißt konkret, daß die Abwicklung der Routine abgebrochen wird.

Existiert für die Routine keine Ausnahmebehandlungsprozedur, dann scheitert sie und es wird eine Ausnahme für die in der Aufruffolge nächst frühere Routine ausgelöst. Dieses Spiel wird rekursiv solange fortgesetzt, bis eine Ausnahmebehandlungsprozedur gefunden wird. Wenn durch den Programmierer gar keine Ausnahmebehandlungsprozedur vorgesehen ist, dann führt eine Ausnahme zum Abbruch der Programmausführung mit einer Fehlermeldung des Eiffel-Laufzeitsystems.

Mögliche Reaktionen auf eine Ausnahme

Für die Reaktion auf eine Ausnahme gibt es nur zwei zulässige Möglichkeiten: Entweder man wiederholt nach „Reperaturarbeiten“ die verunglückte Operation, oder aber man führt Aufräumarbeiten durch und meldet sein Scheitern an die aufrufende Routine.

So zu tun, als ob man erfolgreich gewesen sei und kommentarlos die Kontrolle an den Aufrufer zurückzugeben, wäre keine zulässige Reaktion. An dieser Stelle sei an das Vertragsparadigma erinnert. Wenn nicht die Vorbedingungen verletzt waren, dann handelt es sich beim Scheitern der Operation um eine Vertragsverletzung durch den Programmierer der Routine. Es ist klar, daß die Vorspiegelung der erfolgreichen Abarbeitung der Routine nicht richtig wäre.

Mit „Aufräumen“ ist gemeint, daß eine Ausnahmebehandlungsprozedur das Objekt in einen konsistenten Zustand bringt. Das sollte insbesondere die Erfüllung der Klasseninvariante beinhalten und die Ausbreitung des Schadens begrenzen.

---

[1] engl.: current routine, vgl. [ETL]

Die Schlüsselwörter „rescue“ und „retry“

Ausnahmebehandlungsprozeduren werden durch das Schlüsselwort „rescue“ eingeleitet und stehen im Programm unterhalb des Rumpfes einer Routine. Wird eine Ausnahmebehandlungsprozedur bis zu ihrem Ende abgewickelt, dann wird automatisch eine Ausnahme für die aufrufende Routine ausgelöst. Auf diese Weise erfährt der Aufrufer auch vom Scheitern der Operation.

Will man die Abwicklung der Routine erneut versuchen, so benutzt man die „retry“-Anweisung innerhalb der Ausnahmebehandlungsprozedur. Dies bewirkt die erneute Abwicklung der abgebrochenen Routine von Anfang an. Auf die lokalen Variablen der Routine und auf die Attribute des Objekts kann aus der Ausnahmebehandlungsprozedur heraus schreibend zugegriffen werden. Die lokalen Variablen werden bei einem Wiederversuch nicht erneut initialisiert. Durch die Ausnahmebehandlungsprozedur können möglicherweise Veränderungen vorgenommen werden, die einen erfolgreichen Wiederversuch erwarten lassen.

In Bild 2.13 ist ein Syntaxbeispiel für die Benutzung der rescue/retry-Klauseln abgebildet.

Der Zweck des Ausnahmebehandlungsmechanismus

Bei der Behandlung des Vertragsparadigmas wurde gesagt, daß Zusicherungen nicht verletzt werden dürfen und der Ausnahmemechanismus der Erkenntnis Rechnung trägt, daß es wohl nie möglich sein wird, fehlerfreie Programme zu schreiben.

Die Zusicherungsüberwachung kontrolliert bildlich gesprochen die Vertrauenswürdigkeit der Vertragspartner. Außerdem kann insbesondere die Überprüfung der Vorbedingungen benutzt werden, um die schlimmsten Auswirkungen von Fehlern zu vermeiden. Es sei daran erinnert, daß eine Routine „machen darf, was sie will“, wenn die Vorbedingungen verletzt sind.

Ausnahmebehandlungsprozeduren haben insbesondere den Sinn, die Abwicklung eines Programms nicht gleich bei jeder Ausnahme zu beenden, sondern dem Benutzer eines Programms noch die Chance zu geben, sein Ziel anders zu erreichen.

Bei einem typischen von Benutzereingaben getriebenen Programm ist der Benutzer nach entsprechender Erfahrung häufig in der Lage, Situationen zu vermeiden, in denen Fehler des Programms spürbar werden. Ein Programm bleibt so trotz seiner Fehler benutzbar. Der erneute Versuch einer Operation mit retry ist besonders dann sinnvoll, wenn innerhalb der Operation Benutzereingaben vorkommen, die einen Einfluß auf das Wirksamwerden des Fehlers haben können. Der Benutzer hat dann die Möglichkeit, durch Variation der Eingaben das Auftreten des Fehlers zu vermeiden oder vor einem Absturz noch Daten zu retten.

Ein Ausnahmebehandlungsmechanismus sollte aber nie Bestandteil der spezifizierten Funktionalität des Programms sein. Der Idealzustand eines Programms in Bezug auf Fehler ist der, in dem die Ausnahmebehandlung weggelassen werden kann.

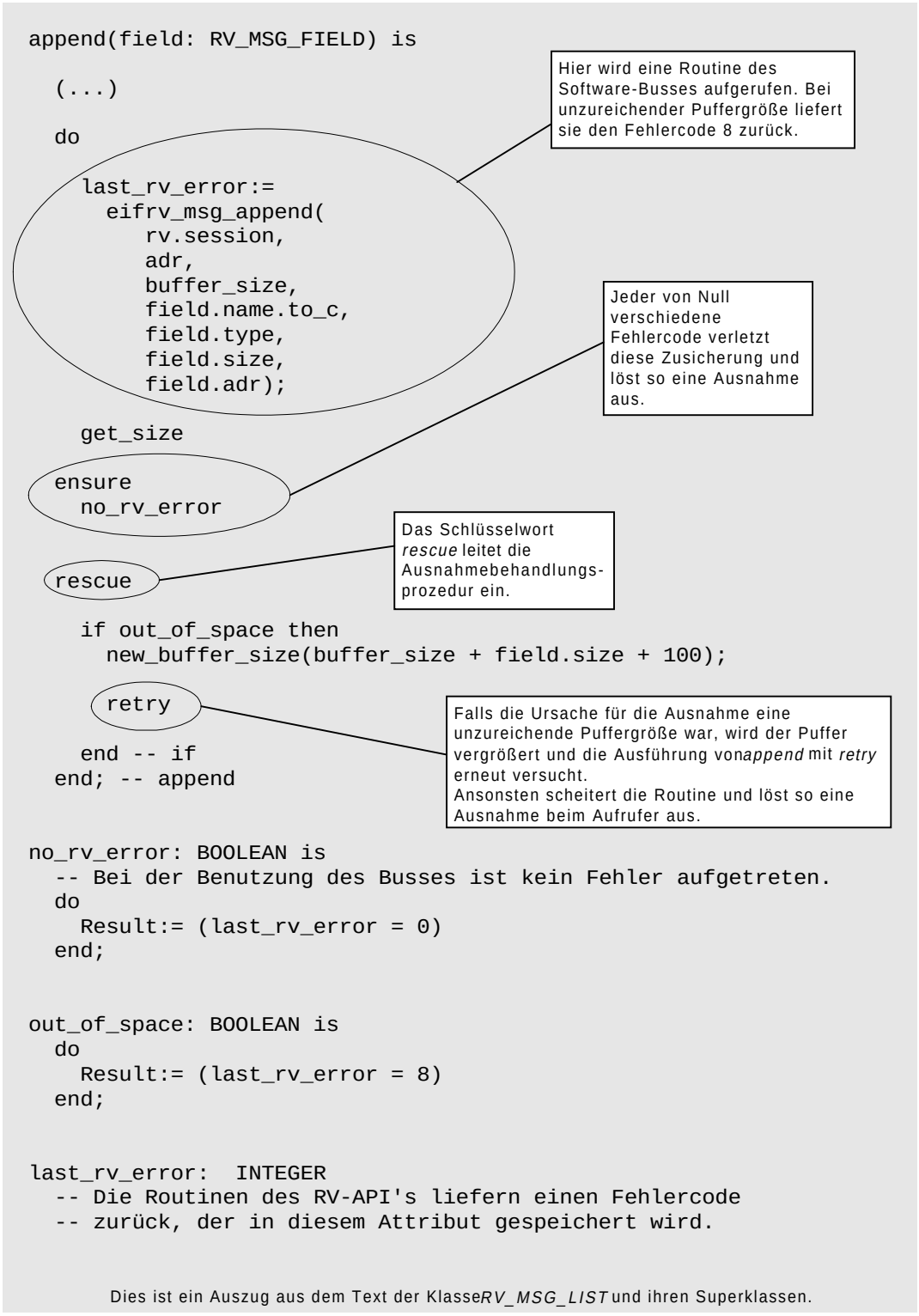


Bild 2.13 Beispiel zum Ausnahmebehandlungsmechanismus

Fehleingaben *des Benutzers* sollten keine Ausnahmen bewirken und nicht in der Zuständigkeit der Ausnahmebehandlung liegen. Dies folgt aus dem Vertragsparadigma. Wenn ein Benutzer durch eine Fehleingabe die Vorbedingungen einer Routine verletzt, dann

gibt es keinerlei Sicherheit, was bei der weiteren Abwicklung der Routine passiert, katastrophale Folgen wären nicht ausgeschlossen. Wie jeder andere Kunde einer Routine auch muß der menschliche Benutzer die Erfüllung der Vorbedingung mit Sicherheit gewährleisten. Dies ist wohl in den meisten Fällen eine zu strenge Anforderung. Die Eingabeüberprüfung sollte dementsprechend durch reguläre Programmkonstruktionen erfolgen.

Es gibt aber noch eine andere Rechtfertigung des Ausnahmebehandlungsmechanismus. In manchen Situationen steht die Überprüfung der Erfüllung aller Vorbedingungen in einem auffälligen Mißverhältnis zu der Wahrscheinlichkeit der Nichterfüllung der Bedingungen. Es liegt dann nahe, die Operation „zu versuchen“ und anschließend den Auftraggeber über Erfolg oder Mißerfolg zu informieren. Ein Beispiel hierfür sind numerische Operationen, die zu Überläufen führen. Ein anderes Beispiel ist das Fehlen von Speicherplatz bei der Erzeugung von Objekten. Es wäre vollkommen unpraktibabel, *vor* jeder Objekterzeugung herauszufinden, ob noch genügend freier Speicherplatz vorhanden ist.

3 Der Anschluß des Rendezvous-Busses an Eiffel

Das Beispielsystem des Software-Technologie-Labors soll auf mehrere Rechner verteilt sein. Es kommen mehrere Sprachen zum Einsatz. Für die Kommunikation zwischen den Komponenten wird der Rendezvous-Software-Bus der Firma TIBCO verwandt. Eine knappe Beschreibung des Busses findet sich in Abschnitt 3.2, dort wird auch auf ausführlichere Dokumentation zum Bus verwiesen.

Um den Bus von der Sprache Eiffel aus benutzbar zu machen, mußte die C-Routinenbibliothek des Busses in Eiffel gekapselt werden. Daher soll im folgenden Abschnitt 3.1 zunächst die C-Sprachschnittstelle von Eiffel beschrieben werden.

Abschnitt 3.3 dient der Darstellung der Kapselung des Busses in Eiffel.

Ich setze für dieses Kapitel eine Grundkenntnis der Programmiersprachen Eiffel und C voraus und möchte daher an dieser Stelle auf [ETL] und [Darnell] verweisen.

3.1 Die C-Schnittstelle von Eiffel

Für die Anbindung des Rendezvous Busses an Eiffel[1] war zunächst das grundsätzliche Problem der Einbindung von C-Routinen in Eiffel-Programme zu lösen, denn der Rendezvous Bus ist benutzbar über eine C-Routinenbibliothek[2]. Ich hoffe, daß der vorliegende Abschnitt nützlich ist für jemanden, der vor dem gleichen Problem der Anbindung einer C-Bibliothek an Eiffel steht. Es gibt eine ganze Reihe von Hindernissen,

[1] Wenn hier von Eiffel die Rede ist, so ist damit die Sprache in der von ISE gelieferten Implementierung gemeint.

[2] In C ist es üblich, alle Routinen Funktionen zu nennen, da C formal keinen Unterschied zwischen Funktionen und Prozeduren macht. Ich behalte hier die in Abschnitt 2.3.4.1 eingeführte Klassifikation bei und verwende weiterhin „Routine“ als Sammelbegriff für Funktionen und Prozeduren.

für deren Überwindung es schwierig ist, hilfreiche Informationen zu finden. In der Hoffnung, anderen den Weg ebnen zu können, möchte ich diese Probleme und die gefundenen Lösungen im folgenden dokumentieren.

3.1.1 C-Routinen-Bibliotheken

Eine C-Bibliothek besteht aus zwei Teilen. Der erste Teil ist eine Sammlung bereits in die Maschinensprache des jeweiligen Rechners übersetzter Funktionen und Prozeduren. Diese Sammlung liegt jedoch nicht als ausführbares Maschinenprogramm vor, sondern als Datei in einem Zwischenformat, das es erlaubt, das Maschinenprogramm an beliebige Stellen des Rechners verschieben zu können. Außerdem sind noch nicht bekannte Adressen durch vorläufige Platzhalter ersetzt. Dieses Zwischenformat wird Objektformat[1] genannt.

Mehrere Dateien im Objektformat können zu einer Bibliotheksdatei verbunden werden. Bibliotheksdateien haben zwar ein eigenes Dateiformat, für die Verwendung spielt dieser Unterschied jedoch keine Rolle.

Eine Objekt- oder Bibliotheksdatei enthält neben dem Programmcode symbolische Namen und diesen zugeordnete Stellen im Programmcode. Diese Information wird gebraucht, um Routinen und Variablen durch einen Namen statt durch eine Adresse zu identifizieren. Durch einen Binder ist es möglich, die Bibliothek mit anderen im Objektformat vorliegenden Programmbestandteilen zu einem ablauffähigen Maschinenprogramm zusammenzufügen.

Mit einer Bibliotheksdatei allein läßt sich wenig anfangen. Beim Aufruf einer Routine können Parameter übergeben werden. Um ein Programm schreiben zu können, das Routinen aus einer Bibliothek benutzt, braucht man Informationen über die Parameter und den Rückgabewert von Routinen sowie über die dort verwendeten Datentypen. Diese und weitere Informationen sind Bestandteil des zweiten Teils einer C-Bibliothek, der sog. Header-Datei. Dort finden sich die Deklarationen von Routinenköpfen, Datentypen und Konstantenvereinbarungen sowie Vereinbarungen von gemeinsam genutzten Variablen.

Zu einer C-Bibliothek gehören also immer zwei Dateien: eine im Bibliotheksformat und die Header-Datei im Textformat.

Die Bibliotheksdatei wird vom Binder benötigt, die Header-Datei wird wie jede andere Header-Datei durch eine *Include*-Anweisung an den C-Präprozessor direkt in den Quelltext des die Bibliothek benutzenden Programms eingebunden.

---

[1] Diese Bezeichnung ist nicht besonders glücklich. Sie hat nichts mit Objekten zu tun, sondern rührt von der Bezeichnung für die Datei her, in die der Übersetzer sein Ergebnis schreibt. Diese Datei wird englisch „object file“ genannt.

Eine derartige Schnittstelle einer Bibliothek ist an die verwendete Programmiersprache, an die Einhaltung bestimmter Konventionen bei der Übersetzung dieser Sprache, sowie an eine bestimmtes Format für die Objekt-Files gebunden. Sie wird Sprachschnittstelle genannt, impliziert aber mehr als die reine Sprache.

3.1.2 Dynamisch gebundene Bibliotheken

Bei der oben beschriebenen Technik der Benutzung von Routinenbibliotheken werden die verwendeten Routinen der Bibliothek zu dem sie benutzenden Hauptprogramm hinzugefügt.

Man betrachte den Fall, daß auf einem Rechner eine Standardbibliothek von einer ganzen Anzahl von nebenläufig abgewickelten Programmen[1] benutzt wird. Dann befindet sich der Code dieser Bibliothek für jedes Programm einmal im Speicher. Das ist eine Verschwendung von Speicherplatz.

Es wäre besser, wenn der Code nur an einer Stelle stünde und von allen Programmen gemeinsam benutzt würde. Genau dies geschieht bei der Verwendung von dynamisch gebundenen Bibliotheken[2]. Darüber hinaus werden die Routinen der Bibliothek erst in den Speicher geladen, wenn sie gebraucht werden.

Die Benutzung einer solchen Bibliothek ist denkbar einfach, wenn eine normale, statische Bibliothek vorliegt, die die DLL kapselt. Die statische Bibliothek wird ganz normal mit dem Hauptprogramm zusammengebunden und benutzt ihrerseits die DLL. Die Rendezvous-Bibliothek liegt auf diese Art vor.

3.1.3 Benutzung einer C-Bibliothek durch ein Eiffel-Programm

Die Schnittstelle des Rendezvous-Busses[3] ist eine C-Sprachschnittstelle. Soll der Bus von Eiffel-Programmen benutzt werden, so braucht auch Eiffel eine Sprachschnittstelle für C. Diese Sprachschnittstelle gibt es. Ich habe bereits erwähnt, daß Eiffel zunächst in C übersetzt wird. Die Implementierung und die Benutzung der Schnittstelle sollte daher eigentlich einfach sein. Es gibt aber eine Anzahl Haken und Ösen.

So gut die Dokumentation von Eiffel ansonsten ist, im Bereich der C-Schnittstelle ist sie widersprüchlich, unvollständig und stimmt nicht mit dem ausgelieferten Produkt überein.

---

[1] Hiermit ist ein echter oder virtueller Mehrprozessorbetrieb auf Hardwareebene gemeint.

[2] engl.: Dynamic-Link Library, Abk.: DLL

[3] im folgenden abgekürzt mit „RV-API“ für „Rendezvous Application Programming Interface“

Das kann damit zusammenhängen, daß eine Sprachschnittstelle mehr umfaßt als die reine Sprache. Sie ist übersetzerabhängig und damit plattformabhängig. Zusätzliche Schwierigkeiten entstehen dadurch, daß sich ISE nicht an bestimmte Vorgaben der ANSI-C Norm hält, um nicht auf ANSI-C kompatible Übersetzer beschränkt zu sein[1].

Was braucht man für eine Schnittstelle zwischen zwei Sprachen?

- Routinen müssen gegenseitig aufrufbar sein.
- Die Übergabe von Daten muß möglich sein. Dafür muß es gemeinsame Datentypen geben.
- Der Zugriff auf gemeinsam genutzte Variablen sollte möglich sein.

Die dritte Forderung muß nicht unbedingt erfüllt werden, da es prinzipiell immer möglich ist, Daten in Argumenten oder als Ergebnis von Routinen zu übergeben.

Wenn ich im folgenden von „C“ und „Eiffel“ rede, so ist das eine bequeme, aber erklärungsbedürftige Redeweise. Man stelle sich dazu ein in Eiffel und C programmiertes System aufgeteilt in zwei Akteure vor, wobei sämtliche in Eiffel programmierten Elemente dem Eiffel-Akteur und alle in C vorliegenden Elemente dem C-Akteur zugerechnet werden.

3.1.3.1 Der Aufruf von argumentlosen C-Prozeduren von Eiffel aus

Eine C-Prozedur wird in ein Eiffel-Programm eingebunden, indem sie an die Stelle des Rumpfes[2] einer Eiffel-Prozedur tritt. Der Rumpf wird dabei durch einen Verweis auf die C-Routine, die durch ihren Namen identifiziert wird, ersetzt. Ein Beispiel zeigt Bild 3.1. Wenn man die „alias“-Klausel wegläßt, wird nach einer C-Routine mit dem Namen der Eiffel-Routine gesucht. Bei der Namensangabe ist darauf zu achten, daß in C der Unterschied zwischen Klein- und Großbuchstaben im Gegensatz zu Eiffel signifikant ist.

3.1.3.2 Der Aufruf von argumentlosen Eiffel-Prozeduren von C aus

Beim Aufruf von argumentlosen Eiffel-Prozeduren von C aus ist eine Besonderheit zu beachten, die darauf beruht, daß Eiffel eine objektorientierte Sprache ist:

Eine Routine ist immer einer Klasse zugeordnet. Die Abwicklung einer Routine erfolgt ausnahmslos im Kontext eines Objekts. Dahinter steht die Vorstellung, daß ein Objekt ein Akteur ist, der „seine“ Routine abwickelt, wenn er dazu beauftragt wird. Auch wenn eine Routine von C aus aufgerufen wird, ist daher immer ein Objekt anzugeben, in dessen

---

[1] ISE benutzt nicht das „ANSI-Prototyping“ für die Parameterleisten von Routinen. Es findet daher eine automatische Typumwandlung nach `int` bzw. `double` statt. Siehe hierzu [DARNELL], Kapitel 9.

[2] Der Rumpf einer Routine, engl. routine body, ist der eigentlich abwickelbare Teil der Routine, ohne Parameterdeklarationen etc. In Eiffel wird der Rumpf einer Routine durch das Schlüsselwort „do“ oder „once“ eingeleitet.

Kontext die Routine abzuwickeln ist. Jeder Eiffel-Routine ist daher beim Aufruf von C aus als erstes Argument ein Verweis auf ein Eiffel-Objekt zu übergeben. Aus C-Sicht gibt es keine argumentlosen Eiffel-Prozeduren. Wie ein Objektverweis übergeben wird, und wie C überhaupt zu Verweisen auf Eiffel-Objekte kommt, darauf wird im folgenden eingegangen.

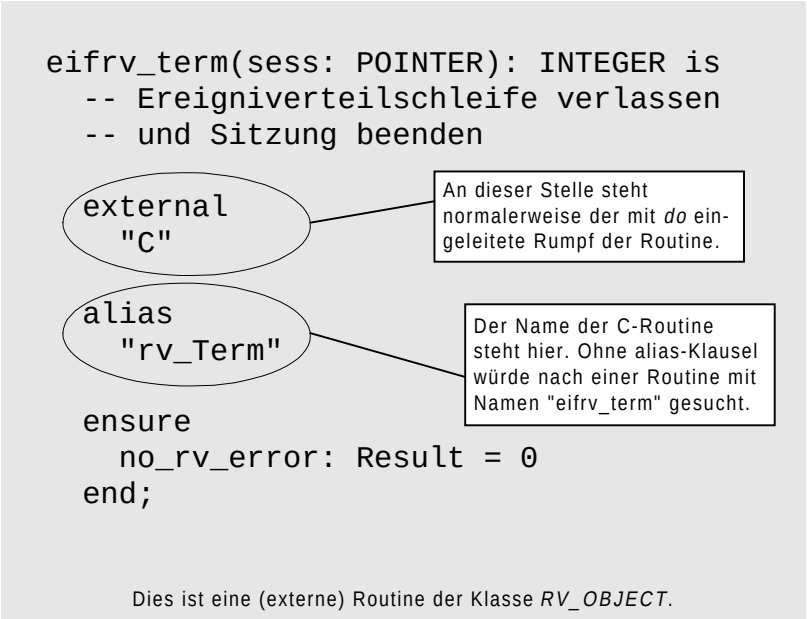


Bild 3.1 Einbindung von C-Routinen

In C gibt es zwei Möglichkeiten, eine aufzurufende Routine zu identifizieren. Der Normalfall ist die Angabe des Namens der Routine. Der C-Übersetzer bzw. der Binder ersetzt den Namen später durch die Adresse der Routine. Es besteht aber auch die Möglichkeit, Variablen einzuführen, die die Adresse einer Routine enthalten. Der in Klammern gesetzte Name der Variablen kann dann an der Stelle des Namens der Routine stehen und bewirkt die Ausführung der durch ihre Adresse identifizierten Routine.

Im Sprachumfang von Eiffel existiert der Adress-Operator und der Basis-Datentyp `Pointer`. Werte vom Typ `Pointer` sind in Eiffel nicht bearbeitbar. Ihr einziger Zweck ist die Bereitstellung von Adressen, die ein C-Programm verwenden kann, um auf Eiffel-Komponenten zuzugreifen. Der Adressoperator ist anwendbar auf lokale Variablen, Objektattribute und Routinen. Als Ergebnis liefert er eine Speicheradresse, die dann an C übermittelt werden kann.

Auf C-Seite kann die so übermittelte Adresse in einer Variablen gespeichert werden, welche wie oben beschrieben zum Aufruf der Routine benutzt werden kann. Beispiele hierfür finden sich in [E&C] sowie im C-Quelltext im Anhang.[1]

[1] Die Adresse der Eiffel-Prozedur „callback_general“ wird nach C übermittelt, dort in einer Variable gespeichert und vom Callback-Verteiler zum Aufruf der Prozedur verwendet, siehe unten.

Eine zweite Möglichkeit, die Adresse einer Eiffel-Routine auf C-Seite bekannt zu machen, ist die Verwendung der C-Bibliothek „CECIL“[1]. Diese Bibliothek stellt Funktionen zur Verfügung, die als Argument den Namen einer Eiffel-Routine akzeptieren und die Adresse der Routine als Ergebnis liefern.

3.1.3.3 Gemeinsame Datentypen

3.1.3.3.1 Die Übergabe von Eiffel-Basistypen

Eine Voraussetzung für den gegenseitigen Aufruf von Routinen mit Argumenten und/oder Ergebnis ist die Kompatibilität der Typen der übergebenen Daten. Die Menge der Eiffel-Basistypen ist abbildbar auf eine echte Teilmenge der Menge der C-Typen. Bei der Übersetzung in C wird jeder Eiffel-Basistyp durch einen geeigneten C-Typ realisiert. Auf welche C-Typen die Eiffel-Basistypen[2] abgebildet werden, kann einer bestimmten Header-Datei[3] entnommen werden. Die Zuordnung ist plattformabhängig. Nachfolgend gemachte Aussagen bezüglich dieser Abbildung gelten für die Windows-NT-Plattform (Version 3.51), den C-Übersetzer Microsoft Visual C++ 2.0 und ISE Eiffel 3.3.

C stellt eine größere Auswahl an Ganzzahl-Datenformaten zur Verfügung als Eiffel. Beispielsweise gibt es den Typ `SHORT INTEGER`. Ein entsprechender Eiffel-Typ existiert nicht. Die Menge der als `SHORT INTEGER` darstellbaren Ganzzahlen ist aber eine Teilmenge der durch den Eiffel-Typ `INTEGER` darstellbaren Zahlen. Man kann daher zunächst eine Typumwandlung (cast) von `SHORT` nach `LONG` vornehmen und dann die Zahl in `LONG`-Darstellung nach Eiffel übergeben, bzw. umgekehrt. `LONG` ist der C-Typ, mit dem der Eiffel-Typ `INTEGER` realisiert wird. Probleme gibt es lediglich bei Ganzzahlen vom C-Typ `UNSIGNED LONG`, bei denen das höchstwertigste Bit gesetzt ist. Sie sind in Eiffel nicht darstellbar.

3.1.3.3.2 Die Übergabe komplexer Datentypen

Es ist wünschenswert, wenn auch nicht notwendig, neben den genannten Basistypen auch komplexere Datentypen übergeben zu können. Auf Eiffel-Seite wäre das der Datenteil eines Objekts. Soweit es sich nicht um „expandierte“ Typen handelt, enthalten Größen[4] in Eiffel immer einen Verweis auf ein Objekt und nicht den Datenteil des Objekts selbst.

---

[1] Eine Beschreibung von CECIL findet sich in [ETL, Kapitel 24.7]. Die zugehörige Header-Datei CECIL.H befindet sich im Unterverzeichnis .\bench\spec\w32msc\include des Eiffel-Installationsverzeichnisses.

[2] Dies sind die Typen `INTEGER`, `CHARACTER`, `REAL`, `DOUBLE`, `POINTER` und `BOOLEAN`.

[3] Es handelt sich um die Datei „`PORTABLE.H`“. Diese Datei ist im Unterverzeichnis .\bench\spec\w32msc\include des Eiffel-Installationsverzeichnisses zu finden.

[4] vgl. Abschnitt 2.3.6, Seite 24.

Entsprechend werden beim Aufruf von C-Routinen von Eiffel aus als Parameter Objektverweise übergeben.

Grundsätzlich ist es auch möglich, den Datenteil eines Objekts an eine C-Routine weiterzugeben, indem das aktuelle Argument der Routine von einem expandierten Typ ist. Wie dies geschehen könnte, ist jedoch nicht dokumentiert. Der Datenteil eines Objekts wird vom Eiffel-Übersetzer vermutlich als C-Structure realisiert. Wenn dies so ist, dann müßte auf C-Seite eine Deklaration dieser C-Structure zur Verfügung gestellt werden, beispielsweise in Form einer durch den Eiffel-Übersetzer generierten Header-Datei.

Im umgekehrten Fall müßte aus einer nach Eiffel übergebenen C-Structure ein Objekt erzeugt werden können. Das heißt, man müßte eine zu einer Structure-Deklaration passende Klasse schreiben können. Für beide Fälle bietet die Eiffel-Entwicklungsumgebung jedoch keinerlei Unterstützung.

In diesem Zusammenhang zu erwähnen ist die Klasse `INTERNAL`, die es ermöglicht, zur Laufzeit einige Informationen über die interne Repräsentation von Eiffel-Objekten zu erhalten. Eine (knappe) Beschreibung dieser Klasse findet man in [EiffelLib].

Daneben gibt es die schon genannte C-Bibliothek „CECIL“, die einen komfortablen Zugriff auf einzelne Attribute von Objekten ermöglicht. Die Übergabe des kompletten Datenteils von Objekten wird durch CECIL jedoch nicht unterstützt.

Die Möglichkeit der Übergabe des Datenteils von Objekten wurde in der vorliegenden Arbeit nur für den Spezialfall der Zeichenketten weitergehend untersucht, da dies für die Kapselung des Rendezvous-Busses ausreichend war.

3.1.3.3.3 Die Übergabe von Objektverweisen

Ein Objektverweis wird bei der Übersetzung des Eiffel-Programms in C realisiert durch einen Zeiger auf den Beginn des Datenteils eines Objekts. Würde ein solcher Zeiger unmittelbar an eine C-Routine weitergegeben, käme es zu Problemen.

Durch die automatische Speicherbereinigung wird der von unbenutzten Objekten beanspruchte Speicher wieder freigegeben. Im Laufe der Zeit kann es dies dazu führen, daß es keine großen zusammenhängenden freien Bereiche im Speicher mehr gibt. Es ist dann für die Speicherbereinigung unumgänglich, eine „Flurbereinigung“ vorzunehmen, und Objekte[1] im Speicher zu verschieben. Man darf sich daher nicht darauf verlassen, daß ein Objekt seinen Platz behält. Die Weitergabe der Adresse eines Objekts ist daher in den meisten Fällen nicht sinnvoll. Zur Lösung dieses Problems erhält die Speicherbereinigung eine Tabelle aufrecht, in der die momentane Adresse von Objekten eingetragen ist. Statt eines direkten Zeigers auf das Objekt wird ein Zeiger auf den Tabelleneintrag mit der aktuellen Adresse des Objekts übergeben.

---

[1] Hier und im folgenden ist genaugenommen der Datenteil, also die Implementierung der Attribute eines Objekts gemeint. Die Routinen eines Objekts werden bekanntlich im Kontext der Klasse gespeichert.

Es gibt eine Asymmetrie bei der Übergabe von Objektreferenzen. Übergibt Eiffel beim Aufruf einer C-Routine Objektverweise, so kommt bei C ein Verweis auf einen Tabelleneintrag an. Erwartet dagegen eine Eiffel-Routine den Verweis auf ein Objekt, so muß C die tatsächliche Objektadresse angeben. Es gibt ein C-Makro mit der Bezeichnung `eif_access`, das den Verweis auf den Tabelleneintrag in die Objektadresse umwandelt. Die Umwandlung sollte nur unmittelbar vor der Übergabe der Objektadresse an Eiffel erfolgen, so daß sichergestellt ist, daß zwischenzeitlich keine Objektverschiebung durch den Speicherbereiniger vorgenommen wird.

Es gibt eine weitere Schwierigkeit mit der Übergabe von Objektverweisen an Eiffel. Wenn Objekte nicht mehr benötigt werden, kann ihr Speicherbereich wieder freigegeben werden. Wie aber soll Eiffel feststellen, ob es auf C-Seite noch einen Verweis auf ein Objekt gibt? Zu diesem Zweck gibt es als Bestandteil der CECIL-Bibliothek das Prozedurenpaar `eif_adopt` und `eif_wean`. Die Prozedur `eif_adopt` schützt ein Objekt vor der Speicherbereinigung, die Prozedur `eif_wean` gibt es wieder frei.

3.1.3.3.4 Die Übergabe von Attribut- und Variablenadressen

Der bereits genannte Adressoperator kann zur Ermittlung der Adresse einer Eiffel-Routine verwendet werden. Die Weitergabe der Adresse einer Routine ist unproblematisch, da Routinen zur Laufzeit nicht im Speicher verschoben werden.

Der Adressoperator kann nicht nur auf Routinen, sondern auch auf die Attribute eines Objekts und auf lokale Variablen angewandt werden. Er liefert dann die Adresse des Speicherplatzes des Attributs oder der Variablen. Auch hier tritt das Problem der Verschiebbarkeit von Objekten auf. Wenn die Aktivierung der Speicherbereinigung ausgeschlossen werden kann, dann kann es allerdings sinnvoll sein, die Adresse an die C-Seite zu übergeben, um ihr so die Möglichkeit zu geben, direkt den Wert des Attributs oder der Variablen zu verändern.

Es ist gängiger C-Stil, das Ergebnis von C-Funktionen nicht als Ergebnis weiterzugeben, sondern direkt an durch den Aufrufer in Form von Adressen angegebene Speicherstellen zu schreiben. Der Adressoperator erlaubt eine Anpassung der Eiffel-Seite an eine derartig vorgegebene Parameterleiste einer C-Routine.

Der Adressoperator ist nur verwendbar als aktueller Parameter eines Routinenaufrufs. Wenn es sich bei der aufgerufenen Routine um eine externe C-Routine handelt, dann darf man davon ausgehen, daß zwischen dem Aufruf und der Abgabe des Abwicklers an die C-

Routine der Speicherbereiniger nicht mehr aktiv wird.[1] Bis zur Rückgabe des Abwicklers an Eiffel[2] behält die übergebene Adresse auf jeden Fall ihre Gültigkeit.

3.1.3.3.5 Die Übergabe von Zeichenketten

Zeichenketten gehören nicht zu den Eiffel-Basistypen. In C werden Zeichenketten realisiert durch Zeiger auf Bytefelder, deren Ende durch ein Nullzeichen markiert wird.

In Eiffel sind Zeichenketten Exemplare der Klasse `STRING`. Intern wird zur Darstellung einer Zeichenkette in Eiffel ein Objekt der Klasse `SPECIAL[CHAR]` verwendet. Dieses wiederum ist nichts anderes als ein Bytefeld, wobei ein Buchstabe wie in C durch ein Byte dargestellt wird. Allerdings verwendet Eiffel kein Nullzeichen zur Markierung des Feldendes. Statt dessen ist die Länge des Felds in einem Attribut der Klasse gespeichert.

Die Klasse `STRING` stellt die Funktion „to_c“ zur Verfügung. Diese Funktion liefert einen Verweis auf ein Objekt vom Typ `SPECIAL[CHAR]`, dem ein Nullzeichen angehängt wurde. Der Datenteil dieses Objekts kann unmittelbar als C-String verwendet werden.

Sollte ein Eiffel-String allerdings ein Nullzeichen enthalten, was erlaubt ist, so bleiben alle Zeichen hinter dem Nullzeichen für C unsichtbar.

Umgekehrt kann man die Prozedur „from_c“ der Klasse `STRING` verwenden, um den Inhalt eines C-Strings in einen Eiffel-String zu kopieren. Die Prozedur erwartet als Argument einen Zeiger (Eiffel-Basistyp `POINTER`) auf das erste Zeichen der Zeichenkette.

Bei der Übergabe einer Zeichenkette in Form eines `SPECIAL`-Objekts an C besteht das bekannte Problem der Verschiebbarkeit dieses Objekts im Speicher. Das ist dann unproblematisch, wenn die empfangende C-Routine die Zeichenkette kopiert und die Kopie weiterverwendet. Bei der Übergabe von Zeichenketten an den Rendezvous-Bus ist dies vermutlich der Fall. Bei Tests ergaben sich keine Probleme, wenn eine Zeichenkette nach der Übergabe verändert wurde. Es kann also davon ausgegangen werden, daß der Rendezvous-Bus mit einer Kopie arbeitet.

Die Problematik der Speicherverwaltung in C

Hier wird ein grundsätzliches Problem der Sprache C deutlich. Der Programmierer ist voll für die Speicherverwaltung verantworlich. Werden Speicherbereiche von mehreren Parteien benutzt, dann ist eine Absprache notwendig darüber, wer für die Speicherfreigabe zuständig ist, und ab wann eine Freigabe erfolgen darf. Die Sprache C stellt zur Lösung dieses

---

[1] Dies wird zwar nirgendwo explizit gesagt, die Verwendung des Adressoperators in der Eiffel-Basisbibliothek läßt aber darauf schließen. Es würde auch keinen Sinn machen, an dieser Stelle den Speicherbereiniger zu aktivieren.

[2] Dies braucht nicht die Rückkehr aus der gerufenen C-Routine sein, die Rückgabe des Abwicklers kann auch ein nachfolgender Aufruf einer Eiffel-Routine von C aus sein.

Problems weder Sprachmittel zur Verfügung, noch gibt sie Konventionen für die notwendige Dokumentation der diesbezüglich getroffenen Entscheidungen.

Ich habe den Eindruck, daß das RV-API sehr sorgfältig programmiert und dokumentiert wurde. Dennoch fehlen die für das Problem der Speicherfreigabe notwendigen Informationen. Dies zeigt in meinen Augen, wie wichtig es ist, daß eine Sprache die Dokumentation derartiger Entscheidungen unterstützt, selbst wenn solche Informationen nicht für den Übersetzer, sondern für den menschlichen Leser von Interesse sind.

Der Nutzen einer automatischen Speicherbereinigung tritt hier deutlich vor Augen. Wenn zwei Seiten auf gemeinsam genutzte Variablen zugreifen, dann weiß in der Regel keine Seite, ob die Variable von der anderen Seite noch benötigt wird. Ohne zusätzliche Absprache ist es für keine der beiden Seiten möglich, den Speicherbereich der Variablen freizugeben. Es ist sinnvoll, diese Aufgabe dem unterliegenden Sprachsystem zu übertragen.

3.2 Der Rendezvous-Software-Bus

Eine kurze Beschreibung des Rendezvous-Software-Busses findet sich in [Otto, Kapitel 3.3]. Ausdrücklich hingewiesen sei hier auf [Auer]. Details und die Beschreibung der C-Routinen findet man in [RVProg] sowie in der Header-Datei `rv.h`[1]. Die Beschreibung des Busses, seiner Funktionsweise und seiner Rolle im Software-Technologie-Labor soll hier nur in aller Kürze erfolgen.

Der Bus stellt zwei Verfahren zur Kommunikation zwischen Akteuren in einem Netzwerk zur Verfügung:

Das Publish/Subscribe-Verfahren ist vergleichbar mit der Veröffentlichung von Nachrichten in Kleinanzeigen einer Zeitschrift. Der Inserent kennt weder die Empfänger seiner Nachricht, noch deren aktuelle Zahl. Ein potentieller Leser kann bestimmte Zeitschriften bei einem Verlag abonnieren und diese Abonnements wieder kündigen. Im Bild entspricht der Softwarebus dem Verlag. Ein Abonnement wird unter der Angabe eines sog. „Themas“[2] eingerichtet. Dieses Thema entspricht dem Namen der Zeitschrift. Es ist möglich, ein Thema mehrfach zu abonnieren.

Das zweite Verfahren ermöglicht eine Punkt-zu-Punkt-Kommunikation. Statt eine Nachricht in der Zeitung zu veröffentlichen, kann ein Sender die Nachricht durch den Bus auch direkt in den Briefkasten des Empfängers bringen lassen. Man kann das mit der Antwort auf Chiffre-Anzeigen vergleichen. Der Sender kennt nicht die tatsächliche (Netzwerk-)Adresse des Empfängers, sondern nur die Chiffre. Die Chiffren entsprechen den „Briefkästen”[3] des Rendezvous-Busses.

Ein typischer Vorgang bei der Busbenutzung ist das Senden einer „Anfrage”[4]. Der Vorgang entspricht dem Aufgeben einer Chiffre-Anzeige. Dabei wird in einer bestimmten Zeitung eine Anzeige veröffentlicht und gleichzeitig eine Chiffre-Nummer bereitgestellt und dem Leser mitgeteilt.

„Briefkästen“ und „Abonnements“ werden zusammengefaßt zu sog. „Endpunkten“. Jedem Endpunkt ist eine eindeutige Identifikationsnummer zugeordnet. Die Bezeichnung Endpunkt kommt wohl daher, daß der Transport einer Nachricht immer an Endpunkten endet. Ein Endpunkt kann verglichen werden mit einem Posteingangskorb, der für bestimmte Sendungen reserviert ist. Ein wichtiges Merkmal aller Endpunkte ist, daß man den Bezug der dort eingehenden Nachrichten unter Angabe der Identifikationsnummer beim Bus kündigen kann.

---

[1] Diese Datei befindet sich im Unterverzeichnis „include“ des Rendezvous-Installations-Verzeichnisses.

[2] englisch: subject

[3] englisch: inbox

[4] englisch: request

Der Rendezvous-Bus stellt Routinen zur Erzeugung und Bearbeitung von Nachrichten, zum Versand von Nachrichten und zur Einrichtung und Kündigung von Endpunkten zur Verfügung. Die Routinen bilden das RV-API[1], das sich wie folgt untergliedern läßt:

- Advisory API
- Event-Manager API
- Communications API
- Message API

Das Advisory API wird benötigt im Zusammenhang mit den Bus selbst betreffenden Fehlermeldungen und Nachrichten. Es wurde nicht benutzt und bleibt in dieser Arbeit unberücksichtigt.

Das Event-Manager API ist eine Schnittstelle zwischen den für die eigentliche Busfunktionalität zuständigen Komponenten und dem Ereignisverteiler. Es bietet die Möglichkeit, statt einem der mitgelieferten einen eigenen Ereignisverteiler zu benutzen. Dies ist angebracht, wenn der Bus als Bestandteil eines Systems eingesetzt werden soll, das nicht nur auf Busereignisse, sondern auch auf andere Ereignisse reagieren soll, der mitgelieferte Windows-Event-Manager aber nicht verwendet werden kann oder soll.

Die für die Entwicklung einer Eiffel-Anwendung mit grafischer Bedienoberfläche notwendige Komponente „Eiffel Build" fehlte zunächst in der Eiffel-Entwicklungsumgebung noch und wurde erst viel später nachgeliefert. Daher war es anfangs nicht möglich, eine grafische Oberfläche zu implementieren. Das Problem der Reaktion auf nicht vom Bus stammende Ereignisse stellte sich daher nicht. Auf Grund dessen wurde bisher die Möglichkeit, einen eigenen Ereignisverteiler zu implementieren, nicht weiter verfolgt. Für die Fortsetzung des Projekts bietet sich hier ein Ansatzpunkt.

Zum Communications API gehören Routinen, die die Veröffentlichung und den Empfang von Nachrichten über den Bus ermöglichen.

Das Message API enthält Routinen zum Erzeugen und Bearbeiten der Nachrichten.

[1] RV-API = Rendezvous Application Programmers Interface – Bezeichnung der C-Routinenbibliothek, mit Hilfe derer der Rendezvous-Software-Bus benutzt wird.

3.3 Die Kapselung des RV-Busses in Eiffel

3.3.1 Struktur eines busbenutzenden Systems

Das Ziel der Kapselung des Busses ist nicht die Konstruktion eines vollständigen Systems. Es soll lediglich die Funktionalität des Busses in Eiffel zur Verfügung gestellt werden. Der Programmierer eines busbenutzenden Eiffel-Systems braucht sich so nicht mehr mit C-Routinen zu beschäftigen, sondern findet eine Eiffel-Klassenbibliothek vor, die ihm die Busbenutzung ermöglicht.

Man betrachte das Bild 3.2. Dort ist eine Aufteilung eines hypothetischen busbenutzenden Systems in vier Akteure vorgenommen worden:

- Anwendungsakteur
- Kapselungsakteur
- Busakteur
- Hilfsroutinen

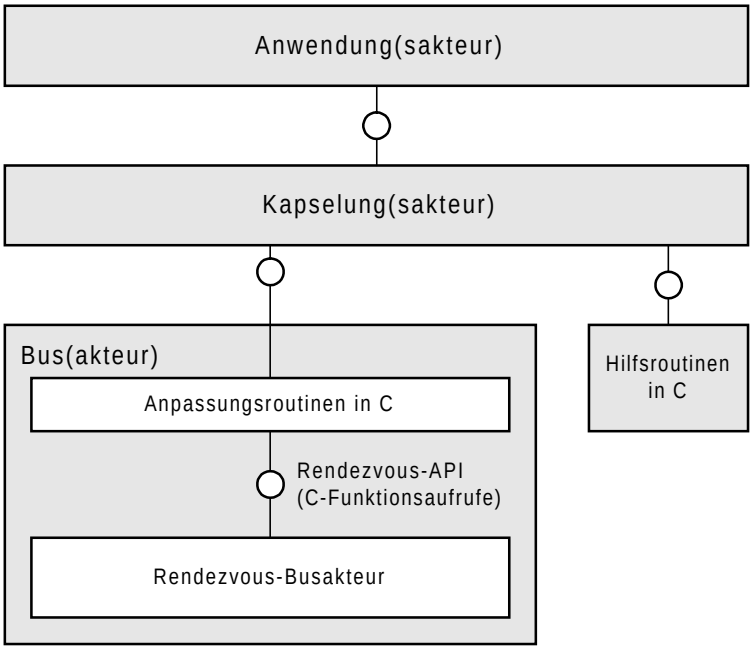


Bild 3.2 Akteure eines busbenutzenden Systems

Der Busakteur läßt sich weiter aufteilen in den Rendezvous-Busakteur und in die Anpassungsroutinen. Die Akteure entsprechen Teilen der Software: Der Rendezvous-Busakteur entspricht der von TIBCO gelieferten Busbibliothek (RV-API). Bei den

Anpassungsroutinen handelt es sich um selbst programmierte C-Routinen, die der Anpassung der Busbibliothek an die C-Schnittstelle von Eiffel dienen.

Die Hilfsroutinen, für die der Akteur rechts unten steht, dienen der Bearbeitung von Daten in C-Repräsentation. Dazu gehört insbesondere die Reservierung und Freigabe von C-Speicherplatz, aber auch beispielsweise eine Kopierprozedur.

Zum „Kapselungs-Akteur“ werden alle in Eiffel geschriebenen Komponenten der Buskapselung zusammengefaßt. Obwohl die in C geschriebenen Komponenten auch der Aufgabe der Kapselung des Busses dienen, soll im folgenden unter „Kapselung“ nur der Eiffel-Teil verstanden werden. Die den Kapselungsakteur bildenden Eiffel-Klassen werden entsprechend „Kapselungs-Klassen“ genannt.

Der Quelltext der Anpassungs- und Hilfsroutinen sowie der Kapselungsklassen findet sich im Anhang.

Die Anwendung entspricht dem unbekannten Teil des Gesamtsystems, das ein zukünftiger Entwickler konstruieren wird.

Die Akteure werden im folgenden der Einfachheit halber nur „Anwendung“, „Bus“ und „Kapselung“ genannt.

3.3.2 Anpassungsroutinen in C

Es ist nicht möglich, das RV-API nahtlos an Eiffel anzubinden. Dies liegt daran, daß nicht alle C-Typen in Eiffel verfügbar sind und deswegen Typumwandlungen (casts) nötig sind. Außerdem besteht das Problem, daß Eiffel Routinenaufrufe nur unter Angabe eines Objekts zuläßt. Beide Probleme wurden weiter oben in Abschnitt 3.1.3 schon angesprochen.

Das RV-API wurde um einige in C geschriebene Routinen erweitert, die die nötigen Anpassungen übernehmen. Es handelt sich dabei mit einer Ausnahme[1] um reine Durchreicher-Routinen, die keine eigene Funktionalität bieten.

3.3.3 Überlegungen zur Gestaltung der Kapselung

Ein Leitprinzip beim objektorientiertem Entwurf ist die Orientierung an den Daten bei der Modularisierung der Software: Routinen werden um die von ihnen bearbeiteten Daten gruppiert.

---

[1] Bei der Ausnahme handelt es sich um den Nachrichtenverteiler, der im folgenden Abschnitt beschrieben wird. Aber auch er ist sehr simpel implementiert.

C ist keine objektorientierte Sprache, und so entspricht das RV-API nicht diesem Prinzip. Das RV-API ist eine Sammlung von Routinen, ohne enge Bindung an Daten oder Objekte. Grundsätzlich bewegt man sich bei der Kapselung einer solchen Routinensammlung zwischen zwei entgegengesetzten Extremen.

Das eine Extrem wäre die Zuordnung aller Routinen zu einem einzigen Objekt. Dieses Objekt wäre ein Monopolakteur, der an seiner Schnittstelle als reiner Durchreicher die gesamte Funktionalität des RV-APIs bereitstellen würde. Dieses Objekt hätte keine eigenen Attribute.

Das andere Extrem wäre eine objektorientierte Neukonstruktion, die intern eine Übersetzung in C-Routinen-Aufrufe leisten müßte. Ein großer Teil der Zustandsinformationen des Busses und insbesondere der Inhalt von Nachrichten müßte zusätzlich in einer objektorientierten Darstellung gespeichert werden.

Beide Extreme haben Nachteile. Im ersten Fall wäre es sehr unkomfortabel, den Bus anzusprechen und die rein prozedurale Struktur würde ihre Spuren bis weit hinein in die Busanwendung hinterlassen.

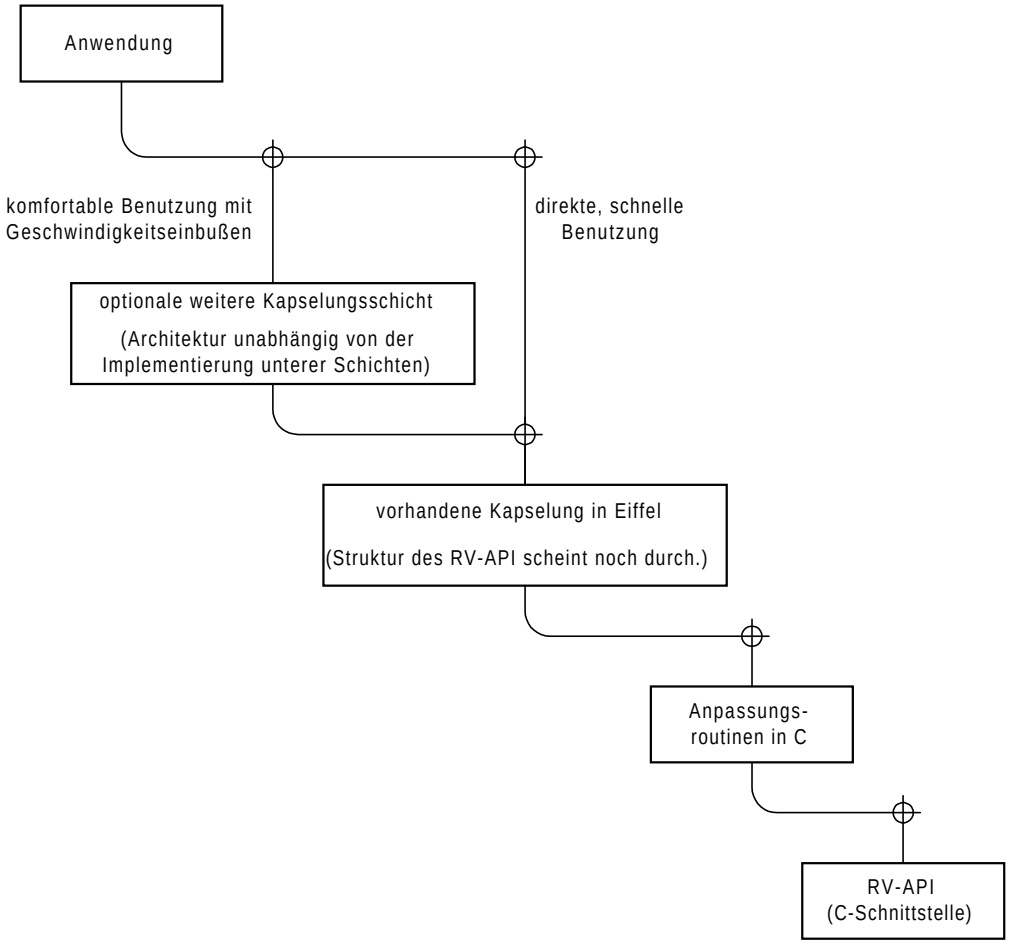


Bild 3.3 Mögliche Schichtung eines busbenutzenden Systems

Eine kompromißlose objektorientierte Kapselung hätte den Nachteil, daß – bildlich gesprochen – die Kapsel sehr dick würde. Ein höherer Speicherplatzbedarf und Geschwindigkeitseinbußen müßten in Kauf genommen werden.

Ich habe mich für einen Kompromiß entschieden: Die Kapselung sollte so objektorientiert sein, wie es möglich ist, ohne zum RV-API ganz quer zu liegen. Diese Entscheidung hat den Vorteil, daß es ohne weiteres möglich ist, eine weitere Hülle vorzusehen, die dann eine noch strenger objektorientierte und komfortablere Schnittstelle bieten kann. In zeitkritischen Fällen bleibt dann immer noch die Möglichkeit, auf die Dienste der inneren Hülle zuzugreifen. Die Schichtung der Schnittstellen zeigt Bild 3.3.

Nur eine Sitzung gleichzeitig

Der RV-Bus ermöglicht gleichzeitig mehrere Sitzungen in einem Prozeß. Dies kann nützlich sein für nebenläufig abgewickelte Programme mit mehreren Ereignis-Verteil-Schleifen. Da Eiffel-Programme streng sequentiell abgewickelt werden, gibt es keinen Grund für mehrere gleichzeitige Sitzungen. Die Kapselung erlaubt daher nur eine Sitzung zur gleichen Zeit.

3.3.4 Akteure und Entitäten der Kapselung

Eine Übersicht über die Kapselungsklassen zeigt das Bild I.[1]

3.3.4.1 Übersicht

Der Busverwalter

Man betrachte das Bild II.

Im oberen Teil befindet sich die Anwendung, im unteren Teil des Bildes befindet sich der Bus.

Dazwischen liegen die durch die Kapselungsklassen implementierten Bestandteile. Dies sind der Busverwalter und zwei grau hinterlegte Bereiche, die Nachrichten und Endpunkte darstellen. Der Busverwalter ist für das Öffnen und Schließen einer Bussitzung sowie für das Starten der Ereignis-Verteil-Schleife[2] verantwortlich.

[1] Die Bilder I und II befinden sich in der Tasche am Ende dieser Schrift.

[2] Siehe weiter unten.

Nachrichten

Das RV-Message API stellt Routinen zur Erzeugung und Bearbeitung von Nachrichten zur Verfügung. Zur Identifikation einer Nachricht wird ein Tripel aus Typ, Größe und Adresse (TGA-Tripel) der Nachricht verwendet. Eine Nachricht wird von einem Eiffel-Objekt der Klasse `RV_MSG_FIELD` repräsentiert. Dieses Objekt hat als Attribut das TGA-Tripel. Die eigentlichen Nachrichtendaten liegen nur in der vom RV-API vorgegebenen C-Repräsentation vor. Der gezeigte Nachrichtenakteur stellt der Anwendung Zugriffsfunktionen auf die Nachricht zur Verfügung, die denen des RV-APIs entsprechen. Im Wesentlichen besteht die Leistung des Nachrichtenakteurs darin, daß eine Nachricht von Eiffel aus nicht durch das TGA-Tripel, sondern durch einen normalen Objektverweis identifizierbar ist. Außerdem entlastet der Akteur die Anwendung von der Pflicht zur Übergabe der Sitzungskennung (Session-ID) an den Bus. Der Schreibpfeil vom Anwendungsakteur auf die gestrichelt umrandete Struktur „Nachricht" zeigt, daß die Anwendung Nachrichten erzeugen kann.

Endpunkte

Rechts im Bild sind die Endpunkte dargestellt. Die Struktur ist ganz analog zu der der Nachrichten. Auch hier dient ein einem Endpunkt zugeordnetes Eiffel-Objekt vornehmlich dem Zweck, einen Endpunkt durch einen Objektverweis identifizieren zu können. Gegenüber dem Bus wird ein Endpunkt durch die Identifikationsnummer `listen_id` identifiziert. Zusätzlich zur `listen_id` ist in der Eiffel-Repräsentation das Attribut `callback` vorgesehen. `Callback` enthält einen Verweis auf den dem Endpunkt zugeordneten Empfänger. Die Speicherung des Verweises in `callback` ist notwendig, weil er bei der Kündigung eines Endpunkts an den Bus übergeben werden muß.

Dies zu erklären bedarf es eines Exkurses:

> Es ist bekannt, daß Objekte, auf die es keinen Verweis mehr gibt, von der Speicherbereinigung zerstört werden können. Das kann auch mit Empfänger-Objekten geschehen. Das Ansprechen des nicht mehr vorhandenen Objekts vom RV-Bus aus wäre dann fatal.
>
> Solange an einem Endpunkt Nachrichten eingehen können, muß der zuständige Empfänger existieren. Um sicherzustellen, daß der Empfänger nicht verschwindet, solange ein zugehöriger Endpunkt noch existiert, wird bei der Einrichtung eines Endpunkts der Empfänger von C aus geschützt.
>
> Dies geschieht durch den Aufruf der CECIL-Prozedur `eif_adopt`[1]. Für die Einrichtung eines Endpunkts wird ein entsprechender Auftrag an den Bus erteilt. Die zuständige Anpassungsroutine[2] reicht diesen Auftrag durch. Bei dieser Gelegenheit

[1] Für die CECIL-Prozeduren vgl. Abschnitt 3.1.3.3.3.

[2] Es handelt sich um die Anpassungsroutinen eifrv_listen_subject, eifrv_listen_inbox und eifrv_rpc.

ruft sie `eif_adopt` auf und schützt dadurch den Empfänger vor der Speicherbereinigung. Bei der Kündigung eines Endpunkts[1] wird mit der CECIL-Prozedur `eif_wean` das Callback-Objekt wieder freigegeben. Den Anpassungsroutinen ist dazu jeweils ein Verweis auf das Callback-Objekt zu übergeben.

Das Attribut `callback` eines Endpunkt-Objekts allein reicht zum Schutz des Empfängers nicht aus, da das Endpunkt-Objekt selbst gelöscht werden könnte. Dann wäre es zwar nicht mehr möglich, diesen Endpunkt zu kündigen, aber das ist nicht unbedingt nötig. Denn durch die Beendigung einer Sitzung werden implizit alle Endpunkte gekündigt.

3.3.4.2 Nachrichten

Der Software-Bus stellt einen Vorrat an elementaren Datentypen zur Verfügung. Neben Ganzzahlen, Gleitkommazahlen, Zeichenketten und boole'schen Werten gibt es Datentypen für IP-Adressen, die Zeit, verschlüsselte Nachrichten und für Bytefolgen. Der Bus sorgt dafür, daß Daten dieser Typen über das Netzwerk auch zwischen unterschiedlichen Plattformen richtig übertragen werden. Bei den Zahlentypen gibt es Varianten mit unterschiedlicher Größe (Ganzzahlen) bzw. Genauigkeit (Gleitkommazahlen).[2]

Man betrachte das ER-Diagramm in Bild 3.4.

Eine Rendezvous-Nachricht wird durch ein Objekt der Klasse „Nachrichtenfeld" repräsentiert. Dieses Nachrichtenfeld kann entweder ein einfaches Feld sein oder ein Listenfeld. Ein einfaches Feld enthält einen der oben aufgezählten elementaren Datentypen, z.B. eine Ganzzahl. Ein Listenfeld dagegen enthält eine Liste von Nachrichtenfeldern. Die in dieser Feldliste enthaltenen Nachrichtenfelder können selbst wieder Listenfelder sein, so daß eine Nachricht einen geschachtelten Aufbau haben kann.

Im ER-Diagramm ist erkennbar, daß sich der Inhalt eines Nachrichtenfelds in einem C-Speicherplatz (also nicht in Attributen von Eiffel-Objekten) befindet. Das TGA-Tripel identifiziert diese C-Repräsentation des Inhalts.

Jedem Nachrichtenfeld kann ein Name gegeben werden. Der Zugriff auf Felder in einer Liste sollte durch den Namen erfolgen und nicht durch die Position in der Liste. Das hat den Vorteil, daß es möglich ist, die Struktur einer Nachricht durch Hinzufügen neuer Felder zu verändern. Nachrichtenempfänger müssen an eine solche Änderung nicht angepaßt werden, wenn sie die neuen Felder ignorieren dürfen. Vgl. hierzu [RVProg].

---

[1] Die zuständige Anpassungsroutine ist eifrv_close.

[2] In der Kapselung wurden bisher nur die Zahlentypen, boole'sche Werte und Zeichenketten implementiert.

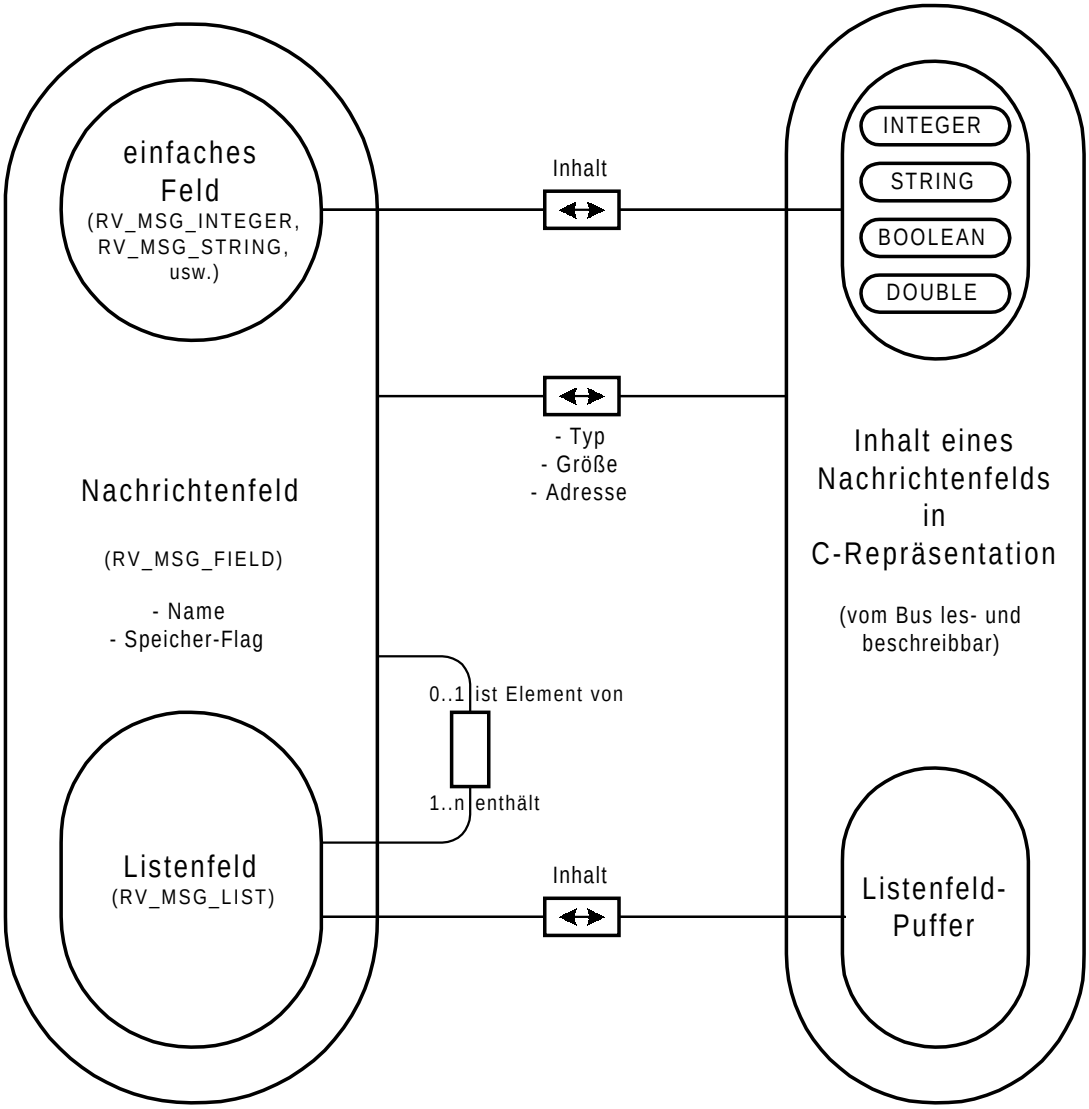


Bild 3.4 Struktur einer Nachricht

3.3.4.2.1 Einfache Nachrichtenfelder

Man betrachte Bild 3.5. Oben sind zwei Akteure dargestellt: die Anwendung und der Busverwalter. Der grau unterlegte Bereich stellt ein einfaches Nachrichtenfeld dar. Im Zentrum liegt der Feld-Akteur. Eine wichtige Komponente des Feld-Akteurs ist der Initialisierungsakteur, der seine Rolle bei der Erzeugung des Nachrichtenfelds spielt.

> Hier sei ein kurzer Exkurs erlaubt. Wie kann ein Akteur eine Rolle spielen bei der Erzeugung seiner selbst? Die Erzeugung eines Eiffel-Objekts kann man in zwei Phasen einteilen. In der ersten Phase erzeugt der Eiffel-Abwickler aufgrund einer entsprechenden Anweisung ein „nacktes" Objekt einer Klasse. Alle Attribute des Objekts sind mit Nullwerten vorbelegt, Größen für Objektverweise sind leer. In der zweiten Phase wird eine Initialisierungsroutine des „nackten" Objekts aufgerufen.

Diese Routine kann nun Attribute initialisieren und andere Initialisierungsoperationen durchführen. Vor der Abwicklung der zweiten Phase braucht die Klasseninvariante noch nicht erfüllt zu sein. Man kann den Standpunkt vertreten, daß es sich nach dem Abschluß der ersten Phase noch nicht um ein Objekt der Klasse handelt. Die zweite Phase gehört damit noch zur Objekterzeugung, auch wenn sie schon durch Routinen des Objekts ausgeführt wird.

Zurück zum Bild. Ganz unten erkennt man den C-Speicherplatz, der den eigentlichen Inhalt des Felds enthält. Der Feldakteur greift darauf über die C-Hilfsroutinen zu. Die Hilfsroutinen führen Typumwandlungen zwischen der C-Repräsentation und der Eiffel-Darstellung des Feldinhalts durch.

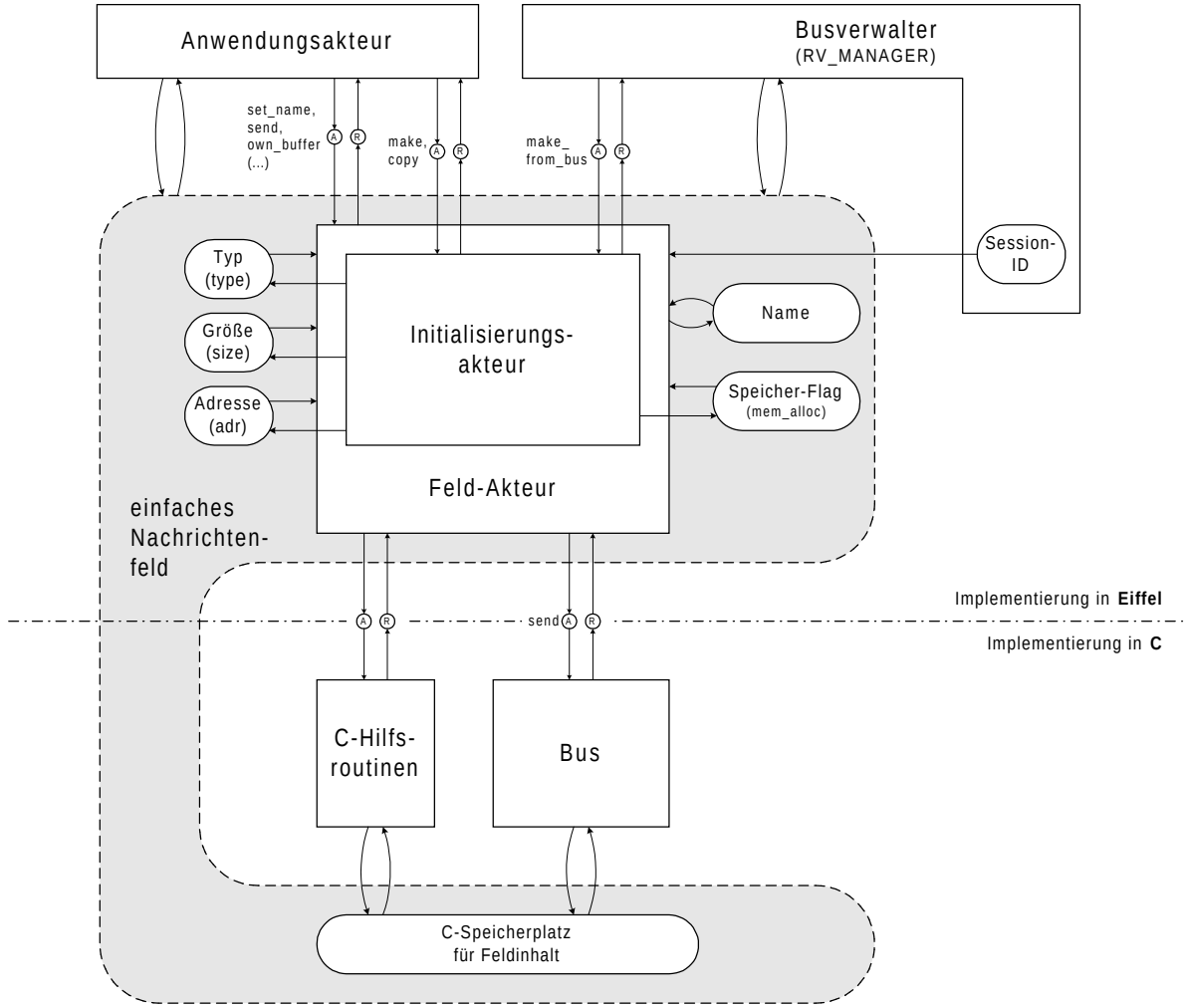


Bild 3.5 Aufbaubild: Erzeugung und Benutzung eines einfachen Nachrichtenfelds

Die Erzeugung eines Felds erfolgt abhängig von der Übertragungsrichtung der Nachricht auf zweierlei Weise. Soll eine Nachricht gesendet werden, so wird sie in der Regel[1] vorher von der Anwendung erzeugt. Die C-Repräsentation wird dann vom Initialisierungsakteur erzeugt. Dafür ist die Reservierung von C-Speicherplatz und eine Typumwandlung von Eiffel nach C nötig, wofür die C-Hilfsroutinen benutzt werden.

Im anderen Fall wird eine Nachricht vom Bus empfangen. Die C-Repräsentation liegt dann schon vor, lediglich das zugehörige Eiffel-Objekt muß erzeugt werden. Diese Aufgabe erledigt der Busverwalter. Er ruft dazu die Prozedur `make_from_bus` des Initialisierungsakteurs auf. Die Anwendung benutzt dagegen die Prozedur `make`.

Der für den Feldinhalt bereitgestellte C-Speicher muß irgendwann wieder freigegeben werden. Wurde er durch den Bus bereitgestellt, so wird er auch durch diesen freigegeben. Entsprechend muß der Speicher von Eiffel aus freigegeben werden, wenn das Feld durch die Anwendung erzeugt wurde. Das Speicher-Flag im Bild rechts neben dem Feld-Akteur dient der Unterscheidung der beiden Fälle.

Der Bus weiß nicht, ob ein Feld von Eiffel noch gebraucht wird. Er legt den Zeitpunkt selbst fest, an dem der Speicher wieder freigegeben wird. Eine weitere Benutzung des Felds auf Eiffel-Seite muß danach unterbleiben. Empfangene Nachrichten bleiben gültig bis zur Beendigung der empfangenden Callback-Prozedur (siehe unten). Werden Felder über diese Zeitspanne hinaus gebraucht, so muß eigener Speicher reserviert werden. Dazu dient die Prozedur `own_buffer` des Initialisierungsakteurs.

Das Attribut „Name” existiert nicht in der C-Repräsentation eines Nachrichtenfelds. Bei der Erstellung von Feldlisten wird allerdings jedes Feld der Liste mit einem Namen assoziiert, so daß die Nachrichtenfeld-Objekte ein geeigneter Ort für dieses Attribut sind.

Hingewiesen sei noch auf die Sitzungskennung, die der Feldakteur für die Kommunikation mit dem Bus benötigt und dafür vom Busverwalter erhält.

Durch die Prozedur `copy` ist es möglich, den Inhalt und den Namen eines anderen Feldes zu übernehmen.[2]

Zum Senden des Feldinhalts dient die Prozedur `send` des Feldakteurs. Der Inhalt des Felds wird dann auf dem Bus unter dem durch das Attribut Name spezifizierten Thema veröffentlicht. Dazu erteilt der Feld-Akteur dem Bus einen Auftrag.

---

[1] Es ist auch möglich, eine empfangene Nachricht wieder zu versenden.

[2] Genaueres über die Semantik von „copy“ entnehme man [EiffelLib].

3.3.4.2.2 Listenfelder

Ein Listenfeld ist ganz analog zu einfachen Nachrichtenfeldern aufgebaut. Sowohl einfache Nachrichtenfelder wie auch Listenfelder gehören zur gemeinsamen Oberklasse Nachrichtenfeld (`RV_MSG_FIELD`). Listenfelder haben einen größeren Funktionsumfang und ein weiteres Attribut. Man betrachte Bild 3.6. Auf die Darstellung der Erzeugung wurde hier verzichtet, weshalb der Initialisierungsakteur entfällt. Die C-Repräsentation des Feldinhalts findet man links unten innerhalb des grau unterlegten Ovals als Listenfeld-Puffer wieder. Der Zugriff auf diesen Puffer erfolgt nicht über die C-Hilfsroutinen, sondern über den Bus. Für die Bearbeitung von Listenfeldern sind die C-Routinen des Rendezvous-Message-API's vorgesehen.

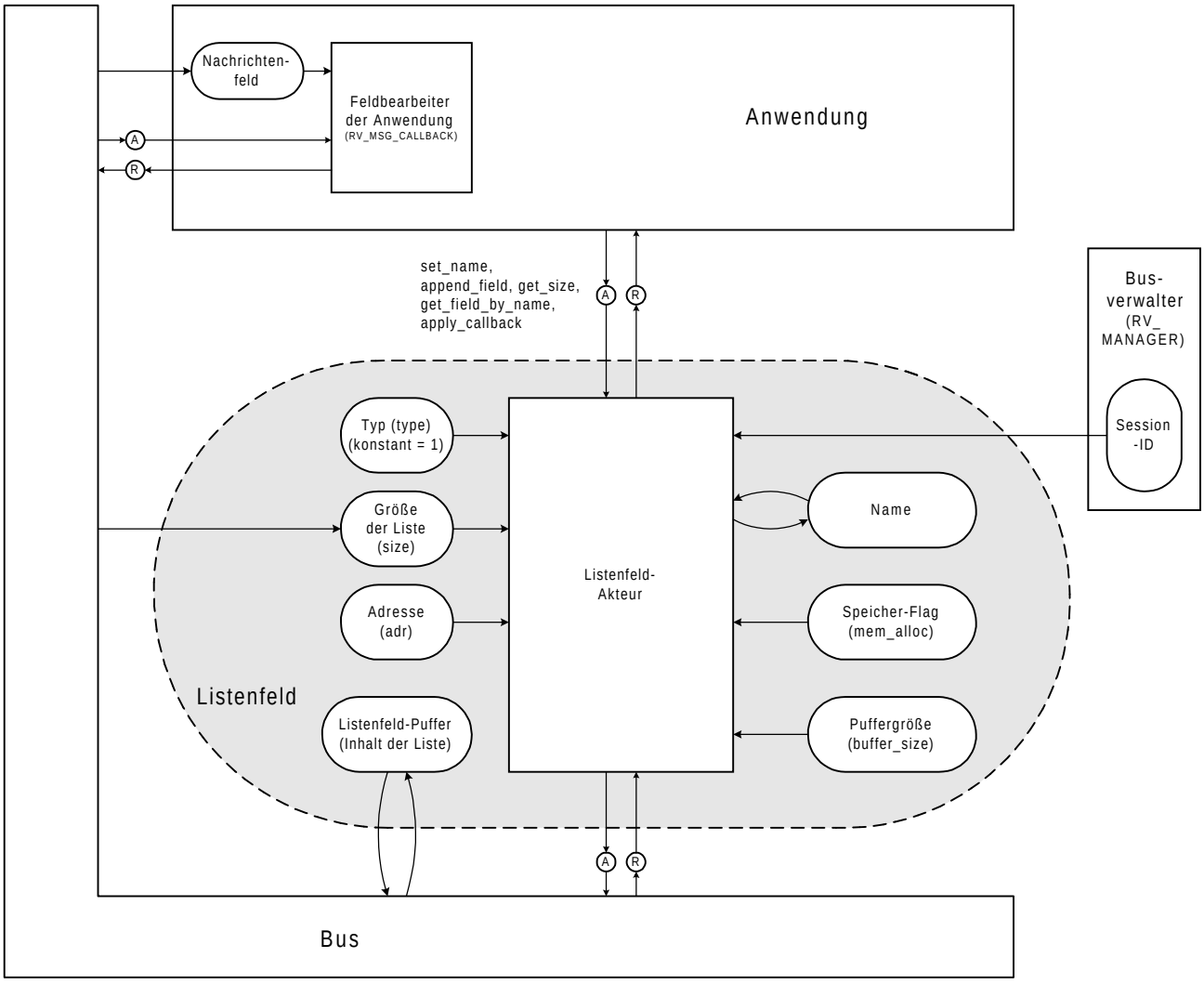


Bild 3.6 Aufbaubild Listenfeld-Bearbeitung
(Die Erzeugung einer Feldliste sowie der Versand sind hier nicht berücksichtigt)

Dieser Puffer kann größer sein als sein Inhalt. Ein Listenfeld enthält nach der Erzeugung zunächst eine leere Liste. Die Größe des Puffers wird bei der Erzeugung festgelegt. An die

Liste können Felder angehängt werden, solange der Puffer noch genügend Platz bietet. Es ist also zu unterscheiden zwischen der Puffergröße im Attribut `buffer_size` und der Größe der Feldliste im bekannten Attribut `size` des TGA-Tripels.

Das ein Listenfeld repräsentierende Eiffel-Objekt hat eine Prozedur (`new_buffer_size`) zur nachträglichen Vergrößerung des Puffers. Die Benutzung dieser Prozedur ist jedoch vergleichsweise teuer, da der gesamte Feldinhalt kopiert werden muß.

Im oberen Teil des Bildes ist als Bestandteil der Anwendung der Feldbearbeiter zu sehen. Dieser spielt seine Rolle beim Parsen einer Feldliste. Normalerweise sollte auf die Felder einer Liste über ihren Namen zugegriffen werden. Es ist aber auch möglich, auf alle Felder der Liste der Reihe nach zuzugreifen. Dazu ruft die Anwendung die Prozedur apply_callback des Listenfeld-Akteurs auf. Dieser gibt den Auftrag an den Bus weiter. Der Bus ruft den Feldbearbeiter der Anwendung jeweils einmal für jedes Feld der Liste auf und übergibt ihm so die Felder der Reihe nach. Es handelt sich um einen Callback-Mechanismus, der ganz analog zu dem weiter unten[1] beschriebenen realisiert ist.

3.3.4.2.3 Veröffentlichung von Nachrichten

Es gibt zwei Methoden, um Nachrichten zu veröffentlichen. Die einfachste ist, die Prozedur „`send`“ eines Nachrichtenfelds aufzurufen.

Die andere Methode ist, eine „Anfrage“ zu stellen. Bei der Erzeugung eines Exemplars der Klasse „Anfrage“ wird zugleich eine Nachricht veröffentlicht, die der Erzeugungsanweisung als Parameter übergeben wird.

Das Thema, unter dem die Nachricht veröffentlicht wird, ist bei beiden Methoden durch das Attribut „`name`“ des Nachrichtenfelds bestimmt.

3.3.4.3 Endpunkte

Man betrachte Bild 3.7. Für die Entitäten „Abonnement”, „Briefkasten” und „Anfrage” gibt es in der Kapselung jeweils eine Klasse[2]. Alle drei Klassen sind Unterklassen von Endpunkt.

Anfragen

Zur Erinnerung: Das Senden einer „Anfrage“ kann man vergleichen mit dem Aufgeben einer Chiffre-Anzeige. Eine Anfrage ist damit zunächst die Veröffentlichung einer Nachricht auf dem Bus. Sie ist aber untrennbar verbunden mit der Bereitstellung einer Chiffre-Nummer, was der Einrichtung eines Endpunkts entspricht. Obwohl die

---

[1] Abschnitt 3.3.5.2

[2] Dies sind die Klassen RV_LISTENER, RV_INBOX und RV_REQUEST.

Bezeichnung „Anfrage“ dies nicht impliziert, hat eine Anfrage die Eigenschaften eines Endpunkts. Ich verwende die Bezeichnung „Anfrage“, um konsistent zu bleiben mit der beim Rendezvous-Bus benutzten Bezeichnung „request“. In der Kapselung gibt es keine Objekte, die Veröffentlichungen repräsentieren. Daher wurden Anfragen den Endpunkten als Unterklasse zugeschlagen. Bei der Erzeugung einer Anfrage wird zwangsläufig eine Nachricht veröffentlicht.

Jede eingehende Nachricht ist mit einem Objekt einer der drei Unterklassen von Endpunkt assoziiert. Es ist möglich, den Eingang von Nachrichten an einem Endpunkt zu stoppen, indem man den Endpunkt kündigt. Die Objekte der Klasse `RV_ENDPOINT` haben das Attribut `listen_id`, das dazu dient, den Endpunkt gegenüber dem Bus zu identifizieren. Endpunkte werden durch die Anwendung erzeugt und gekündigt.

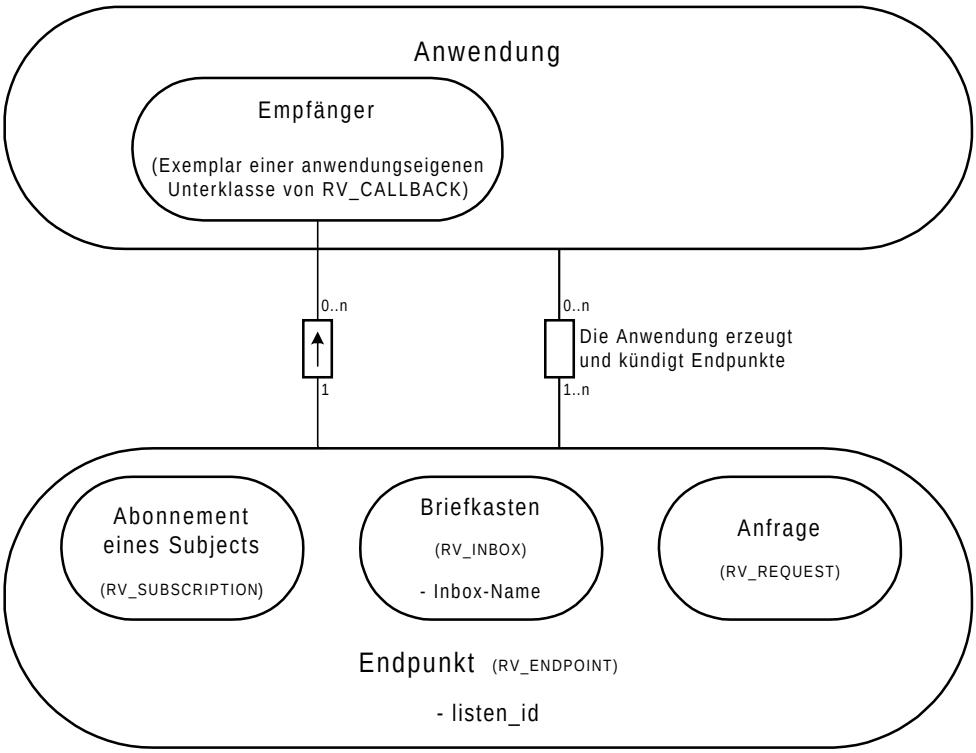


Bild 3.7 Beziehungen zwischen Endpunkten, Empfängern und der Anwendung

Durch die Erzeugung eines „Briefkastens“ läßt man sich – bildlich gesprochen – eine Chiffre-Nummer geben, ohne eine Anzeige aufzusetzen. Im Gegensatz zur „Anfrage” muß man dann natürlich die eigene Chiffre kennen, um sie anderen mitteilen zu können. Deshalb hat die Klasse `RV_INBOX` das Attribut `inbox_name`. Der Sender einer Nachricht benutzt `inbox_name` als Thema, um den Briefkasten als Empfänger zu bestimmen.

3.3.4.4 Empfänger

Empfänger-Akteure werden realisiert durch Exemplare anwendungseigener Unterklassen der Kapselungsklasse RV_CALLBACK. Die Aufgabe eines Empfängerakteurs ist durch die Anwendung bestimmt. Der Empfänger hat aber in jedem Fall die Fähigkeit, Nachrichten vom Bus zu empfangen.

Die Klasse RV_CALLBACK ist aufgeschoben (vgl. Abschnitt 2.3.6). Sie hat eine Prozedur mit Namen `callback_general`. Nach ihrem Aufruf ruft diese Prozedur ihrerseits eine andere, aufgeschobene Prozedur (der gleichen Klasse) namens `callback_special` auf. Dieser Ablauf ist in Bild 3.8 gezeigt.

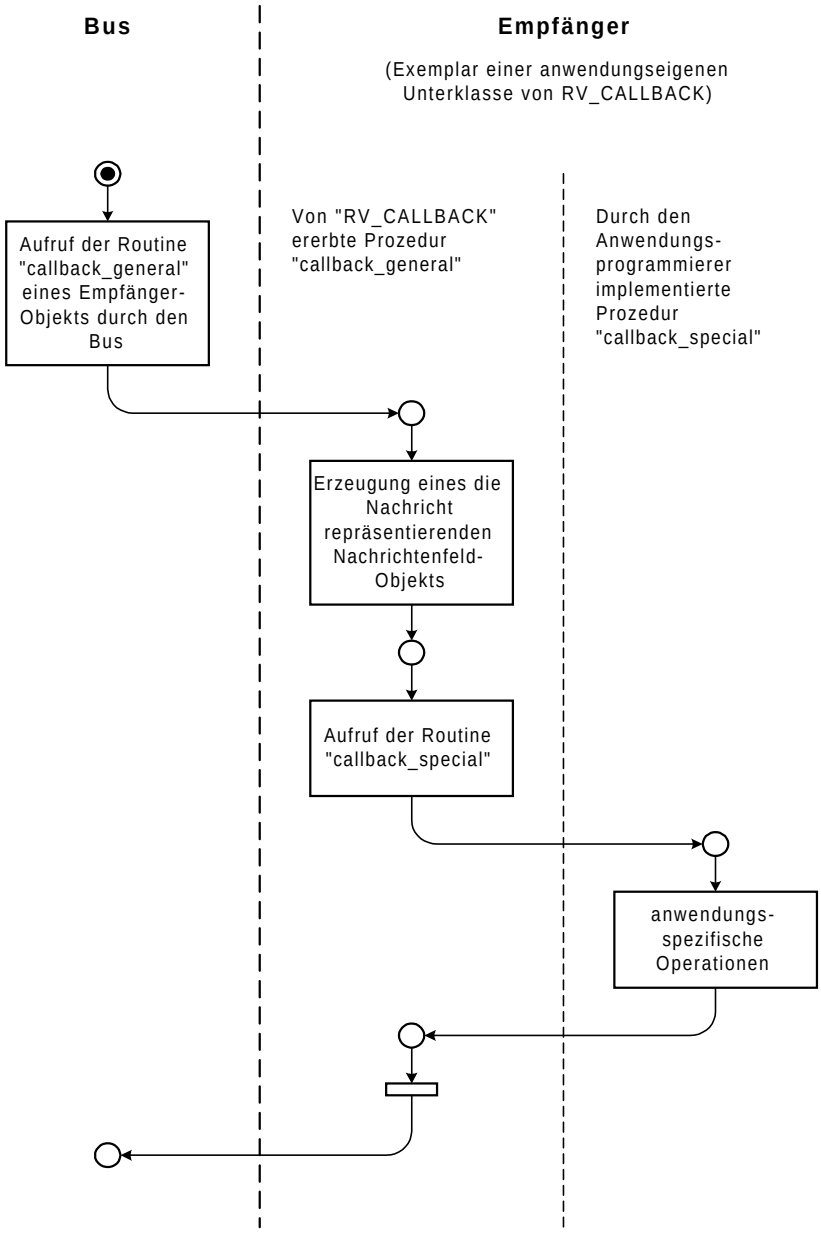


Bild 3.8 Nachrichtenempfang durch Empfänger-Akteur der Anwendung

Die Implementierung von `callback_special` bleibt den Nachkommen der Klasse `RV_CALLBACK` vorbehalten. Unterschiedliche Callback-Routinen werden realisiert durch Unterklassen von `RV_CALLBACK`, die sich in der Implementierung von `callback_special` unterscheiden. Der Bus ruft immer die gleiche Prozedur `callback_general` auf. Unterschiedliche Reaktionen auf einen Callback-Aufruf resultieren aus der Abwicklung der Prozedur `callback_general` im Kontext unterschiedlicher Objekte. Welche Version von `callback_special` ausgeführt wird, hängt davon ab, zu welcher Unterklasse von `RV_CALLBACK` der Empfänger-Akteur gehört.

`RV_CALLBACK` hat ein Attribut `msg` für die empfangene Nachricht. Die Nachricht wird repräsentiert durch ein Exemplar der Klasse `RV_MSG_FIELD`.

3.3.5 Der Empfang von Nachrichten

3.3.5.1 Beteiligte Entitäten

Man betrachte Bild 3.9. Im Mittelpunkt steht die empfangene Nachricht. Eine „empfangene Nachricht“ ist zu unterscheiden von einer Nachricht. Eine Nachricht kann, nachdem sie gesendet wurde, keinem, einem oder mehreren Endpunkten zugehen. Dementsprechend ist einer Nachricht eine unbestimmte Zahl von „empfangenen Nachrichten“ zuzuordnen. Einer empfangenen Nachricht ist genau ein Empfänger zugeordnet. Dieser Empfänger kann jedoch Empfänger mehrerer Nachrichten sein. Ein Empfänger soll hier auch dann als ein solcher betrachtet werden, wenn er (noch) gar keine Nachrichten empfangen hat.

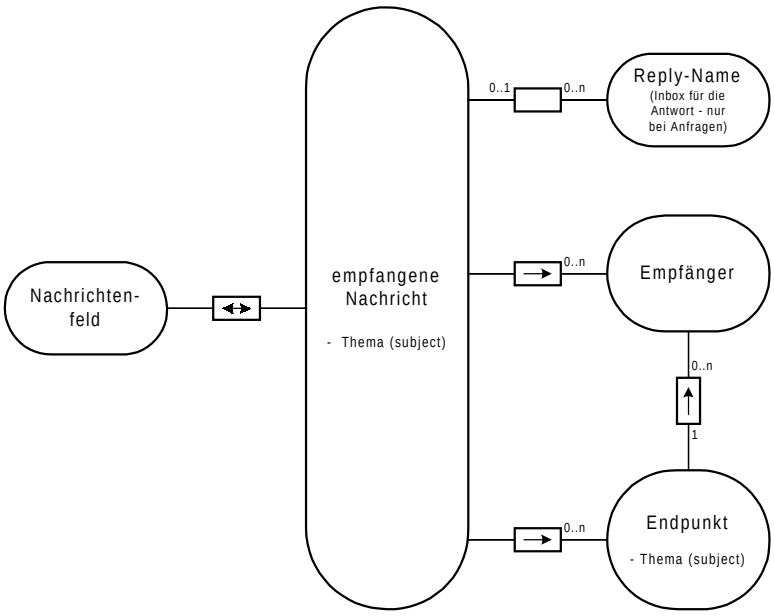


Bild 3.9 ER-Diagramm zum Nachrichtenempfang

Voraussetzung für das Eintreffen irgendwelcher Nachrichten ist das Bestehen eines Endpunkts. Die Einrichtung erfolgt durch die Erzeugung eines entsprechenden Objekts. Bei der Erzeugung wird jeder Endpunkt mit einem Empfänger assoziiert.

Unten rechts sieht man die Entität Endpunkt. Endpunkte haben das Attribut „Thema“. Einen Endpunkt erreichen genau die Nachrichten, deren Thema zu dem des Endpunkts paßt[1].

Wurde eine Nachricht beim Sender durch die Veröffentlichung einer Anfrage erzeugt, so wird mit ihr in Form des „`reply_name`” die Chiffre für eventuelle Antworten übertragen. Dies gilt nicht für anders veröffentlichte Nachrichten.

3.3.5.2 Der Callback-Mechanismus

Der Empfang von Nachrichten wird durch einen Callback-Mechanismus ermöglicht.

In einer Ereignis-Verteil-Schleife wartet der Bus auf eingehende Nachrichten. Sobald eine Nachricht eingeht, für die ein Abonnement besteht, ruft der Bus eine dem Abonnement zugeordnete Prozedur des angeschlossenen Systems auf. Eine solche Prozedur heißt Callback-Prozedur.

Das Programm eines busbenutzenden Systems besteht damit prinzipiell aus einem Initialisierungsteil zum Starten, der Ereignis-Verteil-Schleife (main loop) des Busses und aus einer Sammlung von Callback-Prozeduren, in der die eigentliche Funktionalität des Systems steckt.

Eiffel-Callback-Prozeduren

Die Callback-Prozeduren werden durch die Prozedur `callback_general` der Empfänger-Akteure implementiert.

Wie oben bereits beschrieben wurde, unterscheiden sich Empfänger durch die Implementierung von `callback_special`. Der Aufruf von `callback_general` muß im Kontext des richtigen Empfänger-Objekts erfolgen.

Es bleibt das Problem, wie die Zuordnung des Empfänger-Objekts von C aus erfolgt.

Die Signatur[2] einer Callback-Prozedur ist durch eine Deklaration im Message API festgelegt. Eiffel erwartet jedoch als erstes Argument eines jeden Routinenaufrufs einen Objektverweis. Es ist daher unumgänglich, eine C-Routine zu schreiben, die vom Bus aufgerufen wird und ihrerseits eine Eiffel-Prozedur mit dem nötigen Objektverweis aufruft. Es handelt sich dabei um den in Bild 3.10 gezeigten Callback-Verteiler.

---

[1] Es ist auch möglich, eine ganze Klasse von Themen zu abonnieren. Dies wird realisiert durch die Verwendung von Jokerzeichen (Wildcards). Siehe hierzu [Auer] und [RVProg].

[2] Signatur = Anzahl und Typen der Parameter und ggf. Typ des Ergebnisses einer Routine.

Woher weiß dieser Verteiler, an welchen Empfänger er den Aufruf weiterleiten soll?

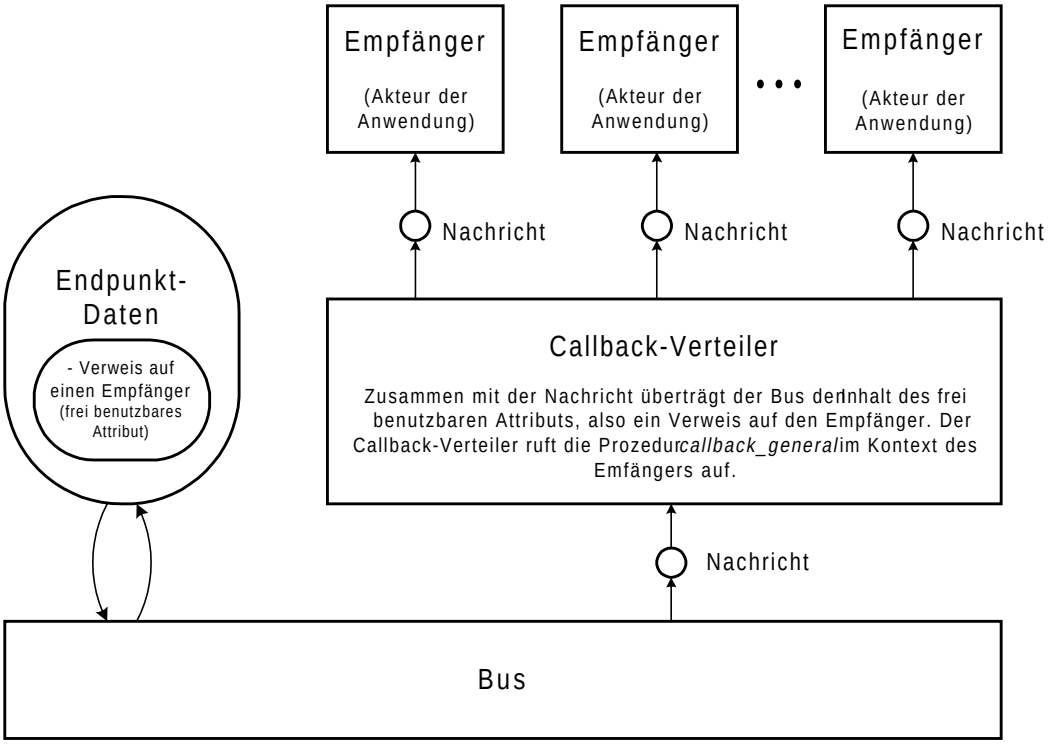


Bild 3.10 Der Callback-Verteiler

Der Bus speichert für jedes Abonnement einen Satz an Daten ab. Zu diesen Abonnement-Daten gehört ein vom Einrichtenden des Abonnements frei bestimmbares Attribut, welches später der vom Bus aufgerufenen Callback-Prozedur übergeben wird. Dieses Attribut wird zur Speicherung eines Verweises auf das für ein Abonnement vorgesehene Empfänger-Objekt benutzt. Der Callback-Verteiler erhält so bei jedem Aufruf vom Bus einen Verweis auf dasjenige Eiffel-Objekt, das die Nachricht bearbeiten soll.

Anhand von Bild 3.11 sollen die Vorgänge beim Nachrichtenempfang gezeigt werden. Beim Empfang der Nachricht ruft der Bus die Prozedur `callback_general`, für die der Akteur rechts unten innerhalb des Empfängers steht. Dabei wird das TGA-Tripel der C-Repräsentation der Nachricht übermittelt. `Callback_general` beauftragt den Busverwalter mit der Erzeugung des zugehörigen Eiffel-Objekts.

Soll der Empfänger den Inhalt des Felds lesen, so gibt er einen Leseauftrag an den Feldakteur.

Der Feld-Akteur greift dann mit Hilfe der C-Hilfsroutinen auf den C-Speicherplatz zu, der bereits vor dem Aufruf von `callback_general` vom Bus bereitgestellt und initialisiert wurde.

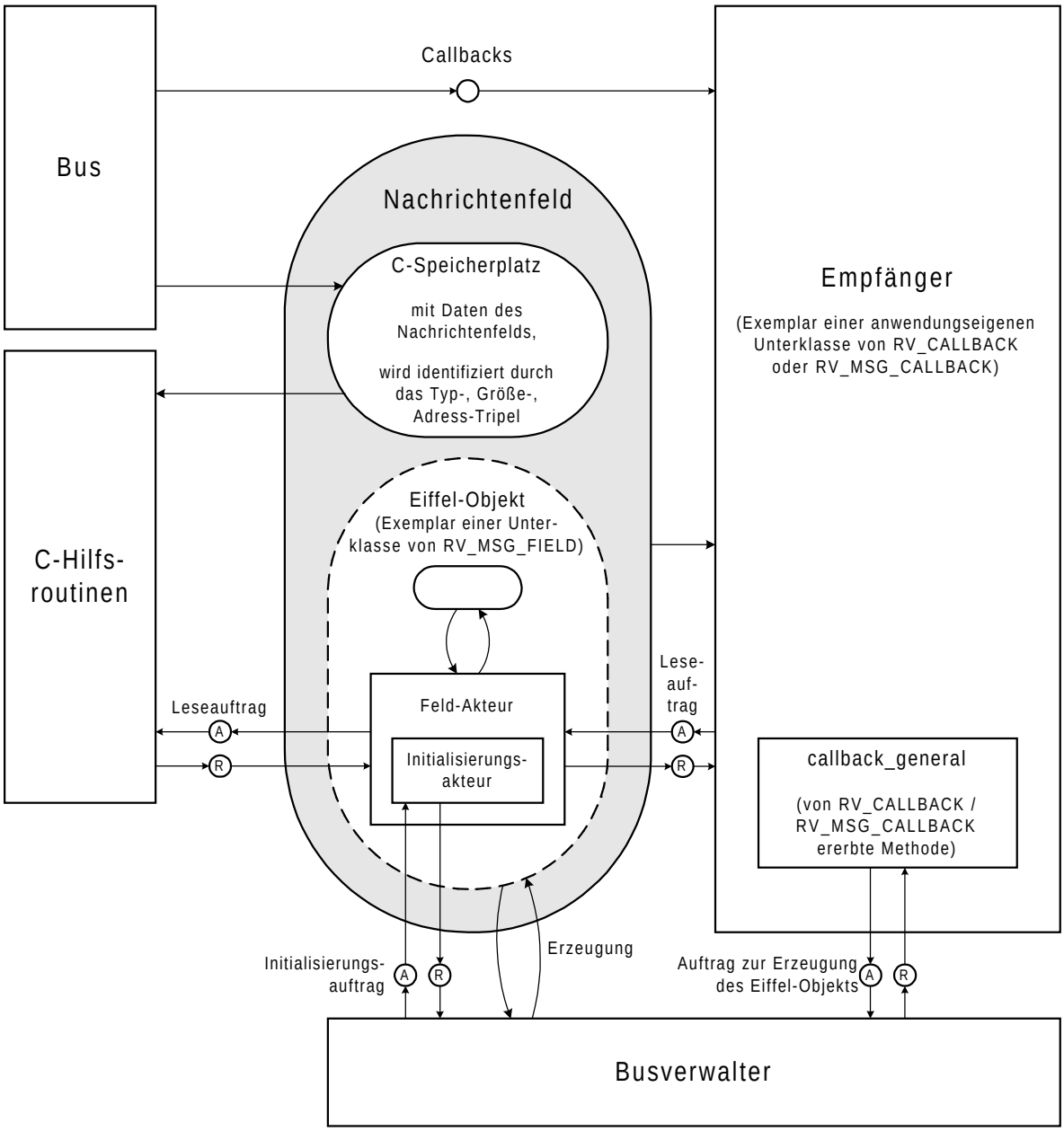


Bild 3.11 Übermittlung einer Nachricht vom Bus zum Empfänger-Akteur der Anwendung

3.3.6 Typische Abläufe der Busbenutzung

Nach der Einführung der wichtigsten Klassen der Kapselung möchte ich die Funktionsweise anhand einiger typischer Abläufe veranschaulichen.

3.3.6.1 Nachrichtenversand

Man betrachte das Petri-Netz in Bild 3.12. Nachrichten können nur innerhalb einer Bussitzung veröffentlicht werden, selbst das Erzeugen eines Listenfelds bedarf einer geöffneten Bussitzung. Zuallererst muß daher die Sitzung geöffnet werden. Dies geschieht durch einen Auftrag an den Busverwalter.

Die Existenz des Busverwalters kann von der Anwendung vorausgesetzt werden, denn der Busverwalter ist ein Monopolakteur[1], der durch eine Einmalfunktion[2] erzeugt wird. Anwendungsklassen können von der Klasse `RV_USER` erben, ihnen steht der Busverwalter durch das Merkmal „`rv`" zur Verfügung. Beim ersten Aufruf von `rv` – von wem auch immer dieser Aufruf kommt – wird der Busverwalter erzeugt.

Bevor eine Nachricht verschickt werden kann, muß ein Nachrichtenfeld erzeugt werden. Die Erzeugung von Nachrichtenfeldern wird im folgenden Abschnitt beschrieben.

Anschließend wird die Bussitzung durch einen Auftrag an den Busverwalter wieder geschlossen. Der Eintritt in die Ereignis-Verteil-Schleife des Busses ist nicht nötig, wenn keine Nachrichten empfangen werden sollen.

---

[1] siehe Abschnitt 2.3.4.1.

[2] siehe Abschnitt 2.3.9.

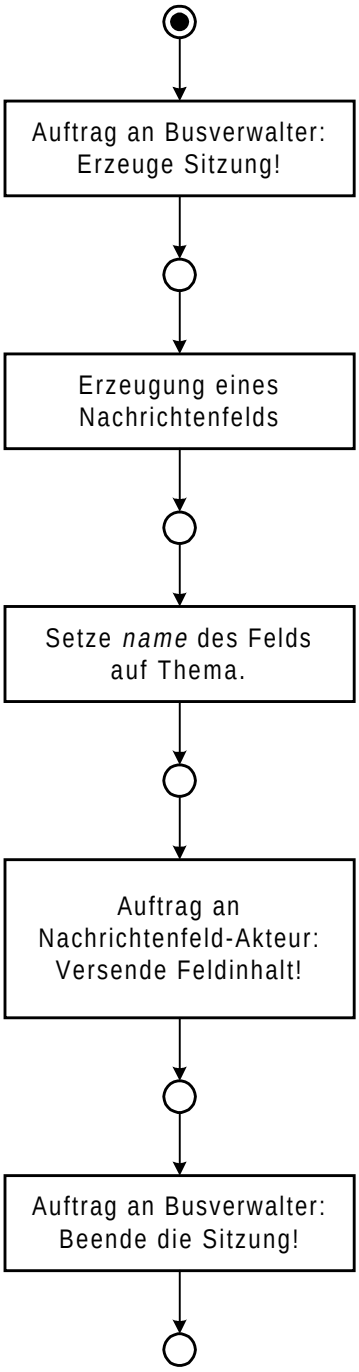


Bild 3.12 Versenden einer Nachricht

Das Erzeugen eines Nachrichtenfelds

Man betrachte das Petri-Netz in Bild 3.13. Es gibt zwei unterschiedliche Abläufe, je nachdem, ob ein einfaches oder ein Listenfeld erzeugt werden soll. Links ist der einfache Fall dargestellt – einem einfachen Nachrichtenfeld wird bei der Erzeugung der Inhalt vorgegeben.

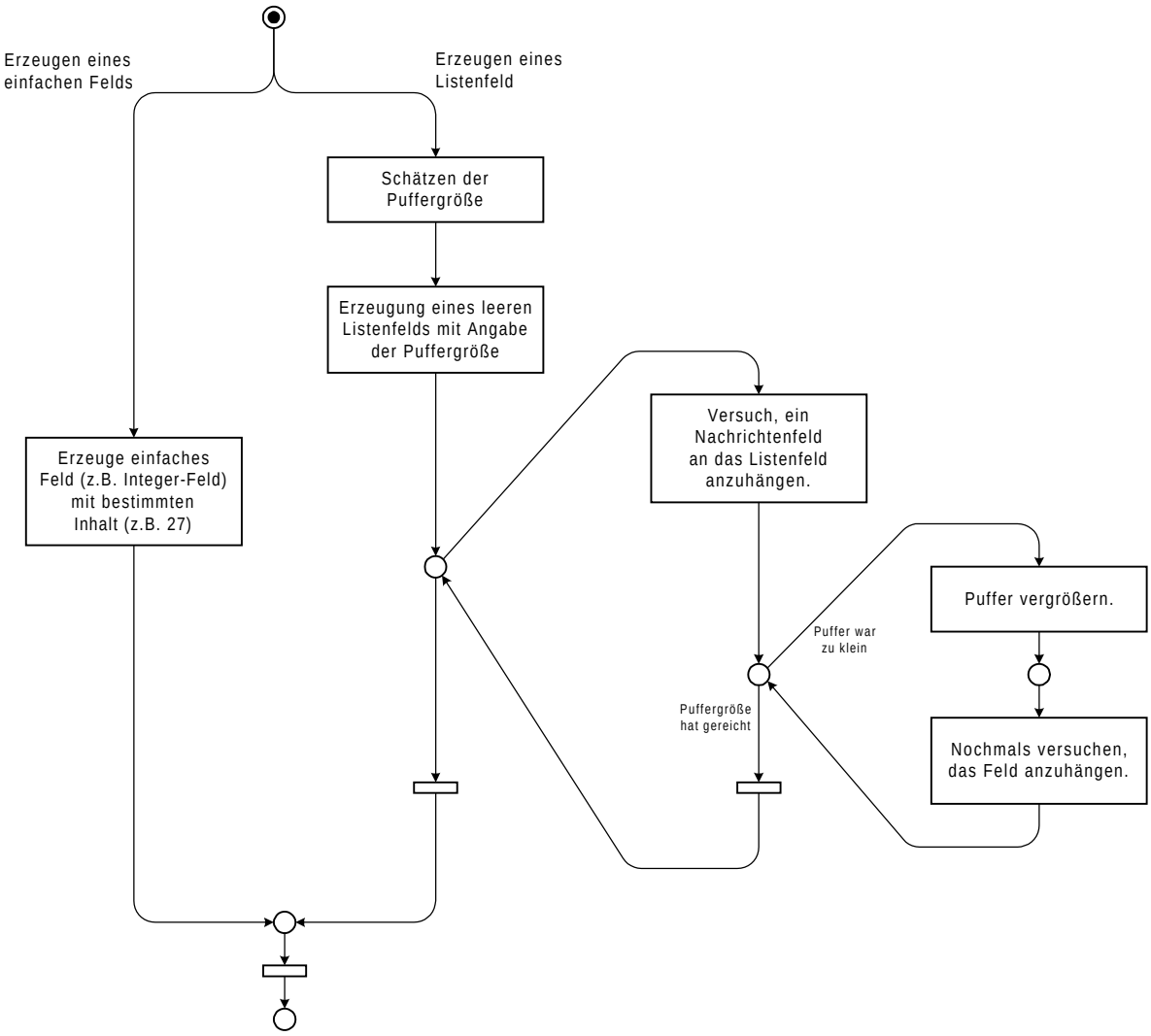


Bild 3.13 Erzeugen eines Nachrichtenfelds

Rechts ist die Erzeugung und das Füllen eines Listenfelds gezeigt. Bei der Erzeugung eines Listenfelds ist die Puffergröße anzugeben. Die Feldliste ist anfangs leer. An die Liste können nacheinander Felder angehängt werden. Voraussetzung dafür ist, daß noch genügend Platz im Puffer ist. Mißlingt das Anhängen eines Felds, dann wird der Puffer vergrößert und das Anhängen erneut versucht. Die Vergrößerung des Puffers ist vergleichsweise zeitaufwendig, da der Inhalt dazu komplett kopiert wird.

3.3.6.2 Nachrichtenempfang

Das Petri-Netz in Bild III[1] zeigt einen für eine nachrichtenempfangende Anwendung typischen Ablauf, der aber nicht in genau dieser Form erfolgen muß.

Etwas komplizierter als der Nachrichtenversand ist der Empfang von Nachrichten. Hier kommt der Callback-Mechanismus und der Ereignisverteiler ins Spiel. Nach der Sitzungseröffnung muß nun auch noch der Eintritt in die Ereignisverteilschleife erfolgen. Die Sitzungseröffnung geschieht durch den Aufruf der Prozedur `make_session` des Busverwalters. Vor dem Eintritt in die Ereignisverteilschleife müssen allerdings Endpunkte für den Nachrichtenempfang und zugehörige Empfängerakteure erzeugt werden. Das Verlassen der Schleife kann nämlich nur durch einen entsprechenden Auftrag eines Nachrichtenempfängers an den Busverwalter veranlaßt werden.

Zunächst werden in einer Schleife Empfängerakteure erzeugt. Daß dies ganz am Anfang geschieht, macht deutlich, daß dafür keine offene Sitzung notwendig ist. Das gilt nicht für die Erzeugung von Endpunkten, was erst in der Schleife unterhalb der Sitzungseröffnung geschieht. Es sind die drei unterschiedlichen Klassen von Endpunkten zu sehen: Abonnements, Briefkästen und Anfragen.

Die gestrichelte Linie grenzt den Zuständigkeitsbereich des Software-Bus-Akteurs gegen den der Anwendung ab. Durch den Aufruf der Prozedur `enter_main_loop` des Busverwalters gibt die Anwendung die Prozessormarke an den Bus ab. Immer wenn eine Nachricht vom Bus kommt, ruft dieser einen Callback-Akteur und gibt damit den Prozessor für die Abwicklung der Callback-Prozedur an die Anwendung zurück. Selbstverständlich können Endpunkte auch innerhalb einer Callback-Ausführung erzeugt werden, dies ist rechts dargestellt.

Der Mehrmarkenplatz macht deutlich, daß der Bus eine Warteschlange für eingehende Nachrichten hat. Nachrichten, die während der Abwicklung eines Callbacks eintreffen, gehen nicht verloren.

Durch den Sitzungsschlußauftrag wird dem Bus signalisiert, daß er die Ereignisverteilschleife verlassen soll. Sobald er das tut, geht die Prozessormarke an die Anwendung zurück. Das Programm der Anwendung wird an der Stelle fortgesetzt, die auf die Anweisung `enter_main_loop` folgt.

[1] Das Bild III befindet sich in der Tasche am Ende dieser Schrift.

4 Java

Java ist eine Sprache fürs Internet. Das hat mehrere Gründe. Kleine Java-Programme, sog. „Applets“, können zusammen mit einer HTML-Seite durch einen WWW-Browser geladen und innerhalb des Browsers ausgeführt werden. Das ist vollkommen unabhängig von der Plattform möglich, auf der der Browser läuft.

Java-Programme werden zunächst in den sog. Java-Bytecode übersetzt. Der Java-Bytecode wird interpretiert. Das ist plattformunabhängig möglich, da jeder java-fähiger Browser einen Java-Bytecode-Interpreter enthält.

Die Plattformunabhängigkeit betrifft nicht nur den Abwickler selbst, sondern auch die Schnittstellen zur Peripherie und dort besonders für die grafische Benutzerschnittstelle. Erreicht wird die Unabhängigkeit durch die Bereitstellung genormter Klassenbibliotheken, die jedem Java-fähigen Browser mitgegeben werden. Java-Applets müssen ohne jedwede manuelle Anpassung gestartet werden können. Die Anforderungen an die Plattform-unabhängigkeit sind daher außergewöhnlich hoch.

Das Verschicken von Programmen übers Internet erfordert eine besondere Berücksichtigung von Sicherheitsinteressen. In Java sind Maßnahmen getroffen worden, um die Gefährdung des Systems, das diese Programme abwickelt, durch Computerviren und andere Angriffe gering zu halten.

Java ist eine objektorientierte Sprache, die sich von der Syntax her an C und C++ anlehnt.

Neben den immer im Rahmen einer Web-Seite und eines Web-Browsers abgewickelten Applets können mit Java auch gewöhnliche Programme (sog. „Applications“) geschrieben werden, die prinzipiell genauso plattformunabhängig sind wie Applets.

Anlaß für die Entwicklung von Java gab die von der Firma Sun verfolgte Idee des „Netzcomputers“. Ein Netzcomputer ist ein einfacher, billiger, für den Massenmarkt gedachter Computer, der Programme nicht auf einer Festplatte installiert hat, sondern sie bei Bedarf von Dienstleistern über das Netz übertragen bekommt. Java ist als Programmiersprache für solche Programme entworfen worden und es ist beabsichtigt, in den Netzcomputern Prozessoren einzusetzen, deren Maschinensprache der Java- Bytecode ist.

4.1 Warum Java?

Neben den bereits genannten Eigenschaften von Java spielt wohl die in der letzten Zeit sehr große Popularität des Internet eine große Rolle für den Erfolg von Java. Der hohe Bekanntheitsgrad, die in Form von kostenlosen Web-Browsern enorme Verbreitung von Java-Interpretern und die Nähe zu der verbreiteten Programmiersprache C++ läßt erwarten, daß Java auch zukünftig eine wichtige Rolle spielen wird.

Der durch die Technik des Ladens und Startens eines Java-Applets von einem Web-Browser aus gegebene Zwang zu absolut unproblematischer Installierbarkeit und Plattform-unabhängigkeit ist wohl einer der herausragenden Vorteile von Java. Entwickler von Java-Programmen brauchen damit keinerlei aufwendige Portierungen auf unterschiedliche Plattformen durchzuführen. Durch die hohe Akzeptanz von Java tritt ein sich selbst verstärkender Standardisierungseffekt ein. Mit der Übertragbarkeit über das Internet tun sich neue Vertriebswege und Vermarktungsmöglichkeiten besonders für kleine Programme auf.

Java verwendet für Text nicht den ASCII-Zeichensatz, sondern Unicode. Unicode codiert ein Zeichen mit 16 Bit und ermöglicht damit die Verwendung einer großen Zahl von Schriften, darunter auch die japanische.

Java lehnt sich in an C++ an, wohl um C++-Programmierern das Erlernen der Sprache zu erleichtern. Die Anlehnung an die Syntax der Sprache C mag für in dieser Sprache erfahrene Programmierer ein Vorteil sein, die Lesbarkeit der Programme wird dadurch aber stark beeinträchtigt.

Im Gegensatz zu C++ gibt es einen automatischen Speicherbereiniger, so daß die mit der Speicherbereinigung verbundenen Probleme von C++ entfallen. Es gibt auch keine Zeigertypen und insbesondere keine Zeigerarithmetik, so daß unerlaubte Speicherzugriffe wesentlich weniger wahrscheinlich sind.

Java ermöglicht das Mehrfacherben, allerdings nur bei vollständig aufgeschobenen Klassen[1]. Programme können vom Java-Interpreter in mehreren nebenläufigen Prozessen abgewickelt werden.

Vergleicht man Mechanismen von Java und Eiffel, so stellt man fest, daß Eiffel letztlich auf einer kleinen Zahl von Konzepten beruht, die allerdings in aller Konsequenz umgesetzt wurden. Die Sprache setzt dem Programmierer in der Benutzung der Konstrukte kaum Grenzen. Java dagegen begrenzt die Benutzbarkeit seiner Konstrukte auf die eher einfachen Standardfälle.

Bei größeren Projekten wird man daher bei Java möglicherweise an Grenzen stoßen.

---

[1] In Java heißen aufgeschobene Klassen „Interface“.

Als Beispiel sei die Exportbeschränkung von Merkmalen genannt: In Eiffel kann explizit für jedes Merkmal eine Liste von Klassen angegeben werden, die das Merkmal benutzen dürfen. Die Exportbeschränkung in Java ist verknüpft mit der Zugehörigkeit von Klassen zu sog. „Packages“. Auch in Eiffel können Klassen in „Cluster“ gruppiert werden. Diese Gruppierung kann aber völlig unabhängig von den Exportbeschränkungen vorgenommen werden.

4.2 Der Vererbungsmechanismus von Java

Die Vererbungsmechanismen von Java und Eiffel unterscheiden sich insbesondere darin, daß in Java der Vererbungsmechanismus durch die Benennung der Merkmale gesteuert wird. In Eiffel wird dagegen die Vererbung nicht durch die Benennung von Merkmalen beeinflußt. Durch den Umbenennungsmechanismus ist in Eiffel ein hohes Maß an Flexibilität gegeben. Die Namenswahl für ein Merkmal kann sich ganz nach der Semantik des Merkmals richten.

4.2.1 Vergleich der Redefinitionsmechanismen von Java und Eiffel

Um in Java einer ererbten Routine eine neue Implementierung zu geben, wird in der Subklasse eine neue Routinenimplementierung unter demselben Namen und mit der gleichen Signatur definiert. Innerhalb der Subklasse kann die Implementierung der direkten Superklasse mithilfe des Schlüsselwort „super“ weiterhin benutzt werden.

Eiffel erlaubt eine freiere Steuerung des Vererbungsmechanismus. Ohne zusätzliche Maßnahmen ist die alte Implementierung $I_{alt}(r)$ einer Routine r weder innerhalb einer Subklasse noch durch Kunden der Subklasse zugänglich – der Routine ist für die Subklasse eindeutig die neue Implementierung $I_{neu}(r)$ zugeordnet. Durch die Möglichkeit des wiederholten Erbens kann in der Subklasse jedoch eine neue Routine s definiert werden, der die alte Implementierung $I_{alt}(r)$ zugeordnet werden kann.

Während in Java durch „super“ alte Implementierungen der *direkten* Superklasse einer Klasse automatisch zugänglich sind, können in Eiffel bei Bedarf alte Implementierungen aus beliebigen Superklassen zugänglich gemacht werden.

Die Notwendigkeit, die Signatur einer Routine bei der Redefinition unverändert zu lassen, verhindert es in Java, den Wertebereich von Parametern einer redefinierten Routine in Subklassen gegenüber der alten Implementierung zu verändern.

Attribute können in Java nicht redefiniert werden. Wird in einer Klasse ein Attribut unter dem gleichen Namen wie in der Superklasse vereinbart, so ist das bisherige Attribut weiterhin vorhanden, aber „überdeckt“, d.h. für Kunden der Klasse nicht zugreifbar. Innerhalb der Klasse kann es mithilfe des Schlüsselworts „super“ benutzt werden.

Die Wahl eines bereits in der Superklasse benutzten Merkmalsnamens hat damit unterschiedliche Wirkungen für Attribute und Routinen.

4.2.2 Das „Überladen“ von Routinennamen

Java ermöglicht das sog. „Überladen“. Dabei tragen Routinen einer Klasse den gleichen Namen, werden aber anhand der Zahl und der Typen ihrer Parameter unterschieden. Bei parameterlosen Routinen ist das nicht möglich.

Mir erscheint dieser Mechanismus als sehr fragwürdig. Wenn Routinen eine unterschiedliche Anzahl von Parametern haben, dann stellt sich die Frage, warum man sie nicht auch unterschiedlich benennen sollte.

Als nächstes betrachte man den Fall, daß zwei gleichbenannte Routinen die gleiche Zahl an Parametern haben. Paßt zu einem Routinenaufruf aufgrund der Parametertypen nur eine einzige Routine, so gilt das oben gesagte: Warum nicht zwei unterschiedliche Namen wählen? Das Lesen eines Programms wird durch das Überladen von Merkmalsnamen sehr erschwert.

Es gibt aufgrund der Typmehrdeutigkeit auch den Fall, daß grundsätzlich mehrere Routinen zu einem Aufruf passen.

> Ein Beispiel:
>
> Die Variable `G` sei vom Typ „`TASSE`“, die Variable `K` vom Typ „`KAFFEETASSE`“. Es gebe die Prozeduren „`TRINKE_AUS(TASSE)`“ und „`TRINKE_AUS(KAFFETASSE)`“.
>
> Der Aufruf `TRINKE_AUS(K)` paßt zu beiden Routinen. In Java wird ihm die „am meisten spezifische“ Prozedur `TRINKE_AUS(KAFFETASSE)` zugeordnet.
>
> Bei Routinen mit mehr als einem Parameter gibt es einen relativ komplizierten Algorithmus, um zu bestimmen, welche Routine mit einem Aufruf gemeint ist. Es ist dann für den Leser eines Programms unmöglich, auf Anhieb zu erkennen, welche Routine zu einem Aufruf gehört.[1]

[1] Vgl. [Arnold, S.108 ff]

4.2.3 Mehrfacherben

In Java können Klassen nur von einer Superklasse erben. Neben Klassen gibt es in Java aber auch sog. „Interfaces“, das sind Klassenbeschreibungen, die Spezifikationen, aber noch keine Implementierungen von Merkmalen enthalten. Sie entsprechen Eiffel-Klassen, die ausschließlich aufgeschobene Merkmale haben.

Zwischen Interfaces ist das Mehrfacherben möglich, auch können Java-Klassen von mehreren Interfaces erben.

Besonders beim Mehrfacherben stellt sich die Frage nach der Auflösung von Namenskonflikten. Wenn Klassenbibliotheken von unterschiedlichen Herstellern stammen, dann kann es leicht vorkommen, daß Merkmale gleich benannt worden sind, aber eine unterschiedliche Semantik haben.

In Java ist es nicht möglich, Namenskonflikte beim Mehrfacherben aufzulösen. In der Regel wird es daher nicht möglich sein, die Technik des Mehrfacherbens im Zusammenhang mit vorgefertigten Klassenbibliotheken einzusetzen.

5 Ausblick

Die vorliegenden Arbeit entstand anhand von Arbeiten zur Realisierung des in der Einleitung beschriebenen Beispielsystems. Zur vollständigen Realisierung des Beispielsystems wäre es noch ein weiter Weg. Es bietet sich daher an, dieses Beispiel als Grundlage für weitergehende Untersuchungen zu benutzen.

Es hat sich gezeigt, daß dieses Vorgehen dazu geeignet war, ein gutes Verständnis der Sprache Eiffel und des Rendezvous-Software-Busses zu erlangen. Durch den Zwang, sich bei der Realisierung des Systems mit all den im Detail steckenden Widrigkeiten auseinanderzusetzen, wurden zugleich die Schwächen und Schwierigkeiten der Produkte deutlich.

Für eine Fortsetzung der Arbeit am Beispielsystem bieten sich einige Ansatzpunkte besonders an:

- Nach wie vor ist das Problem der Reaktion auf Ereignisse aus mehreren Quellen durch eine busbenutzende Anwendung nicht gelöst. Das Problem muß gelöst werden, wenn eine über ein GUI erfolgende Kommunikation einer Busanwendung mit dem Benutzer erforderlich ist. Hier bietet es sich an, einen eigenen Ereignisverteiler zu bauen, der an die dafür vorgesehene Schnittstelle des Busses anzuschließen wäre.[1]

  Mit dem System *EiffelBuild* ist es möglich, Eiffel-Programme mit grafischer Benutzerschnittstelle zu entwickeln. Der Mechanismus der Ereigniskommunikation solcher Programme müßte untersucht werden, um den genannten eigenen Ereignisverteiler entwerfen zu können.

- In [Otto] wird ein Klassengerüst für Multiclient-Multiserver-Anwendungen vorgestellt, das als Basis für die weitere Implementierung des Beispielsystems dienen kann. Es bietet sich an, ein dazu kompatibles Klassengerüst aufbauend auf der vorliegenden Buskapselung auch in Eiffel bereitzustellen.

---

[1] Vgl. Abschnitt 3.3.1.

- Ähnlich der Eiffel-Buskapselung könnte eine Kapselung des Busses in Java erfolgen. Java zeichnet sich besonders durch die Verwendungsmöglichkeiten im Internet aus. So könnte es ermöglicht werden, ein in Java programmiertes Terminal für die Videothekenverwaltung per WWW auf beliebige Kundenrechner laden zu können.

- Nicht zuletzt ist der Anschluß an Datenbanken noch nicht implementiert worden, so daß sich hier ein weites Betätigungsfeld ergibt. Die Bibliothek *EiffelStore* ermöglicht die Anbindung von relationalen Datenbanken an Eiffel. Ähnliche Produkte sind mittlerweile auch für Java erhältlich.

6 Literaturverzeichnis

[Arnold] Ken Arnold, James Gosling
The Java Programming Language
Addison-Wesley, 1996
ISBN 0-201-63455-4

[Auer] Stefan Auer
Teknekron[1] Technology Evaluation
Interner Bericht, SAP-AG, Walldorf 1996

[Bartlett] Neil Bartlett, Alex Leslie, Steve Simkin
Java Programming Explorer
The Coriolis Group, 1996
ISBN 1-883577-81-0

[Darnell] Peter A. Darnell, Philip E. Margolis
C – A Software Engineering Approach
Springer, 1990
ISBN 3-540-97389-3

[Environment] Bertrand Meyer
ISE Eiffel: The Environment
Interactive Software Engineering Inc., 1995
ISE Technical Report TR-EI-39/IE

[E&C] **Using external functions from Eiffel**
(Informationsseite im WWW)
http://www.eiffel.com/ – dort unter FAQ

[1] Teknekron heißt heute TIBCO.

[ETL] Bertrand Meyer
Eiffel – The Language
Prentice Hall, 1992
ISBN 0-13-247925-7

[Gröne] Bernhard Gröne
Bereitstellung einer Laborumgebung und Untersuchung objektorientierter Datenbanktechnologien
Diplomarbeit am Lehrstuhl für Digitale Systeme, Universität Kaiserslautern, 1996

[JavaSpec] James Gosling, Bill Joy, Guy Steele
The Java Language Specification
Version 1.0
Sun Microsystems, Inc., 1996

[K&R] Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language
Prentice Hall, 1978
ISBN 0-13-110163-3

[EiffelLib] Bertrand Meyer
Eiffel – The Libraries
Interactive Software Engineering Inc., 1994
ISE Technical Report TR-EI-44/LI

[Meyer] Bertrand Meyer
Objektorientierte Softwareentwicklung
Hanser / Prentice Hall, 1990
ISBN 3-446-15773-5

[Otto] Johannes Otto
Bereitstellung einer Experimentierumgebung zur Entwicklung von Client-Server-Systemen in Smalltalk
Diplomarbeit am Lehrstuhl für Digitale Systeme, Universität Kaiserslautern, 1996

[Rockwell] Robert Rockwell, Michael H. Gera
The Eureka Software Factory CoRe: A Conceptual Reference Model for Software Factories
in: IEEE Proceedings on Software Engineering Environments, Reading 1993

[RVProg] **Rendezvous Software Bus Programmers's Guide**
Version 1.1, June 1995
Teknekron[1] Software Systems, Inc., 1995

[Visual C++] **Microsoft Visual C++ User's Guide**
Version 2.0 Volume 1
Microsoft Corporation, 1994
ISBN 1-55615-800-9

[Wendt] Siegfried Wendt
Nichtphysikalische Grundlagen der Informationstechnologie
Springer, 1991
ISBN 3-540-51555-0

[Wiegert] Oliver Wiegert
Änderbarkeit durch Objektorientierung
Vieweg, 1995
ISBN 3-528-05518-9

[1] Teknekron heißt heute TIBCO.

7 Anhang

7.1 Adressen

Interactive Software Engineering Inc. (ISE)
270 Storke Road, Suite #7
Goleta, California 93117, U.S.A.
Tel. +1 805-685-1006
Fax: +1 805-685-6869
http://www.eiffel.com/

SIG Computer GmbH
zu den Bettern 4
D-35619 Braunfels
Tel. +49 6472 2096
Fax +49 6472 911 031
http://www.sigco.com/

Tower Technology Corporation
1501 West Koenig Lane
Austin, TX 78756, U.S.A.
Tel.: +1 512 452-9455
Fax +1 512 452-1721
http://www.twr.com/

NICE (Nonprofit International Consortium for Eiffel)
45 Hazelwood
Shankill
Co Dublin
Republic of Ireland
Tel.: +353 1 282 3487
email: nice@atlanta.twr.com

Sun Microsystems, Inc.
2550 Garcia Avenue, Mountain View
California 94043-1100 U.S.A
http://www.sun.com/

TIBCO Inc. (The Information Bus Company)
3165 Porter Drive
Palo Alto, CA 94304 U.S.A.
Tel.: +1 415 846-5000
Fax: +1 415 846-5005
http://www.tss.com/

Universität Kaiserslautern
Fachbereich Elektrotechnik
Lehrstuhl für Digitale Systeme und Datenverarbeitung
Postfach 3049, D-67653 Kaiserslautern
Tel.: +49 631 205-3066
Fax: +49 631 205-3606
http://alwendt1.e-technik.uni-kl.de/

7.2 Eiffel-Quelltexte

7.2.1 Der Busverwalter

```
class RV_MANAGER

-- Dies ist die Klasse des Busverwalters. Der Busverwalter ist ein Monopol-
-- akteur. Er wird durch das Merkmal rv der Klasse RV_USER zugänglich gemacht.
-- Seine Aufgabe ist die Sitzungssteuerung sowie das Erzeugen der Nachrichten
-- repräsentierenden Eiffel-Objekte, wenn diese Nachrichten vom Bus empfangen
-- werden.


inherit RV_OBJECT    -- Insbesondere die C-Routinen des RV-API werden in RV_OBJECT
                     -- bereitgestellt.


feature {RV_USER}


make_session is
     -- Das Erzeugen einer Sitzung muß explizit veranlaßt werden.
     -- Für die meisten den Bus betreffenden Aktionen ist die Existenz einer
     -- Bussitzung Voraussetzung.
     require
          session_not_initialized: not session_exists
     local
          text: STRING;
          err:  INTEGER
     do
          err:= eifrv_init($session, null_ref, null_ref, null_ref);
     ensure
          session_initialized: session_exists and then session/=null;
     end;



main_loop is
     -- Ein Aufruf dieser Prozedur startet die Ereignis-Verteil-Schleife
     -- des RV-Busses. Eine Rückkehr aus dieser Schleife erfolgt erst nach
     -- Beendigung der RV-Sitzung durch Aufruf von "terminate" innerhalb
     -- der Schleife.
     require
          session_exists
     do
          eifrv_main_loop(session)
     ensure
          not session_exists
     end;
```

```
terminate is
     -- Beendigung der Ereignis-Verteil-Schleife des RV-Busses. Die
     -- Kontrolle kehrt nach Beendigung der laufenden Callback-Routine
     -- an den Aufrufer von "main_loop" zurück. Es bleibt zu klären, ob
     -- es zwischen dem Aufruf von rv_Term (s.u.) und dem Ende der
     -- Callback-Abwicklung noch möglich ist, die Session-ID zu benutzen.
     require
          session_exists
     local err: INTEGER
     do
          err:= eifrv_term(session);
          session:= null
     ensure
          not session_exists
     end;



session_exists: BOOLEAN is
     -- Ist eine Bussitzung geöffnet?
     do
          Result:= session/=null
     end;


session: POINTER
     -- Dieses Attribut enthält den Identifikator der Bussitzung. Durch ihn
     -- identifiziert sich die Anwendung gegenüber dem Bus.




feature {RV_CALLBACK, RV_MSG_LIST, RV_MSG_CALLBACK}


field_from_bus( f_type: INTEGER;
                f_size: INTEGER;
                f_adr:  POINTER): RV_MSG_FIELD is
     -- Diese Funktion erzeugt ein Objekt vom Typ RV_MSG_FIELD aus
     -- den vom Bus übergebenen Parametern. Für die Freigabe des
     -- Speichers sorgt der Bus.
     do
          inspect f_type


          -- Listenfeld
          when  1 then
          !RV_MSG_LIST!   Result.make_from_bus(f_type,f_size,f_adr)


          -- Feldinhalt ist String
          when  8 then
          !RV_MSG_STRING! Result.make_from_bus(f_type,f_size,f_adr)


          -- Feldinhalt ist Boolean
          when  9 then
          !RV_MSG_BOOLEAN!Result.make_from_bus(f_type,f_size,f_adr)


          -- Feldinhalt ist Integer
          when 11,12 then
          !RV_MSG_INTEGER!Result.make_from_bus(f_type,f_size,f_adr)
```

```
            -- Feldinhalt ist Double
            when 13 then
            !RV_MSG_DOUBLE! Result.make_from_bus(f_type,f_size,f_adr)


            -- Das Feld enthält einen Datentyp, der (noch) nicht implementiert
            -- wurde.
            else
            !RV_MSG_UNKNOWN!Result.make_from_bus(f_type,f_size,f_adr)
            end

      ensure
            no_memory_allocated: Result.mem_alloc = false
            -- zu der Bedeutung von mem_alloc: siehe Klasse RV_MSG_FIELD
      end


end -- class RV_MANAGER
```

7.2.2 Nachrichtenfelder

7.2.2.1 Die Superklasse der Nachrichtenfeld-Klassen

```
class RV_MSG_FIELD
-- deferred


inherit
     RV_USER
          redefine copy, is_equal end;
     RV_OBJECT
          redefine copy, is_equal end;
     MEMORY
          -- siehe Prozedur 'dispose'
          redefine dispose, copy, is_equal end

-----------------------------------------------------------------------------
-- Exemplarerzeugung                                                       --
-----------------------------------------------------------------------------

feature {NONE} -- only for creation

-- Wird eine Nachricht vom Bus empfangen, so erzeugt der Busverwalter ein Exem-
-- plar der dem Typ entsprechenden Unterklasse von RV_MSG_FIELD. Dafür benutzt
-- er die Prozedur make_from_bus.
-- Soll ein Feld in der Anwendung erzeugt werden, so ist dafür die Prozedur
-- make zu verwenden, die bei Bedarf in den Unterklassen bereitzustellen ist.

make_from_bus(b_type: INTEGER;
                       b_size: INTEGER;
                       b_adr:  POINTER) is
     do
          type:= b_type;
          size:= b_size;
          adr := b_adr;
     ensure
          not mem_alloc
     end
```

```
-------------------------------------------------------------------------------
-- Kopieren und Speicherverwaltung                                           --
-------------------------------------------------------------------------------

feature

copy(other: like current) is
     -- Beim Kopieren eines Felds muß auch die C-Repräsentation kopiert
     -- werden. Zur Semantik von copy siehe Eiffel-Basis-Bibliothek.
     local
          old_adr: POINTER
     do
          name:=clone(other.name);
          type:=other.type;
          size:=other.size;
          old_adr:= adr;
          adr:=eifrv_malloc(size);  -- Neuen Speicher reservieren.
          eifrv_copy(adr, other.adr, size);
          if mem_alloc then
               eifrv_free(adr) end; -- Der alte Inhalt wird verworfen.
          mem_alloc:=true;
     ensure then
          mem_alloc;
          size = other.size
     end;


own_buffer is
     -- Wenn die Nachricht vom Bus kommt, ist sie nur eine beschränkte Zeit
     -- nutzbar. Diese Prozedur kopiert die C-Repräsentation in eigenen Speicher.
     require
          not mem_alloc
     local
          old_adr: POINTER
     do
          old_adr:= adr;
          adr:=eifrv_malloc(size);
          eifrv_copy(adr, old_adr, size)
     ensure
          mem_alloc;
          equal(old current, current)
     end;


is_equal(other: like current):BOOLEAN is
     -- Wann sind zwei Felder gleich?
     do
          Result:=  (other.size = size)
                    -- Vergleich der C-Repräsentation durch eifrv_cmp
                    and then eifrv_cmp(other.adr, adr, size)
                    and then equal(other.name, name)
     end


-------------------------------------------------------------------------------
-- Attribute                                                                 --
-------------------------------------------------------------------------------

feature


name: STRING;
     -- Name des Felds. Dieser Name wird benötigt zum Zusammenbau von Nachrichten.
     -- Ein Feld einer Nachricht wird in der Regel durch den Namen identifiziert.
     -- Beim Senden wird der Name als Thema benutzt.
```

```
mem_alloc:   BOOLEAN;
     -- Wenn C-Speicherplatz für das vorliegende Objekt durch die Eiffel-Seite
     -- reserviert wurde, dann muß dieser wieder freigegeben werden.
     -- Dies geschieht durch die von der Eiffel-Speicherbereinigung
     -- aufgerufene Prozedur dispose.
     -- Wenn mem_alloc=true ist, muß der Speicher freigegeben werden.
     -- mem_alloc ist nur dann false, wenn das Feld vom Bus übergeben wurde. In
     -- diesem Fall ist jedoch große Vorsicht angebracht, da der Speicherplatz
     -- vom Bus nur für eine begrenzte Zeit bereitgestellt wird.



-------------------------------------------------------------------------------
-- Zugriffsroutine                                                           --
-------------------------------------------------------------------------------

set_name(s:STRING) is
     require
          s/=Void;
          small_enough: s.count<=100
     do
          name:=clone(s)
     ensure
          small_enough: name.count<=100
     end;



-------------------------------------------------------------------------------
-- Veröffentlichung des Feldinhalts                                          --
-------------------------------------------------------------------------------

feature {RV_USER}

send is
     -- Diese Prozedur veröffentlicht den Inhalt des vorliegenden
     -- Nachrichtenfelds.
     -- Als Thema (subject) dient der Name (name) des Nachrichtenfelds.
     require
          rv.session_exists;
          sensfull_subject: name/=Void and then name.count>0
     local
          err: INTEGER
     do
          err:= eifrv_send(    rv.session,
                               name.to_c,
                               type,
                               size,
                               adr)
     end




-------------------------------------------------------------------------------
-- Datentyp-Funktionen                                                       --
-------------------------------------------------------------------------------

feature

-- Der Typ eines Felds wird durch das Attribut type in Form einer Zahl
-- gespeichert. Hier erfolgt eine Zuordnung zu den Typen im Klartext.

is_field_list: BOOLEAN is
     do Result:= (type=1) end;

is_integer: BOOLEAN is
     do Result:= (type=11 or type=12) end;
```

```
is_boolean: BOOLEAN is
     do Result:= (type=9) end;

is_double: BOOLEAN is
     do Result:= (type=13) end;

is_string: BOOLEAN is
     do Result:= (type=8) end;

is_bad: BOOLEAN is
     do Result:= (type=0) end;

type_not_implemented: BOOLEAN is
     do
          Result:= not  ( is_field_list or
                          is_integer    or
                          is_boolean    or
                          is_double     or
                          is_string     or
                          is_bad)
     end -- type_not_implemented




-------------------------------------------------------------------------------
-- Interne Merkmale                                                          --
-------------------------------------------------------------------------------

feature {NONE}


dispose is
     -- Diese Prozedur ist ererbt von der Klasse 'memory'. Sie wird von der
     -- Speicherbereinigung aufgerufen, wenn das vorliegende Objekt entfernt
     -- wird. Bei dieser Gelegenheit wird auch der ggf. durch malloc reservierte
     -- C-Speicherplatz wieder freigegeben.
     do
          if mem_alloc then
               eifrv_free(adr);
               mem_alloc:= false end
     end




-------------------------------------------------------------------------------
-- Typ-, Größe-, Adress- Tripel zur                                          --
-- Identifikation des Feldinhalts in C-Repräsentation                        --
-------------------------------------------------------------------------------

feature

type: INTEGER;
size: INTEGER;
adr:  POINTER


end -- class RV_MSG_FIELD
```

7.2.2.2 Ganzzahlen-Felder

```
class RV_MSG_INTEGER

inherit RV_MSG_FIELD

creation
     make, copy

creation {RV_MANAGER}
     make_from_bus


feature {NONE} -- only for creation

make(i: INTEGER) is
     do
          size:= 4;
          -- Erzeugen der C-Repräsentation einer Ganzzahl
          adr:= eifrv_mk_int(i);
          type:= 11;
          mem_alloc:= true;
     ensure
          mem_alloc
     end


feature -- access


invariant
     is_integer

end -- class RV_MSG_INTEGER
```

7.2.2.3 Gleitkommazahlen-Felder

```
class RV_MSG_DOUBLE

inherit RV_MSG_FIELD

creation
     make, copy

creation {RV_MANAGER}
     make_from_bus



feature {NONE} -- only for creation


make(d: DOUBLE) is
     do
          size:= 8;
          adr:= eifrv_mk_double(d);
          mem_alloc:= true;
          type:= 13;
     ensure
          mem_alloc
     end

invariant
     is_double

end -- class RV_MSG_DOUBLE
```

7.2.2.4 Boole’sche Felder

```
class RV_MSG_BOOLEAN

inherit RV_MSG_FIELD

creation
     make

creation {RV_MANAGER}
     make_from_bus

feature {NONE} -- only for creation

make(b: BOOLEAN) is
     do
          size:= 4;
          -- Erzeugen der C-Repräsentation eines boole’schen Felds.
          -- (Die Repräsentation gleicht der von Ganzzahlen.)
          if b then
               adr:= eifrv_mk_int(1)
          else
               adr:= eifrv_mk_int(0)
          end;
          mem_alloc:= true;
          type:= 9
     ensure
          mem_alloc
     end


invariant
     is_boolean

end -- class RV_MSG_BOOLEAN
```

7.2.2.5 Zeichenketten-Felder

```
class RV_MSG_STRING

inherit RV_MSG_FIELD

creation make, copy

creation {RV_MANAGER}
     make_from_bus

feature {NONE} -- only for creation


make(s: STRING) is
     require
          s/=Void
     do
          size:= s.count+1;    -- Bei der C-Repräsentation eines Strings wird das
          -- Ende durch ein Nullzeichen markiert, der C-String ist daher um ein
          -- Byte größer.
          adr:= eifrv_mk_string(s.to_c, size);
          type:= 8;
          mem_alloc:= true
     ensure
          mem_alloc
     end


feature {NONE} -- implementation


invariant
     is_string

end -- class RV_MSG_STRING
```

7.2.2.6 Listenfelder

```
class RV_MSG_LIST

-- Dies ist die komplizierteste Unterklasse von RV_MSG_FIELD. Durch Listenfelder
-- ist ein verschachtelter Aufbau von Nachrichten möglich. Listenfelder haben
-- einen Puffer, der in der Regel größer ist als der Inhalt, um das Anhängen von
-- Feldern zu ermöglichen.

inherit
     RV_MSG_FIELD
          redefine
               copy, own_buffer
          end


creation
     make, copy

creation {RV_MANAGER}
     make_from_bus



feature {NONE} -- only for creation

make(bfr_size: INTEGER) is
     -- Erzeugen einer leeren Liste. Dabei ist die Größe des bereitzustellenden
     -- Puffers für den Inhalt anzugeben.
     require
          bfr_size >= 8
     do
          buffer_size:= bfr_size;
          last_rv_error:=
               eifrv_msg_init(rv.session, $adr, buffer_size);
          type:=1;
          get_size;
          mem_alloc:= true
     ensure
          no_rv_error;
          is_field_list;
          mem_alloc;
     end



feature  -- Duplication


copy(other: like current) is
     do
          name:=clone(other.name);
          type:=other.type;

          if mem_alloc and buffer_size < other.size then
               -- Der bisher reservierte Speicher ist nicht ausreichend.
               -- Er wird freigegeben.
               eifrv_free(adr);
               mem_alloc:= false
          end;
```

```
          if not mem_alloc then
               -- Reservierung neuen Speichers
               adr:= eifrv_malloc(other.size);
               mem_alloc:= true;
               buffer_size:= other.size
          end;

          check
               mem_alloc;
               buffer_size >= other.size;
               adr/=other.adr
          end;
          eifrv_copy(adr, other.adr, other.size);
          size:= other.size;
     end;

own_buffer is
     -- require not mem_alloc, siehe RV_MSG_FIELD
     -- Diese Prozedur wird hier redefiniert, da zusätzlich das Attribut
     -- buffer_size gesetzt werden muß.
     local
          old_adr: POINTER
     do
          old_adr:=adr;
          adr:= eifrv_malloc(size);
          eifrv_copy(adr, old_adr, size);
          mem_alloc:=true;
          buffer_size:=size
     end;


new_buffer_size(bfr_size: INTEGER) is
     -- Diese Prozedur kopiert den Inhalt des Felds in einen neuen Puffer. Die
     -- Größe des neuen Puffers wird mit bfr_size angegeben.
     require
          bfr_size >= size
     local
          old_adr: POINTER
     do
          old_adr:=adr;
          adr:= eifrv_malloc(bfr_size);
          eifrv_copy(adr, old_adr, size);
          if mem_alloc then eifrv_free(old_adr) end;
          mem_alloc:=true;
          buffer_size:= bfr_size
     ensure
          mem_alloc;
          buffer_size = bfr_size
     end;
```

```
-------------------------------------------------------------------------------
feature -- LESENDER ZUGRIFF
-------------------------------------------------------------------------------


get_field_by_name(field_name: STRING): RV_MSG_FIELD is
     -- Liefert das erste Feld der Liste mit dem angegebenen Namen. Wird in der
     -- Liste kein Feld gefunden, wird die Suche rekursiv in ggf. enthaltenen
     -- Listenfeldern fortgesetzt. Es muß ein Feld mit dem angegebenen Namen
     -- geben.
     require
          sensefull_name: field_name/=Void and then field_name.count>=1
     local
          fld_type: INTEGER;
          fld_size: INTEGER;
          fld_adr: POINTER
     do
          last_rv_error:= eifrv_msg_get(
                rv.session,
                adr,
                field_name.to_c,
                $fld_type,     -- Eifrv_msg_get schreibt das Ergebnis direkt in
                $fld_size,     -- diese Variablen.
                $fld_adr);
          -- Das das Feld repräsentierende Eiffel-Objekt wird vom Busverwalter
          -- erzeugt:
          Result:= rv.field_from_bus(fld_type,fld_size,fld_adr)
     ensure
          no_rv_error;
          no_memory_allocated: Result.mem_alloc = false
     end;


field_existence(field_name: STRING): BOOLEAN is
     -- Überprüft, ob die Feldliste ein Feld mit dem angegebenen Namen enthält
     -- (true, wenn ja). Die Angabe bezieht sich ggf auch rekursiv auf. in der
     -- Liste enthaltene Listenfelder.
     require
          sensefull_name: field_name/=Void and then field_name.count>=1
     local
          fld_type:  INTEGER;
          fld_size:  INTEGER;
          fld_adr:   POINTER
     do
          last_rv_error:= eifrv_msg_get(
                rv.session,
                adr,
                field_name.to_c,
                $fld_type,     -- Eifrv_msg_get schreibt das Ergebnis direkt in
                $fld_size,     -- diese Variablen.
                $fld_adr);
          Result:= not no_field_found
     ensure
          no_rv_error or no_field_found;
     end;


no_field_found: BOOLEAN is
     -- Es wurde kein Feld mit dem angegebenen Namen gefunden.
     do
          Result:= (last_rv_error=11)
     end;
```

```
apply_callback(field_name: STRING; callback: RV_MSG_CALLBACK) is
     -- Diese Prozedur bewirkt den Aufruf der Routine callback_special eines
     -- Feldbearbeiters für alle Felder der Liste mit dem angegebenen Namen. Wird
     -- ein leerer String übergeben, erfolgt der Aufruf der Reihe nach für alle
     -- Felder unabhängig von ihrem Namen. In callback ist ein Verweis auf den
     -- Feldbearbeiter zu übergeben.
     require
          field_name/=Void
     do
          last_rv_error:= eifrv_msg_apply(
               rv.session,
               adr,
               field_name.to_c,
               callback)
     ensure
          no_rv_error or no_field_found
     end;


-------------------------------------------------------------------------------
feature -- SCHREIBENDER ZUGRIFF
-------------------------------------------------------------------------------


append(field: RV_MSG_FIELD) is
     require
          field_exists:   field/=Void;
          field_has_name: field.name/=Void
     local
          s:   STRING
     do
          last_rv_error:=
               eifrv_msg_append(
                  rv.session,
                  adr,
                  buffer_size,
                  field.name.to_c,
                  field.type,
                  field.size,
                  field.adr);
          get_size
     ensure
          no_rv_error
     rescue
          if out_of_space then
               new_buffer_size(buffer_size + field.size + 100);
               retry
          end
     end;


out_of_space: BOOLEAN is
     -- Der Puffer war zu klein, um ein Feld mit append anzuhängen.
     do
          Result:= (last_rv_error=8)
     end;
```

```
get_size is
     -- Prozedur zur Aktualisierung des Attributs size nach dem Anhängen eines
     -- Felds an die Liste.
     do
          last_rv_error:= eifrv_msg_length(
               rv.session,
               adr,
               $size)
     ensure
          no_rv_error: last_rv_error=0
     end;


buffer_size:    INTEGER


invariant
     not (mem_alloc and buffer_size<8)

end -- class RV_MSG_LIST
```

7.2.3 Endpunkte

7.2.3.1 Die Superklasse der Endpunkt-Klassen

```
class RV_ENDPOINT

-- deferred

-- Endpunkte sind "Orte", an denen Nachrichten zu einem
-- assoziierten Thema empfangen werden.


inherit RV_OBJECT; RV_USER


feature {RV_USER, RV_MANAGER}


close is
     -- Kündigung des Endpunkts
     do
          eifrv_close(rv.session, listen_id, callback)
     end


feature {NONE} -- interne Attribute


listen_id: POINTER;
     -- ein Endpunkt ist durch die Listen-ID
     -- eindeutig gekennzeichnet.


callback: RV_CALLBACK
     -- Jedesmal, wenn eine für diesen Endpunkt abonnierte Nachricht
     -- empfangen wird, wird die Prozedur "callback_general"
     -- des hier enthaltenen Empfängers ausgeführt.



end -- class RV_ENDPOINT
```

7.2.3.2 Abonnements

```
class RV_SUBSCRIPTION
-- Abonnement aller unter einem Subject veröffentlichten Nachrichten.


inherit RV_ENDPOINT


creation {RV_USER}
     make



feature {NONE} -- only for creation


make (subject: STRING; callback_obj: RV_CALLBACK ) is
     require
          session_exists:    rv.session_exists;
          subject_not_empty: subject/=Void and then subject.count>=1;
          callback_exists:   callback_obj/=Void
     local
          err: INTEGER
     do
          callback:= callback_obj;
          err:= eifrv_listen_subject(
               rv.session,
               $listen_id,
               subject.to_c,
               callback)
     end


end -- class RV_SUBSCRIPTION
```

7.2.3.3 Briefkästen

```
class RV_INBOX

inherit RV_ENDPOINT

creation make


feature {RV_USER, RV_MANAGER}

make (callb: RV_CALLBACK ) is
     require
          session_exists:    rv.session_exists;
          callback_exists:   callb/=Void
     local
          err: INTEGER;
          inbox_name_buffer: POINTER
     do
          callback:= callb;
          -- Der Speicherbereich für den vom Bus gelieferten inbox_name muß
          -- selbst bereitgestellt werden.
          inbox_name_buffer:= eifrv_malloc(RV_MAX_INBOX_NAME);
          err:= eifrv_listen_inbox(
               rv.session,
               $listen_id,
               inbox_name_buffer,
               callback);
          inbox_name:="";
          inbox_name.from_c(inbox_name_buffer);
          eifrv_free(inbox_name_buffer)
     end;



inbox_name: STRING


end -- class RV_INBOX
```

7.2.3.4 Anfragen

```
class RV_REQUEST

inherit RV_ENDPOINT



-- Durch das Erzeugen eines Exemplars dieser Klasse wird
-- gleichzeitig die mitgelieferte Nachricht veröffentlicht.
-- Als Thema wird das Attribut "name" des
-- Nachrichtenfeld-Objekts benutzt.
-- Dem Empfänger wird der vom Bus generierte
-- "reply_name" übermittelt, unter dem Antworten an diesen
-- Endpunkt gesendet werden können.



creation {RV_USER}
     make



feature {NONE} -- only for creation

make (     message     : RV_MSG_FIELD;
           callback_obj: RV_CALLBACK ) is
     require
           rv.session_exists;
           sensefull_subject: message.name/=Void and then message.name.count>=1;
           callback_exists:   callback_obj/=Void
     local
           err: INTEGER
     do
           callback:= callback_obj;
           err:= eifrv_rpc(
                rv.session,
                $listen_id,
                message.name.to_c,
                message.type,
                message.size,
                message.adr,
                callback_obj)
     end -- make


end -- class RV_REQUEST
```

7.2.4 Empfänger

```
deferred class RV_CALLBACK


-- In allen Unterklassen muß make bei der Exemplarerzeugung aufgerufen werden,
-- damit dem Bus die Adresse von callback_general mitgeteilt wird.
-- Eigene Initialisierungsoperationen können in der Prozedur
-- make_special implementiert werden.

inherit RV_OBJECT; RV_USER


feature {NONE} -- only for creation

make is
     do
          init_c;
          make_special
     end;

make_special is
     deferred
     end



feature {RV_MANAGER} -- only for C-callbacks


callback_general (   sess:       POINTER;
                     name:       POINTER;
                     reply_name: POINTER;
                     msg_type:   INTEGER;
                     msg_size:   INTEGER;
                     msg_adr:    POINTER) is
     require
          no_bad_data:    msg_type/=0;
          subject_exists: name/= null
     do
          session:= sess;
          subject:="";
          subject.from_c(name);
          reply_box:="";
          if reply_name /= null then
               reply_box.from_c(reply_name)
          end;
          -- Das das Feld repräsentierende Eiffel-Objekt wird vom Busverwalter
          -- erzeugt:
          msg:= rv.field_from_bus(msg_type,msg_size,msg_adr);
          callback_special;

          msg:=Void
          -- Die C-Repräsentation des Felds verliert bei der Rückgabe ihre
          -- Gültigkeit. Wenn das Feld länger gebraucht wird, muß mit own_buffer
          -- eigener Speicherplatz bereitgestellt werden.

     end; -- callback_general
```

```
callback_special is
      deferred
      end;



feature -- access

session: POINTER;

subject, reply_box: STRING;

msg: RV_MSG_FIELD



feature {NONE} -- implementation

init_c is
      -- Mit dieser Prozedur wird die Adresse von callback_general an den Bus
      -- übergeben.
      once
            eifrv_init_callback ($callback_general)
      end


end -- class RV_CALLBACK
```

7.2.5 Nachrichtenbearbeiter

```
deferred class RV_MSG_CALLBACK


-- Diese Klasse wird benötigt, um über die Felder einer Feldliste
-- (RV_MSG_LIST) zu iterieren. Die Prozeduren "callback_general" und
-- "callback_special" werden für jedes gefundene Feld aufgerufen.

-- Siehe Prozedur apply_callback in der Klasse RV_MSG_LIST.

-- In allen Unterklassen muß make bei der Exemplarerzeugung aufgerufen werden,
-- damit dem Bus die Adresse von callback_general mitgeteilt wird.
-- Eigene Initialisierungsoperationen können in der Prozedur make_special
-- implementiert werden.



inherit RV_OBJECT; RV_USER


feature {NONE} -- only for creation

make is
     do
          init_c;
          make_special
     end;

make_special is
     deferred
     end


feature {RV_MSG_LIST}

callback_general (   sess:       POINTER;
                     field_name: POINTER;
                     field_type: INTEGER;
                     field_size: INTEGER;
                     field_adr:  POINTER):BOOLEAN is
     local
          name:  STRING;
          field: RV_MSG_FIELD
     do
          check
               sess=rv.session
          end;
          -- Das das Feld repräsentierende Eiffel-Objekt wird vom Busverwalter
          -- erzeugt:
          field:= rv.field_from_bus(field_type, field_size, field_adr);
          name:="";
          name.from_c(field_name);
          field.set_name(name);
          Result:= callback_special(field);
     end; -- callback_general
```

```
callback_special(field: RV_MSG_FIELD):BOOLEAN is
      -- falls callback_special TRUE zurückliefert, wird die
      -- Iteration abgebrochen.
      deferred end;


feature {NONE} -- implementation

init_c is
      -- Übermittlung der Adresse von callback_general an C.
      once
            eifrv_init_msg_apply_cb ($callback_general)
      end;


end -- class RV_MSG_CALLBACK
```

7.2.6 Die Systembeschreibungsdatei

```
system
          bus_system

root

          ROOT_CLASS : "make"  -- Diese Klasse und ihre Startprozedur sind durch
                               -- die Wurzelklasse und Startprozedur der
                               -- Anwendung zu ersetzen.

default

     assertion (require);
     precompiled ("$EIFFEL3\precomp\spec\$PLATFORM\base")

cluster         ROOT_CLUSTER: ".";
                RV_CLUSTER:   ".\RV_CLUSTER";

external
     object:    ".\C_Files\eifrv.obj",          -- C-Anpassung- und C-
                                                -- Hilfsroutinen.

                "D:\RV\LIB\rv.lib",             -- RV-API-Bibliothek.

                "D:\MSVC20\LIB\advapi32.lib",   -- Für den Betrieb des Busses
                "D:\MSVC20\LIB\wsock32.lib";    -- werden auch diese beiden
                                                -- Bibliotheken benötigt.

end
```

7.3 C-Quelltexte

7.3.1 C-Anpassungsroutinen

```
/* ANFANG DER DATEI eifrv.c */

/*
Diese C-Quelltext-Datei muß übersetzt und zum Eiffel-System hinzugebunden werden.
Es werden Nicht-Standard Include-Dateien benötigt, die in den folgenden
Verzeichnissen gefunden werden können:
- .\bench\spec\w32msc\INCLUDE (im ISE-Eiffel-Verzeichnis) und
- .\INCLUDE (im Rendezvous-Verzeichnis)
*/



#if(__STDC__ != 1)
#error Please use a compiler conforming to the ANSI Standard!
#endif



/* Include-Dateien: */

/* stdlib.h is included in portable.h */
/* stdio.h is included in rv.h */
/* time.h is included in rv.h */

#include <rv.h>
#include <stddef.h>            /* Für die Definition von NULL */
#include <eiffel.h>



#define max_inbox_size 500
/* Wird für die Bereitstellung eines String-Puffers benötigt.*/



/* Hier werden die Adressen der Eiffel-Routinen abgelegt.*/
static EIF_PROC
eifrv_callback_routine;   /* Platz für die Adresse der Routine callback_general*/
                          /* der Klasse RV_CALLBACK */

static EIF_FN_BOOL
eifrv_msg_apply_cb_routine;     /* Platz für die Adresse der Routine */
                                /* callback_general der Klasse RV_MSG_CALLBACK */



/*Diese Funktion stellt die C-Konstante NULL auch Eiffel zur Verfuegung.*/
EIF_POINTER
eifrv_null()
  {
  return (EIF_POINTER) NULL;
  }
```

```
/* Mit Hilfe dieser Prozedur hinterlegt Eiffel die Adresse der */
/* Routine callback_general der Klasse RV_CALLBACK. */
void
eifrv_init_callback (EIF_PROC proc)
  {
   eifrv_callback_routine = proc;
  }


/* NACHRICHTEN-VERTEILER */
/* Diese Prozedur wird vom Bus als Callback-Prozedur aufgerufen und ruft*/
/* ihrerseits die Routine callback_general eines Eiffel-Empfänger-Objekts*/
/* auf. "obj" enthält einen Verweis auf das Empfänger-Objekt. Die Adresse von*/
/* callback_general muß sich in "eifrv_callback_routine" befinden.*/
void
eifrv_callback(
    rv_Session  session,
    rv_Name     name,
    rv_Name     replyName,
    rvmsg_Type  msgType,
    rvmsg_Size  msgSize,
    rvmsg_Data  msg,
    rv_Opaque   obj)
  {
   /* Eiffel erwartet als ersten Parameter einen Objektverweis.*/
   (eifrv_callback_routine)    (eif_access((EIF_OBJ)obj),
                                (EIF_POINTER)session,
                                (EIF_POINTER)name,
                                (EIF_POINTER)replyName,
                                (EIF_INTEGER)msgType,
                                (EIF_INTEGER)msgSize,
                                (EIF_POINTER)msg);
  }


/*Anpassung der Prozedur "rv_Init", siehe rv.h */
EIF_INTEGER
eifrv_init(EIF_POINTER pointer_to_session,
           EIF_OBJ     service,
           EIF_OBJ     network,
           EIF_OBJ     daemon )
  {
  rv_Error err;
  err = rv_Init( (rv_Session*) pointer_to_session,
                 (char*)       eif_access (service),
                 (char*)       eif_access (network),
                 (char*)       eif_access (daemon));
  return ((EIF_INTEGER) err);
  }
```

```
/* Anpassung der Prozedur "rv_Send", siehe rv.h */
EIF_INTEGER
eifrv_send(EIF_POINTER session,
           EIF_OBJ     name,
        EIF_INTEGER msgType,
        EIF_INTEGER msgSize,
        EIF_POINTER msg)
  {
   rv_Error err;

   err = rv_Send( (rv_Session)   session,
                  (rv_Name)      eif_access(name),
             (rvmsg_Type)   msgType,
             (rvmsg_Size)   msgSize,
             (rvmsg_Data)   msg);
   return ((EIF_INTEGER) err);
  }



/* Anpassung der Prozedur "rv_ListenSubject", siehe rv.h.        */
/* Die Callback-Adresse ist konstant, da immer die Prozedur      */
/* "eifrv_callback" aufgerufen wird. Unterschiedliche Callback-  */
/* Prozeduren werden durch variable Callback-Objekte realisiert, */
/* die mittels "rv_Opaque" an "eifrv_callback" weitergegeben     */
/* werden.                                                       */
EIF_INTEGER
eifrv_listen_subject(EIF_POINTER session,
                     EIF_POINTER pointer_to_listen_id,
                     EIF_OBJ     subject,
                     EIF_OBJ     callback_object)
  {
   rv_Error err;
   EIF_OBJ obj;
   obj = eif_adopt(callback_object);
   err = rv_ListenSubject((rv_Session)   session,
                          (rv_ListenId*) pointer_to_listen_id,
                          (rv_Name)      eif_access(subject),
                          (rv_Callback)  eifrv_callback,
                          (rv_Opaque)    obj);
   return ((EIF_INTEGER) err);
  }



/* Anpassung der Prozedur "rv_ListenInbox", siehe rv.h */
EIF_INTEGER
eifrv_listen_inbox(  EIF_POINTER session,
                     EIF_POINTER pointer_to_listen_id,
                     EIF_POINTER inbox, /*pointer to place for inbox name*/
                     EIF_OBJ     callback_object)
  {
   rv_Error err;
   EIF_OBJ obj;
   obj = eif_adopt(callback_object);
   err = rv_ListenInbox(  (rv_Session)   session,
                          (rv_ListenId*) pointer_to_listen_id,
                          (rv_Name)      inbox,
                          (rv_Size)      max_inbox_size,
                          (rv_Callback)  eifrv_callback,
                          (rv_Opaque)    obj);
   return ((EIF_INTEGER) err);
  }
```

```
/*Anpassung der Prozedur "rv_Rpc", siehe rv.h */
EIF_INTEGER
eifrv_rpc( EIF_POINTER session,
           EIF_POINTER pointer_to_listen_id,
           EIF_OBJ     subject,
           EIF_INTEGER m_type,
           EIF_INTEGER m_size,
           EIF_POINTER m_adr,
           EIF_OBJ     callback_object)
  {
   rv_Error err;
   EIF_OBJ obj;
   obj = eif_adopt(callback_object);
   err = rv_Rpc(     (rv_Session)   session,
                     (rv_ListenId*) pointer_to_listen_id,
                     (rv_Name)      eif_access(subject),
                     (rvmsg_Type)   m_type,
                     (rvmsg_Size)   m_size,
                     (rvmsg_Data)   m_adr,
                     (rv_Callback)  eifrv_callback,
                     (rv_Opaque)    obj);
   return ((EIF_INTEGER) err);
  }


/*Anpassung der Prozedur "rv_Close", siehe rv.h */
EIF_INTEGER
eifrv_close(    EIF_POINTER session,
                EIF_POINTER listen_id,
                EIF_OBJ     callback_object)
  {
   rv_Error err;
   err = rv_Close((rv_Session)  session,
                  (rv_ListenId) listen_id);
   eif_wean(callback_object);
   return ((EIF_INTEGER) err);
  }



/*************************************************************************/
/* Anpassungsroutinen zum RV-Message API, siehe rv.h                     */
/*************************************************************************/


/* Mit Hilfe dieser Prozedur hinterlegt Eiffel die Adresse der Message- */
/* Apply-Callback-Routine callback_general der Klasse RV_MSG_CALLBACK.*/
void
eifrv_init_msg_apply_cb (EIF_FN_BOOL func)
  {
   eifrv_msg_apply_cb_routine = func;
  }
```

```
/* Verteiler von Message-Callback-Aufrufen.                                  */
/* Der Mechanismus ist ganz analog zu dem des Callback-Verteilers.           */
/* Diese Prozedur wird vom Bus als Message-Callback-Prozedur aufgerufen und ruft*/
/* ihrerseits die Routine callback_general eines Nachrichtenbearbeiter-Objekts  */
/* auf. "obj" enthält einen Verweis auf das Nachrichtenbearbeiter-Objekts.   */
/* Adresse von callback_general muß sich in "eifrv_msg_apply_cb_routine"     */
/* befinden.                                                                 */
rv_Opaque
eifrv_msg_apply_cb(
    rv_Session  session,
    rv_Name     field_name,
    rvmsg_Type  field_type,
    rvmsg_Size  field_size,
    rvmsg_Data  field_adr,
    rv_Opaque   obj)
  {
   EIF_BOOLEAN result;
   /*Eiffel exspects the target object as first argument. */
   result= (eifrv_msg_apply_cb_routine) (
                               eif_access((EIF_OBJ)obj),
                              (EIF_POINTER)session,
                              (EIF_POINTER)field_name,
                              (EIF_INTEGER)field_type,
                              (EIF_INTEGER)field_size,
                                (EIF_POINTER)field_adr);
   return ((rv_Opaque)result);
  }



/*Anpassung der Prozedur "rvmsg_Init", siehe rv.h */
EIF_INTEGER
eifrv_msg_init(EIF_POINTER session,
               EIF_POINTER pointer_to_adr,
               EIF_INTEGER buffer_size)
  {
   rv_Error err;
   void* msg;
   msg= malloc((size_t)buffer_size);
   *(EIF_POINTER*)(pointer_to_adr)= (EIF_POINTER)msg;
   err= rvmsg_Init((rv_Session) session,
                   (rvmsg_Msg)  msg,
              (rvmsg_Size) buffer_size);
   return ((EIF_INTEGER) err);
  }
```

```
/*Anpassung der Prozedur "rvmsg_Append ", siehe rv.h */
EIF_INTEGER
eifrv_msg_append(EIF_POINTER session,
                 EIF_POINTER list_adr,
                 EIF_INTEGER buffer_size,
            EIF_OBJ     field_name,
            EIF_INTEGER field_type,
            EIF_INTEGER field_size,
            EIF_POINTER field_adr)
  {
   rv_Error err;
   rvmsg_Msg msg;
   msg= malloc((size_t)buffer_size);
   err= rvmsg_Append(     (rv_Session) session,
                          (rvmsg_Msg)  list_adr,
                          (rvmsg_Size) buffer_size,
                          (rv_Name)    eif_access(field_name),
                          (rvmsg_Type) field_type,
                          (rvmsg_Size) field_size,
                          (rvmsg_Data) field_adr);
   return ((EIF_INTEGER) err);
  }



/*Anpassung der Prozedur "rvmsg_Length", siehe rv.h */
EIF_INTEGER
eifrv_msg_length(EIF_POINTER session,
                 EIF_POINTER list_adr,
                 EIF_POINTER pointer_to_size)
  {
   rv_Error err;
   err= rvmsg_Length((rv_Session)  session,
               (rvmsg_Msg)   list_adr,
               (rvmsg_Size*) pointer_to_size);
   return ((EIF_INTEGER) err);
  }


/*Anpassung der Prozedur "eifrv_msg_get", siehe rv.h */
EIF_INTEGER
eifrv_msg_get(  EIF_POINTER session,
                EIF_POINTER list_adr,
                EIF_OBJ     field_name,
                EIF_POINTER pointer_to_type,
                EIF_POINTER pointer_to_size,
                EIF_POINTER pointer_to_adr)
  {
   rv_Error err;
   err= rvmsg_Get((rv_Session)  session,
             (rvmsg_Msg)   list_adr,
             (rv_Name)     eif_access(field_name),
             (rvmsg_Type*) pointer_to_type,
             (rvmsg_Size*) pointer_to_size,
             (rvmsg_Data*) pointer_to_adr);
   return ((EIF_INTEGER) err);
  }
```

```
/*Anpassung der Prozedur "rvmsg_Apply", siehe rv.h */
EIF_INTEGER
eifrv_msg_apply(     EIF_POINTER session,
                     EIF_POINTER list_adr,
                     EIF_OBJ     field_name,
                     EIF_OBJ     apply_callback_object)
  {
   rvmsg_Error err;
   rv_Opaque discarded;
   eif_adopt(apply_callback_object);
   err= rvmsg_Apply( (rv_Session)          session,
                     (rvmsg_Msg)           list_adr,
                     (rv_Name)             eif_access(field_name),
                     (rvmsg_ApplyCallback) eifrv_msg_apply_cb_routine,
                     (rv_Opaque)           apply_callback_object,
                                           &discarded);
   eif_wean(apply_callback_object);
   return (EIF_INTEGER) err;
  }
```

```
/*
```

7.3.2 C-Hilfsroutinen

```
*/

/* Anpassung der C-Standard-Routine malloc an Eiffel. */
EIF_POINTER
eifrv_malloc(EIF_INTEGER size)
  {
   EIF_POINTER Result;
   Result=(EIF_POINTER)(malloc((size_t)size));
   return (EIF_POINTER)Result;
  }


/* Kopierroutine                                                          */
/* Diese Routine kopiert den Inhalt eines Nachrichtenfelds in einen neu   */
/* bereitgestellten Speicherbereich. Dies ist zum Kopieren von Nachrichten-*/
/* feldern nötig.                                                         */
void
eifrv_copy(EIF_POINTER target,
           EIF_POINTER source,
           EIF_INTEGER size)
  {
   memcpy((void*)target, (void*)source, (size_t)size);
  }


/* Vergleichsroutine                                                      */
/* Diese Routine wird benötigt zum Vergleich des Inhalts zweier Nachrichten*/
/* felder.                                                                */
EIF_BOOLEAN
eifrv_cmp(EIF_POINTER adr1,
          EIF_POINTER adr2,
            EIF_INTEGER size)
  {
   return( (EIF_BOOLEAN) ( (memcmp((void*) adr1,
                                   (void*) adr2,
                                   (size_t)size)) == 0) );
  }
```

```
/* Diese Funktion wandelt die unterschiedlichen Integer-Typen */
/* einer RV-Message in EIF_INTEGER um. Voraussetzung fuer das */
/* Funktionieren sind folgende Groessen der Integer-Typen:    */
/* char: 1Byte, short: 2Byte, int: 4Byte                      */
EIF_INTEGER
eifrv_cv_int(   EIF_INTEGER field_type,
                EIF_INTEGER field_size,
                EIF_POINTER field_adr)
  {
   EIF_INTEGER Result;
      if ((rvmsg_Type)field_type == RVMSG_INT)
        {
           if      (field_size==4)  Result= *((int*  )field_adr);
           else if (field_size==2)  Result= *((short*)field_adr);
           else   /*field_size==1*/ Result= *((char* )field_adr);
        }
      else /*if ((rvmsg_Type)field_type == RVMSG_UINT)*/
        {
           if      (field_size==4)  Result= *((unsigned int  *)field_adr);
           else if (field_size==2)  Result= *((unsigned short*)field_adr);
           else   /*field_size==1*/ Result= *((unsigned char *)field_adr);
        }
   return Result;
  }


/* Diese Funktion wandelt die unterschiedlichen Real-Typen   */
/* einer RV-Message in EIF_DOUBLE um. Voraussetzung fuer das */
/* Funktionieren sind folgende Groessen der Integer-Typen:   */
/* float: 4Byte, double: 8Byte                               */
EIF_DOUBLE
eifrv_cv_double(EIF_INTEGER size, EIF_POINTER adr)
  {
   EIF_DOUBLE Result;
   if     (size==4)  Result= *( (float*)adr);
   else  /*size==8*/ Result= *((double*)adr);
   return Result;
  }


/* Diese Fumnktion stellt Speicher für einen übergebenen */
/* String bereit und kopiert den String dorthin.         */
EIF_POINTER
eifrv_mk_string(EIF_OBJ str, EIF_INTEGER length)
  {
   char* c_str;
   c_str= malloc((int)length);
   strcpy(c_str,(char*) eif_access(str));
   return ((EIF_POINTER) c_str);
  }


/* Diese Fumnktion stellt Speicher für einen übergebenen */
/* Integer-Wert bereit und kopiert den Wert dorthin.     */
/* Voraussetzung: sizeof(EIF_INTEGER) ist 4              */
EIF_POINTER
eifrv_mk_int(EIF_INTEGER i)
  {
   int* c_int;
   c_int= malloc(4);
   *c_int= i;
   return ((EIF_POINTER) c_int);
  }
```

```
/* Diese Funktion stellt Speicher für einen übergebenen  */
/* Double-Wert bereit und kopiert den Wert dorthin.      */
/* Voraussetzung: sizeof(EIF_DOUBLE) ist 8               */
EIF_POINTER
eifrv_mk_double(EIF_DOUBLE d)
  {
   double* c_doub;
   c_doub= (double*) malloc(8);
   *c_doub=(double) d;
   return ((EIF_POINTER) c_doub);
  }


/* ENDE DER DATEI eifrv.c */
```

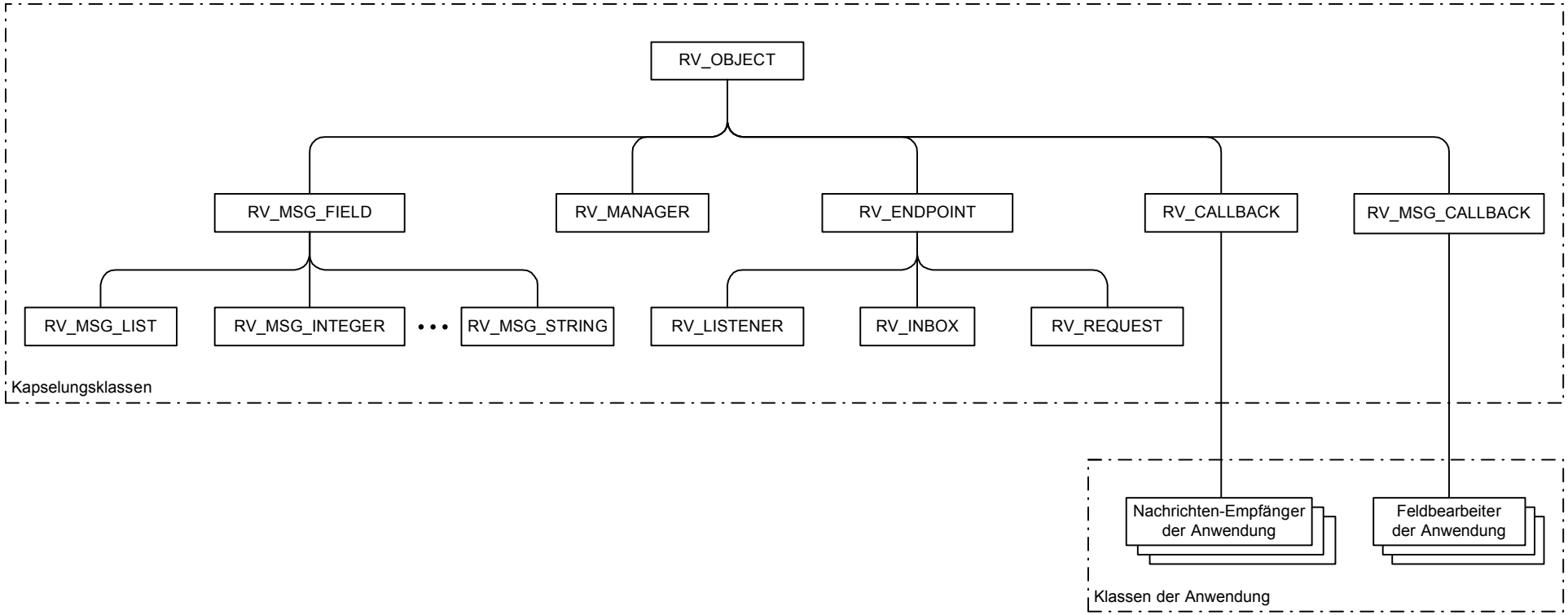


Bild I: Vererbungshierarchie der Kapselungsklassen

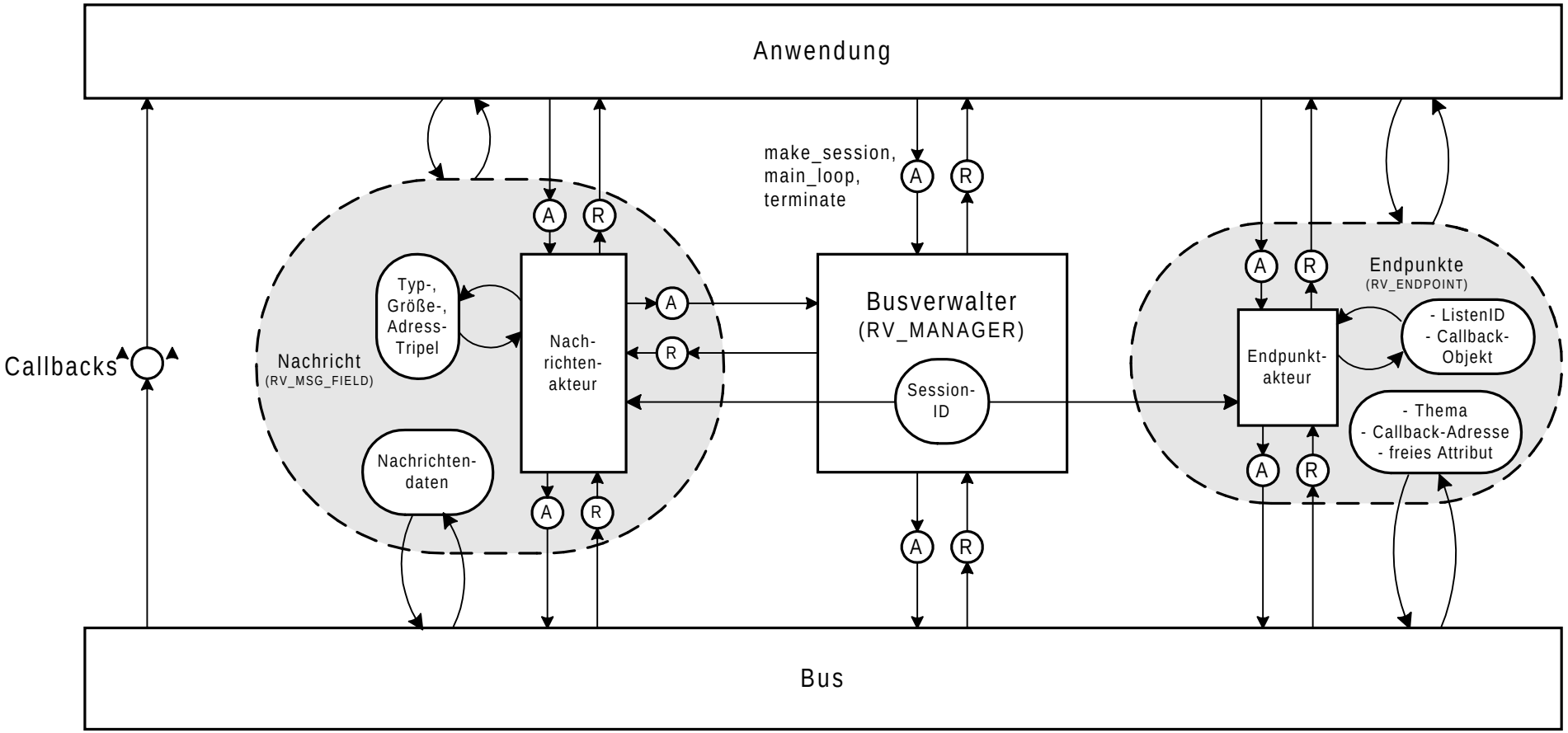


Bild II: Aufbaubild der Kapselung des Rendezvous-Busses in Eiffel

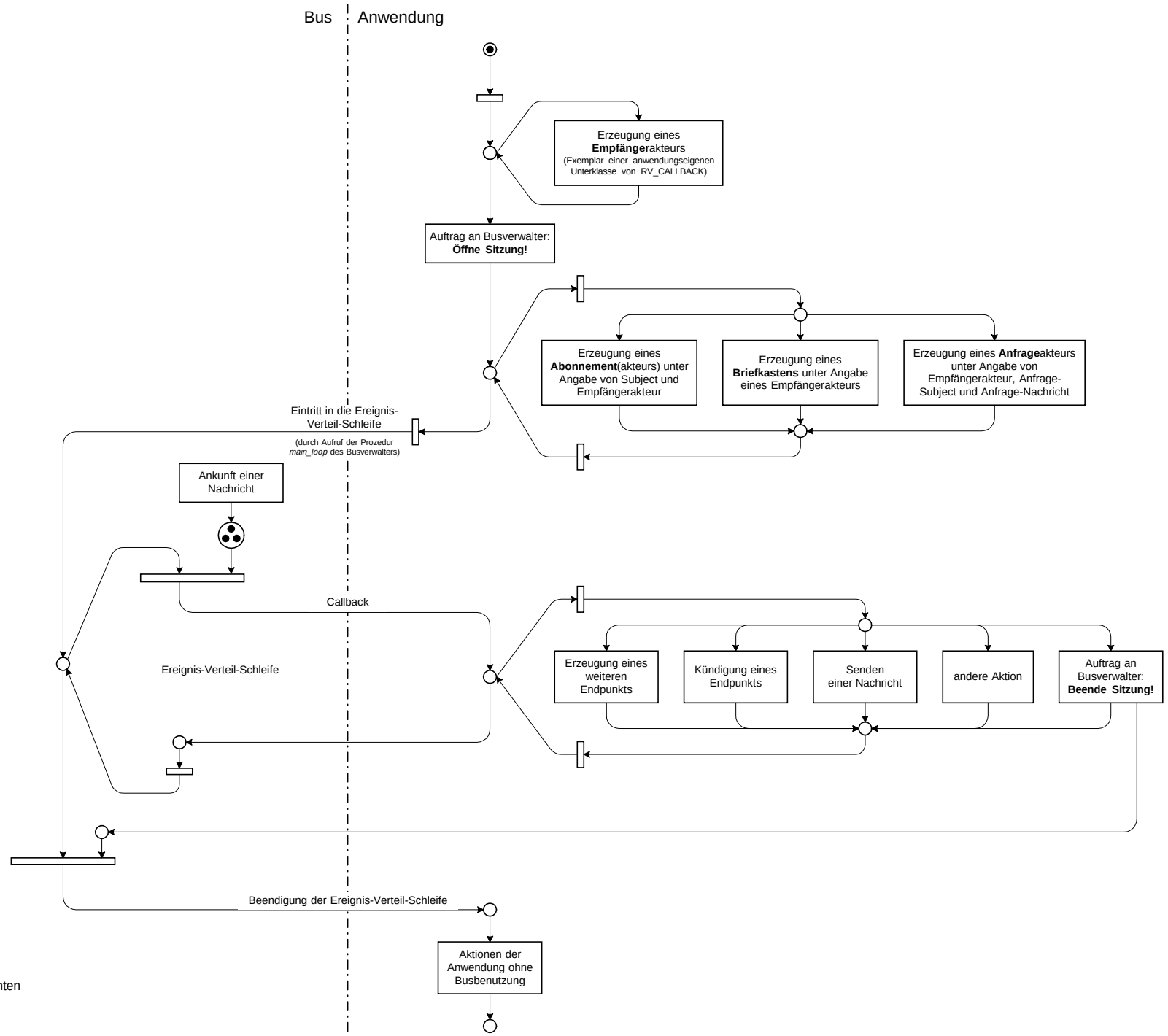


Bild III: Empfang von Nachrichten (typischer Ablauf)