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

Diplomarbeit

Bereitstellung einer Experimentierumgebung zur Entwicklung von Client-Server-Systemen in Smalltalk

Johannes Otto
August 1996

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

Bearbeiter: Johannes Otto
Alfred-Jost-Str. 13
69124 Heidelberg-Kirchheim

Erklärung:

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

Walldorf, den 31. August 1996

(Johannes Otto)

Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen dieser Arbeit beigetragen haben. Mein Dank gilt in erster Linie Herrn Prof. Dr.-Ing. Siegfried Wendt, der die Diplomarbeit im Rahmen seiner Forschungsaktivitäten bei SAP ermöglichte und betreute. Sein mir dabei entgegengebrachtes Vertrauen und der mir zugestandene Entscheidungsfreiraum waren außerordentlich groß und Basis einer durchweg sehr angenehmen Arbeitsatmosphäre.

Des weiteren möchte ich den Firmen meinen Dank aussprechen, die die im Rahmen dieser Arbeit eingesetzten Produkte bereitgestellt haben. Hier ist die Firma **Georg Heeg Objektorientierte Systeme** aus Dortmund zu nennen, die die Smalltalk-Entwicklungsumgebung VISUALWORKS von PARCPLACE/DIGITALK sowie das Objektdatenbank-System GEMSTONE zur Verfügung gestellt hat. Ihre darüber hinaus gebotene kompetente Unterstützung in allen technischen Fragen hat mir sehr geholfen. Der Firma **TIBCO** aus Palo Alto, Kalifornien, USA möchte ich für die Bereitstellung ihres Produkts RENDEZVOUS SOFTWARE BUS danken.

Die vorliegende Arbeit gehört zu einer Gruppe von drei Diplomarbeiten, die in den Monaten Dezember 1995 bis September 1996 im Hause SAP in Walldorf angefertigt wurden. Dabei wirkten neben mir meine Kollegen Bernhard Gröne und Eberhard Iglhaut mit, denen ich viele nützliche Anregungen verdanke. Ihre stete Bereitschaft zur Diskussion war mir sehr hilfreich.

Johannes Otto

Inhaltsverzeichnis

Abbildungsverzeichnis

Die folgenden Bilder befinden sich am Ende dieser Schrift:

1 Einführung

Die vorliegende Arbeit gehört zu einer Gruppe von drei Diplomarbeiten, die in den Monaten Dezember 1995 bis September 1996 im Hause SAP in Walldorf im Rahmen der dortigen Forschungsaktivitäten von Prof. Dr.-Ing. Siegfried Wendt entstanden. Ihr gemeinsames Ziel ist die Bereitstellung einer Experimentierumgebung, in der verschiedene objektorientierte Entwicklungswerkzeuge und Datenbanksysteme integriert sind. Dieses sogenannte Software-Technologie-Labor [Gröne 96, Kapitel 1] soll einerseits dazu dienen, neue Technologien und Produkte im praktischen Einsatz zu testen und zu bewerten. Andererseits soll es aber auch als Entwicklungsplattform für Anwendungsprototypen benutzbar sein, anhand derer neue Softwarearchitektur-Konzepte untersucht und beurteilt werden können. Gegenstand der vorliegenden Arbeit ist die Bereitstellung derjenigen Komponenten des Labors, die die Programmiersprache Smalltalk betreffen. Daneben kommen als weitere Programmiersprachen Eiffel und C++ vor.

Die Arbeit ist in zwei Teile gegliedert. Im ersten Teil (Kapitel 2) werden zunächst die Grundlagen von Smalltalk vermittelt. Dabei werden lediglich die wichtigsten Konzepte vorgestellt. Für die vollständige Smalltalk-Spezifikation inklusive der Syntax-Beschreibung sei der Leser auf das Standardwerk [Goldberg 89] verwiesen. Der umfangreichere zweite Teil (Kapitel 3) dokumentiert den Entwurf einer Architektur für objektorientierte Client-Server-Systeme, so wie sie auf der durch das Software-Technologie-Labor bereitgestellten Entwicklungsplattform realisiert werden können sollen. Eine zentrale Rolle spielt dabei der Einsatz eines Softwarebusses zur Verbindung der Komponenten eines solchen Client-Server-Systems. Im Zusammenhang mit den Entwurfsarbeiten entstanden eine Reihe von Smalltalk-Eigenentwicklungen, die ebenfalls innerhalb von Kapitel 3 vorgestellt werden. Sie ermöglichen es, mit wenig Aufwand konkrete Anwendungssysteme des durch den Architekturentwurf vorgegebenen Aufbaus unter Nutzung von Smalltalk zu realisieren.

2 Smalltalk Grundlagen

Dieses Kapitel behandelt die grundlegenden Konzepte von Smalltalk. Die dabei gemachten Aussagen gelten grundsätzlich für alle verfügbaren Smalltalk-Implementierungen. Überprüft wurden sie durch Benutzung der Smalltalk-Entwicklungsumgebung **VISUALWORKS 2.5** des Herstellers **Parcplace/Digitalk** (Sunnyvale, Kalifornien, USA). Dieses Produkt ist das jüngste aus der Produktreihe, die mit der ersten öffentlich verfügbar gemachten Version von Smalltalk-80 im Jahr 1981 begann.

2.1 Einordnung

Historisches

Die Wurzeln von Smalltalk gehen zurück bis ins Jahr 1969. Damals formulierte Alan Kay in seiner Dissertation[1] eine Vision der Kommunikation zwischen Mensch und Rechner, die er später als Leiter der „Learning Research Group“ des Xerox Palo Alto Research Centers (PARC) realisierte. Zwischen 1970 und 1980 entstanden dort in einer Art Bootstrap-Verfahren mit einer Zykluszeit von etwa zwei Jahren die Smalltalk-Versionen 72, 74, 76, 78 und 80 [Ganzinger 87]. Der Name „Smalltalk“ sollte dabei das Ziel verdeutlichen, eine möglichst einfache und intuitiv erlernbare Art der Kommunikation zwischen Mensch und Rechner zu ermöglichen. Neben der eigentlichen Programmiersprache Smalltalk wurde dort auch das Arbeiten mit der Maus und das Konzept der fensterorientierten Benutzeroberflächen erfunden bzw. entwickelt. Die endgültige Version Smalltalk-80 wurde 1981 der Öffenlichkeit vorgestellt und später von der eigens dazu gegründeten Xerox-Tochter „Parcplace Systems“ (heute Parcplace/Digitalk) vermarktet und weiterentwickelt. Deren Präsidentin Adele Goldberg war auch schon während der Arbeiten im PARC maßgeblich an der Weiterentwicklung bis hin zur Produktreife beteiligt [Hoffmann 90]. Von ihr stammen die Smalltalk-Standardwerke [Goldberg 83] und [Goldberg 84], später ersetzt durch [Goldberg 89]. Nach der Veröffentlichung im Jahr 1981 entstanden getrennt davon weitere Implementierungen anderer Hersteller auf verschiedensten Plattformen. Näheres zur Smalltalk-Geschichte findet man in [Krasner 83].

---

[1] Alan Kay: „The Reactive Engine“, University of Utah, 1969

Fundamentale Eigenschaften

Smalltalk zählt zu den objektorientierten Programmiersprachen. Zur Smalltalk-Spezifikation gehört jedoch neben der eigentliche Sprache auch ein Interaktionskonzept, realisiert durch verschiedene Entwicklungswerkzeuge wie Browser, Debugger und Editoren innerhalb eines eigenen Fenstersystems. Daher muß man mit dem Begriff Smalltalk mehr verbinden als nur die Grammatik einer Programmiersprache.

Smalltalk ist eine typfreie Sprache. Polymorphismus ist also nicht erst durch Benutzung des Vererbungsmechanismus erreichbar.[2] Eine weitere fundamentale Eigenschaft besteht darin, daß Smalltalk-Code nicht – wie bei C++ oder Eiffel – übersetzt, sondern interpretiert wird. Smalltalk unterstützt keine Mehrfachvererbung.

2.2 Programmiermodell

2.2.1 Virtuelle Maschine und Objektspeicher

Bild 2.1 zeigt den Aufbau eines Smalltalk-Systems. Ein solches System besteht aus einer *virtuellen Smalltalkmaschine,* die auf einem *virtuellen Objektspeicher*, dem sogenannten Image, arbeitet. Dieser Objektspeicher enthält gleichberechtigt Benutzerobjekte und Systemobjekte. Benutzerobjekte können Exemplare beliebiger Klassen (Benutzerklassen oder Systemklassen) sein oder aber benutzerdefinierte Klassenobjekte, die die Benutzerklassenbeschreibungen enthalten. (Das Konzept der Klassenobjekte wird im folgenden Abschnitt 2.2.2 noch genauer vorgestellt.) Zu den Systemobjekten zählen zum einen die Systemklassenobjekte, die die mitgelieferte Klassenbibliothek repräsentieren. Zum anderen gehören dazu die Entwicklungswerkzeuge, die selbst Exemplare bestimmter Systemklassen sind. Alle Objekte im Objektspeicher sind frei zugänglich. Insbesondere sind Klassenobjekte unabhängig davon, ob sie System- oder Benutzerklassen repräsentieren, beliebig änderbar. Der Objektspeicher läßt sich als Ganzes abspeichern und laden und ist somit austauschbar. Die Entwicklung einer Smalltalk-Anwendung geschieht durch Erweiterung eines mit allen notwendigen Systemobjekten initial vorbelegten Objektspeichers. Während der Anwendungsprogrammierung fügt der Benutzer neue Objekte zu und/oder ändert Systemobjekte.

Bild 2.2 veranschaulicht, wie man sich den Inhalt eines Objektspeichers vorstellen muß. Er enthält ein durch Verweise unregelmäßig verbundenes Objektnetz, dessen Wurzelobjekt ein Symbol namens **Smalltalk** ist. Dieses verweist auf die Tabelle der Smalltalk-Globalsymbole. Zu den Globalsymbolen gehören vor allem die Klassennamen, aber auch benutzerdefinierte globale Variablennamen. Eine solche globale Variable kann zum Beispiel den Verweis auf das Wurzelobjekt einer Anwendung enthalten. Der Objektspeicher wird saubergehalten durch

[2] Der Begriff Polymorphismus wird in der Literatur unterschiedlich verwendet. Hier gilt die Definition aus [Meyer 90, S. 241].

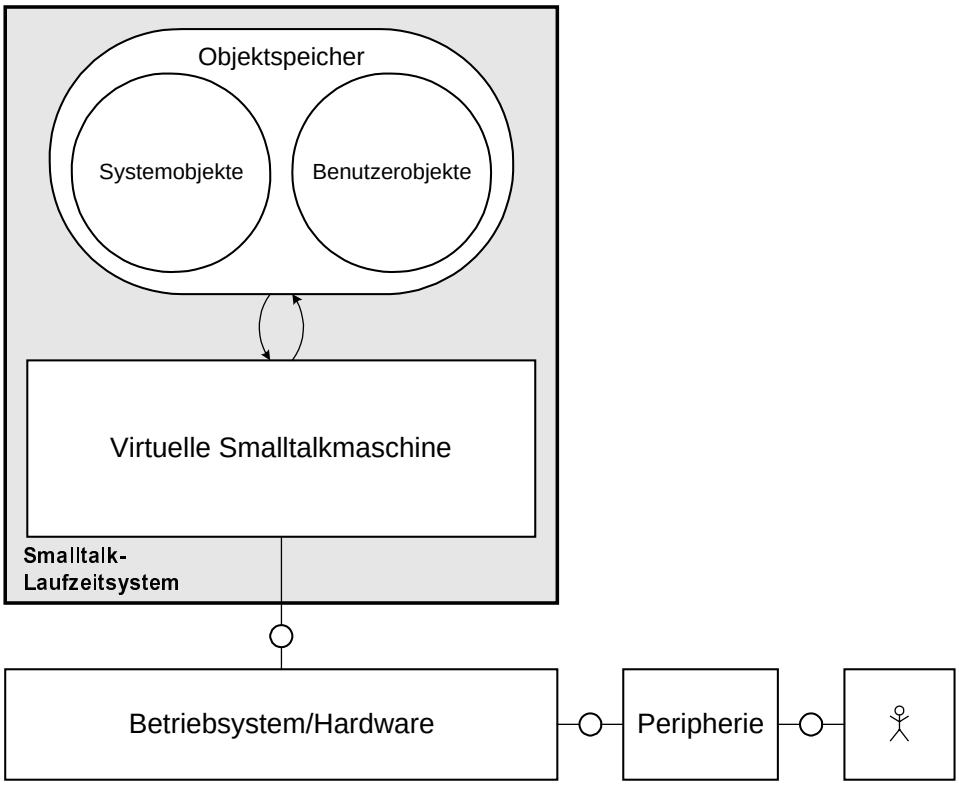


Bild 2.1 Smalltalk-System

die automatische Garbage Collection, die den Speicherbereich aller derjenigen Objekte freigibt, die nicht mehr ausgehend von ‘Smalltalk’ auf irgendeinem Pfad erreichbar sind.

Nach Abschluß der Entwicklung einer Smalltalk-Anwendung möchte man ein Smalltalk-Laufzeitsystem erhalten, in dem die Entwicklungswerkzeuge und andere nicht benötigte System- und Benutzerobjekte nicht mehr vorkommen. Dazu kann man sich mit Hilfe bestimmter Werkzeuge aus dem ursprünglichen Entwicklungs-Image das sogenannte *Anwendungs-Image* ausschneiden lassen. Dieses Anwendungsimage enthält nur noch diejenigen Objekte aus dem ursprünglichen Objektspeicher, die zur Ausführung der Anwendung benötigt werden. Der

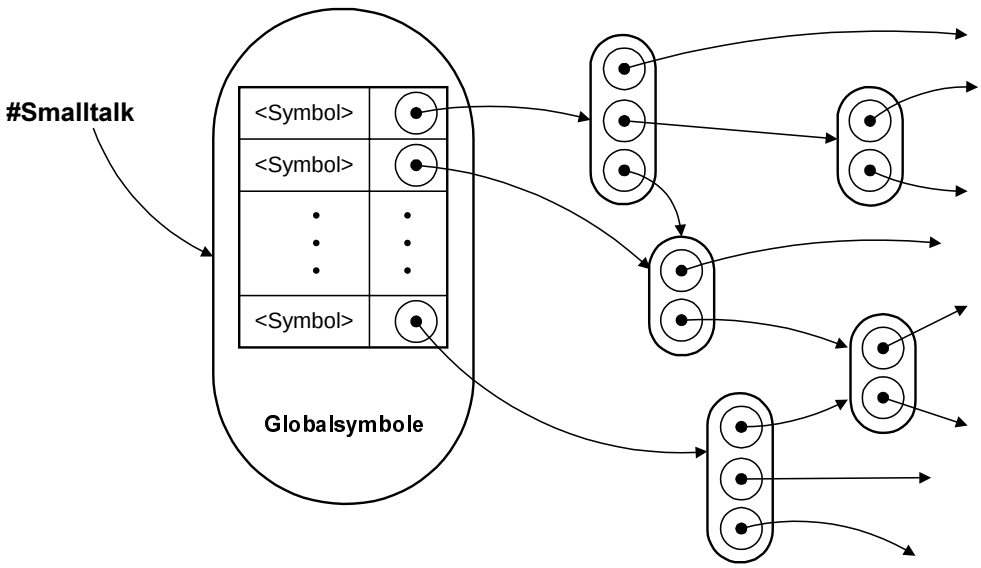


Bild 2.2 Inhalt des Objektspeichers

Anwendungsbenutzer braucht dann zwei Dinge: Zum einen benötigt er das genannte Anwendungsimage, zum anderen eine virtuelle Smalltalkmaschine für diejenige Plattform, auf der er die Anwendung zur Ausführung bringen will. Damit sind Smalltalk-Anwendungen automatisch plattformunabhängig, denn bezogen auf eine Smalltalk-Version eines Herstellers ist die Objektrepräsentation im Image über alle verfügbaren Plattformen einheitlich. Die virtuelle Smalltalkmaschine wird für verschiedene Plattformen angeboten und beinhaltet zusätzlich ein eigenes plattformunabhängiges Fenster-System. Smalltalk-Anwendungen laufen damit ohne irgendwelche Änderungen auf allen vom jeweiligen Smalltalk-Hersteller unterstützten Plattformen.

2.2.2 Objektmodell

Objekte und Klassen

Es wurde bereits im vorigen Abschnitt erwähnt, daß in Smalltalk neben gewöhnlichen Objekten auch die Klassenbeschreibungen als Objekte im Laufzeitsystem auftauchen. Zu jedem Klassenobjekt existiert zusätzlich ein Metaklassenobjekt, das den Bauplan des zugehörigen Klassenobjekts beinhaltet. Das folgende ER-Diagramm (Bild 2.3) verdeutlicht diesen Sachverhalt. Die im Smalltalk-Objektspeicher auftauchenden Objektarten kommen dort als grau unterlegte Entitätstypen im unteren Bildbereich vor. Die anderen Entitäten sind Klassen im Sinne von Typabstraktionen.

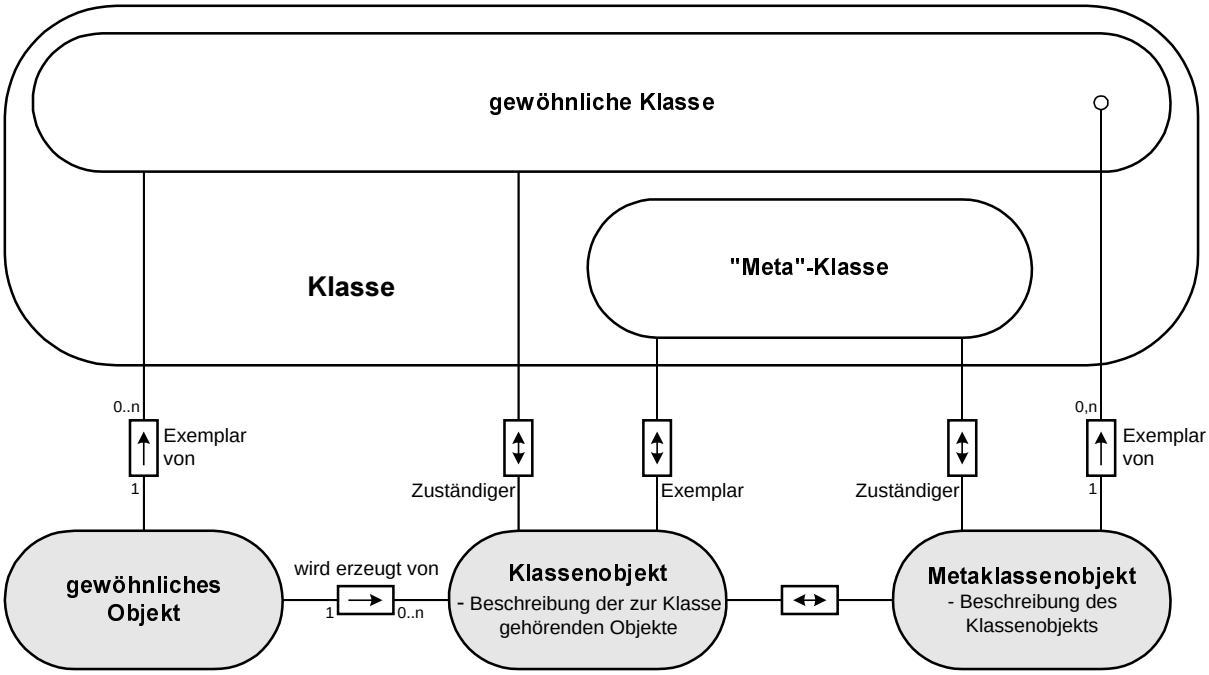


Bild 2.3 Objekte und Klassen in Smalltalk

Gewöhnliche Objekte im Smalltalk-Objektspeicher werden aufgefaßt als Exemplare gewöhnlicher Klassen. Für jede solche Klasse existiert im Objektspeicher ein zuständiges Klassenobjekt, das den Bauplan der Exemplare dieser Klasse enthält. Zusätzlich treten diese Klassenob-

jekte als Akteure auf und übernehmen die Erzeugung der Exemplare der Klasse, für die sie zuständig sind. Auch Klassenobjekte werden als Exemplare von Klassen aufgefaßt. Dabei stellt man sich vor, daß ein Klassenobjekt einziges Exemplar einer sogenannten „Meta“-Klasse ist, für die wiederum ein zuständiges Objekt – das Metaklassenobjekt – im Smalltalk-Objektspeicher existiert. Dieses trägt den Bauplan des zugehörigen Klassenobjekts.

„Meta“ steht deshalb in Anführungszeichen, da es sich bei den hier auftretenden „Meta“-Klassen nicht um eine Klassifikation von Klassen handelt, sondern eine Klassifikation bestimmter Objekte, eben der Klassenobjekte. Und diese Klassifikation geschieht auf eine sehr einfache Art, nämlich indem man sich zu jedem solchen Objekt eine eigene Klasse denkt, deren einziges Exemplar das Objekt ist. Klassenobjekte und Metaklassenobjekte treten also immer als Paare auf. Durch Edition einer neuen Klasse stößt der Benutzer die Erzeugung solcher Objektpaare an.

Alle Metaklassenobjekte sind wiederum Exemplare einer einzigen gewöhnlichen Klasse. Diese Exemplar-von-Beziehung aller Metaklassenobjekte zu einem einzigen Element aus der Menge der gewöhnlichen Klassen ist in Bild 2.3 durch die in den Entitätstypknoten mit der Aufschrift „gewöhnliche Klasse“ hineinragende Kante symbolisiert. Für die Klasse aller Metaklassenobjekte gibt es wiederum ein zuständiges Klassenobjekt im Objektspeicher. Bild 2.4 veranschaulicht diesen Sachverhalt noch einmal auf eine andere Weise. Objekte aus dem Objektspeicher sind hier als Quadrate, Klassen als grau unterlegte Kreise dargestellt. Die Kanten verbinden ein oder mehrere Quadrate mit außerhalb der Quadrate liegenden Kreisen. Sie drücken so die Exemplar-von-Beziehungen zwischen Objekten und Klassen aus. Ein Quadrat, das einen Kreis umrahmt, steht für ein Objekt, das für die innerhalb des Quadrats liegende Klasse zuständig ist, also ein Klassenobjekt oder ein Metaklassenobjekt.

Auf unterster Ebene liegen die gewöhnlichen Objekte. Sie sind Exemplare gewöhnlicher Klassen, die auf der nächsthöheren Ebene durch Kreise symbolisiert werden. Zu jeder dieser gewöhnlichen Klassen existiert jeweils genau ein Klassenobjekt, symbolisiert durch das die Klasse umrahmende Quadrat. Jedes Klassenobjekt ist einziges Exemplar seiner „Meta“-Klasse, die als Kreis über diesem Klassenobjekt auftaucht. Auch für jede „Meta“-Klasse existiert jeweils ein zuständiges Objekt, das Metaklassenobjekt. Alle Metaklassenobjekte sind Exemplare einer einzigen gewöhnlichen Klasse, deren zuständiges Klassenobjekt den Namen **Metaclass** trägt. Auch ‘Metaclass’ ist einziges Exemplar einer „Meta“-Klasse, deren zuständiges Metaklassenobjekt mit **Metaclass class** bezeichnet wird. Dieses Metaklassenobjekt ist wiederum Exemplar der Klasse aller Metaklassenobjekte. ‘Metaclass’ ist also Exemplar derjenigen Klasse, die im Objektspeicher durch das Objekt ‘Metaclass class’ repräsentiert wird. Umgekehrt ist ‘Metaclass class’ Exemplar der Klasse, die durch das Objekt ‘Metaclass’ repräsentiert wird. Man erhält eine zyklische Beziehung in vier Stufen: Das Objekt ‘Metaclass’ ist *Exemplar* seiner „Meta“-Klasse. Das für sie *zuständige Objekt* heißt ‘Metaclass class’ und ist *Exemplar* der Klasse aller Metaklassenobjekte. Deren *zuständiges Objekt* ist wiederum ‘Metaclass’.

‘Metaclass’ und ‘Metaclass class’ sind a priori vorhanden und werden nicht erzeugt. Ansonsten müßte man sich fragen, wer hier wen erzeugt. In allen anderen Fällen sind die Klassenobjekte für die Erzeugung der gewöhnlichen Objekte und die Metaklassenobjekte für die Erzeugung der Klassenobjekte zuständig.

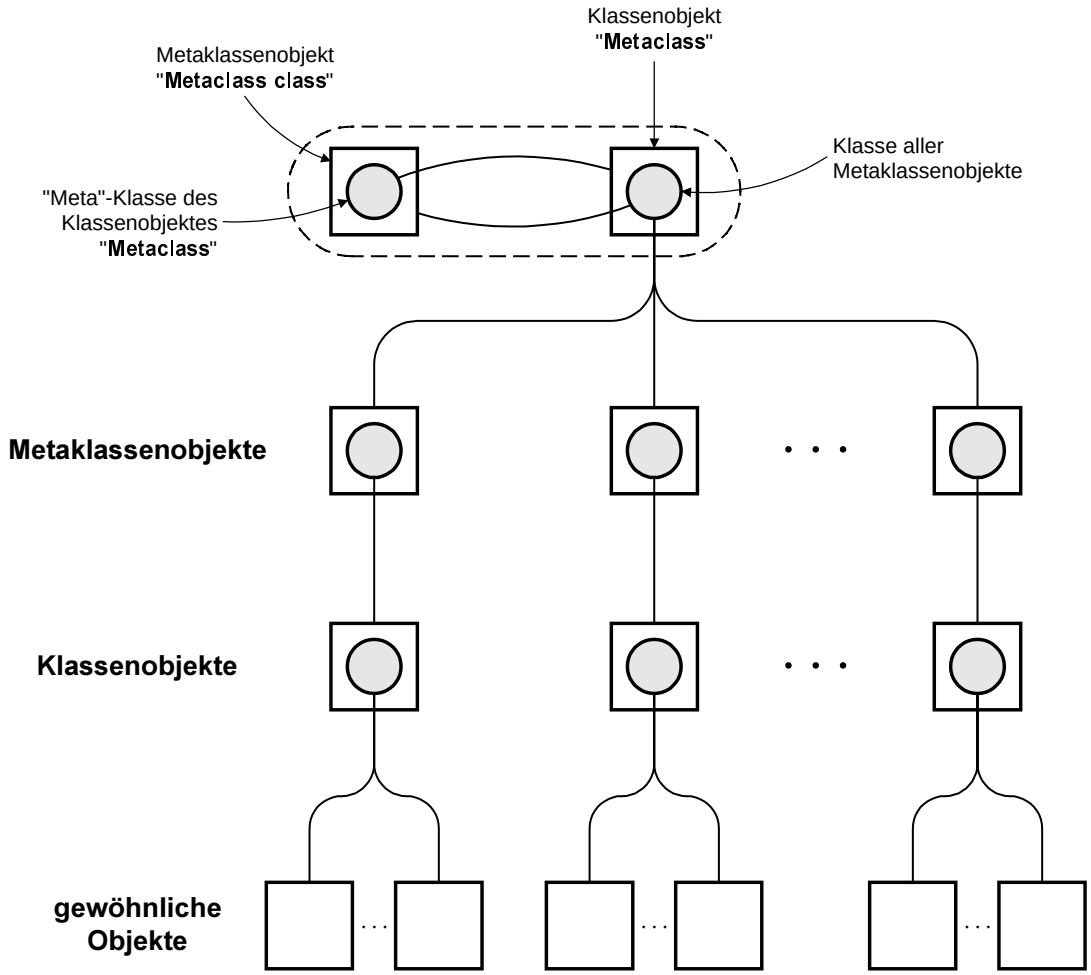


Bild 2.4 Objekte, Klassenobjekte, Metaklassenobjekte

Objektmerkmale

Auch in Smalltalk haben alle Objekte *Attribute* und *Methoden*. Gewöhnliche Objekte (Exemplare gewöhnlicher Klassen) dürfen zusätzlich *Klassenattribute* und *Pool-Attribute* besitzen. Klassenattribute sind Variablen, die sich alle Exemplare einer Klasse gemeinsam teilen. Ändert sich durch Aufruf einer Methode eines Objekts ein solches Klassenattribut, gilt der neue Wert auch für alle anderen Exemplare dieser Klasse. Pool-Attribute werden darüber hinaus von allen Exemplaren einer Gruppe von Klassen geteilt. Da Pool-Attribute damit keiner Klasse fest zugeordnet werden können, werden sie gruppenweise in speziellen Containern, im folgenden *Attribut-Pools* genannt, verwaltet.

Klassenobjekte besitzen ebenfalls Attribute und Methoden. Zur besseren Unterscheidung werden Attribute von Klassenobjekten als *Klassenobjektattribute* und Methoden von Klassenobjekten als *Klassenmethoden* bezeichnet. Mindestens eine Klassenmethode dient immer der Exemplarerzeugung. Es können aber durchaus weitere Klassenmethoden mit beliebiger Semantik eingeführt werden. Klassenobjektattribute sind nicht zu verwechseln mit den oben genannten Klassenattributen. Während auf Attribute von Klassenobjekten nur innerhalb der Klassenmethoden – also der Methoden des Klassenobjekts – zugegriffen werden kann, sind

Klassenattribute sowohl in den Methoden der Objekte als auch in den Klassenmethoden des für die Klasse der Objekte zuständigen Klassenobjekts sichtbar. In der folgenden Tabelle 2.1 sind noch einmal alle Attributarten und ihre Geltungsbereiche zusammengestellt.

Attributart	**Geltungsbereich**	
	Methoden (Objekt)	Klassenmethoden (Klassenobjekt)
Attribute	X	
Pool-Attribute	X	X
Klassenattribute	X	X
Klassenobjektattribute		X

Tabelle 2.1 Attributarten und ihre Geltungsbereiche

Es wurde schon erwähnt, daß Klassenobjekte den Bauplan der Exemplare der durch sie repräsentierten Klasse tragen und Metaklassenobjekte den Bauplan der zugehörigen Klassenobjekte. Dies kann nun präzisiert werden: Klassenobjekte enthalten die Deklarationen der Attribute und Klassenattribute, die Methodendefinitionen sowie Verweise auf eventuell benutzte Attribut-Pools. Im zugehörigen Metaklassenobjekt findet man die Definitionen der Klassenmethoden und die Deklarationen der Klassenobjektattribute.

Smalltalk bietet im Gegensatz zu Eiffel nicht die Möglichkeit einer flexiblen Objektschnittstellendefinition. In Smalltalk sind grundsätzlich alle Attribute privat und alle Methoden öffentlich. Wenn man möchte, daß bestimmte Methoden nicht von außen aufgerufen werden, muß man im Kommentar darauf hinweisen. (Dabei hat sich eine Konvention eingebürgert, auf die in Abschnitt 2.2.4 im Zusammenhang mit der Vorstellung der Entwicklungsumgebung eingegangen wird.) Möchte man umgekehrt Attribute öffentlich zugänglich machen, muß man entsprechende Zugriffsmethoden zum Lesen und Setzen des Attributs definieren.

2.2.3 Interpretationsvorschriften

Verweissemantik

Im Unterschied zu C++ gilt in Smalltalk grundsätzlich Verweissemantik. Alle Variablen beinhalten Zeiger auf Objekte. Die einzigen Ausnahmen machen Variablen, die Ganzzahlen, ASCII-Zeichen oder Wahrheitswerte repräsentieren. Solche Variablen tragen ihre Werte direkt. Da Smalltalk eine typfreie Sprache ist, ist es aber durchaus möglich, daß ein und dieselbe Variable zur Laufzeit einmal eine Integerzahl trägt und ein anderes Mal auf ein komplexes Objekt verweist. Also müssen die Werte von Ganzzahlen, ASCII-Zeichen und Wahrheitswerten intern genauso repräsentiert werden wie Zeiger auf Objekte. Die virtuelle Smalltalkma-

schine erkennt bei Auswertung solcher *Pseudozeiger* anhand der Codierung, daß es sich nicht um eine echte Objektreferenz handelt und extrahiert den Wert jeweils direkt aus der Pseudozeigercodierung.

Smalltalk-Anweisungen

In Smalltalk bezeichnet man den Aufruf einer Methode als das Senden einer Nachricht an ein Objekt. Nach der durch den Empfang einer solchen Nachricht ausgelösten Methodenausführung wird immer ein Verweis auf ein Ergebnisobjekt an den Sender zurückgegeben. Wird in der Methode kein Ergebnisobjekt spezifiziert – handelt es sich bei der Methode also um eine Prozedur –, wird standardmäßig ein Verweis auf das Empfängerobjekt zurückgegeben. Das Senden von Nachrichten wird ausgelöst durch Smalltalk-Anweisungen. Sie haben immer die in Bild 2.5 gezeigte Form:

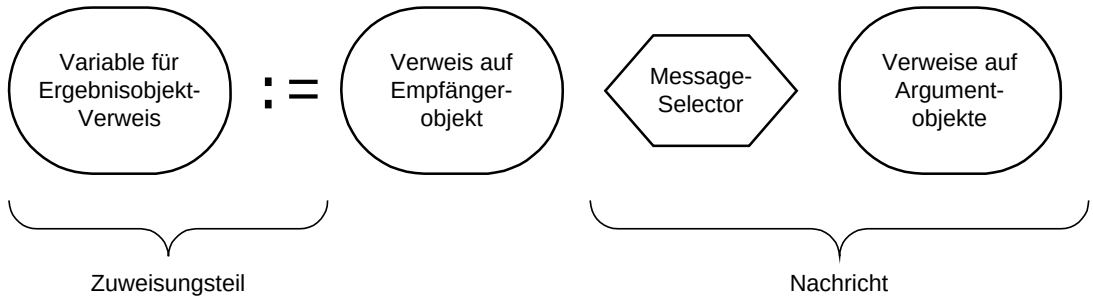


Bild 2.5 Smalltalk-Anweisung

Nach einem optionalen Zuweisungsteil, in dem eine Variable zur Speicherung des Verweises auf das Ergebnisobjekt identifiziert wird, folgt der Verweis auf das Empfängerobjekt. Dahinter wird die Nachricht an das Empfängerobjekt spezifiziert. Eine solche Nachricht besteht aus einem *Message-Selector*, der die aufzurufende Methode identifiziert, und den Verweisen auf die Argumentobjekte, sofern die Methode Argumente besitzt. Sowohl beim Verweis auf das Empfängerobjekt als auch bei den Verweisen auf die Argumentobjekte kann es sich auch um die oben erwähnten Pseudozeiger handeln. Für den Fall, daß der Empfänger durch einen Pseudozeiger identifiziert wird, existiert kein Empfängerobjekt im Objektspeicher. Vielmehr ist hier der Empfänger die virtuelle Smalltalkmaschine. Trotzdem existieren auch zu Wahrheitswerten, Ganzzahlen und ASCII-Zeichen entsprechende Klassenobjekte (**Boolean**, **SmallInteger**, **Character**), in denen jedoch keine Attribute, sondern nur Methoden definiert sind. Die virtuelle Smalltalkmaschine führt nach Nachrichtenempfang die durch diese Methoden definierten Operationen mit dem in der Pseudozeigercodierung enthaltenen Wert aus.

Smalltalk-Anweisungen dürfen verkettet werden. In solchen verketteten Anweisungen stehen hinter dem Identifikator des Empfängerobjekts mehrere Nachrichten, die nach einer festgelegten Prioritätsregel (dokumentiert zum Beispiel in [Goldberg 89, S. 28ff.]) ausgewertet werden. Innerhalb einer Prioritätsebene werden die Nachrichten von links nach rechts ausgewertet. Dabei gilt, daß nachfolgende Nachrichten jeweils an das Ergebnisobjekt, das man als Rückmeldung auf die Vorgängernachricht erhalten hat, gesendet werden. Durch Setzen von Klammern kann man die Abarbeitungsreihenfolge beeinflussen. Beispiele dazu folgen weiter unten.

Neben den in Bild 2.5 gezeigten Anweisungen, die immer einen Methodenaufruf veranlassen, ist auch der Spezialfall der einfachen Zuweisung möglich (Bild 2.6):

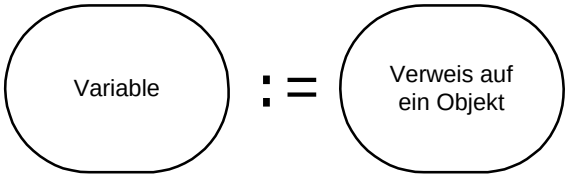


Bild 2.6 Einfache Zuweisung als Spezialfall einer Smalltalk-Anweisung

Die Erfinder von Smalltalk verfolgten das Ziel, eine Sprache zu entwickeln, in der für jede Anweisung dieselbe Interpretationsvorschrift gilt. So gilt die anhand von Bild 2.5 erläuterte Interpretationsvorschrift sogar bei zahlenarithmetischen Ausdrücken und Kontrollstrukturen. Wie unnatürlich diese Art der Interpretation in solchen Fällen ist, zeigen folgende Beispiele:

Beispiel 1:
```
a := 4 + 5 · 6.
```

Hier handelt es sich um eine verkettete Anweisung. Bei deren Auswertung wird zunächst dem „Objekt“ **4** die Nachricht ‘**+ 5**’ geschickt. **+** ist hier der Message-Selector, **5** das Argument-„Objekt“. An das resultierende „Objekt“ **9** wird daraufhin die Nachricht ‘**· 6**’ gesendet. Schließlich wird das Ergebnis-„Objekt“ **54** als Pseudozeiger in Variable **a** gespeichert.

Beispiel 2:
```
5 timesRepeat:
    [ Transcript show: ‘Hello World !’. ]
```

Das „Objekt“ **5** erhält hier eine einargumentige Nachricht, bestehend aus dem Message-Selector **timesRepeat** und einem Block-Objekt als Argument. Ein Block-Objekt enthält eine Sequenz von Smalltalk-Anweisungen. Definiert werden solche Objekte durch Auflistung der enthaltenen Anweisungen innerhalb eckiger Klammern. Das hier gezeigte Block-Objekt enthält genau eine Anweisung, die das Ausdrucken der Zeichenkette ‘Hello World !’ im allgemeinen Ausgabefenster (identifiziert durch das Globalsymbol **Transscript**) veranlaßt. Verweise auf Block-Objekte können, wie alle Objektverweise, Variablen zugewiesen werden. Nach der allgemein gültigen Interpretationsvorschrift wird durch die Anweisung aus Beispiel 2 das „Objekt“ **5** dazu veranlaßt, fünfmal die Ausführung der im übergebenen Block-Objekt enthaltenen Anweisung anzustoßen.

Beide Beispiele zeigen, das in Smalltalk Zahlen als aktionsfähige Objekte aufgefaßt werden sollen. Beispiel 1 zeigt zusätzlich, daß sogar die Prioritätsregel „Punktrechnung vor Strichrechnung“ außer Kraft gesetzt wird, um auch die Zahlenarithmetik in das vorgegebene Interpretationsschema pressen zu können. Erst durch das Setzen von Klammern erhält man bei Auswertung das erwartete Ergebnis:

Beispiel 3:
```
a := 4 + (5 · 6).
```

Abweichend von Beispiel 1 wird hier zunächst die Nachricht ‘**· 6**’ an das „Objekt“ **5** gesendet und anschließend der Pseudoverweis auf das daraus resultierende Ergebnis-„Objekt“ **30** als Argument innerhalb der durch **+** eingeleiteten Nachricht an das „Objekt“ **4** gesendet. Schließlich wird das daraus resultierende Ergebnis-„Objekt“ **34** als Pseudoverweis in der Variable **a** gespeichert.

Es sei angemerkt, daß man als Smalltalk-Entwickler durchaus nicht gezwungen wird, jeden Ausdruck als Anweisung zum Senden von Nachrichten an Objekte zu interpretieren. Beispielsweise kann man in Beispiel 2 die Schreibweise „x timesRepeat: [...]“ als syntaktischen Ausdruck einer zum Befehlsumfang des Smalltalk-Abwicklers gehörenden Anweisung auffassen, die diesen veranlaßt, die angebene Anweisungssequenz x-mal auszuführen. Ebenso liest man die Anweisungen in den Beispielen 1 und 3 besser ohne in Nachrichten und Objekten zu denken, wobei einem allerdings klar sein muß, daß die Prioritätsregel „Punkt- vor Strichrechnung“ bei zahlenarithmetischen Ausdrücken innerhalb von Smalltalk nicht gilt.

Basismethoden

Es wurde bereits gesagt, daß Smalltalk-Anweisungen, die keine einfachen Zuweisungen sind, das Senden von Nachrichten an Objekte auslösen. Auf diesen Objekten werden dann die durch den jeweiligen Message-Selector indentifizierten Methoden ausgeführt. Methoden bestehen im Normalfall wiederum aus Smalltalk-Anweisungen. Bei ihrer Ausführung wird also ebenfalls das Senden von Nachrichten an Objekte und damit die Ausführung weiterer Methoden veranlaßt. Man hat also die Vorstellung eines stark gefädelten Codes, bei dessen Abwicklung die Kontrolle ständig von Objekt zu Objekt weitergegeben wird, ohne daß irgend etwas geschieht – abgesehen von Zuweisungen an Attribute. Da dies nicht bis ins Unendliche fortgesetzt werden kann, muß es auch Objekte geben, in deren Methoden Basisoperationen unmittelbar ausgeführt werden, ohne dabei weitere Nachrichtensendungen auszulösen. Es handelt sich um die sogenannten *Basismethoden*, die nicht mehr in Smalltalk formuliert sind, sondern als einzige Anweisung den Aufruf einer C-Prozedur enthalten.

Man kann sich die virtuelle Smalltalkmaschine als einen durch einen mikroprogrammierten Steuerkreis realisierten Prozessor vorstellen, dessen Befehlsrepertoire diesen Basismethoden entspricht. Basismethoden enthalten als einzige Anweisung einen Sprung ins Mikroprogramm. Das die virtuelle Smalltalkmaschine beschreibende Programm ist komplett in C geschrieben. Die Mikroprogrammiersprache des Smalltalk-Prozessors ist also C. Es ist möglich, benutzerdefinierte Basismethoden einzuführen, indem man das Mikroprogramm um eigene Routinen erweitert. Diese Mikroprogrammerweiterungen realisiert man als C-Module, die nach Übersetzung zusammen mit einem Library-File, das die Beschreibung der ursprünglichen virtuellen Smalltalkmaschine enthält, zu einem neuen ausführbaren Programm gebunden werden. Durch Aufruf dieses Programms erhält man einen neuen, im Befehlsumfang erweiterten Smalltalk-Prozessor.

2.2.4 Entwicklungsumgebung

Smalltalk-Anweisungen können im Smalltalk-System an drei Orten auftauchen. Entweder stehen sie in sogenannten *Workspaces* oder – und das ist der weitaus häufigere Fall – innerhalb von Methoden- oder Klassenmethodenrümpfen. Workspaces stellen Notizzettel in Form eines Editor-Fensters dar, in denen man beliebige Folgen von Smalltalk-Anweisungen eingeben kann. Nach Markierung einer Anweisung oder einer Sequenz von Anweisungen mit der Maus kann man unter Nutzung der Menüsteuerung deren Ausführung veranlassen. Diese Anweisungen werden dann innerhalb des System-Globalkontextes abgewickelt. Es sind dort

alle Globalsymbole und temporär für die Dauer der Abwicklung definierten Variablen sichtbar. Innerhalb von Methoden gilt der jeweilige Methodenkontext, d. h. man hat zusätzlich Zugriff auf alle Attribute, Klassenattribute und Pool-Attribute des Objekts, auf dem die Methode ausgeführt wird, sowie auf methodenlokale, temporäre Variablen. Ebenso werden Anweisungen in Klassenmethoden innerhalb des jeweiligen Klassenmethodenkontextes ausgeführt. Dort hat man Zugriff auf alle Globalsymbole, auf die auch für alle Exemplare der Klasse sichtbaren Klassenattribute und Pool-Attribute, auf die Klassenobjektattribute und auf klassenmethodenlokale, temporäre Variablen (vgl. Tabelle 2.1 auf Seite 8).

Ein Smalltalk-Programmierer schreibt seinen Code nicht als Sequenz von Klassenbeschreibungen in eine oder mehrere Dateien, wie es in Entwicklungsumgebungen für andere Programmiersprachen üblich ist. Vielmehr editiert er immer nur einzelne Granulate von Klassenbeschreibungen und stößt damit die Erzeugung bzw. Änderung der zugehörigen Klassen- und Metaklassenobjekte im Objektspeicher an. Dabei hat er die in Bild 2.7 gezeigte Struktur vor Augen. Diejenigen Entitätstypen aus Bild 2.7, die editierbaren Granulaten entsprechen, sind grau unterlegt.

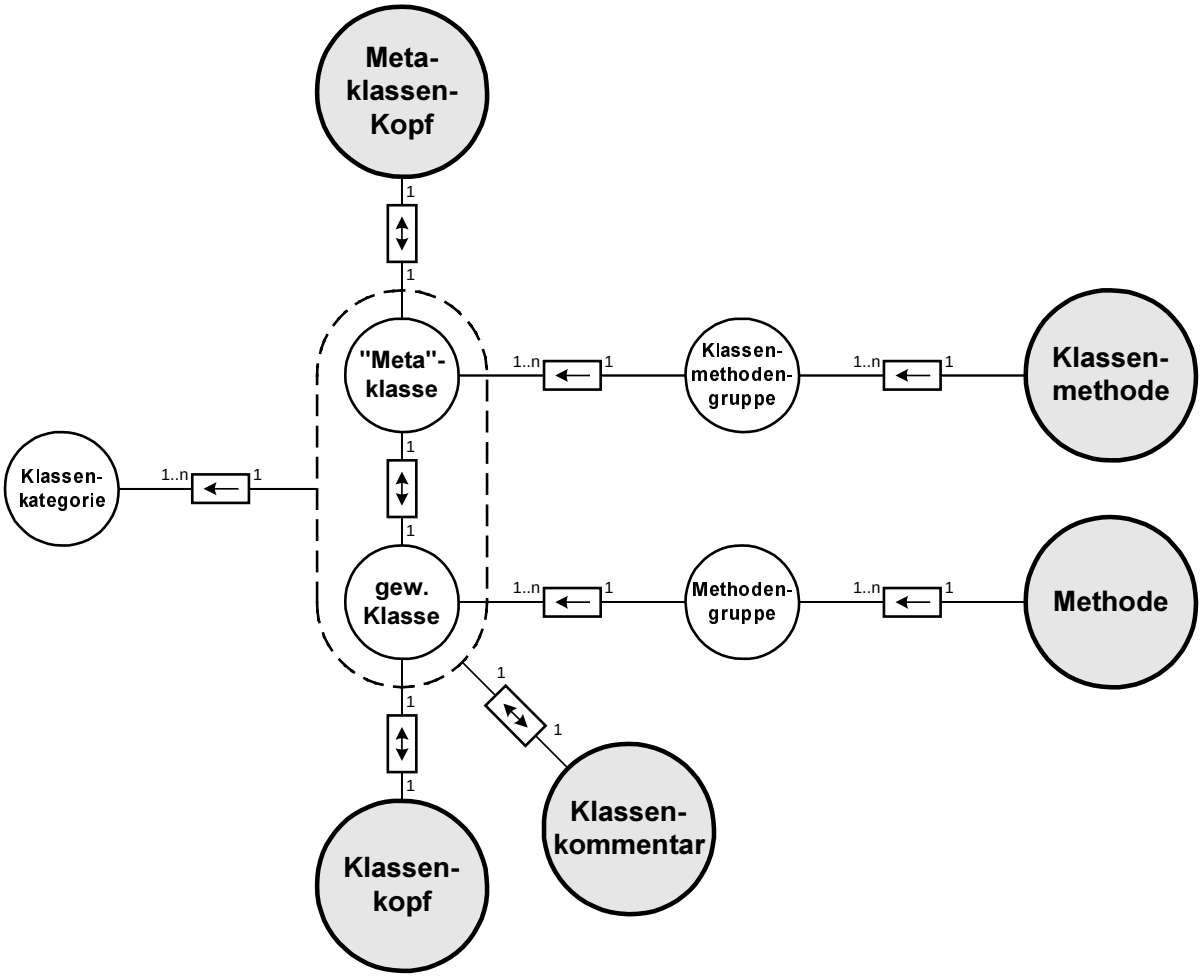


Bild 2.7 Struktur von Smalltalk-Systembeschreibungen

Zu jeder gewöhnlichen Klasse gehört eine „Meta“-Klasse. Während die gewöhnliche Klasse ihre Exemplare – gewöhnliche Objekte – beschreibt, stellt die Metaklasse den Bauplan des zugehörigen Klassenobjekts dar (vgl. Abschnitt 2.2.2). Zu jeder gewöhnlichen Klasse existiert

ein Klassenkopf-Formular. Hier trägt der Programmierer den Klassennamen, den Namen der Oberklasse, die Attributnamen, die Klassenattributnamen und Namen eventueller Attribut-Pools ein. Zu jeder „Meta“-Klasse existiert ebenfalls ein entsprechendes Metaklassenkopf-Formular. Wegen der 1:1-Beziehung zwischen Klassen und „Meta“-Klassen brauchen in diesen Metaklassenkopf-Formularen lediglich noch die Namen der Klassenobjektattribute eingetragen zu werden. Der Name einer „Meta“-Klasse ist bereits festgelegt durch den Klassennamen mit dem Zusatz ‘class’. Bild 2.8 zeigt die beiden Formulare, so wie sie der Programmierer zur Definition eines neuen Paares aus Klasse und „Meta“-Klasse vorfindet. Alle fett gedruckten Bezeichner sind Vorgaben und müssen beim Ausfüllen ersetzt werden. Der Metaklassenkopf ist erst nach Ausfüllen des Klassenkopf-Formulars editierbar. Der Klassenname ist dann bereits automatisch in den Metaklassenkopf übernommen worden.

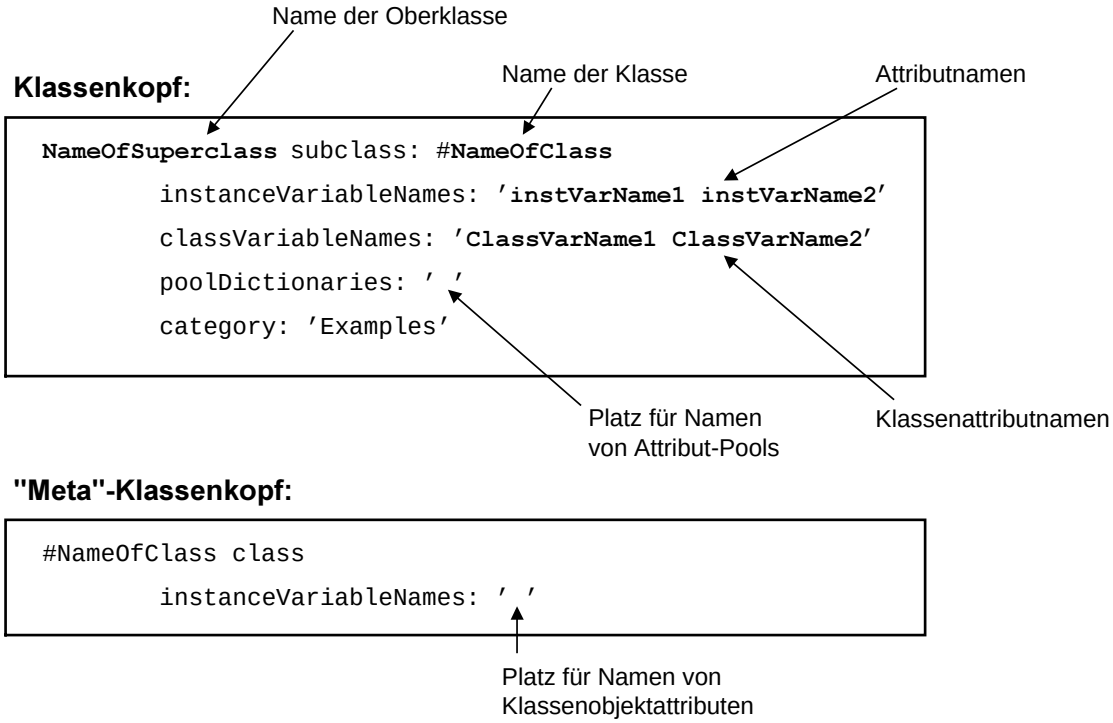


Bild 2.8 Klassenkopf und Metaklassenkopf

Paare aus Klassen und „Meta“-Klassen werden in sogenannten *Klassenkategorien* gruppiert. Jedes Paar gehört genau einer Klassenkategorie (Klassengruppe) an. Klassenkategorien tragen einen Namen, über den sie eindeutig indentifizierbar sind. Der Entwickler kann beliebige zusätzliche Kategorien einführen und Umgruppierungen vornehmen. Neue Klassen werden immer innerhalb einer solchen Klassengruppe definiert. Der Name der Gruppe, zu der eine Klasse gehört, taucht im Klassenkopf im Feld „category“ auf. Jedem Paar aus Klasse und Metaklasse ist zusätzlich ein eigenes Kommentarfeld (*Klassenkommentar*) zugeordnet, das separat editiert wird. Klassen beinhalten die Beschreibungen der Methoden, die auf ihren Exemplaren ausführbar sind. Auch diese Methoden lassen sich innerhalb einer Klasse nach semantischen Gesichtspunkten gruppieren. Jede Methode ist so einer *Methodengruppe* zugeordnet. Ebenso werden Klassenmethoden in *Klassenmethodengruppen* eingeordnet. Solche Gruppen können in beliebiger Anzahl eingeführt werden um die Menge der Methoden bzw. Klassenmethoden bezüglich einer Klasse in überschaubare Partitionen zu gliedern. Die Identifikation erfolgt über einen innerhalb der Klasse bzw. „Meta“-Klasse eindeutigen Gruppennamen. Methoden und Klassenmethoden werden jeweils einzeln editiert.

Alle erwähnten Gruppierungsmöglichkeiten (Klassenkategorien, Methodengruppen, Klassenmethodengruppen) dienen lediglich der semantischen Strukturierung von Systembeschreibungen und werden von der virtuellen Smalltalkmaschine nicht ausgewertet.

Es wurde bereits in Abschnitt 2.2.2 im Zusammenhang mit dem Smalltalk-Objektmodell auf die Existenz einer Konvention hingewiesen, die es erlaubt, Objektschnittstellen flexibler zu gestalten. Von der Sprachdefinition her sind alle Attribute privat und alle Methoden öffentlich. Möchte man nun gewisse Methoden nicht öffentlich zur Verfügung stellen, ordnet man sie in einer Methodengruppe mit dem Namen „private“ bzw. „private - <genauere Gruppenbezeichnung>“ ein. Möchte man bestimmte Methoden nur Exemplaren einer bestimmten Klasse oder einer Auswahl von Klassen zur Verfügung stellen, kann man das ebenso im Methodengruppennamen zum Ausdruck bringen, beispielsweise durch Bezeichnungen wie „services for instances of ...“ oder dergleichen. Damit ist der Aufruf durch Andere zwar nicht technisch unterbunden, aber immerhin hat der Entwickler die Möglichkeit, auf eine einheitliche Weise seine gewünschten Exportbeschränkungen zu formulieren.

2.2.5 Weitere Merkmale

2.2.5.1 Semantische Sprünge

Normalerweise entstehen neue Klassenobjekte und Metaklassenobjekte durch Benutzung der Entwicklungswerkzeuge, allgemeiner gesprochen: Die Erzeugung und der Zusammenbau von Klassenobjekten und Metaklassenobjekten wird im Normalfall durch einen Programmierer von außen veranlaßt, indem er mit gewissen Systemobjekten über grafische Benutzerschnittstellen kommuniziert. Genausogut kann die Erzeugung und der Zusammenbau von Klassen- und Metaklassenobjekte aber auch durch Akteure innerhalb des Smalltalk-Systems veranlaßt werden. Semantische Sprünge sind also möglich, indem man innerhalb einer laufenden Anwendung neue Metaklassenobjekte und Klassenobjekte baut und diese anschließend zur Erzeugung von Exemplaren ihrer Klasse auffordert. Der Smalltalk **System-Browser** – ein Entwicklungswerkzeug zur Edition von Klassen – ist selbst eine solche Anwendung, in der ständig semantischen Sprünge auftreten. Dort wird aus Daten wie Klassenkopf-Formularen und Methodentexten das durch Klassenobjekte und Metaklassenobjekte repräsentierte Programm.

2.2.5.2 Prozessormultiplex

Im Abschnitt 2.2.3 wurde bereits im Zusammenhang mit der Erläuterung der Basismethoden die Analogie zwischen der virtuellen Smalltalkmaschine und einem in Hardware realisierten Prozessor gezeigt. So wie ein moderner Prozessor im Zeitmultiplex mehrere Programme quasinebenläufig abwickeln kann, kann auch die virtuelle Smalltalk-Maschine im Zeitmultiplex mehrere Smalltalk-Programme nebeneinander ausführen. Man spricht von mehreren *Smalltalk-Prozessen* auf einer virtuellen Smalltalkmaschine. Smalltalk-Prozesse werden durch Aufruf der Methode ‘fork’ eines Block-Objekts gestartet. Das Blockobjekt enthält dabei die Anweisungen, die innerhalb des neuen Prozesses ausgeführt werden sollen.

Beispiel:
```
[ 1 to: 100 do:
    [ Transcript show: 'Hello World !'. ]
] fork.
```

Die gezeigte 'fork'-Anweisung veranlaßt den Start eines separaten Smalltalk-Prozesses, innerhalb dessen einhundertmal die Zeichenkette 'Hello World !' ins Standardausgabefenster geschrieben wird. Für solche Smalltalk-Prozesse sind Laufprioritäten aus einem Bereich von 1 bis 100 vergebbar. Es existiert ein eigener Scheduling-Mechanismus, bei dem für jede Prioritätsstufe eine eigene Warteschlange vorgesehen ist, in der auf Fortsetzung wartende Prozesse mit dieser Laufpriorität aufgereiht werden. Prozesse höherer Laufpriorität werden grundsätzlich vorrangig ausgeführt, d. h. ein Prozeß mit niedrigerer Laufpriorität wird angehalten, sobald der Start eines Prozesses höherer Laufpriorität veranlaßt wurde oder ein Prozeß höherer Laufpriorität auf Wiederaufnahme wartet. Innerhalb einer Prioritätsstufe ist der Scheduling-Mechanismus nicht preemptiv. In einem aktiven Prozeß muß der Prozessor explizit an den nächsten in der zu seiner Prioritätsstufe gehörenden Warteschlange abgegeben werden. Als Folge davon wird der dann nicht mehr aktive Prozeß ans Ende dieser Warteschlange geschoben.

Zur Synchronisation von Prozessen werden Semaphoren zur Verfügung gestellt. Exemplare der Klasse **Semaphore** stellen Verwalter von Semaphorplätzen dar. Möchte man in einem Prozeß auf die Belegung des verwalteten Semaphorplatzes warten, drückt man dies durch Aufruf der Basismethode 'wait' des zugehörigen Semaphorverwalters aus. Liegt dort bereits eine Marke, wird sie entfernt, und der Prozeß, innerhalb dessen die Marke angefordert wurde, unterbrechungsfrei fortgesetzt. Liegt dort keine Marke, wird der Prozeß angehalten und der Smalltalk-Prozessor dem nächsten Prozeß in der Warteschlange zugeteilt. Die Belegung eines Semaphorplatzes geschieht durch Aufruf der Basismethode 'signal' seines zugehörigen Verwalters. Falls Prozesse existieren, in denen auf diese Belegung gewartet wurde, wird der davon am längsten wartende Prozeß wieder laufbereit gemacht und die Marke entfernt.

2.2.5.3 Fehlerbehandlungsmechanismus

Smalltalk stellt einen flexiblen Mechanismus zur Behandlung von Ausnahmesituationen zur Verfügung, basierend auf Ausnahmemeldungen und Ausnahmebehandlern. Eine zentrale Rolle spielen dabei die sogenannten *Fehlerklassenverwalter*. Ein Fehlerklassenverwalter ist zuständig für eine Klasse möglicher Ausnahmesituationen – *Fehlerklasse* genannt. Jede solche Fehlerklasse hat eine Position innerhalb einer Hierarchie, an deren Spitze die alle möglichen Ausnahmesituationen umfassende allgemeinste Fehlerklasse steht. In Bild 2.9 ist oben ein Ausschnitt aus dieser Hierarchie dargestellt. Im Unterschied zu Smalltalk-Klassen werden durch Fehlerklassen nicht Objekte, sondern Ausnahmesituationen klassifiziert. Die Objektklassifikation durch Smalltalk-Klassen und die Klassifikationen von Ausnahmesituationen durch Fehlerklassen haben nichts miteinander zu tun. Die Fehlerklassenhierarchie ist demnach auch völlig unabhängig von der Smalltalk-Klassenhierarchie. Für jede Fehlerklasse existiert ein zuständiger Fehlerklassenverwalter als Objekt im Objektspeicher. Fehlerklassenverwalter sind Exemplare der Klasse **Signal**, die bereits im Initial-Objektspeicher vorhanden sind oder aber zur Bauzeit eines Anwendungssystems erzeugt werden. Per Konvention wird jeder Fehlerklassenverwalter an ein Klassenattribut desjenigen Klassenobjekts gebunden, zu dem die von ihm verwaltete Fehlerklasse semantisch am besten „paßt" und durch eine nach dem Klas-

senattribut benannte Klassenmethode öffentlich zugänglich gemacht. Da alle Klassenobjekte über ihre Klassennamen global erreichbar sind (vgl. Abschnitt 2.2.1), sind damit auch alle Fehlerklassenverwalter global zugänglich. So ist beispielsweise der Fehlerklassenverwalter bezüglich der Fehlerklasse, die mögliche Ausnahmesituationen während einer zahlenarithmetischen Operation umfaßt, ein Klassenattribut der Klasse **ArithmeticValue** mit dem Namen **ErrorSignal**. Der Zugriff erfolgt über deren Klassenmethode ‘errorSignal’. Unten im Bild 2.9 sind die Ortsbezeichner (Klassenname und Klassenattributname) der für die darüber gezeigten Fehlerklassen zuständigen Fehlerklassenverwalter angegeben.

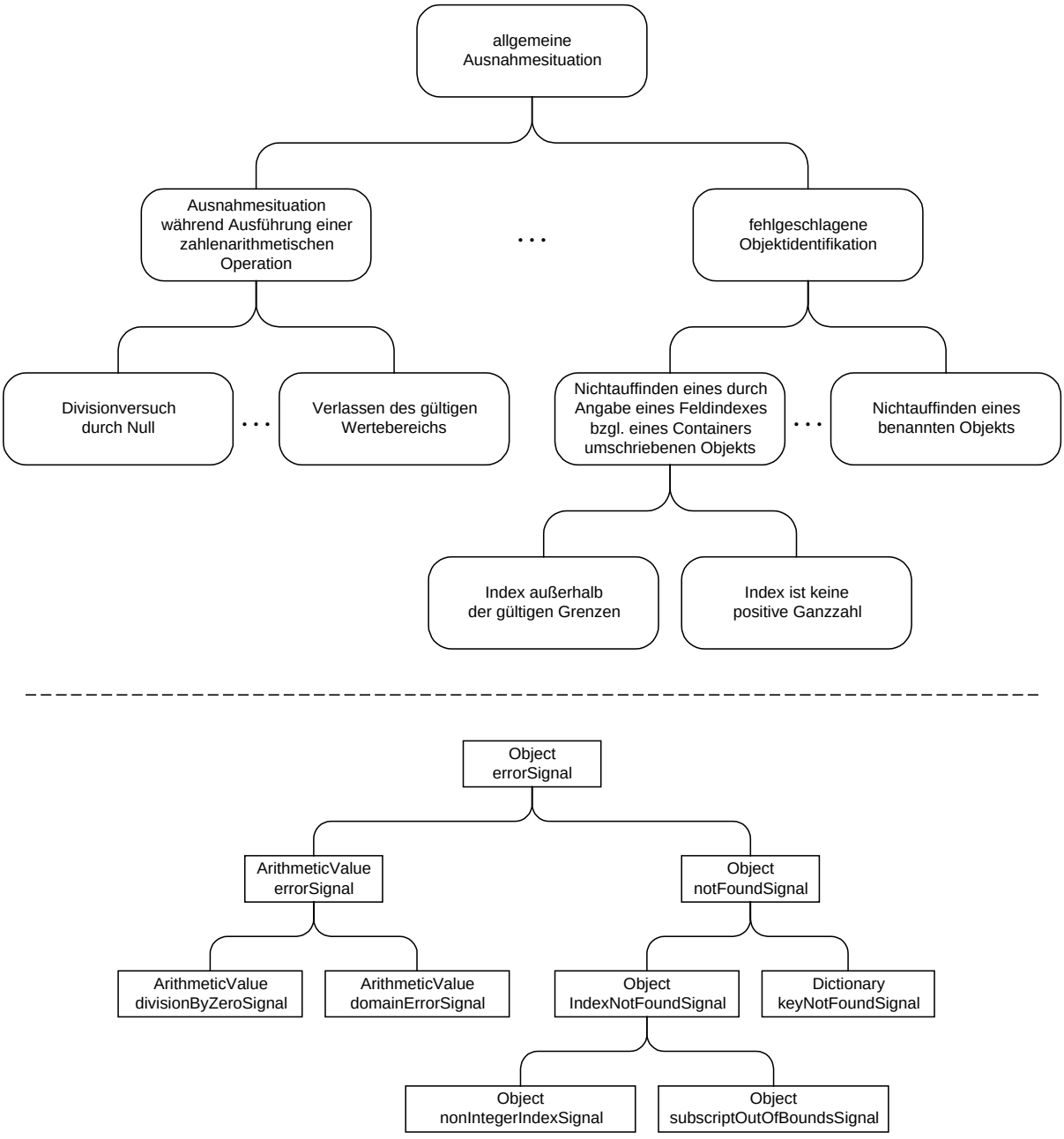


Bild 2.9 Fehlerklassenhierarchie in Smalltalk (Ausschnitt)

Fehlerklassenverwalter können zur Anwendungslaufzeit in zwei Rollen auftreten, nämlich einerseits als *Ausnahmemeldungsauslöser* und andererseits als *Überwachungseinrichtung*. In der ersten Rolle sind es die Instanzen, die bei Erkennung einer Ausnahmesituation durch

Aufruf ihrer Methode ‘raise’ zum Auslösen einer Ausnahmemeldung veranlaßt werden. Anweisungen zum Auslösen von Ausnahmemeldungen finden sich zum Beispiel innerhalb von Methoden derjenigen Systemklassen, die Zahlentypen repräsentieren. Beispielsweise wird dort in sämtlichen die Division betreffenden Methoden, falls der Divisor Null ist, der über die Klassenmethode ‘divisionByZeroSignal’ der Klasse **ArithmeticValue** zugängliche Fehlerklassenverwalter zur Auslösung einer Ausnahmemeldung beauftragt. Das Auslösen von Ausnahmemeldungen kann innerhalb beliebiger Methoden – auch in Methoden von Benutzerklassen und auch innerhalb von Basismethoden – veranlaßt werden. Eine Ausnahmemeldung äußert sich darin, daß durch den zuständigen Fehlerklassenverwalter ein *Ausnahmemeldungsobjekt* als Exemplar der Klasse **Exception** kreiert wird. Ist keine benutzerdefinierte Ausnahmebehandlung vorgesehen worden, tritt das Ausnahmemeldungsobjekt unmittelbar nach seiner Erzeugung als Akteur auf und öffnet ein Meldungsfenster, in dem dem Benutzer die Ausnahmesituation mitgeteilt wird. Der Benutzer hat dann unter Umständen noch die Wahl zwischen Fortsetzung und Abbruch der Abwicklung; in allen schwerwiegenden Ausnahmesituationen kann er jedoch nur noch den Abbruch der Abwicklung bestätigen. Durch Anstoß des Ausnahmemeldungsakteurs wird zusätzlich ein Abwicklungskontextwechsel ausgelöst. Der Abwicklungskontext vor Auftreten der Ausnahmesituation wird gerettet und die Aktionen des Ausnahmemeldungsakteurs laufen also innerhalb eines eigenen Abwicklungskontextes ab. So ist es möglich, bei weniger schwerwiegenden Ausnahmesituationen die Abwicklung auf Wunsch im alten Abwicklungskontext hinter der Anweisung fortzusetzen, die die Ausnahmesituation verursacht hat.

In der zweiten Rolle als Überwachungseinrichtung sind Fehlerklassenverwalter zuständig für das Umleiten von Ausnahmemeldungen an benutzerdefinierte *Ausnahmebehandlungsakteure*. Überwacht werden kann die Abwicklung jeder durch ein Block-Objekt repräsentierten Anweisungssequenz. Während der Abwicklung dieser Anweisungssequenz werden alle diejenigen Ausnahmemeldungen umgeleitet, die durch den überwachenden Fehlerklassenverwalter selbst oder durch Fehlerklassenverwalter, die in der Hierarchie unter ihm stehen, ausgelöst werden. ‘Object errorSignal’ leitet also in der Überwacherrolle alle möglichen Ausnahmemeldungen um, ‘ArithmeticValue errorSignal’ nur diejenigen, die Ausnahmesituationen während zahlenarithmetischen Operationen betreffen. Das Verhalten des jeweiligen Ausnahmebehandlungsakteurs wird als Anweisungssequenz formuliert und ebenfalls durch ein Blockobjekt repräsentiert. Die überwachte Abwicklung einer Anweisungssequenz wird nun veranlaßt durch Übergabe der zwei genannten Block-Objekte an einen Fehlerklassenverwalter. Dies geschieht via Aufruf der Methode ‘handle:do:’ eines solchen Fehlerklassenverwalters:

<zuständiges Signal-Exemplar>
handle: *<Ausnahmebehandler-Block (Handler-Block)>*
do: *<überwacht auszuführender Block (Do-Block)>* **.**

Nach Aufruf der handle:do:-Methode eines Fehlerklassenverwalters beginnt unmittelbar die Abwicklung des Do-Blocks. Tritt währenddessen keine Ausnahmesituation auf, für die der Fehlerklassenverwalter selbst oder einer der ihm untergeordneten „Kollegen“ zuständig ist, passiert nichts weiter. Nach Beendigung des Do-Blocks werden die nach dem handle:do:-Aufruf folgenden Anweisungen ausgeführt. Tritt jedoch eine solche Ausnahmesituation ein, wird die Abwicklung des Do-Blocks abgebrochen, der Abwicklungskontext gerettet, und die Ausführung des Handler-Blocks veranlaßt. Wie schon erwähnt, wird das Auftreten von Aus-

nahmesituationen durch Erzeugen eines Ausnahmemeldungsobjekts durch den zuständigen Fehlerklassenverwalter mitgeteilt. Dieses Ausnahmemeldungsobjekt (Exemplar der Klasse Exception) wird nun aber dem Ausnahmebehandlungsakteur übergeben. Im Handler-Block wird dafür eine Argumentvariable vorgesehen, die vor Ausführung mit einem Verweis auf das Ausnahmemeldungsobjekt belegt wird. Den Attributen des Ausnahmemeldungsobjekts kann der Ausnahmebehandlungsakteur dann Näheres über die Ausnahmebedingungen entnehmen und darauf basierend gegebenenfalls Rettungs- oder Aufräumaktionen veranlassen. Der Handler-Block kann auf mehrere Arten verlassen werden. Dies geschieht immer durch Aufruf einer Methode des Ausnahmemeldungsobjekts. Es kommen dabei folgende Methoden in Frage:

return Es wird die ordnungsgemäße Beendigung des Do-Blocks vorgetäuscht.

reject Die Ausnahmebehandlung wird verweigert und das Ausnahmemeldungsobjekt zurückgewiesen.

proceed Der alte Abwicklungskontext wird wiederhergestellt und die Do-Block-Abwicklung nach der Anweisung fortgesetzt, die die Ausnahmesituation verursacht hat.

restart Der Do-Block wird nochmals von Beginn an ausgeführt (Wiederversuch).

Unter Nutzung dieser Sprachmittel läßt sich der rescue/retry-Mechanismus der Programmiersprache Eiffel (vgl. [Meyer 90, Kap. 7]) nachbauen. Ein Beispiel dazu zeigt Bild 2.10. Dort ist links eine Eiffel-Methode zur Invertierung einer Fließkommazahl dargestellt (entnommen aus [Meyer 90, S. 167f.]). Durch Nutzung des rescue/retry-Mechanismus wird erreicht, daß sie auch für sehr kleine Argumente ein Ergebnis liefert. Ausnahmesituationen, die durch Verlassen des darstellbaren Ergebnis-Wertebereichs oder einen Divisionsversuch durch Null auftreten können, werden sicher gemeistert. Rechts ist dieselbe Methode in Smalltalk formuliert gezeigt. Nach Methodenname, Argumentbezeichnung und Kommentar folgt die Deklaration methodenlokaler Variablen zwischen senkrechten Strichen. Die in Eiffel automatisch durchgeführte Initialisierung der Variable ‘division_versucht’ muß in Smalltalk explizit vorgenommen werden. Anschließend folgt die Beauftragung des Fehlerklassenverwalters ‘ArithmeticValue errorSignal’ zur überwachten Ausführung der Anweisungen innerhalb des Do-Blocks. Die letzte Anweisung „^result“ bewirkt die Rückgabe des Ergebnisses an den Aufrufer, die in Eiffel implizit geschieht. Tritt bei Abwicklung dieses Do-Blocks eine der genannten Ausnahmesituationen auf, wird der im Methodentext darüberliegende Handler-Block ausgeführt. Vorher wird das Handler-Blockargument ‘exception’ mit einem Verweis auf das vom entsprechenden Fehlerklassenverwalter erzeugte Ausnahmemeldungsobjekt belegt. Der Aufruf dessen Methode ‘restart’ entspricht der retry-Anweisung innerhalb der rescue-Klausel in Eiffel. Ferner sieht man deutlich, daß sich Handler-Block und Do-Block aus Smalltalk auf rescue-Klausel und do-Klausel in Eiffel abbilden lassen.

Der Benutzer kann zusätzlich zu den bereits vorhandenen Fehlerklassen, von denen nur einige in Bild 2.9 gezeigt sind, eigene anwendungsspezifische Fehlerklassen einführen. Neue Fehlerklassen können an beliebiger Stelle im Hierarchiebaum als neue Blätter hinzugefügt werden. Zu jeder neuen Fehlerklasse muß ein zuständiger Fehlerklassenverwalter im Objektspeicher erzeugt und benannt werden. Dies geschieht, indem man einen vorhandenen Fehlerklassenverwalter beauftragt, einen neuen Fehlerklassenverwalter zu kreieren und unter sich im

Eiffel

```
quasi_Inverse( x : REAL ) : REAL is
     -- 1/x wenn möglich; sonst 0

local
     division_versucht: BOOLEAN

do
     if not division_versucht then
         Result := 1/x
     else
         Result := 0
     end

rescue
     division_versucht := true;
     retry

end -- quasi_Inverse
```

Smalltalk

```
quasiInverse: x
"1/x wenn möglich, sonst 0"

| divisionVersucht result |
divisionVersucht := false.

ArithmeticValue errorSignal
     handle: [ :exception |
         divisionVersucht := true.
         exception restart.
     ]
     do: [
         divisionVersucht
             ifTrue: [ result := 0. ]
             ifFalse: [ result := 1/x. ].
         ^result.
     ].
```

Bild 2.10 Nachahmung des Eiffel-rescue/retry-Mechanismus in Smalltalk

Hierarchiebaum einzuhängen. Anschließend bindet man das so erzeugte Objekt gemäß der erwähnten Konvention an ein Klassenattribut eines passenden Klassenobjekts und macht es über eine entsprechend benannte Klassenmethode öffentlich zugänglich.

2.2.5.4 Grafische Benutzerschnittstellen

Ein wichtiges Merkmal von Smalltalk ist die Möglichkeit, Anwendungen mit grafischen Benutzerschnittstellen zu entwerfen. Dies wird zum einen durch eine Vielzahl von Systemklassen unterstützt, deren Exemplare für Elemente solcher Oberflächen zuständig sind oder das Erscheinen der Elemente auf dem Bildschirm bewirken. Zum anderen existieren spezielle Entwicklungswerkzeuge zum bequemen Design grafischer Benutzeroberflächen. Beim Entwurf von Anwendungen mit grafischen Benutzerschnittstellen ist eine strikte Trennung der beteiligten Objekte in drei Gruppen vorgesehen:

- Objekte mit darzustellenden Attributen und/oder infolge von Benutzeraktionen auszuführenden Methoden (*Model*-Objekte)
- die Darstellung bewerkstelligende Objekte (*View*-Objekte)
- Objekte, mit Hilfe derer der Benutzer über die grafische Schnittstelle Einfluß auf das Anwendungssystem nehmen kann (*Control*-Objekte).

Bild 2.11 zeigt die vorgesehene Kommunikationsstruktur zwischen diesen Objektgruppen. Model-Objekte tragen weder Information über die Darstellungsform der darzustellenden Attribute noch über die Art der Steuerungselemente auf dem Bildschirm, über die sie beeinflußbar sind. Dieses Konzept der Trennung von Zuständigkeiten trägt auch die Bezeichnung *Model-View-Controller-Paradigma* (MVC-Paradigma [Bücker 95]). Für jede der drei Gruppen stehen eine Reihe von Systemklassen zur Verfügung, deren Exemplare die in ihrer Gruppe benötigten Merkmale besitzen. Viele dieser Klassen sind als Oberklassen für benutzerdefinierte Klassen vorgesehen. Eine sehr ausführliche Dokumentation der Möglichkeiten,

die VISUALWORKS hier anbietet, inklusive der sich dahinter verbergenden Konzepte findet man in [Howard 95]. Im Rahmen der vorliegenden Arbeit wird dieses Gebiet jedoch nicht näher behandelt.

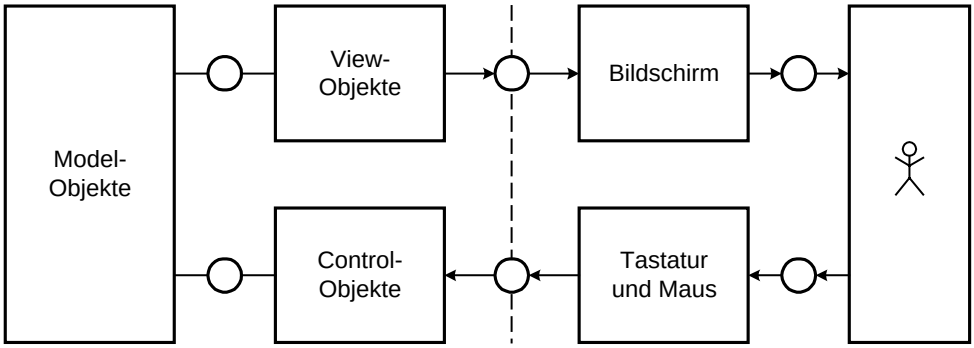


Bild 2.11 Model-, View- und Control-Objekte

2.3 Eignung im Laboreinsatz

Smalltalk Systeme bestehen, wie anhand von Bild 2.1 bereits erläutert wurde, aus einem interpretierenden Abwickler, der auf einem virtuellen Objektspeicher arbeitet. Dieses *Interpreterkonzept* bringt viele Vorteile mit sich. Es wurde gezeigt, daß sich dadurch ohne zusätzlichen Aufwand plattformunabhängige Anwendungen realisieren lassen. Ferner ist es möglich, Änderungen an Anwendungssystemen zu deren Laufzeit vorzunehmen und semantische Sprünge zu veranlassen. Zudem läßt sich durch Definition benutzerdefinierter Basismethoden der Befehlssatz des Abwicklers erweitern.

Smalltalk-Anwendungssysteme werden gebaut, in dem man einem initial vorbelegten Objektspeicher durch Nutzung von Entwicklungswerkzeugen neue Objekte zufügt oder bestehende Objekte ändert. Dieses Konzept der *Systemrealisierung durch Erweiterung bzw. Änderung eines bestehenden Universalsystems* ermöglicht die Nutzung vieler vorgefertigter Komponenten, die bei Bedarf den eigenen Wünschen angepaßt werden können. Nicht benötigte Komponenten können nach Entwicklungsschluß entfernt werden (Ausschneiden des Anwendungs-Image aus dem Entwicklungs-Image, vgl. Abschnitt 2.2.1).

Neben den vielen Vorteilen bringen beide Konzepte aber auch Nachteile mit sich. Der Einsatz von Smalltalk führt – verglichen mit übersetzten Sprachen wie Eiffel und C++ – zu Performance-Einbußen und erhöhtem Speicherplatzbedarf. Smalltalk ist gut geeignet zur Realisierung von Endbenutzeranwendungen, in denen keine Zeitrelevanz gegeben ist. Smalltalk ist sicherlich nicht die Programmiersprache der Wahl zur Realzeitprogrammierung von schnell reagierenden Steuerungssystemen.

Die ansonsten universelle Einsetzbarkeit wird noch gesteigert durch die Verfügbarkeit diverser Schnittstellen zu anderen Software-Systemen. Beispielsweise werden zu der im Rahmen dieser Arbeit verwendeten Smalltalk-Umgebung VISUALWORKS von Parcplace/Digitalk Kom-

ponenten zum Anschluß einer Reihe von relationalen und objektorientierten Datenbanksystemen angeboten – teilweise von Parcplace/Digitalk selbst, teilweise von den jeweiligen Datenbank-Herstellern.[3] Daneben existiert eine komfortable Schnittstelle zur Programmiersprache C. So ist es möglich, Teile eines Anwendungssystems in C zu programmieren oder bereits in C programmierte Universalkomponenten für Smalltalk verfügbar zu machen.

Im Rahmen der vorliegenden Arbeit ist eine universelle Experimentierumgebung mit Laborcharakter aufgebaut worden. Dabei hat sich gezeigt, daß gerade für dieses Anwendungsfeld Smalltalk-Systeme bestens geeignet sind.

[3] Erwähnt werden muß in diesem Zusammenhang die speziell auf Smalltalk zugeschnittene Objektdatenbank **GemStone** vom gleichnamigen Hersteller aus Beverton, Oregon (USA). Dieses Produkt wird zusammen mit anderen Objektdatenbanken in [Gröne 96] näher untersucht.

3 Entwurf einer Architektur für objektorientierte 3-Ebenen-Client-Server-Anwendungen und deren Implementierung in Smalltalk

Dieses Kapitel dokumentiert den Entwurf einer Softwarearchitektur für heterogene 3-Ebenen-Client-Server-Anwendungen, so wie sie im Software-Technologie-Labor realisiert werden können sollen. Eine Grundanforderung an diese Architektur besteht darin, daß es möglich sein muß, die einzelnen Komponenten des Anwendungssystems unter Nutzung unterschiedlicher Programmiersprachen und Entwicklungswerkzeuge zu bauen. Im Rahmen dieser Arbeit wird gezeigt, wie man die Komponenten in Smalltalk realisieren kann. Trotzdem sind dargestellten Konzepte auch nutzbar, wenn man andere objektorientierte Entwicklungsumgebungen gebraucht. Die Architektur sieht eine Reihe von Universalkomponenten vor, die in jedem konkreten Anwendungssystem gleich aussehen. Eine dieser Universalkomponenten ist ein Softwarebussystem, das als fertiges Produkt vom Hersteller zur Verfügung gestellt wurde. Alle weiteren Universalkomponenten sind auf diesem Bussystem aufbauende Eigenentwicklungen, deren Implementierung in Smalltalk ebenfalls nachfolgend dokumentiert wird. Ziel dieser Entwicklungsarbeit war es, eine universell nutzbare Experimentierplattform zu schaffen, auf der unkompliziert Anwendungsprototypen in Smalltalk realisiert werden können.

Abschnitt 3.1 führt zunächst die Anforderungen an die Architektur und die gesteckten Ziele auf. Anschließend werden in Abschnitt 3.2 vier aufeinander aufbauende Teilbereiche des Architekturentwurfs abgegrenzt. Die Abschnitte 3.3 bis 3.6 enthalten schließlich die eigentliche Dokumentation der Entwurfs- und Entwicklungsarbeiten. Dabei wird der Leser Schritt für Schritt durch die vier zuvor abgegrenzten Schichten geführt.

3.1 Anforderungen und Ziele

3.1.1 Technische Anforderungen

3.1.1.1 Drei Ebenen

Nach dem Vorbild des SAP R/3-Systems soll eine 3-Ebenen-Architektur realisiert werden, wie sie Bild 3.1 zeigt. Ganz oben liegt die Präsentationsebene, in der Mitte die Applikationsebene und unten die Datenbank-Ebene.

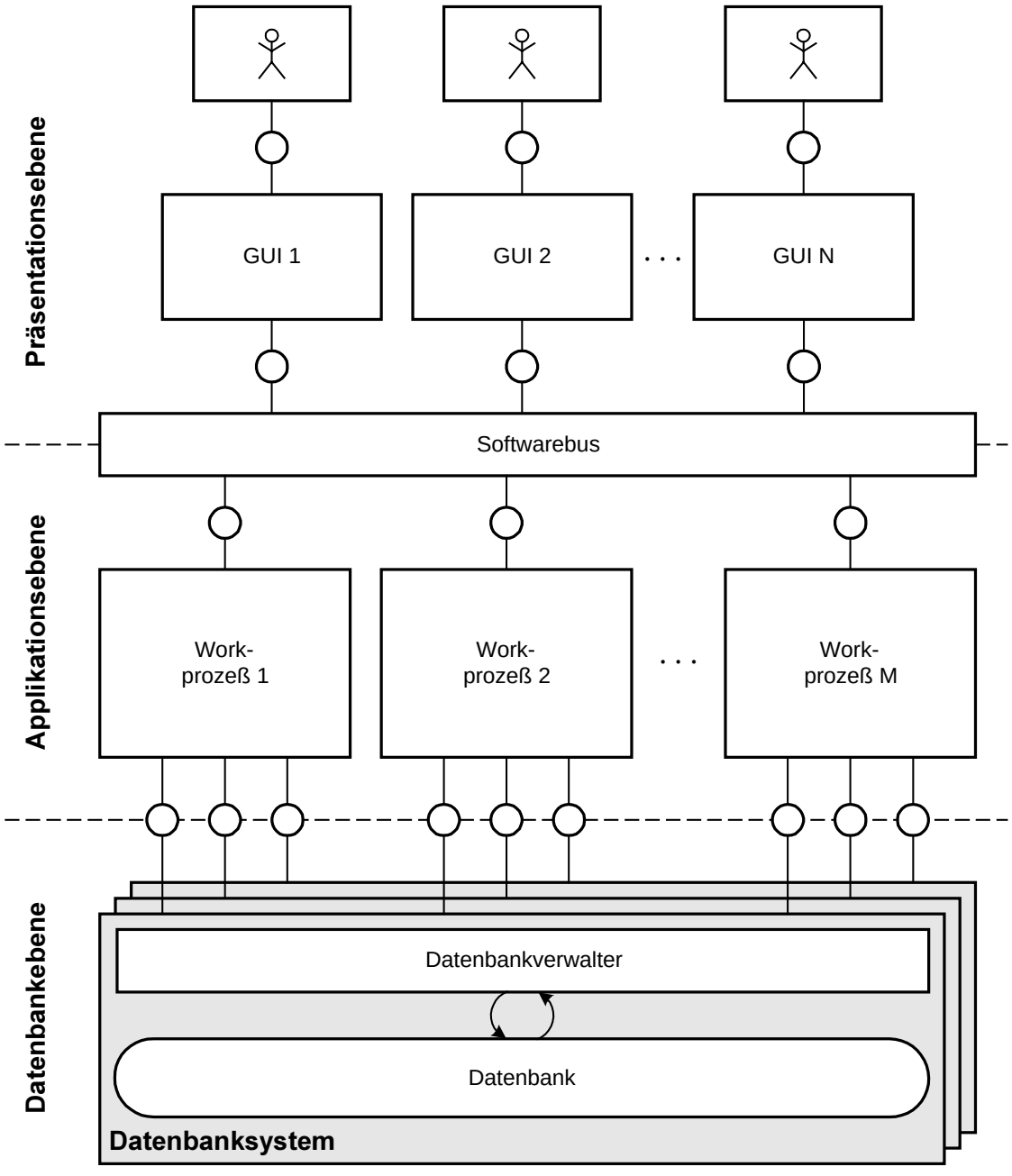


Bild 3.1 Vorgesehener technischer Aufbau von Laboranwendungen

Präsentationsebene: GUIs

In der Präsentationsebene befinden sich die sogenannten *GUIs* (GUI = "Graphical User Interface", dt.: „grafische Benutzerschnittstelle"). Es handelt sich dabei um diejenigen Akteure des Anwendungssystems, mit denen der Anwendungsbenutzer unmittelbar via Bildschirm, Maus und Tastatur kommuniziert. Sie sind zuständig für die Darstellung von Ausgabe-Information und die Aufnahme und Weiterleitung von Eingabe-Information. Die GUIs kommunizieren des weiteren mit den Workprozessen der Applikationsebene. Eine Kommunikation zwischen den einzelnen GUIs innerhalb der Präsentationsebene ist nicht vorgesehen. Ein GUI befindet sich immer auf dem Rechner, den der jeweilige GUI-Benutzer bedient, wobei ein Benutzer gleichzeitig mehrere GUIs auf ein und demselben Rechner bedienen kann. Eine Anwendung sieht im allgemeinen mehrere Typen von GUIs vor, die verschiedenen Benutzerrollen zugeordnet sind. Beispielsweise kann es unterschiedliche GUI-Typen für Sachbearbeiter mit unterschiedlichen Aufgabenbereichen geben, und einen weiteren GUI-Typ für die Systemadministration.

Applikationsebene: Workprozesse

In der Applikationsebene befinden sich die sogenannten *Workprozesse*[4,5]. Workprozesse treten im Anwendungssystem als Dienstleister für anwendungsspezifische Dienste auf. Jeder Workprozeß bietet eine feste Menge von Dienstleistungen an. Wie bei den GUIs gibt es auch bei den Workprozessen unterschiedliche Workprozeßtypen. Workprozesse verschiedenen Typs unterscheiden sich in Art und Menge der angebotenen Dienstleistungen. Die Kunden von Workprozessen sind GUIs oder andere Workprozesse. Damit alle GUIs und Workprozesse sämtliche Dienstleistungen aller Workprozesse nutzen können, ist ein Buskommunikationssystem (Softwarebus) vorgesehen worden, das alle Komponenten der Präsentationsebene und Applikationsebene miteinander verbindet.

Datenbankebene

In der Datenbankebene liegen schließlich verschiedene *Datenbanksysteme*, die von den Workprozessen genutzt werden. Hier können sowohl Objektdatenbanken, als auch relationale Datenbanken auftauchen. Während die Anzahl der GUIs und Workprozesse zur Anwendungslaufzeit durchaus variieren darf, sollen Anzahl und Ort der Datenbanksysteme für jede Anwendung festliegen. Jeder Workprozeß soll potentiell mit jedem Datenbanksystem kommunizieren dürfen.

Es ist vorgesehen, daß jeder der am Anwendungssystem beteiligten Komponenten ein eigener Rechner zugewiesen werden kann. (Ausgenommen davon ist natürlich der Softwarebus. Er besteht intern aus mehreren Komponenten, die auf die Rechner innerhalb der Präsentations- und Applikationsebene verteilt sind.) Es können sich aber auch durchaus mehrere Komponenten auf ein und demselben Rechner befinden.

---

[4] Für Kenner des SAP R/3-Systems sei angemerkt, daß sich Aufgaben und Aufbau von Workprozessen innerhalb der hier vorgestellten Architektur stark von denen der R/3-Workprozesse unterscheiden.

[5] Der Begriff „Workprozeß" dient hier und im folgenden zur Bezeichnung der aktionsfähigen Komponenten der Applikationsebene. Workprozesse sind demnach Akteure.

3.1.1.2 Kommunikation über einen Softwarebus

Wie bereits erwähnt, soll die Kommunikation zwischen Präsentations- und Applikationsebene und die Kommunikation innerhalb der Applikationsebene über einen Softwarebus erfolgen. Es handelt sich dabei um einen Publish-Subscribe-Bus, der ganz ähnlich wie ein Hardwarebus funktioniert. Auf den Bus gegebene Informationen können potentiell von jedem Busteilnehmer empfangen werden. Über seine Empfangsbereitschaft bezüglich bestimmter Informationen entscheidet der jeweilige Busteilnehmer selbst. Busteilnehmer können zur Anwendungslaufzeit „dazugesteckt“ oder weggenommen werden. Das eingesetzte Produkt – der **RENDEZVOUS SOFTWARE BUS** des Herstellers **TIBCO** aus Palo Alto, Kalifornien, USA – wird in Abschnitt 3.3 vorgestellt.

3.1.1.3 Heterogenität

Die Architektur soll massiv heterogene Anwendungssysteme erlauben. Es sollen verschiedene Programmiermodelle koexistieren dürfen. Insbesondere muß es möglich sein, einige Workprozeßtypen rein prozedural und andere objektorientiert zu implementieren. Es ist ferner vorgesehen, daß sämtliche GUI-Typen und Workprozeßtypen in unterschiedlichen Programmiersprachen realisiert werden können. So ist beispielsweise auf Präsentationsebene der Einsatz von Smalltalk und Java für unterschiedliche GUI-Typen und auf Applikationsebene der Einsatz von C, C++, Eiffel und Smalltalk für unterschiedliche Workprozeßtypen denkbar. Ferner sollen innerhalb einer Anwendung mehrere unterschiedliche Datenbanksysteme mitwirken dürfen. Man kann sich beispielsweise vorstellen, die abzulegenden Datentypen aufzuteilen: Datentypen, bei denen das Transaktionsvolumen hoch und der Vernetzungsgrad der abgelegten Daten gering ist, werden einer relationalen Datenbank zugeordnet; Daten mit geringerem Transaktionsvolumen und höherem Vernetzungsgrad werden dagegen in einer Objektdatenbank abgelegt. Die Forderung, massiv heterogene Systeme bauen zu können, bezieht sich auch auf die einsetzbaren Hardwareplattformen. Es soll möglich sein, verschiedene Rechnertypen innerhalb des gleichen Anwendungssystems zu mischen.

Zur Erreichung dieser Ziele müssen also Schnittstellen geschaffen werden, bei denen die Realisierungsform der Kommunikationspartner keine Rolle mehr spielt. Gerade hier lohnt sich der Einsatz des oben genannten Bussystems von TIBCO, da es diese Forderungen in weiten Teilen bereits erfüllt.

3.1.2 Geforderte Anwendungspotentiale

Konkurrenz und Skalierbarkeit

Ein Anwendungssystem soll durch viele Anwendungsbenutzer gleichzeitig bedient werden können. Die Anzahl der GUIs eines Anwendungssystems soll dabei variieren dürfen. Insbesondere soll das Vorhandensein mehrerer GUIs desselben Typs erlaubt sein. Es muß also berücksichtigt werden, daß eine Dienstleistung auf Applikationsebene von mehreren GUIs oder Workprozessen gleichzeitig in Anspruch genommen werden kann. Daher sollen sich

Workprozesse in ihren Aufgabenbereichen überlappen dürfen. Dies kann sich einerseits darin äußern, das zwei Workprozesse unterschiedlichen Typs teilweise identische Dienstleistungen anbieten, oder andererseits mehrere Workprozesse desselben Typs nebeneinander agieren. Durch Erfüllung dieser Anforderung gelangt man zu skalierbaren Anwendungssystemen, bei denen abhängig von der jeweiligen Auslastung neue Workprozesse „eingeklinkt“ oder nicht mehr benötigte weggenommen werden können. Stellt sich beispielsweise zur Laufzeit einer Anwendung heraus, daß durch häufige Benutzung einer bestimmten Dienstleistung ein Engpaß auf Applikationsebene droht, erzeugt man einen weiteren Workprozeß, der den gefragten Dienst ebenfalls leisten kann. Alle Workprozesse teilen sich gemeinsam die Datenbanken der Datenbankebene. Es ist also grundsätzlich mit konkurrierenden Zugriffen auf die einzelnen Datenbanken zu rechnen. Bei Entwurf der Workprozesse muß diesem Aspekt durch die Implementierung entsprechender Transaktionsstrategien Rechnung getragen werden.

Freie Schnittstellenprotokolle

An die durch das erwähnte Softwarebussystem realisierten Schnittstellen zwischen GUIs und Workprozessen bzw. zwischen den Workprozessen untereinander werden spezielle Anforderungen gestellt. Vor allem sollen diese Schnittstellen präsentationsfrei gestaltet werden. Das bedeutet, daß über sie keine Information bezüglich der Darstellungsform der vom GUI präsentierten Daten fließen soll. Beim Schnittstellenentwurf soll ferner bereits berücksichtigt werden, daß ein GUI durch eine andere Komponente ersetzt werden kann, die nicht von einem Menschen an einem Bildschirmarbeitsplatz bedient wird. Ein solcher GUI-Stellvertreter könnte beispielsweise eine spezielle Interface-Komponente zur Verbindung mit einem separaten Anwendungssystem sein oder aber eine Komponente, mit Hilfe derer gesammelte Eingabedaten „offline“ von einem Bandlaufwerk in das System gebracht werden können (siehe Bild 3.2).

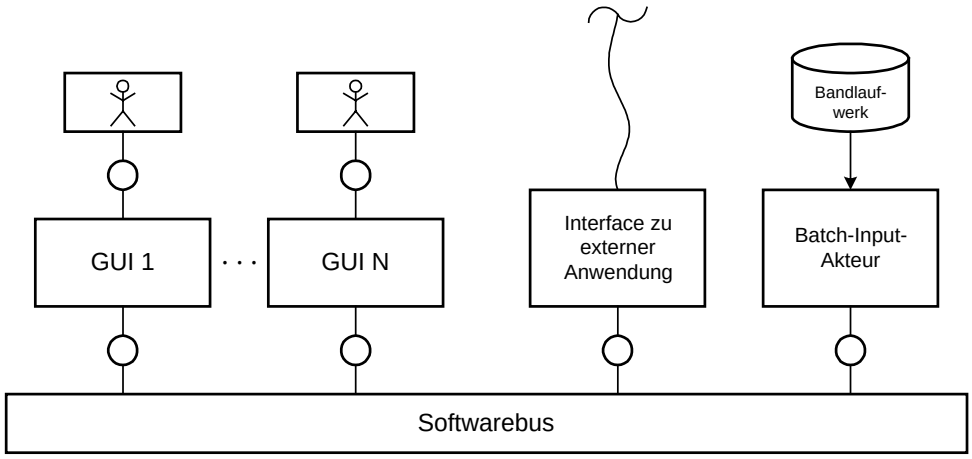


Bild 3.2 Weitere Komponenten in der Präsentationsebene

Eine zweite Anforderung betrifft die Kommunikationsprotokolle zwischen den Komponenten der Präsentations- und der Applikationsebene. Diese Protokolle sollen für die einzelnen Dienstleistungen frei gestaltbar sein. Eingabedaten sollen von den GUIs an die Workprozesse übertragen werden können, sobald sie verfügbar sind, und nicht erst, wenn eine Bildschirmmaske vollständig ausgefüllt worden ist.

3.2 Vorbemerkungen zur realisierten Architektur

Die im vorigen Abschnitt aufgestellten Forderungen sollen von der in den nachfolgenden Abschnitten 3.3 bis 0 dokumentierten Anwendungsarchitektur erfüllt werden. Diese Anwendungsarchitektur betrifft die Komponenten der Präsentations- und der Applikationsebene, also diejenigen Komponenten aus Bild 3.1, die für jede konkrete Anwendung neu zu entwickeln sind. Der Architekturentwurf basiert auf Überlegungen zum optimalen Einsatz des Rendezvous Softwarebusses in Multiclient-Multiserver-Systemen. Er umfaßt vier aufeinander aufbauende Schichten (Bild 3.3). Die unteren drei Schichten beinhalten Smalltalk-Universalkomponenten zur Realisierung von Workprozessen und GUIs mit VISUALWORKS. Die vierte Schicht enthält den für jede konkrete Anwendung neu zu entwickelnden Teil.

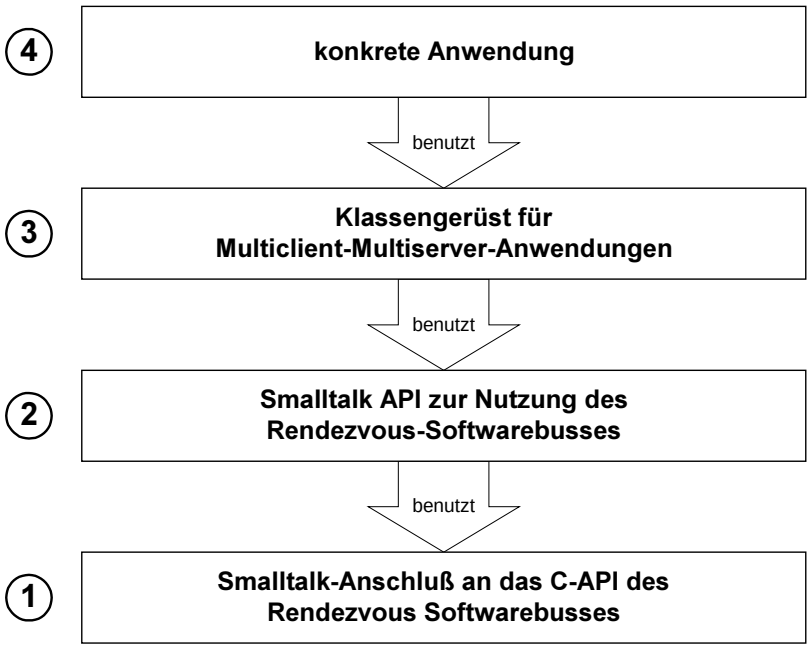


Bild 3.3 Vierschichtige Kommunikations-Architektur

Die unterste Schicht bildet der Rendezvous Softwarebus. Er ist nutzbar über ein *C-API* [6], welches zunächst für Smalltalk verfügbar gemacht werden muß. Dieses nun von Smalltalk aus benutzbare C-API wird in Schicht 2 objektorientiert gekapselt. Im Rahmen dieser Kapselung werden eine Reihe von Klassen eingeführt, mit Hilfe derer bereits beliebige Bus-Anwendungen realisiert werden können. Dieser Klassenkatalog wird als *Smalltalk-API* zur Nutzung des Rendezvous Softwarebusses (kurz: Smalltalk-Rendezvous-API) bezeichnet. Darüber hinaus existiert ein Klassengerüst, welches eine rasche Entwicklung von Anwendungen mit den im vorigen Abschnitt geforderten Eigenschaften ermöglicht (Schicht 3). In den Klassen aus diesem Klassengerüst wird wiederum das Smalltalk-Rendezvous-API genutzt. Schicht 4 beinhaltet schließlich, wie bereits erwähnt, den anwendungsspezifischen Teil der jeweiligen Anwendung. Viele Klassen der Schicht 4 sind Kunden oder Nachkommen von Klassen aus Schicht 3.

---

[6] API: Abkürzung für *Application Programming Interface*; programmiersprachenspezifische Schnittstelle, die Sprachmittel zur Verfügung stellt, mit denen Anwendungssysteme programmiert werden können, aus denen heraus Fremdkomponenten beeinflußbar sind.

3.3 Schicht 1: Der Softwarebus

Der RENDEZVOUS SOFTWARE BUS ist ein Produkt der Firma TIBCO (vormals Teknekron Software Systems) aus Palo Alto, Kalifornien, USA. Es wurde im Sommer 1995 im Rahmen mehrerer Evaluationsprojekte der SAP-AG, Abteilung Basismodellierung, vor Ort in den USA untersucht und getestet. Eine ausführliche Dokumentation der Produktmerkmale befindet sich in dem während dieser Evaluationsarbeiten entstandenen Bericht [Auer 96]. Im folgenden Abschnitt 3.3.1 werden Aufbau und Eigenschaften des Bussystems kurz vorgestellt. Danach folgt in den Abschnitten 3.3.2 und 3.3.3 eine Dokumentation der wichtigsten Kommandos aus dem mitgelieferten C-API. Drei exemplarische Benutzungsabläufe sollen anschließend die vorgesehene C-API-Benutzung veranschaulichen (Abschnitt 3.3.4). In Abschnitt 3.3.5 wird schließlich gezeigt, wie die mitgelieferte C-Sprachschnittstelle für Smalltalk verfügbar gemacht werden kann.

3.3.1 Aufbau und Eigenschaften

Bild 3.4 zeigt das Aufbaumodell des Rendezvous Softwarebusses. In der oberen Bildhälfte befinden sich die als Busteilnehmer auftretenden Anwendungen, in der unteren Bildhälfte der Softwarebus selbst. Wie bereits erwähnt, handelt es sich um einen Publish-Subscribe-Bus, der ganz ähnlich wie ein Hardwarebus funktioniert. Der Softwarebus besteht intern aus einem Kommunikationsakteur und mehreren Filtern. Jedem Busteilnehmer ist ein Filter zugeordnet, das über die gezeigten Auftrag-Rückmeldungs-Schnittstellen vom Busteilnehmer selbst konfiguriert werden kann. Die Busteilnehmer dürfen sich auf verschiedenen, durch ein Netzwerk verbundenen Rechnern befinden. Der gezeigte Kommunikationsakteur muß also intern wiederum aus mehreren auf die beteiligten Rechner verteilten Akteuren aufgebaut sein, die gemäß dem jeweiligen Netzwerkprotokoll (z.B. TCP/IP) miteinander kommunizieren. Der Entwickler der als Busteilnehmer auftretenden Anwendungen braucht sich jedoch um diese Netzwerkprotokolle nicht zu kümmern. Er benutzt ein C-API, über das er die Kommunikation mit anderen Busteilnehmern gemäß folgenden Prinzipien steuern kann:

Busteilnehmer können über ihre Kanäle zum Kommunikationsakteur Nachrichten unter frei definierbaren *Subjects*[7] veröffentlichen. Des weiteren können Busteilnehmer Subjects abonnieren und damit den Empfang von unter diesen Subjects veröffentlichten Nachrichten ermöglichen. Subjects haben die Form einer Zeichenkette. Das Abonnement eines Subjects durch einen Busteilnehmer geschieht durch einen Konfigurationsauftrag an den ihm zugeordneten Filter-Akteur. Dieser Filter-Akteur läßt anschließend alle Nachrichten unter dem abonnierten Subject – gleichgültig, aus welcher Quelle sie stammen – über den aufwärts führenden Kanal vom Kommunikationsakteur zum Busteilnehmer durch. Abonnements können von einem Busteilnehmer auch wieder gekündigt werden. Das hat zur Folge, daß der dem Busteilnehmer zugeordnete Filter-Akteur unter dem gekündigten Subject keine Nachrichten mehr nach oben weitergibt. Ein Busteilnehmer, der eine Nachricht veröffentlicht, kann grundsätz-

[7] subject (engl.), hier: Thema, Fach, Gebiet

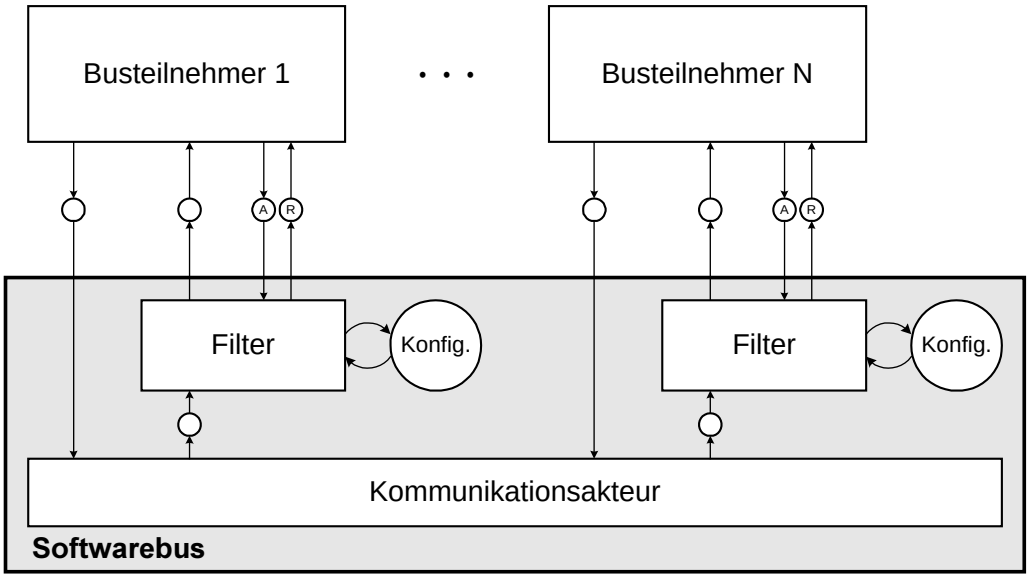


Bild 3.4 Aufbaumodell des Rendezvous Softwarebusses

lich nicht feststellen, ob die veröffentlichte Nachricht überhaupt von jemand empfangen wird bzw. wie viele Busteilnehmer die Nachricht empfangen. Die Veröffentlichung einer Nachricht unter einem Subject führt zur Auslösung eines *Busereignisses* unter diesem Subject.

Neben dem gewöhnlichen Abonnementmechanismus, bei dem das zu abonnierende Subject explizit benannt werden muß, gibt es zusätzlich die Möglichkeit, sogenannte *Inbox-Subjects* zu abonnieren. Ein Inbox-Subject ist ein individuell für einen bestimmten Busteilnehmer generiertes Subject, das von weiteren Busteilnehmern dann nicht mehr abonniert werden kann. Das Abonnement eines solchen Inbox-Subjects erfolgt ebenfalls durch einen Auftrag an den für den Busteilnehmer zuständigen Filter-Akteur – allerdings hier ohne die explizite Angabe des zu abonnierenden Subjects. Der Filter-Akteur generiert daraufhin ein individuelles Subject und teilt dieses in der Rückmeldung zum Abonnenmentauftrag dem Busteilnehmer mit. Das Inbox-Subject ist also zunächst nur dem Busteilnehmer bekannt, der es abonniert hat. Dieser kann es nun als Nachricht unter irgendeinem anderen Subject veröffentlichen und damit weiteren Busteilnehmern bekannt machen. Alle Busteilnehmer, die auf diesem Wege das Inbox-Subject mitgeteilt bekommen haben, sind damit in der Lage, Nachrichten durch Veröffentlichung unter diesem Inbox-Subject gezielt an den garantiert einzigen Abonnenten dieses Inbox-Subjects zu senden. Ein Inbox-Subject ist also eine Art Telefonnummer, unter dem man höchstens einen Busteilnehmer erreicht. Durch diesen Mechanismus lassen sich Punkt-zu-Punkt-Verbindungen zwischen Busteilnehmern aufbauen, wie später in Abschnitt 3.5 gezeigt wird.

Ferner gibt es bei Veröffentlichungen den Sonderfall der sogenannten *Auftragsveröffentlichungen*. Sie werden genauso wie gewöhnliche Veröffentlichungen unter einem beliebigen Subject veröffentlicht, ziehen aber zusätzlich ein *Rückmeldungsabonnement* nach sich. Ein Busteilnehmer, der eine Auftragsveröffentlichung auf den Bus gibt, abonniert damit also implizit ein Inbox-Subject für Rückmeldungen. Dieses generierte Inbox-Subject wird vor dem Absenden an die Auftragsveröffentlichung geheftet. Jeder Busteilnehmer, der eine solche Auftragsveröffentlichung empfängt, ist so in der Lage, eine eventuelle Rückmeldung gezielt an den auftraggebenden Busteilnehmer zurückzusenden.

Die Busbenutzung geschieht durch Aufruf von Buskommandos, die als C-Funktionen innerhalb einer Dynamic-Link-Library (DLL) implementiert sind. Das zugehörige C-Headerfile befindet sich in Anhang A. Es existieren drei Gruppen von Kommandos, die folgendermaßen bezeichnet werden [RVProg 95]:

Communications API Darunter fallen sämtliche Befehle zur Steuerung der oben erläuterten Kommunikationsvorgänge.

Message API Beinhaltet Operationen zum Zusammenbau von komplexen Busnachrichten vor deren Veröffentlichung.

Advisory Message API Bleibt im Rahmen dieser Arbeit unberücksichtigt.

3.3.2 Das Communications API

Im folgenden werden die wichtigsten Befehle aus dem Communications API anhand der zugehörigen C-Funktionsaufrufe erklärt. Sie lassen sich in drei Gruppen aufteilen: Kommandos zur Sitzungsverwaltung, Sende-Befehle und Befehle, mit denen die Empfangsbereitschaft eines Busteilnehmers gesteuert werden kann. C-Funktionsparameter, die zur Wertrückgabe dienen, sind in den dargestellten Funktionsaufrufen durch das Adreß-Präfix „&“ gekennzeichnet. Auf weggelassene Argumente, die zum Verständnis der aufgeführten Kommandos nicht benötigt werden, wird mit „...“ hingewiesen.

Sitzungsverwaltung

Wenn ein Anwendungsakteur als Busteilnehmer auftreten soll, muß für ihn vorher eine Bussitzung eröffnet werden. Dies geschieht durch die Anweisung :

```
rv_Init(&sessionID,...)
```

Über den Ausgangsparameter ‘sessionID’ wird eine eindeutige Sitzungskennung zurückgegeben, mit der sich der Busteilnehmer in Zukunft gegenüber dem Bussystem identifizieren kann. Eine Bussitzung wird beendet mit dem Kommando:

```
rv_Term(sessionID)
```

Dabei muß die Sitzungskennung der zu schließenden Sitzung angegeben werden. Nach Ausführung von ‘rv_Term’ ist diese Sitzungskennung dann ungültig. Ein Aufruf von ‘rv_Term’ bewirkt implizit die Kündigung aller innerhalb der Bussitzung gemachten Abonnements.

Sende-Befehle

Gewöhnliche Veröffentlichungen werden durch ‘rv_Send’ auf den Bus gegeben:

```
rv_Send(sessionID, subject, msgType, msgSize, msgData)
```

Dabei wird als erstes Argument die Sitzungskennung der Sitzung angegeben, innerhalb der der Busteilnehmer eine Veröffentlichung auf den Bus geben will. Als zweites Argument folgt das Subject, unter dem eine Nachricht veröffentlicht werden soll. Das darauf folgende Argu-

ment-Tripel (‘msgSize’, ‘msgType’, ‘msgData’) beschreibt schließlich die zu veröffentlichende Nachricht durch eine Größenangabe (‘msgSize’: Länge in Bytes), eine Typangabe (‘msgType’), und einen Zeiger auf die die Nachricht enthaltende Bytefolge (‘msgData’). Bei Auftragsveröffentlichungen lautet der Sende-Befehl ‘rv_Rpc’:

```
rv_Rpc(sessionID,&replySubscriptionID, subject,
        msgType, msgSize, msgData, callbackFunction, ...)
```

Auch hier gibt man Sitzungskennung (‘sessionID’), Subject (‘subject’) und die Beschreibung der zu veröffentlichenden Nachricht (‘msgSize’, ‘msgType’, ‘msgData’) an. Wie bereits erwähnt, implizieren Auftragsveröffentlichungen das Abonnement eines Inbox-Subjects für Rückmeldungen. Daher muß zusätzlich eine Variable zur Aufnahme der Rückmeldungsabonnement-Kennung (‘replySubscriptionID’) angegeben werden. Diese Variable wird während der Befehlsausführung belegt. Ihr Inhalt dient bei einer späteren Kündigung zur Identifikation des zu kündigenden Abonnements (‘rv_Close’, siehe unten). Des weiteren benötigt ‘rv_Rpc’ die Angabe eines Zeigers auf eine C-Funktion (‘callbackFunction’), die bei Auftritt eines Busereignisses unter dem abonnierten Inbox-Subject ausgeführt werden soll.

Steuerung der Empfangsbereitschaft (Filter-Konfiguration)

Das Abonnement eines Subjects geschieht mit:

```
rv_ListenSubject(sessionID, &subscriptionID, subject,
        callbackFunction, ...)
```

Neben der Kennung der Sitzung (‘sessionID’), innerhalb der das Abonnement durchgeführt werden soll, wird das zu abonnierende Subject angegeben (‘subject’). Es folgt eine Verweis auf eine C-Funktion (‘callbackFunction’), die bei Auftritt eines Busereignisses unter dem abonnierten Subject ausgeführt werden soll. Auch hier wird nach Befehlsausführung über den Ausgangsparameter ‘subscriptionID’ eine eindeutige Abonnementkennung zurückgegeben. Inbox-Subjects werden durch Aufruf von ‘rv_ListenInbox’ abonniert:

```
rv_ListenInbox(sessionID, &subscriptionID, &inboxSubject,
        callbackFunction, ...)
```

Im Unterschied zu ‘rv_ListenSubject’ fehlt die Angabe eines Subjects. Stattdessen wird eine Variable zur Aufnahme des generierten Inbox-Subjects übergeben. Alle weiteren Argumente behalten ihre Bedeutung. Abonnements sind, unabhängig davon, ob sie durch ‘rv_ListenSubject’, ‘rv_ListenInbox’ oder ‘rv_Rpc’ zustandegekommen sind, kündbar mit dem Befehl ‘rv_Close’:

```
rv_Close(sessionID, subscriptionID)
```

Das zu kündigende Abonnenment wird durch seine Abonnementkennung (‘subscriptionID’) identifiziert. Auf hier wird die Angabe der Sitzungskennung (‘sessionID’) benötigt. Um letztendlich überhaupt empfangsbereit zu werden, muß ein Busteilnehmer vorher die Kontrolle an den externen Busakteurs abgeben, indem er dessen Eventloop aufruft:

```
rv_MainLoop(sessionID)
```

Treten nun Busereignisse unter abonnierten Subjects auf, werden von dort aus diejenigen Callback-Routinen aufgerufen, deren Adressen beim jeweiligen Abonnentmentauftrag mitgeteilt worden sind. Die Rückkehr aus ‘rv_MainLoop’ geschieht erst bei Sitzungsbeendigung.

Abgeleitete Entitätstypen

Bei Benutzung des Communications-API hat man folglich die in Bild 3.5 gezeigte Struktur vor Augen. Veröffentlichungen und Abonnements beziehen immer auf eine Bussitzung, die durch eine Sitzungskennung identifizierbar ist. Bei Veröffentlichungen wird unterschieden zwischen gewöhnlichen Veröffentlichungen und Auftragsveröffentlichungen. In beiden Fällen hat man ein Subject, unter dem eine Nachricht veröffentlicht werden soll. Auftragsveröffentlichungen ziehen zusätzlich Rückmeldungsabonnements nach sich. Damit verbunden ist eine Rückmeldungsabonnement-Kennung und eine Callback-Routine, die bei Auftritt einer Rückmeldung ausgeführt wird. Gewöhnliche Abonnements werden durch ihre Abonnementkennung identifiziert und beziehen sich auf ein Subject. Ferner ist auch ihnen jeweils eine Callback-Routine zugeordnet, die bei Auftritt eines Busereignisses unter dem abonnierten Subject aufgerufen wird.

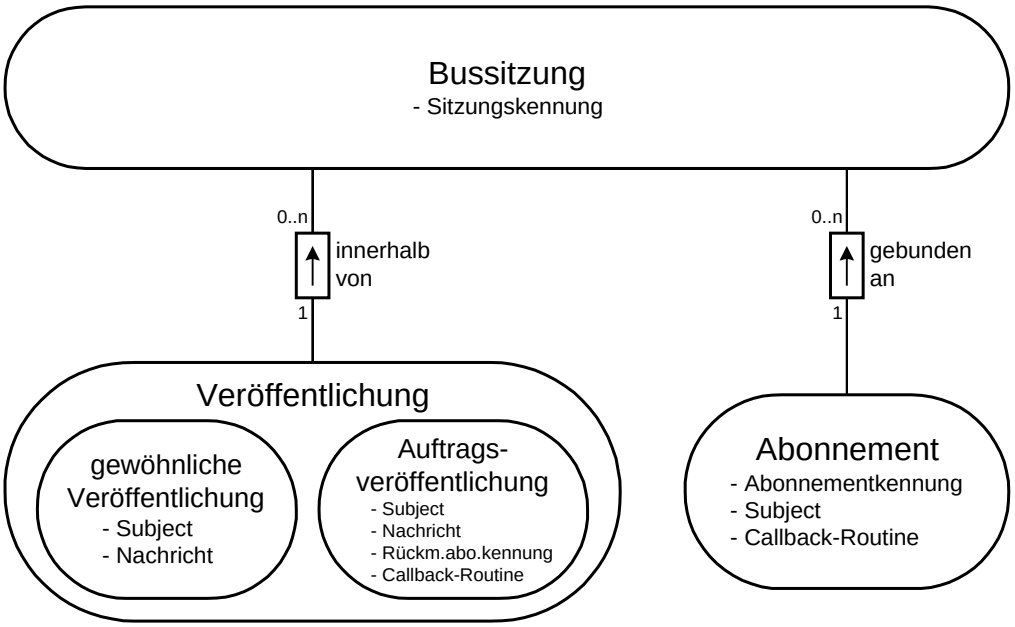


Bild 3.5 Communications-API: abgeleitete Entitätstypen

3.3.3 Das Message API

Einfache und zusammengesetzte Nachrichten

Es wurde bereits gezeigt, daß Busnachrichten durch ein Tripel der Form (Typ, Länge, Inhalt) beschrieben werden. Nachrichten sind also typgebunden. Tabelle 3.1 zeigt Beispiele für Nachrichten einfachen Typs. „Einfach“ soll heißen, daß ihr Inhalt keine Struktur besitzt, sondern aus einem einzigen Element des angegebenen Typs besteht.

Es fällt auf, daß die angegebene Länge der Nachricht vom Typ RVMSG_STRING Null ist, obwohl die enthaltene Zeichenkette aus zehn Zeichen besteht. RVMSG_STRING gehört zu den Nachrichtentypen, bei denen keine Längenangabe gebraucht wird. Die Länge der Zei-

chenkette innerhalb einer String-Nachricht kann zur Laufzeit vom externen Busakteur selbst bestimmt werden. Bei den drei weiteren gezeigten Nachrichtentypen ist jedoch die explizite Angabe der Länge in Bytes erforderlich.

Typ (msgType)	Länge (msgSize)	Inhalt (msgData)
RVMSG_STRING	0	„ein String“
RVMSG_INT	4	4711
RVMSG_REAL	8	1.4335e12
RVMSG_BOOLEAN	1	true

Tabelle 3.1 Beispiele für Nachrichten einfachen Typs

Neben den gezeigten einfachen Nachrichtentypen gibt es zusätzlich den Nachrichtentyp RVMSG_RVMSG. Nachrichten dieses Typs werden als *zusammengesetzte Nachrichten* bezeichnet. Sie bestehen aus beliebig langen Liste von *Nachrichtenfeldern*. Jedes Nachrichtenfeld enthält wiederum eine Nachricht, wobei der Typ dieser Nachricht beliebig sein darf. Das (Typ, Länge, Inhalt)-Tripel sieht bei zusammengesetzten Nachrichten folgendermaßen aus:

RVMSG_RVMSG	0	*<C-Bytearray>*

Auch hier ist keine Längenangabe erforderlich. Das den Nachrichteninhalt repräsentierende C-Bytearray muß vor Veröffentlichung einer solchen zusammengesetzten Nachricht unter Nutzung von Kommandos des Message-API initialisiert und gefüllt werden. Empfänger solcher Nachrichten nutzen ebenfalls das Message-API, um auf die enthaltenen Nachrichtenfelder zuzugreifen. Die wichtigsten der dazu zur Verfügung gestellten Befehle werden im folgenden erklärt.

Zusammenbau von Nachrichten

Bevor der Nachrichtenzusammenbau beginnen kann, muß zunächst ein Bytearray passender Größe initialisiert werden. Dieses wird anschließend als Puffer zur Aufnahme der zusammengesetzten Nachricht via ‘rvmsg_Init’ beim externen Busakteur angemeldet:

```
rvmsg_Init(sessionID, pointerToBuffer, bufferSize)
```

Auch hier identifiziert sich der Busteilnehmer durch seine Sitzungskennung (‘sessionID’). Als nächstes übergibt er einen Zeiger auf den initialisierten Pufferbereich (‘pointerToBuffer’) und dessen Größe in Bytes (‘bufferSize’). Das Füllen eines solchen Nachrichtenpuffers geschieht anschließend durch wiederholten Aufruf von ‘rvmsg_Append’:

```
rvmsg_Append(sessionID, pointerToBuffer, bufferSize,
        fieldName, msgType, msgSize, msgData)
```

‘Rvmsg_Append’ dient zum Anhängen eines neuen Nachrichtenfeldes an eine – eventuell noch leere – Liste von Nachrichtenfeldern innerhalb des Nachrichtenpuffers. Die ersten drei

Argumente identifizieren die Bussitzung des Busteilnehmers (‘sessionID’) und den von ihm durch ‘rvmsg_Init’ angemeldeten Nachrichtenpuffer (‘pointerToBuffer’, ‘bufferSize’). Es folgt die Beschreibung des anzuhängenden Nachrichtenfeldes. Zuerst wird ein Feldname spezifiziert (‘fieldName’), anschließend der Feldinhalt (‘msgType’, ‘msgSize’, ‘msgData’). Nachrichtenfelder dürfen Nachrichten beliebigen Typs enthalten. Reicht die bei ‘rvmsg_Init’ festgelegte Puffergröße nicht aus, um das neue Nachrichtenfeld aufzunehmen, wird eine Fehlermeldung zurückgegeben.

Parsen empfangener Nachrichten

Ein Busteilnehmer, der eine zusammengesetzte Nachricht empfängt, erhält dabei einen Zeiger auf das die Nachrichtenfelder enthaltende C-Bytearray. Mit ‘rvmsg_Get’ hat er anschließend die Möglichkeit, durch Angabe eines Feldnamens auf den Inhalt eines benannten Nachrichtenfeldes zuzugreifen:

```
rvmsg_Get(sessionID, pointerToBuffer, fieldName,
        &msgType, &msgSize, &msgData)
```

Mit den ersten beiden Argumenten identifiziert der Empfänger seine Bussitzung (‘sessionID’) und das die empfangene Nachricht enthaltene Bytearray (‘pointerToBuffer’). Als drittes Argument folgt der Name des Feldes (‘fieldName’), auf dessen Inhalt der Empfänger zugreifen will. Die im Feld enthaltene Nachricht wird nach Ausführung über die Ausgangsparameter ‘msgType’, ‘msgSize’ und ‘msgData’ an den Aufrufer zurückgegeben. Es ist durchaus erlaubt, mehrere Felder innerhalb einer zusammengesetzten Nachricht gleich zu benennen. In diesem Fall liefert ‘rvmsg_Get’ den Inhalt des ersten Feldes in der Feldliste, das den angegebenen Namen trägt. Möchte man auf alle Felder eines Namens oder gar auf sämtliche Felder einer zusammengesetzten Nachricht unabhängig von deren Feldnamen nacheinander zugreifen, bedient man sich des Kommandos ‘rvmsg_Apply’:

```
rvmsg_Apply(sessionID, pointerToBuffer, fieldName,
        applyCallbackFunction, ...)
```

Wiederum muß sich der Aufrufer über seine Sitzungskennung (‘sessionID’) indentifizieren und einen Zieger auf das zu parsende Array (‘pointerToBuffer’) übergeben. Anschließend folgt die Angabe eines Feldnamens (‘fieldName’) und eines Zeigers auf eine C-Funktion (‘applyCallbackFunction’), in der die Bearbeitung der Nachrichtenfelder erfolgen soll, die den angegebenen Namen tragen. Diese C-Funktion muß von einem bestimmten Typ sein (‘rvmsg_ApplyCallback’, siehe Anhang A), d.h. es müssen Argumente vorgesehen werden, in denen beim Aufruf der Feldname und das Nachrichtenumschreibungs-Tripel (Typ, Länge, Inhalt) des jeweils zu bearbeitenden Feldes übergeben werden können. Der Aufruf von ‘rvmsg_Apply’ zieht dann unmittelbar den – unter Umständen mehrfachen – Rückruf dieser Bearbeitungsroutine nach sich. Bei jedem Rückruf wird genau ein Feld mit dem angebenen Feldnamen übergeben. Der Rückruf wiederholt sich so lange, bis alle Felder des bei ‘rvmsg_Apply’-Aufruf spezifierten Namens abgearbeitet sind. Wird kein Feldname angegeben (fieldName = NULL), wird die Bearbeitungsroutine für *jedes* Feld der zusammengesetzten Nachricht zurückgerufen.

Abgeleitete Entitätstypen

Aus der Semantik der Kommandos des Message API lassen sich die in Bild 3.6 gezeigten Entitätstypen ableiten: Jede Veröffentlichung enthält eine Nachricht. Dabei kann es sich um eine einfache Nachricht, d.h. eine Nachricht einfachen Typs, oder eine zusammengesetzte Nachricht handeln. Beide werden durch ein Tripel der Form (Typ, Länge, Inhalt) beschrieben. Zusammengesetzte Nachrichten bestehen aus einer Liste von Nachrichtenfeldern. Jedes Nachrichtenfeld hat einen Namen und enthält – genauso wie eine Veröffentlichung – eine Nachricht. Dies kann wiederum eine einfache oder eine aus Nachrichtenfeldern zusammengesetzte Nachricht sein. Auf diese Weise lassen sich rekursiv hierarchisch strukturierte Nachrichten beliebiger Komplexität zusammenbauen.

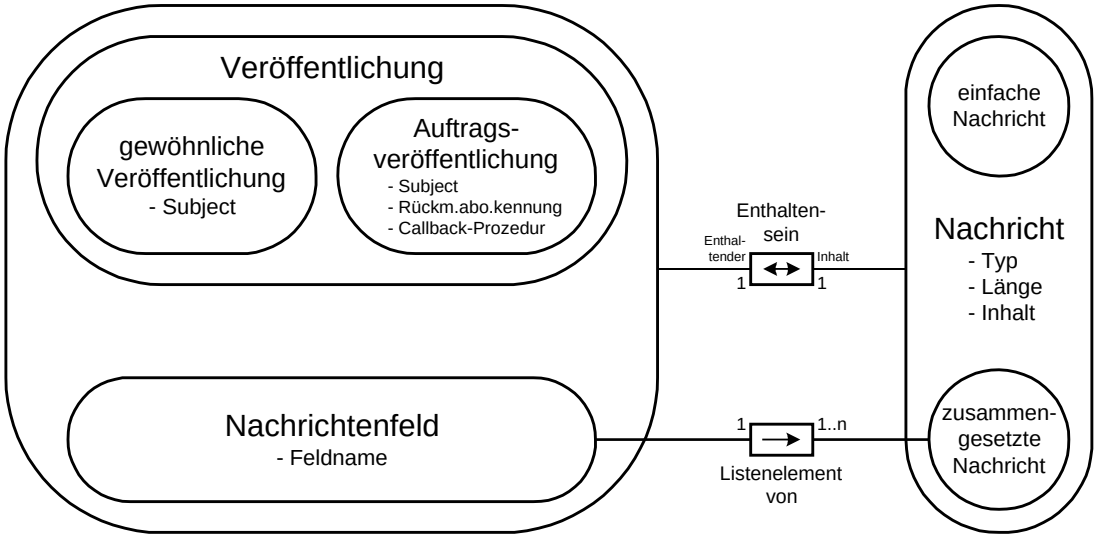


Bild 3.6 Message API: abgeleitete Entitätstypen

3.3.4 Typische Benutzungsabläufe

Anhand von zwei Ablaufbildern soll die vorgesehene Benutzung des Communications API und des Message API noch einmal verdeutlicht werden. Bild 3.7 zeigt links den Ablauf bei Veröffentlichung einer Nachricht: Zunächst wird mit ‘rv_Init’ eine Bussitzung eröffnet. Soll eine zusammengesetzte Nachricht veröffentlicht werden, muß sie vorher zusammengebaut werden. Nach Veröffentlichung der Nachricht durch ‘rv_Send’ kann die Bussitzung wieder geschlossen werden (‘rv_Term’). Rechts im Bild 3.7 ist der Ablauf bei Zusammenbau einer zusammengesetzten Nachricht gezeigt – der Einfachheit halber für den Fall, daß die enthaltenen Nachrichtenfelder selbst keine zusammengesetzten Nachrichten mehr enthalten. Hier muß also zunächst die vollständige Länge der späteren Nachricht bestimmt werden. Anschließend wird Speicherplatz für ein entsprechend großes Array besorgt und dieses via ‘rvmsg_Init’ als Nachrichtenpuffer angemeldet. Danach werden sukzessive die Nachrichtenfelder durch mehrmaligen Aufruf von ‘rvmsg_Append’ angehängt.

Bild 3.8 zeigt den erforderlichen Ablauf, um Nachrichten unter einem bestimmten Subject empfangen zu können. Auch hier muß der Busteilnehmer als erstes eine Bussitzung eröffnen. Anschließend erfolgt das Abonnement des gewünschten Subjects durch Aufruf von

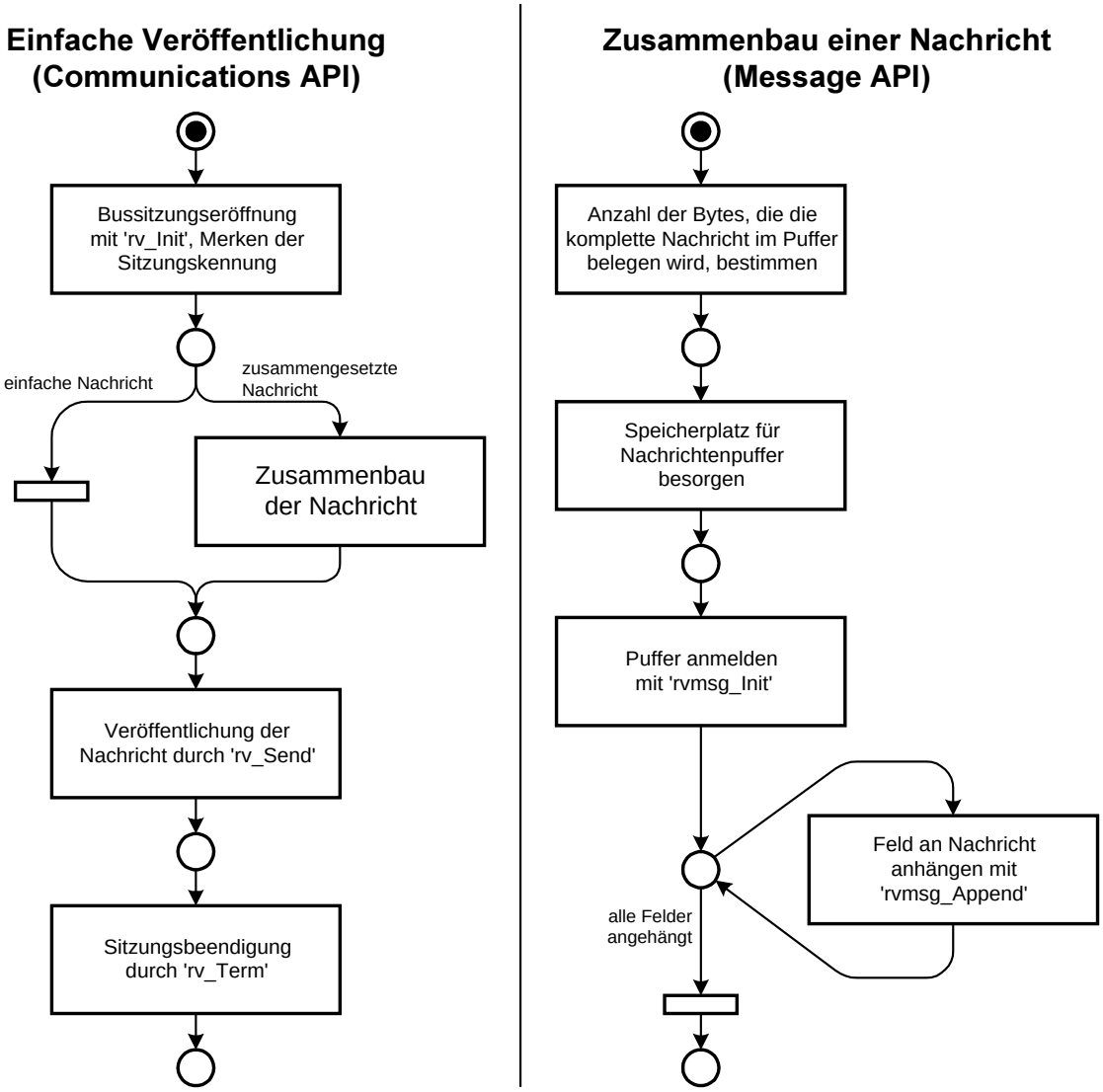


Bild 3.7 Einfache Veröffentlichung und Zusammenbau einer Nachricht

‘rv_ListenSubject’. Um schließlich empfangsbereit zu werden, wird durch Aufruf von ‘rv_MainLoop’ die Kontrolle an den externen Busakteur abgegeben. Dessen Bus-Eventloop wird nun zyklisch durchlaufen, wobei eine Triggerung durch die auftretenden Busereignisse erfolgt (grau unterlegte Transitionen). Bezüglich dieser Ereignisse sind drei Fälle zu unterscheiden:

1. Es handelt sich um ein Busereignis unter dem abonniertem Subject (linker Pfad im Schleifenrumpf): Dann erfolgt der Aufruf derjenigen Callback-Routine, die beim Abonnement angegeben wurde.
2. Es handelt sich um ein Busereignis unter einem anderen, nicht abonnierten Subject (rechter Pfad im Schleifenrumpf): Für den Busteilnehmer passiert nichts weiter.
3. Es handelt sich um das Sitzungsschlußereignis, ausgelöst durch Aufruf von ‘rv_Term’: Die Sitzung wird beendet und die Eventloop verlassen. Der (ehemalige) Busteilnehmer hat die Marke wieder.

Der Aufruf von ‘rv_Term’ kann in diesem Fall nur innerhalb der Callback-Routine erfolgen, zum Beispiel in Abhängigkeit vom Inhalt der empfangenen Nachricht.

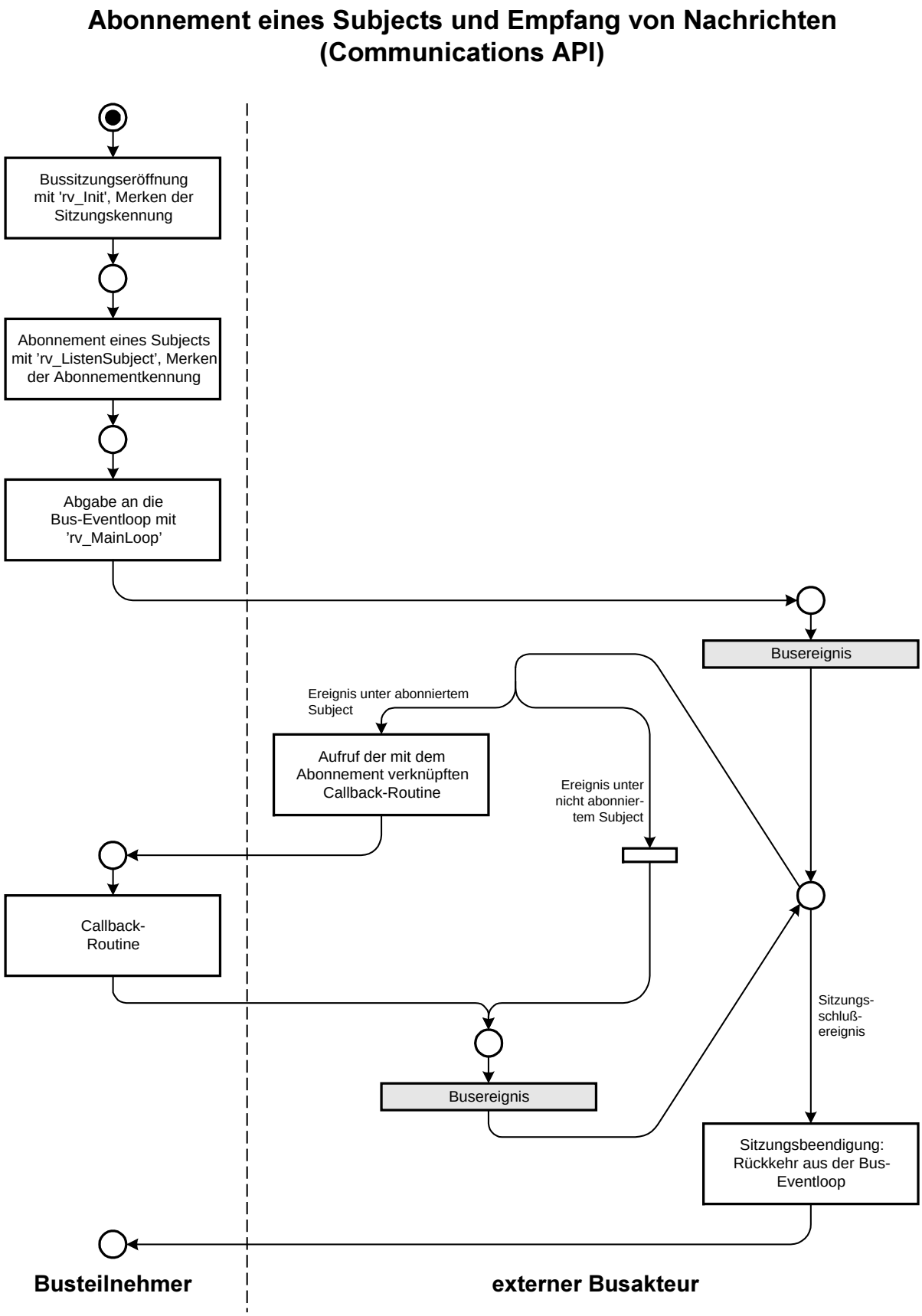


Bild 3.8 Abonnement eines Subjects und Empfang von Nachrichten

3.3.5 Verfügbarmachung der C-Schnittstelle in Smalltalk

Die Benutzung des Rendezvous Softwarebusses geschieht durch Aufrufe von C-Funktionen aus einer zum Lieferumfang des Bussystems gehörenden Dynamic-Link-Library (DLL) mit dem Namen „rv.dll“. Die Schnittstelle dieser DLL – d.h. Communications API, Message API und Advisory Message API – ist durch entsprechende Funktionsdeklarationen im zugehörigen C-Headerfile „rv.h“ öffentlich gemacht. Ferner befinden sich in diesem Headerfile einige zur C-API-Benutzung benötigte Konstanten- und Typdefinitionen. Um Bus-Anwendungen in Smalltalk realisieren zu können, muß man also in der Lage sein, aus Smalltalk heraus C-Funktionen aus einer DLL aufzurufen. Im Falle von VISUALWORKS kann man sich dazu Durchreicherobjekte schaffen, die Smalltalk-Methodenaufrufe in C-Funktionsaufrufe umsetzen und in eine DLL weiterleiten. Mit Hilfe eines speziellen Werkzeuges ist es möglich, durch Parsen von C-Headerfiles Klassenobjekte generieren zu lassen, die als Erzeuger solcher Durchreicherobjekte dienen. Bild 3.9 veranschaulicht dies für den vorliegenden Fall. Links ist das C-Headerfile „rv_st.h“ gezeigt. Es ist eine gegenüber dem Original „rv.h“ leicht modifizierte Version[8], die im Anhang A vollständig abgedruckt ist. Dieses Headerfile ist eine von zwei Eingangsgrößen für das in der Mitte dargestellte DLL&C-Connect-Werkzeug zur Erzeugung von C-Interface-Klassenobjekten. Die zweite Eingangsgröße ist eine Pfadangabe, unter der die DLL, zur der das Headerfile gehört, im Dateisystem des Rechners zu finden ist. Das Werkzeug kreiert anhand dieser beiden Eingaben zwei Objekte im Objektspeicher: zum einen das Klassenobjekt **RVInterface**, zum anderen den Attribut-Pool mit dem Namen **RVInterfaceDictionary**.

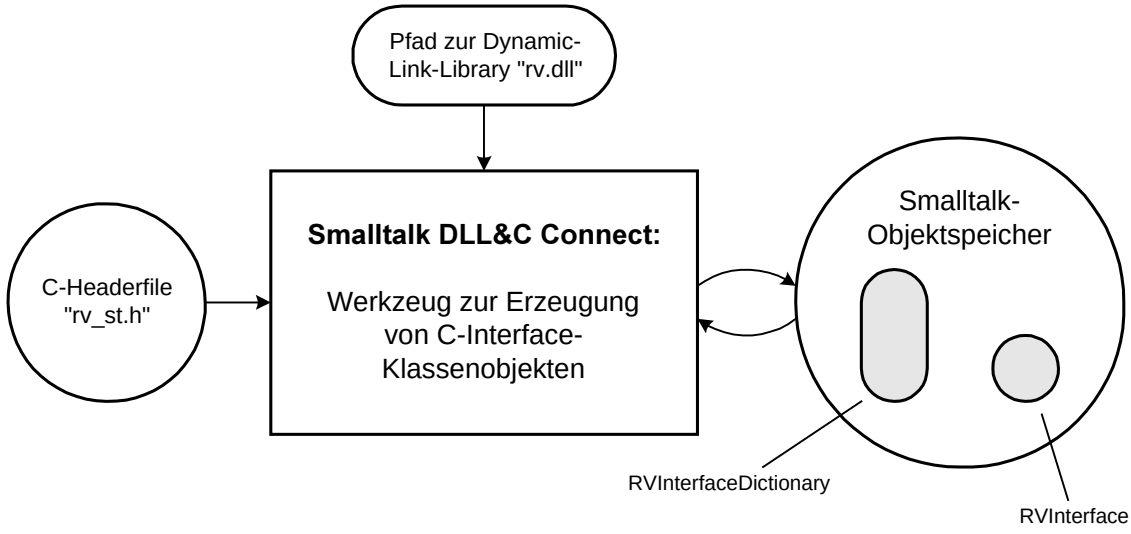


Bild 3.9 Smalltalk DLL&C Connect

Das Klassenobjekt RVInterface dient zum Erzeugen eines einzigen Objekts, des *RV-Interface-Monopolakteurs* (Bild 3.10). Er stellt das Bindeglied zwischen einer Smalltalk-Anwendung (unten) und dem externen Busakteur (oben) dar. Der RVInterface-Monopolakteur übernimmt zwei Aufgaben: Zum einen dient er als Durchreicher für Buskommandos (Bus-Auftrags- und

[8] Die Modifikationen betreffen lediglich die C-Syntax. Sie waren erforderlich, um den Parser des im Bild 3.9 in der Mitte dargestellten Smalltalk-DLL&C-Connect-Werkzeuges überhaupt einsetzen zu können.

Rückmeldungsdurchreicher, im Bild 3.10 rechts in der Mitte). Zum anderen erlaubt er über die Auftrag-Rückmeldungs-Schnittstelle links unten den Zugriff auf seine im RVInterface-Dictionary liegenden Pool-Attribute. Bei diesen Poolattributen handelt es sich einerseits um Konstanten, die gemäß der Konstantendefinitionen des C-Headerfiles generiert wurden, zum anderen um sogenannte C-Factory-Objekte. Diese Factory-Objekte sind anhand der im C-Headerfile vorkommenden Typdefinitionen generiert worden und dienen als Erzeuger für C-Vertreterobjekte. Solche Vertreterobjekte repräsentieren bei C-Funktionsaufrufen aus Smalltalk in den Funktionsargumenten C-Daten des betreffenden Typs. Der Lese-Pfeil vom RVInterfaceDictionary zur Smalltalk Anwendung deutet an, daß der Zugriff auf die Pool-Attribute auch unmittelbar, ohne Beteiligung des RVInterface-Monopolakteurs erfolgen kann, nämlich genau dann, wenn man in den entsprechenden Anwendungsklassen das RVInterfaceDictionary ebenfalls als Attribut-Pool vorsieht.

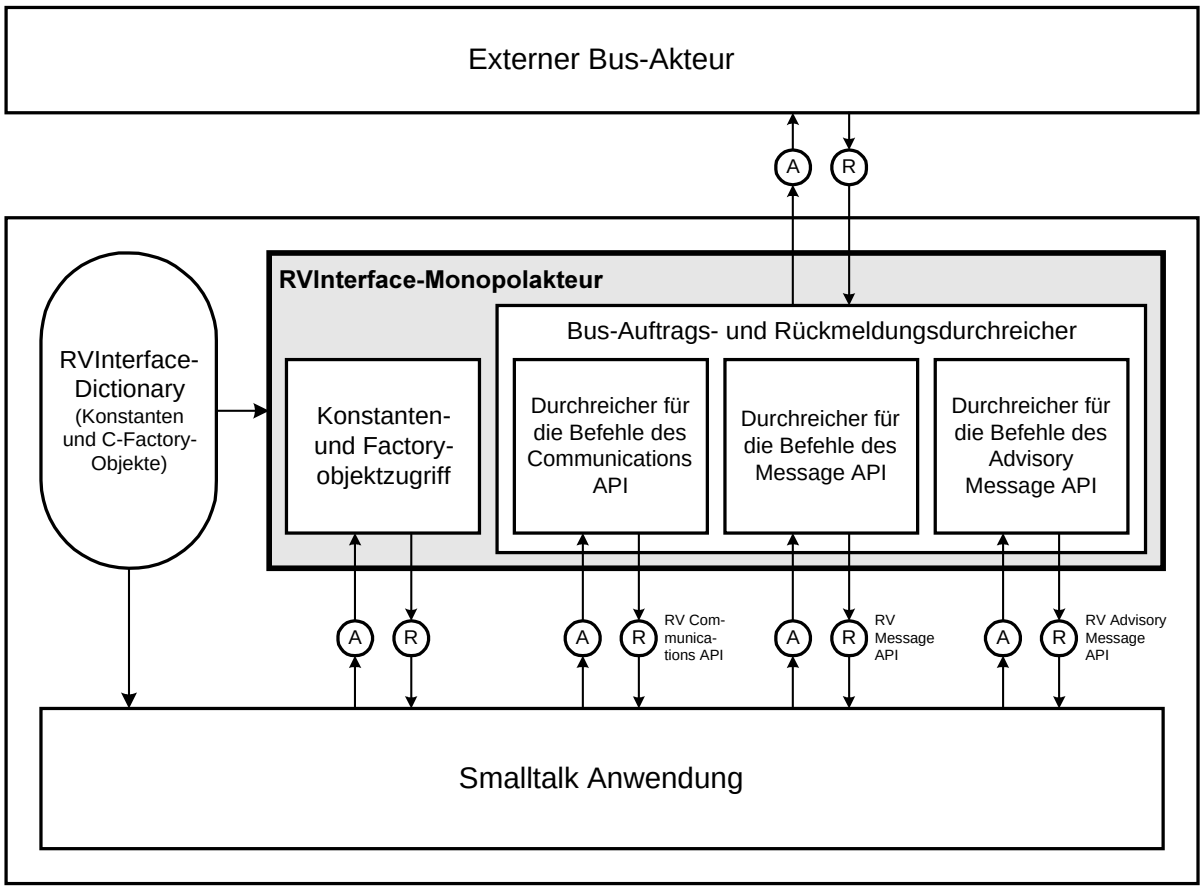


Bild 3.10 RVInterface-Monopolakteur

Der Bus-Auftrags- und Rückmeldungsdurchreicher innerhalb des RVInterface-Monopolakteurs ist realisiert durch einen Satz von Methoden, die eins zu eins den aufrufbaren C-Funktionen in der DLL entsprechen. Bild 3.11 zeigt die Abbildung am Beispiel des Bus-Kommandos ‘rv_Send’ (Communications API, siehe Seite 30). Rechts ist die dem C-Headerfile entnommene Funktionsdeklaration gezeigt, links der Methodenkopf der generierten Smalltalk-Methode. Der das gesamte C-API kapselnde Methodensatz ist gemäß der gegebenen API-Unterteilung in drei Gruppen partitioniert. Für jede dieser Methodengruppen ist ein eigener Akteur innerhalb des Busauftrags- und Rückmeldungsdurchreichers eingezeichnet, jeweils zuständig für das Durchreichen von Befehlen des Commmunications API, des Message API und des Advisory Message API.

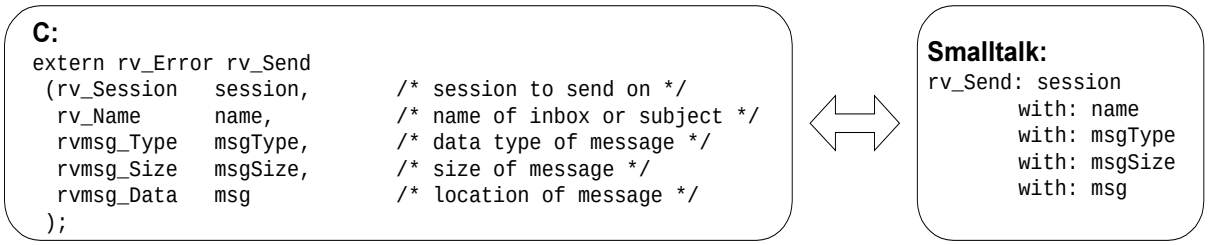


Bild 3.11 Buskommando rv_Send in C-Syntax und in Smalltalk-Syntax

Um nun innerhalb einer Smalltalk-Anwendung den Softwarebus benutzen zu können, muß man alle an der Buskommunikation beteiligten Objekte vom RVInterface-Monopolakteur in Kenntnis setzen, indem man ihn entweder über ein Globalsymbol öffentlich zugänglich macht, oder einen Verweis auf ihn in den Attributen der beteiligten Objekte einträgt. Danach können diese Objekte durch Aufruf der entsprechenden Methoden des RVInterface-Monopolakteurs jene Busbefehle absetzen, die in den Abschnitten 3.3.2 und 3.3.3 aufgeführt wurden. Als Smalltalk-Anwendungsentwickler empfindet man diesen Mechanismus dennoch als wenig komfortabel. Dies liegt zum einen daran, daß die auf die Programmiersprache C zugeschnittene Schnittstelle verlangt, in Methodenargumenten Verweise auf Variablen für Rückgabewerte zu übergeben, die während der Befehlsausführung belegt werden. Abgesehen davon, daß dies in Smalltalk als schlechter Stil gilt, macht es Mühe, vor jedem Methodenaufruf die passenden C-Vertreterobjekte zu erzeugen. Zum anderen ist es ganz allgemein aus Sicht eines Entwicklers objektorientierter Systeme unschön, daß sämtliche Buskommandos von einem einzigen Objekt – dem RVInterface-Monopolakteur – ausgeführt werden. Dadurch wird das objektorientierte Modularisierungsprinzip verletzt, welches besagt, daß die Daten als wesentliches Strukturierungskriterium gelten sollten, und nicht die Operationen [Meyer 90].

Diese Defizite motivierten eine Kapselung des Original-API durch ein objektorientiertes Smalltalk-API, in dem C kaum noch durchscheint. Das so entstandene *Smalltalk-Rendezvous-API* wird im folgenden Abschnitt vorgestellt.

3.4 Schicht 2: Das Smalltalk-Rendezvous-API

Das Smalltalk-Rendezvous-API ist eine Sammlung von Smalltalk-Klassen, die zur Entwicklung beliebiger Busanwendungen genutzt werden kann. Das im Rahmen der vorliegenden Arbeit betrachtete Anwendungsfeld „Client-Server-Systeme“ spielt auf dieser Ebene noch keine Rolle. Der zu den eingeführten Klassen gehörende Quelltext befindet sich im Anhang B.

3.4.1 Die wichtigsten Klassen im Überblick

Bild 3.12 zeigt die wichtigsten Klassen des Smalltalk-Rendezvous-API in einem ER-Diagramm. Die Benennungen der Entitätstypen entsprechen denen der zuständigen Klassenobjekte im Smalltalk-Objektspeicher. Zusätzlich sind in den Entitäts-Knoten die wichtigsten

Attribute der Exemplare der gezeigten Klassen angegeben. Nicht eingetragen sind solche Attribute, die die dargestellten Beziehungen realisieren. Dem Leser sollten bei Betrachtung dieses Bildes die Ähnlichkeiten mit Bild 3.5 und Bild 3.6 auffallen.

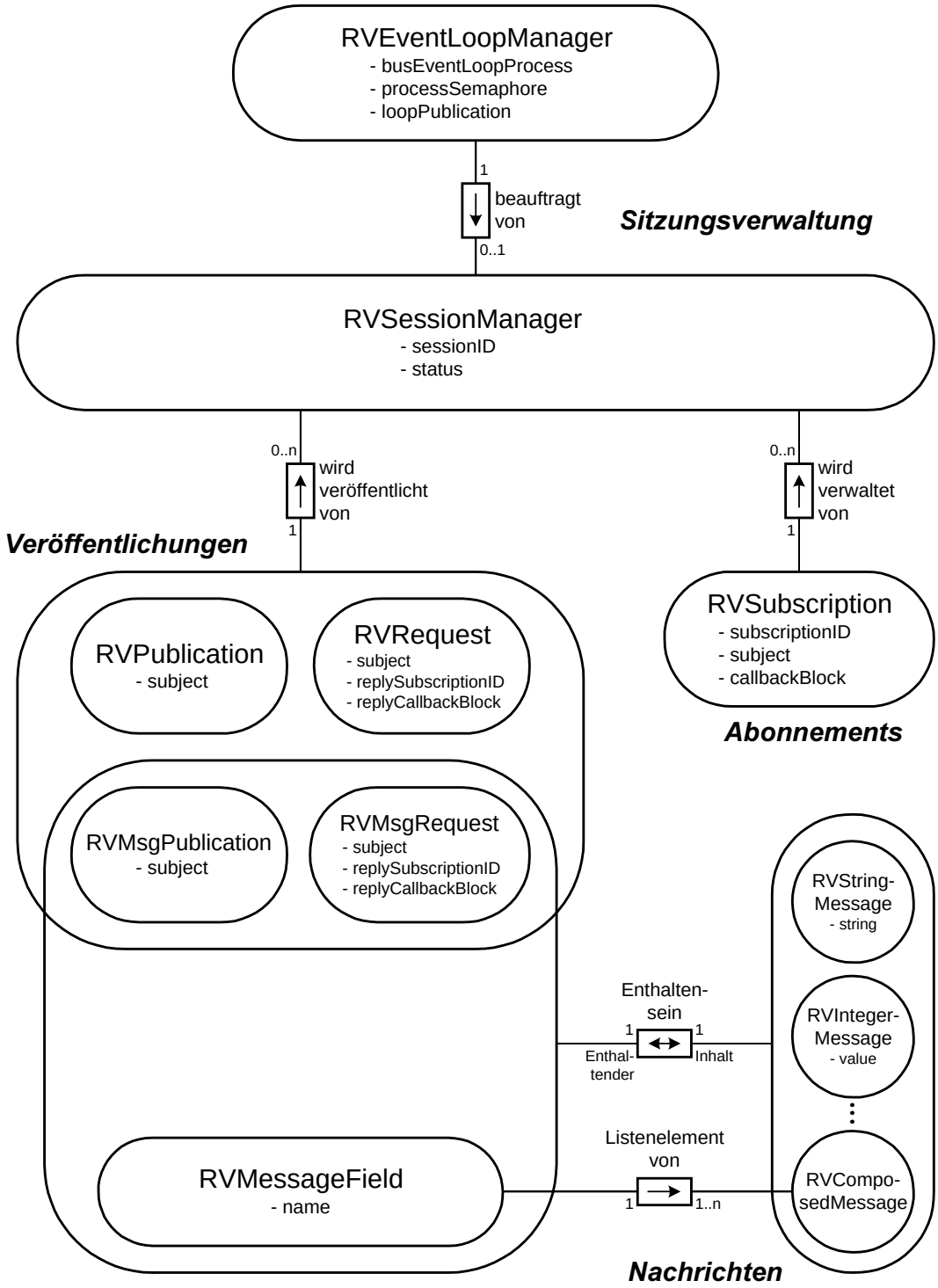


Bild 3.12 Smalltalk-Rendezvous-API: eingeführte Klassen

Die Hauptrolle im Smalltalk-Rendezvous-API spielt die Klasse **RVSessionManager** (Bild 3.12, oben). Exemplare dieser Klasse verwalten jeweils eine Bussitzung. Die Erzeugung solcher Sitzungsverwalter zieht unmittelbar das Öffnen einer Bussitzung und die Belegung

ihres Attributs ‘sessionID’ mit der erhaltenen Sitzungskennung nach sich. Um die jeweilige Smalltalk-Anwendung für Busereignisse empfangsbereit zu machen, bedient sich ein Sitzungsverwalter eines Eventloop-Managers (Exemplar der Klasse **RVEventLoopManager**), der innerhalb eines eigenen Smalltalk-Prozesses die Eventloop des externen Busakteurs aufruft. Eine Begründung für diese Architekturentscheidung wird Abschnitt 3.4.3 nachgereicht. Sitzungsverwalter verwalten in einem eigenen Containerobjekt die an die jeweilige Sitzung gebundenen Abonnements, repräsentiert durch Exemplare der Klasse **RVSubscription** (Bild 3.12, rechts in der Mitte). Diese Abonnement-Objekte tragen in ihren Attributen die jeweilige Abonnementkennung (‘subscriptionID’), das abonnierte Subject (‘subject’) und einen Verweis auf ein Blockobjekt (‘callbackBlock’), das jene Anweisungen enthält, die bei Auftritt eines Busereignisses unter dem abonnierten Subject ausgeführt werden sollen.

Ferner sind die Sitzungsverwalter diejenigen, die Veröffentlichungen auf den Bus geben. Veröffentlichungen werden repräsentiert durch Exemplare der Klassen **RVPublication**, **RVRequest**, **RVMsgPublication** und **RVMsgRequest** (Bild 3.12, links). Dabei wird einerseits unterschieden zwischen gewöhnlichen Veröffentlichungen und Auftragsveröffentlichungen, andererseits zwischen Veröffentlichungen, die Nachrichten enthalten, und nachrichtenfreien Veröffentlichungen. Tabelle 3.2 zeigt, welche Klasse für welchen Veröffentlichungstyp zuständig ist.

	ohne Nachricht	**mit Nachricht**
gewöhnliche Veröffentlichungen	RVPublication	RVMsgPublication
Auftrags-veröffentlichungen	RVRequest	RVMsgRequest

Tabelle 3.2 Veröffentlichungstypen und dafür zuständige Klassen

Zu allen Veröffentlichungstypen gehört ein Attribut, in dem das Subject enthalten ist, unter dem die Veröffentlichung auf den Bus gegeben werden soll. Daneben enthalten Auftragsveröffentlichungs-Objekte die Kennung des Rückmeldungsabonnements (‘replySubscriptionID’) und einen Verweis auf den Block, der bei Auftreten einer Rückmeldung ausgeführt werden soll (‘replyCallbackBlock’). Exemplare der Klassen RVMsgPublication und RVMsgRequest führen Nachrichten mit sich. Solche Nachrichten werden durch Objekte der Klassen **RVStringMessage**, **RVIntegerMessage** und **RVComposedMessage** (Bild 3.12, unten rechts) repräsentiert. RVStringMessage und RVIntegerMessage sind die beiden bisher implementierten Vertreter einfacher Nachrichtentypen in Smalltalk (vgl. RVMSG_STRING und RVMSG_INT in Tabelle 3.1). RVComposedMessage-Exemplare repräsentieren zusammengesetzte Nachrichten (RVMSG_RVMSG). Wie in Abschnitt 3.3.3 erwähnt, bestehen solche zusammengesetzten Nachrichten aus einer beliebig langen Liste von Nachrichtenfeldern. Die einzelnen Felder werden in Smalltalk durch Exemplare der Klasse **RVMessageField** repräsentiert (Bild 3.12, unten links). RVMessageField-Objekte enthalten neben dem Feldnamen (‘name’) wiederum ein Nachrichten-Objekt, also ein Exemplar der Klasse RVStringMessage, RVIntegerMessage oder RVComposedMessage.

3.4.2 Der Sitzungsverwalter

Sitzungsverwalter sind die zentralen Akteure innerhalb von Busanwendungen. Ohne einen Sitzungsverwalter ist eine Buskommunikation mit den Mitteln des Smalltalk-Rendezvous-API nicht möglich. Sitzungsverwalter sind Exemplare der Klasse RVSessionManager. Die Existenz eines Sitzungsverwalters ist gekoppelt an eine zugehörige Bussitzung. Die Eröffnung einer Bussitzung geschieht durch Erzeugung eines Sitzungsverwalter-Exemplars, das Schließen einer Bussitzung erfolgt durch Aufruf der Methode ‘terminate’ des entsprechenden Sitzungsverwalters. Nach Sitzungsschluß ist der zugehörige Sitzungsverwalter unbrauchbar. Innerhalb einer Smalltalk-Anwendung mit Busanschluß gehört die Erzeugung eines Sitzungsverwalters im allgemeinen zu den Initialisierungsaktionen.

Bild 3.13 zeigt den Aufbau eines solchen Anwendungssystems mit Busanschluß. Auf der rechten Seite befindet sich der Sitzungsverwalter, links die restlichen Akteure der Busanwendung, bestehend aus einem Basisanwendungsakteur und mehreren Callback-Akteuren, die durch Busereignisse unter abonnierten Subjects angestoßen werden. Die Erzeugung des Sitzungsverwalters durch den Basisanwendungsakteur ist zum dargestellten Zeitpunkt bereits geschehen. Anschließend können die Anwendungsakteure über die Auftrag-Rückmeldungs-Schnittstelle in der Bildmitte Methoden des Sitzungsverwalters aufrufen. Diese Methoden können jeweils einer von drei Gruppen zugordnet werden. Jede dieser Gruppen ist vertreten

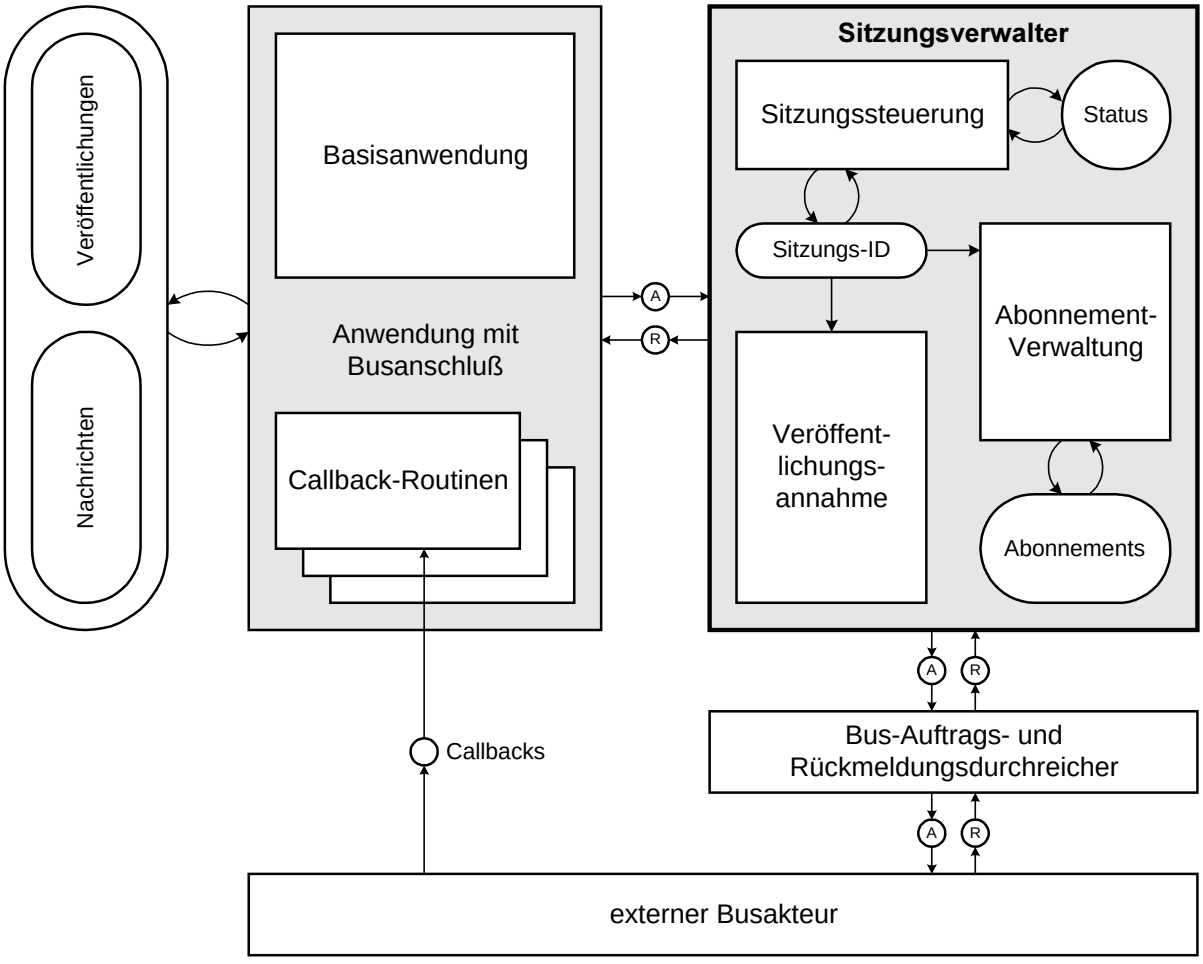


Bild 3.13 Sitzungsverwalter

durch einen eigenen Akteur innerhalb des Sitzungsverwalters. Im einzelnen sind dies der Sitzungssteuerungsakteur, der Veröffentlichungsannahmeakteur und Abonnementverwaltungsakteur. Alle drei können über den Bus-Auftrags- und Rückmeldungsdurchreicher (vgl. Bild 3.10) mit dem externen Busakteur kommunizieren. Sie werden im folgenden genauer vorgestellt.

Sitzungssteuerung

Der Sitzungssteuerungsakteur ist derjenige Akteur, der unmittelbar nach Erzeugung eines Sitzungsverwalter-Exemplars die Bussitzung eröffnet. Die dabei erhaltene Sitzungskennung (Sitzungs-ID) wird von den übrigen Akteuren des Sitzungsverwalters benötigt, um sich beim externen Busakteur zu identifizieren. Folgende öffentliche Methoden des Sitzungsverwalters liegen im Zuständigkeitsbereich des Sitzungssteuerungsakteurs:

- **startMainLoop**
  Veranlaßt die Erzeugung eines Eventloop-Managers (Exemplar der Klasse RVEventLoopManager) und beauftragt diesen anschließend, die Eventloop des externen Busakteurs aufzurufen. Der genaue Mechanismus wird im folgenden Abschnitt 3.4.3 dokumentiert.
- **terminate**
  Bewirkt die Beendigung der Bussitzung. Der Sitzungsverwalter ist anschließend unbrauchbar (tot).

Bei seinen Aktionen fährt der Sitzungssteuerungsakteur eine Status-Information (Status) gemäß des in Bild 3.14 gezeigten Zustandsübergangsgraphen nach.

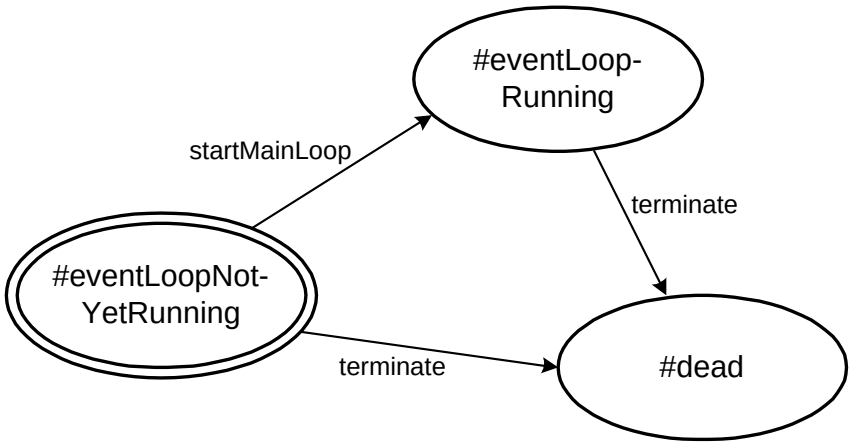


Bild 3.14 Status eines Sitzungsverwalters

Veröffentlichungsannahme

Veröffentlichungen und eventuell darin enthaltene Nachrichten werden von den Anwendungsakteuren in Smalltalk-Repräsentation erzeugt und zusammengebaut (Bild 3.13, links). Dabei werden die dazu innerhalb des Rendezvous-Smalltalk-API zur Verfügung gestellten Klassen genutzt (RVPublication, RVRequest, RVMsgPublication, RVMsgRequest, RVStringMessage, RVIntegerMessage und RVComposedMessage). Um die Veröffentlichungen dann auf den

Bus zu geben, werden sie dem Veröffentlichungsannahmeakteur des Sitzungsverwalters überreicht. Dies geschieht durch Aufruf der Sitzungsverwalter-Methode

- **publish:** <*aPublication*>.

Im einzigen Argument ‘aPublication’ wird ein Verweis auf das die Veröffentlichung repräsentierende Objekt übergeben. Die genaue Implementierung dieses Mechanismus wird in Abschnitt 3.4.3 vorgestellt.

Abonnementverwaltung

Anwendungsakteure können Subjects abonnieren, indem sie eine der folgenden Sitzungsverwalter-Methoden aufrufen:

- **subscribeToSubject:** <*subject*> **withCallback:** <*callbackBlock*>
- **subscribeToInboxWithCallback:** <*callbackBlock*>

‘SubscribeToSubject:WithCallback:’ bewirkt das Abonnement des im ersten Argument ‘subject’ spezifizierten Subjects. Bei Auftritt eines Busereignisses unter diesem Subject wird der durch das zweite Argument ‘callbackBlock’ identifizierte Smalltalk-Block ausgeführt. ‘SubscribeToInboxWithCallback:’ veranlaßt dagegen das Abonnement eines Inbox-Subjects. Dieses vom externen Busakteur generierte Inbox-Subject wird dem Aufrufer anschließend im Ergebnis-Objekt (Exemplar der Smalltalk-Klasse String) mitgeteilt. Das einzige Argument ‘callbackBlock’ enthält wie vorhin einen Verweis auf das Block-Objekt, dessen Anweisungen bei Auftritt eines Busereignisses unter dem abonnierten Subject ausgeführt werden sollen. Bei jedem Abonnement wird vom Abonnementverwaltungsakteur ein Exemplar der Klasse RVSubscription erzeugt und in dem in Bild 3.13 unter dem Abonnementverwaltungsakteur dargestellten Container abgelegt. In ein solches Abonnementobjekt trägt der Abonnementverwaltungsakteur die vom externen Busakteur erhaltene Abonnementkennung, das abonnierte Subject und den Verweis auf das Callback-Blockobjekt ein. So ist insbesondere gewährleistet, daß während der Gültigkeit eines Abonnements in jedem Fall ein Smalltalk-Verweis auf das zugehörige Callback-Blockobjekt existiert. Ein solcher Verweis ist unbedingt notwendig, damit das Callback-Blockobjekt nicht zwischenzeitlich von der Garbage-Collection zerstört wird. Abonnements sind kündbar durch Aufruf der Sitzungsverwalter-Methode

- **unsubscribeToSubject:** <*subject*>.

Im Argument wird das zu kündigende Subject angegeben. Der Abonnementverwaltungsakteur entnimmt dann dem entsprechenden Abonnementobjekt die zugehörige Abonnementkennung und beauftragt seinerseits den externen Busakteur zur Kündigung des Abonnements. Anschließend entfernt er das entsprechende Abonnementobjekt aus dem Container.

Es muß darauf hingewiesen werden, daß die hier vorgestellte Abonnementverwaltung nicht die infolge von Auftragsveröffentlichungen zustandekommenden Rückmeldungsabonnements betrifft. Solche Rückmeldungsabonnements werden im entsprechenden Auftragsveröffentlichungsobjekt (Exemplar der Klasse RVRequest oder RVMsgRequest, siehe Bild 3.12) verwaltet und sind kündbar durch deren Methode ‘closeReplyInbox’.

3.4.3 Das Eventloop-Problem und dessen Lösung

Das Problem

Busanwendungen, in denen Nachrichten empfangen werden können, sind ereignisgetrieben. Die treibenden Ereignisse sind in diesem Fall die durch Veröffentlichungen ausgelösten Busereignisse. Unabhängig davon sind auch alle Smalltalk-Anwendungen, die grafische Benutzerschnittstellen besitzen, ereignisgetrieben, und zwar durch die vom Anwendungsbenutzer ausgelösten sogenannten Windows-Ereignisse. Bei Busanwendungen mit grafischer Benutzerschnittstelle hat man also zwei unabhängige Ereignisquellen und dementsprechend zwei Ereigniskanäle zum Smalltalk-Laufzeitsystem (Bild 3.15). Über den oberen Kanal in Bild 3.15 fließen Benutzerereignisse wie zum Beispiel Mausklicks oder Meldungen bezüglich des Überschreitens bestimmter Grenzen auf dem Bildschirm mit dem Mauszeiger. Sie lösen den Aufruf entsprechender Reaktions-Methoden in Control-Objekten aus. Über den unteren Kanal fließen Busereignisse, die die Ausführung von Smalltalk-Callback-Routinen, repräsentiert durch Blockobjekte, verursachen. Für beide Ereigniseingänge existieren getrennte Eventloops, aus denen heraus die Aufrufe erfolgen. Die Eventloop zum Einfangen von Windows-Ereignissen befindet sich im Zuständigkeitsbereich des Windows-Betriebssystems, die Bus-Eventloop dagegen, wie bereits erwähnt, im Zuständigkeitsbereich des externen Busakteurs. Beide sind also aus der Sicht eines Smalltalk-Anwendungsentwicklers unzugänglich. Die gezeigten Ereigniskanäle sind daher nicht einfach verschmelzbar, indem man innerhalb der einen Eventloop den Schleifenrumpf der anderen als Prozedur aufruft. Es muß also bewerkstelligt werden, daß beide Eventloops unabhängig voneinander durchlaufen werden und sowohl Busereignisse als auch Benutzerereignisse an das Smalltalk-Laufzeitsystem weitergeleitet werden.

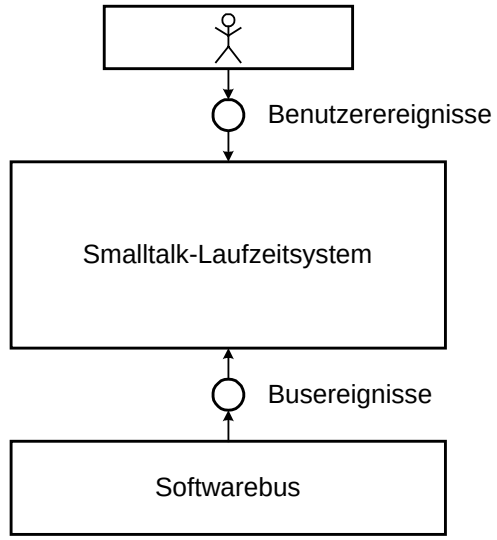


Bild 3.15 Benutzerereigniskanal und Busereigniskanal

Eine potentielle Lösungmöglichkeit für dieses Problems liegt auf der Hand: Man könnte die Bus-Eventloop innerhalb eines eigenen Smalltalk-Prozesses aufrufen, etwa in der Form

```
[ RVI rv_MainLoop: sessionID. ]
        forkAt: (Processor activePriority - 1).
```

‘RVI’ ist hier das Globalsymbol, über das der Bus-Auftrags- und Rückmeldungsdurchreicher zugänglich gemacht ist, ‘sessionID’ die Kennung einer offenen Bussitzung. Durch den Aufruf von ‘forkAt:’ mit dem gezeigten Argument ‘Processor activePriority – 1’ würde erreicht, daß die Laufpriorität des neuen Prozesses eine Stufe unterhalb der aktuellen, bei Aufruf gültigen Laufpriorität liegt. So könnte ausgeschlossen werden, daß der neue Prozeß den alten aufgrund des nichtpreemptiven Scheduling-Mechanismus innerhalb einer Prioritätsebene blockiert. Trotzdem würde dieser neue Smalltalk-Prozeß, sobald er den Smalltalk-Prozessor zugeteilt bekommt, sämtliche anderen Smalltalk-Prozesse – ungeachtet ihrer Prioritäten – für immer blockieren.

Dieses Verhalten ist darin begründet, daß der Aufruf einer Methode des Bus-Auftrags- und Rückmeldungsdurchreichers dasselbe darstellt wie der Aufruf einer Basismethode. Hier wie dort enthält der Methodenrumpf den Aufruf einer C-Routine, also einen Sprung ins Mikroprogramm des um die Fähigkeit der Buskommunikation erweiterten Smalltalk-Prozessors. Während der Ausführung von Mikroprogrammcode findet aber generell keine Prozeßumschaltung statt, also auch nicht während der Ausführung der C-Routine 'rv_MainLoop'. Da die Ausführung dieser Routine bis zum Ende der zugehörigen Bussitzung andauert, wird der Smalltalk-Prozessor über lange Zeit vollständig blockiert. Dies äußert sich für den Benutzer darin, daß die Werkzeuge der Smalltalk-Umgebung keinerlei Reaktionen mehr zeigen – das Smalltalk-System „hängt". Erst bei einem Callback durch den externen Busakteur infolge eines Busereignisses unter einem abonnierten Subject wird wieder für kurze Zeit Makroprogramm, d.h. die Smalltalk-Anweisungssequenz innerhalb des aufgerufenen Smalltalk-Blocks, ausgeführt. Während dieser Zeit ist die Smalltalk-Prozeßumschaltung wieder funktionstüchtig. Sobald jedoch der jeweilige Callback-Block abgearbeitet ist, stellt sich der alte Zustand wieder ein: Der Smalltalk-Prozessor scheint still zu stehen.

Die Lösung

Der Schlüssel zur Beseitigung dieses Mißstands liegt in dem zuletzt geschilderten Sachverhalt: Indem man dafür sorgt, daß häufig genug Busereignisse eintreffen und infolge dessen Smalltalk-Makroprogramm zur Ausführung kommt, lassen sich die dazwischen liegenden Mikroprogrammausführungsintervalle soweit reduzieren, daß die Smalltalk-Umgebung aus Sicht des Benutzers kein ungewöhnliches Verhalten mehr zeigt. Dies zu bewerkstelligen ist die Aufgabe des *Eventloop-Managers*.

Eventloop-Manager sind Exemplare der Klasse **RVEventLoopManager**. Sie werden bei Bedarf von Sitzungsverwaltern erzeugt. Bild 3.16 zeigt einen solchen Eventloop-Manager in Verbindung mit dem ihn erzeugenden Sitzungsverwalter als Bestandteil eines Smalltalk-Busanwendungssystems. Als weitere Akteure kommen vor:

- der Basis-Anwendungsakteur und die Callback-Akteure (vgl. Bild 3.13), hier zusammengefaßt zur „Anwendung mit Busanschluß" ganz links,
- der Bus-Auftrags- und Rückmeldungsdurchreicher (unten),
- der externe Busakteur (rechts unten) und
- der vom Eventloop-Manager erzeugte Busereignispollingakteur; er ist zuständig für die zyklische Auslösung der benötigten Busereignisse.

Im folgenden soll der Ablauf der Aktionen innerhalb des gezeigten Anwendungssystems beginnend mit dem Auftrag zum Start der Bus-Eventloop durch den Basisanwendungsakteur bis hin zur Beendigung der Bussitzung erklärt werden. Das zugehörige Petrinetz befindet sich am Ende dieser Schrift (Bild I). In diesem Petrinetz sind fünf Zuständigkeitsbereiche abgegrenzt: Aktionen der Anwendungsakteure, Sitzungsverwalter-Aktionen, Eventloop-Manager-Aktionen, Aktionen des Busereignispollingakteurs und Aktionen im Zuständigkeitsbereich des externen Busakteurs. Transitionen, die beim Schalten Busereignisse auslösen, sowie Transitionen, die durch Busereignisse getriggert werden, sind grau unterlegt dargestellt. Von links nach rechts laufende Kanten stellen Methodenaufrufe dar. Der jeweilige Methodenbe-

zeichner (bzw. im Falle des externen Busakteurs der Name der C-Funktion) ist kursiv unter die entsprechenden Kanten gedruckt. Diese Bezeichner findet man – neben weiteren – auch im Aufbaubild an den Auftrag-Rückmeldungs-Schnittstellen, über die die entsprechenden Methodenaufrufe erfolgen.

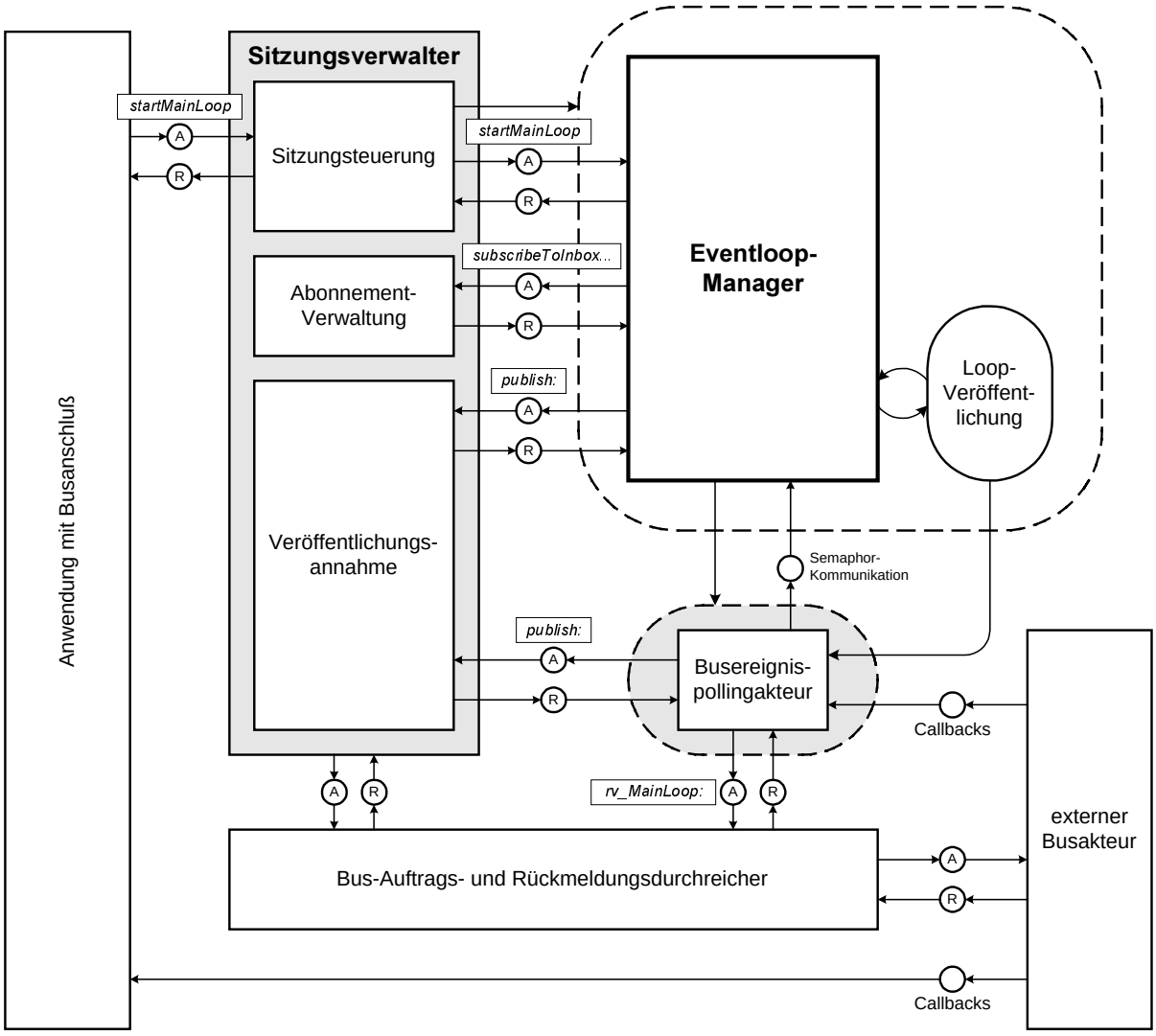


Bild 3.16 Der Eventloop-Manager innerhalb eines Smalltalk-Busanwendungssystems.

Um das Anwendungssystem für Busereignisse empfangsbereit zu machen, beauftragt der Basisanwendungsakteur über die Auftrag-Rückmeldungs-Schnittstelle oben links im Bild 3.16 die Sitzungssteuerung des Sitzungsverwalters zum Starten der Bus-Eventloop (startMain-Loop). Der Sitzungssteuerungsakteur erzeugt daraufhin einen neuen Eventloop-Manager und reicht den Startauftrag an diesen weiter (startMainLoop). In Bild 3.16 wird auf die bereits geschehene Erzeugung des Eventloop-Managers durch den Schreibpfeil vom Sitzungssteue-rungsakteur zum gestrichelt umrandeten Speicherbereich rechts oben hingewiesen. Der Event-loop-Manager veranlaßt nun als erstes das Abonnement eines Inbox-Subjects, indem er die Abonnementverwaltung des Sitzungsverwalters dazu beauftragt (subscribeToInbox...). An-schließend erzeugt er ein Veröffentlichungsobjekt (Exemplar der Klasse RVMsgPublication)

und trägt dort das erhaltene Inbox-Subject sowie die String-Nachricht „LOOP“ ein.[9] Dieses Objekt wird im folgenden Loop-Veröffentlichung genannt, es befindet sich in Bild 3.16 im Speicher rechts neben dem Eventloop-Manager. Unmittelbar nach der Erzeugung der Loop-Veröffentlichung beauftragt der Eventloop-Manager den Veröffentlichungsannahmeakteur, die Loop-Veröffentlichung erstmals auf den Bus zu geben, indem er die Sitzungsverwalter-Methode ‘publish:’ aufruft. Damit bewirkt er das Versenden einer Busnachricht an sich selbst, da er einziger Abonnent des Inbox-Subjects ist. Mit der Busveröffentlichung verbunden ist die Auslösung eines Busereignisses, welches aber noch nicht empfangen werden kann, da die Bus-Eventloop zu diesem Zeitpunkts noch nicht läuft. Die letzte Aktion des Eventloop-Managers vor Rückgabe der Kontrolle an den Basisanwendungsakteur ist der Start eines neuen Smalltalk-Prozesses mit derselben Laufpriorität, wie sie auch der Basisanwendungsprozeß besitzt. In Bild 3.16 wird dies angedeutet durch die Darstellung der Erzeugung des Busereignispolling-Akteurs. Genaugenommen verteilen sich die Aktionen innerhalb des neuen Smalltalk-Prozesses jedoch auf mehrere Akteure. Es sind dies:

- der Busereignispolling-Akteur,
- der externe Busakteur und
- die Callback-Akteure der Anwendung.

Der Busereignispolling-Akteur ist aber derjenige, der für die erste und die letzte Aktion innerhalb dieses Smalltalk-Prozesses zuständig ist. Die erste Aktion ist die Abgabe an die Bus-Eventloop durch Aufruf der Methode ‘rv_MainLoop’ des Bus-Auftrags- und Rückmeldungsdurchreichers. Dieser leitet den Aufruf weiter an den externen Busakteur. Der Durchreichevorgang ist in Bild I nicht explizit dargestellt. Dort verläuft die Aufruf-Kante direkt aus dem Zuständigkeitsbereich des Busereignispolling-Akteurs zur Bus-Eventloop ganz rechts. Die Bus-Eventloop wird getriggert durch Busereignisse, dargestellt durch die beiden grau unterlegten Transitionen mit der Aufschrift „Busereignis“. Im Schleifenrumpf werden vier Fälle bezüglich eines eingegangenen Busereignisses unterschieden:

1. Es handelt sich um ein Busereignis unter einem von einem Anwendungsakteur abonnierten Subject (linker Pfad): Dann wird der mit dem Abonnement verbundene Callback-Block im Zuständigkeitsbereich der Anwendungsakteure aufgerufen. Dies geschieht über den unteren Callback-Kanal in Bild 3.16.

2. Es handelt sich um das durch die Loop-Veröffentlichung ausgelöste Loop-Inbox-Ereignis (mittlerer Pfad): In diesem Fall erfolgt ein Rückruf des bei Abonnement des Loop-Inbox-Subjects durch den Eventloop-Manager angegebenen Callback-Blocks. Dieser Block fällt in der gewählten Modellierung in den Zuständigkeitsbereich des Busereignispolling-Akteurs, weshalb in Bild 3.16 auch ein zweiter Callback-Kanal zum Busereignispolling-Akteur eingezeichnet ist. Der Busereignispolling-Akteur veranlaßt daraufhin zunächst explizit die Abgabe des Smalltalk-Prozessors und die Verschiebung des Prozesses, innerhalb dessen er operiert, ans Ende der zugehörigen Prozeßwarteschlange. Da der Basisanwendungsprozeß dieselbe Laufpriorität besitzt, ist so garantiert, daß vor Rückgabe der Marke

[9] Der Nachrichteninhalt ist für die weiteren Ausführungen ohne Bedeutung.

an den externen Busakteur die weiteren Akteure des Anwendungssystems zum Zuge kommen. Nachdem der Busereignispolling-Akteur den Prozessor wieder hat, beauftragt er den Veröffentlichungsannahmeakteur des Sitzungsverwalters via ‘publish:’, dafür zu sorgen, daß die Loop-Veröffentlichung erneut auf den Bus gegeben wird, und gibt zurück an den externen Busakteur.

3. Es handelt sich um ein Ereignis unter einem nicht abonnierten Subject (rechter Pfad): Aus Sicht des Busanwendungssystems passiert nichts weiter.

4. Es handelt sich um das Sitzungsschlußereignis (unterer Pfad): Die Bus-Eventloop wird verlassen.

Man erinnere sich, daß während der Aktionen des externen Busakteurs, die ja durch Ausführung von Mikroprogramm-Code zustandekommen, der Smalltalk-Scheduling-Mechanismus ausgeschaltet ist. Erst durch einen Rückruf (Callback) an den Busereignispolling-Akteur oder die Callback-Akteure der Anwendung wird der Scheduler für die Zeit der Callback-Routinenausführung wieder aktiv. Daß ein solcher Rückruf sofort nach Eintritt in die Bus-Eventloop erfolgt, dafür sorgt der Eventloop-Manager schon bei seinen ersten Aktionen. Dort wird bereits vor Aufruf der Bus-Eventloop erstmals ein Loop-Inbox-Ereignis ausgelöst. Da die damit verbundene Ereignismeldung bis zum Start der Bus-Eventloop gepuffert wird, wird dieses Busereignis gleich als erstes empfangen und dementsprechend der Busereignispolling-Akteur zurückgerufen (Fall 2 von oben). Da der Busereignispollingakteur, nachdem er den Prozessor wieder hat, erneut ein Loop-Inbox-Ereignis auslöst, erhält man einen Zyklus aus Ereignis-Auslösung und Callback. So wird verhindert, daß eine Marke für längere Zeit im Zuständigkeitsbereich des externen Busakteurs verbleibt. Die Mikroprogrammausführungsintervalle werden also wie gewünscht minimiert.

Der Zyklus läßt sich im Bild 3.16 entgegen dem Uhrzeigersinn zwischen externem Busakteur, Busereignispollingakteur, Veröffentlichungsannahme des Sitzungsverwalters und Bus-Auftrags- und Rückmeldungsdurchreicher verfolgen: Durch das erstmalige Veröffentlichen der Loop-Veröffentlichung wird der Busereignispollingakteur kurze Zeit nach seiner Erzeugung erstmals zurückgerufen. Daraufhin veranlaßt er die erneute Veröffentlichung durch Beauftragung des Veröffentlichungsannahmeakteurs (publish:). Dieser setzt die Smalltalk-Repräsentation der Veröffentlichung um in die erforderliche C-Repräsentation und beauftragt den Bus-Auftrags- und Rückmeldungsdurchreicher zum Weiterleiten der Veröffentlichung an den externen Busakteur. Dadurch wird dort erneut ein Loop-Inbox-Ereignis ausgelöst, welches wiederum den Rückruf des Busereignispolling-Akteurs nach sich zieht.

Daß diese Rückkopplung den Nutzereignisfluß nicht behindert, wird dadurch erreicht, daß in jedem Zyklus der Prozessor abgegeben wird, um anderen Akteuren die Möglichkeit zu geben, Veröffentlichungen zu erzeugen und auf den Bus zu geben. In diesem Zusammenhang muß auch erwähnt werden, daß die Loop-Veröffentlichungen keine zusätzliche Netzwerkbelastung verursachen. Der externe Busakteur erkennt Nachrichten, bei denen Adresse und Absender übereinstimmen, und läßt sie erst gar nicht bis auf Netzwerkebene durch.

Sitzungsbeendigung

Es folgt nun die noch ausstehende Behandlung des Ablaufs bei Beendigung einer Bussitzung. Wie bereits erwähnt, kann ein Akteur der Anwendung – ein Callback-Akteur oder aber der Basisanwendungsakteur – das Schließen einer Bussitzung durch Aufruf der Methode 'terminate' des zugehörigen Sitzungverwalters veranlassen. Der Sitzungssteuerungsakteur des Sitzungsverwalters fordert daraufhin die Beendigung der Bussitzung via 'rv_Term' beim Bus-Auftrags- und Rückmeldungsdurchreicher unter Angabe seiner Sitzungskennung an. Der Bus-Auftrags- und Rückmeldungsdurchreicher gibt diesen Aufruf weiter, worauf beim externen Busakteur ein Sitzungsschlußereignis ausgelöst wird. Als Folge davon wird die Bus-Eventloop verlassen und die zugehörige Prozeß-Marke endgültig nach Smalltalk zurückgegeben. Erst jetzt wird jene Smalltalk-Anweisungen ausgeführt, die nach dem Aufruf der Bus-Eventloop durch 'rv_MainLoop' folgt. Dies ist gleichzeitig die letzte Aktion des Busereignis-polling-Akteurs: Es wird ein Semaphorplatz belegt und der Prozeß, innerhalb dessen der Eventloop-Aufruf stattfand, beendet. Der Sitzungssteuerungsakteur wartet die Belegung dieses Semaphorplatzes ab (waitForTermination) und setzt anschließend seinen Status auf '#dead' (vgl. Bild 3.14). Der Sitzungsverwalter ist damit unbrauchbar geworden. Die Anwendungsakteure sollten daher ihre Verweise auf ihn löschen und ihn der Garbage-Collection überlassen.

Der in Bild I dargestellte Ablauf bei Beendigung einer Bussitzung gilt so nur für den Fall, daß vor Aufruf von 'terminate' die Bus-Eventloop bereits durch 'startMainLoop' in Gang gesetzt wurde. Wurde die Eventloop voher nicht gestartet, wird auch nicht auf deren Beendigung gewartet. Die entsprechende Fallunterscheidung führt der Sitzungssteuerungsakteur anhand seines Status-Eintrags durch.

Ferner muß folgende Einschränkung bezüglich des dargestellten Semaphor-Synchronisationsmechanismus gemacht werden: Er funktioniert dann nicht, wenn im Smalltalk-Objektspeicher mehrere aktive Sitzungsverwalter existieren, bei denen ein Eventloop-Startauftrag bereits erteilt wurde. Dies hängt mit einer Verwendungs-Einschränkung des für VISUALWORKS verfügbaren Smalltalk-C-Interfaces zusammen (siehe [VWDLLCC 94, Seite 81]). Um dennoch mehrere Sitzungsverwalter mit laufender Bus-Eventloop zuzulassen, kann man den Synchronisationsmechanismus global an- und abschalten. Dazu ruft man folgende Klassenmethoden des RVSessionManager-Klassenobjekts auf:

- **setTerminationModeToRough**
  Schaltet den Synchronisationsmechanismus ab. Bei Sitzungsbeendigungen wird nun nicht mehr via 'waitForTermination' auf das Eintreffen eines Semaphors gewartet, sondern durch Aufruf der Eventloop-Manager-Methode 'killEventLoopProcess' der Abbruch des Prozesses, innerhalb dessen die Bus-Eventloop aufgerufen wurde, veranlaßt.
- **setTerminationModeToSynchronized**
  Schaltet den Synchronisationsmechanismus wieder ein.

Der aktuell gültige Sitzungsbeendigungs-Modus wird im Klassenattribut 'TerminationMode' der Klasse RVSessionManager gespeichert. Bild 3.17 zeigt den zugehörigen Zustandsübergangsgraph.

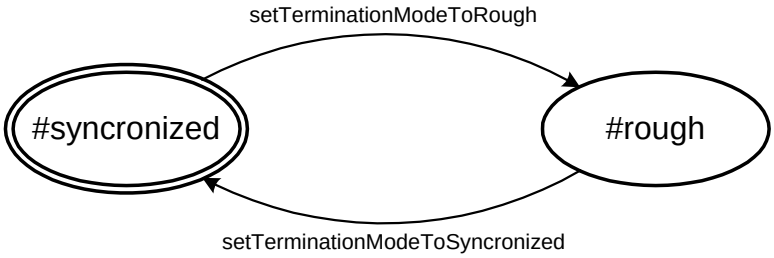


Bild 3.17 Sitzungsbeendigungs-Modi

Der genaue Ablauf der Sitzungsbeendigungsprozedur des Sitzungssteuerungsakteurs sieht daher wie folgt aus (Bild 3.18):

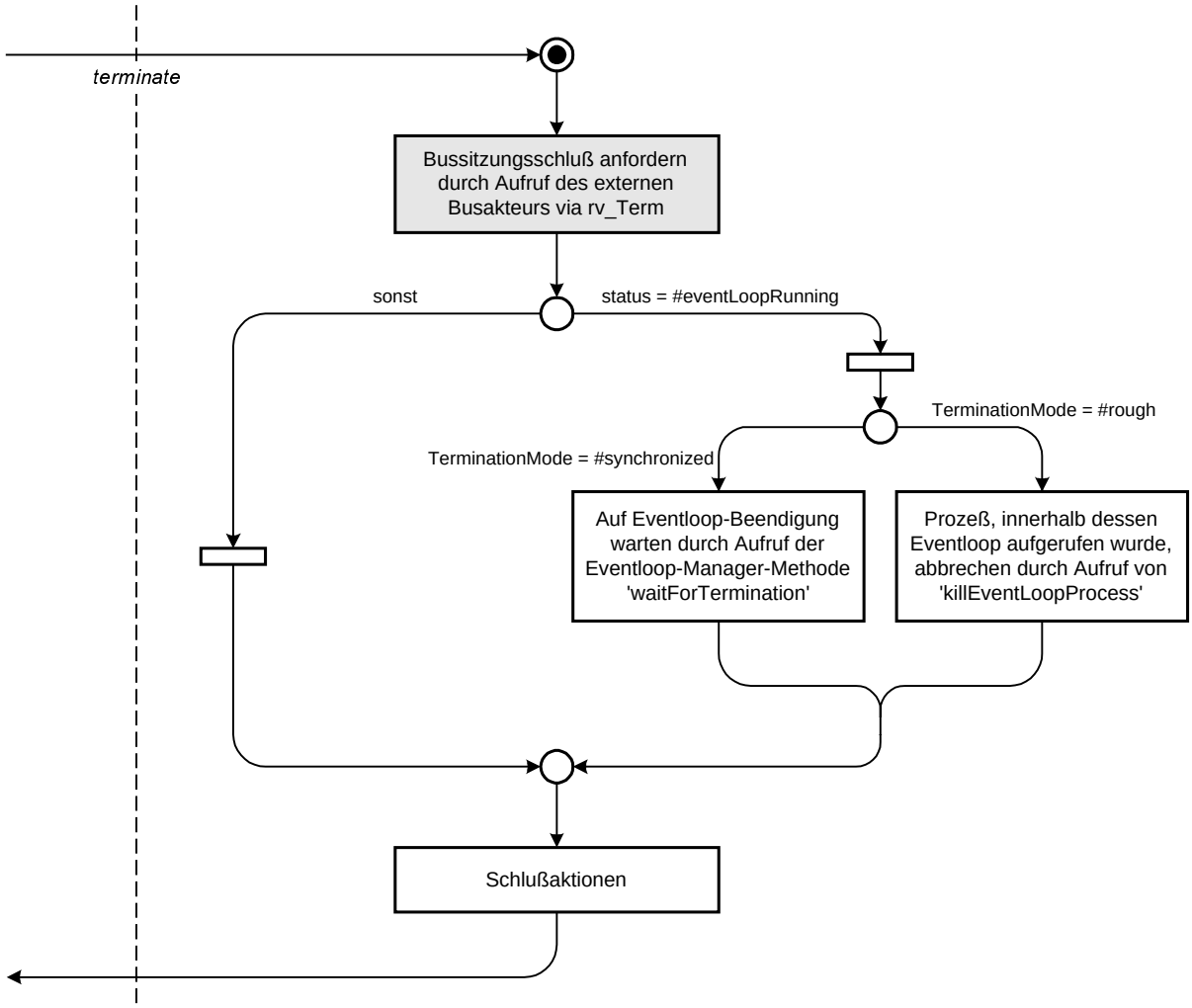


Bild 3.18 Sitzungsbeendigungsprozedur des Sitzungsverwalters

3.4.4 Veröffentlichungen im Detail

In den vorigen beiden Abschnitten wurde der Eindruck erweckt, als könne der Sitzungsverwalter zusammen mit dem Eventloop-Manager sämtliche Operationen ausführen, die zur Buskommunikation erforderlich sind. Exemplare der übrigen Klassen des Smalltalk-Rendezvous-

API – also derjenigen Klassen, die Veröffentlichungen, Nachrichten und Abonnements betreffen (siehe Bild 3.12) – scheinen nichts anderes als spezielle Datencontainer zu sein. Dies ist zwar durchaus die richtige Sichtweise für *Benutzer* des Smalltalk-API, in der nun folgenden Dokumentation der Implementierung der Veröffentlichungsoperationen muß sie aber revidiert werden.

Bild 3.19 zeigt daher noch einmal die an einer Busveröffentlichung beteiligten Smalltalk-Akteure in einer präzisierten Darstellung. Die Anwendungsakteure oben rechts bauen in dem unter ihnen liegenden Speicherbereich Smalltalk-Objekte zusammen, die Veröffentlichungen mit eventuell darin enthaltenen Nachrichten repräsentieren. Dabei werden diese Objekte eher als Daten-Container, denn als Akteure betrachtet. Anschließend können die Anwendungsakteure die Veröffentlichung auf dem Softwarebus veranlassen, indem sie über die Auftrag-Rückmeldungs-Schnittstelle oben links die Sitzungsverwalter-Methode ‘publish:’ aufrufen. Dabei übergeben sie dem Veröffentlichungsannahme-Akteur des Sitzungsverwalters einen Verweis auf das die Veröffentlichung repräsentierende Objekt. Im Unterschied zur bisherigen Darstellung übergibt der Veröffentlichungsannahme-Akteur den Inhalt des Objektes aber nun nicht selbst via Bus-Auftrags- und Rückmeldungsdurchreicher dem externen Busakteur, sondern beauftragt seinerseits das Veröffentlichungsobjekt, dies zu tun. Dazu ruft er über die untere Auftrag-Rückmeldungs-Schnittstelle die Methode ‘publishInSessionWithID:’ des Veröffentlichungsobjekts auf und übergibt als Argument seine Sitzungskennung (Sitzungs-ID). Denn im Sinne der Objektorientierung „weiß“ das jeweilige Veröffentlichungsobjekt, das

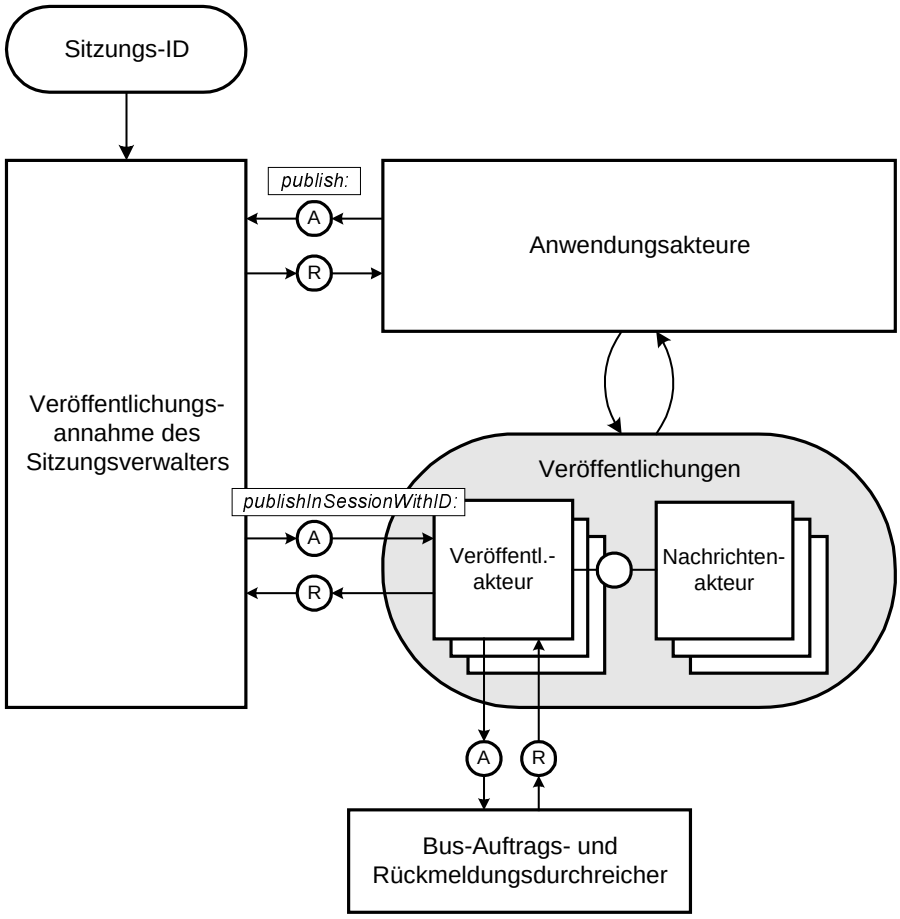


Bild 3.19 Veröffentlichungen: Benutzersicht und Implementierung

ein Exemplar der Klasse RVPublication, RVMsgPublication, RVRequest oder RVMsgRequest sein kann, selbst am besten, wie es seinen Inhalt in C-Repräsentation umzusetzen hat und welche Buskommandos des externen Busakteurs aufgerufen werden müssen. Ab diesem Zeitpunkt tritt also das jeweilige Veröffentlichungsobjekt als Akteur auf. Da auch die Nachrichtenobjekte im Verlauf der Konvertierung in die C-Repräsentation als Akteure auftreten, sind sowohl Veröffentlichungsobjekte als auch Nachrichtenobjekte in Bild 3.19 im Speicher unterhalb der Anwendung in Form von kommunizierenden Akteuren eingezeichnet.

In den folgenden Abschnitten wird nun der genaue Ablauf, angefangen vom ‘publish:’-Auftrag des Anwendungsakteurs bis zur vollständigen Übergabe der Veröffentlichung an den externen Busakteur, erläutert. Dabei werden drei Fälle unterschieden:

1. Nachrichtenfreie Veröffentlichungen, repräsentiert durch Exemplare der Klassen RVPublication und RVRequest.

2. Veröffentlichungen mit Nachrichten einfachen Typs, repräsentiert durch Exemplare der Klassen RVMsgPublication und RVMsgRequest, deren Attribut ‘message’ einen Verweis auf ein Exemplar der Klassen RVStringMessage oder RVIntegerMessage enthält.

3. Veröffentlichungen mit zusammengesetzten Nachrichten. Hier enthält das ‘message’-Attribut einen Verweis auf ein Exemplar der Klasse RVComposedMessage.

Das zugehörige Petrinetz ist in Bild II am Ende dieser Schrift dargestellt. Dort sind für die beteiligten Akteure Zuständigkeitsbereiche abgegrenzt. Transitionen, die den Aufruf eines Buskommandos des externen Busakteurs darstellen, sind grau eingefärbt.

3.4.4.1 Nachrichtenfreie Veröffentlichungen

Bild 3.20 zeigt die bei nachrichtenfreien Veröffentlichungen nach Aufruf von ‘publish:’ aktiven Smalltalk-Akteure. In Bild II lassen sich deren Aktionen verfolgen, indem man im Zuständigkeitsbereich des Veröffentlichungsakteurs dem linken Pfad folgt. Zunächst gibt der Veröffentlichungsannahme-Akteur des Sitzungsverwalters den Veröffentlichungsauftrag inklusive seiner eigenen Sitzungskennung weiter an den Veröffentlichungsakteur (‘publishInSessionWithID: sessionID’). Anschließend ist der Veröffentlichungsakteur bereits in der Lage, die Veröffentlichung über den Bus-Auftrags- und Rückmeldungsdurchreicher auf den Bus zu geben. Dazu ruft er entweder das Buskommando ‘rv_Send’ (gewöhnliche Veröffentlichungen) oder ‘rv_Rpc’ (Auftragsveröffentlichungen) auf und gibt dabei neben der Sitzungskennung das in seinem Attribut ‘subject’ eintragene Subject an. Da in diesen Kommandos immer eine Nachricht

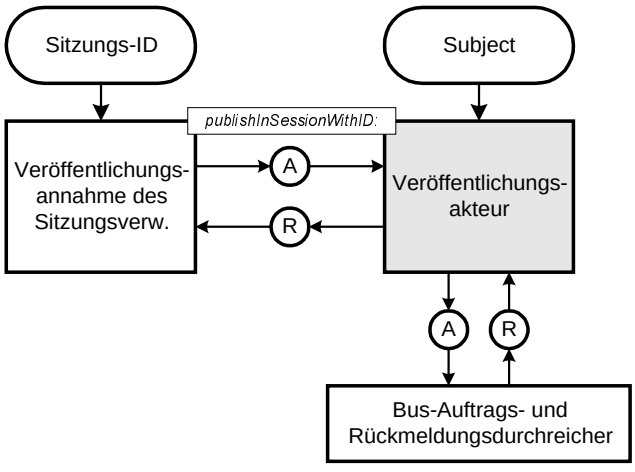


Bild 3.20 Nachrichtenfreie Veröffentlichungen

spezifiziert werden muß, wird ersatzweise eine leere Stringnachricht angegeben. Im Falle einer nachrichtenfreien, gewöhnlichen Veröffentlichung lautet der Busbefehlsaufruf daher:

```
RVI rv_Send: sessionID
        with: subject
        "leere String-Message:"
        with: RVMSG_STRING        "Typ"
        with: 0                   "Länge"
        with: ''.                 "Inhalt"
```

3.4.4.2 Veröffentlichungen mit Nachrichten einfachen Typs

Bei Veröffentlichungen mit Nachrichten einfachen Typs kommt zusätzlich das die einfache Nachricht repräsentierende Objekt – ein Exemplar der Klasse RVStringMessage oder RVIntegerMessage – ins Spiel (siehe Bild 3.21). Den zugehörigen Ablauf vom ‘publish:’-Auftrag bis zur eigentlichen Veröffentlichung findet man in Bild II, indem man im Zuständigkeitsbereich des Veröffentlichungsakteurs dem rechten Pfad und im Zuständigkeitsbereich des Nachrichtenakteurs dem linken Pfad folgt. Der Ablauf unterscheidet also kaum von dem bei nachrichtenfreien Veröffentlichungen. Es wird lediglich zusätzlich die Methode ‘prepareForPublicationInSessionWithID:’ des Nachrichtenobjekts aufgerufen, bevor die eigentliche Veröffentlichung stattfindet. In der durch diesen Methodenaufruf angestoßenen Prozedur passiert allerdings nichts. Der Aufruf ist trotzdem erforderlich, da der Veröffentlichungsakteur nicht wissen kann, ob das Objekt, auf das sein Attribut ‘message’ zeigt, eine einfache oder eine zusammengesetzte Nachricht repräsentiert. Und im Falle einer zusammengesetzten Nachricht ist eine Vorbereitung des die Nachricht repräsentierenden Objektverbundes notwendig, wie im folgenden Abschnitt 3.4.4.3 gezeigt wird.

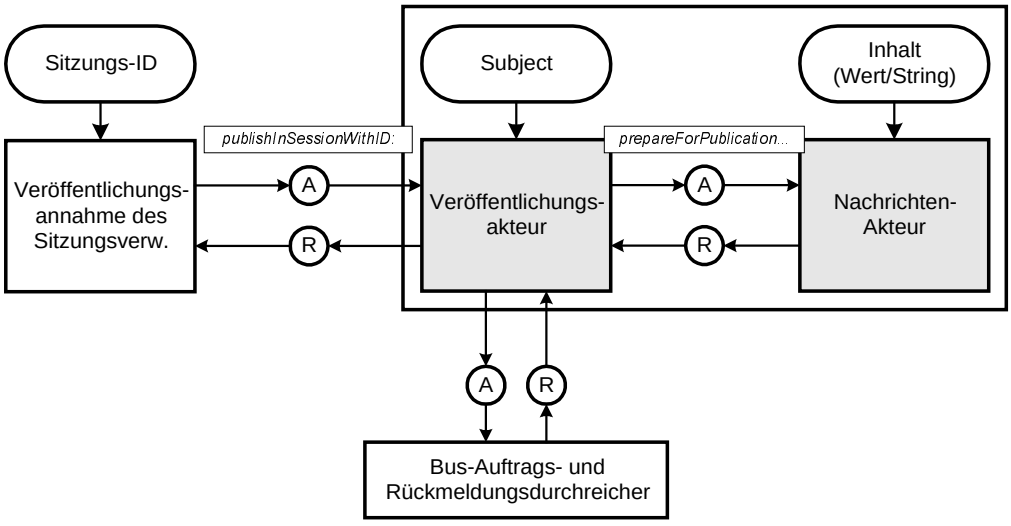


Bild 3.21 Veröffentlichungen mit Nachrichten einfachen Typs

Der Aufruf des Buskommandos rv_Send bzw. rv_Rpc unterscheidet sich von dem bei nachrichtenfreien Veröffentlichungen dahingehend, daß innerhalb der Anweisung an den Busauftrags- und Rückmeldungsdurchreicher durch Zugriff auf das Nachrichtenobjekt das die Nachricht beschreibende (Typ, Länge, Inhalt)-Tripel abgefragt wird. Dazu stellen sämtliche Nachrichten-Klassen die drei Methoden ‘type’, ‘size’ und ‘data’ zur Verfügung, mit denen ihre C-

Repräsentation abgefragt werden kann. Im Falle einer gewöhnlichen Veröffentlichung, die eine Nachricht enthält, sieht der ‘rv_Send’-Aufruf daher folgendermaßen aus:

```
RVI rv_Send: sessionID
        with: subject
        with: message type
        with: message size
        with: message data.
```

3.4.4.3 Veröffentlichungen mit zusammengesetzten Nachrichten

Nachrichtenzusammenbau und Feldzugriff

Bevor die Veröffentlichungsprozedur für zusammengesetzte Nachrichten vorgestellt wird, soll zunächst erklärt werden, wie man solche Nachrichten durch Nutzung des Smalltalk-Rendezvous-API zusammenbauen kann: Als erstes muß ein Objekt der Klasse RVComposedMessage erzeugt werden. Ein solches Objekt besitzt eine Methode, mit deren Hilfe anschließend Nachrichtenfelder erzeugt und angehängt werden können:

- **appendMessage:** *<aMessage>* **withName:** *<fieldName>*

Der Aufruf von ‘appendMessage:withName:’ bewirkt zunächst die Erzeugung eines Nachrichtenfeldobjektes (Exemplar der Klasse RVMessageField), in welches das im ersten Argument übergebene Nachrichtenobjekt (‘aMessage’), sowie der im zweiten Argument spezifizierte Feldname (‘fieldName’) eingetragen wird. Anschließend wird dieses Nachrichtenfeldobjekt an die Liste der zur Nachricht gehörenden Nachrichtenfelder angehängt. Diese Liste ist realisiert durch ein Exemplar der Smalltalk-Klasse List, welches an das Attribut ‘fieldList’ des die zusammengesetzte Nachricht repräsentierenden RVComposedMessage-Exemplars gebunden ist. Nachdem auf diese Weise eine zusammengesetzte Nachricht vollständig zusammengebaut worden ist, kann sie an das Attribut ‘message’ eines Veröffentlichungsobjekts (ein Exemplar der Klasse RVMsgPublication oder RVMsgRequest) gebunden werden.

Möchte man auf einzelne Felder einer zusammengesetzten Nachricht in Smalltalk-Repräsentation zugreifen, kann man sich einer der folgenden Methoden des jeweiligen RVComposed-Message-Exemplars bedienen:

- **getMessageByName:** *<name>*
  Gibt das Nachrichtenobjekt aus dem ersten Feld in der Liste, das den im Argument ‘name’ angegebenen Namen trägt, zurück; vergleichbar mit ‘rvmsg_Get’ (siehe Abschnitt 3.3.3).
- **fieldsDo:** *<aBlock>*
  Veranlaßt die Ausführung der im Argument angegebenen Bearbeitungs-Prozedur (repräsentiert durch ein Blockobjekt) für jedes Feld in der Liste. Der angegebene Smalltalk-Block muß genau eine Argumentvariable besitzen. In ihr wird bei jedem Durchlauf das jeweilige Nachrichtenfeld-Objekt übergeben (vergleichbar mit ‘rvmsg_Apply’).

Wichtig ist, daß sowohl beim Zusammenbau als auch beim Feldzugriff der externe Busakteur nicht beteiligt ist. Die dargestellten Operationen beziehen sich ausschließlich auf die Nachrichtenrepräsentation im Smalltalk-Objektspeicher.

Veröffentlichung

Die Veröffentlichungsprozedur bei zusammengesetzten Nachrichten gestaltet sich daher etwas komplizierter. Man erinnere sich (vgl. Abschnitt 3.3.3), daß zusammengesetzte Nachrichten in C unter Nutzung des Message-API innerhalb einer Sitzung zusammengebaut werden müssen: Zuerst muß ein C-Bytearray passender Größe initialisiert und beim externen Busakteur angemeldet werden, anschließend füllt man dieses Array via ‘rvmsg_Append’ sukzessive mit Nachrichtenfeldern. Baut man dagegen in Smalltalk – wie oben gezeigt – unter Nutzung der Klassen des Rendezvous-Smalltalk-API zusammengesetzte Nachrichten auf, findet zur Nachrichten-Bauzeit keine Kommunikation mit dem externen Busakteur statt. Die Umsetzung der Smalltalk-Repräsentation in die erforderliche C-Repräsentation geschieht erst unmittelbar vor Veröffentlichung der zusammengesetzten Nachricht durch Aufruf der Methode ‘prepareForPublicationInSessionWithID:’ des entsprechenden RVComposedMessage-Exemplars. Anschließend liefern die bereits erwähnten Methoden ‘type’, ‘size’ und ‘data’ das die Nachricht beschreibende Tripel in der gewünschten Form. Den Ablauf dazu findet man im Bild II, indem man im Zuständigkeitsbereich des Nachrichtenakteurs dem rechten Pfad folgt. Bild 3.22 zeigt die beteiligten Akteure.

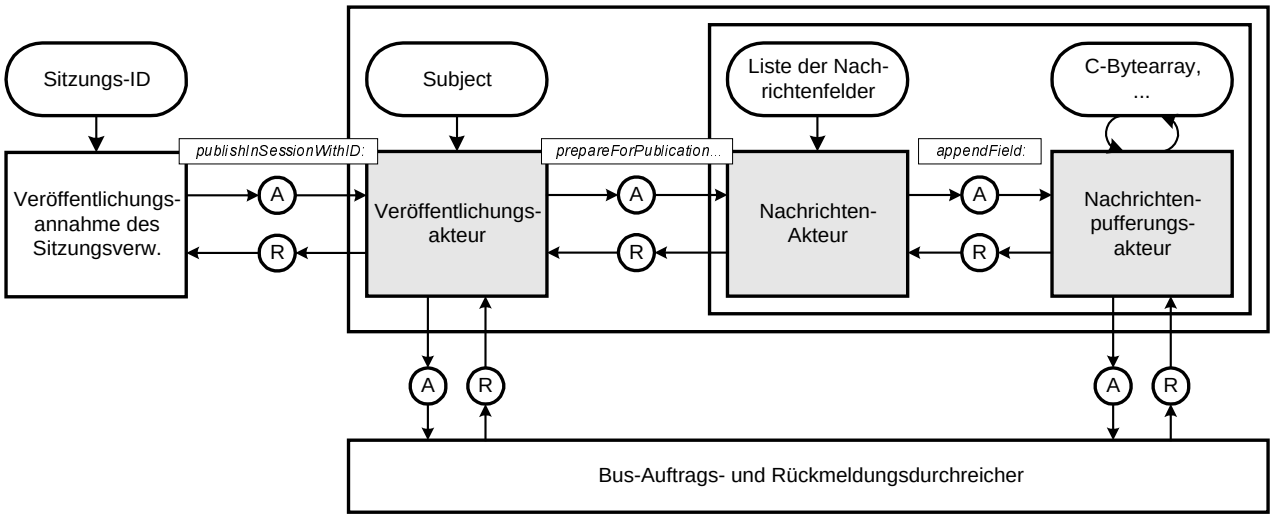


Bild 3.22 Veröffentlichungen mit zusammengesetzten Nachrichten

Als erstes bestimmt der (RVComposedMessge-)Nachrichtenakteur die erforderliche Größe des Bytearrays, das später die C-Repräsentation der zusammengesetzten Nachricht enthalten soll. Da zusammengesetzte Nachrichten wiederum zusammengesetzte Nachrichten enthalten können, muß dabei der gesamte Hierarchiebaum der Nachricht rekursiv durchgearbeitet werden. Sämtliche Nachrichtenklassen stellen dazu die Methode ‘sizeInBuffer’ zur Verfügung, die bezüglich einer Nachricht die Anzahl der im Array benötigten Bytes angibt. Ist die vollständige Länge bestimmt, erzeugt der Nachrichtenakteur einen Nachrichtenpufferungsakteur als Exemplar der Klasse **RVMessageBuffer** (eine bisher unerwähnt gebliebene Klasse des Smalltalk-Rendezvous-API) und beauftragt diesen, ein C-Bytearray der berechneten Größe zu

reservieren und beim externen Busakteur anzumelden. Anschließend übergibt der Nachrichtenakteur dem Nachrichtenpufferungsakteur nacheinander jedes Nachrichtenfeldobjekt aus seiner Liste der Nachrichtenfelder. Dazu ruft er zyklisch dessen Methode ‘appendField:’ auf, wobei er im Argument einen Verweis auf das jeweilige Nachrichtenfeldobjekt angibt.

Der Nachrichtenpufferungakteur fügt nun Nachrichtenfeld für Nachrichtenfeld in das von ihm initialisierte C-Bytearray ein. Er ruft dazu für jedes Feld das Buskommando ‘rvmsg_Append’ auf und übergibt den jeweiligen Feldnamen und das (Typ, Länge, Inhalt)-Tripel, das die im Feld enthaltene Nachricht beschreibt:

```
RVI rvmsg_Append: sessionID        "Sitzungskennung"
        with: cBuffer              "Puffer-Array"
        with: cBufferSize          "Puffergröße"
        with: msgField name        "Feldname"
        "Nachricht:"
        with: message type         "Typ"
        with: message size         "Länge"
        with: message data.        "Inhalt"
```

Dabei macht er von den drei Methoden ‘type’, ‘size’ und ‘data’ des entsprechenden Nachrichtenobjekts Gebrauch. Da jedoch die im Feld enthaltene Nachricht wiederum eine zusammengesetzte Nachricht sein kann, muß er vor Aufruf dieser Methoden die Vorbereitung des Nachrichtenobjekts auf die Veröffentlichung veranlassen. Es geschieht daher zuvor ein erneuter – rekursiver – Aufruf von ‘prepareForPublicationInSessionWithID:’ beim im Nachrichtenfeld enthaltenen Nachrichtenobjekt.

Nachdem alle Felder in das C-Bytearray übertragen wurden, kopiert sich der Nachrichtenakteur dieses Bytearray in ein eigenes Attribut und überläßt den Nachrichtenpufferungsakteur der Garbage-Collection. Seine Methoden ‘data’, ‘size’ und ‘type’ liefern nun bei Aufruf durch den Veröffentlichungsakteur dieses C-Bytearray, seine Länge und die erforderliche Typangabe (RVMSG_RVMSG) zurück.

Dem Leser stellt sich eventuell die Frage, warum die C-Repräsentation einer zusammengesetzten Nachricht nicht bereits bei Erzeugung der entsprechenden Smalltalk-Nachrichtenobjekte gleich „mit“ erzeugt wird, statt unmittelbar vor der Veröffentlichung einen unter Umständen recht umfangreichen Konvertierungsmechanismus in Gang zu setzen. Die Entscheidung für letzteres fiel aus zwei Gründen:

1. Im Vorhinein läßt sich die erforderliche Länge des zur Aufnahme einer zusammengesetzten Nachricht erforderlichen C-Bytearrays nicht exakt bestimmen sondern nur schätzen. Fügte man einer zusammengesetzten Nachricht ein Feld an, dessen C-Repräsentation in das bereits initialisierte Bytearray nicht mehr hineinpassen würde, müßte man dieses vollständig in ein größeres umkopieren. Dieser Vorgang könnte sich bezüglich einer Nachricht mehrmals wiederholen.

2. Der Bau von zusammengesetzten Nachrichten sollte unabhängig von der Existenz einer Bussitzung möglich sein. Würde beim Hinzufügen eines Nachrichtenfeldobjekts zu einer zusammengesetzten Nachricht in Smalltalk-Repräsentation bereits das Buskommando ‘rvmsg_Append’ aufgerufen, wäre ein Nachrichtenzusammenbau nur unter Beteiligung eines Sitzungsverwalters möglich, da ‘rvmsg_Append’, wie jedes andere Buskommando auch, die Angabe einer gültigen Sitzungskennung erfordert.

3.4.5 Nachrichtenempfang im Detail

Der Empfang von Busnachrichten wird ausgelöst durch Busereignisse, genauer: Ein Busereignis unter einem abonnierten Subject führt zum Aufruf (Callback) des beim Abonnementauftrag mitgeteilten Smalltalk-Callback-Blocks. Ein solcher Callback-Block muß immer folgende Form haben:

```
[ :sess :name :replyName :msgType :msgSize :msg :arg |
    "... Callback-Routine ..."
]
```

Es müssen also eine Reihe von Argumenten vorgesehen werden, die beim Callback vom externen Busakteur folgendermaßen belegt werden:

sess	Sitzungskennung des Empfängers
name	Subject, unter dem das Busereignis ausgelöst wurde
replyName	Inbox-Subject für Rückmeldungen, nur belegt bei Empfang einer Auftragsveröffentlichung
msgType	Typ der empfangenen Nachricht
msgSize	Länge der empfangenen Nachricht in Bytes
msg	Verweis auf die die Nachricht enthaltende Bytefolge
arg	nicht belegt

Die mit dem empfangenen Busereignis verbundene Nachricht wird im Argument-Tripel (‘msgType’, ‘msgSize’, ‘msg’) des Callback-Blocks an den Empfänger übergeben. Es handelt sich dabei um die C-Repräsentation (Typ, Länge, Inhalt) der empfangenen Nachricht, die vor einer Weiterverarbeitung zuerst in die entsprechende Smalltalk-Repräsentation übersetzt werden muß. Dieser Übersetzungsvorgang führt zum Erzeugen von Objekten der Smalltalk-Klassen RVStringMessage, RVIntegerMessage oder RVComposedMessage. Da die Erzeugung dieser Objekte grundsätzlich Aufgabe der für die jeweilige Klasse zuständigen Klassenobjekte ist, befindet sich die Implementierung der Übersetzungsroutinen in entsprechenden Klassenmethoden. Das Ablaufschema für einen solchen Übersetzungsvorgang ist in Bild III am Ende dieser Schrift dargestellt. Dort sind Zuständigkeitsbereiche für einen Anwendungsakteur und ingesamt vier Klassenakteure abgegrenzt. Neben den für die drei genannten Klassen RVStringMessage, RVIntegerMessage und RVComposedMessage zuständigen Klassenobjekten (Klassenakteuren) kommt zusätzlich der für die Klasse **RVMessage** zuständige Klassenakteur vor. RVMessage ist die bisher nicht erwähnte Oberklasse der drei genannten Nachrichtenklassen. Bild 3.23 zeigt den entsprechenden Ausschnitt aus der Smalltalk-Klassenhierarchie.

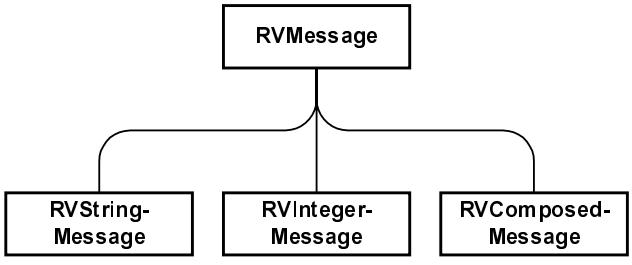


Bild 3.23 Nachrichtenklassen (Ausschnitt aus dem Smalltalk-Klassenbaum)

Zu jeder der drei Blatt-Klassen existiert jeweils eine Klassenmethode, mit deren Hilfe eine empfangene Nachricht von ihrer C-Repräsentation in ein Smalltalk-Nachrichtenobjekt der betreffenden Klasse gewandelt werden kann. Der Aufruf dieser Klassenmethoden aus Callback-Blöcken der oben gezeigten Form lautet jeweils:

- **RVStringMessage parseMessage:** msg
  (bei Nachrichten vom Typ RVMSG_STRING)
- **RVIntegerMessage parseMessage:** msg
  (bei Nachrichten vom Typ RVMSG_INT)
- **RVComposedMessage parseMessage:** msg **inSessionWithID:** sess
  (bei Nachrichten vom Typ RVMSG_RVMSG)

In allen drei Fällen gibt die jeweilige Klassenmethode ein Exemplar derjenigen Klasse zurück, für die das beauftragte Klassenobjekt zuständig ist: im ersten Fall ein Exemplar der Klasse RVStringMessage, im zweiten Fall ein Exemplar der Klasse RVIntegerMessage und im dritten Fall ein Exemplar der Klasse RVComposedMessage. Die Attribute dieser Exemplare werden so initialisiert, daß die erzeugten Nachrichtenobjekte den im Argument angegebenen C-Nachrichten entsprechen. Es handelt sich also um spezielle Exemplarerzeugungs-Methoden. Eine Vorbedingung für ihre ordnungsgemäße Abwicklung ist jedoch, daß die übergebene Nachricht auch tatsächlich den Nachrichtentyp besitzt, den die jeweilige Klasse in Smalltalk vertritt. So würde beispielsweise die Anweisung ‘RVStringMessage parseMessage: msg’ fehlschlagen, wenn die im Argument übergebene C-Nachricht vom Typ RVMSG_INT ist. Man muß sich also vor Aufruf einer solchen Klassenmethode sicher sein, von welchem Typ die empfangene Nachricht ist. Ist dies nicht der Fall, steht zusätzlich die Klassenmethode des Klassenobjekts RVMessage zur Verfügung, die folgendermaßen aufgerufen wird:

- **RVMessage parseMessage:** msg **msgType:** msgType **msgSize:** msgSize
  inSessionWithID: sess
  (bei Nachrichten unbekannten Typs)

In den Argumenten dieser Klassenmethode werden alle Angaben gemacht, die zum Übersetzen einer beliebigen Nachricht von C- in Smalltalk-Repräsentation erforderlich sind. Ein solcher Aufruf ist in Bild III links oben dargestellt. Bei Ausführung dieser Methode wird eine Fallunterscheidung bezüglich des im Argument ‘msgType’ übergebenen Typs gemacht. Es kommen vier Fälle vor, die sich im Bild III auf vier Spalten aufteilen und im folgenden von links nach rechts behandelt werden:

1. Der übergebene Nachrichtentyp ist weder RVMSG_STRING, noch RVMSG_INT, noch RVMSG_RVMSG; es handelt sich also um eine Nachricht, deren Typ in Smalltalk bis dato nicht unterstützt wird: Der RVMessage-Klassenakteur veranlaßt den Abbruch der Methodenabwicklung und die Erzeugung einer entsprechenden Ausnahmemeldung.

2. Es handelt sich um eine Nachricht des Typs RVMSG_STRING: Der RVMessage-Klassenakteur beauftragt den RVStringMessage-Klassenakteur zur Erzeugung und Initialisierung eines Objekts der Klasse RVStringMessage. Dazu ruft er dessen bereits vorgestellte Klassenmethode ‘parseMessage:’ auf und gibt dabei im Argument nur noch den Nachrichteninhalt weiter, da zur Übersetzung von String-Nachrichten keine weiteren Angaben erforder-

lich sind. Der RVStringMessage-Akteur setzt den so erhaltenen C-String in einen Smalltalk-String um und speichert ihn im Attribut ‘string’ des erzeugten RVStringMessage-Objekts. Dieses RVStringMessage-Exemplar erhält anschließend der Anwendungsakteur als Ergebnisobjekt zurück.

3. Es handelt sich um eine Nachricht des Typs RVMSG_INT: Durch Aufruf der Methode ‘parseMessage:’ des RVIntegerMessage-Klassenakteurs wird analog zu Fall 2 die Erzeugung und Initialisierung eines RVIntegerMessage-Exemplars veranlaßt und an den Anwendungsakteur zurückgegeben.

4. Es handelt sich um eine Nachricht des Typs RVMSG_RVMSG, also um eine zusammengesetzte Nachricht: Dann erfolgt ein Aufruf der bereits erwähnten Klassenmethode ‘parseMessage:inSessionWithID:’ des RVComposedMessage-Klassenakteurs. Hier wird neben dem Nachrichteninhalt zusätzlich die Sitzungskennung übergeben. Sie wird vom RVComposedMessage-Klassenakteur bei Aufruf des Buskommandos ‘rvmsg_Apply’ (vgl. Abschnitt 3.3.3) benötigt, der unmittelbar nach Erzeugung des RVComposedMessage-Objekts erfolgt. Die den ‘rvmsg_Apply’-Aufruf zeigenden Transitionen sind in Bild III grau unterlegt dargestellt. Das Buskommando ‘rvmsg_Apply’ wird hier benutzt, um die zyklische Ausführung einer Bearbeitungsprozedur für sämtliche Felder der Nachricht zu veranlassen, bei seinem Aufruf wird also kein Feldname angegeben. Die Bearbeitungsprozedur selbst ist in Bild III durch die beiden zwischen den grau eingefärbten Transitionen liegenden Transitionen dargestellt. Sie wird innerhalb eines Smalltalk-Blocks spezifiziert, der folgende Form haben muß:

```
[ :sess :fieldName :msgType :msgSize :msg :arg |
    "... Feldbearbeitungs-Prozedur ..."
]
```

Auch hier sind wiederum eine Reihe von Blockargumenten vorgesehen, die vom externen Busakteur bei jedem Aufruf folgendermaßen belegt werden:

sess die aktuelle Sitzungskennung

fieldName der Name des zur Bearbeitung übergebenen Nachrichtenfeldes

msgType, **msgSize**, **msg**
der Feldinhalt, eine durch das übliche Tripel umschriebene Nachricht in C-Repräsentation

arg nicht belegt

Bei jedem Durchlauf der Bearbeitungsprozedur wird als erstes die Übersetzung der in den Blockargumenten ‘msgType’, ‘msgSize’ und ‘msg’ übergebenen Nachricht von C- in Smalltalk-Repräsentation veranlaßt, indem rekursiv die allgemeine Übersetzungsmethode des RVMessage-Klassenakteus aufgerufen wird (parseMessage: msg msgType: msgType msgSize: msgSize inSessionWithID: sess). Das daraufhin als Ergebnis zurückgelieferte Nachrichtenobjekt – es kann ein Exemplar jeder beliebigen Blatt-Klasse aus Bild 3.23 sein – wird zusammen mit dem Feldnamen in einem zuvor erzeugten Nachrichtenfeldobjekt gespeichert und an die Liste der Nachrichtenfelder innerhalb des zu Anfang erzeugten RVComposedMessage-Exemplars angehängt.

Nachdem auf diese Weise alle Nachrichtenfelder übersetzt und angehängt worden sind, wird das vollständig initialisierte RVComposedMessage-Exemplar als Ergebnisobjekt an den Anwendungsakteur zurückgereicht.

Durch den vorgestellten Übersetzungsmechanismus hat man also die Möglichkeit, durch Aufruf einer einzigen Methode die Übersetzung beliebiger Busnachrichten von C- in Smalltalk-Repräsentation zu veranlassen. In allen Fällen, in denen man sich des Typs einer empfangenen Busnachricht nicht sicher sein kann, sollte der entsprechene Smalltalk-Callback-Block daher folgendermaßen beginnen:

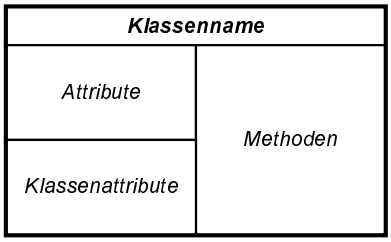
```
[ :sess :name :replyName :msgType :msgSize :msg :arg |

    "Übersetzung der empfangenen Nachricht veranlassen:"
    receivedMessage :=
        RVMessage parseMessage: msg
            msgType: msgType
            msgSize: msgSize
            msg: msg
            inSessionWithID: sess.

    "... Rest der Callback-Routine ..."
]
```

3.4.6 Der vollständige Klassenbaum

Bild IV am Ende dieser Schrift zeigt einen Ausschnitt aus dem Smalltalk-Klassenhierarchiebaum. Alle zum Smalltalk-Rendezvous-API gehörenden Klassen sind dort als fett umrandete Knoten dargestellt. Die Wurzel eines Smalltalk-Klassenbaums bildet immer die Klasse **Object**, alle weiteren Klassen sind direkte oder indirekte Nachkommen davon. Ganz links findet sich die Klasse RVInterface wieder. Es handelt sich um eine automatisch generierte Unterklasse von ExternalInterface, deren einziges Exemplar den RVInterface-Monopolakteur darstellt (siehe Abschnitt 3.3.5). ExternalInterface ist eine vorgefertigte Smalltalk-Klasse, die zum Lieferumfang des VISUALWORKS-DLL&C-Connect-Pakets gehört. Alle weiteren Klassen des Smalltalk-Rendezvous-API sind Nachkommen von **RVObject**. Durch den Knoten mit der Inschrift „...“ ganz rechts soll gezeigt werden, wo sich die Anwendungsklassen einer Busanwendung befinden würden.

3.4.6.1 Vererbungsbeziehungen

Im folgenden wird die Wahl der in Bild IV gezeigten Vererbungsbeziehungen begründet. Dabei wird von der nebenstehenden Darstellung für Klassen Gebrauch gemacht. Unter dem Klassennamen befinden sich links oben die Namen der Attribute, darunter die der Klassenattribute und auf der rechten Seite die Methodenbezeichner oder ersatzweise ein erklärender Text.



RVObject

Bild 3.24 zeigt die Klasse RVObject in dieser Darstellungsweise. Sie ist Oberklasse aller weiteren Klassen des Smalltalk-Rendezvous-API mit Ausnahme der Klasse RVInterface. Es handelt sich um eine *aufgeschobene* Klasse (vgl. [Meyer 90]: „deferred class“), d.h. von dieser Klasse sollen direkt keine Exemplare erzeugt werden. Durch ihr Klassenattribut ‘RVI’ bietet sie allen Exemplaren ihrer Unterklassen Zugriff auf den RVInterface-Monopolakteur. So konnte die Bindung des Monopolakteurs an ein Globalsymbol vermieden werden. Daneben werden durch weitere Klassenattribute Fehlerklassenverwalter (vgl. Abschnitt 2.2.5.3) zur Verfügung gestellt, die für Ausnahmesituationen zuständig sind, die bei Ausführung von Buskommandos auftreten können. Die Methoden der Klasse RVObject erlauben die Meldung solcher Ausnahmesituationen durch alle Exemplare ihrer Unterklassen auf eine einheitliche Weise. In RVObject werden keine Attribute definiert.

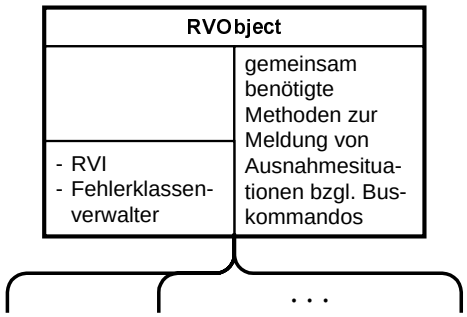


Bild 3.24 RVObject

Veröffentlichungsklassen

Bild 3.25 zeigt in der oberen Bildhälfte das Klassifikationsschema für Veröffentlichungen so, wie es sich aus den bisherigen Betrachtungen ergibt, und völlig unabhängig von irgendeiner Implementierungsform. Von diesen sieben Klassen ist nicht jede durch eine eigene Smalltalk-Klasse im Smalltalk-Klassenbaum vertreten, sondern es ist eine Abbildung auf lediglich vier Smalltalk-Klassen vollzogen worden, die in der unteren Bildhälfte gezeigt sind. Es handelt sich dabei um die vier bekannten Veröffentlichungsklassen RVPublication, RVRequest, RVMsgPublication und RVMsgRequest. Aus Anwendersicht übernimmt jede dieser Smalltalk-Klassen die Rolle einer Blattklasse des oberen Hiearchiebaums: RVPublication-Exemplare repräsentieren gewöhnliche, nachrichtenfreie Veröffentlichungen, RVRequest-Exemplare nachrichtenfreie Auftragsveröffentlichungen. Aus Implementierungssicht übernehmen RVPublication und RVRequest auch die Rollen der Oberklassen aus dem oberen Klassifikationsschema.

Da in sämtlichen Veröffentlichungsobjekten ein Attribut zur Aufnahme des Subjects, unter dem die jeweilige Veröffentlichung auf den Bus gegeben werden soll, vorgesehen ist, wird es bereits in der Klasse RVPublication für alle vier Veröffentlichungsklassen definiert, inklusive der Methode ‘subject:’ zum Belegen dieses Attributs. Die zweite Methode der Klasse RVPublication (‘publishInSessionWithID:’) gilt nur für gewöhnliche, nachrichtenfreie Veröffentlichungen, da in ihr das Versenden einer leeren String-Nachricht hart eincodiert ist (siehe Seite 55). Demnach wird diese Methode auch in der Unterklasse RVMsgPublication (links) redefiniert. Ferner wird in RVMsgPublication ein zusätzliches Attribut ‘message’ zur Aufnahme eines Nachrichtenobjekts definiert, welches durch die Methode ‘message:’ belegt werden kann.

In der Klasse RVRequest auf der rechten Seite werden zusätzlich zum geerbten Attribut ‘subject’ vier weitere Attribute eingeführt, die ihrerseits wieder an RVMsgRequest weitervererbt werden:

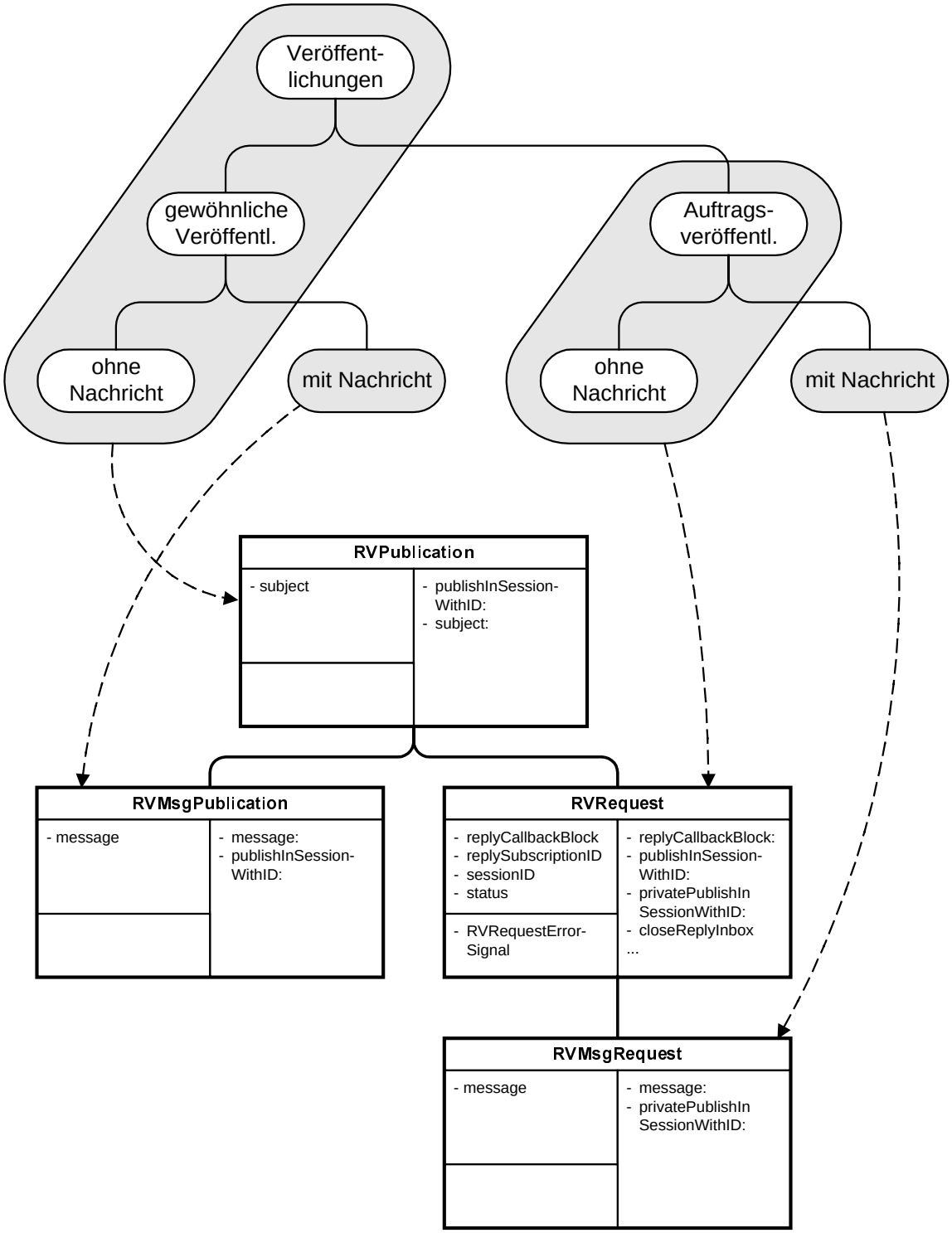


Bild 3.25 Veröffentlichungsklassen

replyCallbackBlock Dient zur Aufnahme des Smalltalk-Blocks, der bei Auftritt eines Busereignisses unter dem bei Veröffentlichung implizit abonnierten Rückmeldungs-Subject aufgerufen werden soll. Dieses Attribut kann mit Hilfe der Methode ‘replyCallbackBlock:’ belegt werden.

replySubscriptionID Hier wird die durch den externen Busakteur vergebene Rückmeldungsabonnementkennung abgespeichert.

sessionID Dient zur Speicherung der Sitzungskennung innerhalb der Methode ‘publishInSessionWithID:’, da die Sitzungskennung später bei Kündigung des Rückmeldungsabonnements innerhalb der Methode ‘closeReplyInbox’ noch einmal benötigt wird.

status Dient zur Aufnahme einer Zustandsinformation, die Auskunft darüber gibt, ob das Rückmeldungsabonnement bereits gekündigt wurde oder nicht.

Daneben wird als Klassenattribut ein weiterer Fehlerklassenverwalter (RVRequestErrorSignal) eingeführt, der für die Ausnahmesituation zuständig ist, die auftritt, wenn man eine Auftragsveröffentlichung zum wiederholten Male auf den Bus gibt, ohne voher das Rückmeldungsabonnement zu kündigen.

Auch in RVRequest muß die geerbte Methode ‘publishInSessionWithID:’ redefiniert werden. Um eine nochmalige vollständige Redefinition in RVMsgRequest zu vermeiden, wird der für RVMsgRequest-Exemplare abweichende Teil der Veröffentlichungsprozedur in einer weiteren Methode mit dem Namen ‘privatePublishInSessionWithID:’ implementiert. Diese nichtöffentliche Methode wird als Unterprozedur innerhalb von ‘publishInSessionWithID:’ aufgerufen. Nur ‘privatePublishInSessionWithID:’ wird dann in RVMsgRequest erneut redefiniert. Ferner wird in RVMsgRequest analog zu RVMsgPublication ein Attribut zur Aufnahme des Nachrichtenobjekts (‘message’), sowie eine Methode zur Belegung dieses Attributs (‘message:’) hinzugefügt.

Nachrichtenklassen

Bild 3.26 zeigt den Teilbaum aus der Smalltalk-Klassenhierarchie, der die Nachrichtenklassen umfaßt. RVMessage ist die aufgeschobene Oberklasse all jener Klassen, die die verschiedenen Nachrichtentypen in Smalltalk vertreten. In ihr sind bereits alle Methoden angegeben, die jedes Nachrichtenobjekt besitzen muß. Im einzelnen sind dies:

prepareForPublicationInSessionWithID:
Dient der Vorbereitung von Nachrichtenobjekten unmittelbar vor ihrer Veröffentlichung. Diese Methode enthält keine Anweisungen und muß bei denjenigen Unterklassen von RVMessage redefiniert werden, die Nachrichtentypen vertreten, bei denen eine mehrschrittige Umwandlung der Smalltalk-Repräsentation in die C-Repräsentation erforderlich ist. Bezogen auf die drei bisher implementierten Nachrichtentypen ist dies nur bei RVMSG_RVMSG (vertreten durch RVComposedMessage) der Fall, vgl. Abschnitt 3.4.4.3 und Bild II.

type Gibt den Nachrichtentyp zurück.

size Gibt die Länge einer Nachricht in Bytes zurück. Die in RVMessage definierte ‘size’-Methode gibt Null zurück. Sie kann für all jene Nachkommen beibehalten werden, die Nachrichtentypen vertreten, bei denen keine explizite Längenangabe erforderlich ist. Bezogen auf die drei bisher implementierten Nachrichtentypen ist dies bei RVStringMessage und RVComposedMessage der Fall.

data Gibt den Nachrichteninhalt in C-Repräsentation zurück.

sizeInBuffer Gibt den für die repräsentierte Nachricht benötigten Platz in einem Bytearray an, das zur Aufnahme einer zusammengesetzten Nachricht dient. Dieser Wert kann unter Umständen von dem bei ‘size’ gelieferten Wert abweichen.

Die Methoden ‘type’, ‘data’ und ‘sizeInBuffer’ sind aufgeschobene Methoden, d.h. bei ihnen ist eine Definition in sämtlichen Unterklassen Pflicht. In RVMessage werden noch keine Attribute spezifiziert. Dies geschieht erst in den Unterklassen.

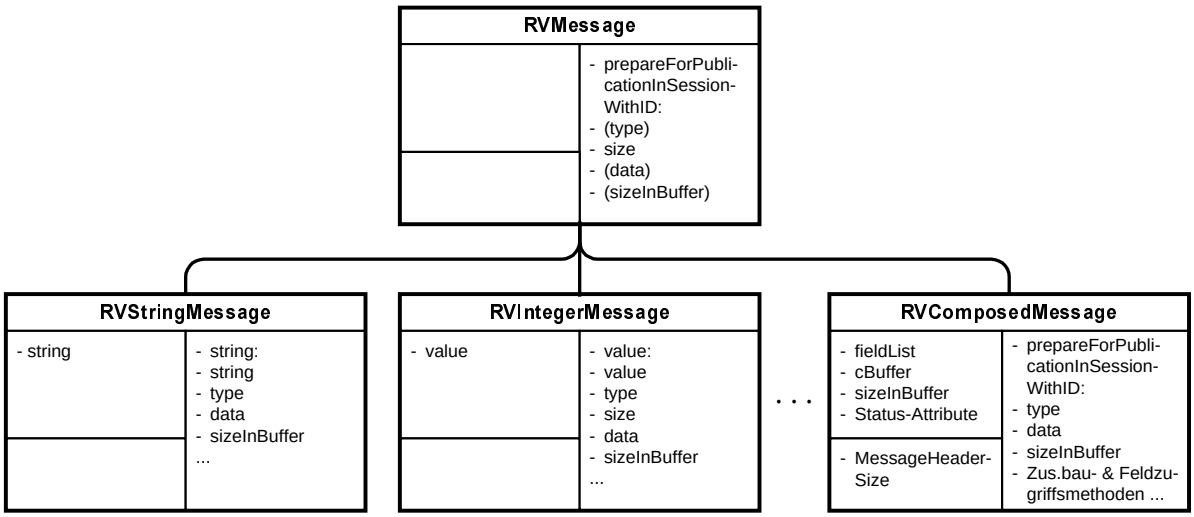


Bild 3.26 Nachrichtenklassen

RVStringMessage führt das Attribut ‘string’ ein. Es dient zur Aufnahme des Nachrichteninhaltes in Smalltalk-Repräsentation, d.h. zur Aufnahme eines Verweises auf ein Smalltalk-String-Objekt. Die zusätzlichen Methoden ‘string’ und ‘string:’ dienen zum Lesen und Setzen dieses Attributs. Da Smalltalk-Strings und C-Strings kompatibel sind, ist die Methode ‘data’ mit der Methode ‘string’ identisch. Innerhalb von RVIntegerMessage wird dagegen in der Methode ‘data’ eine Übersetzung der im Attribut ‘value’ gespeicherten Smalltalk-Ganzzahl in eine C-Integerzahl durchgeführt und ein Verweis auf diese C-Repräsentation an den Aufrufer zurückgegeben. Ferner muß in RVIntegerMessage die Methode ‘size’ redefiniert werden, da bei Nachrichten vom Typ RVMSG_INT eine explizite Längenangabe erforderlich ist. In der Klasse RVComposedMessage werden eine ganze Reihe von Attributen eingeführt:

fieldList Verweist auf die Smalltalk-Liste der zur zusammengesetzten Nachricht gehörenden Nachrichtenfeldobjekte.

cBuffer Enthält nach Ausführung der Methode ‘prepareForPublicationInSessionWithID:’ einen Verweis auf das C-Bytarray, das die C-Repräsentation der zusammengesetzten Nachricht beinhaltet.

sizeInBuffer Enthält nach Ausführung der Methode ‘sizeInBuffer’ die Anzahl benötigter Bytes innerhalb eines zu einer weiteren zusammengesetzten Nachricht gehörenden C-Bytearrays.

In zwei weiteren Attributen (‘Status-Attribute’) werden Zustandsinformationen mitgeführt, die Auskunft darüber geben, ob die Attribute ‘cBuffer’ und ‘sizeInBuffer’ noch auf dem neuesten Stand sind. In den zugehörigen Methoden ‘prepareForPublicationInSessionWithID:’ und

‘sizeInBuffer’ werden deren Werte berücksicht, um zu verhindern, das die unter Umständen recht aufwendigen Übersetzungs- und Berechnungsprozeduren unnötig oft durchlaufen werden. Das Klassenattribut ‘MessageHeaderSize’ enthält eine Ganzzahl-Konstante, die innerhalb der Methode ‘sizeInBuffer’ zur korrekten Berechnung der benötigten Puffergröße benötigt wird. Im Fall von RVComposedMessage kommen zu den bei allen Nachrichtenobjekten benötigten Methoden noch jene Methoden hinzu, die zum Zusammenbau von Nachrichten und zum Zugriff auf Nachrichtenfelder benötigt werden (‘Zus.bau & Feldzugriffsmethoden’). Sie wurden bereits zu Anfang von Abschnitt 3.4.4.3 vorgestellt.

3.4.6.2 Erzeugungsbeziehungen

Bild V am Ende dieser Schrift zeigt sämtliche Klassen des Smalltalk-Rendezvous-API, bei denen eine Erzeugung von Exemplaren vorgesehen ist. Es fehlen demnach die aufgeschobenen Klassen RVObject und RVMessage. Ferner kommen links im Bild die Anwendungsklassen vor, also die Klassen, die der Anwendungsprogrammierer zusätzlich schreibt, um eine Busanwendung zu realisieren. Der in Bild V verwendete Diagrammtyp dient zur Darstellung von Schichtungsrelationen. Im vorliegenden Fall werden damit die Erzeugungsbeziehungen zwischen den gezeigten Klassen dargestellt. Das Diagramm ist folgendermaßen zu lesen: Existiert zwischen zwei Klassenknoten A und B eine Verbindung derart, daß eine beim Knoten A nach unten austretende Kante über einen oder mehrere Kreuzungsknoten zum Knoten B führt, dann gilt: Exemplare der Klasse A erzeugen Exemplare der Klasse B.

Ganz oben sind jene beiden Klassen dargestellt, bei denen eine Erzeugung von Exemplaren bereits zur Bauzeit geschieht. Gewöhnlich existiert zu einer Smalltalk-Anwendung immer eine Anwendungsinitialklasse, deren einziges Exemplar zur Anwendungsbauzeit erzeugt und über ein Globalsymbol zugänglich gemacht wird. Dieses Anwendungsinitialobjekt besitzt dann eine bestimmte Methode, die den Start der Anwendung ermöglicht. Auch bei RVInterface ist die Exemplarerzeugung bereits zur Bauzeit geschehen, und zwar zur Bauzeit des Rendezvous-Smalltalk-API. Das einzige Exemplar dieser Klasse ist der in Abschnitt 3.3.5 vorgestellte RVInterface-Monopolakteur. Ausgehend vom Anwendungsinitialobjekt werden zur Anwendungslaufzeit für gewöhnlich weitere Exemplare anderer Anwendungsklassen erzeugt. Diese erzeugen dann neben weiteren Anwendungsobjekten (in Bild V nicht gezeigt) Exemplare der Klassen des Smalltalk-Rendezvous-API. Dabei fällt auf, daß durch Anwendungsobjekte Exemplare von lediglich acht der insgesamt 14 Klassen des Smalltalk-Rendezvous-API erzeugt werden. Um die weiteren sechs Klassen braucht sich der Anwendungsprogrammierer also nicht zu kümmern. Eine Ausnahme machen die Klassen RVMessage und RVMessageField. Obwohl durch Anwendungsobjekte keine Exemplare dieser beiden Klassen erzeugt werden, sollte der Anwendungsprogrammierer sie trotzdem kennen, denn das für RVMessage zuständige Klassenobjekt dient den Anwendungsakteuren als Übersetzer empfangener Nachrichten, und RVMessageField-Exemplare werden nach Aufruf der Methode ‘fieldsDo:’ eines RVComposedMessage-Objekts an Anwendungsakteure übergeben.

3.5 Schicht 3: Ein Klassengerüst für Multiclient-Multiserver-Anwendungen

Das im vorigen Abschnitt 3.4 dokumentierte Smalltalk-Rendezvous-API kann bereits zum Bau beliebiger Busanwendungen benutzt werden. Die Zielsetzung der vorliegenden Arbeit ist es jedoch, darüber hinaus eine Umgebung speziell zur Realisierung der in Abschnitt 3.1 spezifizierten Anwendungssysteme zur Verfügung zu stellen. In diesen 3-Ebenen-Client-Server-Anwendungen kommunizieren mehrere Clients (GUIs oder Workprozesse) mit mehreren Servern (Workprozesse) über einen gemeinsamen Kanal, den Rendezvous-Softwarebus. Um solche Anwendungen mit möglichst wenig Aufwand in Smalltalk realisieren zu können, wird aufbauend auf dem Rendezvous-Smalltalk-API ein zusätzlicher Katalog von Klassen bereitgestellt. Dieses Klassengerüst zur Realisierung von Multiclient-Multiserver-Systemen wird im folgenden hergeleitet. Der zugehörige Smalltalk-Quelltext befindet sich in Anhang C.

3.5.1 Begriffsklärung

In diesem Abschnitt werden zunächst eine Reihe von Begriffen erklärt, die in den folgenden Abschnitten auftreten. Als erstes werden vier Akteursbezeichnungen eingeführt bzw. geklärt, die innerhalb der Applikationsebene (vgl. Bild 3.1 auf Seite 23) Verwendung finden:

Workprozeß, Dienstleister, Dienstanbieter und Auftragsabwickler (Bild 3.27)

Ein *Workprozeß* soll verstanden werden als eine Sammlung von *Dienstleistern*. Zu einem Workprozeß gehören also mehrere Dienstleister, die für jeweils unterschiedliche Dienstleistungen zuständig sind. Ein Dienstleister besteht selbst wiederum aus genau einem *Dienstanbieter* und einer variablen Anzahl von *Auftragsabwicklern*. Der Dienstanbieter innerhalb eines Dienstleisters bietet die betreffende Dienstleistung über den Softwarebus nach außen an. Wird daraufhin von außen eine Dienstleistung beim Dienstanbieter nachgefragt, erzeugt dieser einen Auftragsabwickler zur einmaligen Ausführung der Dienstleistung. Innerhalb eines Dienstleisters können gleichzeitig mehrere Auftragsabwickler aktiv sein.

Man kann sich einen Workprozeß als ein Dienstleistungs-Unternehmen vorstellen, das mehrere verschiedene Dienstleistungen abdeckt. Innerhalb dieses Unternehmens gibt es für jede Dienstleistung genau einen Vertriebsmitarbeiter (≙ Dienstanbieter), der diese Dienstleistung nach außen anbietet. Hat dieser Vertriebsmitarbeiter einen Kunden gewonnen, stellt er zur einmaligen Ausführung der Dienstleistung für den betreffenden Kunden eine neue Fachkraft (≙ Auftragsabwickler) ein. Nach Ausführung der Dienstleistung verläßt die Fachkraft das Unternehmen wieder. Ein Vertriebsmitarbeiter inklusive der aktuell von ihm beschäftigten Fachkräfte bilden zusammen eine Abteilung (≙ Dienstleister).

Es folgen drei Akteursbezeichnungen, die bei Kunden von Workprozessen – das können Akteure aus der Präsentations- und der Applikationsebene sein – auftauchen:

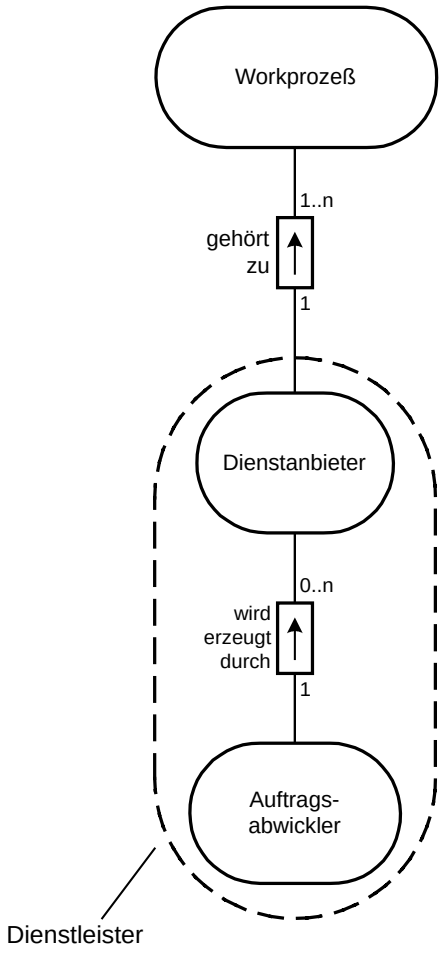


Bild 3.27 Workprozeß, Dienstleister, Dienstanbieter und Auftragsabwickler

Kunde, Auftraggeber und Dienstvermittler (Bild 3.28)

Als *Kunden* werden solche Akteure bezeichnet, die während ihrer Lebensdauer eine oder mehrere der innerhalb der Applikationsebene angebotenen Dienstleistungen in Anspruch nehmen. Dies können zum einen GUI-Akteure der Präsentationsebene sein, zum anderen aber auch Auftragsabwickler aus der Applikationsebene, die während der Ausführung einer Dienstleistung ihrerseits eine weitere Dienstleistung in Anpruch nehmen. Alle diese als Kun-

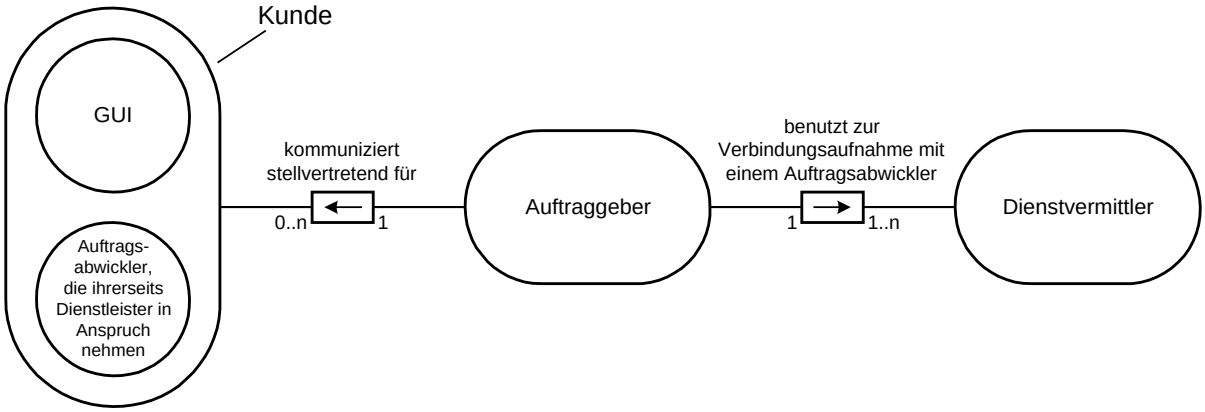


Bild 3.28 Kunde, Auftraggeber und Dienstvermittler

den auftretende Akteure bedienen sich bei der Dienstleistungsinanspruchnahme jeweils eines sogenannten *Auftraggebers*, der stellvertretend für sie über den Softwarebus mit dem die Dienstleistung ausführenden Auftragsabwickler kommuniziert. Zuvor muß jedoch der Auftraggeber einen *Dienstvermittler* bitten, ihm eine Verbindung zu einem Auftragsabwickler eines passenden Dienstleisters bereitzustellen. Dieser Dienstvermittler kommuniziert dazu mit allen Dienstanbietern, die die gewünschte Dienstleistung anbieten.

Bild 3.29 auf der folgenden Seite zeigt die Buskommunikationsbeziehungen zwischen den in den vorigen beiden ER-Diagrammen vorkommenden Akteurstypen noch einmal explizit. Dienstvermittler dienen Auftraggebern zur Vermittlung von Auftragsabwicklern. Während eines Vermittlungsvorgangs kommuniziert ein Dienstvermittler über den Softwarebus mit allen Dienstanbietern, die die vom Kunden gewünschte Dienstleistung anbieten. Anschließend kommuniziert ein Auftraggeber dann mit genau einem exklusiv für ihn arbeitenden Auftragsabwickler. Die Auftragsabwicklung kann sich dabei durchaus über mehrere Dialogschritte erstrecken. Die Menge der Auftragsabwickler ist in Bild 3.29 in zwei Untermengen partitioniert. Auftragsabwickler aus der oberhalb des Trennstriches liegenden Untermenge treten im Verlaufe ihrer Auftragsbearbeitung selbst wieder als Kunden auf.

3.5.2 Das Server-Auswahl-Protokoll

Im folgenden soll das Kommunikationsprotokoll vorgestellt werden, das die Kommunikation zwischen Dienstvermittlern auf Kundenseite und Dienstanbietern auf Workprozeßseite regelt. Diese Kommunikation dient, wie bereits erwähnt, der Vermittlung eines Auftragsabwicklers an einen Auftraggeber. Nun soll es durchaus möglich sein, daß dieselbe Dienstleistung durch Dienstleister aus mehreren Workprozessen geleistet werden kann (vgl. Abschnitt 3.1.2). Daher werden an einem aktuellen Vermittlungsvorgang grundsätzlich alle Dienstanbieter, die die gewünschte Dienstleistung anbieten, beteiligt. Während der Vermittlung erhält genau einer dieser Dienstanbieter den Zuschlag. Die das Kommunikationsprotokoll darstellenden Petrinetze (Bild 3.30 und Bild 3.31) zeigen die Aktionen des Dienstvermittlers und jeweils eines einzigen Dienstanbieters. In Bild 3.30 ist dies der Dienstanbieter, der den Zuschlag erhält, in Bild 3.31 ein anderer. Vor der Erläuterung des entworfenen Protokolls anhand dieser Diagramme soll zunächst die verwendete Symbolik erklärt werden.

Die Kommunikation zwischen Dienstvermittler und Dienstanbietern erfolgt über den Softwarebus. Dazu werden Veröffentlichungen unter bestimmten Subjects, die die beteiligten Kommunikationspartner abonniert haben, auf den Bus gegeben. Bei Abwicklung der gezeigten Petrinetze ist also jeder Markenfluß über eine Zuständigkeitsgrenze mit einem Busereignis verbunden. Handelt es sich dabei um ein Busereignis, das durch eine Veröffentlichung unter einem Inbox-Subject ausgelöst wurde, ist die Empfänger-Stelle, in die die Marke nach Überschreiten der Zuständigkeitsgrenze wandert, mit einem Telefonsymbol gekennzeichnet. Dies soll veranschaulichen, daß derjenige Akteur, in dessen Zuständigkeitsbereich eine solche Stelle liegt, der einzige Empfänger des ausgelösten Busereignisses ist. (Zur Erinnerung: Inbox-Subjects können immer nur von einem einzigen Busteilnehmer abonniert werden.) Handelt es sich dagegen um ein Busereignis, das durch eine Veröffentlichung unter einem ge-

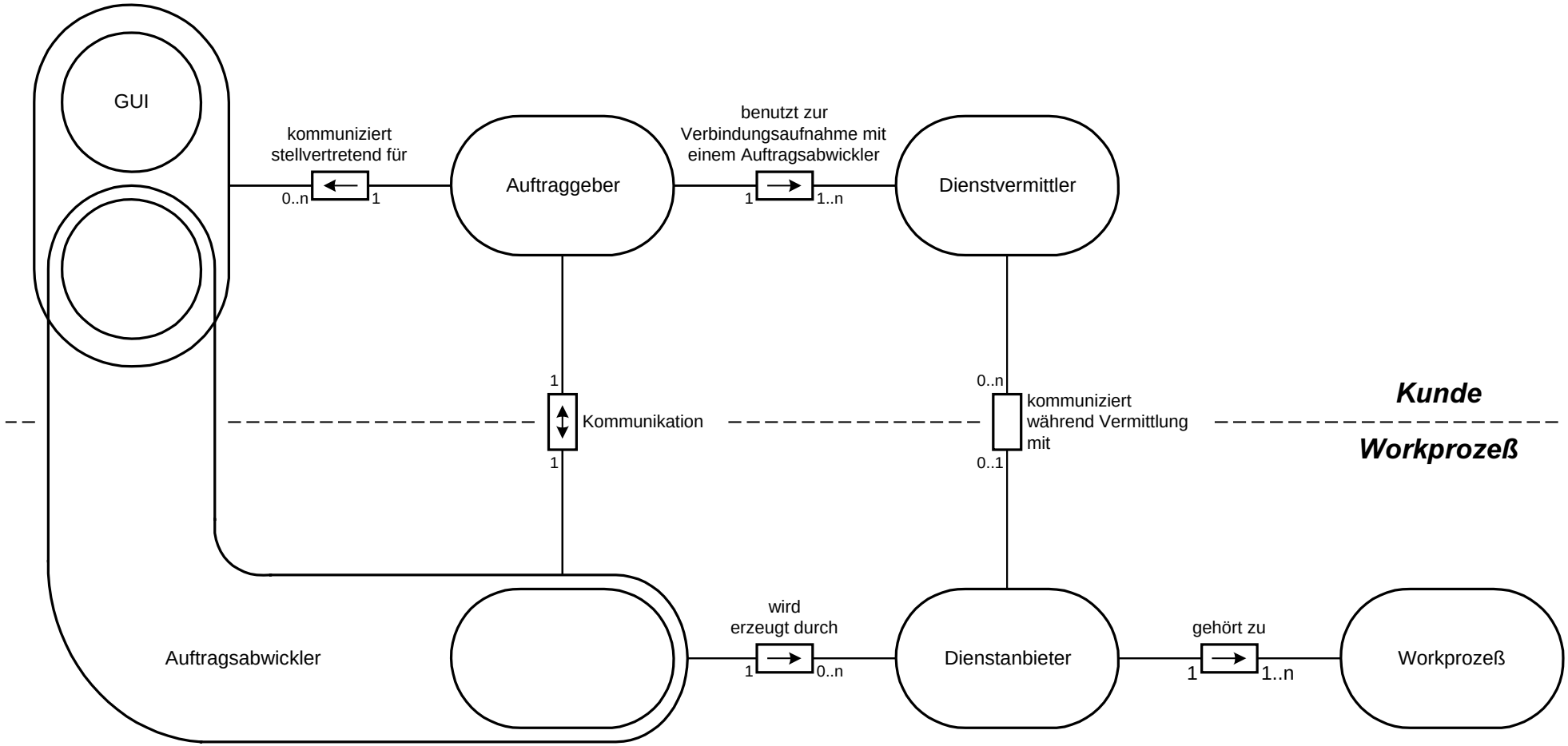


Bild 3.29 Kommunikationsbeziehungen: Auftraggeber und Auftragsabwickler, Dienstvermittler und Dienstanbieter

wöhnlichen Subject ausgelöst wurde, ist die Kante, über die die Marke den Zuständigkeitsbereich des Senders verläßt, mit einem Lautsprechersymbol gekennzeichnet. Dies soll veranschaulichen, daß das ausgelöste Busereignis potentiell von mehreren Akteuren (hier: mehreren Dienstanbietern) empfangen werden kann (Broadcast). In der gewählten Veranschaulichung ist das Abonnement eines Inbox-Subjects gleichbedeutend mit dem Anmelden eines Telefons. Die Telefonnummer (das Inbox-Subject) wird dabei vom externen Busakteur vergeben. Die Kündigung eines Inbox-Subjects ist gleichbedeutend mit dem Abmelden des zugehörigen Telefons. Löst das Schalten einer Transition die Anmeldung eines Telefons aus, ist dies durch einen Pfeil von der betreffenden Transition zum jeweiligen Telefonsymbol dargestellt.

Nach der Erkärung der gewählten Symbolik folgt nun anhand von Bild 3.30 die Behandlung des entworfenen Protokolls:

Ausschreibung

Der Dienstvermittler beginnt den Vermittlungsvorgang mit der Veröffentlichung einer Ausschreibung. Das Subject, unter dem er die Ausschreibung veröffentlicht, ist das *dienstspezifische Ausschreibungs-Subject*, das jeder Dienstanbieter der gewünschten Dienstleistung abonniert hat. Es hat die Form „*<Dienstleistungs-ID>*.CallForTenders“[10]. Der dienstspezifische Teil (Dienstleistungs-ID) ist dem Dienstvermittler zuvor vom Auftraggeber im Vermittlungsauftrag mitgeteilt worden. Die Veröffentlichung der Ausschreibung ist technisch gesehen eine Auftragsveröffentlichung, die das Abonnement eines Inbox-Subjects (☎ $_{V1}$) nach sich zieht. Anschaulich gesprochen würde der Inhalt einer solchen Ausschreibungsveröffentlichung eines Dienstvermittlers lauten: „Ich benötige jemanden, der mir die Dienstleistung X ausführt. Entsprechende Angebote sind unter Telefonnummer ‘☎$_{V1}$’ an mich zu richten.“

Angebot

Empfängt ein Dienstanbieter eine solche Ausschreibung, merkt er sich zunächst einmal das mitgelieferte Rückmeldungs-Subject (☎$_{V1}$). Anschließend antwortet er auf die Ausschreibung mit einer Angebotsveröffentlichung unter diesem Subject. Die Angebotsveröffentlichung stellt technisch gesehen wiederum eine Auftragsveröffentlichung dar, und zieht das Abonnement eines Inbox-Subjects für den eventuellen Angebotszuschlag nach sich (☎ $_{Anb.}$). Anschaulich gesprochen würde sie lauten: „Unser Unternehmen kann Dienst X für Sie leisten. Wenn Sie dieses Angebot annehmen, rufen Sie mich unter Telefonnummer ‘☎$_{Anb.}$’ zurück.“

Zuschlag

Der Dienstvermittler kündigt nun nach Empfang des ersten Angebots sofort sein Angebots-Inbox-Subject (☎ $_{V1}$), um keine weiteren Angebote mehr zu erhalten. Durch eine erneute Auftragsveröffentlichung unter dem bei Angebotsempfang erhaltenen Rückmeldungs-Subject (☎$_{Anb.}$) teilt er dann mit, daß er das Angebot annimmt und eine Zuschlagsbestätigung erwartet: „Ich nehme ihr Angebot an. Bitte bestätigen Sie mir den Zuschlag durch einen Rückruf unter Telefonnummer ‘☎$_{V2}$’.“

---

[10] Call for Tenders: engl. Fachbegriff für ‘Ausschreibung’

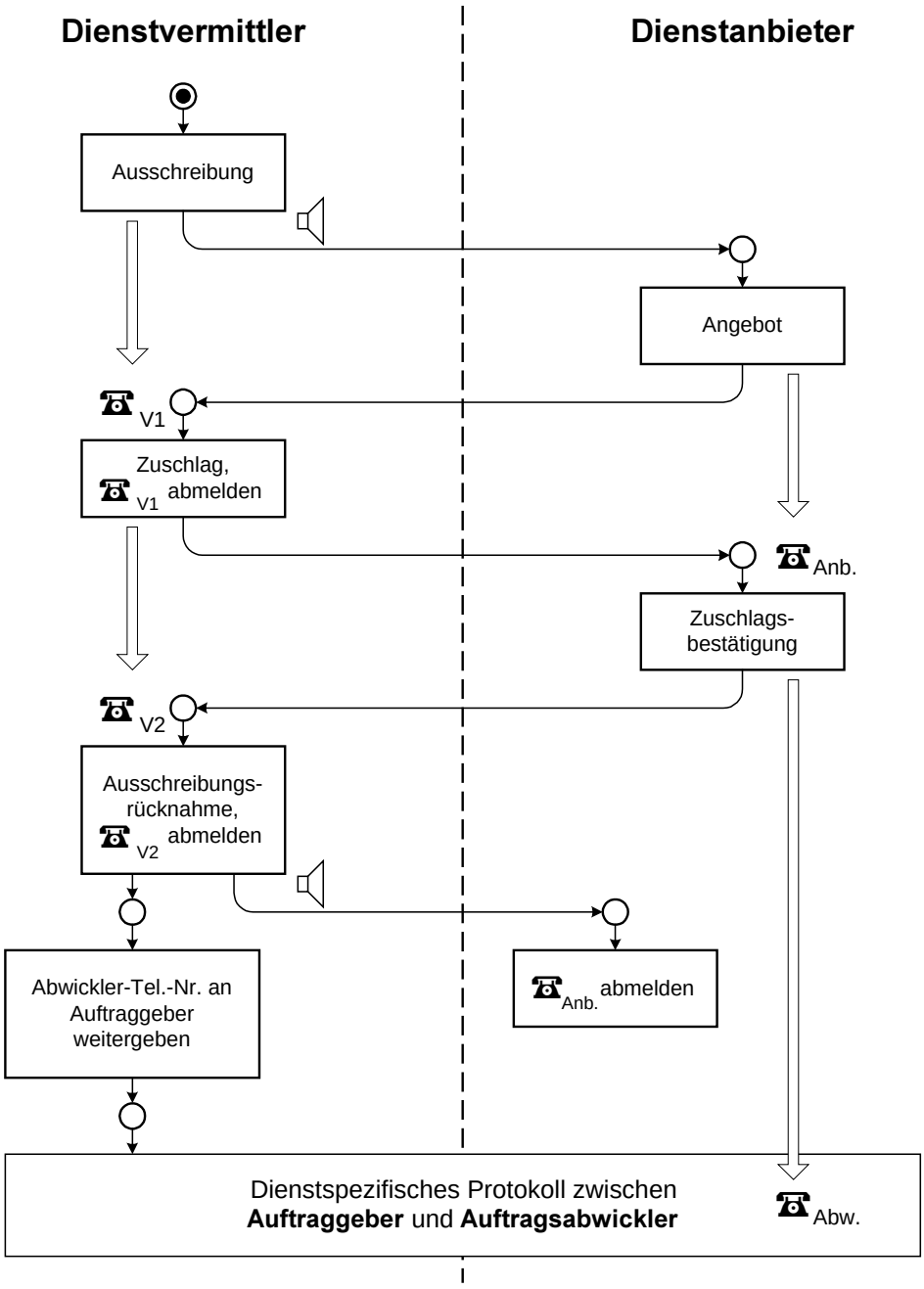


Bild 3.30 Serverauswahlprotokoll, Fall 1: Dienstanbieter erhält Zuschlag

Zuschlagsbestätigung

Empfängt der Dienstanbieter nun den Zuschlag, ist er auch für die Bereitstellung des Auftragsabwicklers zuständig. Daher erzeugt er zunächst einen Auftragsabwickler und veranlaßt diesen zum Abonnement eines Inbox-Subjects (☎ Abw.). Anschließend erfragt er beim Auftragsabwickler dieses Inbox-Subject und teilt es innerhalb einer gewöhnlichen Veröffentlichung unter dem bei Zuschlagsempfang erhaltenen Rückmeldungs-Subject (☎ V2) dem Dienstanbieter mit: „Schön, daß Sie unser Angebot angenommen haben. Ab sofort steht exklusiv für Sie eine Fachkraft unter Telefonnummer ‘☎Abw.’ zur Verfügung.“

Ausschreibungsrücknahme

Nach Empfang dieser Zuschlagsbestätigung kann sich der Dienstvermittler sicher sein, daß ein Auftragsabwickler bereitsteht. Er kündigt das Zuschlagsbestätigungs-Inbox-Subject (☎ $_{V2}$) und teilt allen Dienstanbietern durch eine gewöhnliche Veröffentlichung mit, daß die zu Anfang gemachte Ausschreibung nunmehr obsolet ist. Diese Ausschreibungsrücknahme-Veröffentlichung erfolgt unter dem *dienstspezifischen Ausschreibungsrücknahme-Subject* „*<Dienstleistungs-ID>*.CallForTendersCancellation“, welches ebenfalls alle die betreffende Dienstleistung anbietenden Dienstanbieter abonniert haben. Zusätzlich teilt er innerhalb der Ausschreibungsrücknahme-Veröffentlichung sein nicht mehr gültiges Angebots-Inbox-Subject (☎ $_{V1}$) mit: „Der Auftrag zur Ausführung der Dienstleistung X ist vergeben. Meine Telefonnummer ‘☎$_{V1}$’ zur Abgabe von Angeboten ist nicht mehr gültig.“

Nach Empfang der Ausschreibungsrücknahme-Veröffentlichung kündigt der Dienstanbieter das Inbox-Subject für Zuschläge (☎ $_{Anb.}$). Da er zu diesem Zeitpunkt bereits anderen Dienstvermittlern weitere Angebote gemacht haben kann, benötigt er als weitere Angabe das mitgelieferte ehemalige Angebots-Inbox-Subject (☎ $_{V1}$). Anhand dieser Angabe kann er über eine Zuordnungstabelle ermitteln, welches Zuschlags-Inbox-Subject er nun konkret zu kündigen hat. Dieser Vorgang wird später noch genauer erkärt. Der Dienstvermittler gibt unmittelbar nach Rücknahme der Ausschreibung das Inbox-Subject, unter dem der vom Dienstanbieter bereitgestellte Auftragsabwickler erreichbar ist (☎ $_{Abw.}$), an den Auftraggeber weiter. Anschließend kommunizieren Auftraggeber und Auftragsabwickler gemäß einem dienstspezifischen Protokoll.

Bild 3.31 zeigt den Ablauf der Kommunikation zwischen Dienstvermittler und Dienstanbieter für den Fall, daß der Dienstanbieter keinen Zuschlag auf sein Angebot erhält. Da der Dienstvermittler nach Eingang des ersten Angebots sein Angebots-Inbox-Subject sofort kündigt, gehen alle weiteren Angebote ins Leere, hier dargestellt durch die markenverzehrende Transition im Zuständigkeitsbereich des externen Busakteurs. Die Kündigung des bei Angebotsveröffentlichung abonnierten Zuschlags-Inbox-Subjects erfolgt nach Empfang der Ausschreibungsrücknahme-Veröffentlichung, wie oben beschrieben.

Der Leser fragt sich vermutlich, warum der Dienstvermittler seine Ausschreibung erst nach Eingang der Zuschlagsbestätigung zurücknimmt und nicht bereits unmittelbar nach Abgabe des Zuschlags (vgl. Bild 3.30). Man sollte doch erwarten, daß der Zuschlag vor der Ausschreibungsrücknahme-Veröffentlichung beim Dienstanbieter eintrifft, auch ohne daß eine Synchronisation durch Abwarten der Zuschlagsbestätigung stattfindet. Allgemeiner gesprochen: Man sollte erwarten, daß zwei Busereignismeldungen, die von derselben Quelle (hier: vom Dienstvermittler) ausgelöst werden, bei einer Senke, die für beide empfangsbereit ist (hier: der Dienstanbieter, der den Zuschlag erhält), in derselben Reihenfolge ankommen, in der sie abgeschickt werden. Diese Erwartung wird jedoch vom Softwarebussystem dann nicht erfüllt, wenn die beiden Ereignisse *typverschieden* sind. Dabei werden zwei Ereignistypen unterschieden:

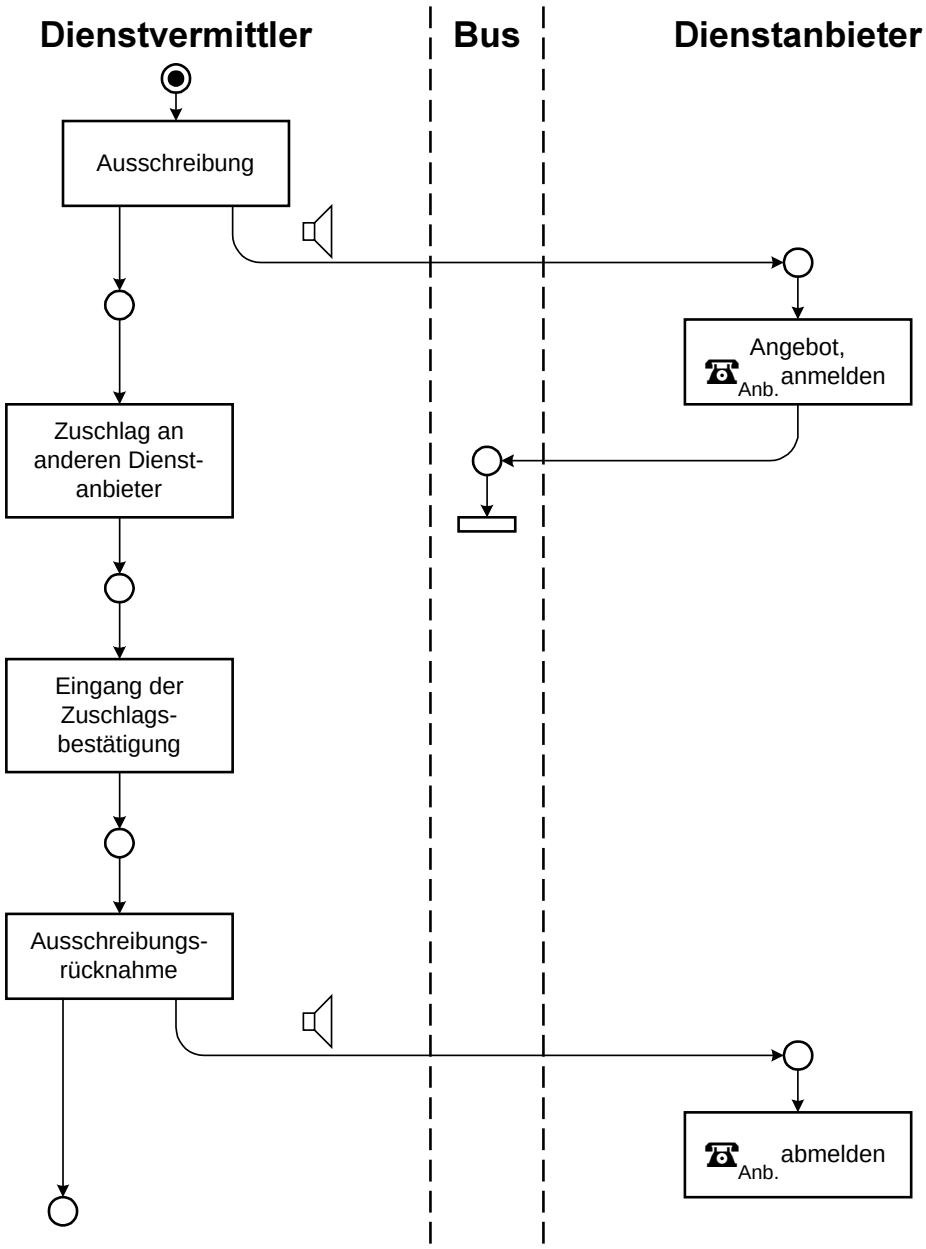


Bild 3.31 Serverauswahlprotokoll, Fall 2: Dienstanbieter erhält keinen Zuschlag

- *Inbox-Ereignisse:*
  Das sind Ereignisse, die durch Veröffentlichungen unter Inbox-Subjects ausgelöst werden.
- *Broadcast-Ereignisse:*
  Das sind Ereignisse, die durch Veröffentlichungen unter gewöhnlichen Subjects ausgelöst werden.

Da die Zuschlag-Veröffentlichung unter einem Inbox-Subject geschieht, die Ausschreibungsrücknahme-Veröffentlichung jedoch unter einem gewöhnlichen Subject, hat man es im vorliegenden Fall also tatsächlich mit typverschiedenen Busereignissen zu tun. Daher ist es durchaus möglich, daß eine unmittelbar *nach* der Zuschlag-Veröffentlichung auf den Bus gegebene Ausschreibungsrücknahme-Veröffentlichung beim Dienstanbieter *vorher* eintrifft. Das würde

dazu führen, daß der Dienstanbieter sein Zuschlags-Inbox-Subject zu früh kündigt und damit die Zuschlagsmeldung, die bereits zu ihm unterwegs ist, nicht mehr empfängt. Dies hätte wiederum die fatale Folge, daß kein Auftragsabwickler mehr bereitgestellt würde. Die zusätzliche Synchronisation über die Zuschlagsbestätigung ist also unbedingt erforderlich.

3.5.3 Aufbau der beteiligten Akteure

In den folgenden Abschnitten wird nun das Aufbaumodell der bereits in Abschnitt 3.5.1 eingeführten Akteure gezeigt. Dieses Modell entstand unter Beachtung der Heterogenitätsforderung aus Abschnitt 3.1.1.3 ohne Festlegung auf eine bestimmte Implementierungssprache. Daher wird man in den Aufbaubildern dieses Abschnitts keine Akteure des Smalltalk-Rendezvous-API finden. Statt dessen wird eine abstraktere Darstellung des Softwarebusses in Form eines einzigen Akteurs gewählt. In den folgenden Abschnitten wird mehrfach zwischen der Workprozeß-Seite und der Kundenseite gewechselt, um die über den Bus kommunizierenden Akteure jeweils paarweise nacheinander zeigen zu können. Zur Orientierung ist die Betrachtung von Bild 3.29 hilfreich.

3.5.3.1 Dienstleister

Bild 3.32 zeigt den Aufbau eines Dienstleisters (Workprozeß-Seite). Dieser besteht intern aus einem Dienstanbieter (links) und im allgemeinen mehreren Auftragsabwicklern (rechts). Der Dienstanbieter besteht wiederum aus drei Akteuren: dem Ausschreibungsempfänger, dem Ausschreibungsrücknahme-Reakteur und dem Auftragsannahme-Akteur. Alle gezeigten Akteure besitzen die Fähigkeit zur Buskommunikation. Kanäle, über die Veröffentlichungen unter Inbox-Subjects fließen, sind mit einem Telefonsymbol gekennzeichnet, Kanäle, über die Veröffentlichungen unter gewöhnlichen Subjects fließen, mit einem Lautsprechersymbol.

Der Ausschreibungsempfänger ist Abonnent des dienstspezifischen Ausschreibungs-Subjects. Bei Empfang einer Ausschreibung antwortet er mit einem Angebot unter dem mitgelieferten Angebots-Inbox-Subject des Dienstvermittlers. Das Angebot hat die Form einer Auftragsveröffentlichung, wobei das implizierte Rückmeldungs-Abonnement hier dem Auftragsannahme-Akteur zugeordnet wird. Der Auftragsannahme-Akteur ist somit empfangsbereit für den Zuschlag bezüglich des gemachten Angebots. Unmittelbar vor der Angebotsabgabe legt der Ausschreibungsempfänger in dem unter ihm liegenden Speicher ein Wertepaar ab, welches aus dem Angebots-Inbox-Subject des ausschreibenden Dienstvermittlers und der Abonnementkennung des durch die Angebotsveröffentlichung implizierten Rückmeldungs-Abonnements besteht. Dieses Wertepaar benötigt später der Ausschreibungsrücknahme-Reakteur.

Der Ausschreibungsrücknahme-Reakteur ist Abonnent des dienstspezifischen Ausschreibungsrücknahme-Subjects. Nach Empfang einer Ausschreibungsrücknahme sucht er in dem unter ihm liegenden Speicher nach jenem Wertepaar, das das innerhalb der Ausschreibungsrücknahme-Meldung mitgeteilte obsolete Angebots-Inbox-Subject enthält. Diesem Wertepaar entnimmt er die Kennung des nunmehr überflüssigen Rückmeldungsabonnements der Zuschlagsmeldung und kündigt es. Anschließend entfernt er das betreffende Wertepaar aus dem

Speicher. Auf diesem Weg entzieht der Ausschreibungsrücknahme-Reakteur dem Auftragsannahme-Akteur die Empfangsbereitschaft bezüglich des nach Ausschreibungsrücknahme ohnehin garantiert nicht mehr eintreffenden Zuschlags.

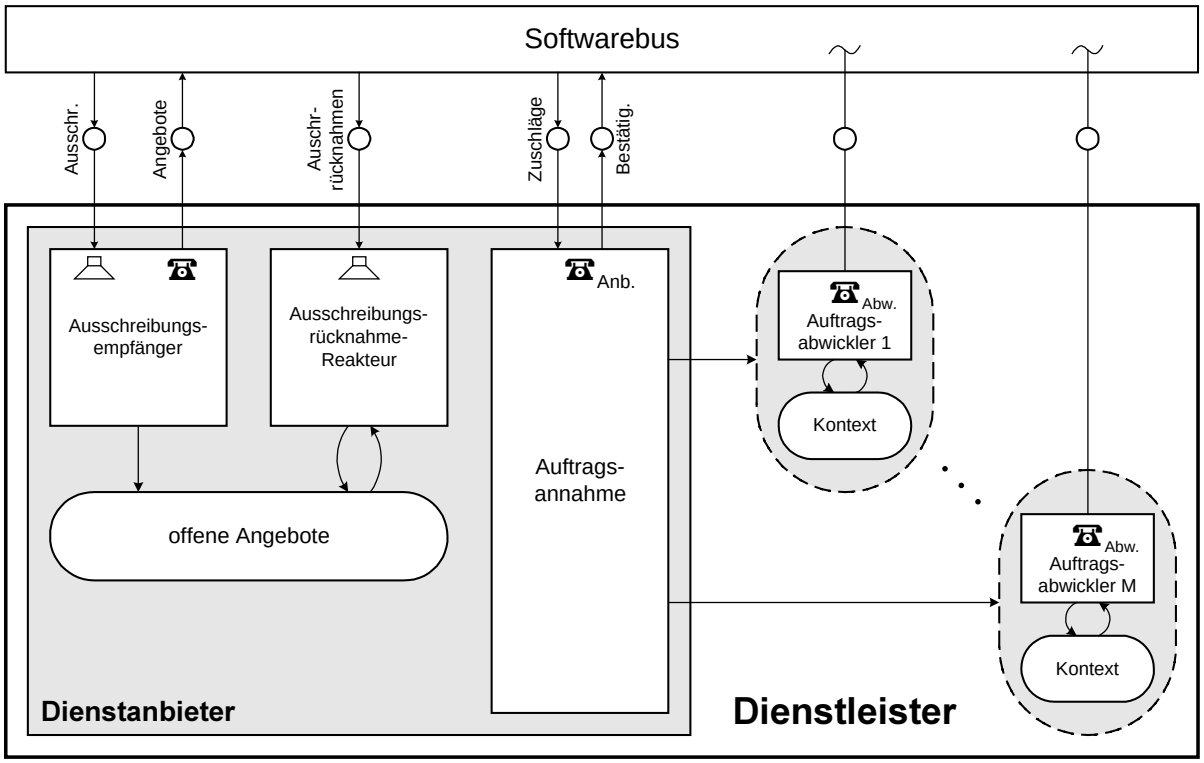


Bild 3.32 Aufbau eines Dienstleisters

Empfängt der Auftragsannahme-Akteur während seiner Empfangsbereitschaft eine Zuschlagsmeldung, erzeugt er als erstes einen Auftragsabwickler und veranlaßt ihn zum Abonnement eines eigenen Inbox-Subjects. Dieses Auftragsabwickler-Inbox-Subject meldet er anschließend innerhalb der Zuschlagsbestätigung demjenigen Dienstvermittler zurück, der den Zuschlag erteilt hat. Die Fähigkeit des Auftragsannahme-Akteurs zur Auftragsabwicklererzeugung ist in Bild 3.32 symbolisiert durch Schreibpfeile auf die gestrichelt umrandeten Speicher-Ovale, die jeweils einen Auftragsabwickler enthalten. Die Auftragsabwickler haben nach Abschluß des Vermittlungsvorgangs über die unbeschrifteten Kanäle eine Punkt-zu-Punkt-Verbindung zu ihren Auftraggebern. In einem lokalen Speicherbereich (Kontext) können sie jeweils den aktuellen Zustand ihrer Auftragsabwicklung festhalten. Auftragsabwickler werden nach Beendigung der Dienstausführung zerstört.

Der Auftragsannahmeakteur kann durchaus gleichzeitig für Zuschlagsmeldungen mehrerer Dienstvermittler empfangsbereit sein. Dies ist immer dann der Fall, wenn aktuell beim Ausschreibungsempfänger mindestens zwei Ausschreibungen mehr eingegangen sind, als Ausschreibungsrücknahmen beim Ausschreibungsrücknahme-Reakteur. Ein Dienstanbieter kann also gleichzeitig an mehreren Ausschreibungsverfahren beteiligt sein. Stellt man sich die gezeigten Akteure als Personen innerhalb eines Unternehmens vor, ist der Auftragsannahme-Akteur ein Sachbearbeiter, dem vom Ausschreibungsempfänger ständig Telefone auf seinen Schreibtisch gestellt werden, während ihm der Ausschreibungsrücknahme-Reakteur diese Telefone wieder wegnimmt. Dabei können sich auf dem Schreibtisch des Auftragannahme-Akteurs zu einem Zeitpunkt unterschiedlich viele Telefone befinden. Klingelt eines davon,

stellt der Auftragsannahme-Akteur sofort einen neuen Auftragsabwickler ein und meldet diesem ein Telefon an. Danach hebt er den Hörer des klingelnden Telefons auf seinem Schreibtisch ab, bedankt sich für die Annahme des Angebots, und teilt dem anrufenden Dienstvermittler die Telefonnummer des neu eingestellten Auftragsabwicklers mit. Anschließend telefonieren Auftraggeber und Auftragsabwickler miteinander zum Zwecke der Dienstausführung.

3.5.3.2 Dienstvermittler

Bild 3.33 zeigt den Aufbau eines Dienstvermittlers (Kundenseite). Dienstvermittler dienen Auftraggebern zur Vermittlung eines Auftragsabwicklers. Sie bestehen intern aus drei Akteuren: einem Vermittlungsinitiator (Ausschreiber), einem Angebotsempfänger und einem Zuschlagsbestätigungsempfänger. Der Auftraggeber (links) richtet seinen Vermittlungsauftrag über die gezeigte Auftrag-Rückmeldungs-Schnittstelle an den Vermittlungsinitiator. Dabei gibt er im Argument den dienstspezifischen Teil des Ausschreibungs- und Ausschreibungsrücknahme-Subjects an. Der Vermittlungsinitiator veröffentlicht daraufhin eine Ausschreibung und geht anschließend in einen passiven Wartezustand über. Die Ausschreibung hat die Form einer Auftragsveröffentlichung. Durch das damit verbundene Rückmeldungsabonnement wird der Angebotsempfänger empfangsbereit für Angebote. Da die Angebotsrückmeldungen der Dienstanbieter durch den Softwarebus-Akteur zwangssequentialisiert werden, wird irgendein Angebot als erstes beim Angebotsempfänger eintreffen.

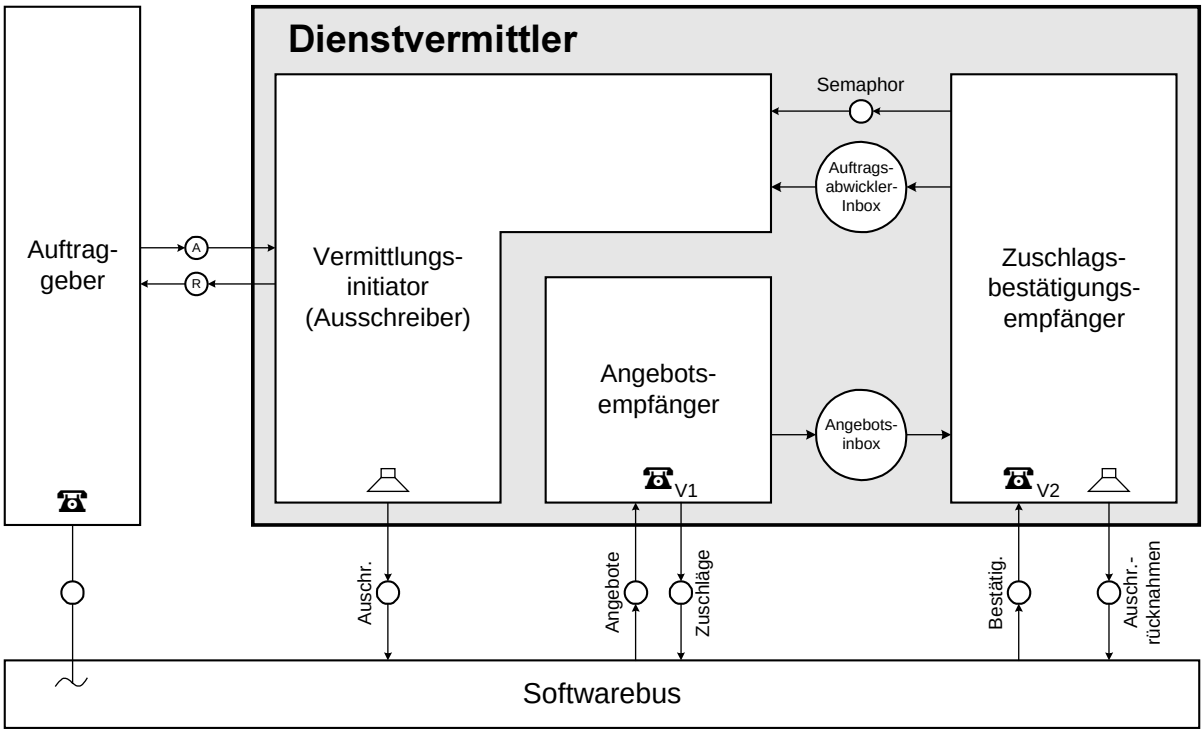


Bild 3.33 Aufbau eines Dienstvermittlers

Der Eingang des ersten Angebots löst, wie jedes Busereignis, den Aufruf einer Callback-Routine durch den externen Busakteur aus. Hier ist es diejenige Callback-Routine, die die Aktionen des Angebotsempfängers beschreibt. Bei einem solchen Callback wird grundsätzlich

immer über ein bestimmtes Routinenargument das Subject mitgeteilt, unter dem das empfangene Busereignis ausgelöst wurde (vgl. Seite 59). In diesem Fall ist es das Angebots-Inbox-Subject, das bei Veröffentlichung der Ausschreibung generiert und abonniert wurde. Dieses Angebots-Inbox-Subject legt der Angebotsempfänger in dem rechts neben ihm gezeigten Speicher ab, damit der Zuschlagsbestätigungsempfänger später darauf zugreifen kann. Daraufhin kündigt er das Abonnement dieses Subjects sofort wieder, um keine weiteren Angebote mehr zu empfangen. Anschließend erteilt er dem Dienstanbieter, der das empfangene Angebot gemacht hat, den Zuschlag, indem er als Rückmeldung eine weitere Auftragsveröffentlichung auf den Bus gibt.

Das damit verbundene Rückmeldungsabonnement macht nun den Zuschlagsbestätigungsempfänger empfangsbereit. Empfängt dieser schließlich die Bestätigung des Zuschlags, gibt er als erstes die Ausschreibungsrücknahme in Form einer gewöhnlichen Veröffentlichung auf den Bus. Innerhalb dieser Veröffentlichung teilt er das Angebots-Inbox-Subject mit, das in dem Speicher unten links neben ihm durch den Angebotsempfänger bereitgestellt wurde. Das zusammen mit der Zuschlagsbestätigung erhaltene Inbox-Subject des vom Dienstanbieter bereitgestellten Auftragsabwicklers legt er im Speicher darüber ab. Anschließend befreit er durch die Belegung einer Semaphorstelle den Vermittlungsinitiator aus dessen Wartezustand. Der Vermittlungsinitiator übergibt daraufhin als Rückmeldung zum Vermittlungsauftrag das Auftragsabwickler-Inbox-Subject an den Auftraggeber.

Dienstvermittler haben während des Vermittlungsvorgangs die Master-Rolle. Sie sind im Gegensatz zu Dienstanbietern immer nur an genau einem Vermittlungsvorgang beteiligt. Sie sind universell einsetzbar, da der dienstspezifische Teil der verwendeten Subjects für Ausschreibung und Ausschreibungsrücknahme erst im Vermittlungsauftrag festgelegt wird.

3.5.3.3 Workprozeß

Bild 3.34 zeigt den Aufbau eines Workprozesses. Wie bereits erwähnt, gehören zu einem Workprozeß mehrere Dienstleister (Bildmitte). Diese Dienstleister werden von einem Workprozeßverwaltungs-Akteur (links) erzeugt und gesteuert. In Bild 3.34 ist die bereits erfolgte Erzeugung von N Dienstleistern durch den Schreibpfeil auf den gestrichelt umrandeten, die Dienstleister enthaltenden Speicherbereich symbolisiert. Die Steuerung der Dienstleister geschieht über die horizontal verlaufenden Auftrags- und Rückmeldungs-Kanäle, wobei mit Steuerung das Aktivieren und Deaktivieren bestimmter Dienstleister gemeint ist. Innerhalb der Dienstleister sind die aktuell vorhandenen Auftragsabwickler gezeigt. Ihre Anzahl ändert sich über der Zeit. Die Indizierung soll verdeutlichen, daß die Zahl der aktuell vorhandenen Auftragsabwickler für jeden Dienstleister verschieden sein kann: Ein Dienstleister i enthält zum dargestellten Zeitpunkt M_i Auftragsabwickler. Die Dienstanbieter sind nicht gezeigt. Ferner sind die verschiedenen Kommunikationskanäle zum Softwarebus zu jeweils einem Kanal pro Dienstleister verdichtet worden. Die Auftragsabwickler bedienen sich während der Dienstausführung der unten dargestellten Such- und Zugriffsakteure, um auf transiente oder persistente Daten zuzugreifen. Mit transienten Daten sind solche Daten gemeint, die nur für die Akteure des dargestellten Workprozesses zugänglich sind und nur während der Lebensdauer dieses Workprozesses existieren. Persistente Daten sind dagegen potentiell für sämtliche Akteure der Applikationsebene zugänglich und existieren über die Lebensdauer eines

Workprozesses hinaus. Sie werden in den verschiedenen Datenbanken der Datenbankebene abgelegt. Jene Such- und Zugriffsakteure, die den Zugriff auf persistente Daten erlauben, müssen also die Fähigkeit haben, mit den Datenbankverwaltern aus Bild 3.1 auf Seite 23 zu kommunizieren.

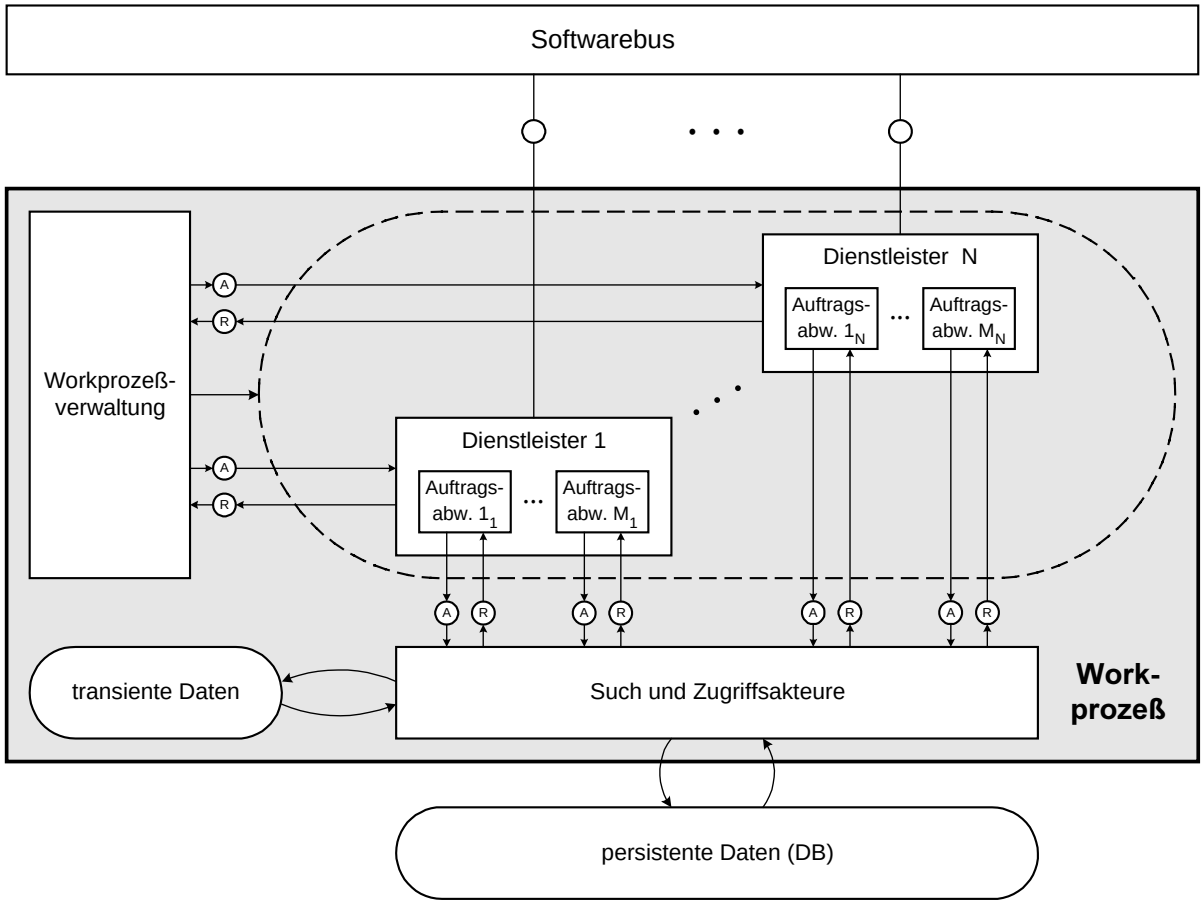


Bild 3.34 Aufbau eines Workprozesses

Das in Bild 3.34 gezeigte Aufbaumodell ist bezüglich des Datenbankzugriffs noch sehr grob, damit die unterschiedlichen Ausprägungen der verfügbaren Datenbanksysteme vereinigt werden können. Bei Entwurf eines konkreten Anwendungssystems muß dieser Teil des Aufbaumodells für jedes innerhalb der Anwendung verwendete Datenbanksystem separat präzisiert werden.

3.5.3.4 Auftragsabwickler

Bild 3.35 zeigt den inneren Aufbau eines Auftragsabwicklers. Ein solcher Auftragsabwickler teilt sich in mehrere Teilauftragsbearbeiter auf, die von einem Dispatcher (links) mit den entsprechenden Teilaufträgen bedient werden. Dieser Dispatcher ist der Abonnent des Inbox-Subjects des Auftragsabwicklers. Er empfängt zentral alle vom Auftraggeber kommenden Instruktionen. Diese Instruktionen werden innerhalb zusammengesetzter Busnachrichten codiert. Anhand des Inhalts des ersten Nachrichtenfeldes erkennt der Dispatcher, welcher Teilauftragsbearbeiter zuständig ist, und leitet die Instruktion an diesen weiter. Der zuständige Teilauftragsbearbeiter führt dann den geforderten Teilauftrag aus. Dabei bedient er sich der

Such- und Zugriffsakteure (unten) und arbeitet auf dem Inhalt des auftragsabwicklerlokalen Auftragskontextes (rechts). Unter Umständen erfolgt nach Teilauftragsausführung über den Kanal nach oben zum Softwarebus eine Rückmeldung an den Auftraggeber. Um solche Rückmeldungen überhaupt zu ermöglichen, gibt der Auftraggeber im allgemeinen im allerersten Teilauftrag (Initialauftrag) das Inbox-Subject bekannt, unter dem er über den Softwarebus erreichbar ist. Der Initialauftragsbearbeiter speichert dieses Subject dann im Auftragskontext, so daß es für alle Teilauftragsbearbeiter zugänglich ist.

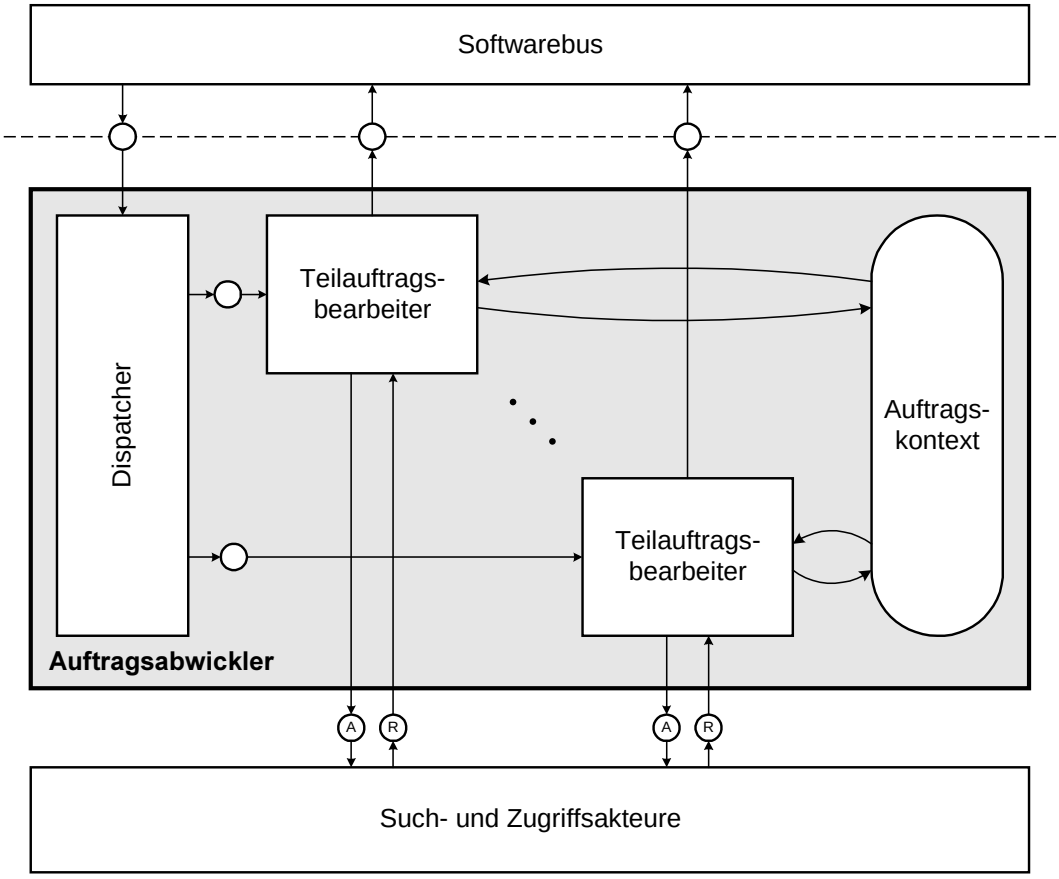


Bild 3.35 Aufbau eines Auftragsabwicklers

Während Dienstanbieter unabhängig von der angebotenen Dienstleistung immer gleich aussehen, handelt es sich bei den Auftragsabwicklern um dienstleistungsspezifische Komponenten. Innerhalb eines Dienstleisters sind alle Auftragsabwickler gleich aufgebaut. Auftragsabwickler für verschiedene Dienstleistungen können sich jedoch stark unterscheiden. Diese Unterschiede äußern sich in der Anzahl und den Fähigkeiten der Teilauftragsbearbeiter sowie in Struktur und Umfang des Auftragskontextes. Die Erfindung von Auftragsabwicklertypen ist gleichbedeutend mit der Erfindung von Dienstleistungen und Teil des konkreten Anwendungsentwurfs.

3.5.3.5 Auftraggeber

Bild 3.36 zeigt als Gegenstück zum Auftragsabwickler den Aufbau eines Auftraggebers. Neben dem Auftraggeber in der Bildmitte ist oben der Kundenakteur (GUI oder Auftragsabwickler) und links der vom Auftraggeber in Anspruch genommene Dienstvermittler gezeigt.

Ein Auftraggeber bietet dem Kundenakteur an der Schnittstelle zum Befehlsumsetzer einen Katalog von Kommandos an, über die der Kunde eine Dienstleistung der Applikationsebene in Anspruch nehmen kann. Dabei muß sich der Kunde an ein festgelegtes Protokoll halten. Bei Aufruf eines erlaubten Kommandos sorgt der Befehlsumsetzer für dessen Ausführung, indem er Teilaufgaben an die weiteren mit ihm über Auftrag-Rückmeldungs-Schnittstellen verbundenen Akteure delegiert. Einer dieser Akteure ist der Dienstvermittler, die weiteren Akteure sind die sogenannten Dialogschrittakteure innerhalb des Auftraggebers. Der Befehlsumsetzer hat also die Funktion, Kundenkommandos in Auftragssequenzen an die Dialogschrittakteure bzw. den Dienstvermittler umzusetzen. Das erste Kommando des Kundenakteurs wird grundsätzlich in einen Vermittlungsauftrag an den Dienstvermittler umgesetzt, damit ein Auftragsabwickler auf Applikationsebene zur Dienstausführung bereitgestellt wird. Weiterhin kann durch den ersten Kundenbefehl bereits ein Dialogschrittakteur dazu veranlaßt werden, einen ersten Teilauftrag an den Auftragsabwickler zu versenden. Werden explizite

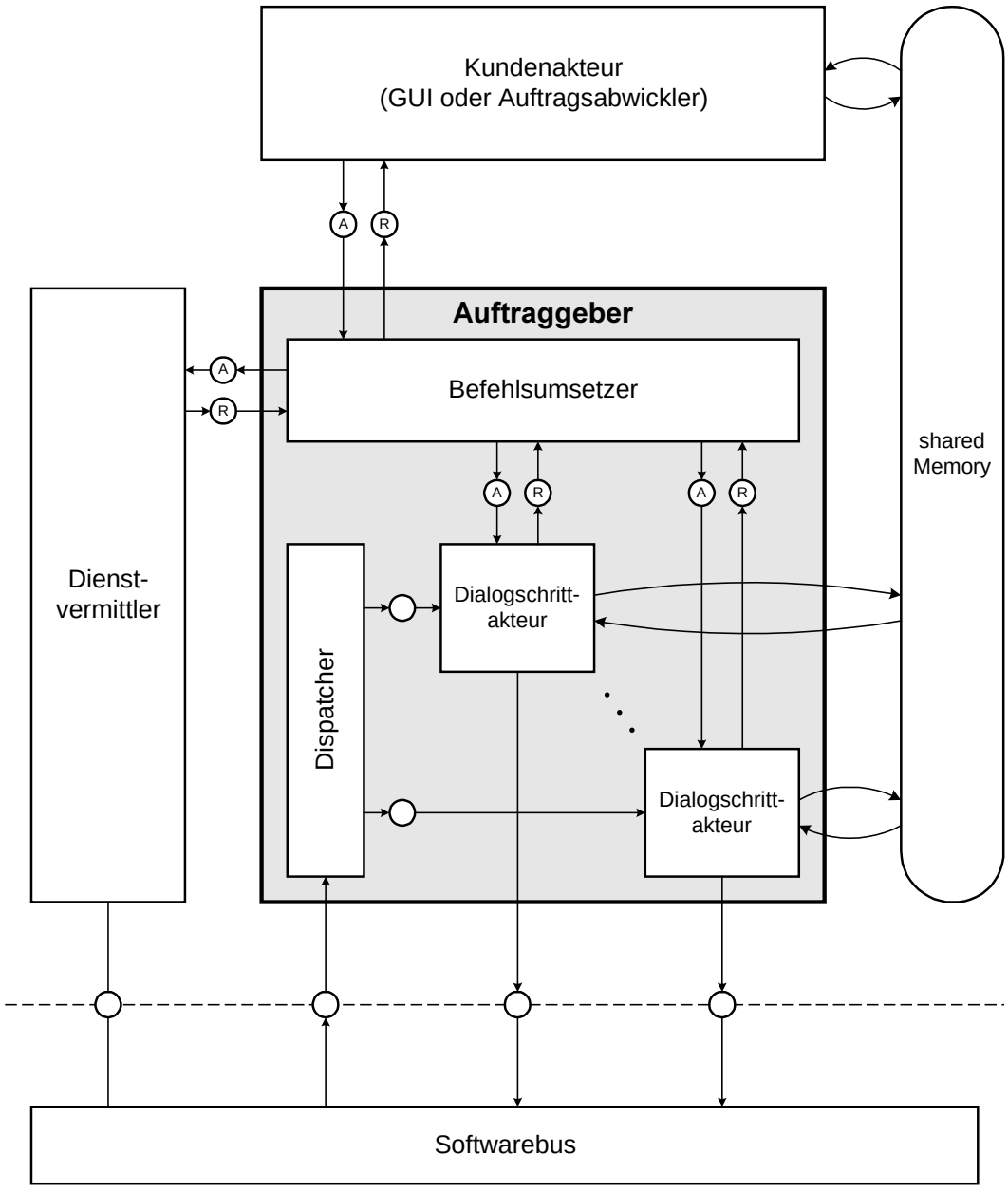


Bild 3.36 Aufbau eines Auftraggebers

Rückmeldungen des Auftragsabwicklers gewünscht, muß ebenfalls während der Umsetzung des ersten Kundenkommandos ein Inbox-Subject für den Auftraggeber abonniert werden und im ersten Teilauftrag an den Auftragsabwickler übergeben werden (vgl. Abschnitt 3.5.3.4). Dieses Abonnement macht den Dispatcher des Auftraggebers (im Bild 3.36 links unten) empfangsbereit für Rückmeldungen des Auftragsabwicklers. Die Teilaufträge an den Auftragsabwickler und die Rückmeldungen des Auftragsabwicklers an den Auftraggeber werden grundsätzlich innerhalb zusammengesetzter Busnachrichten formuliert. Wie beim Auftragsabwickler, erkennt auch der Dispatcher des Auftraggebers anhand der im ersten Nachrichtenfeld enthaltenen Nachricht, für welchen Dialogschrittakteur eine eingegangene Rückmeldung bestimmt ist, und leitet sie an diesen weiter. Solche Rückmeldungen des Auftragsabwicklers enthalten im allgemeinen Daten, die für den Kundenakteur bestimmt ist – beispielsweise eine Treffermenge bezüglich einer Datenbankabfrage durch einen Teilauftragsbearbeiter. Diese Daten können über den rechts dargestellten gemeinsam genutzten Speicher (shared Memory) zum Kundenakteur weitergegeben werden.

Nach dem ersten Kundenkommando folgen im allgemeinen weitere Kundenkommandos, die vom Befehlsumsetzer in Aufträge an einen oder mehrere Dialogschrittakteure umgesetzt werden. So kann ein einziges Kommando des Kunden eine ganze Reihe von Teilaufträgen an den Auftragsabwickler auslösen. Die Teilauftragsvergabe verläuft in Form von gewöhnlichen Veröffentlichungen unter dem Inbox-Subject des Auftragsabwicklers. Dabei bietet das Softwarebussystem von sich aus keine Möglichkeit an, auf die eventuelle Rückmeldung eines Teilauftragsbearbeiters zu warten. Um also überhaupt Dialogschritte vom Typ Auftrag-Rückmeldung zu ermöglichen, muß auf Auftraggeberseite ein entsprechender Synchronisationsmechanismus zur Verfügung gestellt werden. Bild 3.37 zeigt in einer detaillierteren Darstellung des Auftraggebers, wie dieser Synchronisationsmechanismus realisiert wird. Der Einfachheit halber sind in diesem Bild nur für einen Dialogschrittakteur die Kanäle zum Befehlsumsetzer und Dispatcher eingezeichnet.

Bei Eingang eines Kommandos des Kundenakteurs sorgt der Teilauftragsvergabe-Akteur des Befehlsumsetzers (oben) für die Umsetzung des Kundenbefehls in entsprechende Teilaufträge für den Auftragsabwickler. Dazu beauftragt er jeweils die Sender der Dialogschrittakteure zur Abgabe eines Teilauftrags. An den Teilauftragsbearbeiter zu übergebende Daten entnimmt der Sender dabei entweder dem Sendeauftrag des Teilauftragsvergabe-Akteurs oder aber dem Shared-Memory-Bereich rechts neben ihm. Sind auf diese Weise alle Teilaufträge auf den Weg gebracht, beauftragt der Teilauftragsvergabe-Akteur den unter ihm liegenden Synchronisationsakteur, ihn zu benachrichtigen, wenn alle Rückmeldungen bezüglich der vergebenen Teilaufträge eingegangen sind, und geht anschließend in einen passiven Wartezustand über. Dabei teilt er dem Synchronisationsakteur mit, wie viele der abgegebenen Teilaufträge beim Auftragsabwickler zu Rückmeldungen führen, d.h. wie viele Rückmeldungen eingehen müssen, bis der Teilauftragsvergabe-Akteur „geweckt“ werden darf.

Der Dispatcher leitet nun bei Eingang einer Rückmeldung diese an den Empfänger innerhalb des zugehörigen Dialogschrittakteurs weiter. Der Empfänger schreibt die zusammen mit der Rückmeldung vom Teilauftragsbearbeiter gelieferten Daten in den Shared-Memory-Bereich und meldet anschließend dem Synchronisationsakteur, daß die Rückmeldung bezüglich des vom Sender abgeschickten Teilauftrags eingegangen ist. Sobald auf diese Weise die erforderliche Anzahl an Rückmeldungsempfangs-Mitteilungen beim Synchronisationsakteur einge-

troffen ist, befreit dieser den Teilauftragsvergabe-Akteur aus dessen Wartezustand. Der Teilauftragsvergabe-Akteur kann daraufhin die erfolgreiche Ausführung des Kundenkommandos an den Kundenakteur zurückmelden. Der erläuterte Synchronisationsmechanismus läßt sich mit Mehrmarken-Semaphoren realisieren.

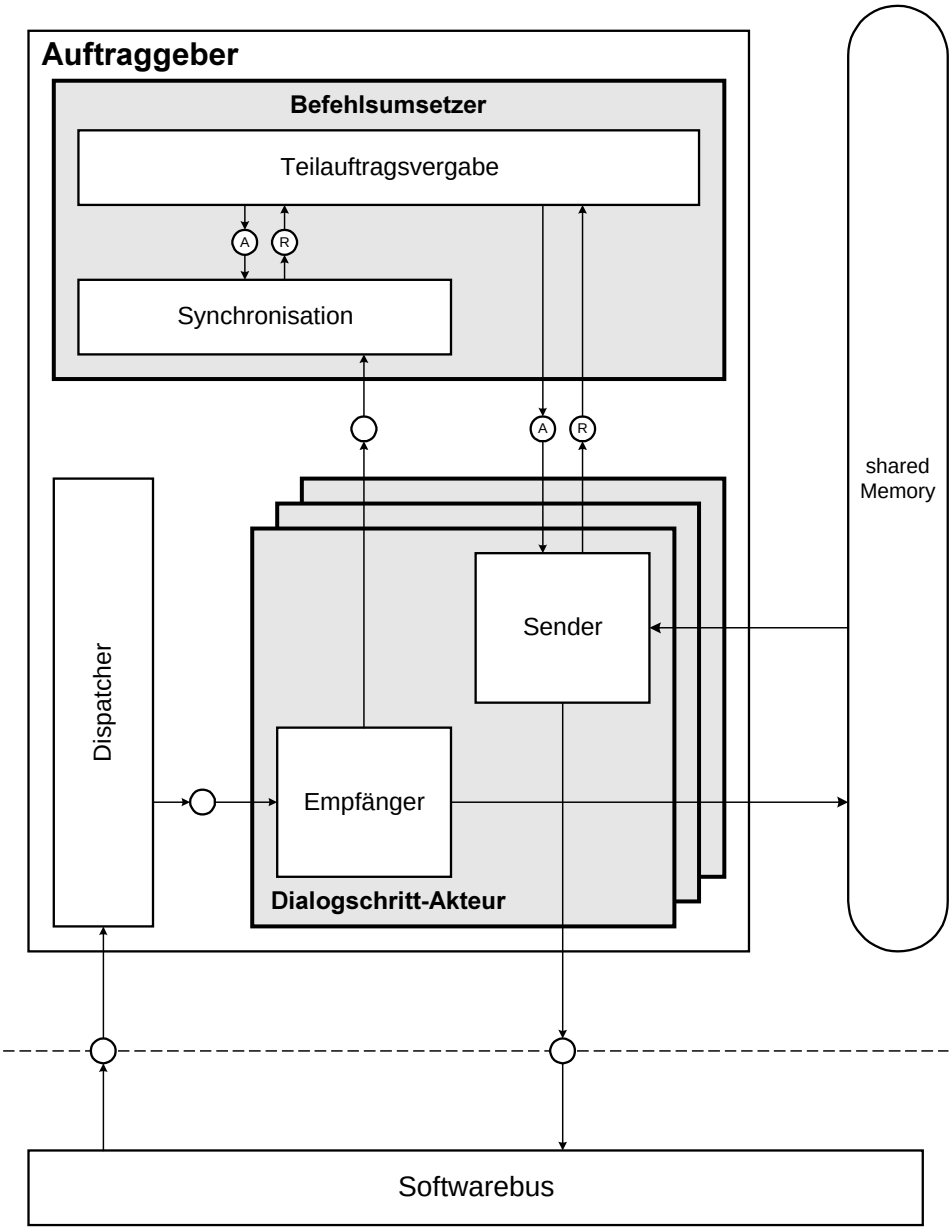


Bild 3.37 Befehlsumsetzer und Dialogschrittakteur

Auch die Auftraggeber sind, genauso wie die Auftragsabwickler, dienstleistungsspezifische Komponenten. Innerhalb eines konkreten Anwendungssystems treten daher zur Nutzung der verschiedenen Dienstleistungen Auftraggeber unterschiedlichen Typs auf. Auftraggeber eines bestimmten Typs sind festgelegt auf die Kommunikation mit Auftragsabwicklern eines bestimmten Typs. In den meisten Fällen werden Auftragsabwicklertypen und Auftraggebertypen paarweise entworfen und stehen in einer 1:1-Beziehung. Es ist aber auch denkbar, daß ein Auftragsabwicklertyp mit mehreren Auftraggebertypen verträglich ist.

3.5.4 Eingeführte Smalltalk-Klassen

Im folgenden werden nun die Klassen vorgestellt, deren Exemplare die im vorigen Abschnitt behandelten Akteure in Smalltalk repräsentieren. Die Abschnitte 3.5.4.1 und 3.5.4.2 sollen zunächst einen kurzen Überblick über alle Klassen geben, die zur Benutzung durch den Anwendungsprogrammierer vorgesehenen sind. In Abschnitt 3.5.4.3 folgt schließlich eine genauere Betrachtung aller Klassen des Multiclient-Multiserver-Klassengerüstes inklusive ihrer Vererbungsbeziehungen. Der vollständige Smalltalk-Quelltext zu diesen Klassen befindet sich in Anhang C.

3.5.4.1 Klassen auf Workprozeß-Seite

Bild 3.38 zeigt in einer Gegenüberstellung zwei ER-Diagramme. Auf der linken Seite befindet sich das bereits in Abschnitt 3.5.1 zur Begriffsklärung verwendete Diagramm aus Bild 3.27, rechts daneben sind innerhalb eines ähnlichen ER-Diagramms die Bezeichnungen der zugehörigen Smalltalk-Klassen gezeigt. Eine Sonderrolle spielt hier der Entitätstyp-Knoten mit der Aufschrift „SeviceProcessor class“ ganz rechts, der für eine Smalltalk-„Meta“-Klasse steht. Entitäten dieses Typs sind keine gewöhnlichen Smalltalk-Objekte, sondern Smalltalk-Klassenobjekte, und zwar solche Klassenobjekte, die für die Klasse ServiceProcessor oder ihre Unterklassen zuständig sind.

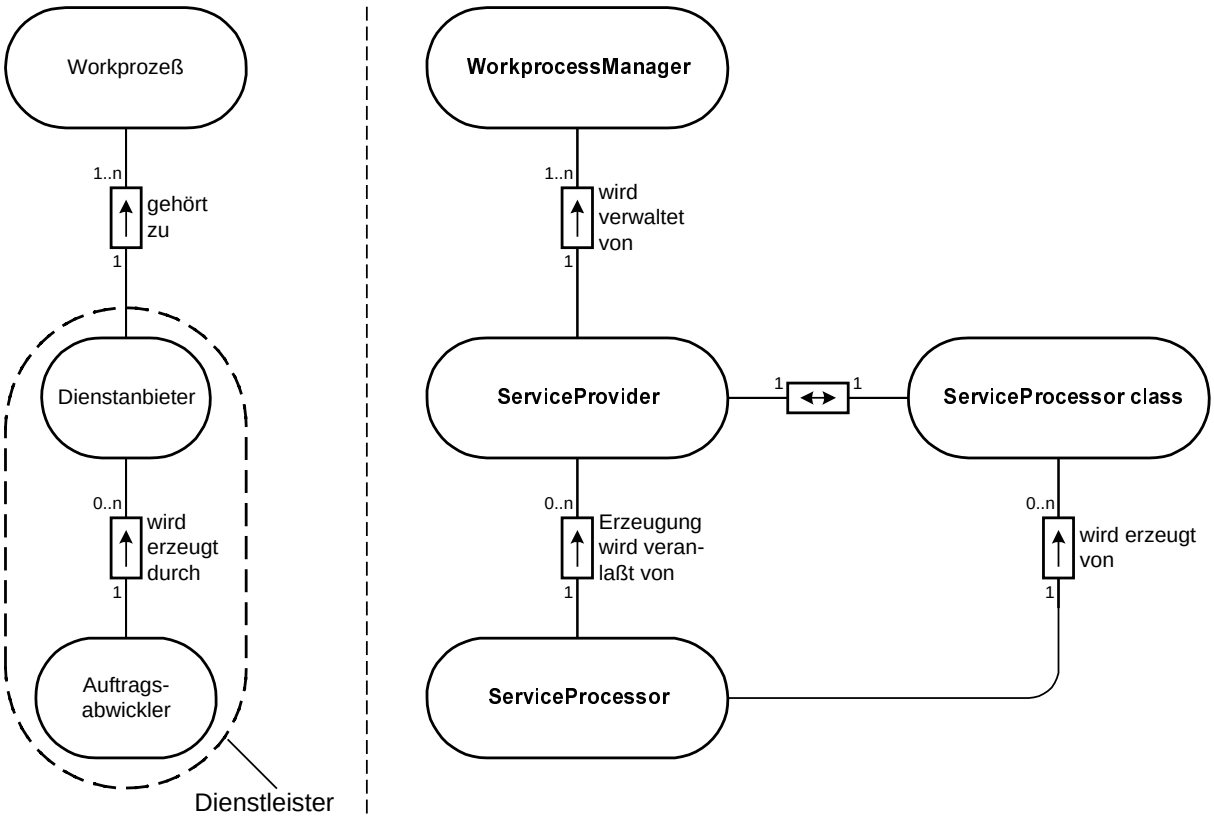


Bild 3.38 Smalltalk-Klassen auf Workprozeß-Seite

WorkprocessManager

WorkprocessManager ist eine aufgeschobene Klasse. Unterklassen von WorkprocessManager werden beim Entwurf einer konkreten Anwendung gebildet. Exemplare solcher WorkprocessManager-Unterklassen repräsentieren Workprozeß-Verwalter für Workprozesse eines bestimmten Typs (vgl. Bild 3.34 und Abschnitt 3.1.1.1). Sie übernehmen die Erzeugung und Steuerung der Dienstleister. Ferner sind sie für die Eröffnung einer Bussitzung durch Erzeugen eines Sitzungsverwalters (Exemplar der Klasse RVSessionManager) zuständig. Jeder Workprozeß besitzt genau einen solchen Sitzungsverwalter, den alle Akteure innerhalb des Workprozesses gemeinsam zur Buskommunikation benutzen.

Die Klasse WorkprocessManager stellt bereits Methoden zur Workprozeßsteuerung, Dienstleistererzeugung und Dienstleistersteuerung zur Verfügung. In den konkreten Unterklassen muß lediglich noch die aufgeschobene Methode ‘createServices’ definiert werden, in der bei Aktivierung eines Workprozesses die Erzeugung aller Dienstleister veranlaßt wird.

ServiceProvider

Exemplare der Klasse ServiceProvider repräsentieren Dienstanbieter. Ihre Erzeugung wird bei Workprozeß-Aktivierung vom entsprechenden WorkprocessManager-Exemplar veranlaßt. Während eines Vermittlungsvorgangs kommunizieren sie mit Dienstvermittlern auf Kundenseite und sind zuständig für die Erzeugung von Auftragsabwicklern. Zur Auftragsabwickler-Erzeugung bedienen sie sich eines ihnen eindeutig zugeordneten Auftragsabwickler-Klassenobjekts, das ihnen unmittelbar nach ihrer Erzeugung durch den Workprozeßverwalter mitgeteilt wird („ServiceProcessor class“, rechts im Bild 3.38).

ServiceProcessor

ServiceProcessor ist wiederum eine aufgeschobene Klasse. Unterklassen von ServiceProcessor repräsentieren Auftragsabwicklertypen. Exemplare solcher Unterklassen stellen also dienstleistungsspezifische Auftragsabwickler dar. Nach Erzeugung durch einen Dienstanbieter kommunizieren sie über den Softwarebus mit ihren Auftraggebern. Die Klasse ServiceProvider stellt bereits Methoden zum Abonnieren und Kündigen des Auftragsabwickler-Inbox-Subjects, eine Methode zur ordnungsgemäßen Beendigung der Auftragsabwicklung sowie die Implementierung des Dispatchers (vgl. Bild 3.35) zur Verfügung. Das Verhalten der einzelnen Teilauftragsbearbeiter muß vom Anwendungsprogrammierer in Form von Methoden in den Unterklassen festgelegt werden.

3.5.4.2 Klassen auf Kundenseite

Bild 3.39 zeigt in einer ähnlichen Gegenüberstellung wie vorhin die auf Kundenseite eingeführten Smalltalk-Klassen. Oben im Bild befindet sich noch einmal das ER-Diagramm mit den Akteursbezeichnungen aus Bild 3.28, unten dasjenige mit den zugehörigen Smalltalk-Klassenbezeichnungen. Falls der Kunde ein GUI-Akteur ist, ist er in Smalltalk meist durch ein Exemplar einer Unterklasse von ApplicationModel (eine Smalltalk-Systemklasse) realisiert. Des weiteren können auch Auftragsabwickler – also Exemplare der Klasse ServiceProcessor – als Kunden auftreten.

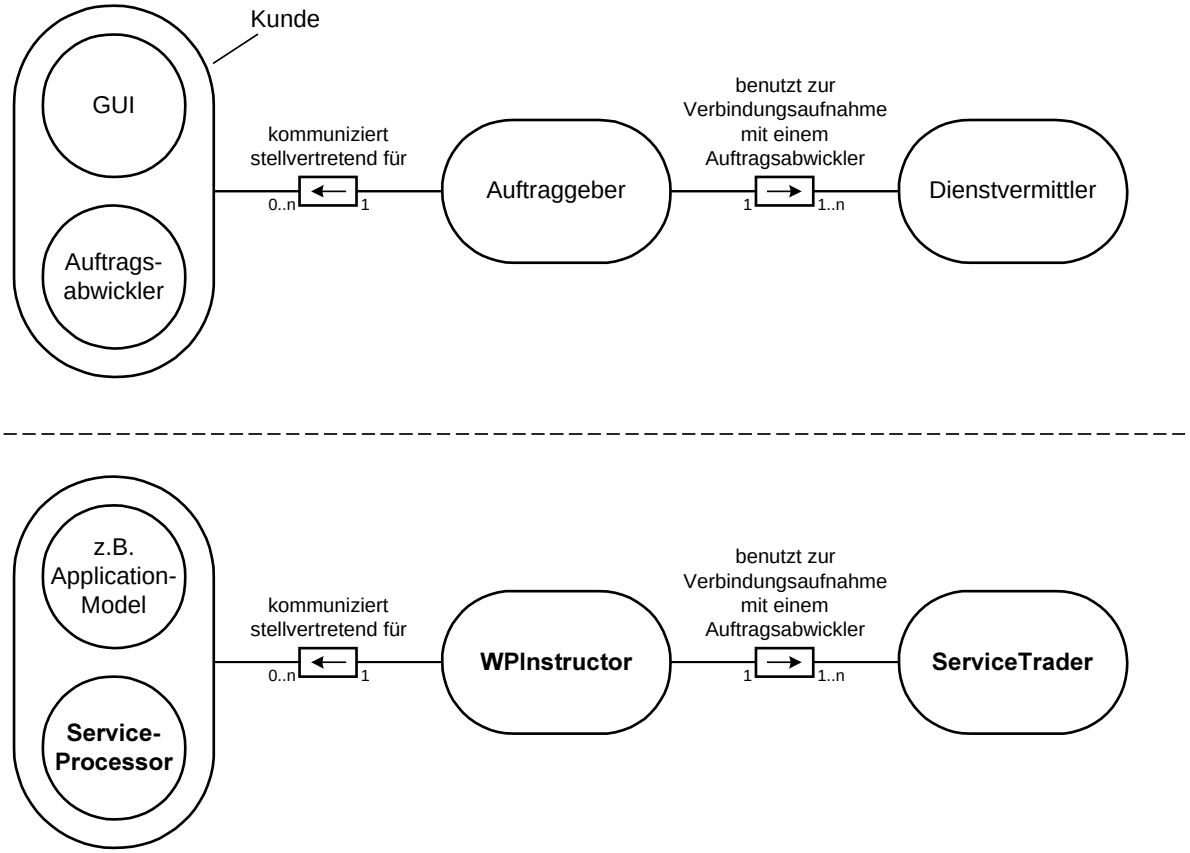


Bild 3.39 Smalltalk-Klassen auf Kundenseite

WPInstructor

WPInstructor ist eine aufgeschobene Klasse. Ihre Unterklassen vertreten Auftraggebertypen in Smalltalk. Exemplare dieser Unterklassen stellen also dienstleistungsspezifische Auftraggeber dar. Sie kommunizieren im Auftrag von Kunden mit Auftragsabwicklern. Die Klasse WPInstructor beinhaltet bereits die Definition von Methoden zum Abonnieren und Kündigen des Auftraggeber-Inbox-Subjects sowie zur Beschaffung eines Auftragsabwickler-Inbox-Subjects. Ferner ist ein Attribut zur Aufnahme eines Semaphor-Verwalters (Exemplar der Smalltalk-Klasse Semaphore) vorgesehen, um den in Abschnitt 3.5.3.5 behandelten Synchronisationsmechanismus zu realisieren. Auch die Implementierung des Rückmeldungs-Dispatchers (vgl. Bild 3.36) wird bereits durch die Klasse WPInstructor zur Verfügung gestellt. Auch hier muß das Verhalten der einzelnen Dialogschrittakteure vom Anwendungsprogrammierer in den Methoden der Unterklassen festgelegt werden.

ServiceTrader

Exemplare der Klasse ServiceTrader repräsentieren universell einsetzbare Dienstvermittler. Sie besitzen die Fähigkeit, gemäß dem in Abschnitt 3.5.2 vorgestellten Vermittlungsprotokoll mit mehreren Dienstanbietern über den Softwarebus zu kommunizieren. Dazu muß bei ihrer Erzeugung ein zuständiger Bussitzungsverwalter angegeben werden. Über die Methode ‘tradeService: *<serviceSubject>*’ läßt sich anschließend ein Vermittlungsvorgang anstoßen, wobei im Argument ‘serviceSubject’ der dienstspezifische Teil der bei Ausschreibung und

Ausschreibungsrücknahme verwendeten Subjects angegeben wird. Als Ergebnisobjekt erhält man einen Smalltalk-String zurück, der das Auftragsabwickler-Inbox-Subject eines auf Applikationsebene bereitgestellten Auftragsabwicklers enthält.

3.5.4.3 Implementierung

Bild 3.40 zeigt die insgesamt sechs Klassen des Multiclient-Multiserver-Klassengerüsts als grau schraffierte Knoten in einem Ausschnitt des Smalltalk-Klassenhierarchiebaums. Links sind die Klassen der Workprozeß-Seite, rechts die der Kundenseite gezeigt. Es fällt auf, daß die Klassen WPInstructor und ServiceProcessor eine gemeinsame Oberklasse (RVCallable) haben. Damit wird dem in weiten Teilen ähnlichen Aufbau der durch ihre Exemplare repräsentierten Akteure Rechnung getragen (vgl. Bild 3.35 und Bild 3.36). Bei den aufgeschobenen Klassen WPInstructor, ServiceProcessor und WorkprocessManager ist eine Unterklassenbildung durch den Anwendungsprogrammierer vorgesehen. Dies ist in Bild 3.40 durch die leeren Blattknoten unterhalb der zu den drei Klassen gehörenden Klassenknoten symbolisiert. Im folgenden werden die wichtigsten Implementierungsaspekte dieser sechs Klassen vorgestellt.

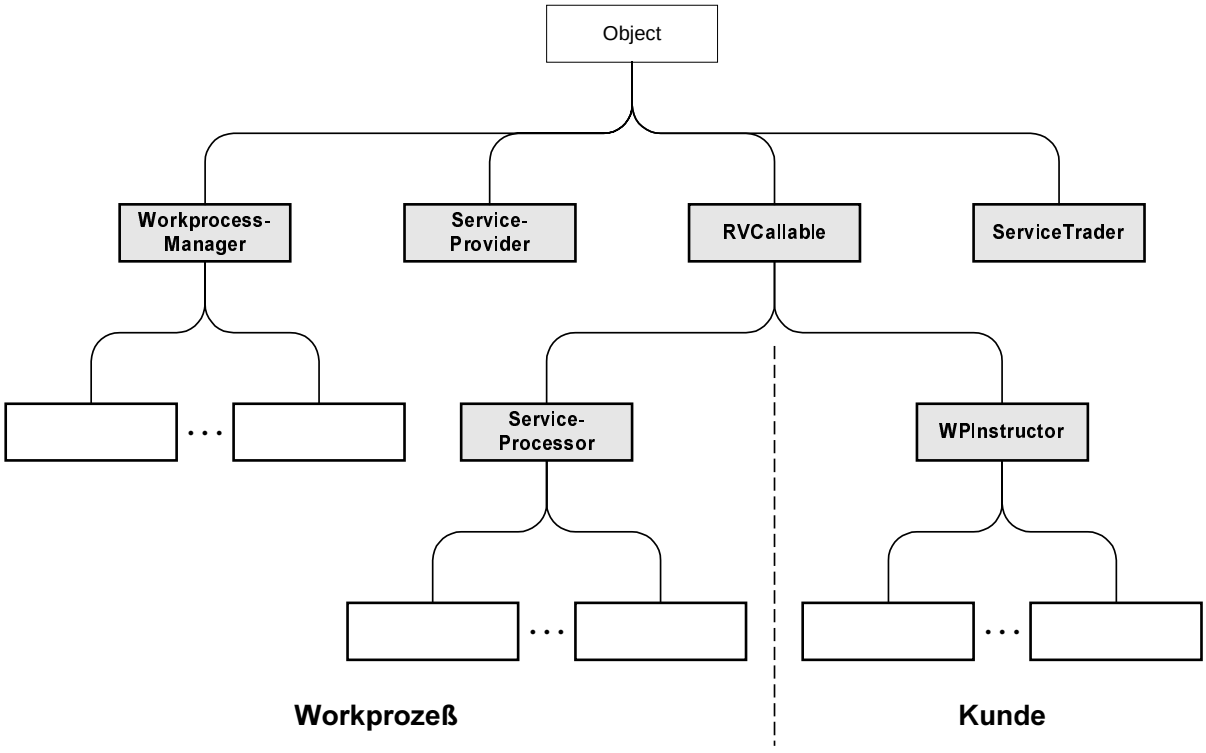


Bild 3.40 Das Multiclient-Multiserver-Klassengerüst

WorkprocessManager

Bild 3.41 zeigt die in der Klasse WorkprocessManager definierten Methoden und ihre Aufrufbeziehungen in einem Schichtungsdiagramm. Die beiden Methode ‘start’ und ‘stop’ sind öffentliche Methoden, alle weiteren Methoden sind privat. Wie bereits erwähnt, handelt es sich bei der Methode ‘createServices’ um eine aufgeschobene Methode, die in den Unterklassen neu definiert werden muß.

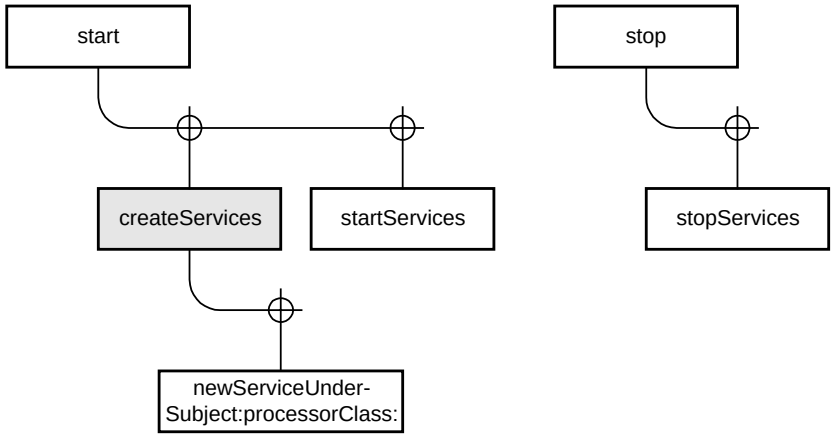


Bild 3.41 WorkprocessManager-Methoden

Mit den Methoden ‘start’ und ‘stop’ kann man ein Exemplar einer WorkprocessManager-Unterklasse dazu veranlassen, den von ihm verwalteten Workprozeß zu aktivieren bzw. zu deaktivieren. Bild 3.42 zeigt die Abläufe, die durch den Aufruf dieser beiden Methoden angestoßen werden. Nach dem Startauftrag an einen Workprozeß-Verwalter wird zunächst eine Bussitzung eröffnet, indem ein Bussitzungsverwalter als Exemplar der Klasse RVSessionManager erzeugt und an das WorkprocessManager-Attribut ‘sessionManager’ gebunden wird. Anschließend wird durch Aufruf der Methode ‘createServices’ die Erzeugung sämtlicher zum Workprozeß gehörender Dienstanbieter veranlaßt. Die dabei erzeugten ServiceProvider-Exemplare werden in einem an das WorkprocessManager-Attribut ‘services’ gebundenen Containerobjekt (Exemplar der Smalltalk-Containerklasse Set) abgelegt. Nach ihrer Erzeugung werden alle diese Dienstanbieter innerhalb der Methode ‘startServices’ aktiviert. Um die Akteure des Workprozesses schließlich empfangsbereit zu machen, wird der Bussitzungsverwalter zum Starten der Bus-Eventloop beauftragt.

Der Aufruf der WorkprocessManager-Methode ‘stop’ bewirkt zunächst die Deaktivierung aller Dienstanbieter. Diese Deaktivierung geschieht durch Aufruf der eigenen Methode ‘stopServices’. Anschließend wird der ‘services’-Container geleert, die Dienstanbieter werden der Garbage-Collection überlassen. Danach wird die Bussitzung geschlossen.

Die Klasse WorkprocessManager stellt also bereits alle Grundoperationen zur Verwaltung eines Workprozesses zur Verfügung. In den Unterklassen von WorkprocessManager, den konkreten Repräsentanten von Workprozeßtypen, muß lediglich noch die aufgeschobene Methode ‘createServices’ definiert werden, die zur Erzeugung der Dienstanbieter der zum Workprozeß gehörenden Dienstleister dient. Dabei kann von der geerbten Methode

newServiceUnderSubject: <*serviceSubject*> **processorClass:** <*serviceProcessorClass*>

Gebrauch gemacht werden. Diese private WorkprocessManager-Methode übernimmt jeweils die Erzeugung und Initialisierung eines Exemplars der Klasse ServiceProvider, das den jeweiligen Dienstanbieter repräsentiert. Das erste Argument ‘serviceSubject’ dient zur Festlegung des dienstspezifischen Teils von Ausschreibungs- und Ausschreibungsrücknahme-Subject, das zweite Argument ‘serviceProcessorClass’ gibt das für die Erzeugung von Auftragsabwicklern zuständige Klassenobjekt an. Der Rumpf der Methode ‘createServices’ einer WorkprocessManager-Unterklasse braucht also lediglich aus einer Sequenz von ‘newServiceUnderSubject: processorClass:’-Aufrufen zu bestehen.

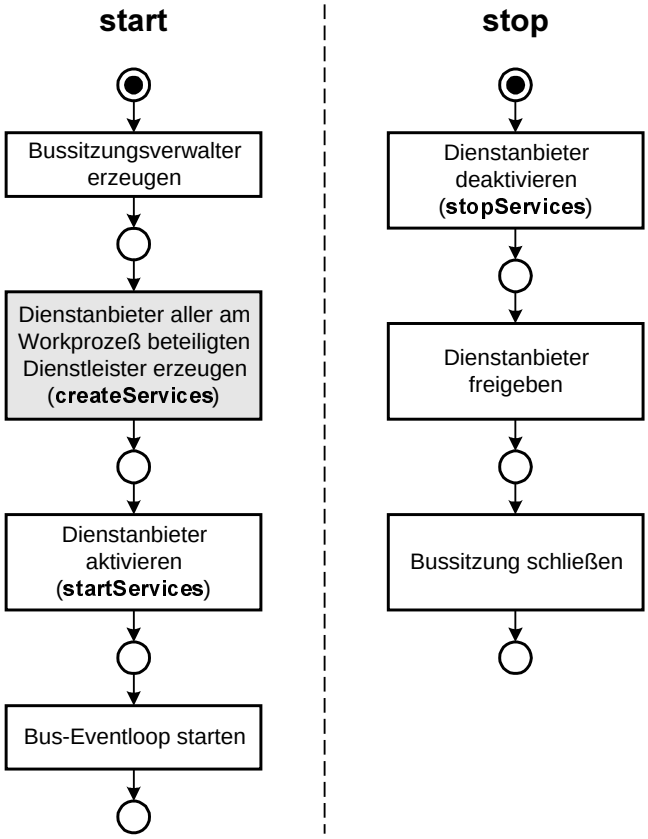


Bild 3.42 Die Methoden ‘start’ und ‘stop’ eines Workprozeßverwalters

ServiceProvider

Bei Erzeugung von ServiceProvider-Exemplaren durch die oben genannte WorkprocessManager-Methode werden folgende ServiceProvider-Attribute durch den Workprozeß-Verwalter initialisiert:

sessionManager Wird mit einem Verweis auf den durch den Workprozeßverwalter erzeugten Bussitzungsverwalter belegt.

serviceSubject Wird mit dem dienstspezifischen Teil des Ausschreibungs- und Ausschreibungsrücknahme-Subjects belegt.

processorClass Wird mit einem Verweis auf das für die Auftragsabwicklererzeugung zuständige Klassenobjekt belegt.

Die weiteren fünf ServiceProvider-Attribute werden unmittelbar nach Exemplarerzeugung durch die eigene Exemplarinitialisierungs-Methode ‘initialize’ belegt:

callForTendersCallbackBlock (Ausschreibungsempfänger-Block)
Verweist auf ein Blockobjekt, das die Anweisungen enthält, die nach Eingang einer Ausschreibung ausgeführt werden sollen.

callForTendersCancellationCallbackBlock (Ausschreibungsrücknahmereaktions-Block)
Verweist auf ein Blockobjekt, das die Anweisungen enthält, die nach Empfang einer Ausschreibungsrücknahme ausgeführt werden sollen.

arwardCallbackBlock (Auftragsannahme-Block)
Verweist ein Blockobjekt, das die Anweisungen enthält, die nach Eingang einer Zuschlagsmeldung ausgeführt werden sollen.

openOffers Wird mit einem Exemplar der Smalltalk-Containerklasse Dictionary (Feld mit assoziativem Zugriff) belegt und dient zur Aufnahme der offenen Angebote.

activeProcessors Wird mit einem Exemplar der Smalltalk-Containerklasse Set (ungeordnete Menge) belegt und dient zur Aufnahme der vom Dienstanbieter erzeugten Auftragsabwickler.

Die drei genannten Blockobjekte enthalten also die Verhaltensbeschreibungen von Ausschreibungsempfänger, Ausschreibungsrücknahme-Reakteur und Auftragsannahme-Akteur (vgl. Bild 3.32). Ihre Erzeugung geschieht durch Aufruf folgender privater ServiceProvider-Methoden:

- **callForTendersBlock**
  Gibt das Ausschreibungsempfänger-Blockobjekt zurück.
- **callForTendersCancellationBlock**
  Gibt das Ausschreibungsrücknahmereaktions-Blockobjekt zurück
- **awardBlock**
  Gibt das Auftragsannahme-Blockobjekt zurück.

Nach der vollständigen Initialisierung eines Dienstanbieter-Exemplars, die, wie gezeigt, teilweise von außen und teilweise durch eigene Methoden veranlaßt wird, stehen dem Workprozeßverwalter folgende beiden ServiceProvider-Methoden zur Aktivierung bzw. Deaktivierung des Dienstanbieters zur Verfügung:

- **start**
  Aktiviert den Dienstanbieter, indem beim Bussitzungsverwalter das Abonnement des Ausschreibungs- und des Ausschreibungsrücknahme-Subjects unter Angabe von Ausschreibungsempfänger-Block und Ausschreibungsrücknahmereaktions-Block veranlaßt wird.
- **stop**
  Deaktiviert den Dienstanbieter durch Kündigung der beiden genannten Subjects.

Bild 3.43 zeigt die in den drei Blockobjekten enthaltene Verhaltensbeschreibung in Form von Petrinetzen. Trifft eine Ausschreibung bei einem aktivierten Dienstanbieter ein, wird der Ausschreibungsempfänger-Block (links) ausgeführt. Zunächst wird ein Angebotsveröffentlichungsobjekt (in den Petrinetzen kurz: „Angebot“) in Form eines RVRequest-Exemplars erzeugt. In dieses RVRequest-Exemplar wird als Subject das erhaltene Angebots-Inbox-Subject des Dienstvermittlers und als Callback-Blockobjekt das im Attribut ‘awardCallback-Block’ bereitstehende Auftragsannahme-Blockobjekt eingetragen. Anschließend wird dieses Angebotsveröffentlichungsobjekt unter dem Angebots-Inbox-Subject im Verzeichnis der offenen Angebote (‘openOffers’-Dictionary) eingetragen. Erst dann wird das Angebot durch ‘publish:’-Auftrag an den Sitzungsverwalter auf den Bus gegeben und damit implizit ein eigenes Inbox-Subject zum Empfang des eventuellen Zuschlags abonniert.

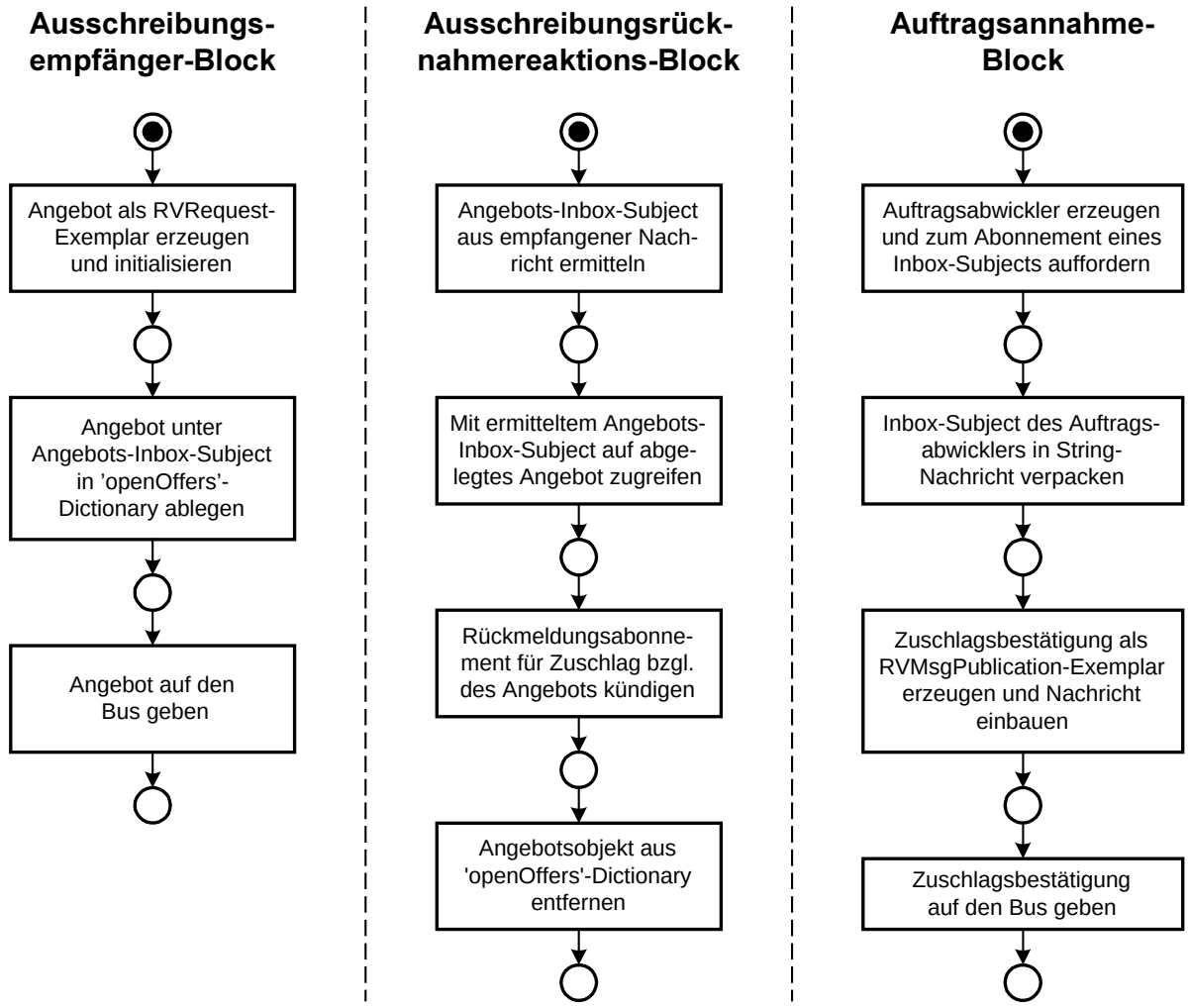


Bild 3.43 Ausschreibungsempfänger-Block, Ausschreibungsrücknahmereaktions-Block und Auftragsannahme-Block

Bei Eintreffen einer Ausschreibungsrücknahmemeldung wird der im Bild 3.43 in der Mitte gezeigte Ausschreibungsrücknahmereaktions-Block ausgeführt. Dort wird zunächst das in der Ausschreibungsrücknahme-Veröffentlichung mitgeteilte Angebots-Inbox-Subject ermittelt. Dieses wird sodann benutzt, um im Verzeichnis der offenen Angebote auf das Angebotsveröffentlichungsobjekt zuzugreifen, welches das nunmehr obsolete Angebot repräsentiert. Durch Aufruf der Methode ‘closeReplyInbox’ dieses RVRequest-Exemplars (vgl. Seite 45) wird das Rückmeldungsabonnement für den Zuschlag bezüglich des Angebots gekündigt. Anschließend wird das Angebotsveröffentlichungsobjekt aus dem Verzeichnis der offenen Angebote entfernt.

Falls vor Ausschreibungsrücknahme ein Zuschlag bezüglich eines Angebots eintrifft, wird der links im Bild 3.43 gezeigte Auftragsannahme-Block ausgeführt. Dort wird als erstes die Erzeugung eines Auftragsabwickler-Exemplars veranlaßt und der neu erzeugte Auftragsabwickler zum Abonnement eines Inbox-Subjects aufgefordert. Dieses Inbox-Subject wird daraufhin erfragt und in eine Busnachricht (Exemplar der Klasse RVStringMessage) verpackt. Anschließend erfolgt die Erzeugung eines Zuschlagsbestätigungsobjekts in Form eines Exemplars der Klasse RVMsgPublication, in das die Busnachricht sowie das erhaltene Rückmeldungs-Subject für die Zuschlagsbestätigung eingetragen werden. Schließlich wird die Zuschlagsbestätigung durch ‘publish:’-Auftrag an den Sitzungsverwalter auf den Bus gegeben.

Zur Auftragsabwickler-Erzeugung wird bei Ausführung des Auftragsannahmeblocks die private ServiceProvider-Methode

- **newProcessor**

aufgerufen. Dort wird das im Attribut ‘processorClass’ enthaltene Klassenobjekt, das für den angebotenen Auftragsabwicklertyp zuständig ist, zur Exemplarerzeugung beauftragt. Der dabei erhaltene Auftragsabwickler wird zunächst mit einem Verweis auf den Bussitzungsverwalter sowie einem Verweis auf den Dienstanbieter selbst initialisiert und anschließend zum Abonnement eines Inbox-Subjects beauftragt. Danach wird der frisch erzeugte Auftragsabwickler schließlich im Container ‘activeProcessors’ abgelegt, um ihn vor der Garbage-Collection zu bewahren. Die letzte noch unerwähnte ServiceProvider-Methode

- **removeProcessor**

dient dem Auftragsabwickler nach Beendigung der Dienstausführung zum Löschen dieses Verweises aus dem ‘activeProcessors’-Container.

RVCallable, ServiceProcessor und WPInstructor

RVCallable ist gemeinsame Oberklasse von ServiceProcessor und WPInstructor. In ihr ist die für beide Akteurstypen benötigten Komponente, der Dispatcher, implementiert. Die weiteren Merkmale von Auftragsabwicklern und Auftraggebern werden erst in den Klassen ServiceProcessor und WPInstructor festgelegt. Bild 3.44 zeigt die drei Klassen in einer bereits in Abschnitt 3.4.6.1 verwendeten Darstellungsform.

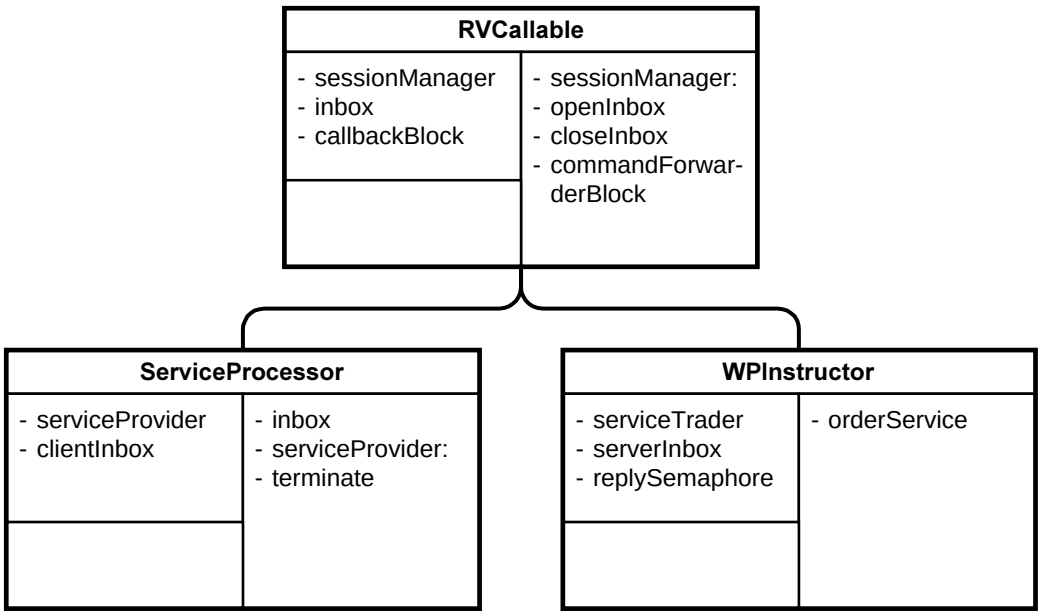


Bild 3.44 RVCallable, ServiceProcessor und WPInstructor

Das Attribut ‘sessionManager’ der Klasse RVCallable dient zur Aufnahme eines Verweises auf einen Bussitzungsverwalter. Die Belegung erfolgt unmittelbar nach Exemplarerzeugung von außen über die Methode ‘sessionManager:’. Ebenfalls unmittelbar nach Exemplarerzeugung wird das Attribut ‘callbackBlock’ durch die eigene Exemplar-Initialisierungmethode mit

einem Blockobjekt belegt, dessen Erzeugung durch Aufruf der privaten Methode ‘commandForwarderBlock’ geschieht. Dieses Blockobjekt enthält die Verhaltensbeschreibung des Dispatchers. Die Methoden ‘openInbox’ und ‘closeInbox’ dienen zum Abonnieren und Kündigen eines Inbox-Subjects für den Auftragsabwickler bzw. Auftraggeber. Als Callback-Blockobjekt wird beim Abonnement eben dieses im Attribut ‘callbackBlock’ enthaltene Dispatcher-Blockobjekt angegeben. Das erhaltene Inbox-Subject wird im Attribut ‘inbox’ abgespeichert.

Der Dispatcher ist bei Auftragsabwicklern und Auftraggebern der zentrale Empfänger von Busnachrichten. Er hat die Aufgabe, anhand des Inhalts des ersten Nachrichtenfeldes einer empfangenen Nachricht zu entscheiden, an welchen Akteur er die Busnachricht weiterzuleiten hat. Wie bereits erwähnt, ist es vorgesehen, daß das Verhalten jedes dieser Akteure vom Anwendungsprogrammierer in Form einer eigenen Methode in den jeweiligen ServiceProcessor- bzw. WPInstructor-Unterklassen festgelegt wird. Die Weiterleitung einer Busnachricht an einen Teilauftragsbearbeiter oder Dialogschrittakteur geschieht dann innerhalb des Dispatcher-Blocks durch Aufruf einer solchen Methode. Es wird vorausgesetzt, daß alle Busnachrichten an Auftragsabwickler und Auftraggeber folgende Form haben:

(command: <*Methodenname als String-Nachricht*>, argument: <*Argument-Nachricht*>)

Es muß sich also stets um eine aus zwei Nachrichtenfeldern zusammengesetzte Nachricht handeln. Das erste Feld muß den Namen ‘command’ tragen und eine String-Nachricht enthalten. Das zweite Feld muß den Namen ‘argument’ tragen und darf eine beliebige Nachricht enthalten. Der im ersten Nachrichtenfeld enthaltene String gibt die Methode an, die beim Auftragsabwickler bzw. Auftraggeber ausgeführt soll; die im zweiten Feld enthaltene Nachricht dient zur Übergabe von Daten. Im Dispatcher-Block wird nun der Aufruf der im ersten Nachrichtenfeld benannten Methode veranlaßt. Eine solche Methode muß stets genau ein Argument besitzen, in dem bei ihrem Aufruf die Argument-Nachricht aus dem zweiten Nachrichtenfeld übergeben wird.

In einer verallgemeinerten Betrachtungsweise sind Teilaufträge und Teilauftragsrückmeldungen also nichts anderes als über den Softwarebus geleitete, asynchrone Methodenaufrufe, die als zusammengesetzten Busnachrichten in der oben gezeigten Form formuliert werden. Zur Identifizierung des Empfängerobjekts dient dabei sein durch ‘openInbox’ abonniertes Inbox-Subject. Bei Umsetzung eines solchen entfernten Methodenaufrufs im Dispatcher-Block des Empfängers tritt ein semantischer Sprung auf: Aus der den Methodennamen enthaltenden String-Nachricht wird ein von der virtuellen Smalltalk-Maschine interpretierbarer Message-Selector.

Die in Bild 3.44 links unten gezeigte Klasse ServiceProcessor erweitert nun die bereits von RVCallable zur Verfügung gestellten Merkmale um weitere, speziell für Auftragsabwickler benötigte Merkmale. Das Attribut ‘serviceProvider’ wird eingeführt, um einen Verweis auf den Dienstanbieter aufzunehmen, der die Erzeugung des Auftragsabwicklers veranlaßt hat. Die Belegung dieses Attributs geschieht von außen mit Hilfe der Methode ‘serviceProvider:’ durch den erzeugenden Dienstanbieter selbst. Das zweite Attribut ‘clientInbox’ dient zur Ablage des Inbox-Subjects des Auftraggebers. Da ein Dienstanbieter den Auftragsabwickler nach seinem Inbox-Subject fragen können muß, wird des weiteren die Zugriffsmethode ‘inbox’ zur Verfügung gestellt. Die dritte Methode ‘terminate’ dient schließlich zur ordnungs-

gemäßen Beendigung der Auftragsabwicklung. Bei ihrer Ausführung wird zum einen das Inbox-Subject gekündigt (durch Aufruf von ‘closeInbox’) und zum anderen die Streichung des Verweises im ‘activeProcessors’-Container des zugehörigen Dienstanbieters veranlaßt.

In der rechts im Bild 3.44 dargestellten Unterklasse WPInstructor werden analog dazu auftraggeberspezifische Merkmale eingeführt. Das Attribut ‘serviceTrader’ dient zur Aufnahme eines Verweises auf den vom Auftraggeber in Anspruch genommenen Dienstvermittler. Nach dem Vermittlungsvorgang wird das erhaltene Auftragsabwickler-Inbox-Subject in dem weiteren Attribut ‘serverInbox’ abgelegt. Das dritte Attribut ‘replySemaphore’ enthält ein Exemplar der Smalltalk-Klasse Semaphore, um die Synchronisation mit dem Auftragsabwickler zu ermöglichen (vgl. Abschnitt 3.5.3.5 und Bild 3.37). Die Attribute ‘serviceTrader’ und ‘replySemaphore’ sowie das geerbte Attribut ‘sessionManager’ werden bereits unmittelbar nach der Exemplarerzeugung in einer Initialisierungsmethode belegt. Dabei werden die Verweise auf den Bussitzungsverwalter und den Dienstvermittler von außen als Argumente der speziellen Exemplarerzeugungs-Methode ‘newInSession:serviceTrader:’ des Klassenobjekts übergeben. In einem Klassenobjekt-Attribut mit dem Namen ‘serviceSubject’ wird schon zur Anwendungsbauzeit der dienstspezifische Teil von Ausschreibungs- und Ausschreibungsrücknahme-Subject für den repräsentierten Auftraggebertyp festgelegt.

Die WPInstructor-Methode ‘orderService’ dient zum Anstoß des Vemittlungsvorgangs bei dem über das Attribut ‘serviceTrader’ zugänglichen Dienstvermittler. Während ihrer Ausführung wird beim Klassenobjekt der Inhalt des Klassenobjekt-Attributs ‘serviceSubject’ erfragt und an den Dienstvermittler weitergegeben. Das vom Dienstvermittler zurückerhaltene Inbox-Subject wird im Attribut ‘serverInbox’ abgespeichert.

ServiceTrader

Die einzige Methode der Klasse ServiceTrader, die bei einem Dienstvermittler-Exemplar von außen aufgerufen werden darf, ist die bereits erwähnte Methode ‘tradeService:’, die zur Vermittlung eines Auftragsabwicklers an einen Auftraggeber dient:

- **tradeService:** <*serviceSubject*>

Dabei wird im Argument der dienstspezifische Teil von Ausschreibungs- und Ausschreibungsrücknahme-Subject übergeben. Als Ergebnis erhält der Aufrufer das Inbox-Subject eines auf Applikationsebene bereitgestellten Auftragsabwicklers zurück. Alle weiteren ServiceTrader-Methoden werden pro Objekt nur ein einziges Mal bereits unmittelbar nach der Exemplarerzeugung ausgeführt und dienen zur Initialisierung der Attribute. ServiceTrader-Objekte besitzen folgende Attribute:

sessionManager Wird mit einem Verweis auf einen Bussitzungsverwalter belegt. Dieser Verweis muß bereits im Argument der speziellen Exemplarerzeugungs-Methode ‘newInSession:’ von außen übergeben werden.

callForTenders Enthält das Ausschreibungsveröffentlichungsobjekt in Form eines RVRequest-Exemplars.

award Enthält das Zuschlagsveröffentlichungsobjekt, ebenfalls in Form eines RVRequest-Exemplars.

callForTendersCancellation
: Enthält das Ausschreibungsrücknahme-Veröffentlichungsobjekt, ein Exemplar der Klasse RVMsgPublication

offerInbox
: Dient nach Eingang des ersten Angebots zur Aufnahme des Angebots-Inbox-Subjects des Dienstvermittlers.

serviceProcessorInbox
: Dient nach Eingang der Zuschlagsbestätigung zur Ablage des erhaltenen Auftragsabwickler-Inbox-Subjects.

serverReadySemaphore
: Enthält ein Exemplar der Klasse Semaphore, mit dem die Synchronisation zwischen Vermittlungsinitiator und Zuschlagsbestätigungsempfänger bewerkstelligt wird (vgl. Bild 3.33).

Mit Ausnahme von ‘offerInbox’ und ‘serviceProcessorInbox’ werden alle diese Attribute bereits bei Initialisierung eines ServiceTrader-Exemplars belegt. In das Ausschreibungsveröffentlichungsobjekt (‘callForTenders’) wird dabei ferner bereits ein Callback-Blockobjekt eingetragen, das die Verhaltensbeschreibung des Angebotsempfängers enthält. Genauso wird das Zuschlagsveröffentlichungsobjekt (‘award’) mit einem Callback-Blockobjekt initialisiert, das die bei Eingang der Zuschlagsbestätigung auszuführenden Anweisungen enthält.

Die Aktionen des Vermittlungsinitiators sind in der Methode ‘tradeService’ implementiert. Bild 3.45 zeigt diese Aktionen im Petrinetz auf der linken Seite. Als erstes ergänzt der Vermittlungsinitiator anhand des übergebenen Arguments das Ausschreibungs- und das Ausschreibungsrücknahme-Subject und trägt die beiden Subjects in die zugehörigen Veröffentlichungsobjekte (‘callForTenders’ und ‘callForTendersCancellation’) ein. Anschließend veröffentlicht er die Ausschreibung via Aufruf der Methode ‘publish:’ des Bussitzungsverwalters. Nach dieser Aktion stoppt die Ausführung der Methode ‘tradeService:’: Der Vemittlungsinitiator wartet passiv, bis die Semaphormarke vom Zuschlagsbestätigungsempfänger eingegangen ist. Bei der Veröffentlichung der Ausschreibung wird implizit ein Inbox-Subject für Angebote abonniert. Trifft nun das erste Angebot ein, kommt der Angebotsempfänger-Block (Bild 3.45, Mitte) zur Ausführung. Der Angebotsempfänger veranlaßt als erstes die Kündigung des Angebots-Inbox-Subjects, um den Empfang weiterer Angebote zu verhindern. Zusätzlich speichert er das Angebots-Inbox-Subject im Attribut ‘offerInbox’ ab, damit der Zuschlagsbestätigungsempfänger es später in die Ausschreibungsrücknahmeveröffentlichung einbauen kann. Anschließend ergänzt der Angebotsempfänger das Zuschlagsveröffentlichungsobjekt um das bei Angebotsempfang erhaltene Rückmeldungs-Inbox-Subject des Dienstanbieters und gibt die dadurch repräsentierte Zuschlagsmeldung auf den Bus. Die Veröffentlichung der Zuschlagsmeldung zieht das Abonnement eines Inbox-Subjects für die Zuschlagsbestätigung nach sich. Trifft die Zuschlagsbestätigung beim Dienstvermittler ein, wird der Zuschlagsbestätigungsempfänger-Block links im Bild 3.45 ausgeführt. Die erste Aktion des Zuschlagsbestätigungsempfängers ist die Kündigung des Inbox-Subjects, unter dem die Zuschlagsbestätigung einging. Anschließend ermittelt er aus der mit der Zuschlagsbestätigung erhaltenen String-Nachricht das Inbox-Subject des vom Dienstanbieter bereitgestellten Auftragsabwicklers und legt es im Attribut ‘processorInbox’ ab. Darauf folgt die Vorbereitung der Ausschreibungsrücknahme, indem das im Attribut ‘offerInbox’ verfügbare obsolete Angebots-Inbox-Subject als String-Nachricht in das Ausschreibungsrücknahme-

Veröffentlichungsobjekt eingetragen wird. Nachdem die Ausschreibungsrücknahme dann auf dem Bus veröffentlicht worden ist, übergibt der Zuschlagsbestätigungsempfänger über den im Attribut ‘serverReadySemaphore’ enthaltenen Semaphorverwalter eine Semaphormarke an den darauf wartenden Vermittlungsinitiator. Erst jetzt wird die Ausführung der Methode ‘tradeService:’ fortgesetzt und das nun im Attribut ‘processorInbox’ verfügbare Auftragsabwickler-Inbox-Subject an den Aufrufer zurückgegeben.

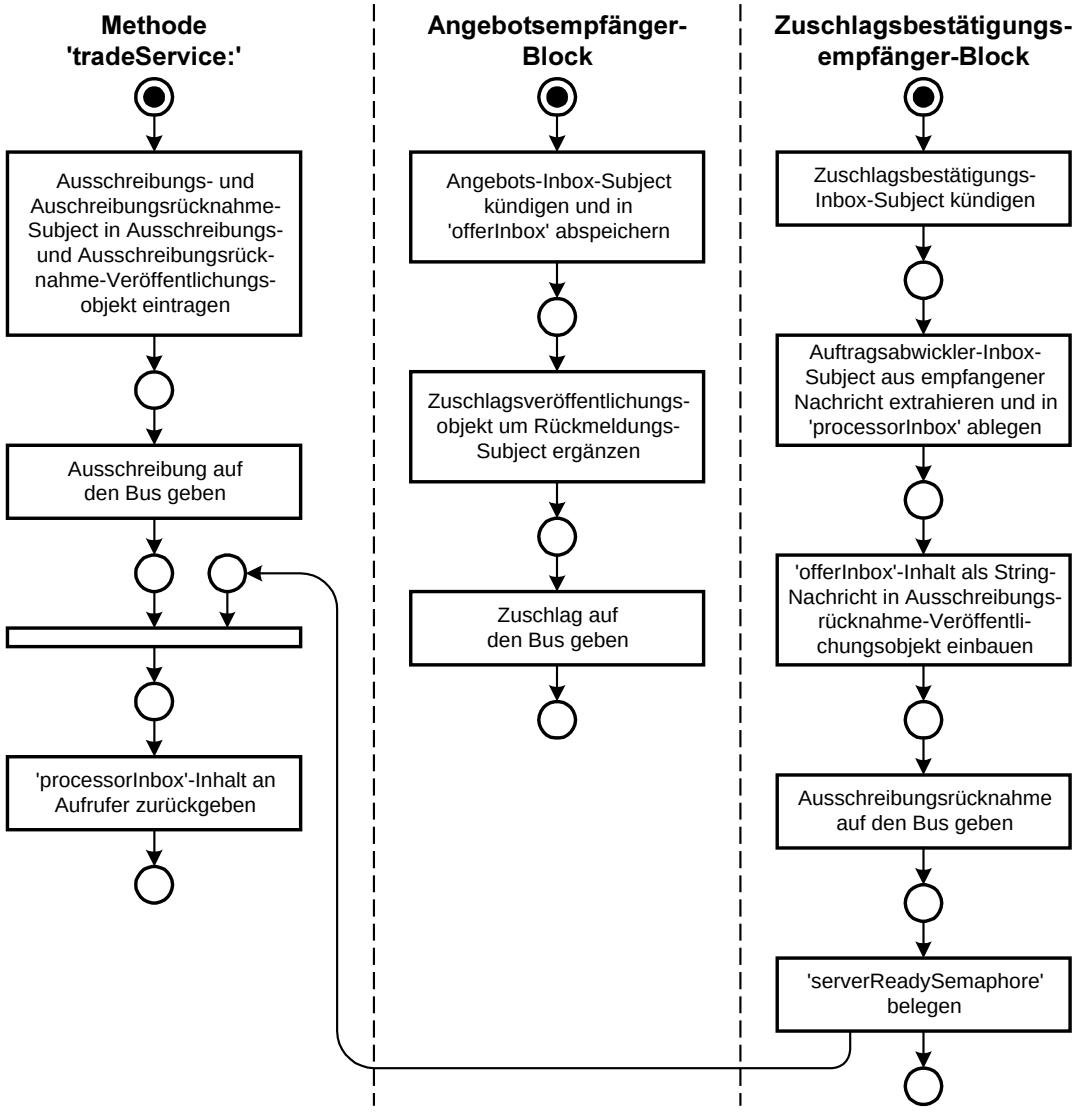


Bild 3.45 Implementierung des Dienstvermittlungsablaufs

3.6 Schicht 4: Die konkrete Anwendung

Aufbauend auf dem im vorigen Abschnitt dokumentierten Klassengerüst können nun mit wenig Aufwand Anwendungssysteme mit den in Abschnitt 3.1 geforderten Eigenschaften realisiert werden. Als Lehrbeispiel für solch ein Anwendungssystem ist ein Verwaltungssystem zur Unterstützung des Leihbetriebs einer Videothek vorgesehen. Der Entwurf dieser Beispielanwendung liegt bereits vor, die Implementierung unter Nutzung der in den Abschnitten 3.3 bis 3.5 vorgestellten Universalkomponenten ist jedoch bislang noch nicht geschehen. Um an dieser Stelle dennoch Anschauungsmaterial bieten zu können, wurde ein sehr einfacher Anwendungsprototyp realisiert, der im folgenden vorgestellt wird. Der zugehörige Smalltalk-Quelltext befindet sich in Anhang D.

3.6.1 Ein sehr einfacher Anwendungsprototyp

Bild 3.46 zeigt den technischen Aufbau des realisierten Anwendungsprototyps. Das Anwendungssystem besitzt lediglich eine Präsentations- und eine Applikationsebene. Datenbanksysteme sind nicht mit einbezogen worden. Es gibt nur jeweils einen GUI-Typ und einen Workprozeßtyp. Wie gefordert, dürfen in einem Anwendungssystem jedoch beliebig viele GUIs und beliebig viele Workprozesse vorhanden sein. GUIs und Workprozesse können sich dabei jeweils auf separaten, entfernten Rechnern befinden. Das Anwendungssystem bietet seinen Benutzern die Möglichkeit, sich über eine grafische Benutzerschnittstelle eine lange Liste von Strings portionsweise anzeigen zu lassen. Die String-Liste enthält alphabetisch sortiert alle Smalltalk-Globalsymbolnamen. Sie wird auf Applikationsebene bereitgestellt und auf Anfrage in Blöcken zu je fünf Listenelementen an die GUI-Akteure hochgereicht. Bild 3.47 zeigt die durch einen GUI-Akteur angebotene grafische Benutzerschnittstelle. Über sie kann ein Benutzer in der Liste der Smalltalk-Globalsymbolnamen blättern. Ein Mausklick auf die mit ‘Vor’ beschriftete Tastfläche bewirkt die Anzeige der nächsten fünf Listenelemente im Anzeigebereich auf der linken Seite. Ebenso führt das Betätigen des Tasters mit der Aufschrift ‘Zurück’ zur Anzeige der vorigen fünf Listenelemente. Ein Mausklick auf die Tastfläche ‘Beenden’ bewirkt das Schließen des die grafische Benutzerschnittstelle enthaltenden Fensters und zieht die Zerstörung des entsprechenden GUI-Akteurs in Bild 3.46 nach sich.

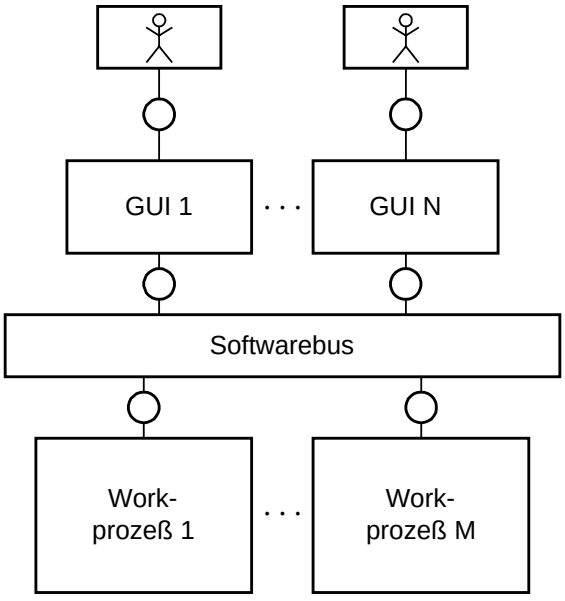


Bild 3.46 Aufbau des Anwendungsprototyps

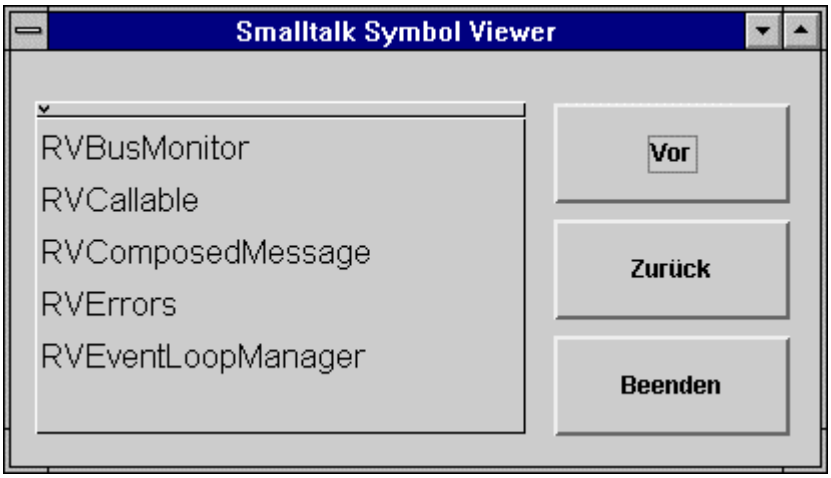


Bild 3.47 Anwendungsprototyp: GUI

3.6.2 Eingeführte Smalltalk-Klassen

Zur Realisierung dieser Anwendung wurden in Smalltalk zusätzlich vier Klassen eingeführt, die in Bild 3.48 als grau unterlegte Blattknoten in einem Ausschnitt des Smalltalk-Klassenhierarchiebaums dargestellt sind. Um die Workprozesse zu realisieren, sind die Klassen StringListWP als Unterklasse von WorkprocessManager und StringListForwarder als Unterklasse von ServiceProcessor eingeführt worden. Zur Realisierung der GUIs kommen des weiteren die Klasse StringListForwarderInstructor als Unterklasse von WPInstructor und die Klasse StringListGUI dazu. StringListGUI erbt von den beiden Smalltalk-Systemklassen ApplicationModel und Model die Merkmale, die notwendig sind, damit ein Benutzer mit ihren Exemplaren über eine grafische Benutzerschnittstelle kommunizieren kann.

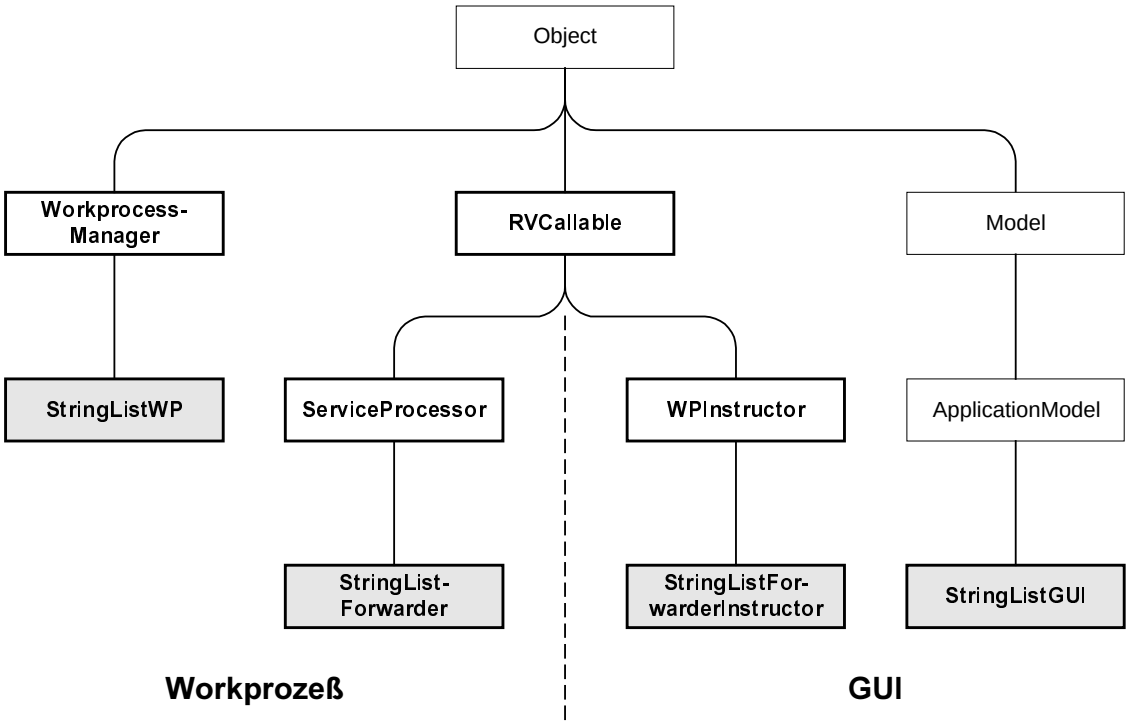


Bild 3.48 Smalltalk-Klassen zur Realisierung des Anwendungsprototyps

Ein Exemplar der Klasse StringListWP repräsentiert einen Workprozeßverwalter für einen sehr einfachen Workprozeß, der nur einen einzigen Dienstleister enthält. Die angebotene Dienstleistung besteht aus dem lesenden Zugriff auf Intervalle aus einer Liste von Strings. Exemplare der Klasse StringListForwarder sind Auftragsabwickler, die diese Dienstleistung ausführen. Sie sind zuständig für die Initialisierung der String-Liste mit den alphabetisch sortierten Smalltalk-Globalsymbolnamen und besitzen die Fähigkeit, Intervalle dieser String-Liste auf Anfrage in zusammengesetzte Busnachrichten zu verpacken und an ihre Auftraggeber weiterzuleiten. Exemplare der Klasse StringListForwarderInstructor sind solche Auftraggeber. Sie fordern auf Initiative eines Kundenakteurs jeweils fünf Elemente der beim Auftragsabwickler bereitgestellten String-Liste an. Nach Erhalt übergeben sie diese über ein im gemeinsamen Zugriff liegendes Objekt einem Exemplar der Klasse StringListGUI, das einen solchen Kundenakteur darstellt. Wie bereits erwähnt, sind die StringListGUI-Exemplare diejenigen Objekte, mit denen der Anwendungsbenutzer über die in Bild 3.47 gezeigte grafische Schnittstelle kommunizieren kann.

3.6.3 Dienstspezifische Kommunikationsprotokolle und davon abgeleitete Methoden

Bei Entwurf eines konkreten Anwendungssystems muß für jede Dienstleistung ein dienstspezifisches Kommunikationsprotokoll zwischen Auftraggebern und Auftragsabwicklern und darüberhinaus ein zugehöriges Kommunikationsprotokoll zwischen Kundenakteuren und Auftraggebern spezifiziert werden. Aus dem Auftraggeber-Auftragsabwickler-Protokoll läßt sich ableiten, welche über den Softwarebus aufrufbaren Methoden in der jeweiligen Auftraggeber- und Auftragsabwcklerklasse definiert werden müssen. Das Kunde-Auftraggeber-Protokoll dient zur Festlegung des Katalog der vom Kunden beim Auftraggeber aufrufbaren Kommandos. Bild VI am Ende dieser Schrift zeigt bezogen auf die einzige im Anwendungsprototyp vorhandene Dienstleistung beide Protokolle in Form eines einzigen Petrinetzes. Dabei ist für jeden der drei genannten Akteure ein Zuständigkeitsbereich abgegrenzt. Während ein Markenfluß aus dem Zuständigkeitsbereich des Kundenakteurs zum Auftraggeber einen gewöhnlichen, lokalen Methodenaufruf darstellt, führen Markentransporte zwischen Auftraggeber und Auftragsabwickler zu asynchronen, entfernten Methodenaufrufen über den Softwarebus. In beiden Fällen sind die zugehörigen Methodenbezeichner an die die Zuständigkeitsbereichsgrenzen schneidenden Kanten geschrieben. Um zu verdeutlichen, daß zwischen Auftraggeber und Auftragsabwickler das Softwarebussystem liegt, ist die Zuständigkeitsgrenze in Form einer zweifachen Strichlinie dargestellt. Die Synchronisation des Auftraggebers mit dem Auftragsabwickler unter Nutzung des beim Auftraggeber vorhandenen Semaphorverwalters ist in Bild VI nicht explizit gezeigt.

Kundenkommandos

Man erkennt, daß der Auftraggeber dem Kundenakteur drei Kommandos zur Verfügung stellt:

- **init**
- **getElementsFrom:** <*lowerBound*> **to:** <*upperBound*>
- **closeConnection**

Nachdem der Kundenakteur innerhalb seiner Startaktionen einen Auftraggeber erzeugt hat, beauftragt er ihn mit dem Kommando ‘init’, eine Verbindung zu einem Auftragsabwickler aufzubauen und den ersten Dialogschritt zu vollziehen. Anschließend kann er beim Auftraggeber die Anzahl der Elemente der auf Auftragsabwicklerseite generierten String-Liste erfragen. Mit dem zweiten Kommando ‘getElementsFrom:To:’ veranlaßt der Kundenakteur den Auftraggeber nun zyklisch zur Besorgung des jeweils anzuzeigenden Listenintervalls. In den beiden Argumenten gibt er dabei die Intervallgrenzen des gewünschten Listenausschnitts an. Der Auftraggeber besorgt daraufhin diesen Listenausschnitt und speichert ihn für den Kundenakteur zugänglich ab, sodaß dieser ihn auf dem Bildschirm anzeigen kann. Der Aufruf des Kommandos ‘getElementsFrom:to:’ wiederholt sich jedesmal, wenn der GUI-Benutzer den ‘Vor’- oder den ‘Zurück’-Taster betätigt. Ein Mausklick auf den ‘Beenden’-Taster zieht den Aufruf des dritten Kundenkommandos ‘closeConnection’ nach sich. Mit ‘closeConnection’ veranlaßt der Kundenakteur die ordnungsgemäße Beendigung der Auftraggeber-Auftragsabwickler-Kommunikation.

Teilaufträge

Nach Aufruf des Kundenkommandos ‘init’ beim Auftraggeber beauftragt dieser zunächst den ihm bei Exemplarerzeugung zugeordneten Dienstvermittler zur Vermittlung eines Auftragsabwicklers. Anschließend kennt er das Inbox-Subject dieses Auftragsabwicklers und kann drei verschiedene Teilauftragsbearbeitungs-Methoden bei ihm aufrufen:

- **init:** <*argumentMessage*>
- **elementsFromTo:** <*argumentMessage*>
- **close:** <*argumentMessage*>

Alle drei Methoden sehen ein Argument vor, in dem der Dispatcher bei Umsetzung des Methodenaufrufs die Argument-Nachricht in Smalltalk-Repräsentation übergibt. Im Falle von ‘init:’ enthält diese Argument-Nachricht das Inbox-Subject des Auftraggebers, welches dieser zuvor abonniert hat. Bei Ausführung der Methode ‘init:’ speichert der Auftragsabwickler dieses Inbox-Subject in seinem Attribut ‘clientInbox’ ab, generiert die geforderte String-Liste und meldet deren Länge zurück. Im Falle der Teilauftragsbearbeitungsmethode ‘elementsFromTo:’ hat die Argument-Nachricht folgende Form:

(from: <*lowerBound*>, to: <*upperBound*>)

Es handelt sich also um eine zusammengesetzte Nachricht aus zwei Nachrichtenfeldern. Das erste Feld trägt den Namen ‘from’ und enthält die Untergrenze des vom Auftraggeber gewünschten Intervalls in Form einer Integer-Nachricht. Das zweite Feld trägt den Namen ‘to’ und enthält die Intervall-Obergrenze in Form einer Integer-Nachricht. Bei Ausführung der Methode ‘elementsFromTo:’ verpackt der Auftragsabwickler das angeforderte Listenintervall ebenfalls in einer zusammengesetzten Busnachricht und schickt es als Rückmeldung an den Auftraggeber zurück. Im Falle der dritten über den Bus aufrufbaren StringListForwarder-Methode ‘close:’ bleibt die Argument-Nachricht leer, da keine Argumentübergabe erforderlich ist. Der Aufruf von ‘close:’ bewirkt beim Auftragsabwickler die Beendigung der Dienstausführung.

Teilauftragsrückmeldungen

Bei den ersten beiden Teilauftragsbearbeitungsmethoden ‘init:’ und ‘elementsFromTo:’ sind, wie gesehen, Rückmeldungen vorgesehen. Diese Rückmeldungen geschehen ebenfalls in Form von asynchronen Methodenaufrufen über den Softwarebus, in diesem Fall durch den Auftragsabwickler beim Auftraggeber. Daher stellt auch der Auftraggeber zwei über den Softwarebus aufrufbare Methoden zur Verfügung:

- **initReply:** <*argumentMessage*>
- **elementsFromToReply:** <*argumentMessage*>

Im Falle von ‘initReply:’ enthält die übergebene Argumentnachricht die Elementanzahl der vom Auftragsabwickler generierten String-Liste in Form einer Integer-Nachricht. Bei ‘elementsFromToReply:’ wird als Argumentnachricht eine zusammengesetzte Nachricht übergeben, die den vom Auftraggeber angeforderten Listenausschnitt enthält. Dabei trägt jedes Feld dieser Nachricht genau ein Element des Listenausschnitts. Als Feldname wird jeweils die Positionszahl des Elements in der Stringliste (der Listenindex) angegeben. Das in Bild 3.47 angezeigte Listenintervall wurde beispielsweise in einer zusammengesetzten Busnachricht folgenden Aufbau zum Auftraggeber geschickt:

(896: <‘RVBusMonitor’>, 897: <‘RVCallable’>, 898: <‘RVComposedMessage’>,
899: <‘RVErrors’>, 900: <‘RVEventLoopManager’>)

Sowohl in der Methode ‘initReply:’ als auch in der Methode ‘elementsFromToReply:’ wird dafür gesorgt, daß der Inhalt der übergebenen Argumentnachricht „ausgepackt“ und dem Kundenakteur zugänglich gemacht wird.

3.6.4 Weitere Implementierungsaspekte

In diesem Abschnitt werden die vier eingeführten Klassen StringListWP, StringListForwarder, StringListForwarderInstructor und StringListGUI genauer behandelt und die wichtigsten Aspekte ihrer Implementierung vorgestellt.

StringListWP

Das einzige Merkmal der Klasse StringListWP ist die Methode ‘createServices’. Es handelt sich dabei um die Dienstanbieter-Erzeugungsprozedur, die in jeder Unterklasse von WorkprocessManager neu definiert werden muß. Da im vorliegenden Fall nur ein Dienstanbieter vorgesehen ist, enthält sie lediglich einen einzigen Aufruf der geerbten Methode ‘newServiceUnderSubject:processorClass:’:

```
self
    newServiceUnderSubject: ‘WP.StringListForwarder’
    processorClass: StringListForwarder.
```

Das erste Argument legt den dienstspezifischen Teil von Ausschreibungs- und Ausschreibungsrücknahme-Subject fest, das zweite Argument gibt das zugehörige Auftragsabwickler-Klassenobjekt an.

StringListForwarder

Ganz allgemein müssen in einer Auftragsabwicklerklasse immer zwei Dinge festgelegt werden. Dies ist zum einen die Struktur des Auftragskontextes und zum anderen das Verhalten der Teilauftragsbearbeiter (vgl. Bild 3.35 auf Seite 81). Die Festlegung der Auftragskontext-Struktur geschieht durch Einführung von Attributen und eventuellen Initialisierungsmethoden bezüglich dieser Attribute. Die Verhaltensbeschreibung der Teilauftragsbearbeiter geschieht durch die weiteren Methoden. Jedem Teilauftragsbearbeiter ist dabei genau eine über den Softwarebus aufrufbare Methode zugeordnet.

Bei den im Anwendungsprototyp vorkommenden Auftragsabwicklern stellt die im ersten Dialogschritt generierte String-Liste der alphabetisch sortierten Smalltalk-Globalsymbolnamen den Inhalt des Auftragskontextes dar. Daher wird in der Klasse StringListForwarder das Attribut ‘stringList’ zur Aufnahme dieser Liste zur Verfügung gestellt. Weitere Attribute sind im vorliegenden Fall nicht erforderlich. Die in der Klasse StringListForwarder eingeführten Methoden sind in Bild 3.49 zusammen mit ihren Aufrufbeziehungen dargestellt. Zusätzlich kommt die von ServiceProcessor geerbte Methode ‘terminate’ vor. Diese Methoden beschreiben alle zusammen das Verhalten von insgesamt drei Teilauftragsbearbeitern, die über die drei bereits genannten Methoden ‘init:’, ‘elementsFromTo:’ und ‘close:’ über den Softwarebus angestoßen werden können.

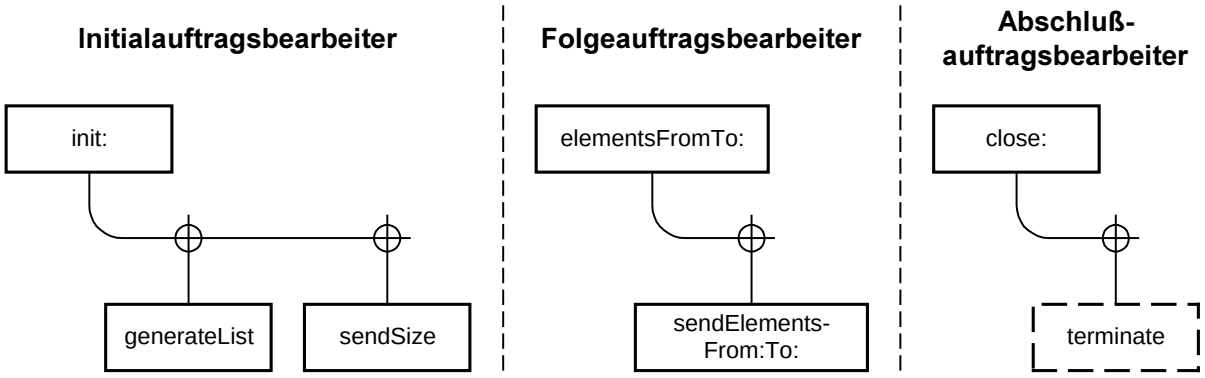


Bild 3.49 Methoden der Klasse StringListForwarder

Der durch ‘init:’ aufrufbare Initialauftragsbearbeiter legt das vom Auftraggeber erhaltene Inbox-Subject im geerbten Attribut ‘clientInbox’ ab, generiert die geforderte String-Liste (‘generateList’) und sendet die Anzahl der Elemente dieser Liste an den Auftraggeber zurück (‘sendSize’). Der Folgeauftragsbearbeiter schickt auf Anfrage Intervalle aus der generierten String-Liste an den Auftraggeber zurück. Angestoßen werden kann er durch Aufruf der Methode ‘elementsFromTo:’, in der zunächst aus der Argumentnachricht die Intervallgrenzen des gewünschten Listenintervalls extrahiert werden. In der von dort aufgerufenen Methode ‘sendElementsFrom:To:’ wird dieses Listenintervall anschließend in eine zusammengesetzte Busnachricht kopiert und als Rückmeldung an den Auftragsabwickler gesendet. Der über die Methode ‘close:’ erreichbare Abschlußauftragsbearbeiter hat nichts weiter zu tun, als die Auftragsabwicklung ordnungsgemäß zu beenden. Dazu ruft er die geerbte Methode ‘terminate’ auf.

Sobald die beiden Klassen StringListWP und StringListForwarder zur Verfügung stehen, lassen sich bereits Workprozesse des im Anwendungsprototyp vorkommenden Typs erzeugen und aktivieren:

```
WP := StringListWP new.
WP start.
```

Die erste der gezeigten Smalltalk-Anweisungen veranlaßt die Erzeugung eines Workprozeßverwalters. Um ihn vor der Garbage-Collection zu bewahren, wird er nach seiner Erzeugung an das Globalsymbol ‘WP’ gebunden. Die zweite Anweisung, der Aufruf der Workprocess-Manager-Methode ‘start’, veranlaßt diesen Workprozeßverwalter zur Aktivierung des von ihm verwalteten Workprozesses.

StringListForwarderInstructor

Auftraggeberklassen enthalten in ihren Methoden die Verhaltensbeschreibung der Dialogschrittakteure und des Befehlsumsetzers (vgl. Bild 3.36). Ferner müssen Attribute inklusive entsprechender Zugriffsmethoden vorgesehen werden, um den Datenaustausch zwischen Auftraggebern und Kundenakteuren zu ermöglichen (Bild 3.36: shared Memory).

Auftraggeber der Klasse StringListForwarderInstructor des Anwendungsprotyps besitzen folgende Attribute:

listSize Dient zur Aufnahme der vom Auftragsabwickler nach dem Initialauftrag zurückgemeldeten Größenangabe bezüglich der String-Liste. Die gleichnamige Zugriffsmethode ‘listSize’ erlaubt dem Kundenakteur lesenden Zugriff auf dieses Attribut.

selectionInPartOfList
Dient zur Aufnahme eines Verweises auf ein Objekt im Zuständigkeitsbereich des Kundenakteurs, in das der Auftraggeber das in der Rückmeldung zu einem Folgeauftrag enthaltene Listenfragment eintragen kann. Dieses Attribut läßt sich daher über die Zugriffsmethode ‘selectionInPartOfList:’ durch den Kundenakteur belegen.

Das Verhalten des Befehlsumsetzers eines Auftraggebers der Klasse StringListForwarderInstructor wird in den drei bereits erwähnten Kundenkommando-Methoden ‘init’, ‘getElementsFrom:to:’ und ‘closeConnection’ beschrieben. Da mit den ersten beiden Kundenkommandos ‘init’ und ‘getElementsFrom:to:’ Dialogschritte vom Typ Auftrag-Rückmeldung angestoßen werden, muß vom Befehlsumsetzer jeweils für die Synchronisation mit dem Auftragsabwickler gesorgt werden. Die zugehörigen Methoden enthalten daher ‘wait’-Anweisungen an den im geerbten Attribut ‘replySemaphore’ zur Verfügung gestellten Semaphorverwalter. In weiteren fünf Methoden wird das Verhalten von insgesamt drei Dialogschrittakteuren beschrieben. Gemäß Bild 3.37 wird zwischen Sendern und Empfängern unterschieden:

Dialogschrittakteur 1: Initialauftragserteilung

- Sender: **sendInitialRequest**
- Empfänger: **initReply:** <*argumentMessage*>

Dialogschrittakteur 2: Folgeauftragserteilung

- Sender: **orderElementsFrom:** *<lowerBound>* **to:** *<upperBound>*
- Empfänger: **elementsFromToReply:** *<argumentMessage>*

Dialogschrittakteur 3: Abschlußmeldung

- Sender: **sendCloseNotification**
- Empfänger: nicht vorhanden

Die beiden Empfängermethoden ‘initReply:’ und ‘elementsFromToReply:’ enthalten als letzte Anweisung einen ‘signal’-Auftrag an den erwähnten Semaphorverwalter, um nach Rückmeldungseingang den Befehlsumsetzer aus seinem Wartezustand zu befreien.

Die Klasse StringListForwarderInstructor sieht des weiteren eine Klassenmethode für das zuständige Klassenobjekt vor. Sie trägt den Namen ‘initialize’ und dient dazu, das Klassenobjektattribut ‘serviceSubject’ mit dem dienstspezifischen Teil von Ausschreibungs- und Ausschreibungsrücknahme-Subject (‘WP.StringListForwarder’) zu belegen. Dieser wird bei Ausführung der geerbten Methode ‘orderService’ an den Dienstvermittler übergeben (vgl. Seite 95).

StringListGUI

Die Klasse StringListGUI enthält die Merkmale der im Anwendungsprototyp vorkommenden Kundenakteure. Folgende Attribute werden eingeführt:

sessionManager
: Dient zur Aufnahme eines Bussitzungsverwalters für den GUI-Akteur.

instructor
: Verweist auf den für den Kundenakteur zuständigen Auftraggeber.

listSize
: Dient zur Speicherung der Länge der auf Auftragsabwickler-Seite bereitgestellten String-Liste.

cursor
: Laufindex bezüglich der String-Liste; wird beim Vorwärtsblättern erhöht und beim Rückwärtsblättern erniedrigt.

selectionInPartOfList
: Verweist auf das für Auftraggeber und Kundenakteur gleichermaßen zugängliche Objekt zur Aufnahme des anzuzeigenden Listenfragments. Es handelt sich um ein Exemplar der Systemklasse SelectionInList. Exemplare dieser Klasse dienen speziell zur Aufnahme von Listen, die grafisch angezeigt werden sollen.

Des weiteren werden in der Klasse StringListGUI folgende Methoden definiert:

start
: Aktiviert den GUI-Akteur. Es handelt sich um die erste Methode, die nach Erzeugung eines StringListGUI-Exemplars aufgerufen wird. In ihr werden die oben gezeigten Attribute initialisiert, die ersten Dialogschritte zwischen Auftraggeber und Auftragsabwickler angestoßen, und das die grafische Benutzerschnittstelle enthaltende Bildschirmfenster geöffnet. Den genauen Ablauf zeigt Bild 3.50.

getNextInterval

Wird bei Betätigung der ‘Vor’-Taste der grafischen Benutzerschnittstelle aufgerufen. In dieser Methode werden als erstes die Intervallgrenzen bezüglich des nächsten anzuzeigenden Intervalls bestimmt. Daraufhin wird der Auftraggeber via Kundenkommando ‘getElementsFrom:to:’ zur Besorgung dieses Intervalls aufgefordert und der Laufindex (‘cursor’) nachgeführt.

getPreviousInterval

Wird bei Betätigung der ‘Zurück’-Taste ausgeführt. Der angestoßene Ablauf ist analog zu dem bei ‘getNextInterval’.

terminate Wird bei Betätigung der ‘Beenden’-Taste aufgerufen. In dieser Methode wird zunächst das Kundenkommando ‘closeConnection’ zur Beendigung der Dienstinanspruchnahme abgesetzt, darauf das die Benutzerschnittstelle enthaltende Bildschirmfenster geschlossen und schließlich der Bussitzungsverwalter zur Sitzungsbeendigung aufgefordert.

Die beiden Methoden ‘getNextInterval’ und ‘getPreviousInterval’ enthalten Kontrollstrukturen, die verhindern, daß der Benutzer über Anfang oder Ende der Liste hinausblättern kann. Das Auffrischen der Anzeige nach Besorgen eines neuen Listenausschnitts geschieht automatisch. Es gehört zu den Merkmalen eines SelectionInList-Objekts, bei Änderung der in ihm enthaltenen Liste die entsprechenden View-Objekte zu benachrichtigen. Die bisher nicht genannte Methode ‘selectionInPartOfList’ zum Zugriff auf dieses Objekt sowie die Klassenmethode ‘windowSpec’ sind beim Entwurf der grafischen Benutzerschnittstelle mit Hilfe der von VISUALWORKS zur Verfügung gestellten Werkzeuge automatisch generiert worden. In der Klassenmethode ‘windowSpec’ wird insbesondere für die Assoziation der drei Methoden ‘getNextInterval’, ‘getPreviousInterval’ und ‘terminate’ mit den drei Tasten der grafischen Benutzerschnittstelle (Bild 3.47) gesorgt.

Sobald die Klassen StringListGUI und StringListForwarderInstructor zur Verfügung stehen, lassen sich GUI-Akteure erzeugen und aktivieren durch folgende Anweisungen:

```
GUI := StringListGUI new.
GUI start.
```

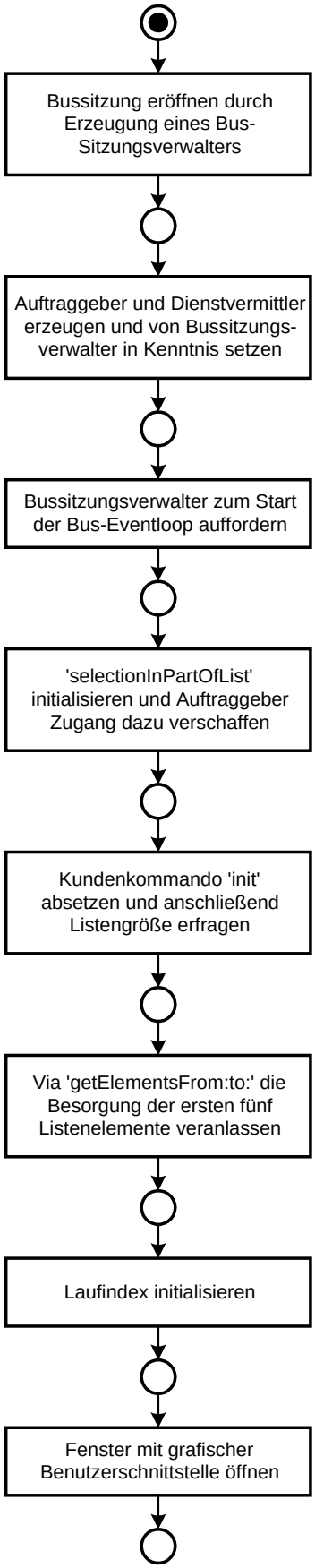


Bild 3.50 StringListGUI-Methode ‘start’

4 Zusammenfassung und Ausblick

Im ersten Teil dieser Arbeit wurden die Grundkonzepte von Smalltalk vorgestellt. Der Aufbau eines Smalltalk-Laufzeitsystems aus virtueller Maschine und Objektspeicher ermöglicht es, plattformunabhängige Anwendungssysteme zu realisieren. Die Systementwicklung geschieht dabei durch inkrementelle Erweiterung des Objektspeicherinhalts um gewöhnliche Objekte, Klassenobjekte und Metaklassenobjekte. Ferner hat man die Möglichkeit, das Befehlsrepertoire des Smalltalk-Abwicklers durch weitere Basismethoden zu erweitern. Merkmale wie Prozessormultiplex, Ausnahmebehandlung und grafische Benutzerschnittstellen machen Smalltalk für weite Anwendungsbereiche einsetzbar. Zwar trüben die teilweise recht unnatürlichen Modellvorstellungen, die bei Interpretation von Kontrollstrukturen und zahlenarithmetischen Ausdrücken vorausgesetzt werden, das Bild etwas, insgesamt gilt aber die Aussage des Abschnitts 2.3, in der der Einsatz von Smalltalk im Software-Technologie-Labor empfohlen wird.

Der zweite Teil der vorliegenden Arbeit belegt diese Aussage. Dort wurde eine objektorientierte Architektur für 3-Ebenen-Client-Server-Anwendungen vorgestellt und ihre Implementierung in Smalltalk vorgeführt. Dabei wurden vier Schichten voneinander abgegrenzt und nacheinander dokumentiert. Schicht 1 enthält den Rendezvous-Softwarebus, der als Kommunikationskanal zwischen den Komponenten der Applikationsebene und Präsentationsebene dient. Die vom Hersteller dieses Bussystems zur Verfügung gestellte Sprachnittstelle sieht C als Programmiersprache für Busanwendungen vor. Nach Einführung des RVInterface-Monopolakteurs läßt sich diese Schnittstelle auch aus Smalltalk heraus nutzen. Das auf diese Weise für Smalltalk verfügbar gemachte C-API erwies sich jedoch als ungeeignet für die Benutzung innerhalb objektorientierter Anwendungen. In Schicht 2 wurde daher eine objektorientierte Kapselung in Form des Smalltalk-Rendezvous-API durchgeführt. Dabei wurden eine Reihe von Klassen eingeführt, deren Exemplare Veröffentlichungen, Nachrichten und Abonnements in Smalltalk repräsentieren bzw. zur Verwaltung von Bussitzungen dienen. Schicht 3 bietet darauf aufbauend ein Klassengerüst an, mit dem die Entwicklung von 3-Ebenen-Client-Server-Systemen in Smalltalk stark vereinfacht wird. Diesem Klassengerüst liegt ein Systemmodell zugrunde, daß eine Reihe von Akteurstypen einführt, denen unterschiedliche Aufgabenbereiche zugeordnet werden. Die zugehörigen Akteursbezeichner lauten Workprozeß, Dienstleister, Dienstanbieter, Auftragsabwickler, Auftraggeber, Dienstvermittler und Kunde. Schicht 4 beinhaltet schließlich die anwendungsspezifischen Komponenten eines konkreten Anwendungssystems. Die Entwicklung eines konkreten Anwendungssystems war jedoch nicht Gegenstand dieser Arbeit. Daher stellte Abschnitt 3.6 auch nur einen einfachen Anwendungs-

prototyp ohne Datenbankanschluß vor, der lediglich zur Veranschaulichung der Nutzungsmöglichkeiten des Multiclient-Multiserver-Klassengerüstes dient.

Die im Rahmen der vorliegenden Arbeit entworfene Anwendungsarchitektur bietet Raum für Erweiterungen und Verfeinerungen. Beispielsweise wäre es wünschenswert, daß ein einziger Auftraggeber gleichzeitig mehrere Auftragsabwickler zur Ausführung unterschiedlicher Dienstleistungen in Anspruch nehmen kann. Bei Verteilung der gleichzeitig benutzten Dienstleister auf verschiedene Workprozesse ließe sich so der Nebenläufigkeitsgrad innerhalb eines Anwendungssystems deutlich erhöhen, da ein Auftraggeber bei Ausführung eines Kundenkommandos dann mehrere Workprozesse zur selben Zeit beschäftigen könnte. Eine Klasse mit dem Namen „MultiProcessorInstructor“, deren Exemplare solche Auftraggeber in Smalltalk repräsentieren, ist bereits in Planung. Des weiteren ist es denkbar, innerhalb der Workprozesse Akteure einzuführen, die gegenüber den GUIs der Präsentationsebene in einer Master-Rolle auftreten können. Dann wäre es möglich, Ereignismeldungen aus der Applikationsebene in die Präsentationsebene zu senden, um GUI-Akteure beispielsweise davon in Kenntnis zu setzen, daß bestimmte zur Anzeige gebrachte Daten nicht mehr aktuell sind. Bisher ist dies nicht vorgesehen. Auftragsabwickler und Dienstanbieter sind gegenüber den Auftraggebern und Dienstvermittlern der Präsentationsebene immer in einer Slave-Rolle. Als Master treten Auftragsabwickler nur dann auf, wenn sie ihrerseits über Auftraggeber weitere Dienstleistungen der Applikationsebene in Anspruch nehmen.

Man erkennt also, daß weitere Arbeiten zur Vervollkommnung der vorgestellten Architektur lohnenswert sind. Die im Rahmen der vorliegenden Arbeit bereitgestellten Komponenten bieten aber bereits eine leistungsfähige Experimentierumgebung zur Entwicklung von Client-Server-Systemen in Smalltalk.

Literaturverzeichnis

[Auer 96] Stefan Auer
Teknekron Technology Evaluation
Interner Bericht, SAP-AG, 1996

[Bücker 95] Matthias C. Bücker, Joachim Geidel, Matthias M. Lachmann
Programmieren in Smalltalk mit VisualWorks
Springer, 1995

[Ganzinger 87] Harald Ganzinger, Georg Heeg, Hubert Baumeister, Michael Rüger
Smalltalk-80
Informationstechnik it, 29. Jahrgang, Heft 4/87, S. 241-251

[Goldberg 83] Adele Goldberg, David Robson
Smalltalk-80: The Language and its Implementation
Addison-Wesley, 1983

[Goldberg 84] Adele Goldberg
Smalltalk-80: The Interactive Programming Environment
Addison-Wesley, 1984

[Goldberg 89] Adele Goldberg, David Robson
Smalltalk-80: The Language
Addison-Wesley, 1989

[Gröne 96] Bernhard Gröne
Bereitstellung einer Laborumgebung und Untersuchung objektorientierter Datenbanktechnologien
Diplomarbeit am Lehrstuhl für Digitale Systeme, Universität Kaiserslautern, 1996

[Hoffmann 87] Hans-Jürgen Hoffmann
Smalltalk verstehen und anwenden
Hanser, 1987

[Hoffmann 90] Hans-Jürgen Hoffmann
Smalltalk - Stand & Perspektiven aus der Sicht eines Software-Ingenieurs
Technische Hochschule Darmstadt, Fachgebiet Programmiersprachen und Übersetzer, 1990

[Howard 95] Tim Howard
The Smalltalk Developers Guide to VisualWorks
SIGS Books, 1995

[Krasner 83] Glenn Krasner
Smalltalk-80: Bits of History, Words of Advise
Addison-Wesley, 1983

[Meyer 90] Bertrand Meyer
Objektorientierte Softwareentwicklung
Hanser/Prentice Hall, 1990

[Mittendorfer 89] Josef Mittendorfer
Objektorientierte Programmierung mit C++ und Smalltalk
Addison-Wesley, 1989

[RVProg 95] **Rendezvous Software Bus Programmer's Guide**
Teknekron Software Systems Inc., 1995

[VWDLLCC 94] **VisualWorks DLL and C Connect – User's Guide**
ParcPlace Systems, 1994

[VWUserG 95] **VisualWorks User's Guide**
ParcPlace-Digitalk, 1995

[Wendt 91] Siegfried Wendt
Nichtphysikalische Grundlagen der Informationstechnik
Springer, 1991

[Wiegert 95] Oliver Wiegert
Änderbarkeit durch Objektorientierung
Vieweg, 1995

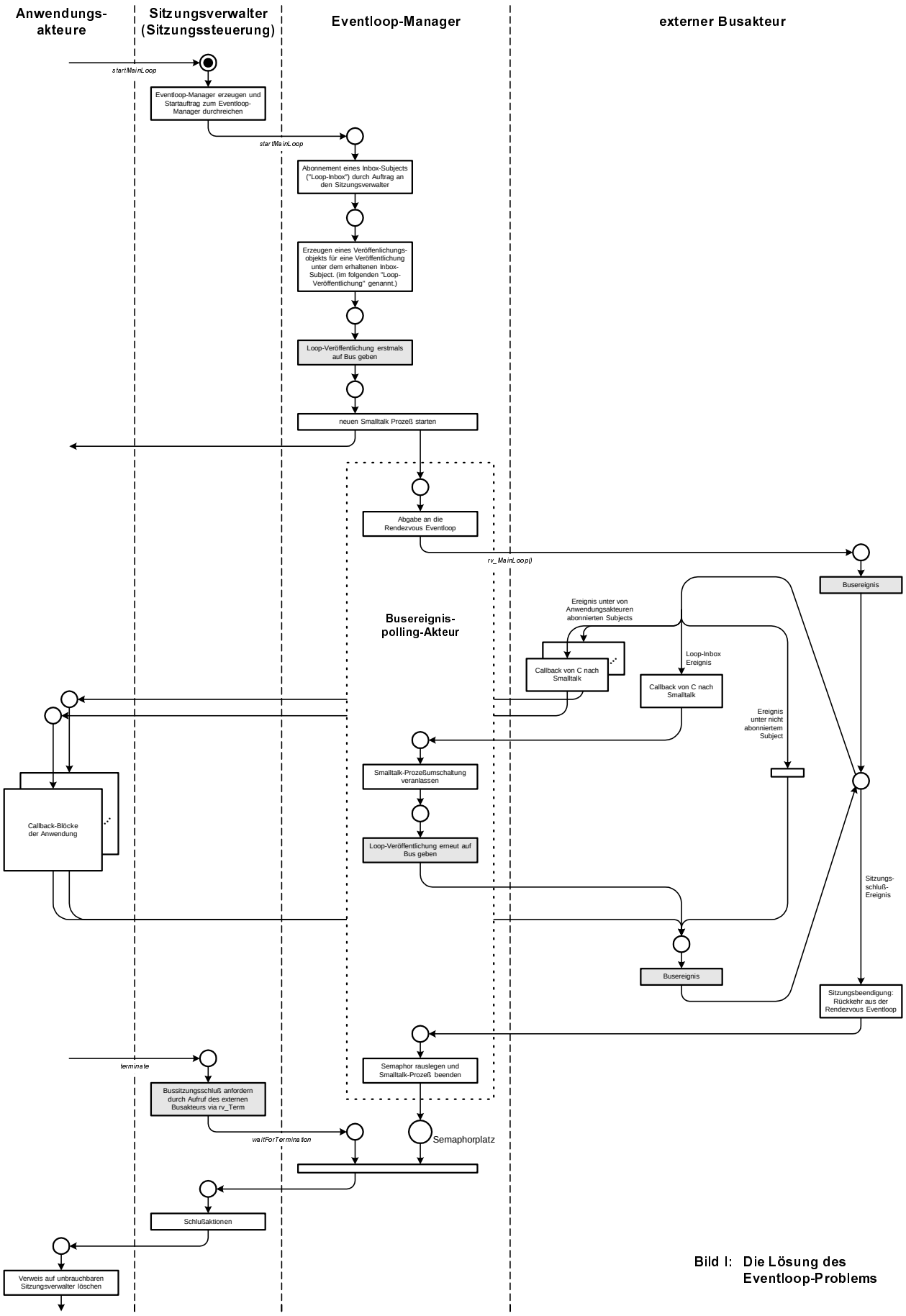


Bild I: Die Lösung des Eventloop-Problems

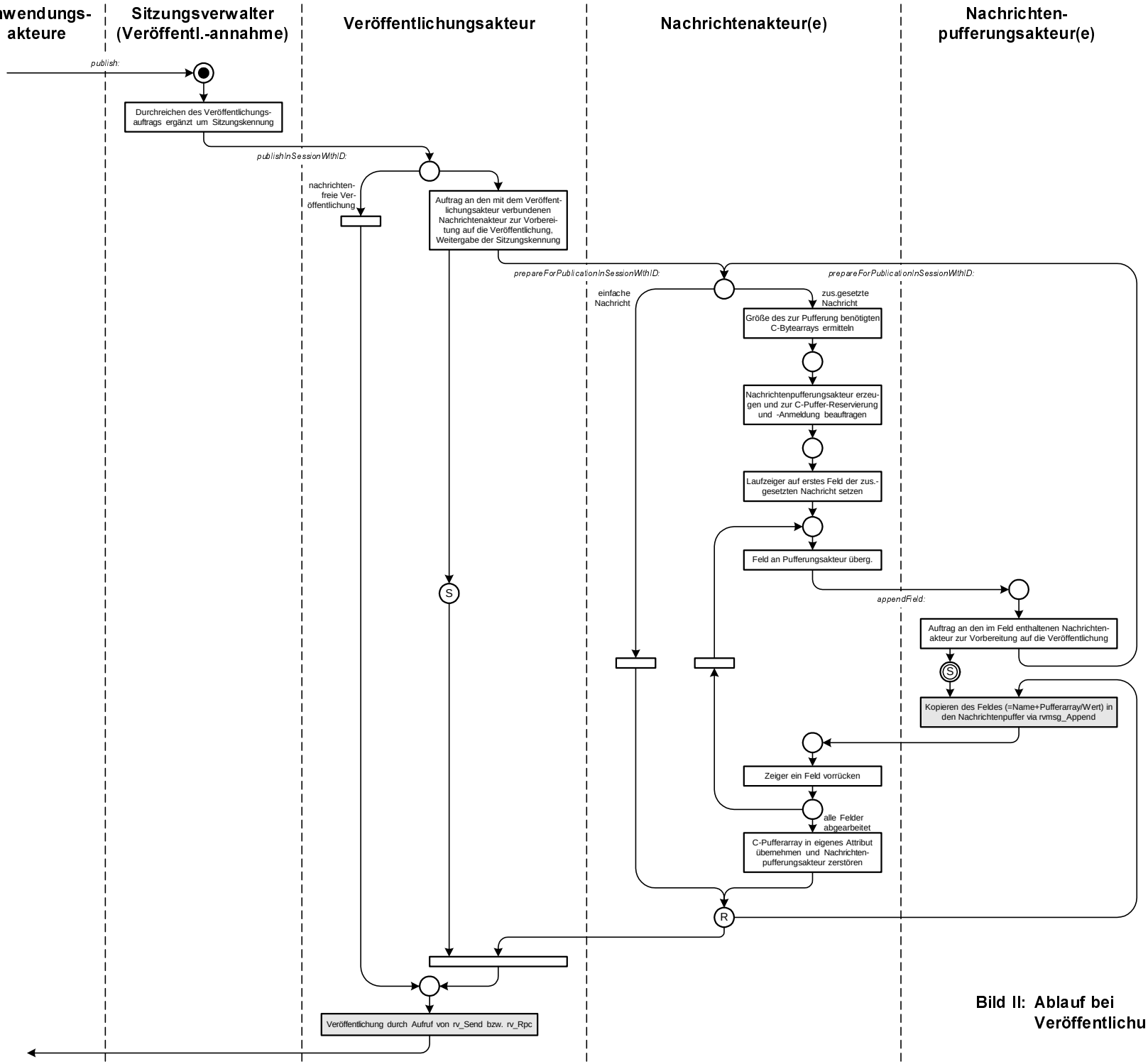


Bild II: Ablauf bei Veröffentlichungen

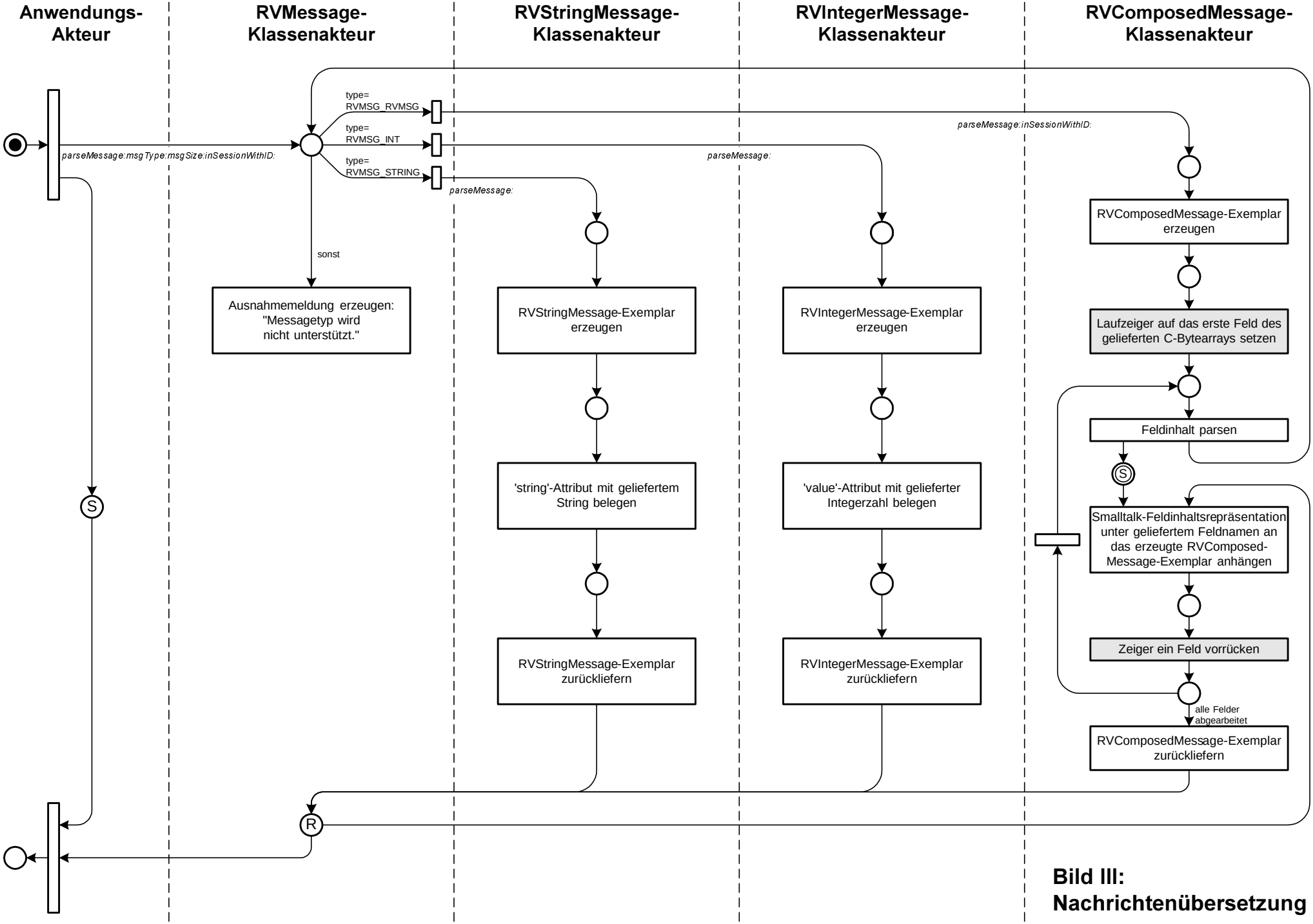


Bild III: Nachrichtenübersetzung

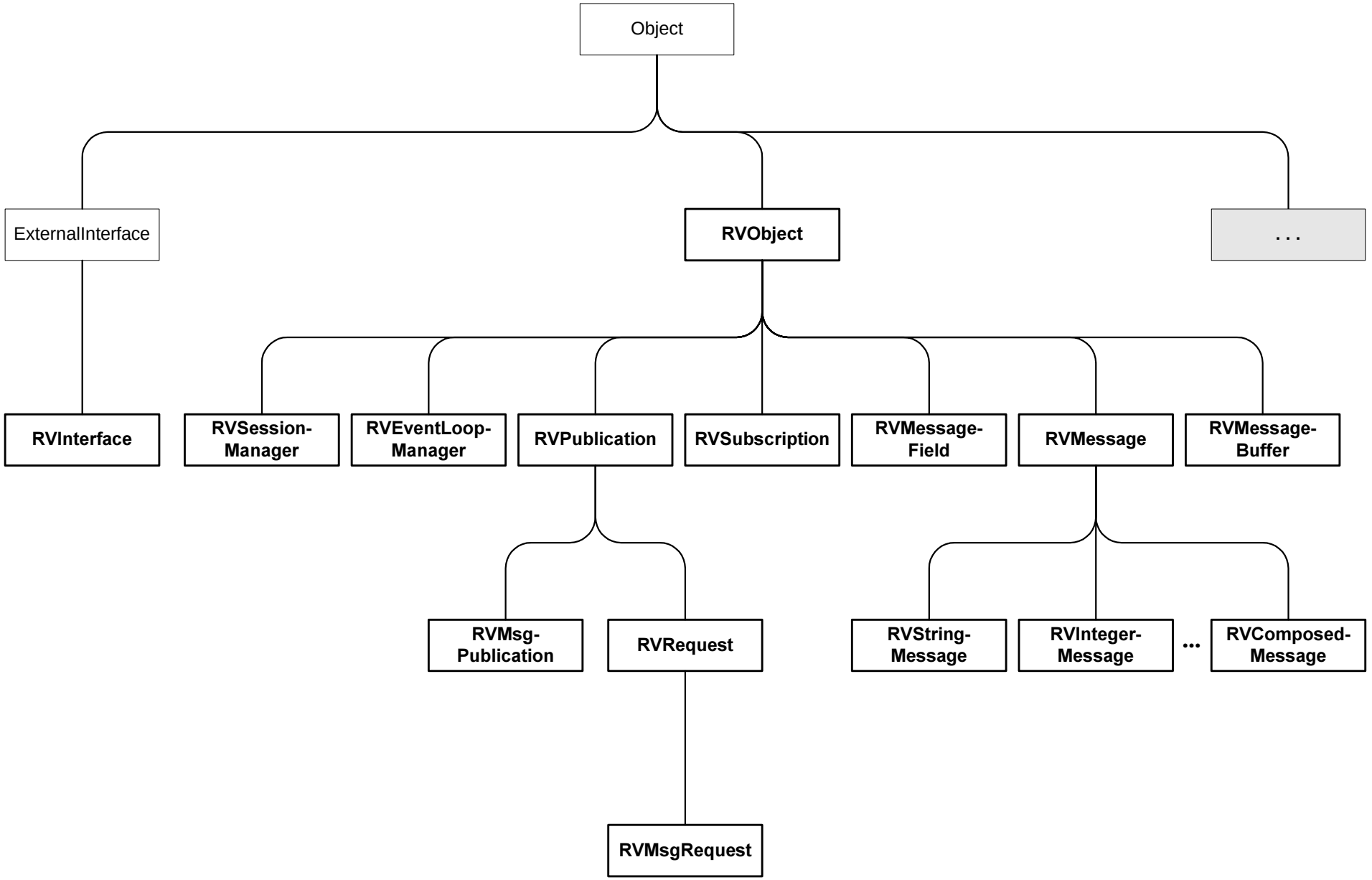


Bild IV: Smalltalk-Rendezvous-API: vollständiger Klassenbaum

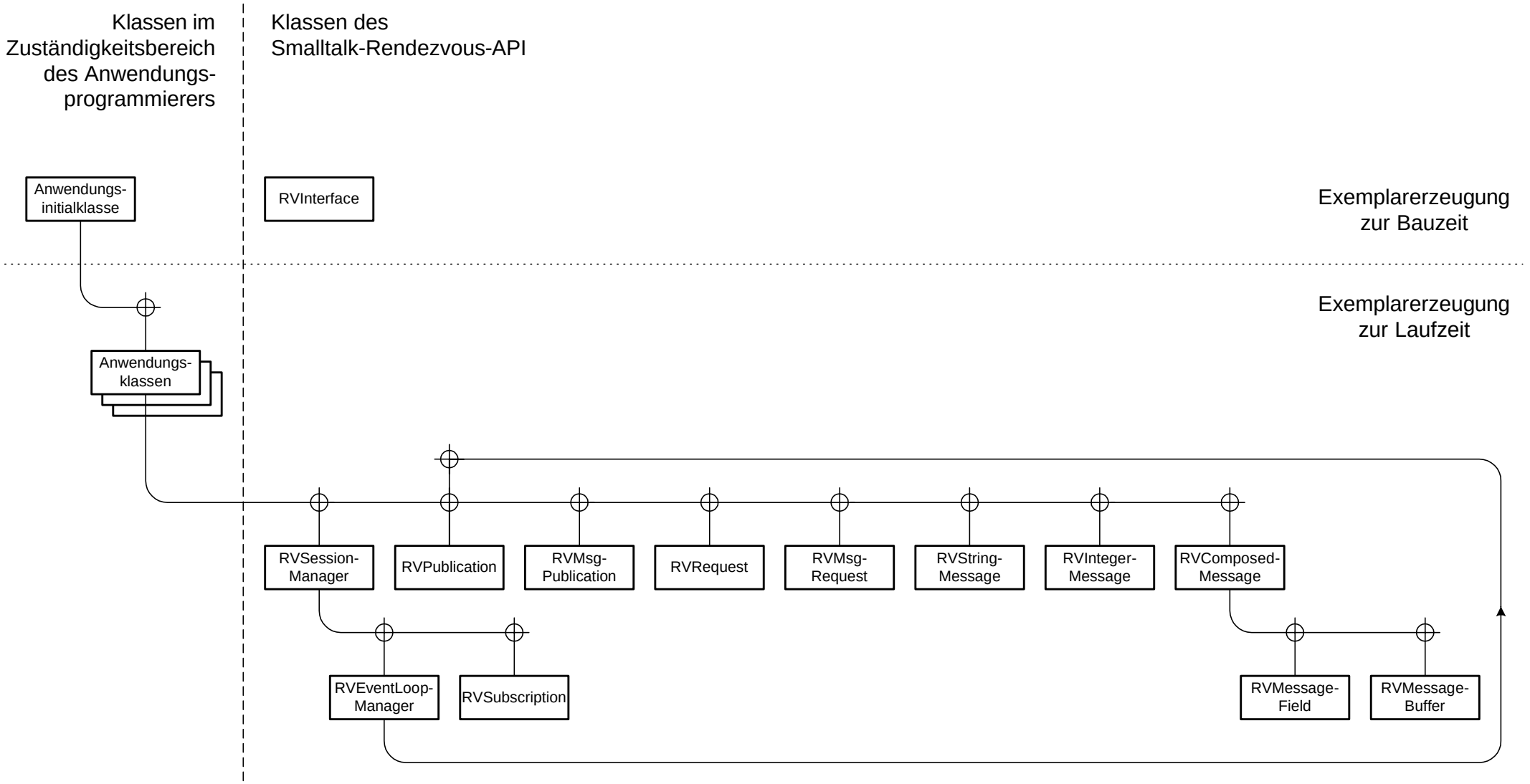


Bild V: Smalltalk-Rendezvous-API: Erzeugungsbeziehungen

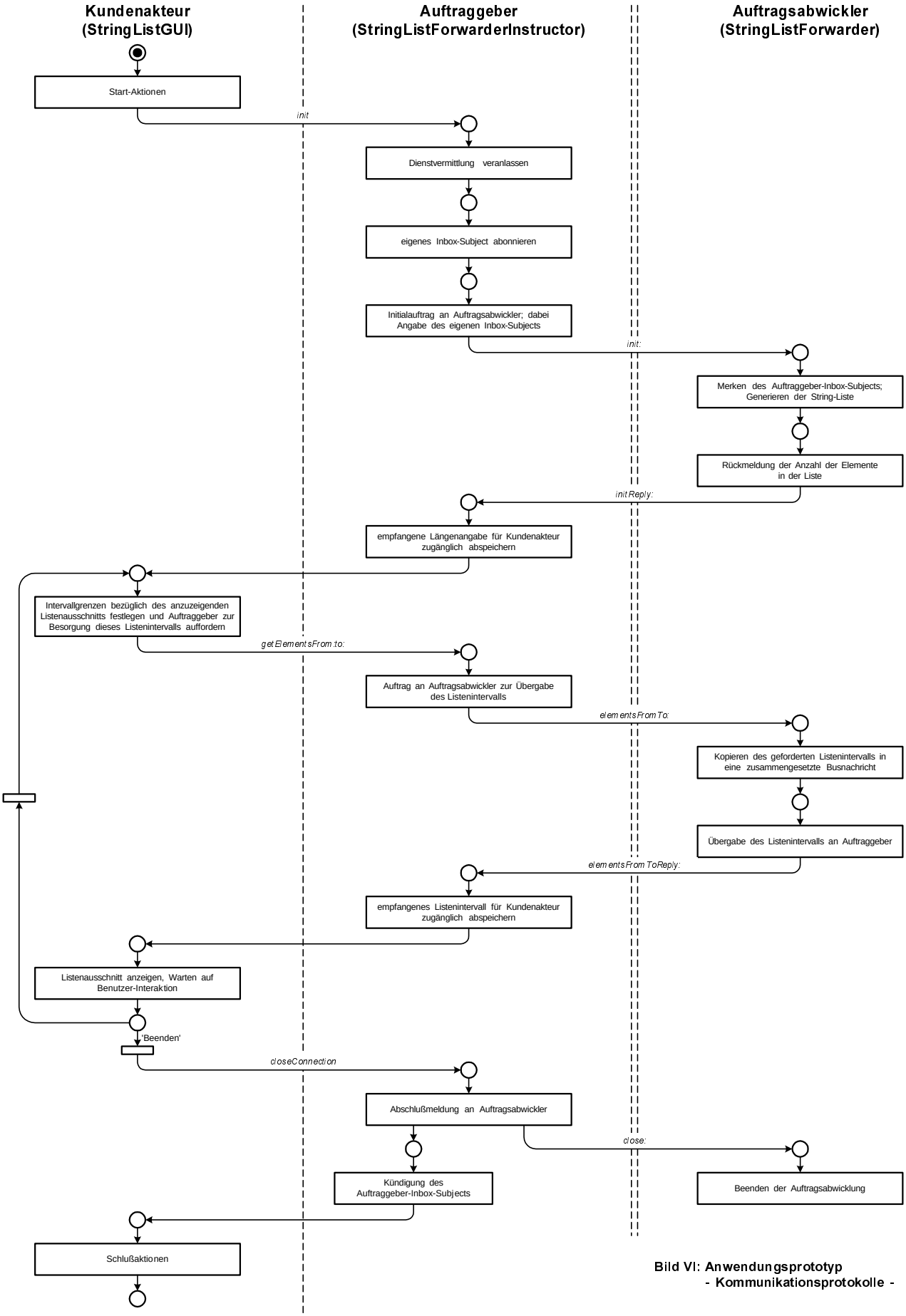


Bild VI: Anwendungsprototyp
- Kommunikationsprotokolle -