Inhaltsverzeichnis

Kapitel 1

Einleitung

Seit mit SIMULA die erste objektorientierte Programmiersprache das Licht der Welt erblickte, hat sich auf diesem Gebiet vieles getan. Die objektorientierte Denkweise ist für die Entwicklung von Software allgemein anerkannt. Inzwischen gibt es für alle Abschnitte des Software-Lebenszyklus objektorientierte Techniken. Der Schritt aus den Forschungslabors heraus ist geschafft, objektorientierte Softwareentwicklung wird industriell mit mehr oder weniger Erfolg praktiziert, mehr noch: die Weiterentwicklung und Forschung kommt nicht nur aus den Universitäten, sondern auch aus der Industrie.

Was macht eine Programmiersprache objektorientiert? In [Boo94, Seite 57] findet sich folgende Definition:

> "Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet sind."

Die Schlüsselkriterien dieser Definition, die auch von anderen Autoren aufgezählt werden, sind:

- Objekte (Daten + Algorithmen) als fundamentale Bausteine
- Klassen
- Vererbung

Sprachen, denen die letzte Eigenschaft fehlt, werden vielfach als *objektbasiert* aber nicht als objektorientiert bezeichnet. Die heute bekanntesten objektorientierten Programmiersprachen wie Java [AG98], C++ [Str97] oder Smalltalk [GR89] erfüllen die genannten Kriterien. Die Flut von Literatur für diese Sprachen ist nahezu unüberschaubar.

Daneben gibt es jedoch Programmiersprachen, die auf andere Art und Weise objektorientiert sind als obige Definition vorschreibt. Im Bereich des Prototyping, also der schnellen Entwicklung von Systemen zur Evaluierung von Anforderungen, wurden klassenlose Sprachen entwickelt, die ein Objekt direkt und nicht auf dem Umweg über eine Klasse beschreiben. Diese Sprachen haben bis heute nicht den Bekanntheitsgrad klassenbasierter Sprachen erreicht und sind zum großen Teil Forschungsprojekte ohne industrielle Anwendung. Zu den bekanntesten unter ihnen zählen SELF [SU95] und NewtonScript [Smi95], letztere hat durch ihren Einsatz im Newton PDA (Personal Digital Assistent) [New] sehr wohl industrielle Bedeutung und wird nicht nur zum Prototyping genutzt.

Klassenlose Sprachen realisieren Vererbung direkt zwischen Objekten. Ein Objekt erbt die Eigenschaften seines Elternobjekts, indem es Nachrichten, die es nicht selbst bearbeiten kann, an dieses weiterleitet. Dieser Vorgang wird *Delegation* genannt. Neue Objekte können als *Abwandlungen* von bereits bestehenden Objekten, sogenannter *Prototypen* direkt erzeugt werden. Ein Kindobjekt kann

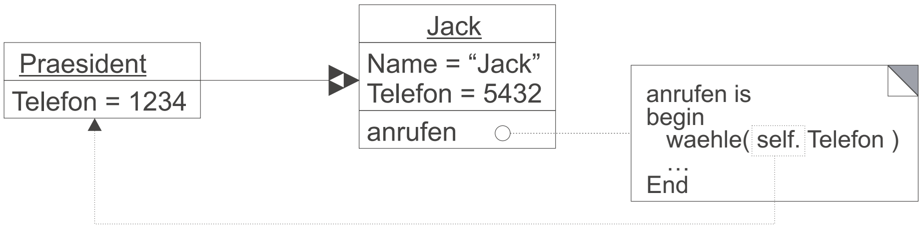

Abbildung 1.1: Beispiel für eine Delegationsbeziehung zwischen zwei Objekten. Das Diagramm benutzt die UML-Notation [Bur97].

sein Elternobjekt zur Laufzeit dynamisch austauschen und so sein eigenes Verhalten ändern. Diese hohe Flexibilität ist für die Entwicklung von Prototypen hilfreich, der Umweg über Klassen wäre in vielen Fällen viel zu langwierig und kompliziert. Klassenlose Sprachen, die Vererbung über Delegationsbeziehungen zwischen Objekten realisieren, werden im folgenden als *delegationsbasiert* [Weg87] bezeichnet.

Objektorientierte Sprachen dienen zur Entwicklung kommerzieller Produkte, bei denen Korrektheit, Verläßlichkeit und Wartbarkeit wichtige (nicht immer erreichte) Kriterien sind. Bei delegationsbasierten Sprachen kommt es mehr auf Flexibilität an. Laufzeitfehler können in Kauf genommen werden, da das System nur der Bestimmung von Anforderungen dient und selbst (hoffentlich) nicht in kritischen Bereichen eingesetzt wird.

1.1 Was ist Delegation

Delegation ist eines der Schlüsselkonzepte dieser Diplomarbeit. Es muß daher in all seinen Einzelheiten und Auswirkungen untersucht werden. Dies wird auch geschehen. Trotzdem soll bereits in der Einleitung eine kurze Erklärung gegeben werden, um Lesern, die bisher nicht mit Delegation zu tun hatten, ein Gefühl für dieses Konzept zu geben.

In Abbildung 1.1 delegiert ein Objekt `Praesident` (*delegierendes Objekt*, *Kindobjekt*) an ein Objekt `Jack` (*delegiertes Objekt*, *Elternobjekt*). Dies soll bedeuten, daß eine Person namens Jack von Beruf Präsident ist, z.B. von einer Organisation oder einem Land. `Praesident.Telefon` ist die Dienstnummer, `Jack.Telefon` die Privatnummer von Jack. Ferner kann man Jack anrufen, wozu die Methode `Jack.anrufen` dient.

Wenn nun `Jack.anrufen` aufgerufen wird, so klingelt das Telefon mit der Nummer 5432, das hätte jeder auch so erwartet. Wird hingegen `Praesident.anrufen` aufgerufen, so bietet das Objekt `Praesident` selbst kein Merkmal `anrufen` an. Da jedoch die Delegationsbeziehung zu Jack besteht, wird dort nach einem passenden Merkmal gesucht, und auch gefunden. Welcher Apparat klingelt nun? Es ist der mit der Nummer 1234, also der Dienstapparat. Jedes Objekt besitzt ein (implizites) Attribut, das auf das Objekt selbst verweist. Dieses Attribut wird als `self`-Referenz bezeichnet. Ruft ein Objekt Merkmale von sich selbst auf, bedeutet dies konzeptionell den Aufruf eines Merkmals des Objekts, auf das `self` verweist. In Abbildung 1.1 wird dies dadurch verdeutlicht, daß die Methode `anrufen` nicht auf `Telefon`, sondern auf `self.Telefon` zugreift. In den meisten Programmiersprachen kann auf die Qualifizierung eines Aufrufs mit `self` verzichtet werden, die explizite Angabe von `self` wird in dieser Arbeit trotzdem oft genutzt um Sachverhalte zu verdeutlichen. Diese `self`-Referenz verweist auch nach der Weiterleitung von Aufrufen über eine Delegationsbeziehung weiterhin auf das delegierende Objekt. Die Suche nach einem passenden Merkmal für den Aufruf `self.Telefon` beginnt also wieder bei `Praesident`. Die Bindung von Merkmalen erfolgt erst beim Aufruf (spätes Binden). Da `Praesident` ein Merkmal `Telefon` hat, wird dieses ausgewählt.

Aus Sicht einer objektorientierten Programmiersprache ist dieses Verhalten vielleicht etwas verwunderlich. Schließlich werden beim Zugriff auf Attribute ohne Angabe eines bestimmten Objekts die

Attribute des Objekts gewählt, das den Zugriff ausführt, also müßte 5432 gewählt werden. Wenn man die Sache jedoch losgelöst von der Programmierung sieht, so ist dieses Verhalten ganz natürlich. Will man den Präsidenten anrufen, so wird die Dienstnummer gewählt, will man Jack privat anrufen, wird eben die Privatnummer gewählt.

Delegation wird in [Weg87] wie folgt definiert:

> "Delegation ist ein Mechanismus, der es Objekten erlaubt, Verantwortlichkeit für die Ausführung von Operationen oder das Lesen von Werten an ein oder mehrere Elternobjekte zu delegieren. Ein Schlüsselmerkmal ist, daß Selbstreferenzierungen im Elternobjekt dynamisch das Kindobjekt bezeichnen ..."

1.2 Zielsetzung und Gliederung der Diplomarbeit

Das Ziel dieser Diplomarbeit ist es, die Vorteile objektorientierter und delegationsbasierter Programmiersprachen, also die Sicherheit und die Flexibilität, in einer Sprache zu vereinigen. Zu diesem Zweck soll die objektorientierte Programmiersprache Eiffel [Mey97] um Konzepte delegationsbasierter Sprachen erweitert werden. Dabei wird es nur um die konzeptionelle Einführung neuer Sprachkonstrukte gehen. Die Implementierung eines Compilers erfolgt im Rahmen der Arbeit nicht. Entsprechend werden eingeführte Konstrukte auch nicht auf ihre Umsetzbarkeit in einem Compiler oder einem Laufzeitsystem untersucht.

Um eine ausreichende Motivation für eine solche Erweiterung zu bekommen, wird in Kapitel 2 erläutert, warum ein Software-System in bestimmten Situationen mit Delegationsbeziehungen besser modelliert werden kann als mit Vererbung. Kapitel 3 nimmt zunächst objektorientierte und delegationsbasierte Systeme getrennt unter die Lupe. Neben dem statischen und dynamischen Verhalten werden dort besonders die physikalischen Strukturen von Interesse sein, die für den Ablauf des jeweiligen Systems aufgebaut werden. Dies ist für das Verständnis der auftretenden Probleme von entscheidender Bedeutung.

In Kapitel 4 und den vier folgenden Kapiteln wird die Programmiersprache Eiffel um Delegation erweitert. Die Erweiterungen werden nacheinander eingeführt und durch kurze Beispiel-Quelltexte und Diagramme motiviert und erläutert. Die Wechselwirkungen zwischen einzelnen Erweiterungen werden dabei besonders herausgestellt. Nach und nach wird die Sprache E+ entstehen.

In Kapitel 9 wird E+ dazu genutzt, einige kleine Beispiele zu implementieren. Mangels eines Compilers für E+ sind sie natürlich nicht übersetzbar. Sie werden jedoch aufzeigen, wo mit einer um Delegation erweiterten objektorientierten Programmiersprache die Umsetzung bestimmter Probleme vereinfacht werden kann und wo dies nicht möglich ist. Diese Beispielanwendungen werden bestimmte Entwurfsmuster [GJHV95] benutzen, die Delegationsbeziehungen nutzen.

Das abschließende Kapitel 10 wird noch einmal über die Arbeit resümieren und das Konzept Delegation aufgrund der damit gemachten Erfahrungen bewerten. Ein Ausblick auf mögliche Fortsetzungen schließt die Arbeit ab.

1.3 Notationen und Konventionen

In Anhang B werden die für das Verständnis dieser Arbeit notwendigen Begriffe definiert und zueinander in Beziehung gesetzt. Schlüsselbegriffe, wie z.B. *Delegation* werden zusätzlich bei ihrer Einführung vorgestellt. Für Begriffe aus dem allgemeinen Wortschatz der objektorientierten Softwareentwicklung erfolgt dies nicht. Ab Kapitel 4 sind Grundkenntnisse in der Programmiersprache Eiffel erforderlich, da die Beispiele bis auf wenige Ausnahmen in dieser Sprache formuliert werden. Eiffel wird in [Mey97, Mey92] beschrieben.

In den Beispielen, die als Diagramme oder Quelltexte zur Veranschaulichung und Motivation bestimmter Sachverhalte dienen, haben Klassen und Eigenschaften häufig nur symbolische Namen. Ein

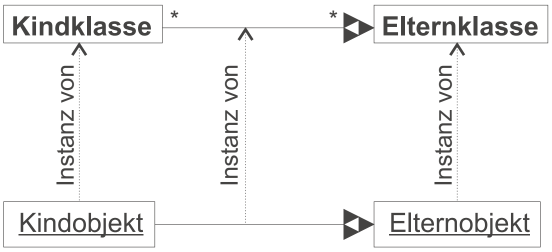

Abbildung 1.2: Delegationsbeziehungen werden in der UML als Assoziationen mit dem Stereotyp "Delegation" dargestellt. Das Symbol hebt die feststehende *self*-Referenz hervor.

verständliches Beispiel, mit Namen aus dem alltäglichen Sprachgebrauch (z.B. Person, Kunden, usw.) ist oft recht schwierig zu finden, insbesondere wenn es um einen kleinen, fest umrissenen Sachverhalt geht, der ohne einen größeren Zusammenhang betrachtet wird. Für die Benennung von Bezeichnern sollen daher folgende Konventionen gelten:[1]

- Klassen werden mit Großbuchstaben bezeichnet, z.B. `A`, `B`, `X` oder `Y`.
- Konkrete Klassennamen werden in Quelltext-Beispielen in Großbuchstaben geschrieben. Besteht der Name aus mehreren Wörter, werden die einzelnen Wörter durch einen Unterstrich voneinander getrennt, z.B. `MAUS_BEARBEITER`. In Diagrammen werden auch Kleinbuchstaben verwendet. Dadurch ergibt sich eine wesentlich bessere Lesbarkeit. Durch diese Abweichung sollten keine Verständnisprobleme entstehen.
- Objekte erhalten das Präfix `obj_`, gefolgt vom Namen der Klasse, deren Instanz sie sind, als Kleinbuchstabe. Ein Objekt `obj_a` ist demnach eine Instanz der Klasse `A`. Sollten in einem Diagramm oder Quelltextbeispiel mehrere Instanzen einer Klasse vorkommen, werden sie durchnummeriert, z.B. `obj_a1`, `obj_a2`, usw.
- Merkmale einer Klasse werden ebenfalls durchnummeriert. Wenn eine Unterscheidung zwischen Attributen und Methoden nicht erforderlich ist, werden die Namen `merkmal_1`, `merkmal_2`, usw. benutzt. Anderfalls werden Namen wie `methode_1` oder `attribut_1` vergeben.

Die Diagramme zur Veranschaulichung von Sachverhalten benutzen als Notation die *Unified Modeling Language (UML)*, wie sie in [Bur97] vorgestellt wird. Delegationsbeziehungen zwischen Klassen und Objekten erhalten, da sie Thema dieser Arbeit sind, ein eigenes Symbol (siehe Abbildung 1.2). Dieses Symbol wurde in [Cos98] eingeführt. Es drückt die Verwandtschaft mit Vererbungsbeziehungen aus; der rückwärts gerichtete Pfeil symbolisiert die beim ursprünglichen Empfänger beginnende Suche nach Methoden oder Attributen. In der UML kann die besondere Semantik von Elementen durch *Stereotypen* spezifiziert werden. Eine Delegationsbeziehung ist daher eine Assoziation mit dem Stereotyp "Delegation". Werkzeuge, die diesen Stereotyp interpretieren können, werden das Delegationssymbol darstellen, andere Werkzeuge eine gewöhnliche Assoziation mit dem Stereotypen "<< Delegation >>" als Text anzeigen.

1.4 Danksagungen

Zunächst bedanke ich mich bei Herrn Dr. Hasselbring für die Betreuung der Arbeit und die Anregungen zum Inhalt und zur Form der Arbeit. Herrn Prof. Doberkat danke ich für die Einschränkung des ursprünglich geplanten Themenbereichs auf die in dieser Arbeit behandelten Kernpunkte und für die Übernahme des Zweitgutachtens. Armin Freese, Klaus Kneupner und Wolfgang Franke haben mir

[1] Die Konventionen folgen im wesentlichen den Namenskonventionen für Eiffel, wie sie in [Mey97, Abschnitt 26.2] vorgeschlagen werden.

beim Anfertigen der Arbeit durch Diskussionen und kritische Betrachtungen meiner Überlegungen viel geholfen, wofür ihnen besonderer Dank gilt. Herr Dr. Dißmann hat mich bereits lange vor Beginn der Diplomarbeit durch Diskussionen und Informationen über Delegation dazu motivert, mich näher mit dem Thema zu befassen, wodurch letztendlich diese Arbeit zustande gekommen ist. Günter Kniesel hat mir durch seine Arbeiten über Delegation viele neue Anregungen gegeben, wofür ich ihm ebenfalls zu danken habe.

Kapitel 2

Motivation für die Modellierung mit Delegation

Das einleitende Beispiel über Delegation hat zwar in aller Kürze aufgezeigt, wie Delegation funktioniert, die Frage, ob die Benutzung sinnvoll ist, blieb jedoch offen. In diesem Kapitel soll durch ausgewählte Beispiele gezeigt werden, wo Delegationsbeziehungen zu einfacheren und eleganteren Modellen führen. Neben einer rollenbasierten Systemsicht wird auf *Assimilation* eingegangen. Anschließend werden *Entwurfsmuster* vorgestellt.

Insbesondere in Abschnitt 2.1 könnte der Eindruck entstehen, daß Vererbung für die natürliche Modellierung eines Anwendungsbereichs nicht geeignet ist. Dieser Eindruck ist falsch und soll deshalb schon vor seiner Entstehung korrigiert werden. Vererbung ist wichtiges Konzepte in der objektorientierten Softwareentwicklung und für viele Anwendungsbereiche das Mittel der Wahl. Es gibt jedoch einige Fälle, in denen Vererbung zu unflexibel ist. Die folgenden Beispiele sind speziell ausgewählt, um dies zu zeigen.

2.1 Rollenbasierte Systemsicht

In diesem Beispiel soll ein Ausschnitt einer Universität modelliert werden. Um das Beispiel möglichst einfach zu halten, werden nur einige wenige Personengruppen betrachtet. In dem Modell sollen Studenten und Angestellte berücksichtigt werden. Es wird gezeigt, unter welchen Voraussetzungen eine rollenbasierte Modellierung [DG95, FH97] einer mittels Vererbung überlegen ist.

2.1.1 Die "klassische" Modellierung

Eine naheliegende Möglichkeit, den Ausschnitt des Anwendungsbereichs zu modellieren, ist sicherlich die in Abbildung 2.1 dargestellte Form. Die gemeinsamen Eigenschaften von Studenten und Angestellten, z.B. Name, Anschrift, Geburtsdatum, usw, werden in einer gemeinsamen Oberklasse `Person` zusammengefaßt. Die beiden Klassen `Angestellter` und `Student` erben von ihr und beinhalten die Eigenschaften, die spezifisch für eine bestimmte Personengruppe sind.

Diese Art der Modellierung ist zwar klar und einfach, man stößt mit ihr jedoch dann auf Schwierigkeiten, wenn Studenten während ihres Studiums als studentische Hilfskräfte arbeiten, also auch Angestellte sind. Da ein `Student`-Objekt keine Angestelltendaten aufnehmen kann, gibt es keine Möglichkeit, einen Studenten direkt als Angestellten zu führen. Als Lösung bietet sich an, für eine studentische Hilfskraft ein `Angestellten`-Objekt zu erzeugen, daß die gleichen Personendaten enthält wie das zugehörige `Student`-Objekt und zusätzlich die Angestelltendaten aufnimmt. In der Verwaltungspraxis

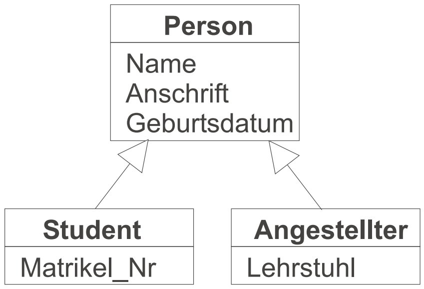

Abbildung 2.1: Modellierung mit Einfacherbung.

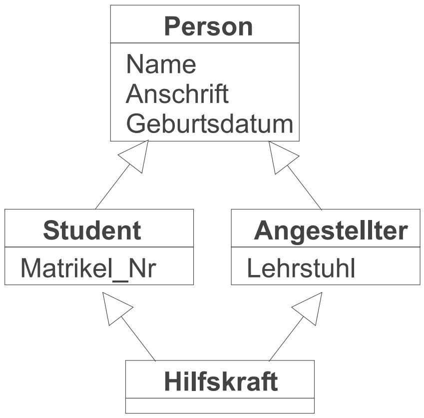

Abbildung 2.2: Durch Mehrfacherbung können mehrere Sichten auf ein Objekt bestehen. Diese Sichten sind jedoch statisch.

einer Universität ist diese Lösung vielleicht praktikabel, unbefriedigend ist sie aber trotzdem. Zur Laufzeit existieren zwei Objekte, die vollständig unabhängig voneinander sind, jedoch trotzdem denselben Menschen repräsentieren. Die Personendaten dieses Menschen sind doppelt vorhanden und müssen manuell konsistent gehalten werden.

2.1.2 Modellierung mit Mehrfacherbung

Eine alternative Modellierung, die das beschriebene Problem lösen würde, ist der in Abbildung 2.2 gezeigte Ansatz, der auf Mehrfacherbung basiert. Es wird eine neue Klasse `Hilfskraft` angelegt, die sowohl von `Angestellter`, als auch von `Student` erbt. Die Eigenschaften der Klasse `Person` werden jetzt zwar auf zwei Pfaden geerbt, sind jedoch in der Klasse `Hilfskraft` nur einmal vorhanden. Bei der späteren Implementierung muß eine objektorientierte Programmiersprache entsprechende Konstrukte dafür zur Verfügung stellen.

Obwohl dieser Entwurf für eine studentische Hilfskraft nun genau ein Objekt benötigt und Personendaten nicht mehr redundant gehalten werden müssen, hat auch dieser Entwurf zwei entscheidende Nachteile:

1. Man stelle sich ein Szenario vor, bei dem mehr als zwei Klassen zusammenzufassen sind. Wenn jede mögliche Kombination von Klassen modelliert wird, wächst die Zahl der Klassen stark an. Es kommt zu einer *kombinatorischen Explosion* von Klassen. Der Entwurf erhält nur durch die Zahl der Klassen eine beachtliche Komplexität. Ein relativ einfaches Ausgangsproblem wird hinter einer großen Menge von Klassen quasi versteckt.

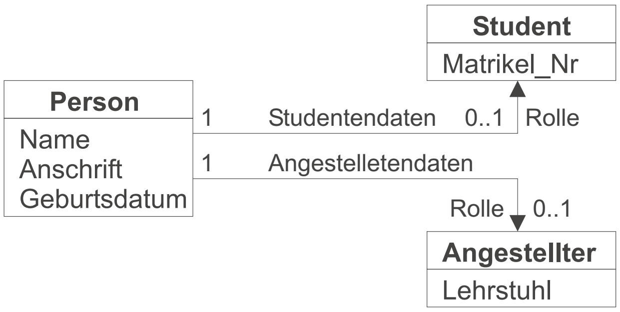

Abbildung 2.3: In diesem Beispiel sind Rollenfüller und Rollen auf mehrere Objekte verteilt. Eine dynamische Anpaßbarkeit ist zwar gewährleistet, jedoch geht aus den Assoziationen nicht hervor, daß es sich um eine Rollenbeziehung handelt.

2. Ein Student kann entweder als einfacher Student *oder* als Hilfskraft geführt werden. Bei der Immatrikulation muß zwischen beiden Möglichkeiten eine Entscheidung getroffen werden, die bis zur Exmatrikulation unveränderlich ist. Um dieses Problem zu lösen, müßten entweder für alle Studenten auch Angestellendaten gespeichert werden, was eine Verschwendung von Speicherplatz bedeuten würde, oder es müßte bei jedem Wechsel zwischen “nur studieren” und “studieren und arbeiten” das “alte” Objekt gelöscht und ein ”neues” angelegt werden. Beide Methoden sind wenig elegant.

2.1.3 Modellierung mit Assoziationen

Bisher wurde die bei der objektorientierten Modellierung übliche Sichtweise benutzt, daß ein Student eine Person ist, bzw. daß eine Person ein Student oder ein Angestellter ist. Vererbung induziert genau diese Sichtweise. Eine alternative Sichtweise wäre es, wenn man Studenten und Angestellte als *Rollen* sehen würde, die Personen einnehmen können.

Diese rollenbasierte Sichtweise läßt sich mit Hilfe von Assoziationen modellieren, etwa so wie in Abbildung 2.3 dargestellt. Für jeden Studenten existieren zwei Objekte, eines ist Instanz der Klasse `Person`, das andere Instanz der Klasse `Student`. Im Gegensatz zu der in Abschnitt 2.1.1 vorgeschlagenen Notlösung referenziert das `Person`-Objekt jedoch das `Student`-Objekt, d.h. man kann über ein `Person`-Objekt indirekt z.B. auf die Matrikelnummer zugreifen, sofern die Person Student ist. Nach [FH97] wird das `Student`-Objekt in diesem Fall als *Rolle* bezeichnet, das `Person`-Objekt als *Rollenfüller*, also als jemand, der eine Rolle einnimmt. Wenn ein Student eine Anstellung als studentische Hilfskraft bekommt, wird ein `Angestellten`-Objekt angelegt und mit dem `Person`-Objekt verbunden. Bei Auflösung des Dienstverhältnisses kann das `Angestellten`-Objekt wieder gelöscht werden. Die Existenz eines Rollen-Objektes hängt dabei von der Existenz eines Person-Objektes ab, umgekehrt besteht diese Abhängigkeit nicht.

Einfache Assoziationen, z.B. durch Referenzen realisiert, haben jedoch einen Nachteil. In der realen Welt fragt man eine Person einfach nach ihrer Matrikelnummer, und sie wird diese nennen, wenn sie Student ist. In dem bisher entwickelten, rollenbasierten Modell ist die Sache komplizierter. Hier muß man z.B. die Person Jack (der Präsident hat früher mal studiert) zunächst nach dem Studenten Jack befragen. Man erhält eine Referenz auf das `Student`-Objekt und kann erst dann die Matrikelnummer ermitteln. Es entsteht ein Zustand, den man Schizophren nennen könnte.

2.1.4 Delegation

Mit Delegation kann man in diesem Fall die Vorteile von Vererbung und Assoziationen zu einem wirklich rollenbasierten System vereinen. Wenn zwischen dem `Person`-Objekt (*delegierendes Objekt* oder *Kindobjekt*) und dem `Student`-Objekt (*delegiertes Objekt* oder *Eobj*) eines Studenten eine Delegationsbeziehung besteht, kann der Zugriff auf die Matrikelnummer direkt über das `Person`-Objekt erfolgen. Da dieses keine Möglichkeit hat, eine solche Frage selbst zu beantworten, wird die Anfrage automatisch an das zugehörige `Student`-Objekt weitergeleitet. Das `Person`-Objekt spielt seine Rolle als Student. Der Initiator der Anfrage sieht die Indirektion nicht. Für ihn sieht es so aus, als hätte das `Person`-Objekt selbst geantwortet.

Bei der rollenbasierten Modellierung ist der Rollenfüller gegenüber den Rollen explizit ausgezeichnet. Im folgenden Abschnitt wird gezeigt, daß man in bestimmten Situationen auf diese Hervorhebung eines Objekts verzichten kann.

2.2 Assimilation

Wenn die Anforderungen an das System vorsehen, daß sowohl über ein `Person`-Objekt auf die Studenten- und Angestelltendaten zugegriffen werden muß, als auch der umgekehrte Weg möglich sein soll, so verschwimmt das Konzept der Rollen. Es gibt dann keinen ausgezeichneten Rollenfüller und untergeordnete Rollen mehr. Statt dessen ergibt sich die Situation, daß mehrere Objekte gemeinsam eine einzige Entität aus dem Anwendungsbereich (in diesem Fall einen Menschen) repräsentieren, man spricht von einem *Objektverbund*. Ein solcher Verbund kann dynamisch erweitert oder verkleinert werden. Die Aufnahme eines Objeks in einem Verbund wird Assimilation [KS96, Kat97] genannt. Ein solcher Verbund hat nach außen hin nur eine einzige Identität; es wird ja nur eine einzige Entität des Anwendungsbereichs repräsentiert.

Ein Benutzer greift zwar immer noch auf ein einzelnes Objekt zu, dies ist für ihn jedoch nur ein Eintrittspunkt, über den er Zugriff auf den gesamten Verbund erhält. Die Situation ähnelt den Voraussetzungen für den Einsatz des Entwurfsmusters *Fassade* [GJHV95, Seite 185]. Ein vom Verbund assimiliertes Objekt stellt einem Benutzer eine homogene Schnittstelle zur Verfügung, die die Schnittstellen anderer Objekte des Verbunds beinhaltet.

Die Verbindungen zwischen den einzelnen Objekten sind dann idealerweise Delegationsbeziehungen. Damit ist die Forderung erfüllt, über ein Objekt transparent auf den ganzen Objektverbund zugreifen zu können. Natürlich müssen nicht zwangsläufig zwischen je zwei Objekten (direkt oder indirekt) Delegationsbeziehungen bestehen. Dies würde zu einer *kombinatorischen Explosion* von Beziehungen führen. Vielmehr wird Art der Benutzung des Verbunds Hinweise auf notwendige und unnötige Verbindungen liefern und aufzeigen, über welche Objekte auf andere zugegriffen werden muß.

Das Prinzip der Assimilation hat mit COM [Rog98] seit einiger Zeit eine weitverbreitete praktische Anwendungsbasis. Mittels Assimilation wird erreicht, daß ein Objekt die Schnittstellen (in COM gibt es mehrere für jedes Objekt) eines anderen Objeks als die eigenen anbieten kann. Das Konzept wird in der COM-Literatur Aggregation genannt, eine Bezeichnung, die mit Blick auf die sonst übliche *part-of-Semantik* dieses Begriffes nicht ganz zutrifft.

2.3 Entwurfsmuster

Entwurfsmuster [GJHV95] dienen dazu, in der Entwurfsphase eines Software-Systems zur Lösung bestimmter Probleme vorgefertigte Standardbausteine zur Verfügung zu haben, die sich für dieses Problem bereits in anderen Systemen bewährt haben. Entwurfsmuster beschreiben eine bestimmte Anordnung von Klassen und Objekten und die Beziehungen und Wechselwirkungen zwischen ihnen. Sie sind jedoch nicht mit Klassenbibliotheken zu verwechseln (siehe [Dob96]). Wenn ein Entwickler beim Entwurf ein bestimmtes Problem isolieren kann, zu dem er ein passendes Entwurfsmuster kennt, braucht er das Rad nicht neu zu erfinden, sondern erhält durch Anwendung des Musters schnell eine flexible und erprobte Lösung [HZ97].

Eine wichtige Methode, mit denen Entwurfsmuster die Flexibilität, Wiederverwendbarkeit und Erweiterbarkeit eines Entwurfs verbessern, ist die *Dekomposition von Funktionalität* in mehrere Objekte.

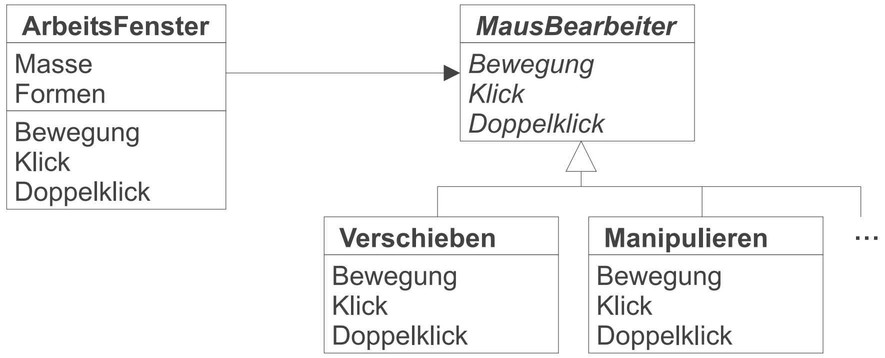

Abbildung 2.4: Das Strategy-Pattern erlaubt den dynamischen Austausch von Algorithmen.

Zusätzlich wird Vererbung dazu benutzt, um gemeinsame Schnittstellen zu schaffen. Von abgeleiteten Klassen instanziierte, zur Laufzeit flexibel austauschbare Objekte implementieren die Schnittstellen mit unterschiedlicher Funktionalität.

Zur Veranschaulichung soll eine mögliche Implementierung für eine allseits bekannte Anwendung dienen. Ein Zeichenprogramm erlaubt es dem Anwender, verschiedene Symbole aus einer Werkzeugleiste auszuwählen und mit der Maus auf einer Arbeitsfläche zu plazieren. Einmal plazierte Symbole können durch "anfassen" mit der Maus verschoben oder in der Größe verändert werden.

Das Fenstersystem, auf dem die Anwendung aufsetzt, übernimmt eine Vorverarbeitung der Maus-Eingaben und schickt der Anwendung Nachrichten über die Aktionen, die der Anwender mit dem Zeigegerät ausführt. Das Problem ist nun folgendes: Die Ereignisse müssen zu sinnvollen Reaktion des Programms führen. Es stellt sich heraus, daß abhängig von der gerade laufenden Benutzeraktion die Reaktionen sehr unterschiedlich sein können. Versucht der Anwender, die Größe eines Symbols zu verändern, muß beim Bewegen der Maus etwas anderes geschehen, als wenn ein neues Symbol auf die Arbeitsfläche gezogen wird.

Eine gute, weil bereits erprobte, erweiterbare und flexible Lösung bietet das Entwurfsmuster *Zustand* [GJHV95, Seite 305 ff.]. Jeder der möglichen Zustände wird in eine eigene Klasse gekapselt. Alle Zustands-Klassen erben, wie in Abbildung 2.4 dargestellt, die Schnittstelle der abstrakten Oberklasse `MAUS_BEARBEITER`. Der Empfänger der Mausnachrichten, im Beispiel die Klasse `ARBEITS_FENSTER`, arbeitet mit den Zustands-Objekten, indem er die empfangenen Mausnachrichten über die gemeinsame Schnittstelle an ein konkretes Zustands-Objekt weiterleitet. Die Klasse `ARBEITS_FENSTER` kennt die einzelnen Zustände nicht. Eine entsprechende Implementierung vorausgesetzt, können sich die einzelnen Zustands-Objekte untereinander zum jeweils aktuellen machen.

Durch den Einsatz des Musters Zustand erzielt man eine Entkopplung zwischen dem Arbeitsfenster und der Funktionalität, die darin je nach aktuellem Zustand nutzbar ist. Wenn die Anwendung um neue Funktionen bereichert werden soll, braucht für die Verarbeitung der Mausereignisse nur eine neue Zusands-Klasse entworfen zu werden. Die Klasse `AREITS_FENSTER` kann unverändert bleiben.

Welchen Vorteil bietet Delegation hierbei? Die Klasse `ARBEITS_FENSTER` hat verschiedene Methoden, die vom Fenstersystem für verschiedene Mausaktionen des Anwenders aufgerufen werden. Diese Aufrufe müssen an das Zustands-Objekt weitergeleitet werden. Zwischen `ARBEITS_FENSTER` und `MAUS_BEARBEITER` besteht eine Delegationsbeziehung.

In [GJHV95] wird eine Implementierung in C++ vorgeschlagen, die hier ohne konzeptionelle Änderung nach Eiffel übertragen wurde. Sie führt in der Klasse `ArbeitsFenster` zu Methoden der folgenden Bauweise:

```
bewegung( xpos : integer, ypos : integer ) is
do
   zustand.bewegung( Current, xpos, ypos )
end

klick is
do
   zustand.klick( Current )
end
```

Mit einer Programmiersprache, die Delegationsbeziehungen direkt unterstützt, werden die trivialen Implementierungen der obigen Methoden überflüssig. Wenn `ARBEITS_FENSTER` an `MAUS_BEARBEITER` delegiert, kümmert sich das Laufzeitsystem der Sprache automatisch um die Weiterleitung der Ereignisse. Der Zeiger auf das Arbeitsfenster, der jeder Methode als zusätzlicher Parameter übergeben wird, könnte entfallen, da `Current` weiterhin auf das Arbeitsfenster zeigen würde (ssiehe Abschnitt 1.1). Ein weiterer Vorteil sticht vielleicht nicht sofort ins Auge: die Klasse `ARBEITS_FENSTER` kann auch dann unverändert bleiben, wenn neue Mausereignisse hinzukommen. Es brauchen dann nur `MAUS_BEARBEITER` und ihre Unterklasseen angepaßt zu werden.

Wie dieses Beispiel mit E+ einfacher und eleganter implementiert werden kann, ist Thema des Abschnitts 9.3. An diser Stelle wird kein Beispiel in E+ angegeben, da dies ein Vorgriff auf Konstrukte bedeuten würde, die erst in den Kapiteln 5 und 6 beschrieben werden.

Kapitel 3

Delegation

In diesem Kapitel werden objektorientierte und delegationsbasierte Programmiersprachen konzeptionell betrachtet. Beide Paradigmen haben ihre Vor- und Nachteile und beide haben ihre Anwendungsgebiete. Die Unterschiede und Gemeinsamkeiten werden herausgearbeitet und sollen Ansatzpunkte für die Integration von Konzepten aus beiden Paradigmen in einer Sprache geben.

3.1 Objektorientierte Programmiersprachen

Eine Vererbungsbeziehung zwischen zwei Klassen wird oft auch als *Generalisierung/Spezialisierung* bezeichnet, d.h. die erbende Klasse ist eine Spezialisierung der vererbenden Klasse, die wiederum eine Generalisierung der erbenden Klasse ist [Mey97]. Der Prozeß der Klassenfindung, sowie die Aufstellung einer sinnvollen, d.h. für einen Problembereich angemessenen Hierarchie von Vererbungsbeziehungen ist Thema vieler Bücher über objektorientierte Analyse und Entwurf, z.B. [Boo94, RBP+91, Jac92]. Für die folgende Betrachtung von objektorientierten Systemen sind solche Aspekte jedoch weitgehend ohne Bedeutung.

3.1.1 Statische vs. dynamische vs. explizite vs. implizite Typisierung

In objektorientierten Programmiersprachen ist die Menge der Methoden eines Objekts, durch die Klasse des Objekts festgelegt. Spätestens beim Aufruf einer Methode für ein bestimmtes Objekt muß daher festgestellt werden, zu welcher Klasse das Objekt gehört. Mit dieser Information kann dann geprüft werden, ob der Aufruf erlaubt ist. Dieser Vorgang wird als *dynamische Typisierung* bezeichnet und bietet hohe Flexibilität bei der Entwiclung eines Software-Systems. Für den Fall, das ein Aufruf nicht erlaubt ist, kommt es zu einem Laufzeitfehler, der zu einem Abbruch des Programms führen kann. Ein bekannter Vertreter dynamisch getypter objektorientierter Sprachen ist Smalltalk [GR89].

Im Gegensatz dazu enthalten *statisch getypte* Sprachen Regeln, die es durch statisch Analyse des Programmtextes, d.h. ohne vorherige Ausführung ermöglichen, die Gültigkeit eines Methodenaufrufs festzustellen. Da Objekte meist über Referenzen angesprochen werden, erlauben es diese Regeln, die Menge der möglichen Objekte hinter einer Referenz (den dynamische Typ [Has95]) festzustellen. Über eine Referenz können dann nur solche Methodenaufrufe erfolgen, die für alle möglichen Objekte erlaubt sind. Dieses Verfahren ist pessimistisch. Ein Methodenaufruf, der nicht für alle möglichen Objekte erlaubt ist, wird abgewiesen, auch dann, wenn ein Entwickler sicher ist, daß die Referenz zum Zeitpunkt des Aufrufs auf ein Objekt verweist, für das der Aufruf erlaubt wäre. Ein in einer statisch getypten Sprache geschriebenes Programm ist dafür nach Überprüfung durch einen Compiler garantiert frei von Typfehlern. Dies ist ein wichtiges Kriterium für die Verläßlichkeit eines Software-Systems. Vertreter statisch getypter objektorientierter Sprachen sind z.B. C++, Java und Eiffel.

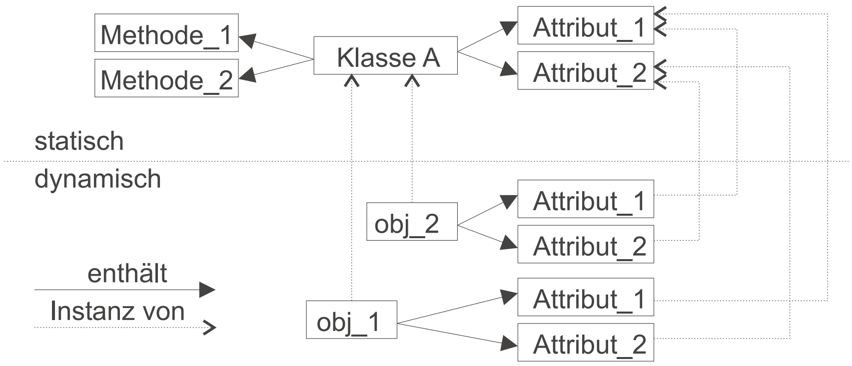

Abbildung 3.1: Attribute und Methoden werden in objektorientierten Systemen unterschiedlich behandelt

In den drei genannten Sprachen muß der statische Typ einer Referenz explizit vom Entwickler angegeben werden. Dieses Verfahren wird explizite Typisierung [Mey89] genannt. Der Entwickler gibt mit dem statischen Typ die beabsichtigte Verwendung einer Referenz an. Ein Compiler ist in der Lage, den statischen Typ, auch ohne explizite Angabe, aus dem Kontext heraus abzuleiten. Die Typangabe durch den Entwickler ist daher redundant. Sie ermöglicht es einem Compiler jedoch, Abweichungen zwischen der beabsichtigten und der tatsächlichen Verwendung einer Referenz festzustellen und dadurch bei der Aufdeckung logischer Fehler zu helfen. Eine Sprache mit statischer aber nicht expliziter Typisierung ist ML [Ull94].

3.1.2 Unterschiede zwischen Attributen und Methoden

Bei der Instanziierung eines Objekts muß soviel Speicherplatz reserviert werden, daß alle Attribute Platz finden. Jedes Objekt hat seinen eigenen Satz von Attributen. Die Änderung von Attributwerten an einem Objekt ändert nicht die Attribute anderer Objekte.

Alle Objekte einer Klasse benutzen jedoch für ihre Attribute die gleichen Namen. Diese Namen sind durch die gemeinsame Klasse vorgegeben und können nicht objektweise verändert werden. Wenn ein Aufruf `obj_A.attribut_1` für ein Objekt einer Klasse `A` erlaubt ist, so ist er auch für jede andere Instanz dieser Klasse erlaubt. Dieses Verhalten wird als *Namensteilung* [BD96] bezeichnet.

Für Methoden sieht die Sache etwas anders aus. Eine Methode wird nur einmal für alle Objekte einer Klasse implementiert. Alle Objekte teilen sich also eine einzige Implementierung. Wird diese Implementierung geändert, so ändert sie sich für alle Objekte. Dieses Verhalten wird als *Eigenschaftsteilung* bezeichnet. Abbildung 3.1 veranschaulicht diesen für objektorientierte Programmiersprachen typischen Unterschied in der Behandlung von Attributen und Methoden.

3.1.3 Die Semantik von Vererbungsbeziehungen

In dem in Abbildung 3.2 dargestellten Szenario erbt eine Klasse `B` von einer Klasse `A`. Die Klasse `A` enthält zwei Methoden, `methode_1` und `methode_2`. Um ihre Aufgabe verrichten zu können, ruft `methode_1` `methode_2` auf. In `B` wird `methode_2` redefiniert.

Ein Objekt `obj_b` das Instanz von `B` ist, ist durch die Vererbungsbeziehung auch (indirekte) Instanz von `A`. Der Aufruft `obj_b.methode_1` bewirkt die Ausführung der Methode `A.methode_1`. Da in `B` keine passende Methode existiert, wird in der Oberklasse von `B` gesucht und die Methode `A.methode_1` gefunden. Innerhalb von `A.methode_1` wird `methode_2` aufgerufen. Der Aufruf ist kein Prozeduraufruf, wie er in der imperativen Programmierung gemacht wird, sondern ein Methodenaufruf, also der Aufruf

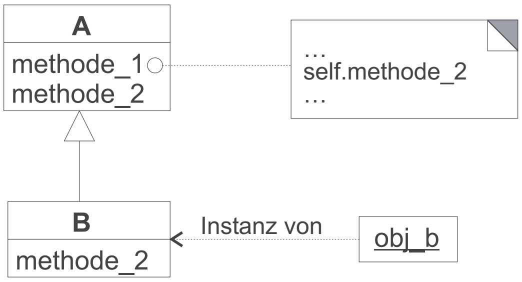

Abbildung 3.2: Vererbungsbeziehungen sind Delegationsbeziehungen zwischen Klassen.

einer Operation für ein bestimmtes Objekt. Konzeptionell steht dort also `self.methode_2`. Die `self`-Referenz (siehe Abschnitt 1.1) veweist auf `obj_b`. Das Laufzeitsystem berücksichtigt daher, daß `obj_b` eine Instanz von `B` ist und beginnt die Suche nach einer passenden Implementierung bei `B`. In `B` existiert eine Methode `methode_2`, die den Vorzug vor `A.methode_2` erhält. Dieser Vorgang wird *spätes Binden* genannt.

Das beschriebene Verhalten entspricht genau der Definition von *Delegation* in Abschnitt 1.1. Eine Vererbungsbeziehung ist also eine Delegationsbeziehung zwischen Klassen.

3.2 Delegationsbasierte Programmiersprachen

In objektorientierten Sprachen wird die Struktur und das Verhalten eines Objekts durch seine Zugehörigkeit zu einer bestimmten Klasse festgelegt. Diese Zugehörigkeit ist im allgemeinen über die gesamte Lebenszeit eines Objekts unveränderlich. Alle Instanzen einer Klasse haben die gleiche Struktur und die gleichen Eigenschaften. Ein Objekt wird beschrieben, indem seine Klasse beschrieben wird. Für bestimmte Anwendungen ist dies jedoch zu unflexibel. Im folgenden Abschnitt wird daher die Entwicklung delegationsbasierter Programmiersprache begründet.

3.2.1 Gründe für die Entwicklung klassenloser Prototyping-Sprachen

Klassen stellen Mengen dar, die Objekte als Elemente enthalten. Sie definieren die Eigenschaften, die allen Elementen (Objekten) der Menge gemeinsam sind. Anstatt mit einzelnen Objekten umzugehen, werden in klassenbasierten Sprachen zunächst Mengen von Objekten beschrieben, ehe daraus ein Objekt erzeugt werden kann. Die klassenbasierte Modellierung eines Software-Systems sucht also zunächst nach Verallgemeinerungen, bevor konkrete Objekte behandelt werden können. Gängige objektorientierte Analyse- und Entwurfsmethoden legen daher auch viel Wert auf den Prozeß der Klassenfindung (siehe [Boo94, RBP$^+$91, Jac92]).

Diese Vorgehensweise entspricht nach [Lie86] jedoch nicht der menschlichen Denkweise. Stattdessen lernen Menschen anhand von konkreten Beispielen, von denen sie auf andere Situationen schließen. Prototyping-Sprachen, wie z.B. SELF [US87], versuchen die menschliche Denkweise besser anzunähern. Da beim Prototyping zunächst Anforderungen an ein späteres System festgelegt werden sollen, haben (klassenlose) Prototyping-Sprachen hier entscheidende Vorteile gegenüber objektorientierten Programmiersprachen. Sie erlauben es, schnell lauffähige Anwendungen zu entwickeln und unterstützen die schnelle Umsetzung von Änderungen. So kann "mal eben" das Verhalten eines Objeks geändert werden, ohne daß dies sofort Auswirkungen auf viele andere Objekte hat.

Prototyping-Sprachen haben in der Regel kein statisches Typsystem. Daraus ergeben sich Nachteile bezüglich der Verläßlichkeit und Korrektheit komplexer Software-Systeme. Prototyping-Sprachen werden daher meist nur für die Anforderungsermittlung, nicht jedoch für die entgültige Realisierung eines Software-Systems eingesetzt.

3.2.2 Selbstbeschreibende Objekte

Durch das Fehlen von Klassen als Vorlagen oder Baupläne für Objekte muß die Struktur und das Verhalten eines Objeks in diesem selbst beschrieben werden. In den folgenden Beschreibungen wird die Terminologie der Sprache SELF benutzt. Die vorgestellten Konzepte gelten jedoch zum großen Teil auch für andere klassenlose Sprachen, z.B. für Newton-Skript [Smi95].

Ein Objekt besteht aus einer Menge von *Slots*. Ein Slot kann man sich dabei als ein Fach vorstellen, in dem ein anderes Objekt, genauer: eine Referenz darauf, Platz findet. Der große Vorteil des Slot-Konzepts ist die identische Behandlung von Attributen und Methoden. Attribute, auch solche mit elementaren Datentypen (z.B. integer), sind in SELF Objekte. Um den Zustand eines Attributs zu erfahren, schickt man an das Objekt eine Anfragenachricht, worauf sich das Objekt selbst zurückliefert, was genau dem gewünschten Attributwert entspricht. Methoden sind ebenfalls Objekte. Die Ausführung einer Methode erfolgt durch Kopieren einer prototypischen *Aktivierungsumgebung*.

Ein Objekt kann einen *Eltern-Verweis* enthalten. Das ist ein Verweis auf ein anderes Objekt, das *Elternobjekt*. Zusätzlich existiert ein sogenannter *`self`-Verweis*, dieser zeigt immer auf das Objekt, das eine Nachricht empfangen hat (siehe Abschnitt 1.1).

3.2.3 Erzeugung von Objekten

Es wäre sicherlich mühsam, für eine Reihe gleichartiger Objekte jedesmal die Implementierung von Hand vorzunehmen. In SELF können Objekte daher durch Kopieren bereits existierender Objekte, sogenannte Prototypen, erzeugt werden. Dabei werden sowohl Attribute als auch Routinen kopiert. Änderungen an der Implementierung von Routinen der Kopie haben also keine Auswirkungen auf die Routinen der Vorlage. Durch Änderungen an den Slots der Kopie oder durch Hinzufügen neuer erhält das neue Objekt ein von seiner Vorlage abweichendes Verhalten.

In SELF können beliebige Objekte als Prototypen dienen, d.h. Prototypen sind auch ganz normale Objekte. Benutzt man bestimmte Objekte nur dazu, um andere Objekte von ihnen zu kopieren, so hat man das Konstrukt der Klasse praktisch nachgebildet.

3.2.4 Vererbung

Ein SELF-Objekt kann ein Elternobjekt haben. In diesem Fall erbt das Kindobjekt alle Eigenschaften seines Elternobjekts. Dabei werden keine Slots kopiert. Wenn das Kindobjekt eine Nachricht empfängt, so sucht es zunächst bei sich selbst nach einem passenden Slot. Existiert ein solcher, so wird dieser benutzt. Wird jedoch kein passender Slot gefunden, wird die Nachricht an das Elternobjekt weitergeleitet und dort nach einem passenden Slot gesucht. Wurde die Kette bis zu einem elternlosen Objekt verfolgt, ohne daß ein passender Slot gefunden wurde, so handelt es sich um eine illegale Nachricht und damit um einen Laufzeitfehler.

Alle Nachrichten, die bei der Abarbeitung einer Routine versendet werden, sind an ein bestimmtes Objekt gerichtet. Dies ist entweder ein anderes Objekt, oder das sendende Objekt selbst. Der erste Fall dient der Kommunikation zwischen Objekten und ist hier nicht weiter interessant. Im zweiten Fall spricht man von *Nachrichten an `self`*.

Wenn nun ein Objekt eine empfangene Nachricht mangels eines passenden Slots an sein Elternobjekt weiterleitet, so bleibt der `self`-Verweis auf dem weiterleitenden Objekt stehen. Sollte das Elternobjekt Nachrichten an `self` senden, so erfolgt die Suche nach einem passenden Slot also beginnend beim Empfänger der ursprünglichen Nachricht. Diese Verfahrensweise wird als Delegation bezeichnet.

Dank dieses Mechanismus kann ein Objekt Eigenschaften eines direkten oder indirekten Elternobjekts sehr einfach redefinieren. Es reicht aus, einen Slot gleichen Namens einzurichten. Nachrichten werden dann nicht mehr delegiert, sondern direkt von dem eigenen Slot bearbeitet. Wurde eine Nachricht weitergeleitet und werden von einem direkten oder indirekten Elternobjekt zum Slot passende Nachrichten an `self` geschickt, so übernimmt der neue Slot die Bearbeitung. Für die Elternobjekte bleibt diese Verhaltensänderung verborgen.

3.2.5 Abstrakte Objekte: Traits

In SELF gibt es die Möglichkeit, abstrakte Objekte oder Schablonen, genannt *Traits*, zu erzeugen. Dadurch wird zwar das einheitliche Konzept von Nachrichten und Slots aufgeweicht, jedoch erweisen sich Schablonen in der Praxis als sehr nützlich.

Zur Erklärung von Schablonen sei an dieser Stelle noch einmal an abstrakte Klassen in objektorientierten Sprachen erinnert. Dort wurden Eigenschaften einer Klasse zwar spezifiziert, jedoch nicht implementiert. Ähnlich verhält es sich mit Schablonen. Eine Schablone ist also ein Objekt, das in seinen Methoden Slots anspricht, die weder von der Schablone selbst, noch von einem direkten oder indirekten Elternobjekt definiert werden. Demnach kann eine Schablone keine externen Nachrichten empfangen. Sie kann jedoch als Elternobjekt für andere Objekte dienen. Die fehlenden Slots werden von den Kindobjekten implementiert.

Wenn eine Schablone eine Nachricht empfängt, so geschieht dies, da Schablonen keine externen Nachrichten empfangen können, durch Weiterleitung von einem Kindobjekt. Eine Routine kann nun Nachrichten für einen abstrakten Slot an `self` schicken. Der Delegationsmechanismus sorgt dafür, daß `self` auf ein Kindobjekt zeigt, das den abstrakten Slot konkret implementiert hat. In einer Schablone kann also Funktionalität spezifiziert werden, ohne sie bis ins letzte Detail ausarbeiten zu müssen. Schablonen verfolgen demnach den gleichen Zweck wie abstrakte Klassen.

3.2.6 Attribute und Methoden

In delegationsbasierten Programmiersprachen gibt es keine Klassen, in denen Vorgaben für eine Menge von Instanzen gemacht werden könnten. Folglich bestimmt jedes Objekt selbst, wie seine Attribute und Methoden aussehen.

Wenn ein Objekt `obj_x` von einem Objekt `obj_y` erbt, so erbt es dessen Attribute und Methoden. Ändert sich der Wert eines Attributes in `obj_y` so tut er dies auch in `obj_x` und allen anderen Objekten, die ebenfalls von `obj_y` erben. Dies ist nicht verwunderlich, da es sich physisch in allen Objekten um ein und dasselbe Attribut handelt. Die abgeleiteten Objekte verweisen quasi nur auf das Attribut des Elternobjekts.

Bei Methoden ist es genauso. Attribute und Methoden werden also gleich behandelt, es besteht Eigenschaftsteilung, womit Namensteilung natürlich impliziert wird [BD96].

3.2.7 Simulation von Vererbung durch Delegation

In Abschnitt 3.2.3 wurde bereits erwähnt, daß Klassen durch Prototypen simuliert werden können. In Verbindung mit Delegation läßt sich auch Vererbung zwischen Klassen simulieren. Abbildung 3.3 zeigt, wie Objekte angeordnet werden müssen, um die aus objektorientierten Programmiersprachen bekannte Vererbungsbeziehung zu simulieren. Einige Objekte dienen dabei als Klassen, die auf ein einzelnes prototypisches Objekt verweisen, das zur Instanziierung kopiert wird. Dieser Prototyp enthält die Attribute der späteren Instanzen. Die Methoden werden in einem gesonderten Objekt zusammengefaßt. Dadurch entsteht eine Art Methodentabelle, wie sie Compiler für objektorientierte Programmiersprachen generieren [Kni96, Abschnitt 4.1]. Die Abbildung kann als Kontrolle dafür genutzt werden, ob das Prinzip der Delegation bereits richtig verstanden wurde.

Anhänger delegationsbasierter Sprachen (z.B. [BD96, Lie86, SU95]) zeigen damit, daß Delegation das allgemeinere Konzept ist. Die Simulation ist allerdings sehr aufwendig. Viele Objekte müssen erzeugt werden und miteinander kommunizieren. Dabei ist das in Abbildung 3.3 dargestellte Modell noch

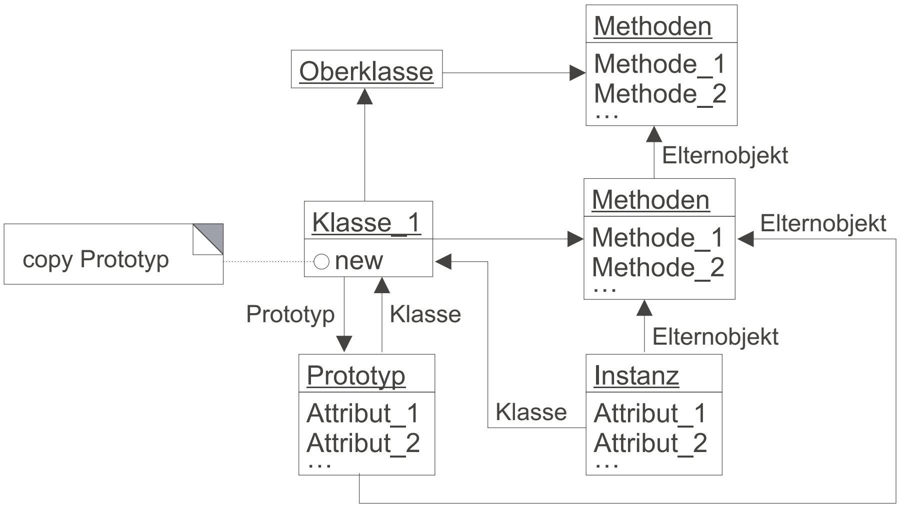

Abbildung 3.3: Objekte können so angeordnet und zueinander in Beziehung gesetzt werden, daß sich mit Vererbung simulieren läßt.

unvollständig, es fehlen z.B. Klassenvariablen. Die Vorteile, die die von vielen objektorientierten Sprachen genutzte statische Typisierung bietet, bleiben ebenfalls ungenutzt. Die Simulation von Vererbung ist also nur ein Beispiel für die Ausdruckskraft delegationsbasierter Sprachen, praktische Bedeutung hat sie nicht.

In [Ste87] wird gezeigt, daß beide Paradigmen, Delegation und Vererbung, letztendlich die gleiche Ausdruckskraft haben. Dazu ist jedoch ein komplexer, mathematisch formaler Beweis erforderlich, weshalb hier nicht näher darauf eingegangen wird.

3.3 Integration von Delegation in objektorientierte Systeme

Im Rahmen der getrennten Betrachtung beider Paradigmen hat sich ein wesentlicher Unterschied herausgestellt: delegationsbasierte Sprachen bieten mehr Flexibilität und schnellere Entwicklung zu Lasten fehleranfälligerer Endprodukte, für objektorientierte Sprachen vertauschen sich Vor- und Nachteile. Im folgenden werden einige konzeptionelle Aspekte der Integration beider Paradigmen sprachenunabhängig untersucht. Eine detailiertere Betrachtung erfolgt speziell für Eiffel ab Kapitel 4.

3.3.1 Ziele der Integration

Eine Integration beider Paradigmen kann von zwei Seiten erfolgen. Zum einen könnten delegationsbasierte Systeme um die Sicherheit statischer Typisierung erweitert werden, andererseits könnten objektorientierte Systeme flexibler gemacht werden, indem Delegation integriert wird. In [SM95] wird z.B. der erste Ansatz verfolgt. In dieser Arbeit soll hingegen das objektorientierte Paradigma als Grundlage dienen und um Aspekte delegationsbasierter Sprachen erweitert werden.

Die am weitesten verbreiteten objektorientierten Programmiersprachen (z.B. Java und C++) sind statisch getypt (siehe Abschnitt 3.1.1). Typfehler und eine Reihe anderer Fehlerquellen (siehe [Mey95, Seite 213 ff.]) können damit von Laufzeitfehlern zu Übersetzungsfehlern abgeschwächt werden, wodurch die Verläßlichkeit eines Software-Systems erhöht wird. Wenn ein Compiler bei der Übersetzung eines Programms keine Typfehler meldet, so kann man sicher sein, daß es keine Typfehler mehr enthält. Eine so absolute Aussage läßt sich durch Testen eines Systems nicht erhalten [Has95].

Durch Delegation soll bei der Entwicklung und Implementierung der beim Anwender zum Einsatz kommenden Endprodukte mehr Flexibilität möglich sein. Die Vorteile objektorientierter Sprachen sollen durch neue Konstrukte jedoch so wenig wie möglich abgeschwächt werden. Die Beherrschung realer Systeme ist bereits ohne Delegation, allein durch ihre Größe, schwierig genug. Wenn zusätzlich mit Delegation umgegangen werden muß, steigt die Komplexität weiter. Die Forderung nach der Erhaltung der Vorteile objektorientierter Sprachen ist daher sehr wichtig.

3.3.2 Delegationsbeziehungen zwischen Klassen

Um in einem objektorientierten System mit Delegation arbeiten zu können, ist es zunächst sinnvoll, die Delegationsbeziehung zwischen Klassen zu definieren. Als Erinnerung wird vorab noch einmal die Definition für Delegation zwischen Objekten gegeben:

> Ein (delegierendes) Objekt `a` leitet empfangene Nachrichten, für die es selbst keine passende Methode besitzt, an ein (delegiertes) Objekt `b` weiter, das die Nachricht bearbeitet oder seinerseits wieder weiterleitet. Schickt `b` im Rahmen der Bearbeitung der Nachricht weitere Nachrichten an sich selbst, so startet die Suche nach einer passenden Methode beim ursprünglichen Empfänger der Nachricht `a`. Das delegierende Objekt heißt *Kindobjekt*, das delegierte Objekt *Elternobjekt*.

Eine Delegationsbeziehung zwischen Klassen ist jetzt sehr einfach:

> Eine (delegierende) Klasse `A` steht in einer Delegationsbeziehung mit einer (delegierten) Klasse `B`, wenn jede Instanz von `A` in einer Delegationsbeziehung zu einer Instanz von `B` steht. Die delegierende Klasse wird *Kindklasse* genannt, die delegierte Klasse heißt *Elternklasse*.

In einem delegationsbasierten System kann ein Objekt eine Methode seines Elternobjekts einfach dadurch überschreiben, daß es sie erneut definiert. Beim Empfang einer entsprechenden Nachricht steht dann sofort eine Methode zur Verfügung und eine Weiterleitung ist nicht erforderlich. In einem objektorientierten System kann dies sehr ähnlich erfolgen. Wenn eine Klasse eine Methode ihrer Elternklasse erneut definiert, so steht sie in Instanzen dieser Klasse direkt zur Verfügung. Die neu definierte Methode verhindert die Weiterleitung einer Nachricht an das delegierte Objekt. In [Kni96] wird ausführlich beschrieben, wie ein Compiler entsprechenden Code generieren kann.

3.3.3 Die Erweiterung des Typsystems

In vielen objektorientierten Programmiersprachen gibt es eine Äquivalenz zwischen Klassen und Typen, d.h. mit jeder Deklaration einer Klasse wird zugleich ein Typ definiert. Wenn ein Objekt `obj_a` Instanz der Klasse `A` ist, so kann man alternativ sagen: `obj_a` ist vom Typ `A`. Eine Vererbungsbeziehung zwischen zwei Klassen `A` und `B`, bei der `B` von `A` erbt, definiert gleichzeitig ein Subtyp-Beziehung zwischen beiden Klassen. Durch die Vererbungsbeziehung ist sichergestellt, daß alle auf Instanzen von `A` anwendbaren Operationen auch auf Instanzen von `B` angewendet werden können. Dies ist eine wichtige Voraussetzung für Subtyp-Beziehungen und Polymorphiemus. Referenzen vom statichen Typ `A` können auf Objekte vom Typ `B` verweisen.

Wenn statt einer Vererbungs- eine Delegationsbeziehung zwischen den Klassen `A` und `B` besteht, wobei `B` an `A` delegiert, existieren zur Laufzeit zwei Objekte `obj_a` und `obj_b`. Wenn auf `obj_b` eine Operation angewendet wird, die in `B` nicht definiert ist, wird automatisch in der Elternklasse `A` nach einer passenden Operation gesucht und diese auf `obj_a` angewendet. Damit stellt die Delegationsbeziehung sicher, daß alle auf Instanzen von `A` anwendbaren Operationen auch auf Instanzen `B` angewendet werden können. Auch Delegationsbeziehungen definieren also Subtyp-Beziehungen. Es bietet sich daher an, eine Definition für Subtyp-Beziehungen zu formulieren, die sowohl Vererbungs- als auch Delegationsbeziehungen umfaßt (siehe Abbildung 3.4):

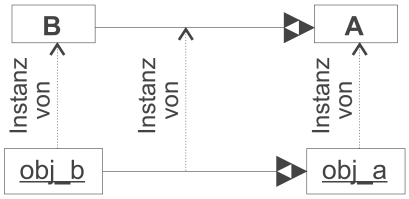

Abbildung 3.4: Eine Delegationsbeziehung zwischen Klassen stellt eine Subtyp-Beziehung dar.

Eine Klasse `B` ist Subtyp einer Klasse `A`, wenn `B` entweder von `A` (direkt oder indirekt) erbt oder an `A` (direkt oder indirekt) delegiert.

3.4 Zusammenfassung

Objektorientierte und delegationsbasierte Programmiersprachen wurden für unterschiedliche Aufgaben konzipiert und verfolgen daher unterschiedliche, teilweise sogar gegensätzliche Ansätze. Die objektorientierten Sprachen werden zur Entwicklung großer komplexer Software-Systeme eingesetzt, wo Korrektheit und Robustheit wichtige Kriterien sind. Delegationsbasierte Sprachen kommen vor allem beim Prototyping zum Einsatz. Hier zählen Flexibilität und intuitives Verständnis. Die Integration von Delegation in das objektorientierte Paradigma erfolgt durch Deklaration von Delegationsbeziehungen zwischen Klassen. Dies bedeutet, daß alle Instanzen der Kindklassen zur Laufzeit an Instanzen der Elternklassen delegieren. Die Austauschbarkeit eines Elternobjekts ist dann zwar nicht beliebig, jedoch ist das Konzept flexibler als die gewöhnliche Vererbung zwischen Klassen.

Kapitel 4

Die Erweiterung von Eiffel

Ziel der Diplomarbeit ist es, eine konkrete, objektorientierte Programmiersprache um objektbasierte Vererbung, d.h. um Delegation zu erweitern. Im Vorfeld der Arbeit wurden mehrere Sprachen auf ihre Vor- und Nachteile im Bezug auf dieses Ziel betrachtet. Nach reiflicher Überlegung fiel die Wahl auf Eiffel. Da diese Entscheidung große Auswirkungen auf den Verlauf der Diplomarbeit hat, soll sie in diesem Kapitel ausführlich begründet werden. Anschließend werden einige grundlegende Richtlinien für die Spracherweiterung aufgestellt.

4.1 Mögliche Kandidaten für eine Erweiterung

Es gibt eine Vielzahl von objektorientierten Programmiersprachen. Einige haben inzwischen eine große Bedeutung bei der industriellen Softwareentwicklung, andere sind Forschungs-Prototypen. In [Sau89] werden über 80 objektorientierte Sprachen behandelt und kategorisiert. Dieses Papier ist annähernd 10 Jahre alt. Wie man sich leicht vorstellen kann, sind in dieser Zeit viele weitere objektorientierte Programmiersprachen (z.B. Java) entwickelt worden, so daß die Zahl der Sprachen heute wesentlich größer sein dürfte. In [Inc98] werden einige der bekanntesten objektorientierten Programmiersprachen tabellarisch mit Eiffel verglichen. Diese Zusammenfassung gibt einen groben Überblick über die Stärken und Schwächen der Sprachen. Bei der Bewertung ist jedoch zu beachten, daß sie von der führenden Firma für Eiffel-Werkzeuge stammt.

Im folgenden werden einige bekannte Programmiersprachen vorgestellt, die Kandidaten für eine Erweiterung sein könnten. Es werden kurz die Besonderheiten jeder Sprache erwähnt und ihr Ausscheiden aus dem Rennen begründet.

4.1.1 C++

Die in der kommerziellen Softwareentwicklung zur Zeit am häufigsten genutzte objektorientierte Programmiersprache ist C++ [Str97]. Sie wurde als objektorientierte Erweiterung von C von Bjarne Stroustrup an den AT&T Bell Laboratories entwickelt [Str94]. Inzwischen gibt es sehr effiziente und schnelle Compiler für diese Sprache und für nahezu jede Zielplattform ausgereifte Entwicklungsumgebungen.

Als Obermenge von C enthält C++ auch eine ganze Reihe nicht objektorientierter Konstrukte. Obwohl man in C++ gut verständliche und leicht wartbare Quelltexte schreiben kann, ist besonders für ehemalige C-Programmierer die Versuchung groß, mit allen Tricks und Kniffen zu arbeiten, was zu schwer wartbaren und fehleranfälligen Programmen führt. Der Sprachumfang von C++ ist enorm. Bei einer Erweiterung dieser Sprache wären eine ganze Reihe von Randbedingungen zu beachten gewesen, die zum einen zu Seiteneffekten in der neuen Sprache führen, zum anderen von den Kernkonzepten der Delegation ablenken würden.

Ein weiterer, im Zusammenhang mit Delegation schwerwiegender Nachteil tritt bei Mehrfacherben auf. C++ unterstützt Mehrfacherben zwar prinzipiell, bietet jedoch keine adäquaten Konstrukte, um die dabei auftretenden Probleme (z.B. Namenskonflikte) zu behandeln (siehe Abschnitt 4.3).

4.1.2 Java

Bei Java [AG98] handelt es sich um eine Sprache, die, obwohl erst wenige Jahre alt, bereits einen hohen Grad an Bekanntheit erreicht hat und auf der ganzen Welt kommerziell eingesetzt wird. Java wurde von Sun Microsystems ursprünglich für eingebettete Systeme, z.B. Waschmaschinen oder Mobilfunk-Telefone (Handys) entwickelt. Ihre Bekanntheit erlangte sie jedoch als die Programmiersprache für das World-Wide-Web. Durch sogenannte Java-Applets kann ein WWW-Server Programme in HTML-Seiten einbetten, die auf dem Client Rechner ausgeführt werden.

Ein Java-Compiler erzeugt statt echtem Maschinen-Code einen plattformunabhängigen *Byte-Code*, was dem von frühen Turbo-Pascal Versionen oder Visual Basic erzeugten P-Code entspricht. Dieser wird auf dem ausführenden Rechner durch die *Java Virtual Machine (VM)* interpretiert. Neben der Plattformunabhängigkeit erreicht man dadurch eine für den Einsatz im Internet wichtige Sicherheit. Die Virtual Machine kann auszuführenden Programmen z.B. Zugriffe auf die Hardware oder das Dateisystem verbieten. Für Programme in Maschinensprache ist das kaum effizient möglich.

Syntaktisch ist Java sehr eng an C++ angelehnt. Eine der Entwicklungsziele von Java war sicherlich, daß Programmierer mit C++ Erfahrung die neue Sprache nach möglichst kurzer Umstiegszeit nutzen können. Allerdings bereinigt Java eine ganze Reihe von Kritikpunkten an C++. Der Kern der Sprache ist klein und übersichtlich gehalten (was hoffentlich in künftigen Versionen der Sprache so bleibt). Es gibt in Java keine Zeiger-Arithmetik (auch ein Zugeständnis an die Sicherheit), was eine Fehlerquelle weniger bedeutet. Außerdem unterstützt Java automatisches Speicher-Management (Garbage-Collection). Für viele Anwendungsbereiche kann man Java als eine echte Verbesserung von C++ ansehen.

Neben Klassen bietet Java *Schnittstellen* (Interfaces) an. Dabei handelt es sich konzeptionell um Klassen, die nur abstrakte Methoden enthalten. Eine Schnittstelle kann von mehreren Schnittstellen erben (das Java-Schlüsselwort für Vererbung lautet: `extends`). Eine Klasse kann mehrere Schnittstellen implementieren (Java-Schlüsselwort: `implements`), jedoch nur von einer anderen Klasse erben. Schnittstellen sind in Java Typen. Referenzen vom statischen Typ einer Schnittstelle verweisen auf Objekte, deren Klassen diese Schnittstelle implementieren. Die Vererbung zwischen Schnittstellen und die Implementierung von Schnittstellen dient dem *Subtyping*, d.h. dem Aufbau einer *Typhierarchie*. Die Vererbung zwischen Klassen dient der Wiederverwendung von Code. Beide Konzepte sollten nicht verwechselt oder vermischt werden [Dob96]. Java trennt, im Gegensatz zu C++ oder Eiffel, sauber zwischen ihnen.

Wenn mehrere Schnittstellen Methoden mit identischer Signatur enthalten, so kommt es zu keinem Konflikt in einer Klasse, die diese Schnittstellen implementiert. Die Klasse wird nur eine Implementierung anbieten, die für jede dieser Methoden genutzt wird. Da zwischen Klassen nur einfaches Erben erlaubt ist, sind Namenskonflikte in Java ausgeschlossen. Entsprechend stellt Java keine Konstrukte zur Auflösung solcher Konflikte zur Verfügung. Dies ist der Grund, warum Java als Grundlage für eine statisch getypte objektorientierte Programmiersprache mit Delegation nicht gewählt wurde. Bei der Erweiterung um Delegation treten Probleme wie das Auflösen von Namenskonflikten auf. Eine Sprache, die hier bereits adäquate Konstrukte anbietet, hat daher klare Vorteile.

In [Cos98] und [Sch98] wird Java trotz dieses Nachteils als Grundlage für eine Erweiterung um Delegation benutzt. Als Ergebnis dieser Arbeiten stehen ein Compiler und eine VM zur Verfügung, die Delegation in Java unterstützen. Zur Lösung von Namenskonflikten wurden Konstrukte aus Eiffel in Java integriert. Die vielleicht offensichtlichere Weg, die Übernahme von Mechanismen aus C++, wurde glücklicherweise nicht gewählt.

4.1.3 Smalltalk

Smalltalk [GR89] ist eigentlich nicht nur eine Sprache, sondern eine komplette Entwicklungsumgebung, die sowohl die eigentliche Sprache, als auch Werkzeuge umfaßt; das eine ist ohne das andere schwer denkbar [Boo94, Seite 587]. Smalltalk basiert im wesentlichen auf zwei Grundideen:

1. Alles wird als Objekt behandelt (auch Klassen, integers, usw.).

2. Objekte kommunizieren über Nachrichten miteinander.

Für den Entwickler gibt es keinen Unterschied zwischen den selbst entwickelten Teilen einer Anwendung und der Entwicklungsumgebung. Die gesamte Funktionalität dieser Umgebung kann bequem in die damit entwickelten Anwendungen integriert werden.

Smalltalk unterstützt nur einfaches Erben. Alle Klassen erben von der obersten Basisklasse `Object` [Mey97, Seite 1126 ff.], so daß die Vererbungshierarchie stets ein Baum ist. Um auch Klassen wie Objekte behandeln zu können, existiert in Smalltalk das Konstrukt der *Metaklasse*, also der Klasse einer Klasse. Demnach ist eine Klasse Instanz ihrer Metaklasse. Klassenvariablen (in C++ und Java als `static` deklarierte Elemente) lassen sich in Smalltalk elegant als Variablen der Metaklasse beschreiben. Smalltalk ist nicht statisch getypt.

In [FH97] wird eine Erweiterung von Smalltalk um Delegation ausführlich vorgestellt. Die Umsetzung von Delegationsmechanismen ist in dieser Arbeit enger an die Konzepte delegationsbasierter Programmiersprachen angelehnt. Die dynamische Typisierung von Smalltalk wird geschickt genutzt, um die Flexibilität delegationsbasierter Sprachen weitgehend zu erhalten. Die Betrachtung von Delegation beschränkt sich im wesentlichen auf die Modellierung von Rollen. Gegenüber der Erweiterung von Java um Delegation (siehe Abschnitt 4.1.2) ergeben sich somit interessante Unterschiede. Die Arbeit stellt Delegation als Entwurfsmuster vor, das nach dem in [GJHV95] vorgeschlagenen Schema beschrieben wird.

4.1.4 BETA

Die Programmiersprache BETA [DD96, MMPN93] weicht konzeptionell und syntaktisch stark von den bisher vorgestellten objektorientierten Sprachen ab. BETA kennt an Stelle von Klassen, Methoden und Attributen nur das *Muster (Pattern)*. Mit Mustern wird die Struktur einer Anwendung beschrieben. Muster enthalten sowohl Submuster, als auch ausführbaren Programm-Code (ähnlich der Schachtelung von Prozeduren in Pascal). Auf diese Weise lassen sich Methoden einfach nachbilden. Muster können wie Klassen instanziiert und diese Instanzen, genau wie in anderen Sprachen, referenziert werden. Vererbung wird in BETA über virtuelle Muster realisiert, dabei wird nur einfaches Erben unterstützt. Auch mit generischen Mustern kann gearbeitet werden.

Die Sprache unterstützt einige sehr interessante Konzepte, so z.B. die Erweiterung von Methoden durch das *inner*-Konstrukt. Dabei wird die Implementierung einer Methode (teilweise) ausgelassen und stattdessen ein Platzhalter gesetzt. Die fehlende Implementierung kann später nachgeliefert werden. Konventionelle objektorientierte Sprachen können Methoden nur vollständig oder gar nicht (abstrakte Methoden) implementieren. BETA ermöglicht auch die Ausführung solcher teilweise implementierten Methoden. Dieses Sprachkonstrukt eignet sich zum Beispiel dazu, um sehr elegante Durchläufe durch Container zu realisieren. Der Durchlauf-Algorithmus wird von einem Muster implementiert, die Aktionen, die für jedes besuchte Element ausgeführt werden sollen, können durch eine *inner*-Anweisung ersetzt und später implementiert werden.

Ein weiteres Merkmal der Sprache ist, daß zwischen dem Zugriff auf Attribute und Methoden kein syntaktischer Unterschied besteht. Ein Aufrufer erfährt also nicht, ob ein Wert gespeichert war oder berechnet wurde. Zwischen statisch und dynamisch erzeugten Instanzen wird hingegen unterschieden. BETA wurde nicht gewählt, weil nur einfaches Erben unterstützt wird.

4.2 Die Programmiersprache Eiffel

Die Programmiersprache Eiffel wurde von Bertrand Meyer entwickelt. Er ist gleichzeitig Gründer der Firma ISE Inc., die die heute am weitesten verbreitete Entwicklungsumgebung für Eiffel entwickelt und vertreibt. In die Sprache flossen, mehr als in andere Sprachen, softwaretechnische Aspekte mit ein. So unterstützt Eiffel durch das Vertragsmodell Korrektheitsüberprüfungen. Durch diese Zusicherungen ist Eiffel begrenzt auch als Entwurfssprache einsetzbar, wodurch zwischen Entwurf und Implementierung kein Wechsel der Notation erforderlich ist. In der ISE-Entwicklungsumgebung ist ein Interpreter integriert, der während der Entwicklung für relativ kurze Turnaround-Zeiten sorgt. Für das fertige Produkt wird aus Gründen der Portabilität C-Code generiert, der dann zusammen mit dem Laufzeitsystem von einem C-Compiler in Maschinensprache übersetzt wird.

Eiffel wird durch das 1997 in der zweiten Auflage erschienene Buch [Mey97] beschrieben. Herr Meyer entwickelt darin neben der Sprache selbst auch eine eigene Philosophie über objektorientierte Programmierung, wie sie seiner Meinung nach aussehen sollte. Diese Philosophie beeinflußt die eigentliche Programmiersprache, so daß einige Konzepte von Eiffel nur im Zusammenhang mit dieser zu verstehen sind. Beispiele dafür sind der Verzicht auf übliche Steuerstrukturen wie `return` oder `break` oder die Beschränkung auf nur ein einziges Schleifenkonstrukt. Die von Herrn Meyer favorisierte Terminologie weicht von der sonst üblichen an vielen Stellen ab.

Eiffel unterstützt Mehrfacherben und bietet wie kaum eine andere Sprache Möglichkeiten, die dabei auftretenden Probleme zu behandeln. Darauf wird im folgenden Abschnitt näher eingegangen.

4.3 Jonglieren mit Vererbungstechniken

Das in Abbildung 4.1 gezeigte Klassendiagramm modelliert ein typisches Unterrichtsbeispiel für Mehrfacherben. Eine abstrakte Oberklasse `Fahrzeug` hat zwei konkrete Unterklassen `Schiff` und `Auto`, die für die Fortbewegung zu Wasser und Land befähigen. Ein Amphibienfahrzeug kann sowohl auf Straßen als auch im Wasser fahren. Diese Klasse erbt daher natürlicherweise sowohl von `Schiff` als auch von `Auto`.

Die Merkmale `Länge` und `Breite` sind bereits in der Basisklasse `Fahrzeug` definiert, da sie für Schiffe und Autos auf die gleiche Art und Weise ermittelt werden können. Die Geschwindigkeit wird jedoch bei beiden Fortbewegungsmitteln unterschiedlich gemessen, weshalb dieses Merkmal in der Oberklasse nur deklariert ist und erst in den jeweiligen abgeleiteten Klassen implementiert wird.

`Amphibienfahrzeug` erbt nun je drei Eigenschaften von seinen Oberklassen. `Länge` und `Breite` sollen dabei nur jeweils einmal in der abgeleiteten Klasse vorkommen. In C++ erfolgt dies durch virtuelles Erben:

```
class Auto : virtual public Fahrzeug
{ ... }

class Schiff : virtual public Fahrzeug
{ ... }

class Amphibienfahrzeug : public Auto, public Schiff
{ ... }
```

Die Entscheidung, daß `Länge` und `Breite` nur einmal geerbt werden, wird also bereits in `Schiff` und `Auto` getroffen, wo sie überhaupt nicht hingehört. Währendn diese beiden Klassen entwickelt werden, ist sie noch nicht akut. In Eiffel sieht die Sprachdefinition vor, daß eine Eigenschaft, die von ein und derselben Basisklasse über mehrere Pfade geerbt und auf keinem der Pfade redefiniert wird, auch nur einmal in der erbenden Klasse vorhanden ist. Die Klassen `Schiff` und `Auto` brauchen beim Einfügen der Klasse `Amphibienfahrzeug` also nicht verändert zu werden.

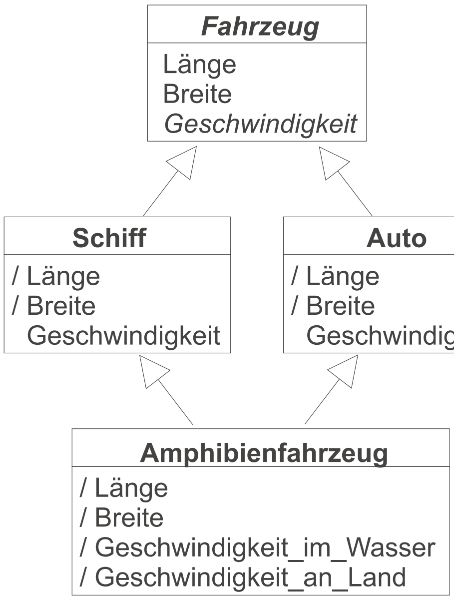

Abbildung 4.1: Ein klassisches Beispiel für Mehrfacherben ist ein Amphibienfahrzeug, das sowohl ein Wasser- als auch ein Landfahrzeug ist. In solchen Szenarien kann Eiffel seine Fähigkeiten voll ausspielen.

Die Eigenschaft `Geschwindigkeit` bedarf jedoch einer anderen Behandlung. Ein Amphibienfahrzeug hat sowohl eine Höchstgeschwindigkeit für die Straße, als auch fürs Wasser und beide sind sicher nicht identisch. In C++ ist die Vermischung beider Verfahren nicht möglich, hier kann nur für alle Eigenschaften zusammen entschieden werden, ob sie virtuell geerbt werden sollen oder nicht. Außerdem muß im Falle des nicht virtuellen Erbens der Benutzer der Klasse `Amphibienfahrzeug` die Basisklasse spezifizieren, deren Eigenschaft angesprochen werden soll, also

```
MethodeEinesKunden( Amphibienfahrzeug* af )
{
   int gw,gl; // Geschwindigkeiten fuer Wasser und Land
   gw = af->Landfahrzeug::Geschwindigkeit;
   gl = af->Wasserfahrzeug::Geschwindigkeit;
   ...
}
```

Die explizite Codierung der Basisklasse setzt beim Benutzer (unnötiges) Wissen über die Vererbungshierarchie von `Amphibienfahrzeug` voraus und macht dadurch Änderungen dieser Hierarchie aufwendiger. In Eiffel kann für jede Eigenschaft getrennt angegeben werden, ob sie einmal oder mehrmals vorhanden sein soll. Die Klasse `Amphibienfahrzeug` sähe in Eiffel etwa wie folgt aus:

```
class Amphibienfahrzeug
inherit
   Auto
      rename geschwindigkeit as land_geschwindigkeit
   end
   Schiff
      rename geschwindigkeit as wasser_geschwindigkeit
      select Auto.land_geschwindigkeit
   end
   ...
end -- class Amphibienfahrzeug
```

Der Entwickler der Klasse kann hier genau festlegen, wie sie sich einem Benutzer präsentieren soll. Der Benutzer muß nur die Schnittstelle der Klasse sehen, mit der er arbeitet und braucht sich um Mehrfacherben kaum Gedanken zu machen. Die `select`-Anweisung ist unbedingt notwendig, da eine Instanz der Klasse `Amphibienfahrzeug` polymorph über eine Referenz vom statischen Typ `Fahrzeug` angesprochen werden kann. In diesem Fall liefert der Aufruf `Fahrzeug.Geschwindigkeit` die Höchstgeschwindigkeit auf Straßen, da auch Amphibienfahrzeuge hauptsächlich an Land fahren.

Dieses kleine Beispiel vermittelt einen Eindruck davon, wie ein Entwickler in Eiffel mit verschiedenen Vererbungstechniken jonglieren kann. Die Vielzahl der angebotenen Möglichkeiten können jedoch dazu verführen, Konstrukte zu benutzen, weil sie vorhanden sind, und nicht, weil sie in einer bestimmten Situation sinnvoll sind. Dadurch können schwer durchschaubare Klassenhierarchien entstehen. Auch für Eiffel gilt: weniger ist oft mehr. Weitere Beispiele zum Thema Mehrfacherben finden sich in [Str97] und [Mey97]. Folgende Aussage meinerseits spiegelt die Unterschiede zwischen den beiden Sprachen wieder:

> *Wenn C++ das Spielen im Sandkasten der Objektorientierung ermöglicht, so bietet Eiffel einen ganzen Abenteuerspielplatz.*

Diese Aussage entstammt absichtlich der Kinderwelt. Sie soll keine Aussage darüber sein, wie gut oder wie schlecht sich eine der genannten Sprachen für die Entwicklung realer, großer, kommerzieller Software-Systeme eignet. In [You93] wird ausführlich dargelegt, daß man mit einer genialen Programmiersprache allein ein Projekt nur schneller in den Sand setzt als ohne.

4.4 Grundsätze für die Spracherweiterung

E+ soll die vielen Möglichkeiten, die Eiffel in Verbindung mit Vererbungsbeziehungen bietet (siehe Abschnitt 4.3), auf Delegationsbeziehungen übertragen. Es wird sich zeigen, daß einige Konstrukte unverändert übernommen werden können, während andere in dem neuen Kontext eine erheblich andere Semantik haben. Außerdem wird es zu Seiteneffekten kommen, die Delegationsbeziehungen auf Vererbungsbeziehungen haben werden.

Da mit Delegationsbeziehungen objektbasierte Vererbung realisiert werden soll, wird Delegation, wenn möglich, so wie Vererbung behandelt. Es wird sich zeigen, daß an einigen Stellen die genaue Art der Sub-/Superklassen-Beziehung (also Vererbungs- oder Delegationsbeziehung)nicht von Interesse ist.

Bei einigen Erweiterungen kann man sicherlich über den Sinn und Zweck im praktischen Einsatz streiten. Bei einigen Konstrukten ist die Frage "Wird das jemals jemand brauchen?" durchaus berechtigt. Eine Diskussion über den praktischen Nutzen wird trotzdem nur an wenigen Stellen kurz angerissen. Diese Diplomarbeit soll zeigen, was möglich ist, nicht was sinnvoll ist. Fundierte Aussagen über den praktischen Nutzen eines Konstrukts sind nur nach einem empirischen Test von E+ möglich. Außerdem wird in [SU95] betont, daß die Einführung neuer Sprachkonstrukte nicht durch einzelne praktische

Beispiele motiviert sein sollte, da sich dies nach einiger Zeit meist als Fehler herausstellt. Im Evaluierungsteil wird später die Anwendbarkeit der neuen Sprache analytisch unter die Lupe genommen (sihe Kapitel 9).

Weiterhin bleiben Implementierungsaspekte unberücksichtigt. Die Definition der neuen Sprache wird unabhängig davon erfolgen, wie leicht oder wie schwer sich ein Compiler und eine Laufzeitumgebung realisieren lassen. Auch Effizienzgesichtspunkte bleiben außen vor.

Bisher wurde für die Referenz, die in jedem Objekt auf das Objekt selbst verweist, mit *self* bezeichnet. Diese Bezeichnung ist allgemein üblich. In Eiffel wird sie jedoch `Current` genannt, weshalb im folgenden auch dieser Begriff benutzt wird.

Kapitel 5

Delegationsbeziehungen und Delegationsreferenzen

In diesem Kapitel wird Eiffel um einige grundlegende Sprachkonstrukte zur Unterstützung von Delegation erweitert. Dabei wird die Wechselwirkung der Erweiterung mit bestehenden Konstrukten und Eigenschaften der Sprache beschrieben. An einigen Stellen werden mögliche Alternativen diskutiert.

5.1 Deklaration von Delegationsbeziehungen

Um überhaupt mit Delegation arbeiten zu können, muß zunächst eine Delegationsbeziehung zwischen zwei Klassen hergestellt werden können. Dazu erhält die delegierende Klasse neben dem `inherit`-Teil einen `delegate`-Teil. Dort werden Delegationsbeziehungen (eine Klasse kann mehrere Elternklassen haben, analog zur Mehrfacherbung) deklariert und mögliche Modifikationen vorgenommen. Da zu einer Delegationsbeziehung immer ein bestimmtes Objekt gehört, muß eine Referenz angegeben werden, die auf das delegierte Objekt verweist. Die Deklaration dieser Referenz erfolgt, wie die anderer Merkmale der Klasse, auch im `feature`-Teil.

```
class A
...
feature{ ANY }
   merkmal_1 is ...
   ...
end -- class A
```

```
class B
delegate A to del_ref
...
feature{ NONE }
   del_ref : A
   ...
end -- class B
```

Die Klasse `B` delegiert also an die Klasse `A`. Wenn für eine Instanz von `B` das Merkmal `merkmal_1` aufgerufen wird, wählt das Laufzeitsystem mittels `del_ref` eine Instanz von `A` als Kindobjekt aus und ruft deren Merkmal `merkmal_1` auf. Das `del_ref` für einen Kunden nicht zugreifbar ist (exportiert an `NONE`) spielt für die Delegation keine Rolle. Die Benutzung von `B` durch einen Kunden könnte etwa wie folgt aussehen:

```
class KUNDE
feature
   obj_b : B

   benutze_B is
   do
      obj_b.merkmal_1
   end
   ...
end -- class KUNDE
```

Es stellt sich die Frage, warum ein Kunde wissen muß, welches Merkmal (in diesem Fall `del_ref`) auf das Kindobjekt verweist. Diese Frage ist zum jetzigen Zeitpunkt durchaus berechtigt. Es handelt sich dabei um ein Implementierungdetail von `B`, das einen Kunden nicht interessieren sollte. In Abschnitt 7.2 werden jedoch Erweiterungen eingeführt, die es sinnvoll erscheinen lassen, wenn ein Kunde weiß, welches Merkmal das Kindobjekt identifiziert.

5.1.1 Diskussion

In [Cos98] werden die Delegationsbeziehung und die Delegationsreferenz in einem Schritt deklariert. Dadurch wird dem Leser eines Quelltexts das Springen zwischen der einen und der anderen Deklaration abgenommen. Die hier für Eiffel vorgeschlagene Zweiteilung stellt für den Leser einen gewissen Umstand und daher einen Nachteil dar. Allerdings folgt die Erweiterung damit dem Charakter der Sprache. So muß die Absicht, ein von einer Oberklasse geerbtes Merkmal zu redefinieren, im `inherit`-Teil der Unterklasse angegeben werden, während die eigentliche Redefinition im `feature`-Teil erfolgt. Daher wurde für Delegationsbeziehungen ebenfalls eine Zweiteilung bei der Deklaration gewählt. Die Zweiteilung ist zudem flexibler. Die Delegationsreferenz könnte z.B. auch von einer Oberklasse geerbt werden. Ein durchdachtes Entwicklungswerkzeug könnte den Entwickler hier gut unterstützen.

5.1.2 Delegation über parameterlose Funktionen

Eine interessante Eigenschaft von Eiffel kann durch die Zweiteilung der Deklaration die Flexibilität weiter erhöhen: Die Syntax eines Merkmalsaufrufs in Eiffel unterscheidet nicht zwischen Attributen und parameterlosen Funktionen. Im `delegate`-Teil einer Klasse wird das Merkmal, das die Verbindung zum Elternobjekt herstellt, nur über seinen Namen angesprochen. Dies bedeutet, daß das Elternobjekt auch von einer Funktion berechnet werden kann. Das Laufzeit-System ruft eine Methode auf und benutzt deren Ergebnis, statt eine Referenz auszuwerten.

```
class A                 class B
   ...                  delegate A to del_ref
end -- class A          feature{ NONE }
                           del_ref is
                           do
                              -- Berechne, an welche Instanz von A jetzt
                              -- delegiert werden soll
                              Result := ergebnis_der_berechnung
                           end
                        end -- class B
```

Wenn sich das Elternobjekt häufig ändert, muß eine Referenz oft korrigiert werden. Dies ist vielleicht aufwendiger, als wenn die Ermittlung des Elternobjekts bis zur tatsächlichen Delegation aufgeschoben wird. Je seltener tatsächliche Delegationen gegenüber Änderungen des Elternobjekts sind, um so eher lohnt sich eine parameterlose Funktion zu seiner Bestimmung.

Im folgenden wird der Begriff Delegationsreferenz sowohl für echte Attribut-Referenzen, als auch für Funktionen benutzt. Nur wenn die Unterscheidung zwischen beiden von Bedeutung ist, werden die jeweiligen Begriffe benutzt.

5.1.3 Exportstatus von Delegationsreferenzen

Eiffel verbietet Kunden grundsätzlich den schreibenden Zugriff auf die Attribute einer Klasse.[1] Der Exportstatus eines Attributs bestimmt daher nur, wer das Attribut *lesen* darf. Somit ist es einem Kunden grundsätzlich nicht möglich, die Delegationsreferenz eines Objekts zu verändern. Stattdessen muß in der Kindklasse die Funktionalität für den Austausch des Elternobjekts vorhanden sein.

Der Exportstatus einer Delegationsreferenz kann daher problemlos wie bei einem gewöhnlichen Attribut gehandhabt werden (Delegationsreferenzen sind im Prinzip ja gewöhnliche Attribute). Näheres zu den Möglichkeiten, die Eiffel bei der Exportkontrolle bietet, werden im Zusammenhang mit dem Zugriff auf delegierte Merkmale in Abschnitt 5.2 erläutert.

5.1.4 Objekterzeugung

Eine Delegationsbeziehung wird zwischen Klassen deklariert. Um zur Laufzeit tatsächlich delegieren zu können, muß jedoch ein Elternobjekt vorhanden sein, das die delegierten Merkmalsaufrufe bearbeiten kann. Bisher wurde nicht darauf eingegangen, woher ein Kindobjekt weiß, an welches Elternobjekt es delegieren soll.

Eine automatische Lösung dieses Problems scheidet aus. Das Laufzeitsystem kann bei der Erzeugung eines Kindobjekts durch einen Kunden nicht automatisch das Elternobjekt erzeugen. Dafür gibt es eine Reihe von Gründen.

- Zum einen erfüllt eine beliebige Instanz der Elternklasse im allgemeinen sicher nicht die an ein spezielles Elternobjekt gestellten Anforderungen.
- Weiterhin werden in Eiffel Objekte grundsätzlich nicht automatisch erzeugt (siehe [Mey97, Kapitel 8.3]). Die Begründung ist: Klassen können sich gegenseitig benutzen. Eine Klasse `A` kann z.B. ein Attribut vom Typ `B` haben, `B` wiederum ein Attribut vom Typ `A`. Eine automatische Objekterzeugung führt dann zu einer Endlosschleife.
- Schließlich kann die Elternklasse eine *Initialisierungsmethode*[2] haben. Diese speziellen Methoden - außerhalb der Eiffel-Welt oft als *Konstruktoren* bezeichnet - werden bei der Erzeugung eines neuen Objekts ausgeführt (siehe folgendes Quelltext-Beispiel) und haben die Aufgabe, das Objekt korrekt zu initialisieren. Ist für eine Klasse eine Initialisierungsmethode definiert, wird ihre Instanziierung ohne den Aufruf dieser Methode vom Compiler mit einer Fehlermeldung abgewiesen.

Um für ein Elternobjekt sorgen zu können, muß daher die Kindklasse eine Initialisierungsmethode haben. Dies ist der einzige Weg, die Verfügbarkeit eines passenden Elternobjekts sicher zu stellen. Eine Ausnahme bilden *optionale Delegationsbeziehungen*, die in Abschnitt 5.5.2 behandelt werden.

Das folgende Beispiel zeigt eine Kindklasse und ihre Instanziierung durch einen Kunden, der dabei die Initialisierungsmethode der Kindklasse aufruft:

[1] Bei Redefinition tritt im Zusammenhang mit Delegation eine Verletzung dieser Regel auf, siehe Abschnitt 6.2.5

[2] Die Übersetzung des englischen Begriffs *creation-procedure* mit *Initialisierungsmethode* wurde einer wörtlichen Übersetzung vorgezogen, da diese Methode erst nach der eigentlichen Erzeugung eines neuen Objekts vom Laufzeitsystem aufgerufen wird. Mit der Erzeugung im engeren Sinne hat sie daher nichts zu tun.

```
class B
delegate A to del_ref;
creation initialisierungsmethode
feature{ ANY }
   initialisierungsmethode is
   do
      -- Operationen, die del_ref
      -- auf ein passendes Objekt
      -- verweisen lassen
   end
   ...
end -- class B
```

```
class KUNDE
feature
   instanziiere_B is
   local
      obj_b : B
   do
   !!obj_b.initialisierungsmethode
      ...
   end
   ...
end -- class KUNDE
```

Eine Initialisierungsmethode kann, wie in [Mey97, Kapitel 8.4] beschrieben, formale Parameter haben. Im weiteren Verlauf der Arbeit wird die Initialisierungsmethode in Beispiel-Quelltexten nicht mehr explizit angegeben.

5.2 Zugriff auf geerbte Merkmale durch Kindklassen

Beziehungen zwischen Klassen sind in Eiffel entweder *offen* oder *geschlossen*. In diesem Abschnitt wird das *offen/geschlossen-Prinzip* (*open/closed-Prinzip*) kurz vorgestellt und anschließend festgelegt, in welche Kategorie Delegationsbeziehungen fallen.

5.2.1 Das offen/geschlossen-Prinzip von Eiffel

In Eiffel wird beim Aufruf eines Merkmals grundsätzlich zwischen zwei Situationen unterschieden:

1. Der Zugriff kann durch eine andere Methode der selben Klasse oder einer ihrer Ober- oder Unterklassen erfolgen.
2. Ein Kunde kann mittels einer Referenz auf ein Objekt der Klasse ein Merkmal aufrufen.

Eiffel erlaubt Aufrufe der ersten Art ohne Einschränkungen. Anders als z.B. in C++ oder Java, wo Merkmale einer Klasse als `private` deklariert werden können, kann eine Klasse in Eiffel vor ihren Unterklassen nichts verbergen. Ein Entwickler einer Klasse muß sich deshalb sehr gut in der Vererbungshierarchie auskennen, in die er seine neue Klasse einzuordnen beabsichtigt. Beide Aufrufe in der Methode `OBERKLASSE.methode_3` sind daher erlaubt:

```
class OBERKLASSE
feature{ANY} -- exportiert
   merkmal_1 ...

feature{NONE} -- nicht exportiert
   merkmal_2 ...
end -- class OBERKLASSE
```

```
class UNTERKLASSE
inherit OBERKLASSE
feature
   methode_3 is
   do
      merkmal_1 ... -- erlaubt
      merkmal_2 ... -- erlaubt
   end
end -- class UNTERKLASSE
```

Merkmalsaufrufe durch Kunden können hingegen beschränkt werden. Ein Kunde kann nur auf Merkmale zugreifen, die die Klasse für ihn sichtbar gemacht hat. Der schreibende Zugriff auf Attribute ist für Kunden gänzlich verboten. Dadurch wird die in Kapitel 5.1.2 näher beschriebene Transparenz zwischen parameterlosen Funktionen und Attributen möglich. Außerdem wird so die Kapselung des inneren Zustands eines Objekts garantiert, da dieser von außen nicht geändert werden kann. Das folgende Beispiel benutzt wieder die Klasse `OBERKLASSE` aus dem vorherigen Beispiel:

```
class KUNDE
feature
   methode_1 is
   local
      obj : OBERKLASSE
   do
      !!obj;
      obj.merkmal_1; -- erlaubt
      obj.merkmal_2; -- nicht erlaubt
   end
end -- class KUNDE
```

Es gilt also stets: eine Vererbungsbeziehung ist *offen*, eine Benutzbeziehung ist *geschlossen*.

5.2.2 Delegationsbeziehungen sind geschlossen

Es stellt sich nun die Frage, in welche Kategorie Delegationsbeziehungen fallen, genauer: ist der Aufruf eines Merkmals der Elternklasse durch eine Kindklasse offen oder geschlossen? Sprachen wie SELF (siehe Abschnitt 3.2) realisieren mittels Delegation die Vererbungsbeziehung. Von diesem Standpunkt aus betrachtet ist Delegation *offen*. Auf der anderen Seite ist die Kapselung eines der fundamentalen Konzepte objektorientierter Softwareentwicklung. Ein Objekt kapselt einen inneren Zustand und stellt Kunden eine Schnittstelle zur Verfügung, um mit dem Objekt arbeiten und den Zustand verändern zu können. Bei der Vererbungsbeziehung handelt es sich um eine Beziehung zwischen Klassen. Bei der Instanziierung wird die Vererbungshierarchie aufgelöst, das Ergebnis ist ein einzelnes Objekt, das die Merkmale aller direkten und indirekten Klassen enthält, deren Instanz es ist. Dadurch, daß die Vererbungsbeziehung *offen* ist, wird auf Objektebene also die Kapselung nicht verletzt. Bei einer Delegationsbeziehung, d.h. Vererbung zwischen Objekten, existieren zwei konkrete Objekte, die beide für sich einen Zustand kapseln.

Aus diesem Grund wird Delegation als *geschlossen* betrachtet. Eine Kindklasse darf nur die Merkmale der Elternklasse aufrufen, die es auch in einer Benutzbeziehung aufrufen dürfte.

```
class ELTERNKLASSE
feature{ ANY }
   merkmal_1 ...
feature{ NONE }
   merkmal_2 ...
end -- class ELTERNKLASSE
```

```
class KINDKLASSE
delegate ELTERNKLASSE to del_ref
feature
   del_ref : ELTERNKLASSE;

   methode_1 is
   do
      del_ref.merkmal_1 ... -- 1 erlaubt
      del_ref.merkmal_2 ... -- 2 Fehler
      merkmal_1             -- 3 erlaubt
      merkmal_2             -- 4 Fehler
   end
end -- class KINDKLASSE
```

Die Aufrufe 1 und 2 erfolgen wie gewöhnliche Aufrufe eines Kunden. Statt `del_ref` könnte auch eine andere Referenz auf das Objekt benutzt werden. Die Aufrufe 3 und 4 haben Delegationssemantik, d.h. `Current` bleibt auf dem Kindobjekt stehen.

Durch geschlossene Delegationsbeziehungen ergeben sich Nachteile im Bezug auf die Flexibilität bei der Redefinition von Merkmalen einer Elternklasse (siehe Abschnitt 6.2). Der Entwickler einer Klasse kann vielleicht nicht voraussehen, in welcher Form später an diese Klasse delegiert wird. Ein späterer Benutzer kann mangels Zugriff auf bestimmte Merkmale die Klasse nicht optimal an seine Anforderungen anpassen. Durch die Möglichkeit zum selektiven Export (siehe Abschnitt 5.3.1) können diese

Nachteile jedoch gemildert, wenn auch nicht ganz behoben werden. Sollte es gar nicht anders gehen, kann für die Delegationsbeziehung eine neue Unterklasse der Elternklasse gebildet werden, die der Kindklasse die benötigten Eigenschaften zur Verfügung stellt. Letzlich überwiegt der Vorteil der Kapselung des inneren Zustands. In [Cos98] wird Delegation als offene Beziehung behandelt, d.h. die Sichtweise der Vererbung auf Objektebene erhält den Vorzug vor der der Kapselung eines Zustandes. Mangels praktischer Erfahrung kann an dieser Stelle nicht bewertet werden, welche Lösung die bessere ist.

5.3 Zugriff auf geerbte Merkmale durch Kunden

Nachdem im vorangegangenen Abschnitt die Zugriffsmöglichkeiten von Kindklassen auf die Merkmale ihrer Elternklassen festgelegt wurden, geht es in diesem Abschnitt um Benutzbeziehungen. Es wird untersucht, wie ein Kunde auf Merkmale einer Kindklasse zugreifen darf, wenn diese aus einer Delegationsbeziehung von einer Elternklasse geerbt werden.

5.3.1 Selektiver Export von Merkmalen

Eines der Konzepte, die Eiffel vielen anderen Sprachen voraus hat, ist der *selektive Export* von Merkmalen. Statt ein Merkmal entweder an alle Kunden oder an keinen zu exportieren, kann der Exportstatus feiner eingestellt werden. Im folgenden Beispiel exportiert die Klasse `A` das Merkmal `merkmal_1` an die Klasse`B`, indem es `B` in die Exportliste des `feature`-Abschnitts aufnimmt, in dem `merkmal_1` deklariert wird. Gleichzeitig mit `B` erhalten auch alle Unterklassen von `B` Zugriff auf das Merkmal.

```
class A
feature{ B }
   merkmal_1 ...
...
end -- class A
```

Eine Klasse, die keine explizit deklarierte Oberklasse hat, erbt implizit von der abstrakten Klasse `ANY`. Durch diesen Trick gibt es in jedem Eiffel-System stets genau eine Klasse — eben `ANY` — zu der alle Klassen konform sind. `ANY` selbst ist leer und erbt von `GENERAL`. Dort werden einige allgemeine Merkmale implementiert, z.B. `copy`, `clone` oder Standardein- und -ausgabe. Für eine genaue Beschreibung sei auf [Mey92, Kapitel 16.2] verwiesen. Soll ein Merkmal an alle Klassen exportiert werden, dies entspricht `public` in C++ oder Java, so muß es an `ANY` exportiert werden.

Der selektive Export stellt eine große Hilfe für den Entwurf von Subsystemen dar, die in Eiffel, wie in vielen anderen objektorientierten Sprachen, kein eigenständiges syntaktisches Konstrukt sind. Die Funktionalität eines Subsystems wird durch eine oder mehrere Klassen modelliert. Die Kunden des Subsystems können nur auf die an `ANY` exportierten Merkmale zugreifen. Die Klassen des Subsystems können sich untereinander weitreichendere Zugriffsmöglichkeiten einräumen, ohne vollständig auf Kapselung verzichten zu müssen.

5.3.2 Der Exportstatus delegierter Merkmale

Wenn eine Kindklasse an eine Elternklasse delegiert, kann sie in der Regel nicht auf alle Merkmale der Elternklasse zugreifen, da Delegation eine *geschlossene* Beziehung ist. Die Delegationsbeziehung bewirkt gleichzeitig, daß die Schnittstelle der Elternklasse auch für Kunden der Kindklasse zur Verfügung steht. Zugriffe von Kunden auf delegierte Merkmale unterliegen den Exportbestimmungen der Elternklasse. Dies hat zur Folge, daß ein Kunde `KUNDE` einer Kindklasse `KINDKLASSE` möglicherweise auf Merkmale der Elternklasse `ELTERNKLASSE` zugreifen kann, auf die `KINDKLASSE` nicht zugreifen darf. Das folgende Beispiel verdeutlicht diese Situation:

```
class ELTERNKLASSE
feature{ KUNDE }
   merkmal_1 ...
   ...
end -- class ELTERNKLASSE


class KUNDE
feature
   obj : KIND_KLASSE

   methode_1 is
   do
      obj.merkmal_1
      -- OK, merkmal_1 wird von
      -- ELTERNKLASSE an KUNDE
      -- exportiert
   end
end -- class KUNDE
```

```
class KINDKLASSE
delegate ELTERNKLASSE to del_ref
feature
   methode_1 is
   do
      -- Aufruf als Kunde
      del_ref.merkmal_1

      -- Aufruf mit Delegationssemantik
      merkmal_1

      -- beide Aufrufe sind falsch, da
      -- merkmal_1 nicht an KINDKLASSE
      -- exportiert wird
   end
   ...
end -- class KINDKLASSE
```

5.3.3 Änderungen am Exportstatus von delegierten Merkmalen

Da Delegation *geschlossen* ist, sollten die Exportbeschränkungen, die eine Elternklasse vornimmt, durch eine Kindklasse nicht ausgehebelt werden können. Ein aufgrund einer Delegationsbeziehung in die Schnittstelle einer Kindklasse eingeblendetes Merkmal wird daher, wie in Abschnitt 5.3.2 gezeigt, von der Kindklasse an genau die selben Kunden exportiert wie von der Elternklasse.

In einer Vererbungsbeziehung hat die Unterklasse uneingeschränkten Zugriff auf die Merkmale der Oberklasse (Vererbung ist *offen*). Daher ist es ihr auch möglich, den Exportstatus geerbter Merkmale beliebig zu verändern. Dies erfolgt im `inherit`-Teil der Unterklasse:

```
class OBERKLASSE
feature{ NONE }
   merkmal_1 ...
...
end -- class OBERKLASSE

class UNTERKLASSE
inherit OBERKLASSE
   export{ ANY } merkmal_1 end
...
end -- class UNTERKLASSE
```

```
class KUNDE
feature
   obj_o : OBERKLASSE;
   obj_u : UNTERKLASSE;

   methode_1 is
   do
      obj_o.merkmal_1 -- nicht erlaubt
      obj_u.merkmal_1 -- erlaubt
   end
end -- class KUNDE
```

Für Delegationsbeziehungen kann diese Regelung nicht angewendet werden. Da Delegation eine *geschlossene* Beziehung ist, würde eine Lockerung von Exportbeschränkungen die Kapselung des Elternobjekts verletzen. Daher darf eine Kindklasse den Exportstatus für Merkmale der Elternklasse nur einschränken, nicht jedoch erweitern.

Das folgende Beispiel zeigt die Klasse `ELTERNKLASSE`, die ein Merkmal `merkmal_1` an die Klasse `C1` exportiert. Die Klasse `KINDKLASSE`, eine Kindklasse von `ELTERNKLASSE`, darf das geerbte Merkmal `merkmal_1` nur an `C1`, in diesem Fall erfolgt keine Änderung und die Anweisung ist überflüssig, oder eine Subklasse von `C1` exportieren. Der Export an eine andere Klasse stellt einen Fehler dar.

```
class C ... end
class C1 ... end

class C2
   inherit C1
   ...
end -- class C2

class ELTERNKLASSE
feature{ C1 }
   merkmal_1 ...
   ...
end -- class ELTERNKLASSE

class KINDKLASSE
delegate ELTERNKLASSE to del_ref
   export{ C1 } merkmal_1 -- erlaubt aber ueberfluessig
   export{ C2 } merkmal_1 -- erlaubt
   export{ C  } merkmal_1 -- nicht erlaubt
end
...
end -- class D1
```

5.3.4 Typfehler zur Laufzeit

Eiffel ist eine statisch getypte Programmiersprache. Dies bedeutet, daß bereits zur Übersetzungszeit festgestellt werden kann, ob ein Merkmalsaufruf `obj.merkmal( arg )` erlaubt ist. Ein Compiler kann prüfen, ob alle Objekte, auf die `obj` zur Laufzeit verweisen kann, über ein Merkmal `merkmal` mit den formalen Parametern `arg` verfügen. Allerdings bringt die Flexibilität, die Eiffel beim Umgang mit Vererbung bietet, zwei Konstrukte mit sich, die auch zur Laufzeit Typfehler verursachen können. Eines der Konstrukte, die *covariante Redefinition*, wird in Abschnitt 6.2.6 behandelt. Das andere ist die soeben beschriebene Einschränkung des Exportstatus. Das folgende Beispiel konstruiert einen solchen Typfehler:

```
class ELTERNKLASSE
feature{ ANY }
   merkmal_1 ...
   ...
end -- class ELTERNKLASSE

class KINDKLASSE
delegate ELTERNKLASSE to del_ref
   export{ NONE } merkmal_1
end
...
end -- class KINDKLASSE

class KUNDE
feature
   obj_e : ELTERNKLASSE;
   obj_k : KINDKLASSE;

   verursache_typfehler is
   do
      obj_e = obj_k;  -- Zuweisung erlaubt,
                      -- KINDKLASSE delegiert an ELTERNKLASSE
      obj_e.merkmal_1 -- Typfehler, merkmal_1 wird von KINDKLASSE
                      -- nicht exportiert und obj_e verweist auf
                      -- eine Instanz von KINDKLASSE
   end
end -- class KUNDE
```

Diese Fehlerquelle tritt bei Vererbungsbeziehungen ebenso auf wie bei Delegationsbeziehungen, das Problem ist also bekannt und wird in [Mey97] ausführlich behandelt. Dort wird unter anderem angeführt, daß es in realen Systemen nur äußerst selten auftritt, weil die Möglichkeit zur Exportbeschränkung in Verbindung mit Polymorphie kaum genutzt wird. Trotzdem ist das Problem vorhanden und ein Entwickler muß sich dessen bewußt sein.

Abbildung 5.1: Delegationsbeziehungen werden vererbt. Subklassen delegieeren an die selben Klassen wie ihre Superklassen.

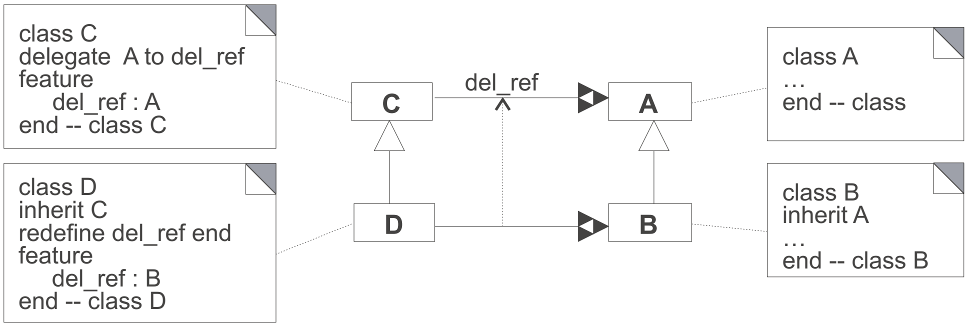

Abbildung 5.2: Eine Delegationsreferenz kann in Subklassen covariant redefiniert werden.

5.4 Vererben von Delegationsreferenzen

Delegationsreferenz en sind im Prinzip ganz normale Merkmale einer Klasse. Sie werden daher auch wie solche vererbt. Die Delegationsbeziehung wird ebenfalls vererbt. Wenn eine Klasse `KINDKLASSE` an eine Klasse `ELTERNKLASSE` delegiert, so delegieren auch Subklassen von `KINDKLASSE` an `ELTERNKLASSE`. Wenn eine Klasse `KINDKLASSE2` an `KINDKLASSE` delegiert, so delegiert sie — indirekt — ebenfalls an `ELTERNKLASSE` (siehe Abbildung 5.1). Es gibt diesbezüglich also keinen prinzipiellen Unterschied zwischen Vererbung und Delegation.

5.4.1 Redefinition von Delegationsreferenzen

Wie alle Merkmale einer Klasse kann auch eine Delegationsreferenz *covariant* redefiniert werden. Covariant bedeutet, daß der neue Typ der Referenz konform zum alten sein muß. Für Kunden der Superklasse ergeben sich dadurch keine Unterschiede bei der polymorphen Benutzung von Subklassen. Abbildung 5.2 stellt ein solches Szenario dar.

Zu Problemen kann es kommen, wenn eine Methode der Klasse `C` der Delegationsreferenz ein Objekt vom Typ `A` zuweist. Dies kann zu Laufzeit-Typfehlern führen (siehe Abschnitt 6.2.6). Da Vererbung eine offene Beziehung ist, kann und muß sich der Entwickler der Klasse `D` genau darüber informieren, welche Methoden in `C` zu solchen Problemen führen können. Noch schwieriger wird es, wenn zwischen `C` und `D` eine Delegationsbeziehung bestehen würde. Dann könnten in `C` Methoden enthalten sein, die der Delegationsreferenz ein Objekt vom Typ `A` zuweisen, auf die der Entwickler der Klasse `D` jedoch keinen Zugriff hat (siehe Abschnitt 5.3.2). Dieses Problem läßt sich mit Sprachmitteln nicht lösen. Hier ist ein sorgfältiger und möglichst einfacher Entwurf der einzige Schutz.

5.4.2 Trennen von Delegationsbeziehungen in Subklassen

Das Trennen einer Delegationsbeziehung bedeutet, daß eine Subklasse eine geerbte Delegationsbeziehung nicht mehr nutzen möchte. Eine Klasse `KONTEXT` benötigt zur Ausführung ihrer Aufgaben einen Algorithmus, der jedoch zur Laufzeit änderbar sein muß (siehe Abbildung 5.3). Die Realisierung erfolgt durch Umsetzung des Entwurfsmuster Strategie [GJHV95, Seite 315 ff.]. `STRATEGIE` ist die Basisklasse für die in Klassen gekapselten Algorithmen. `SPEZIELLER_KONTEXT` stellt eine Spezialisierung

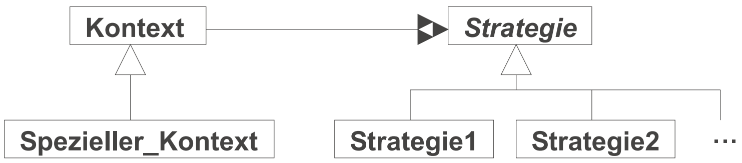

Abbildung 5.3: In bestimmten Situationen kann es notwendig sein, eine geerbte Delegationsbeziehung aufzulösen. Dazu bietet E+ die `undelegate`-Anweisung an.

von `KONTEXT` dar, in der der Algorithmus nicht variabel sein muß. Daher soll `SPEZIELLER_KONTEXT` den benötigten Algorithmus selbst implementieren, womit die Delegationsbeziehung überflüssig wird.

Eigentlich ist zur Lösung einer Delegationsbeziehung kein besonderes Konstrukt erforderlich. Eine Subklasse, die die Delegationsbeziehung ihrer Superklasse lösen will, braucht nur alle Merkmale der Elternklasse zu redefinieren, womit die Delegationsbeziehung zwar weiter besteht, faktisch jedoch nicht mehr zum Einsatz kommt (siehe dazu Abschnitt 6.2). Die Philosopie von Eiffel verlangt jedoch wann immer es geht die explizite Formulierung von Absichten, auch wenn, oder gerade weil Redundanz entsteht. Zu leicht könnte also ein Entwickler in obigem Beispiel die Redefinition eines Merkmals vergessen, das z.B. in einer Oberklasse von `STRATEGIE` deklariert ist (in dem Beispiel in Abbildung 5.3 existiert eine solche Oberklasse nicht, im allgemeinen kann `STRATEGIE` jedoch eine Oberklasse haben). Daher muß die Auflösung der Delegationsbeziehung explizit angegeben werden:

```
class SPEZIELLER_KONTEXT
inherit KONTEXT
   undelegate STRATEGIE to del_ref
   redefine -- alle Merkmale von STRATEGIE,
            -- die bisher noch nicht redefiniert wurden
   end
...
end -- class SPEZIELLER_KONTEXT
```

Durch die Redefinition aller Merkmale bleibt die Typkonformität erhalten. `SPEZIELLER_KONTEXT` kann nach wie vor über eine Referenz vom statischen Typ `STRATEGIE` angesprochen werden, da alle für `STRATEGIE` erlaubten Aufrufe auch für `SPEZIELLER_KONTEXT` erlaubt sind. Wenn in der Klasse `SPEZIELLER_KONTEXT` keine `undelegate`-Anweisung steht, jedoch trotzdem alle Merkmale von `STRATEGIE` redefiniert werden, muß ein Compiler eine Fehlermeldung ausgeben. Die Delegationsbeziehung würde für die Klasse `SPEZIELLER_KONTEXT` und alle ihre Subklassen unwiderruflich gelöst. Der Entwickler muß dies explizit angeben.

Es ist fraglich, ob es Situationen gibt, in denen eine Delegationsbeziehung tatsächliche getrennt werden sollte. Die Trennung wird jedoch nicht erst durch die `undelegate`-Anweisung möglich. Durch Redefinition aller Merkmale der Elternklasse ist eine Trennung ohne weiteres möglich, ebenso wie auf die gleiche Weise eine Vererbungsbeziehung gelöst werden kann. Die `undelegate`-Anweisung ist daher Redundant und dient nur der expliziten Formulierung von Absichten. Abschließend muß noch erwähnt werden, daß eine Delegationsbeziehung nur dann getrennt werden kann, wenn die Elternklasse der Kindklasse genügent Zugriffsrechte einräumt, damit diese alle Merkmale redefinieren kann.

5.5 Sichere und unsichere Delegationsbeziehungen

Die Identifizierung eines Elternobjekts erfolgt über eine Referenz. Referenzen können jedoch `Void` sein, d.h. sie verweisen auf kein Objekt. Ist eine Delegationsreferenz `Void`, kann keine Delegation von Merkmalsaufrufen erfolgen. Für den Kunden eines Objekts ist es jedoch wichtig zu wissen, wann er auf Merkmale eines Elternobjekts zugreifen kann und wann nicht. Diese Problematik soll im folgenden genauer untersucht werden.

5.5.1 Ungültige Delegationsreferenzen

Bisher wurden die in der Praxis störenden, gerade deshalb aber wichtigen `Void`-Referenzen ignoriert. In Eiffel sind alle Referenzen ungültig, bis sie explizit initialisiert werden.

Wenn der Kunde eines Kindobjekts ein Merkmal aufruft, muß dieser Aufruf an ein Elternobjekt delegiert werden, falls das Kindobjekt nicht selbst ein passendes Merkmal hat. Das Elternobjekt wird durch eine Delegationsreferenz identifiziert, die ein Merkmal der Kindklasse ist. Wenn diese Referenz zur Zeit des Aufrufs `Void` ist, kann keine Delegation erfolgen, es kommt zu einem Laufzeitfehler. Dieser Fehler ist vom Charakter her ein Typfehler, denn es wurde auf ein Merkmal eines Objekts (des Kindobjekts) zugegriffen, das in diesem nicht enthalten ist. Die Delegation und damit der Umstand, daß das Merkmal auch im Falle einer gültigen Delegationsreferenz nicht im Kindobjekt selbst liegt, bleibt vor dem Kunden verborgen.

Im allgemeinen kann durch statische Analyse des Quelltextes nicht ermittelt werden, ob eine Referenz zu einem bestimmten Zeitpunkt `Void` ist. Ein Compiler der dazu in der Lage wäre, könnte die folgende `if`-Abfrage wegoptimieren und dadurch das Halteproblem [Weg93] lösen:

```
if del_ref = Void then exit_program end
```

Zwar kann ein guter Compiler in vielen Fällen eine Warnung, bzw. Fehlermeldung ausgeben, trotzdem wird eine konzeptionell saubere Lösung benötigt. Dafür bietet sich das Vertragsmodell von Eiffel an, weshalb die Lösung dieses Problems in Abschnitt 7.2 beschrieben wird.

5.5.2 Garantierte und optionale Delegation

Wenn eine Delegationsreferenz `Void` ist, stellt dies in vielen Fällen keinen Fehler dar. In Abschnitt 2.1 wird im Zusammenhang mit rollenbasierter Modellierung absichtlich mit ungültigen Delegationsreferenzen gearbeitet. Dies bedeutet nämlich, daß eine bestimmte Rolle zur Zeit nicht eingenommen werden kann. Für einen Kunden muß jedoch feststellbar sein, ob eine Delegationsreferenz `Void` werden kann und wenn ja, ob zu einem bestimmten Zeitpunkt ein bestimmter Merkmalsaufruf gültig ist. Bei allen bisherigen Betrachtungen wurde immer davon ausgegangen, daß Delegationsreferenzen auf gültige Objekte verweisen. Diese stillschweigende Annahme soll ab jetzt zum Gesetz werden.

```
class B
delegate A to del_ref;
...
end -- class B
```

In diesem Beispiel kann ein Kunde von `B` annehmen, daß `del_ref` immer auf ein gültiges Elternobjekt verweist. In Abschnitt 7.2 wird diese Annahme in das Vertragsmodell von Eiffel integriert.

Für den anderen Fall, also die Duldung von ungültigen Delegationsreferenzen muß der Entwickler einer Klasse den späteren Benutzern einen expliziten Hinweis geben. Eine Delegationsreferenz ist dann wie folgt zu deklarieren.

```
class B
delegate A optional to del_ref;
...
end -- class B
```

Das Schlüsselwort `optional` besagt, daß ein Kunde im Prinzip *nie* davon ausgehen kann, daß die Delegationsreferenz gültig ist. Wie die Gültigkeit eines Merkmalsaufrufs zu einem konkreten Zeitpunkt durch einen Kunden festgestellt werden kann, ist Thema des folgenden Abschnitts.

5.5.3 Die versuchte Zuseisung

Wie kann ein Kunde feststellen, ob eine Delegationsreferenz gültig ist? Intuitiv sicherlich durch die explizite Prüfung dieser Referenz, also etwa durch

```
if obj.del_ref /= Void then ...
```

Da die Delegationsreferenz für den Kunden jedoch nicht sichtbar sein muß, scheidet diese Lösung aus. Außerdem kann Delegation über mehrere Objekte hinweg erfolgen, so daß sich der Kunde durch einen Pfad hangeln müßte, was unnötiges Wissen über die Klassenhierarchie voraussetzen würde.

Die Lösung des Problems ist jedoch einfach. Sie basiert darauf, daß Eiffel kontrolliertes *Downcasting von Referenzen* erlaubt. Angenommen, in einer Klassenhierarchie existieren die folgenden Klassen:

```
class OBERKLASSE ... end              class UNTERKLASSE
                                      inherit OBERKLASSE
                                      ...
                                      end -- class UNTERKLASSE
```

Wenn z.B. eine Methode eines Kunden als formalen Parameter eine Referenz auf ein Objekt vom Typ `OBERKLASSE` hat, möchte sie vielleicht prüfen, ob sich hinter dieser Referenz nicht ein Objekt vom Typ `UNTERKLASSE` verbirgt (Polymorphismus), ob es also auch über eine Referenz dieses Typs angesprochen werden könnte (siehe folgendes Beispiel). Dieser Vorgang, wird als *Downcasting* bezeichnet. Der Begriff "Downcasting" ist im Bezug auf Eiffel nicht ganz korrekt, da das Objekt selbst nicht verändert wird, sondern nur die Referenz konvertiert wird. In C++ kann auch das Objekt selbst und nicht nur die Referenz konvertiert werden. Downcasting gilt im allgemeinen als schlechter Programmierstil, es deutet oft auf einen schlecht durchdachten Entwurf hin [Mey95, Seite 169 ff.]. In [Mey97, Kapitel 16.5] wird jedoch an einem Beispiel verdeutlicht, daß Downcasting manchmal notwendig ist. In Eiffel existiert zu diesem Zweck die versuchte Zuweisung:

```
class KUNDE
   methode_1( obj_o : OBERKLASSE ) is
   local
      obj_u : UNTERKLASSE;
   do
      obj_u ?= obj_o
      if obj_u /= Void then
         -- Zuweisung war erfolgreich,
         -- es handelt sich um ein Objekt vom Typ UNTERKLASSE
      else
         -- Zuweisung war nicht erfolgreich
      end
   end
end -- class KUNDE
```

Hier wird die Zuweisung von `obj_o` an `obj_u` also nur dann ausgeführt, wenn das von `obj_o` referenzierte Objekt tatsächlich vom Typ `UNTERKLASSE` ist, sonst wird `obj_u` `Void`. In C++ und Java wird sicheres Downcasting durch Ausnahmen realisiert. Wenn die Konvertierung fehlschlägt, wird eine Ausnahme ausgelöst, die vom Kunden aufgefangen werden muß. In [Mey97, Kapitel 16.5] wird dieses Verfahren scharf kritisiert. Es wird angeführt, das der Fehlschlag einer Konvertierung keine außergewöhnliche Situation im Ablauf eines Programms ist und das eine Ausnahme deshalb der falsche Weg ist.

Die versuchte Zuweisung läßt sich syntaktisch unverändert auch für Delegationsbeziehungen übernehmen. Wie in Abschnitt 3.3.3 beschrieben, ist ein Objekt `obj_b` von Typ `B`(siehe Abbildung 5.4), das an ein Objekt `obj_a` vom Typ `A` delegiert, auch vom Typ `A`, ganz so, als würde `B` von `A` erben. Durch

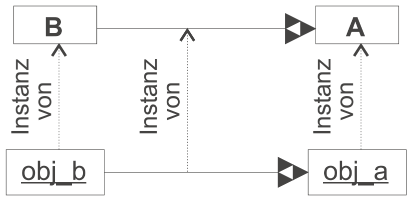

Abbildung 5.4: Da eine Delegationsbeziehung eine Subtyp-Beziehung induziert, kann eine Referenz vom statischen Typ der Elternklasse auf eine Instanz der Kindklasse verweisen.

die Delegationsbeziehung stehen alle exportierten Merkmale von `A` auch für Kunden der Klasse `B` zur Verfügung.

Bei Delegationsbeziehungen ist die linke Seite einer versuchten Zuweisungs eine Referenz vom Typ der Elternklasse, die rechte Seite eine Referenz auf das zu testende Objekt, eine Instanz der Kindklasse. Der Zuweisungsversuch wird nur dann erfolgreich sein, wenn die Delegationsreferenz gültig ist. Wenn `B optional` an `A` delegiert und die Delegationsreferenz `Void` ist, wird der Zuweisungsversuch `obj_a ?= obj_b` fehlschlagen und `obj_a` auf `Void` setzen. Es wird also nicht die Existenz bestimmter Merkmale überprüft, sondern festgestellt, ob ein Objekt aktuell einen bestimmten Typ hat. Im Gegensatz zum Zuweisungsversuch bei Vererbungsbeziehungen wird eine Referenz vom Typ der Superklasse an ein Objekt der Subklasse gebunden, es findet also ein "Upcast" statt.

5.5.4 Typfehler zur Laufzeit

Dem Vorteil, daß sich der Zuweisungsversuch für Delegations- ebenso einfach wie für Vererbungsbeziehungen nutzen läßt, steht ein Nachteil im Weg, der bei unvorsichtigem Umgang mit Delegation zu Laufzeit-Typfehlern führen kann.

Wenn die Überprüfung einer Delegationsreferenz erfolgreich war, so hat ein Kunde eine Referenz vom statischen Typ der Elternklasse, die jedoch auf eine Instanz der Kindklasse zeigt. Die Verbindung zwischen der Referenz und dem Objekt ist jedoch nur so lange gültig, wie die Delegationsreferenz gültig ist. Angenommen, besagte Referenz wird an einen anderen Kunden weitergegeben, der sich nicht bewußt ist, eine Referenz auf ein Kindobjekt zu erhalten. Der Aufruf eines Merkmals der delegierten Klasse führt zu einem Typfehler, wenn die Delegationsreferenz zwischenzeitlich ungültig geworden ist. Das folgende Beispiel konstruiert einen solchen Fehler und macht deutlich, daß optionale Delegationsbeziehungen nur mit Vorsicht behandelt werden dürfen.

```
class A
feature:
   merkmal_1 ...
...
end -- class A

class B
delegate A optional to del_ref;
feature{ KUNDE }
   delegation_aufheben is
   do
      del_ref := Void
   end
...
end -- class B
```

```
class KUNDE
feature

   methode_1 is
   local
      obj_a : A
   do
      obj_a := get_ref
      obj_a.merkmal_1 -- Typfehler
   end

   get_ref : A is
   local
      obj_a : A
      obj_b : B
   do
      !!obj_b
      ...
      obj_a ?= obj_b
      if obj_a /= Void then
         Result := obj_a
         obj_b.kill_delegation
      else
         -- fuer das Beispiel unwichtig
      end
   end
end -- class KUNDE
```

5.5.5 Delegierte Delegationsreferenzen

In Abschnitt 5.1.1 wurde begründet, warum Delegationsreferenzen, wie andere Merkmale auch, im `feature`-Teil einer Klasse deklariert werden. Dadurch ergibt sich eine neue Möglichkeit: eine Delegationsreferenz muß nicht in der Klasse deklariert werden, die die Delegationsbeziehung deklariert. Sie kann auch aus einer Superklasse stammen. In Abbildung 5.5 a) erbt eine Klasse `B` von einer Klasse `A` und delegiert an eine Klasse `C`. Die Delegationsreferenz für die Delegationsbeziehung zu `C` erbt `B` von `A`. In Eiffel könnten dieses Beispiel wie folgt geschrieben werden:

```
class C ... end -- class C

class A
feature{ B }
   ref_C : C
end -- class A
```

```
class B
inherit A
delegate C to ref_C
...
end -- class B
```

Durch die Vererbungsbeziehung zwischen `A` und `B` verfügen alle Instanzen von `B` über ein Merkmal `ref_C`. Daher ist diese Möglichkeit umkompliziert. Wenn `B`, statt von `A` zu erben, an `A` delegiert, und diese Delegationsbeziehung ist optional, kommt es zu der in Abbildung 5.5 b) dargestellten Situation. Um einen Merkmalsaufruf an `C` delegieren zu können, muß zunächst der Zugriff auf die Delegationsreferenz `ref_C` an `A` delegiert werden. Da optional an `A` delegiert wird, ist dies jedoch nicht immer möglich. Die Gültigkeit der Delegationsbeziehung zu `C` ist also davon abhängig, ob die Delegationsbeziehung zu `A` gültig ist, d.h. implizit delegiert `B` an `C` ebenfalls optional. Da implizite Sachverhalte der Philosophie von Eiffel widersprechen, muß ein Compiler eine Fehlermeldung ausgeben, wenn `B` wie folgt geschrieben wird:

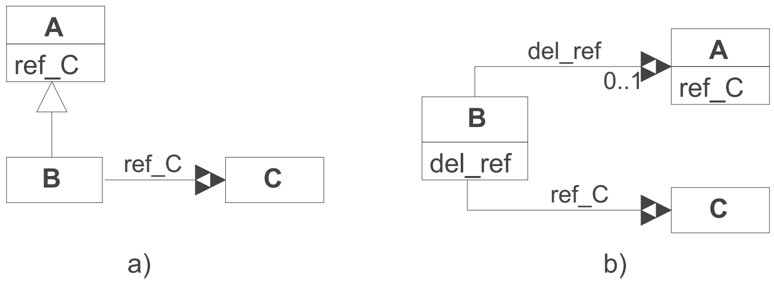

Abbildung 5.5: Delegationsreferenzen können geerbte Merkmale sein

```
class B
delegate A optional to del_ref
delegate C to ref_C -- Fehler, muss auch optional sein
feature
   del_ref : A
   ...
end -- class B
```

5.5.6 Verschärfung von geerbten Delegationsbeziehungen

Wenn eine Klasse `B` eine optionale Delegationsbeziehung von einer Superklasse `A` erbt, hat sie die Möglichkeit, die unsichere Beziehung sicher zu machen, d.h. auf `optional` zu verzichten. Dies kann durch erneute Aufführung der Beziehung im `delegate`-Teil der Klasse geschehen. Für Kunden, die Instanzen der Klasse `B` polymorph über eine Referenz vom statischen Typ `A` ansprechen, bedeutet diese Verschärfung kein Nachteil. Der umgekehrte Fall, also eine geerbte sichere Delegationsbeziehung optional zu machen, ist natürlich nicht erlaubt.

```
class C ... end -- class C

class A
delegate C optional to del_ref
feature
   del_ref : C
end -- class A
```

```
class B
inherit A
delegate C to del_ref
...
end -- class B
```

5.6 Zusammenfassung

Delegationsbeziehungen werden wie Vererbungsbeziehungen zu Beginn einer Klasse im `delegate`-Teil vereinbart. Das Elternobjekt wird über eine Delegationsreferenz zur Laufzeit identifiziert, die im `feature`-Teil einer Klasse wie andere Merkmale auch deklariert wird. Sie kann auch wie ein gewöhnliches Attribut benutzt werden. Anstatt durch Attribute kann auch über parameterlose Funktionen delegiert werden. Die Delegationsbeziehung zwischen einer Kindklasse und einer Elternklasse ist eine geschlossene Beziehung. Delegationsbeziehungen werden an Subklassen vererbt und können von dort bei Bedarf getrennt oder covariant redefiniert werden. Eine Kindklasse kann den Exportstatus von Merkmalen der Elternklasse einschränken jedoch nicht erweitern. Einschränkungen können zu Laufzeit-Typfehlern führen. Eine Delegationsreferenz muß immer auf ein gültiges Objekt verweisen wenn sie nicht explizit als `optional` deklariert wurde. Ein Kunde kann durch versuchte Zuweisung ermitteln, ob eine als `optional` deklarierte Delegation zu einem bestimmten Zeitpunkt möglich ist. Optionale Delegationsbeziehungen können zu Laufzeit-Typfehlern führen.

Kapitel 6

Vererbungstechniken für Delegationsbeziehungen

Die in Abschnitt 4.3 angedeuteten Möglichkeiten von Eiffel zum Umgang mit und zur Manipulation von Vererbungsbeziehungen sollen nun für Delegationsbeziehungen angepaßt und, falls nötig, erweitert werden. Für alle Manipulationen, die im folgenden untersucht werden, gilt grundsätzlich, daß Delegation eine *geschlossene* Beziehung ist. Wenn die Elternklasse bestimmte Merkmale nicht an die Kindklasse exportiert, darf die Kindklasse sie auch nicht verändern.

6.1 Umbenennen delegierter Merkmale

Das Umbenennen von Merkmalen ist ein Mechanismus, der es dem Entwickler einer Unterklasse ermöglicht, geerbte Merkmale unter einem anderen Namen zur Verfügung zu stellen. Für die Umbenennung gibt es im wesentlichen zwei Gründe:

- Zum einen kann es sinnvoll sein, den Namen eines geerbten Merkmals an den neuen Kontext der Unterklasse anzupassen. Dies kommt vor allem dann häufig vor, wenn Klassen aus einer Bibliothek als Oberklassen dienen und deren Merkmale an die Terminologie des Anwendungsbereichs angepaßt werden.
- Der zweite Grund für das Umbenennen ist die Auflösung von Namenskonflikten. Da Eiffel Mehrfacherben unterstützt, kann es vorkommen, daß mehrere geerbte Merkmale den selben Namen tragen. Der Entwickler muß dann durch Umbenennungen dafür sorgen, daß die Namen wieder eindeutig sind.

Für E+ ist dieses Konstrukt unproblematisch. Bei der Deklaration einer Delegationsbeziehung kann (oder muß) der Entwickler einen `rename`-Abschnitt angeben und Merkmale der Elternklasse umbenennen. Auf jeden Fall müssen auch bei Delegationsbeziehungen Namen eindeutig sein.

Ein wichtiger Aspekt im Zusammenhang mit Umbenennungen ist, daß ein Merkmal durch die Umbenennung seine Identität behält. Dies ist in folgender Situation wichtig:

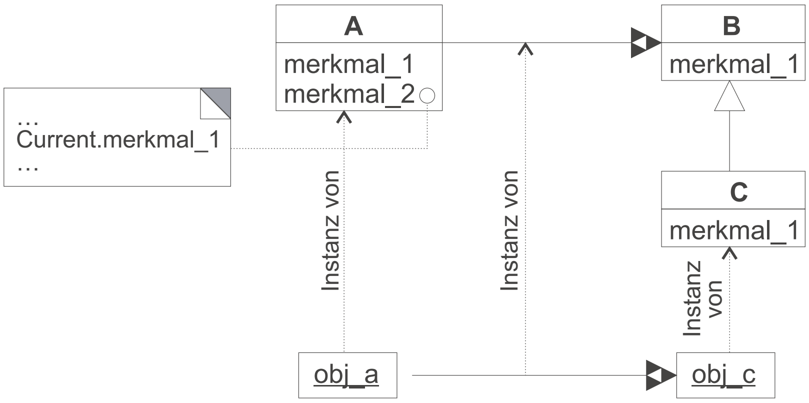

Abbildung 6.1: Aus Vererbungs- und aus Delegationsbeziehungen geerbte Merkmale können gleichberechtigt nebeneinander stehen. In diesem Fall erhält das Merkmal aus der Vererbungsbeziehung den Vorzug.

```
class OBERKLASSE_1
feature
   merkmal_1 ...
   ...
end -- class OBERKLASSE_1

class OBERKLASSE_2
   merkmal_1 ...
   ...
end -- class OBERKLASSE_2
```

```
class UNTERKLASSE
inherit
   OBERKLASSE_1
      rename   merkmal_1 as merkmal_2;
      redefine merkmal_2;
   end
   OBERKLASSE_2
...
end -- class U
```

Die Klasse `UNTERKLASSE` erbt zwei Merkmale gleichen Namens von ihren beiden Oberklassen. Eines wird redefiniert und umbenannt, das andere wird nicht verändert. Wenn nun eine andere von `OBERKLASSE_1` geerbte Methode `merkmal_1` aufruft, wird das Merkmal `merkmal_2` von `UNTERKLASSE` aufgerufen, dies ist das korrekte Merkmal für diesen Aufruf, obwohl ein Merkmal `merkmal_1` zur Verfügung steht.

6.2 Redefinition delegierter Merkmale

Genau wie Merkmale von Oberklassen können auch Merkmale von Elternklassen redefiniert und an den Kontext der Kindklasse angepaßt werden. Im großen und ganzen gelten für beide Arten der Redefinition die gleichen Regeln. Allerdings gibt es einige Besonderheiten, die bei Delegationsbeziehungen zu beachten sind.

6.2.1 Vorfahrtsregeln für Merkmalsaufrufe

In dem in Abbildung 6.1 dargestellten Szenario wird die Methode `methode_1` sowohl durch die an `B` delegierende Klasse `A`, als auch durch die von `B` erbende Klasse `C` redefiniert. Wenn nun die Delegationsreferenz einer Instanz `obj_a` von `A` zur Laufzeit eine Instanz von `C` referenziert, existieren für den Aufruf `obj_a.merkmal_1` zwei Implementierungen, die beide das ursprüngliche Merkmal in `B` ersetzen. Da die Reihenfolge, in der `A` und `C` entwickelt werden beliebig ist, kann in keiner der beiden Klassen dieser Konflikt gelöst werden.

In einem solchen Fall erhält die Version in `A` den Vorzug vor der in `C`. Der Grund ist der, daß für einen Kunden von `A` die in `A` eingeführte Implementierung näher liegt. Unter Umständen weiß er überhaupt

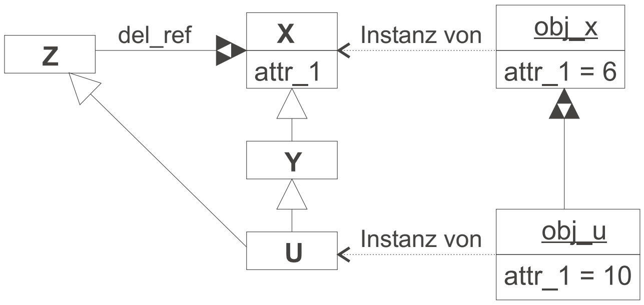

Abbildung 6.2: Wenn ein Merkmal von einer indirekten Superklasse über mehrere Pfade geerbt wird, muß zwischen Vererbungs- und Delegationsbeziehungen unterschieden werden.

nichts von der Existenz von `C`. Für den Entwickler der Klasse `C` bedeutet dies keine Einschränkung. Wenn er in einer anderen Methode der Klasse den Aufruf `merkmal_1` macht, kann er auch ohne Delegation nicht davon ausgehen, daß genau seine Implementierung vom Laufzeitsystem gewählt wird. In einer von `C` abgeleiteten Klasse könnte `merkmal_1` erneut redefiniert werden, was in Verbindung mit spätem Binden zur Wahl dieser Implementierung führen würde.

6.2.2 Wiederholtes Erben eines Merkmals

In Abschnitt 4.3 wird ein Beispiel für wiederholtes Erben gegeben. Die Klasse erbt `Amphibienfahrzeug` ein Merkmal (`Breite`) von einer Oberklasse indirekt über zwei unterschiedliche Pfade (`Schiff` und `Auto`). Es wurde dort erklärt, warum das Merkmal trotzdem nur einmal in `Amphibienfahrzeug` vorkommt und warum das Merkmal `Geschwindigkeit` explizit dupliziert werden muß.

Bei Delegationsbeziehungen kann eine ähnliche Situation auftreten (siehe auch Abbildung 6.2):

```
class X
feature
   attr_1 ...
end -- class X

class Y
   inherit X
   ...
end -- class Y
```

```
class Z
   delegate X to del_ref
   ...
end -- class Z

class U
inherit Y, Z end
...
end -- class U
```

Auch hier erbt `U` das Attribut `X.attr_1` sowohl über `Y`, als auch über `Z`. Man muß jedoch berücksichtigen, daß zwischen `Z` und `X` eine Delegationsbeziehung besteht, d.h. ein Aufruf von `merkmal_1` für ein Objekt der Klasse `Y` wird an ein anderes Objekt, nämlich das Elternobjekt, auf das `del_ref` verweist, weitergeleitet. Das Attribut `attr_1` benötigt in `U` zweimal physisch Speicherplatz, einmal in der Instanz von `U` selbst und auch in dem durch die geerbte Referenz `del_ref` bezeichneten Elternobjekt. Hier liegt also in `U` ein echter Konflickt vor, der Compiler muß `U` als fehlerhaft zurückweisen.

Um `U` korrekt zu schreiben, muß der Konflikt aufgelöst werden, indem entweder das von `Y` oder das von `Z` geerbte Attribut umbenannt oder eine andere Technik zur Auflösung von Konflikten angewendet wird.

6.2.3 Redefinitionen, die keine sind

In [Kni96] wird ein Problem angesprochen, das über die im vorherigen Abschnitt vorgestellten Zweideutigkeiten hinaus zu Problemen bei der Wahl einer Implementierung führt. Dieses Problem tritt

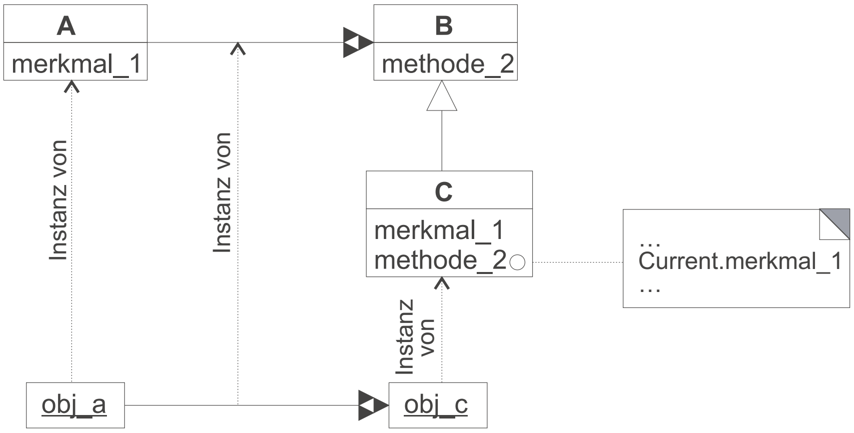

Abbildung 6.3: Das Merkmal `A.merkmal_1` ist keine Redefinition des Merkmals `C.merkmal_1`. Wenn `obj_a.merkmal_2` aufgerufen wird, wird `A.merkmal_1` bei der anschließenden Suche nach einer Methode `merkmal_1` von `D` aus nicht beachtet und `C.merkmal_1` aufgerufen.

nur in Sprachen wie C++ oder Java auf. Die Behandlung hilft jedoch beim Verständnis der Zusammenhänge in einem System mit Delegation. Es wird daher kurz beschrieben. Dabei wird auch deutlich, warum dieses Problem in Eiffel nicht auftreten kann, obwohl Eiffel Vererbung konzeptionell genau so behandelt wie C++ oder Java es tun.

In Abbildung 6.3 wird ein Szenario dargestellt, in dem es sowohl in der Kindklasse `A`, als auch in einer von der Elternklasse `B` abgeleiteten Klasse `C` ein Merkmal `merkmal_1` gibt. Wenn nun die Delegationsreferenz von `obj_a` auf eine Instanz von `C` verweist, so wird der Aufruf von `methode_2` durch einen Kunden delegiert. Die Methode `methode_2` ruft ihrerseits das Merkmal `merkmal_1` auf. Da `Current` immer noch auf `obj_a` zeigt, könnte man meinen, daß nun `A.merkmal_1` aufgerufen würde, da `C.merkmal_1` redefiniert wird. Bei genauerer Betrachtung wird jedoch klar, daß es sich um keine Redefinition handelt. Das Merkmal `merkmal_1` kann vom Entwickler der Klasse `A` nicht mit der Absicht zur Redefinition eingeführt worden sein, da ihm die Existenz von `C.merkmal_1` überhaupt nicht bekannt sein muß. Der Merkmalsaufruf in `methode_2` muß also korrekterweise zu `C.merkmal_1` geleitet werden.

In C++ oder Java wäre dies ein Problem da die Auswahl von Methoden und Attributenn in diesen Sprachen nach dem Namen (genauer: nach der Signatur) erfolgt. Auch die Redefinition von Methoden erfolgt aufgrund der Signatur. Um eine Methode zu redefinieren, reicht es, in der Unterklasse eine neue Methode mit gleicher Signatur zu schreiben. In Eiffel stellt dies kein konzeptionelles Problem dar. Um ein Merkmal zu redefinieren, muß es im `inherit`-Abschnitt, bzw. `delegate`-Abschnitt entsprechend aufgeführt werden. Da `merkmal_1` im `redefine`-Abschnitt von `A` nicht erwähnt wird, findet auch keine Redefinition statt.

6.2.4 Redefinition abstrakter Merkmale

In der objektorientierten Programmierung gibt es den Begriff der *abstrakten Methoden*. Eine Klasse deklariert eine Methode, implementiert sie aber nicht. Andere Methoden dieser Klasse können die abstrakte Methode jedoch aufrufen. Wegen der fehlenden Implementierung bleibt die Klasse unvollständig und kann daher nicht instanziiert werden. Die abstrakte Methode wird in einer Unterklasse implementiert. Der Aufruf der abstrakten Methode in der Oberklasse wird zur Laufzeit an eine Methode der Unterklasse gebunden. Dieser Bindevorgang wird *spätes Binden* genannt.

In Eiffel werden abstrakte Merkmale als *aufgeschoben* (Schlüsselwort: `deferred`) bezeichnet. Eine Klasse ist abstrakt, wenn sie mindestens ein abstraktes Merkmal enthält. Die Implementierung in einer Unterklasse wird *effecting* genannt. Da es sich um eine Implementierung und nicht um eine

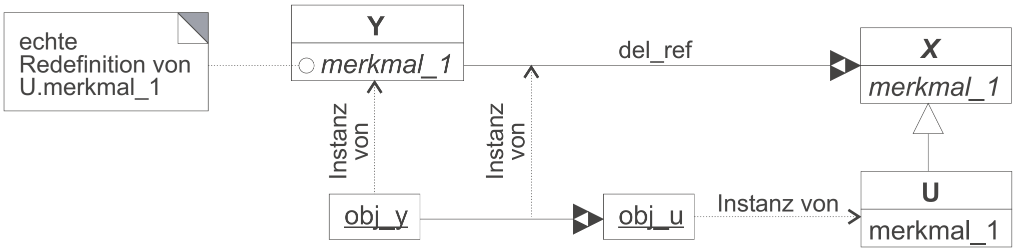

Abbildung 6.4: Abstrakte Merkmale aus Delegationsbeziehungen müssen immer redefiniert werden, da zur Laufzeit an eine Subklasse der Elternklasse delegiert wird und so eine Implementierung ersetzt wird.

echte Redefinition handelt, braucht das abstrakte Merkmal nicht in der `redefine`-Liste im `inherit`-Abschnitt der Unterklasse aufgeführt zu werden. Eine abstrakte Klasse und die Implementierung eines abstrakten Merkmals in einer Unterklasse sieht beispielsweise so aus:

```
deferred class X
feature
   merkmal_1 is
   deferred
   end
   ...
end -- class X
```

```
class Y
inherit X
feature
   merkmal_1 is
   do
      ...
   end
   ...
end -- class Y
```

Bei Delegationsbeziehungen zwischen zwei Klassen muß man abstrakte Merkmale mit anderen Augen betrachten (siehe Abbildung 6.4). Wenn die Elternklasse `X` eine abstrakte Klasse ist, wird das Elternobjekt zur Laufzeit immer Instanz einer Subklasse von `X`, z.B. von `U` sein. Da `U` instanziierbar ist, wurden alle abstrakten Merkmale von `X` dort implementiert. Wenn `Y` ein Merkmal `merkmal_1` enthält, handelt es sich nicht um eine einfache Implementierung des abstrakten Merkmals aus `X`, sondern um eine echte Redefinition des Merkmals von `U`. Der Entwickler von `Y` braucht zwar keine konkrete Subklasse von `X` zu kennen, er weiß aber, daß es eine geben muß. Sollen also abstrakte Merkmale aus einer Delegationsbeziehung in einer Kindklasse implementiert werden, müssen sie in der `redefine`-Liste aufgeführt werden.

6.2.5 Schreibzugriff auf Attribute durch Elternklassen

Wie in Abschnitt 5.2.1 bereits erwähnt, verbietet Eiffel grundsätzlich den schreibenden Zugriff auf Attribute durch Methoden anderer Objekte, Benutzbeziehungen sind *geschlossene* Beziehungen. In E+ muß diese Regel aufgeweicht werden, da ein schreibender Zugriff in bestimmten Fällen ermöglicht werden muß.

Wenn, wie im folgendne Beispiel (siehe Abbildung 6.5), eine Klasse `X` ein Attribut `attr_1` deklariert, so dürfen die eigenen Methoden darauf schreibend zugreifen. Die Wertzuweisung in `X.methode_1` ist also korrekt.

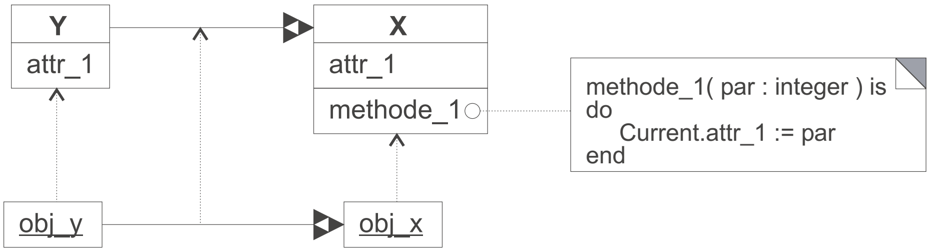

Abbildung 6.5: In bestimmten Situationen können Elternobjekte auf Attribute ihrer Kindobjekte schreibend zugreifen.

```
class X
feature{ ANY }
   attr_1 : integer
   methode_1( par : integer ) is
   do
      attr_1 := par
   end
   ...
end -- class X
```

```
class Y
delegate X to del_ref
   redefine attr_1 end
feature
   del_ref : Y
   attr_1 : integer
   ...
end -- class Y
```

```
class KUNDE
feature
   eine_methode is
   local
      obj_y : Y
   do
      !! obj_y
      obj_y.methode_1( 42 ) -- erlaubt, der Aufruf wird delegiert
      check obj_y.attr_1 = 42 end
   end
end -- class KUNDE
```

Die Klasse `Y` delegiert an `X` und redefiniert `attr_1`. Zur Laufzeit ruft `KUNDE` die Methode `methode_1` auf. Dieser Aufruf wird an das Elternobjekt, eine Instanz von `X`, delegiert. Die Regeln für Delegationsbeziehungen (siehe Kapitel 3.3) sagen, daß der Zugriff auf `attr_1` innerhalb von `methode_1` auf das redefinierte Attribut `Y.attr_1` erfolgt. Dabei handelt es sich jedoch um das Attribut eines anderen Objekts.

Dieses Beispiel verdeutlicht die Notwendigkeit, den Schreibzugriff von Elternklassen auf Attribute in Kindklassen zu ermöglichen, auch wenn dadurch die ursprüngliche Regel des Schreibverbots für Attribute in anderen Objekten aufgeweicht wird. Eine Alternative zu dieser Lösung kann nur bedeuten, daß die Redefinition von Attributen in Kindklasseen verboten wird und nur in Vererbungsbeziehungen erlaubt ist. Damit würde jedoch ein wesentlicher Vorteil von Delegation nicht genutzt werden können. In Kapitel 9 wird zudem deutlich werden, daß die Redefinition von Attributen in Delegationsbeziehungen bei der Implementierung von Entwurfsmustern hilfreich ist.

Wenn eine Kindklasse keine Attribute ihrer Elternklasse redefiniert, braucht sich ihr Entwickler um mögliche Schreibzugriffe durch Elternobjekte nicht zu kümmern. Redefiniert er hingegen Attribute, so ist davon auszugehen, daß er sich der Problematik bewußt ist und um die Schreibzugriffe durch die Elternklasse weiß. Eine versehentliche Redefinition ist nicht möglich, da sie explizit in der `redefine`-Liste der Kindklasse angekündigt werden muß, der Compiler würde sonst einen Fehler melden. Die Stabilität eines mit Eiffel entwickelten Software-Systems sollte daher unter dieser Aufweichung nicht leiden.

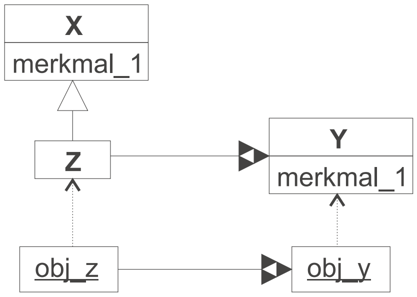

Abbildung 6.6: Wenn eine Delegationsreferenz `Void` ist, kann unter Umständen Merkmale anderer Klassen stellvertretend aufgerufen werden.

6.2.6 Covariante Redefinition

Bei der Redefinition von geerbten Merkmalen darf ihre Signatur in der erbenden Klasse verändert werden. Der Typ von Attributen, bzw. die Typen der Parameter und der Typ des Rückgabewerts von Methoden dürfen durch zum ursprünglichen Typ *konforme* Typen (siehe Abschnitt 3.3.3) ersetzt werden. Die Wahrung der Konformität bedeutet, daß die Klasse des neuen Typs eine Subklasse der Klasse des alten Typs sein muß. Wenn man in der Klassenhierarchie von der allgemeinen zur speziellen Klasse wandert, so werden auch redefinierte Typen immer spezieller. Daher wird diese Art der Redefinition *covariant* genannt. Die Vorteile gegenüber der *contravarianten* Redefinition werden in [Mey97] ausführlich erläutert. Redefinitionen im Rahmen von Delegationsbeziehungen dürfen Typen ebenfalls covariant abändern. Es ergibt sich also kein Unterschied zwischen Vererbung und Delegation.

Covariante Redefinition führt jedoch zu möglichen Laufzeit-Typfehlern. Neben Einschränkungen beim Export (siehe Abschnitt 5.3.3) stellt sie das zweite Loch im statischen Typsystem von Eiffel dar. Typfehler durch covariante Redefinition treten aber auch bei Vererbungsbeziehungen auf und werden durch Delegationsbeziehungen nicht verschärft. Daher werden sie in dieser Arbeit nicht näher behandelt.

6.3 Auswahl delegierter Merkmale

Delegationsbeziehungen können als `optional` deklariert werden (siehe Abschnitt 5.5.2). Das Elternobjekt und damit ein bestimmtes Merkmal ist nicht in jedem Fall verfügbar. Angenommen, in einem konkreten System ergibt sich die folgende Situation (siehe auch Abbildung 6.6):

```
class X
   feature merkmal_1 ...
   ...
end -- class X
```

```
class Y
   feature merkmal_1 ...
   ...
end -- class Y
```

```
class Z
   inherit X
   delegate Y optional to del_ref
   select Y.merkmal_1
   ...
end -- class Z
```

Der Aufruf `obj_z.merkmal_1` ist für eine Instanz von `Z` damit eindeutig, der Entwickler hat dem Merkmal aus der Delegationsbeziehung eindeutig den Vorrang gegeben. Wenn bei einem Aufruf die Delegationsreferenz ungültig ist, wird die Sache komplizierter. Natürlich könnte der Aufruf einfach zu einem Laufzeitfehler führen, den der Aufrufer behandeln muß. Andererseits steht aber immer noch das Merkmal aus der Vererbungsbeziehung zur Verfügung. Sind diese beiden Merkmale austauschbar, d.h. haben sie die gleiche Semantik?

Wenn `merkmal_1` wie in Abschnitt 4.3 gezeigt, auf unterschiedlichen Pfaden aus ein und derselben Superklasse geerbt worden wäre, hätten beide Merkmale `X.merkmal_1` und `Y.merkmal_1` die gleiche Semantik, ja sogar die gleiche Implementierung. Auch wenn dies nicht der Fall sein sollte, so kann man annehmen, daß sie eine zumindest ähnliche Semantik haben. Wenn eine Instanz von `Z` einer Referenz vom statischen Typ `X` zugewiesen wird, so wird der Aufruf `obj_x.merkmal_1` durch die `select`-Anweisung delegiert, es wird also `Y.merkmal_1` aufgerufen. Der Entwickler von `Z` würde dies wahrscheinlich nicht wollen, hätten beide Merkmale eine völlig unterschiedliche Semantik. Dann könnte er eines der beiden Merkmale umbenennen und auf die `select`-Anweisung verzichten. Man kann also davon ausgehen, daß die Merkmale in gewissen Grenzen austauschbar sind.

Wenn nun bei einem Aufruf `obj_z.merkmal_1` die Delegationsreferenz `del_ref Void` ist, kann unter Umständen `X.merkmal_1` aufgerufen werden. Ob dies sinnvoll ist, kann der Entwickler von `Z` nur anhand des konkreten Falls entscheiden. Falls ja, sollte er auf jeden Fall die Möglichkeit haben, automatisch das andere Merkmal benutzen zu lassen, wenn `del_ref` bei einem Aufruf von `merkmal_1` `Void` ist. Die `select`-Anweisung wird für E+ daher erweitert. In obigem Beispiel kann sie wie folgt geschrieben werden:

```
select Y.merkmal_1 then X.merkmal_1
```

Im allgemeinen kann für jedes Merkmal einer `select`-Anweisung eine Liste von Implementierungen angegeben werden. Die Liste wird durch eine garantiert verfügbare Implementierung beendet. Diese wird entweder aus einer sicheren Delegations- oder einer Vererbungsbeziehung stammen. Die Angabe weiterer Implementierungen wäre sinnlos, da diese nie benutzt würden. Bricht die Liste mit einer unsicheren Implementierung ab, so ist dies ebenfalls korrekt. Der Aufruf dieses Merkmals kann dann nach wie vor zu einem Laufzeitfehler führen.

6.4 Super-Aufruf

Durch Redefinition erhält eine geerbte Methode eine neue Implementierung. Oft ist es jedoch sinnvoll, innerhalb der neuen auf die ursprüngliche Implementierung zurückzugreifen. Es wird also eine Möglichkeit benötigt, die redefinierte Methode aufzurufen. Ein solcher Aufruf wird allgemein *Super-Aufruf* genannt.

In Eiffel existiert zu diesem Zweck in jeder redefinierenden Methode das Symbol `Precursor`. Es steht stellvertretend für die Implementierung der Methode in der Oberklasse. Wenn durch Mehrfacherben `Precursor` nicht eindeutig ist, eine Methode also in mehr als einer Oberklasse definiert ist, kann mit {`Oberklasse`} `Precursor` die aufzurufende Methode spezifiziert werden.

Dieser Ansatz kann für E+ unverändert übernommen werden. Wenn eine Klasse eine durch eine Delegationsbeziehung geerbte Methode redefiniert, verweist `Precursor` auf die ursprüngliche Version der Methode in der Elternklasse. Sollte es zwischen Elternklassen oder zwischen Ober- und Elternklassen zu Mehrdeutigkeiten kommen, kann `Precursor` entsprechend qualifiziert werden. Das folgende Beispiel verdeutlicht den `Precursor`-Aufruf:

```
class X                              class Y
feature{ Y }                         delegate X to del_ref
   methode_1 is                         redefine methode_1 end
   do                                feature{ ANY }
      -- Methodenname auf               methode_1 is
      -- Standardausgabe                do
      -- ausgeben                          Precursor
      io.put_string                        io.put_string(
         ( "X.methode_1" )           " redefiniert durch Y.methode_1")
   end                                  end
   ...                                  ...
end -- class X                       end -- class Y
```

Der Aufruf von `methode_1` für ein Objekt vom Typ `Y` führt zu folgender Ausgabe:

```
X.methode_1 redefiniert durch Y.methode_1
```

Beim `Precursor`-Aufruf wird `Current` nicht verändert, es gelten also die Regeln für eine gewöhnliche Delegation. Aus diesem Grund ist streng zwischen den folgenden Aufrufen zu unterscheiden, die zwar die selbe Methode ausführen, jedoch einmal mit und einmal ohne Delegationssemantik:

```
Precursor          -- delegiert an X.methode_1
del_ref.methode_1  -- ruft X.methode_1 als Kunde auf
```

6.5 Zusammenfassung

Die Vererbungs-Techniken von Eiffel lassen sich auch für Delegationsbeziehungen nutzen. An mehreren Stellen sind jedoch syntaktische und semantische Erweiterungen nötig. Diese resultieren zu einem großen Teil daraus, daß zur Laufzeit nicht nur ein Objekt existiert und daß Merkmale physisch mehrfach vorhanden sind. Die Kapselung von Objekten wird an einigen Stellen verletzt. Es ergeben sich mögliche Laufzeit-Typfehler. Einige können auch bei Vererbungsbeziehungen auftreten, andere werden erst durch Delegationsbeziehungen möglich.

Kapitel 7

Das Vertragsmodell

Im täglichen Leben kommt es an vielen Stellen zu Verträgen zwischen zwei Parteien über die Erbringung einer Leistung. Der eine Vertragspartner (Lieferant) verpflichtet sich dazu, die im Vertrag vereinbarte Leistung zu erbringen, der andere (Kunde) wiederum verpflichtet sich, für die vertragsgemäß erbrachte Leistung den vereinbarten Preis zu zahlen

Für viele Verträge ist die Schriftform vorgeschrieben (z.B. für Grundstücksgeschäfte). Voraussetzungen und Umfang der vereinbarten Leistung werden niedergeschrieben. Wenn ein Vertragspartner der Meinung ist, der andere habe den Vertrag gebrochen, so sollte dies — möglicherweise durch hinzuziehen eines Gerichts — aus dem schriftlichen Vertrag heraus beweisbar sein.

In einem objektorientierten Software-System rufen Objekte gegenseitig Merkmale anderer Objekte auf. Ein Merkmalsaufruf kann als Vertrag aufgefaßt werden, d.h. ein Objekt tritt als Lieferant einer Leistung (in Form eines Merkmals) auf und ein anderes Objekt nimmt diese Leistung als Kunde in Anspruch. Der Kunde muß als Gegenleistung einen Kontext schaffen, in dem der Merkmalsaufruf erlaubt ist. Die Bezahlung erfolgt quasi im voraus.

Diese Sicht auf Merkmalsaufrufe wird in Eiffel *"Programmierung als Vertrag" (programming by contract)* genannt. Die Sprache bietet Konstrukte an, um Verträge zu schließen und — was sehr wichtig ist — die Einhaltung dieser Verträge automatisch überwachen zu lassen. Im folgenden Abschnitt wird das Vertragsmodell vorgestellt. Anschließend wird gezeigt, daß auch Delegation als Vertrag zwischen Objekten gesehen werden kann. Die notwendige Erweiterung des Vertragmodells für Delegation ist ein weiteres Thema in diesem Kapitel.

7.1 Das Vertragsmodell in Kürze

"Programmieren als Vertrag" dient in Eiffel dazu, die Semantik einer Klasse oder eines Merkmals unabhängig von der Implementierung zu beschreiben. Die Art der Beschreibung ähnelt der algebraischen Beschreibung abstrakter Datentypen [Has95] kommt aber an deren Mächtigkeit nicht heran. Merkmale werden durch boolsche Ausdrücke beschrieben. Im Gegensatz zu echten algebraischen Spezifikationen können diese Ausdrücke keine Quantoren enthalten. Dafür lassen sie sich effizient vom Laufzeitsystem überprüfen.

7.1.1 Vor- und Nachbedingungen von Merkmalen

Ein Merkmal kann angeben, welche Vorbedingungen bei seinem Aufruf gelten sollen. Der Aufrufer darf das Merkmal nur dann aufrufen, wenn er die Gültigkeit der Vorbedingung sichergestellt hat. Das Merkmal selbst kann stillschweigend davon ausgehen, daß die Bedingung erfüllt ist. Eine beliebte Vorbedingung ist das Verbot von `Void`-Referenzen als aktueller Parameter. Eine Methode könnte z.B. so aussehen:

```
eine_methode( par : X ) : integer is
require
   nicht_void: par /= Void;
do
   -- tu etwas
ensure
   nicht_negativ: Result > 0;
end
```

Ein Entwickler darf sich bei der Implementierung von `eine_methode` darauf verlassen, daß `par` nicht `Void` ist, er muß dies in der Methode selbst nicht prüfen. Gleichzeitig verpflichtet er sich, eine positive Zahl zurückzuliefern, worauf sich der Aufrufer wiederum ohne erneute Prüfung verlassen kann.

Bei der Redefinition eines Merkmals können die Zusicherungen verändert werden. Dabei gilt jedoch, daß die neue Implementierung höchstens so viel verlangen darf wie die redefinierte (die Vorbedingung wird gelockert) und mindestens genau so viel leisten muß (die Nachbedingung wird verschärft).

7.1.2 Klassen-Invarianten

Zusätzlich zu den Vor- und Nachbedingungen für Merkmale können für eine Klasse eine Reihe von Invarianten festgelegt werden, die immer gelten müssen. Eine Klassen-Invariante ist gewissermaßen für *jedes* Merkmal eine zusätzliche Vor- und Nachbedingung. Ein beliebtes Beispiel dafür ist z.B. die Realisierung eines Stapels, die folgende Invariante haben könnte:

```
class STAPEL
...
invariant
   -- wenn der Stapel leer ist, wird die Frage
   -- nach der Anzahl der Elemente 0 ergeben
   ist_leer implies anzahl_elemente = 0
end
```

In [Mey97] finden sich viele weitere Beispiele, die alle Aspekte von Vor- und Nachbedingungen und Invarianten (allgemein als Zusicherungen bezeichnet) beleuchten. Die Praxis zeigt, daß es sich um gute Hilfsmittel handelt um Fehler zu vermeiden oder zumindest möglichst früh zu entdecken. Eine vollständige Beschreibung der Semantik einer Klasse oder Methode ist damit jedoch nicht möglich.

7.1.3 Überprüfung von Verträgen

Eiffel ermöglicht die automatische Prüfung von Verträgen. Wenn ein Eiffel-Programm mit den entsprechenden Einstellungen übersetzt wird, werden vor und nach jedem Merkmalsaufruf die Zusicherungen der Methode und alle Klassen-Invarianten überprüft. Jede Vertragsverletzung (ein boolscher Ausdruck wird `False`) führt dann zu einer Ausnahme, meistens gleichbedeutend mit dem Abbruch des Programms. Aus Effezienzgründen kann diese Überprüfung ganz oder teilweise abgeschaltet werden.

Damit wird auch klar, welche Fehler durch Zusicherungen gefunden werden sollen: logische Fehler und Programmierfehler. Auf keinen Fall können mit dem Vertragsmodell Eingabefehler des Anwenders abgefangen werden, schon allein deshalb nicht, weil in der Kundenversion die Überprüfungen meist deaktiviert sind.

Ein Nachteil von "Programmierung als Vertrag" ist offensichtlich. Da die Semantik eines Systems nicht vollständig spezifiziert werden kann, lassen sich auch nicht alle logischen Fehler aufdecken. Hinzu kommt, daß Testen in realen Systemen niemals vollständig erfolgen kann und daher nicht die Abwesenheit, sondern nur die Existenz von Fehlern zeigen kann. Trotzdem bietet das Vertragsmodell in der Praxis eine große Hilfe, auch wenn es anfänglich etwas Denk- und Arbeitsaufwand kostet, die Zusicherungen zu formulieren.

7.2 Delegation als Vertrag

Mit einer Delegationsbeziehung bietet ein Objekt seinen Kunden Merkmale eines anderen Objeks als die eigenen an. Dieses Angebot verpflichtet das Objekt aber gleichzeitig dazu, für ein entsprechendes Elternobjekt zu sorgen, das Aufrufe dieser Merkmale behandeln kann. Delegation kann also als ein impliziter Vertrag zwischen Kunde und Anbieter gesehen werden. Dieser Abschnitt behandelt Delegationsbeziehungen in diesem neuen Licht.

```
class X
delegate Y to del_ref
...
end -- class X
```

Die obige Deklaration einer Delegationsbeziehung impliziert, daß die Delegationsreferenz niemals `Void` sein darf (siehe Abschnitt 5.5.2). Dies stellt eine Verpflichtung für die Klasse `X` dar. Eiffel bietet mit dem Vertragsmodell eine erprobte Möglichkeit, diese Verpflichtung überprüfbar zu machen. Durch die Deklaration wird der Klasse *implizit* eine neue Klassen-Invariante hinzugefügt:

```
del_ref /= Void;
```

In der Testphase eines Systems wird eine Ausnahme ausgelöst, sobald nach Beendigung einer Methode (also auch nach der Initialisierungsmethode) die Delegationsreferenz `Void` ist. Während der Ausführung einer Methode stellt eine `Void`-Referenz hingegen keinen Fehler dar. Damit kann natürlich nicht garantiert werden, daß Delegationsreferenzen bei Benutzung niemals ungültig sind, die Sicherheit basiert auf Testverfahren. Ein guter Compiler darf also durchaus eine Fehlermeldung ausgeben, wenn er durch statische Analyse zu dem Ergebnis kommt, daß eine Delegationsreferenz auf `Void` gesetzt wird.

Die Aussage, daß eine Delegationsreferenz prinzipiell nie gültig sein muß (als `optional` deklarierte Delegationsbeziehung), scheint für Kunden ein schwieriges Hindernis für die Benutzung einer Klasse zu sein. In vielen Fällen kann der Entwickler einer Klasse seinen Kunden jedoch die Benutzung erleichtern, indem er spezifiziert, wann eine Referenz als gültig angenommen werden kann und wann nicht. Der beste Ort für solche Hinweise ist eine Klassen-Invariante:

```
zustand_x implies del_ref /= Void;
zustand_y implies del_ref = Void;
```

Die Ausdrücke `zustand_x` und `zustand_y` können mögliche Zustände eines Objeks auf einem höheren Abstraktionsniveau beschreiben. Dadurch muß ein Kunde sich nicht vor jedem Merkmalsaufruf davon überzeugen, ob Delegation möglich ist.

Dies ist auch der Grund dafür, daß bei der Deklaration einer Delegationsbeziehung das Merkmal angegeben wird, über das ein Elternobjekt später identifiziert wird. Ein Benutzer kann dadurch Hinweise in der Klassen-Invariante entsprechend deuten, auch wenn er auf die Delegationsreferenz selbst nicht zugreifen kann.

7.3 Zusicherungen aus Delegationsbeziehungen

Wenn Attribute der Elternklasse in einer Zusicherung auftauchen und gleichzeitig von einer Kindklasse redefiniert werden, so stellt sich die Frage, welche der physisch mehrfach vorhandenen Attribute für die Überprüfung der Zusicherung herangezogen werden sollen. Nach den in Abschnitt 6.2.5 durchgeführten Untersuchungen können dies nur die redefinierten Attribute in der Kindklasse sein. Das folgende Beispiel konstruiert einen solchen Fall:

```
class X
feature{ Y }
   n : integer;
   methode_1 is
   do
      -- Aufrufe zaehlen
      n := n + 1
   end

   methode_2 is
   require n > 0
   do
      -- tu etwas, das nur nach
      -- dem Aufruf von
      -- methode_1 moeglich ist
   end
end -- class X
```

```
class Y
delegate X to del_ref
   redefine n end
feature{ ANY }
   n       : integer;
   del_ref : X;

  methode_3 is
  do
     del_ref.methode_2
  end
end -- class Y
```

Ein Kunde, der `obj_y.methode_1` aufruft, bewirkt eine Veränderung von `obj_y.n`. Wenn er anschließend `obj_y.methode_2` aufruft, so geht dies in Ordnung, da `obj_y.n` = 1 ist und dieses Attribut aufgrund der Delegationsbeziehung für die Überprüfung der Vorbedingung benutzt wird. Der anschließende Aufruf von `methode_3` verursacht hingegen einen Fehler. In `methode_3` erfolgt der Aufruf von `X.methode_2` ohne Delegationssemantik. Daher wird für die Überprüfung der Vorbedingung `methode_2`in nicht `Y.n`, sondern `X.n` benutzt. Dieses Attribut ist jedoch gleich 0.

Die Überprüfung der Zusicherung `n` > `0` ist also davon abhängig, woher der Aufruf von `methode_2` erfolgt. Ohne Delegationsbeziehungen spielt es keine Rolle, über welche Referenz ein Aufrufer ein Merkmal aufruft.

Das Problem tritt noch stärker hervor, wenn man eine Klassen-Invariante in der Elternklasse annimmt, die sowohl durch Kindklasseen redefinierte, als auch in der Elternklasse verbliebene Merkmale zueinander in Beziehung setzt. Dann ist es möglich, daß die Invariante aus Sicht über die Delegationsbeziehung erfüllt ist, dasselbe Objekt, über eine gwöhnliche Referenz betrachtet, jedoch in einem inkonsistenten Zustand ist. Dieses Objekt kann außerhalb der Delegationsbeziehung nicht mehr benutzt werden, da jeder Merkmalsaufruf eine Vertragsverletzung darstellen würde. Der umgekehrte Fall ist natürlich auch möglich: auf ein bisher konsistentes Objekt wird über eine Delegationsbeziehung zugegriffen. Dann könnte das Objekt inkonsistent sein, da Zusicherungen sich nun auf in der Kindklasse redefinierten Merkmale beziehen und diese die Anforderungen der Invariante nicht erfüllen.

An dieser Stelle muß betont werden, daß sich dieses, zugegeben schwerwiegende Problem, nicht durch die Zusicherungen eingeschlichen hat, sondern auch vorher vorhanden war und hier nur aufgedeckt worden ist. Eine Delegationsbeziehung zwischen zwei Objekten ist also nicht so locker wie eine Kunden-Lieferanten-Beziehung. Sie hat hier eher den Charakter einer Vererbungsbeziehung. Man kann eine Klasse in der Vererbungshierarchie auch nicht ohne weiteres an eine andere Stelle setzen.

Wie läßt sich diesem Problem begegnen? Leider nicht durch entsprechende Vorkehrungen in der Sprache selbst. Nur eine vorsichtige und bedachte Entwicklung der Kindklassen kann Probleme dieser Art vermeiden. Sprachbasierte Sicherungen sind nur mit Verzicht auf Redefinitionen zu realisieren.

7.4 Der unerfüllbare Vertrag

Durch die Einführung von Delegation schleicht sich ein weiteres Problem ein. Es besteht die Möglichkeit, daß Ausnahmen aufgrund von Vertragsverletzungen auftreten, obwohl alle Klassen ihren Vertrag korrekt erfüllen. Im folgenden Beispiel wird so ein *unerfüllbarer* Vertrag geschlossen.

Wenn eine Klasse übersetzt werden soll, müssen dem Compiler alle Superklassen bekannt sein. Neben den in der Klasse selbst festgelegten Zusicherungen sind Superklassen die einzigen Quellen für

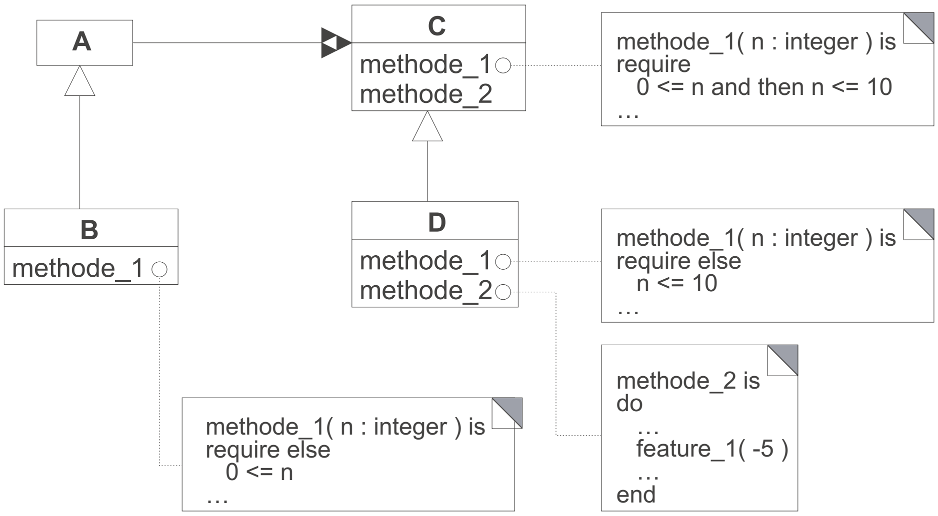

Abbildung 7.1: Dieses Szenario führt zu einer unvermeidbaren Zusicherungs-Verletzung.

vertragliche Verpflichtungen, die die Merkmale einer Klasse einzuhalten haben. Daher kann sich ein Entwickler darauf verlassen, daß er bei Beachtung aller geerbten und eigenen Zusicherungen keinen Code schreibt, der zur Laufzeit zu Vertragsverletzungen führt.

Eine Delegationsbeziehung bedeutet, daß mehrere Objekte zur Erfüllung eines Vertrags zusammenarbeiten. Im Unterschied zu reinen Vererbungshierarchien kann über die Reihenfolge der Entwicklung einzelner Klassen im Allgemeinen keine Aussage gemacht werden. In dem in Abbildung 7.1 dargestellten Beispiel muß `C` vor `A` und `D` entwickelt werden. `A` muß wiederum bei der Entwicklung von `B` zur Verfügung stehen. Über die Entwicklungsreihenfolge von `B` und `D` kann keine Aussage gemacht werden.

Angenommen, `C.methode_1` stellt die folgende Anforderung:

```
methode_1( n : integer ) is
require 0 <= n and then n <= 10;
   ....
end
```

Dann hat `B` die Möglichkeit, die von `C` über A geerbte Methode `methode_1` zu redefinieren und die Vorbedingung zu lockern. Beispielsweise könnte die Vorbedingung in `B` lauten: `0 <= n`. Genauso könnte `D` `methode_1` redefinieren und als Vorbedingung `n < 10` angeben. Beide Redefinitionen erweitern die Menge der möglichen Werte von `n`, sind also korrekte Vertragsänderungen. In `D` wird auch `methode_2` redefiniert. In der neuen Implementierung wird `methode_1` wird mit `n = -5` aufgerufen, was den Vertrag von `D.methode_1` erfüllt.

Ein Fehler tritt nun auf, wenn `obj_b`, eine Instanz von `B`, in der von `A` geerbten Delegationsreferenz auf eine Instanz von `D` verweist. Ein Kunde könnte `obj_b.methode_2` aufrufen. Der Aufruf würde an `D.methode_2` delegiert. Innerhalb dieser Methode erfolgt der Aufruf `Current.methode_1( -5 )`. Da `Current` nach wie vor auf `obj_b` zeigt, wird `B.methode_1` vom Laufzeitsystem ausgewählt. Für diese Implementierung führt der aktuelle Wert von `n` zu einer Vertragsverletzung.

Dieses Beispiel ist konstruiert und es ist fraglich, ob die Situation in einem realistischen System auftreten wird. Der Lerneffekt ist jedoch, daß Delegationsbeziehungen Risiken in ein System bringen. Bei reinen Vererbungsbeziehungen kann dieser Fehler nicht auftreten, da die Reihenfolge der Entwicklung von Klassen vorgegeben ist. In Verbindung mit Delegation ist die Änderung von Verträgen also noch kritischer als bei reinen Vererbungsbeziehungen. Mehr denn je gilt also, daß eine Redefinition die Implementierung einer Methode ändert aber nicht ihre Semantik ändern darf.

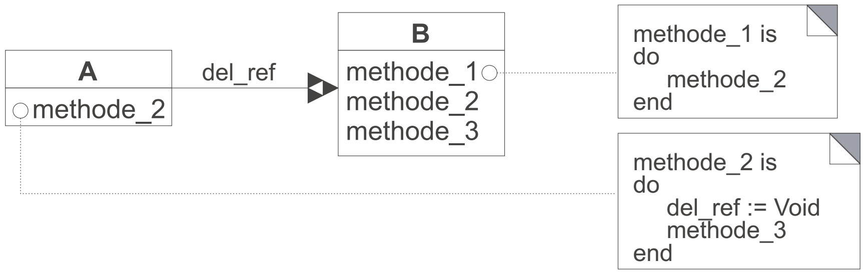

Abbildung 7.2: Da Delegationsreferenzen im Verlauf eines Merkmalsaufrufs `Void` werden können, muß der Delegationspfad eingefroren werden.

7.5 Das Einfrieren des Delegationspfades

Eine Delegationsreferenz identifiziert zur Laufzeit ein Elternobjekt. Wenn eine Methode des Elternobjekts Merkmale von `Current` aufruft, beginnt die Suche nach diesen Merkmalen beim Kindobjekt. Dies war eine der grundlegenden Eigenschaften von Delegation. Bisher wurde noch nicht erläutert, wie diese Suche konzeptionell erfolgen muß, damit das gewünschte Verhalten erzielt werden kann. Der Grund ist, daß die bisherigen Überlegungen zum Vertragsmodell in die Erläuterung mit einfließen müssen. Es wird sich herausstellen, daß die Suche nach der richtigen Methode *nicht* mit Hilfe der Delegationsreferenz erfolgen kann, sondern daß der Delegationspfad bei der Weiterleitung eines Aufrufs von einem Kindobjekt zu seinem Elternobjekt eingefroren werden muß.

In Abbildung 7.2 delegiert eine Klasse `A` an eine Klasse `B`. Die Delegationsbeziehung ist nicht optional. Daher besteht für `A` die Verpflichtung, immer für eine gültige Delegationsreferenz zu sorgen (siehe Abschnitte 5.5.2 und 7.2). Die implizite Klassen-Invariante `del_ref /= Void` muß vor und nach jedem Methodenaufruf gelten. Es wurde jedoch bereits gesagt, daß `Void`-Referenzen während der Ausführung einer Methode erlaubt sind. Wenn nun ein Kunde das Merkmal `A.methode_1` aufruft, wird der Aufruf an eine Instanz von `B` delegiert. Innerhalb von `B.methode_1` wird `A.methode_2` aufgerufen, da `A` die von `B` geerbte Methode redefiniert hat. Die Delegationsreferenz wird auf `Void` gesetzt.

Diese Anweisung sieht etwas plump aus, sie dient jedoch nur zur Vereinfachung. In der Praxis könnten eine ganze Reihe von Anweisungen und weiteren Merkmalsaufrufen zu diesem Ergebnis führen, sodaß ein Compiler vielleicht keine Warnung ausgeben kann. Trotzdem ist die `Void`-Referenz kein Fehler, es muß nur irgendwie sichergestellt werden, daß bei der Beendigung von `methode_1` die Referenz wieder auf ein gültiges Objekt verweist. Der Aufruf von `methode_3` innerhalb von `A.methode_2` muß wieder an `B` delegiert werden, da in `A` keine passende Methode existiert. Dies ist aber nicht möglich, da `del_ref` zur Zeit kein Elternobjekt identifiziert.

Wenn die `Void`-Referenz kein Fehler ist, darf sie auch nicht zu einem Fehler führen. Daher wird der Delegationspfad, d.h. die Verbindungen zwischen einem Kindobjekt und seinen direkten und indirekten Elternobjekten eingefroren. Wenn ein Merkmalsaufruf delegiert wird, wird die Delegationsreferenz nur bei der erstmaligen Ermittlung des Elternobjekts ausgewertet. Anschließend wird eine temporäre, d.h. nur für diesen Aufruf gültige Kopie erzeugt. Der Zugriff auf das Elternobjekt muß bis zur entgültigen Beendigung des Aufrufs immer über diese Kopie erfolgen. Der Austausch des Elternobjekts im Rahmen eines delegierten Aufrufs hat daher auch erst nach Beendigung dieses Aufrufs Auswirkungen.

7.6 Zusammenfassung

Zusicherungen sind hilfreiche Werkzeuge zum schnelleren Auffinden von Fehlern. Außerdem unterstüzen sie bei der Spezifizierung und Dokumentation von Systemen. Da die Verbindung von Vererbung und Delegation die Komplexität eines Systems erhöht, sind Zusicherungen gerade in E+ wichtig.

Delegationsbeziehungen lassen sich hervorragend in das Vertragsmodell von Eiffel integrieren. Delegation als Vertrag aufzufassen, bietet dem Anwender die Möglichkeit, den neuen Sprachkonstrukten und der gestiegenen Komplexität mit bekannten Hilfsmitteln zu begegnen. Es hat sich aber gezeigt, daß durch Delegationsbeziehungen Probleme auftreten, die eine unkomplizierte Nutzung der neuen Mechanismen praktisch unmöglich machen. Die Kapselung von Objekten wird, obwohl Delegation eine geschlossene Beziehung ist, verletzt. Objekte können inkonsistent werden. Diese Probleme sind nicht E+-spezifisch. Sie treten auch in delegationsbasierten Sprachen wie SELF auf. Damit sich ein System wie erwartet verhalten kann, muß der Delegationspfad eingefroren werden.

Es muß noch einmal betont werden, daß die geschilderten Probleme durch das Vertragsmodell nicht eingeführt, sondern lediglich entdeckt worden sind. Insofern hat sich das "Programmierung als Vertrag" schon vor dem ersten echten E+-Programm als sehr nützlich erwiesen.

Kapitel 8

Weitere Aspekte der Spracherweiterung

8.1 Weiterleiten ohne Delegation

Die bisher betrachteten Delegationsbeziehungen wiesen zwei wesentliche Merkmale auf:

1. Merkmalsaufrufe werden von einem Objekt automatisch an ein anderes Objekt weitergeleitet.
2. `Current` zeigt auch nach der Weiterleitung auf den ursprünglichen Empfänger der Nachricht.

Offensichtlich ist das erste Merkmal Voraussetzung für das zweite. Umgekehrt besteht diese Abhängigkeit nicht. Es können Merkmalsaufrufe von einem Objekt an ein anderes weitergeleitet werden, wobei `Current` mitwandert. Die Semantik einer Delegationsbeziehung geht dadurch verloren. Das Elternobjekt kann nicht mehr feststellen, ob eine Nachricht von einem normalen Benutzer oder von einem Kindobjekt stammt. Echte Delegationsbeziehungen führen, das wurde in den bisherigen Kapiteln deutlich, zu sehr komplexen Inter–Objekt–Beziehungen. Unter Umständen können Objekte sogar in einen inkonsistenten Zustand gebracht werden.

Für viele Anwendungen von Delegation ist die automatische Weiterleitung von Merkmalsaufrufen der wesentliche Aspekt. Die Möglichkeit zur dynamischen Erweiterbarkeit eines Objeks muß nicht zwangsläufig einhergehen mit der Redefinition von Merkmalen der Elternklasse. Eine solche Beziehung zwischen Objekten könnte man als *"Delegation light"* bezeichnen. In E+ soll es dafür ein eigenes Sprachkonstrukt geben:

```
class ELTERNKLASSE
   ...
end -- class ELTERNKLASSE
```

```
class KINDKLASSE
   forward ELTERNKLASSE to del_ref

feature
   del_ref : ELTERNKLASSE
   ...
end -- class KINDKLASSE
```

Anstelle von `forward ELTERNKLASSE to del_ref` könnte auch geschrieben werden

```
forward ELTERNKLASSE optional to del\_ref
```

Bezüglich `Void`–Referenzen gilt für eine `forward`–Beziehung das in Abschnitt 5.5.2 für Delegationsbeziehungen gesagte. Auch der Zugriff auf Merkmale des Elternobjekts unterliegt den gleichen Regeln wie für Delegationsbeziehungen (siehe Abschnitt 5.3).

Der Deklaration einer `forward`–Beziehung kann natürlich kein `redefine`–Abschnitt folgen, da Redefinitionen keinen Sinn ergeben wenn `Current` mitwandert. Die beiden anderen Abschnitte, die einer `delegate`–Anweisung folgen können sind `export` und `rename`. Sie wären auch für `forward`-Beziehungen denkbar. Sind sie auch sinnvoll?

Der Exportstatus für geerbte Merkmale kann in einer Delegationsbeziehung eingeschränkt werden. Wie in Abschnitt 5.3.4 gezeigt wird, stellt dies eine mögliche Fehlerquelle dar. Wenn eine einfache Weiterleitung mittels `forward` gewünscht wird, sollten Risiken vermieden werden. Eine Kindklasse darf in einer `forward`–Beziehung den Exportstatus von geerbten Merkmalen nicht ändern.

In Abschnitt 6.1 wurde die Umbenennung von geerbten Merkmalen behandelt. Es hat sich gezeigt, daß dieses Konstrukt keine ungewollten Begleiterscheinungen hat. Es könnte also auch für eine `forward`–Beziehung genutzt werden, ohne doch noch mögliche Fehlerquellen einzubauen. Trotzdem soll die Umbenennung geerbter Elemente in einer `forward`–Beziehung nicht möglich sein. Diese Beziehung soll ihre einfache Semantik behalten, Umbenennungen können nach wie vor mit Delegationsbeziehungen realisiert werden.

Eine `forward`–Beziehung kann natürlich durch eine Delegationsbeziehung simuliert werden. Ein Entwickler muß lediglich auf einige Möglichkeiten, die Delegation bietet, verzichten. Trotzdem bietet die `forward`–Beziehung den Vorteil, daß sie beim ersten Lesen eines Quelltextes sofort "Entwarnung" gibt; der Leser kann sich sofort auf die schwierigen Konstrukte konzentrieren.

8.2 Generische Klassen

Häufig werden für die Implementierung eines Software-Systems Container benötigt, d.h. Objekte, die eine Menge anderer Objekte verwalten und den effizienten Zugriff auf diese Elemente erlauben. In Eiffel und vielen anderen klassenbasierten objektorientierten Programmiersprachen stehen daher vorgefertigte Containerklassen als Teil einer Standardbibliothek zur Verfügung. Von Containerklassen wird häufig gefordert, daß sie Elemente eines bestimmten Typs (z.B. eine Liste von `Person`-Objekten) verwalten. Die Algorithmen der Containerklasse selbst sind jedoch meist unabhängig vom Typ der Elemente. Die Algorithmen für eine verkettete Liste ändern sich nicht, wenn statt `Person`-Objekten `Schiff`-Objekte verwaltet werden sollen.

Einige Programmiersprachen bieten zur Realisierung von Containerklassen *generische Klassen* an. Das folgende Beispiel zeigt, wie eine generische Klasse für eine Liste deklariert werden kann, und wie ein Kunde damit eine Liste von `PERSON`-Objekten verwalten kann:

```
class LIST[ T ]
...
feature{ ANY }
   item( index : integer ) : T is ...
   head : T is ...
   count : integer is ...
   ...
end -- class LIST
```

```
class KUNDE
feature
   personen_liste:LIST[ PERSON ]
   ...
end -- class KUNDE
```

Der *Typparameter* `T` ermöglicht die Deklaration von Merkmalen, ohne ihren genauen Typ bei der Entwicklung der Klasse `LIST` angeben zu müssen. `T` kann für jeden beliebigen Typ stehen und wie ein echter Typ benutzt werden. Die Klasse `LIST` ist nicht direkt instanziierbar. Bei der Benutzung der Klasse muß für `T` ein konkreter Typ eingesetzt werden.

Das Einsetzen eines konkreten Typs für einen Typparameter wird als Instanziierung bezeichnet, d.h. eine Instanz einer generischen Klasse ist eine Klasse, von der Objekte instanziiert werden können. Bedenkt man, daß nach einmaligem Schreiben der Klasse `LIST` Listen von beliebigen Objekten zur Verfügung stehen, bieten generische Klassen eine gute Möglichkeit zur Wiederverwendung [Has95].

8.2.1 Generezität und Vererbung

Generezität und Vererbung sind orthogonale Konzepte. Sie beeinflussen sich gegenseitig nicht. Nach [Mey89] ist objektorientierte Programmierung ohne Generezität nur eingeschränkt möglich. Generische Klassen bieten die Möglichkeit, Mengen von Objekten typsicher zu benutzen. Wenn ein Merkmal als `LIST[ PERSON ]` deklariert wurde, sind darin zu jedem Zeitpunkt nur Instanzen von Klassen enthalten, die zu `PERSON` konform sind, also Instanzen der Klasse `PERSON` oder einer ihrer Subklassen.

Generezität läßt sich durch Vererbung und covarianter Redefinition simulieren [Mey97, Anhang B]. Die Benutzung einer generischen Klasse durch Einsetzen eines Typs ist für einen Entwickler jedoch einfacher als die Entwicklung einer neuen Unterklasse, weshalb die Simulation von Generezität nur konzeptionell aber nicht praktisch interessant ist.

Jede Klasse kann eine generische Klasse als Oberklasse haben. Sie muß dann die Typparameter durch konkrete Typen ersetzen. Dies können konkrete Typen, also in Eiffel Klassen sein, oder, wenn die Unterklasse selbst generisch ist, wiederum Typvariablen sein. In der Standardbibliothek von Eiffel findet sich ein Beipiel hierfür. Eine generische Klasse `LIST[ T ]` stellt eine abstrakte Schnittstelle für allgemeine Listen bereit. Eine Klasse `LINKED_LIST[ T ]` implementiert diese Schnittstelle durch eine verkettete Liste. Beide Klassen könnten etwas so aussehen:

```
deferred class LIST[ T ]
feature{ ANY }
   item( index : integer ) : T is
   deferred
   end
   ...
end -- class LIST
```

```
class LINKED_LIST[ G ]
inherit LIST[ G ]
   redefine index end
feature{ ANY }
   item( index : integer ) : G is
   do
      ...
   end
   ...
end -- class LINKED_LIST
```

Zusätzlich bietet Eiffel *eingeschränkte Generezität* [Mey97, Kapitel 16.4], d.h. eine generische Klasse akzeptiert als Typparameter nicht jeden beliebigen Typ, sondern schränkt die Menge der möglichen Typparameter ein. Dadurch kann die generische Klasse Annahmen über die Schnittstellen der einzusetzenden Typen machen. Uneingeschränkte Generezität ist konzeptionell betrachtet ein Spezialfall der eingeschränkten Generezität, es dürfen nur zu `ANY` konforme Typen als Parameter eingesetzt werden.

8.2.2 Generezität und Delegation

Die Delegation an eine generische Klasse erfolgt analog zum Erben von einer generischen Klasse. Die Typparameter der Elternklasse müssen eingesetzt werden. Wenn die Kindklasse selbst eine generische Klasse ist, können auch Typparameter der Kindklasse eingesetzt werden. Im folgenden Beispiel delegiert die Klasse `K1` an eine verkettete Liste von Zahlen (`integer`), die Klasse `K2` ist selbst generisch und reicht ihren Typparameter an `LINKED_LIST` weiter:

```
class LINKED_LIST[ T ]
   ...
end -- class LINKED_LIST

class K1
delegate LINKED_LIST[ integer ] to p
creation make
feature{ ANY }
   make is
   do
      !!p
   end

   p : LINKED_LIST[ integer ]
   ...
end -- class K1
```

```
class K2[ G ]
delegate LINKED_LIST[ G ] to p
creation make
feature{ ANY }
   make ist
   do
      !!p
   end

   p : LINKED_LIST[ G ]
   ...
end -- class K2
```

Generezität, auch eingeschränkte Generezität, die hier nicht weiter betrachtet wurde, fördern in Verbingung mit Delegation keine Überraschungen zu Tage. Generische Klassen können sowohl als Elternklassen, als auch als Kindklassen verwendet werden. Es gilt im wesentlichen das über Generezität und Vererbung (Abschnitt 8.2.1) gesagte.

8.3 Nebenläufigkeit und Persistenz

Eiffel unterstützt sowohl Nebenläufigkeit als auch Persistenz durch Sprachkonstrukte, bzw. durch entsprechende Klassen in der Standardbibliothek. In dieser Diplomarbeit werden beide Themen jedoch nicht behandelt. Nebenläufigkeit in objektorientierten Systemen ist nicht leicht zu handhaben und ein aktuelles Forschungsgebiet [Pap95]. Für Delegationsbeziehungen kommen dadurch neue Aspekte hinzu. Beispielsweise könnte ein Ausführungsfaden eine Delegationsreferenz verändern, während ein anderer gerade über diese Referenz delegiert. Eiffel unterstützt persistente Objekte in der Form, daß ein Objekt und alle von ihm über Referenzen erreichbaren Objekte, in eine Datei geschrieben und später wieder hergestellt werden können. Die Persistenz von Objekten ist aber ein interdisziplinäres Gebiet und erfordert auch eine Beschäftigung mit Datenbanken, insbesondere objektorientierten Datenbanken [Heu97].

Kapitel 9

Evaluierung

In diesem Kapitel soll E+ seinen praktischen Nutzen unter Beweis stellen. Es soll zum einen festgestellt werden, ob Delegationsbeziehungen die Implementierung eines Software-Systems vereinfachen. Zum anderen soll auch die Spracherweiterung an sich kritisch betrachtet werden. Da bei der Entwicklung von E+ (siehe Kapitel 4 – 8) neue Konstrukte ohne Berücksichtigung ihrer praktischen Anwendbarkeit in Eiffel integriert wurden, muß sich erst noch zeigen, ob alle diese Konstrukte praktisch anwendbar sind, oder ob sie vielleicht nie gebraucht werden.

Für fundierte, umfassende Aussagen über die Praxistauglichkeit wäre sicherlich die Implementierung realer, bereits implementierter Software-Systeme erforderlich. Ein Vergleich der bisherigen mit der neuen Implementierung könnte dann qualitative und quantitative Fakten über den Nutzen von Delegation in objektorientierten Programmiersprachen liefern. Ein solches Vorhaben ist im Rahmen einer Diplomarbeit jedoch kaum zu bewältigen. Stattdessen soll die Evaluierung durch die Implementierung[1] von Entwurfsmustern nach [GJHV95] geschehen. Im folgenden Abschnitt wird kurz begründet, warum diese sich für einen Test der neuen Sprache besonders eignen.

9.1 Entwurfsmuster zum Tesen von E+

In Abschnitt 2.3 wurden Entwurfsmuster bereits vorgestellt. An einem konkreten Beispiel wurde gezeigt, wie die Dekomposition von Funktionalität in mehrere Objekte erfolgen kann und wie dadurch mehr Flexibilität und eine bessere Erweiterbarkeit eines Entwurfs erzielt werden kann.

Neben der Dokumentation von bestehenden Architekturen ermöglichen Entwurfsmuster auch den schnelleren Entwurf neuer Systeme. In [HZ97] wird gezeigt, daß durch den bewußten Einsatz von Entwurfsmustern die schnelle Entwicklung eines flexiblen Software-Systems möglich ist. Durch die Entwurfsmuster werden bereits Teile der Architektur vorgegeben, so daß sich ein Entwickler darum nicht mehr zu kümmern braucht. Die in [GJHV95] angegebenen Beispiel-Implementierungen können als Grundlage für die Implementierung eines neuen Systems herangezogen werden. Oft ist nur eine Anpassung der Bezeichner-Namen an die Terminologie des Anwendungsbereichs erforderlich.

Der praktische Wert der Evaluierung basiert darauf, daß Entwurfsmuster aus real existierenden Systemen extrahiert wurden. Wenn Entwurfsmuster Schlüsselkonstrukte in existierenden Architekturen sind, führt eine vereinfachte Implementierung von Entwurfsmustern wahrscheinlich zu einer vereinfachten Implementierung realer Anwendungen.

Die Beispiel-Implementierungen liegen meist in C++ vor. Sie werden, sofern es für das Verständnis notwendig ist, noch einmal ganz oder teilweise in Eiffel wiedergegeben. Sie werden daraufhin untersucht,

[1] Diese Formulierung ist eigentlich nicht korrekt. Entwurfsmuster beschreiben Strukturen und Verhalten unabhängig von einer konkreten Anwendung. Sie lassen sich nicht direkt implementieren. Stattdessen ist die Implementierung von Beispielszenarien gemeint, die Entwurfsmuster enthalten.

ob aufgrund der fehlen Unterstützung von Delegation durch die Programmiersprache Anweisungen zur Simulation von Delegationsbeziehungen enthalten sind. Anschließend wird versucht, diese Simulation durch Sprachkonstrukte von E+ zu vermeiden. Es muß sich dann zeigen, ob die Implementierung eines Entwurfs geradliniger erfolgen kann und ob der entstandene Quelltext signifikant einfacher und kürzer geworden ist. Die in E+ gegebene Gefahr von Laufzeitfehlern soll explizit berücksichtigt werden.

Entwurfsmuster stellen ein aktuellen Forschungsgebiet der Informatik dar. Neben [GJHV95] gibt es weitere Literatur zu diesem Thema, z.B. [CS95, CVK96]. Im folgenden sollen mit *Entwurfsmustern* jedoch nur die in [GJHV95] definierten und behandelten gemeint sein. Ob Aussagen auch auf Entwurfsmuster aus anderen Quellen übertragbar sind, wird hier nicht untersucht. Nicht alle in Entwurfsmuster machen Gebrauch von Delegation. Für die Evaluierung werden natürlich solche Muster betrachtet werden, bei denen Delegation ein wesentliches Merkmal ist.

9.2 Das Muster Kette von Verantwortlichkeiten

Das Entwurfsmuster *Kette von Verantwortlichkeiten* wird in [GJHV95] wie folgt definiert:

> Einem Objekt in einer Hierarchie wird eine Nachricht geschickt. Das Objekt kann diese entweder selbst bearbeiten oder sie an das Elternobjekt in der Hierarchie weiterleiten, das Elternobjekt verfährt auf die gleiche Weise. Spätestens in einem Wurzelobjekt muß Nachricht bearbeitet werden können.

Diese Beschreibung liest sich fast wie die Definition von Delegation. In [GJHV95, Seite 223 ff.] wird folgendes Beispiel konstruiert: Eine Anwendung besteht aus mehreren Dialogen, die wiederum Bedienelemente enthalten, mit denen der Anwender den Dialog steuern kann. Zu jedem Bedienelement kann er sich einen Hilfetext anzeigen lassen. Existiert zu einem Bedienelement kein spezifischer Hilfetext, soll der Hilfetext des umschließenden Objeks, z.B. des Dialogs angezeigt werden. Diese Kette kann bis zum Wurzelobjekt, der Anwendung selbst, fortgesetzt werden, die dann einen allgemeinen Hilfetext anzeigt.

9.2.1 Das Muster in Eiffel implementiert

Für die Implementierung in einer objektorientierten Programmiersprache könnte der in Abbildung 9.1 dargestellte Entwurf genutzt werden. Die Klassen `ANWENDUNG`, `DIALOG` und `SCHALTFLAECHE` erben von einer Oberklasse `HILFE_MANAGER`. Diese Klasse verwaltet die Weiterleitung einer Hilfe-Anforderung entlang einer Kette von Objekten. Objekte abgeleiteter Klassen brauchen lediglich in ihrer Initialisierungsmethode für einen Hilfetext zu sorgen, falls dies gewünscht wird. In Eiffel könnte dieser Entwurf wie folgt implementiert werden:

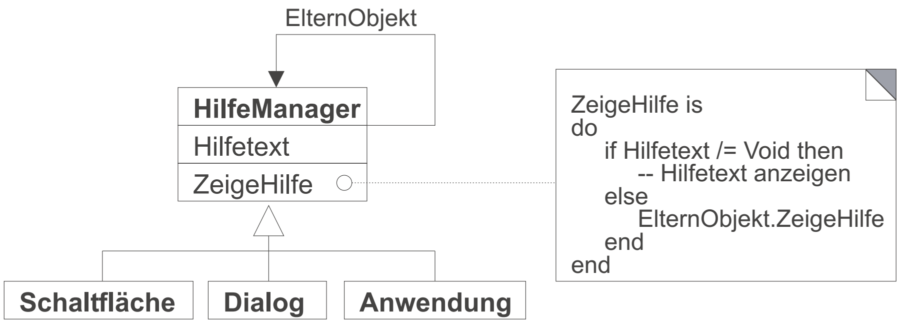

Abbildung 9.1: Das Entwurfsmuster *Kette von Verantwortlichkeiten*

```
class HILFE_MANAGER
feature{ ANY }
   zeige_hilfe is
   do
      if help_text /= void then
         ausgabe( hilfe_text )
      else
         eltern_obj.zeige_hilfe
      end
   end

feature{ NONE }
   hilfe_text : STRING
   eltern_obj : HILFE_MANAGER

end - class HILFE_MANAGER
```

```
class DIALOG
inherit HILFE_MANAGER
creation init

feature{ ANY }
   init is do
      -- setze Attribut hilfe_text
      -- fuer dieses Objekt
   end
end -- class DIALOG
```

Durch Redefinition von `zeige_hilfe` kann eine Unterklasse das Verhalten anpassen, z.B. den Hilfetext aus einer Datei oder Datenbank auslesen.

9.2.2 Das Muster in E+ implementiert

Um dieses Muster angemessen Modellieren zu können, sind zunächst einige Vorüberlegungen anzustellen. Der Anwender kann sich zu einem Bedienelement (Schaltfläche, Dialog, usw.) einen Hilfetext anzeigen lassen. Nicht für alle Bedienelemente existiert jedoch ein individueller Hilfetext. In diesem Fall wird die Frage nach einem Hilfetext an das umschließende Bedienelement weitergeleitet. Da die Anwendung als alles umschließendes Element einen allgemeinen Hilfetext ausgibt (z.B. "Diesem Eintrag ist kein Hilfethema zugeordent", siehe Windows 95), erhält der Anwender in jedem Fall Feedback auf seine Aktion. Ob ein Bedienelement, also ein bestimmtes Objekt, einen individuellen Hilfetext hat, entscheidet sich unabhängig von der Klasse, deren Instanz ein Objekt ist. Einige Schaltflächen haben z.B. einen individuellen Hilfetext, andere nicht. In diesem Fall hat ein Objekt, unabhängig von seiner Klasse, ein bestimmtes Attribut (`hilfe_text`) oder nicht. In delegationsbasierten Programmiersprachen ist dies kein Problem. In objektorientierten Programmiersprachen können Instanzen der selben Klasse natürlich nicht unterschiedliche Attribute haben, weshalb die Implementierung in Abschnitt 9.2.1 für alle Objekte einen Hilfetext vorsieht, der unter Umständen leer ist.

Auch in E+ haben alle Instanzen einer Klasse die gleichen Attribute. Es gibt jedoch einen Ansatzpunkt, mit dem das gewünschte Verhalten simuliert werden kann. Da Attribute meist Referenzen auf Objekte sind, kann eine `Void`-Referenz dazu "mißbraucht" werden, die Abwesenheit eines bestimmten Attributs anzuzeigen. In Abbildung 9.2 delegieren die Klassen der Bedienelemente optional an einen Hilfetext.

Abbildung 9.2: Mit optionaler Delegation lassen sich unterschiedliche Attribute für Instanzen der selben Klasse simulieren.

Die Wurzelklasse `ANWENDUNG` delegiert nicht optional. Wenn die Delegationsreferenz, also das Attribut `hilfe_text`, `Void` ist, bedeutet dies, daß das Bedienelement keinen individuellen Hilfetext hat.

Jede Bedienelemente-Klasse (Ausnahme: `ANWENDUNG`, die Wurzelklasse) delegiert zusätzlich an die Klasse, in deren Instanzen die eigenen Instanzen zur Laufzeit eingebettet sind. Eine Schaltfläche ist Bestandteil eines Dialogs, folglich delegiert `SCHALTFLAECHE` an `DIALOG`. Die Methode `zeige_hilfe` wird in der Wurzelklasse `ANWENDUNG` implementiert. Eine Implementierung in E+ könnte wie folgt aussehen:

```
class SCHALTFLAECHE
delegate
   DIALOG optional to eltern_dialog
      redefine hilfe_text
   end
   HILFETEXT optional to hilfe_text
      select hilfe_text.text then
             eltern_dialog.text
   end
feature{ ANY }
   ...
feature{ NONE }
   button_hilfe  : HILFETEXT
   eltern_dialog : DIALOG

end - class SCHALTFLAECHE

class HILFETEXT
feature{ ANY }
   text : STRING
end -- class HILFETEXT
```

```
class DIALOG
delegate
   ANWENDUNG to eltern_anwendung
      redefine hilfe_text
   end
   HILFETEXT optional to hilfe_text
      select hilfe_text.text then
             eltern_anwendung.text
   end
feature{ ANY }
   ...
feature{ NONE }
   hilfe_text       : HILFETEXT
   eltern_anwendung : ANWENDUNG

end - class SCHALTFLAECHE

class ANWENDUNG
delegate HILFETEXT to hilfe_text
feature{ ANY }
   zeige_hilfe is
   do
      anzeigen( Current.hilfe_text )
   end
feature{ NONE }
   hilfe_text : HILFETEXT

end -- class ANWENDUNG
```

Dieser Quelltext ist sicherlich das komplexeste, bisher in dieser Diplomarbeit vorgestellte Beispiel. Zunächst sollen die Konstrukte in den einzelnen Klassen daher kurz erläutert werden. Zusätzlich zu den im folgenden gemachten Aussagen sei noch darauf hingewiesen, daß alle Klassen bis auf `HILFETEXT` Initialisierungsmethoden benötigen (siehe Abschnitt 5.1.4), um die Elternobjekte und Hilfetexte korrekt setzen zu können. Um das Beispiel nicht noch komplizierter zu machen, wurden sie nicht explizit angegeben.

`HILFETEXT`
: Diese Klasse enthält den Hilfetext, der zu einem bestimmten Bedienelement angezeigt werden soll. Sie ist unkompliziert.

`ANWENDUNG`
: Die Klasse delegiert nicht optional an `HILFETEXT`. Dabei dient `hilfe_text` als Delegationsreferenz. In Instanzen von `ANWENDUNG` enthalten die Attribute `hilfe_text` und (von `HILFETEXT` geerbt) `text`.

`DIALOG`
: Die Klasse delegiert nicht optional an `ANWENDUNG` und optional an `HILFETEXT`. Aus der ersten Beziehung wird das Attribut `hilfe_text` geerbt und redefiniert. Dadurch kann ein Dialog-Objekt einen eigenen Hilfetext enthalten. Würde keine Redefinition stattfinden, würde `hilfe_text`, aufgrund der Delegationsbeziehung, auf den Hilfetext der Anwendung verweisen. Da `ANWENDUNG` das Attribut `text` von `HILFETEXT` erbt und `DIALOG` von `ANWENDUNG` und von `HILFETEXT` über Delegationsbeziehungen erbt, ist das Attribut `text` in `DIALOG` zweimal vorhanden. Durch die `select`-Anweisung wird das von `HILFETEXT` geerbte Attribut als bevorzugtes gewählt und für den Fall, das dieses nicht verfügbar ist, das von `ANWENDUNG` geerbte Attribut angegeben (siehe dazu Abschnitt 6.3).

`SCHALTFLAECHE`
: Die Klasse steht zu `DIALOG` in der gleichen Beziehung wie `DIALOG` zu `ANWENDUNG`. Die für `DIALOG` gemachten Aussagen gelten daher analog für `SCHALTFLAECHE`.

Um zu zeigen, daß diese Implementierung das gewünschte Verhalten hat, muß Schritt für Schritt überlegt werden, was bei einer Hilfe-Anfrage durch den Anwender geschieht. Angenommen, zu einer Schaltfläche soll ein Hilfetext angezeigt werden. Dies geschieht durch den Aufruf `eine_schaltflaeche.zeige_hilfe`. In `SCHALTFLAECHE` existiert eine solche Methode jedoch nicht. Dennoch ist der Aufruf korrekt, da `SCHALTFLAECHE` indirekt an `ANWENDUNG` delegiert, wo `zeige_hilfe` implementiert ist. Der Aufruf wird daher also an `ANWENDUNG.zeige_hilfe` delegiert. `Current` zeigt jedoch nach wie vor auf die Schaltfläche. Für den Aufruf `Current.text` in dieser Methode wird daher versucht, `SCHALTFLAECHE.text` zu benutzen. Wenn dieses vorhanden ist, die Schaltfläche also einen individuellen Hilfetext hat, wird es benutzt, sonst wird durch die `select`-Anweisung auf das Attribut von `DIALOG` zugegriffen. Dort wird dann das gleiche Verfahren angewendet und eventuell zu `ANWENDUNG` weitergegangen, wo mit Sicherheit ein Attribut `text` vorhanden ist. Das Verfahren stellt also sicher, daß dem Anwender der korrekte Hilfetext präsentiert wird.

9.2.3 Vor- und Nachteile der E+ Implementierung

Das Beispiel erscheint auf den ersten Blick sehr kompliziert und die elegante und einfache Implementierung von `zeige_hilfe` ändert daran nichts. Dieser Eindruck entsteht aber teilweise dadurch, daß Delegation ein in der objektorientierten Softwareentwicklung ungewöhnliches Kontrukt ist. Wenn man längere Zeit mit Delegation gearbeitet hat, werden die Zusammenhänge vielleicht intuitiv klarer.

In der ersten Implementierung (ohne E+) haben alle Objekte ein Attribut `hilfe_text`. Eine Methode muß feststellen, ob dieses Text enthält und in Abhängigkeit von diesem Ergebnis fortfahren. Die zweite Implementierung ist natürlicher. Sie erlaubt es, natürlich nur simuliert, bestimmten Objekten, unabhängig von ihrer Klasse, ein bestimmtes Attribut zu haben oder nicht. Dieser Nachteil von klassenbasierter Vererbung wird in [Smi95] aufgegriffen und ausführlich diskutiert. Es wird begründet, warum zur Programmierung von Benutzungsoberfläche allgemein delegationsbasierte Sprachen besser geeignet sind. Mit E+ können diese Nachteile teilweise ausgeglichen werden.

9.3 Das Muster Zustand

Das Entwurfsmuster Zustand wurde bereits in Abschnitt 2.3 vorgestellt. Es wird auch in [JZ91] im Zusammenhang mit Delegation erwähnt. Mit Hilfe dieses Musters kann ein Objekt sein Verhalten in Abhängigkeit von seinem inneren Zustand grundlegend ändern. Für den Benutzer sieht es so aus, als hätte das Objekt seine Klasse gewechselt. Das Muster ist eines der Muster, die Delegation als wichtigen, wenn nicht essentiellen Bestandteil enthalten. Das Beispiel mit dem grafischen Werkzeug aus Abschnitt 2.3 soll im folgenden weiter genutzt werden.

Zur Erinnerung: In einem Fenster können grafische Objekte plaziert und mit der Maus manipuliert werden. Das unterliegende Fenstersystem ruft Methoden dieses Fensters auf um Aktionen des Anwenders mitzuteilen. Je nach aktuellem Zustand (Objekt wird verschoben, Objekt wird in der Größe verändert, ...) führen diese Anwenderaktionen zu sehr unterschiedlichen Reaktionen des Programms.

9.3.1 Das Muster in Eiffel implementiert

In [GJHV95, Seite 305 ff.] wird folgende Implementierung vorgeschlagen: Die Instanzen der Klasse `ARBEITS_FENSTER` sind die Objekte, die ihr Verhalten abhängig vom internen Zustand ändern sollen. Das Verhalten selbst wird nicht in `ARBEITS_FENSTER` selbst implementiert, sondern an die abstrakte Klasse `MAUS_BEARBEITER` delegiert.

```
class ARBEITS_FENSTER
feature{ ANY }
   bewegung( new_x : integer, new_y : integer ) is do
      bearbeiter.bewegung( Current, new_x, new_y )
   end

   klick is do
      bearbeiter.klick( Current )
   end
   ...
feature{ MAUS_BEARBEITER }
   zustand_wechseln( neuer_zustand : MAUS_BEARBEITER ) is do
      bearbeiter := neuer_zustand
   end
   objekte : LIST[ GRAFIK ]
   ...
feature { NONE }
   bearbeiter : MAUS_BEARBEITER
   ...
end - class ARBEITS_FENSTER
```

Die Klasse `MAUS_BEARBEITER` definiert die Schnittstelle für alle Klassen, die ein zustandsspezifisches Verhalten implementieren. Der Parameter `fenster` ist notwendig, damit das Bearbeiter-Objekt weiß, in welches Fenster gezeichnet werden soll. Er ist also eine Folge der Entwurfsentscheidung, das Muster Zustand zu verwenden.

```
deferred class MAUS_BEARBEITER
feature{ ANY }
   bewegung( fenster:ARBEITS_FENSTER, new_x:integer, new_y:integer )
   is deferred end

   klick( fenster : ARBEITS_FENSTER )
   is deferred end
   ...
end - class MAUS_BEARBEITER
```

Die Klasse VERSCHIEBEN_BEARBEITER ist ein Beispiel für eine konkrete Implementierung eines Zustands. Wenn der Anwender die Maus bewegt, wird ein Objekt verschoben und der Fensterinhalt neu gezeichnet. Klickt der Anwender hingegen mit der Maus, so beendet er die Verschiebung. Dann wird das Fenster in einen neuen Zustand versetzt.

```
class VERSCHIEBEN_BEARBEITER
inherit MAUS_BEARBEITER
feature{ ANY }
   bewegung( fenster:ARBEITS_FENSTER, new_x:integer, new_y:integer )
   is
    -- korrigiere Koordinaten des zu verschiebenden Objekts
      -- zeichne Objekte neu
   end

   klick( fenster : ARBEITS_FENSTER ) is
   local neuer_zustand : LEERLAUF_BEARBEITER
   do
      -- beende das Verschieben
      !!neuer_zustand
      fenster.zustand_wechseln( neuer_zustand )
   end
   ...
end - class VERSCHIEBEN_BEARBEITER
```

Ein Vorteil, den Eiffel gegenüber C++ bietet, ist der, daß ARBEITS_FENSTER durch selektiven Export (siehe Abschnitt 5.3.1) genau festlegen kann, auf welche Merkmale die Bearbeiter-Klassen zugreifen dürfen. Trotzdem bleibt der Nachteil, daß die trivialen Implementierungen der vom Fenstersystem aufgerufenen Methoden in ARBEITS_FENSTER von Hand gemacht werden müssen. Wenn man berücksichtigt, wie viele Ereignisse ein modernes Fenstersystem in einer Anwendung auslösen kann (siehe z.B. [Pet96]), ist dies ein nicht zu unterschätzender Aufwand. Auch der zusätzliche Parameter `fenster`, den jede Methode von MAUS_BEARBEITER hat, ist nicht wünschenswert.

9.3.2 Das Muster in E+ implementiert

In MAUS_BEARBEITER werden die Methoden zur Bearbeitung von Mausereignissen nach wie vor als abstrakt deklariert (siehe Abbildung 9.3). Allerdings ist der zusätzlichen Parameter `fenster` jetzt nicht mehr erforderlich. Das Attribut `objekte` darf jedoch nicht als abstrakt deklariert werden, ebensowenig wie das Attribut `bearbeiter`, das von ARBEITS_FENSTER redefiniert wird und über das ein Bearbeiter sich selbst durch einen anderen ersetzen kann. Die Begründung dafür wird in Abschnitt 6.2.4 behandelt. Da die von MAUS_BEARBEITER erbenden Klassen instanziierbar sein müssen, dürfen sie keine abstrakten Eigenschaften enthalten. Anstatt in jedem konkreten Bearbeiter die beiden Eigenschaften zu implementieren, werden sie einmal in der Oberklasse implementert.

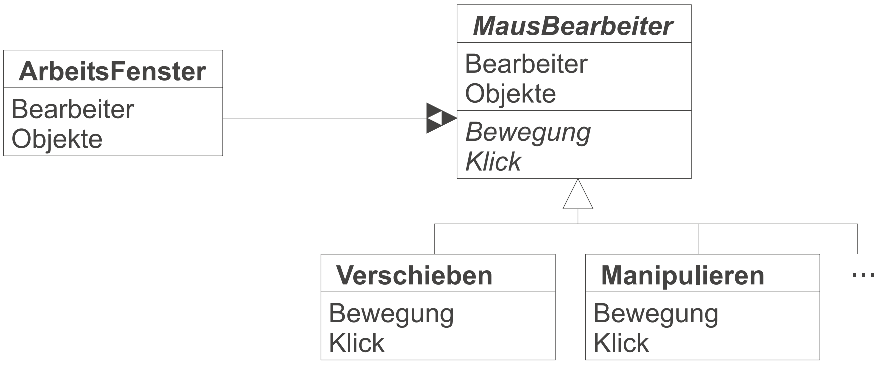

Abbildung 9.3: Das Entwurfsmuster *Zustand*

```
deferred class MAUS_BEARBEITER
feature{ ANY }
   bewegung( new_x : integer, new_y : integer ) is deferred end
   klick is deferred end
   ...
feature{ ARBEITS_FENSTER }
   objekte    : LIST[ GRAFIK ]
   bearbeiter : MAUS_BEARBEITER
   ...
end - class MAUS_BEARBEITER
```

Die so vorbereitete Klasse `MAUS_BEARBEITER` kann von `ARBEITS_FENSTER` nun als Elternklasse benutzt werden. Die Eigenschaften `bearbeiter` und `objekte` werden redefiniert. Wenn zur Laufzeit an ein konkretes Bearbeiter-Objekt delegiert wird, greift dieses automatisch auf diese redefinierten Eigenschaften zu. In der Initialisierungsmethode wird die Delegationsreferenz gesetzt.

```
class ARBEITS_FENSTER
delegate MAUS_BEARBEITER to bearbeiter
redefine objekte, bearbeiter end
creation make

feature{ ANY }
   make is
   local
      initialer_zustand : LEERLAUF_BEARBEITER
   do
      !!initialer_zustand
      bearbeiter := initialer_zustand
      !!objekte
   end

feature { NONE }
   objekte : LIST[ GRAFIK ]
   bearbeiter : MAUS_BEARBEITER
   ...
end - class ARBEITS_FENSTER
```

Die Implementierung einer konkreten Unterklasse von `MAUS_BEARBEITER` birgt keine Überraschungen. Der zusätzliche Parameter für das aufrufende Fenster fällt weg und zum setzen eines neuen Zustands kann direkt die Eigenschaft `fenster` der Oberklasse benutzt werden, die von `ARBEITS_FENSTER` redefiniert wurde.

9.3.3 Vor- und Nachteile der E+ Implementierung

Die KLasse `ARBEITS_FENSTER` ist einfacher geworden. Dort brauchen die Methoden zur Weiterleitung von Mausereignissen nicht mehr aufgeführt zu werden. In der Initialisierungsmethode muß zusätzlich das Elternobjekt erzeugt werden. Die Bearbeiter enthalten jeweils zwei neue Eigenschaften (`objekte` und `zustand`). Da es sicht nur um Referenzen handelt, ist der zusätzliche Bedarf an Speicher minimal.

Ein Nachteil ist, daß die Instanzen konkreter Unterklassen von `MAUS_BEARBEITER` an eine bestimmte Instanz von `ARBEITS_FENSTER` gebunden werden. Da sie wahrscheinlich weitere private Attribute benötigen, enthalten sie eigene und fremde, d.h. in `ARBEITS_FENSTER` redefinierte Attribute. Wenn alle Attribute zusammen den Zustand eines Objekts ausmachen, sind die eigenen und fremden Attribute möglicherweise voneinander abhängig. Eine Realisierung der Bearbeiter-Klassen als *Singletons*, wie sie in [GJHV95] vorgeschlagen wird, ist daher nicht möglich.

Kapitel 10

Zusammenfassung und Ausblick

Die bisherigen Zusammenfassungen am Ende einzelner Kapitel bezogen sich speziell auf die im Kapitel eingeführten und untersuchten Konstrukte. In diesem letzten Kapitel der Diplomarbeit werden die Ergebnisse der Arbeit abschließend bewertet. Anschließend werden mögliche Fortsetzungen vorgeschlagen, die als Ausblick die Arbeit abschließen.

10.1 Ergebnisse der Diplomarbeit

Die Wahl von Eiffel als Plattform für eine Spracherweiterung hat sich als vorteilhaft erwiesen. Die vielfältigen Möglichkeiten, die diese Sprache bei der Anpassung von Vererbungsbeziehungen bietet, verlangten für die Erweiterung um Delegationsbeziehungen eine genaue Untersuchung von Seiteneffekten. Dadurch traten Probleme deutlicher hervor, als dies vielleicht bei anderen Sprachen der Fall gewesen wäre. Die angepaßten Konstrukte konnten ihre grundsätzliche Bedeutung beibehalten, was ein Zeichen für die enge Verwandschaft zwischen Vererbung und Delegation ist.

Durch den Einsatz objektorientierter Programmiersprachen sollen korrekte und robuste Software-Systeme entstehen. Bei der Betrachtung des Vertragsmodells wurden jedoch schwerwiegende Sicherheitslücken offensichtlich, die durch die Erweiterung um Delegation entstanden sind. Durch die enge Verbindung mehrerer Objekte in Delegationsbeziehungen sind in E+ neue Quellen für Laufzeitfehler hinzugekommen. Die Komplexität eines Software-Systems steigt dadurch beachtlich. Wenn Korrektheit und Robustheit entscheidende Kriterien für ein zu entwickelndes System sind, stellt sich die Frage, ob Delegation nicht mehr Schaden als Nutzen bringt.

Die Evaluierung hat gezeigt, daß E+ in bestimmten Situationen zu einfacheren Implementierungen eines Entwurfs führen kann. Vorteile delegationsbasierter Sprachen, z.B. Zuordnung von Merkmalen auf Objekt- statt auf Klassenebene, können mit E+ simuliert werden. Es ist anzunehmen, daß nach ausreichender Einarbeitung in die neuen Konstrukte, ein Entwickler damit ähnlich flexible Lösungen entwickeln kann, wie mit delegationsbasierten Sprachen. Ob die komplexeren Inter-Objekt-Beziehungen zu rechtfertigen snd, kann nicht allgemein entschieden werden.

E+ hat also sowohl Vor- als auch Nachteile. Die aus der Sicht eines Sprachdesigners sicherlich hochinteressanten Möglichkeiten, die sich mit Delegation eröffnen, sind aus softwaretechnischer Sicht in vielen Fällen bedenklich. Die Nutzung von Delegation muß vorsichtig und überlegt geschehen, führt dann aber auch zu sehr eleganten Lösungen. Als Mittelweg könnte sich die `forward`-Beziehung (siehe Abschnitt 8.1) erweisen, die einen Teil der durch Delegation gebotenen Flexibilität ohne die damit verbundenen Risiken bietet.

10.2 Ausblick

Eine mögliche Fortsetzung dieser Arbeit liegt in der Entwicklung und Implementierung eines Compilers für E+. Dazu bieten die Arbeiten [Kni96, Cos98, Sch98, FH97] eine gute Ausgangsbasis.

In dieser Arbeit wurde Delegation als geschlossene Beziehung zwischen Klassen untersucht. Dadurch bleibt die Kapselung des inneren Zustands eines Objekts erhalten. Es ergeben sich daraus aber auch einige Schwierigkeiten (siehe z.B. Abschnitt 5.4.2). Es könnte daher sinnvoll sein, Delegation als offene Beziehung zu sehen. Vielleicht ergeben sich bei einer genauen Untersucheung der in E+ eingeführten Konstrukte dann Vorteile, die den Verlust der Kapselung aufwiegen.

Einige Aspekte delegationsbasierter Programmiersprachen wurden in eine objektorientierte Sprache integriert. E+ ist dadurch aber keine Prototyping-Sprache geworden. Für die Evaluierung von Anforderungen, d.h. zur schnellen Entwicklung von Prototypen, ist sie nicht geeignet. Eine noch weitergehende Integration beider Paradigmen könnte klassenlose, selbstbeschreibende Objekte in Eiffel integrieren. Eine Teilmenge der Sprache könnte dann zum Prototyping eingesetzt werden und ein Prototyp könnte inkrementell in ein objektorientiertes Endprodukt überführt werden, ohne daß ein Wechsel der Sprache erforderlich wäre. Benutzungsoberflächen könnten auch im Endprodukt weiterhin klassenlos implementiert werden. Natürlich müßte genau untersucht werden, ob eine solche hybride Sprache tatsächlich Vorteile bringt.

Anhang A

Syntaktische Spezifikation von E+

Die Syntax von E+ entspricht der von Eiffel. Ein syntaktisch korrektes Eiffel-Programm ist daher auch ein syntaktisch korrektes E+-Programm. Wo E+ Eiffel erweitert, kommen natürlich auch neue syntaktische Konstrukte hinzu. Diese neuen Konstrukte werden im folgenden beschrieben. Die Notation entspricht der in [Mey92, Seite 557 ff.]. Die einzelnen Schreibweisen haben folgende Bedeutung:

[*Bezeichner*]	:	die Angabe von *Bezeichner* ist optional
{ *Bezeichner* ... }	:	eine Menge mit *Bezeichner* als Elemente
"*Symbol*"	:	*Symbol* ist ein Zeichen
Schlüsselwort	:	Eiffel/E+ Schlüsselwörter werden **fett** dargestellt

Konstrukte, die sich gegenüber Eiffel nicht verändert haben, werden nicht explizit definiert sondern nur mit einem entsprechenden Hinweis versehen aufgelistet.

Class_declaration	$\hat{=}$	[Indexing] Class_Header [Formal_generics] [Obsolete] [Inheritance] [Delegation] [Creators] [Features] [Invariant] **end** ["–" **class** Class_name]
Indexing	$\hat{=}$	*unverändert zu Eiffel*
Class_header	$\hat{=}$	*unverändert zu Eiffel*
Formal_generics	$\hat{=}$	*unverändert zu Eiffel*
Obsolet	$\hat{=}$	*unverändert zu Eiffel*
inheritance	$\hat{=}$	*unverändert zu Eiffel*
Delegation	$\hat{=}$	Delegate_list
Delegate_list	$\hat{=}$	{ Delegatee ";" ... }
Delegatee	$\hat{=}$	**delegate** Class_type **to** Feature_name [Feature_adaption]
Feature_adaption	$\hat{=}$	[Rename] [New_exports] [Undefine] [Redefine]

		[select]
		end
Feature_name	$\widehat{=}$	*unverändert zu Eiffel*
Class_type	$\widehat{=}$	*unverändert zu Eiffel*
Rename	$\widehat{=}$	*unverändert zu Eiffel*
New_exports	$\widehat{=}$	*unverändert zu Eiffel*
Redefine	$\widehat{=}$	*unverändert zu Eiffel*
Undefine	$\widehat{=}$	*unverändert zu Eiffel*
Select	$\widehat{=}$	**select** Selected_features
Selected_features	$\widehat{=}$	{Ordered_list "," ... }
Ordered_list	$\widehat{=}$	Feature_name [Subordinated_list]
Subordinated_list	$\widehat{=}$	{ Subordinated_feature ... }
Subordinated_feature	$\widehat{=}$	**then** Feature_name
Creators	$\widehat{=}$	*unverändert zu Eiffel*
Features	$\widehat{=}$	*unverändert zu Eiffel*
Invariant	$\widehat{=}$	*unverändert zu Eiffel*
Feature_name	$\widehat{=}$	*unverändert zu Eiffel*

Anhang B

Glossar

Die folgende Übersicht erklärt oder definiert die wichtigsten Begriffe aus der Welt der objektorientierten Programmierung. Wird bei der Erklärung eines Begriffes auf einen anderen, im Glossar enthaltenen Begriff verwiesen, so ist dies durch einen Pfeil (↑,bzw. ↓) gekennzeichnet. Bei einigen Begriffen wird statt einer ausführlichen Erklärung ein Verweis auf einen Abschnitt der Arbeit angegeben. Dort erfolgt eine ausführliche und systematische Erklärung des Begriffs.

abstraktes Merkmal
: Eine Klasse kann ein Merkmal↓ spezifizieren, ohne es zu implementieren. Solche Merkmale werden abstrakte Merkmale genannt. Sie dienen der Schaffung von Schnittstellen↓. Eine Klasse↓, die mindestens ein abstraktes Merkmal enthält, wird abstrakte Klasse genannt. Von abstrakten Klassen können keine Instanzen erzeugt werden. Die abstrakten Merkmale werden in Unterklassen↓ implementiert.

Aktivierungsumgebung
: Eine Aktivierungsumgebung (Activation-Record) ist eine Datenstruktur, die alle zur Ausführung einer bestimmten Prozedur notwendigen Daten enthält. Dazu zählen z.B. lokale Variablen. Beim Aufruf der Prozedur wird die Aktivierungsumgebung auf dem Stapel des aufrufenden Prozesses instanziiert. Der Begriff stammt aus dem Bereich Compilerbau, eine gute Beschreibung ist z.B. [ASU86].

Anwender
: Mit Anwender wird der Endanwender eines Systems bezeichnet, also die Person, die mit dem fertigen Produkt arbeitet.

Attribut
: Ein Attribut ist eine einem Objekt↓ zugeordnete Variable zur Speicherung eines Werts. In objektorientierten Programmiersprachen sind die Attribute eines Objekts durch seine Zugehörigkeit zu einer Klasse↓ festgelegt. Attribute sind in der Regel für Kunden↓ nicht zugänglich. Die Werte können nur indirekt durch den Aufruf von Methoden↓ gelesen oder geschrieben werden.

Delegation
: Wenn ein Objekt↓ eine Nachricht↓ empfängt, zu der es selbst keine passende Methode↓ hat, leitet es die Nachricht an ein Elternobjekt↓ weiter, das die Nachricht bearbeitet oder selbst weiterleitet. Sendet ein Objekt im Verlauf der Bearbeitung einer Nachricht weitere Nachrichten an sich selbst (`self`), beginnt die Suche nach einer passenden Methode beim Empfänger der ursprünglichen Nachricht (siehe Abschnitt 1.1).

delegierendes Objekt
: siehe Kindobjekt↓

delegiertes Objekt
siehe Elternobjekt↓

delegationsbasiert
Eine klassenlose Programmiersprache ist delegationsbasiert, wenn sie Vererbung zwischen Objekten↓ durch Delegation↑ realisiert (siehe Abschnitt 3.2).

Eigenschaft
siehe Merkmal↓

Elternklasse
Eine Klasse↓ `A` ist Elternklasse einer Klasse `B`, wenn `B` von `A` über eine Delegationsbeziehung erbt, d.h. wenn `B` an `A` delegiert. Bei Vererbungsbeziehungen wird statt Elternklasse der Begriff Oberklasse↓ benutzt.

Elternobjekt
Ein Objekt↓ `obj_a` ist ein Elternobjekt eines Objekts `obj_b`, wenn `obj_b` an `obj_a` delegiert. In E+ gilt zusätzlich: damit `obj_a` ein Elternobjekt von `obj_b` sein kann, muß die Klasse↓ `A` von `obj_a` eine Elternklasse der Klasse `B` von `obj_b` sein.

Entwickler
Ein Entwickler ist die Person, die ein Software-System oder einen Teil des Systems entwirft oder implementiert.

Instanziierung
Der Vorgang, ein Objekt↓ gemäß der durch eine Klasse↓ gegebene Vorlage zu erzeugen, wird Instanziierung genannt. Das Objekt ist dann eine Instanz der Klasse.

Instanzvariable
siehe Attribut↑

Kindklasse
Eine Klasse↓ `A` ist Kindklasse einer Klasse `B`, wenn `A` von `B` über eine Delegationsbeziehung erbt, d.h. wenn `A` an `B` delegiert. Bei Vererbungsbeziehungen wird statt Kindklasse der Begriff Unterklasse↓ benutzt.

Kindobjekt
Ein Objekt↓ `obj_a` ist ein Kindobjekt eines Objekts `obj_b`, wenn `obj_a` an `obj_b` delegiert. In E+ gilt zusätzlich: damit `obj_a` ein Kindobjekt von `obj_b` sein kann, muß die Klasse↓ `A` von `obj_a` eine Kindklasse↑ der Klasse `B` von `obj_b` sein [Kni96].

Klasse
Eine Klasse spezifiziert die Daten (Attribute↑) und die Methoden↓ eines Objekts↓ [Dob96]. Eine Klasse ist also eine Vorlage für die Erzeugung von Objekten mit gleichem Verhalten, die als Instanzen↑ der Klasse bezeichnet werden. In Eiffel gibt es eine Äquivalenz zwischen Klassen und Typen↓ [Mey97].

klassenbasierte Programmiersprache
Eine Programmiersprache ist klassenbasiert, wenn jedes Objekt↓ Instanz einer Klasse↑ ist [Weg87].

Kunde
Wenn Objekte↓ einer Klasse↑ `A` Nachrichten↓ an Objekte einer Klasse `B` senden, so wird `A` als Kunde von `B` bezeichnet. Anders gesagt: `A` benutzt Eigenschaften↑ von `B`. `B` heißt Lieferent↓ von `A`.

Lieferant
Wenn Objekte↓ einer Klasse↑ `A` Nachrichten↓ an Objekte einer Klasse `B` senden, so wird `B` als Lieferant von `B` bezeichnet. Anders gesagt: `A` benutzt Eigenschaften↑ von `B`. `A` heißt Kunde↑ von `B`.

Merkmal
Ein Merkmal ist ein Attribut↑ oder eine Methode↓.

Methode
Eine Methode ist eine einem Objekt↓ zugeordnete Routine. Über Methoden kann von außen auf ein Objekt zugegriffen werden. Die Menge aller Methoden eines Objekts bilden (zusammen mit den von außen zugreifbaren Attributen) seine Schnittstelle↓.

Nachricht
Der Aufruf einer Methode↑ eines Objekts↓ in der Form `obj.methode` wird als das Senden der Nachricht `methode` an das Objekt `obj` bezeichnet. Eine Nachricht ist erlaubt, wenn das Objekt eine Methode hat, die aufgrund des Aufrufs ausgewählt werden kann um den Aufruf zu bearbeiten.

Oberklasse
Eine Klasse↑ `A` ist Oberklasse einer Klasse `B`, wenn `B` von `A` über eine Vererbungsbeziehung (nicht über eine Delegationsbeziehung) erbt. Jede Klasse ist Oberklasse von sich selbst [Mey97]. Bei Delegationsbeziehungen wird statt Oberklasse der Begriff Elternklasse↑ benutzt.

Objekt
Objekte sind autonome Entitäten, die auf Nachrichten↑ reagieren und einen inneren Zustand haben [Weg87]. Ein Objekt hat eine Identität, die das Objekt unabhängig von seinem Zustand (den aktuellen Werten der Attribute↑) von anderen Objekten unterscheidbar macht. In klassenbasierten Programmiersprachen↑ sind Objekte stets Instanzen↑ von Klassen↑ [Dob96].

Objektbasierte Programmiersprache
Eine Programmiersprache ist objektbasiert, wenn sie mit Ausnahme von Vererbung alle Eigenschaften einer objektorientierten Programmiersprache unterstützt.

Objektorientierte Programmiersprache
siehe Definition in Kapitel 1.

Objektverbund
Ein Objektverbund besteht aus einer Menge von Objekten↑, die zur Erfüllung einer Aufgabe kooperieren. Ein Benutzer greift über ein Objekt auf den Verbund zu. Die Kooperation der Objekte untereinander ist für ihn transparent. Für ihn sieht es so aus, als arbeite er nur mit einem einzelnen Objekt (siehe Abschnitt 2.2).

Operation
siehe Methode↑.

Rolle
Eine Rolle ist eine Sichtweise auf eine bestimmte Entität des Anwendungsbereichs.

Rollenfüller
Ein Rollenfüller ist ein Objekt, daß die Funktionalität für eine bestimmte Rolle↑ zur Verfügung stellt.

Schnittstelle
Eine Schnittstelle ist eine Menge von Operationen/Methoden↑, beschrieben durch ihre Signaturen↓. Die Schnittstellen eines Objekts↑ charakterisieren vollständig die Nachrichten↑, die an ein Objekt gesendet werden können. [Dob96] Zu der durch die Signaturen gegebenen syntaktischen Beschreibung gehört zusätzlich eine semantische Beschreibung, die die Bedeutung der Methoden festlegt. Oft erfolgt diese Beschreibung informell durch Kommentare oder vom Quelltext unabhängige Dokumentationen. In Eiffel kann eine semantische Beschreibung teilweise durch Zusicherungen erfolgen (siehe Kapitel 7).

Signatur
Die Signatur einer Operation/Methode↑ besteht aus ihrem Namen, dem Rückgabewert und den formalen Parametern.

Subklasse
Eine Klasse↑ `A` ist Subklasse einer Klasse `B`, wenn `A` entweder eine Unterklasse↓ oder eine Kindklasse↑ von `B` ist.

Subtyp
Ein Typ↓ `A` ist ein Subtyp eines Typs `B`, wenn die Schnittstelle↑ von `A` die Schnittstelle von `B` enthält.

Superklasse
Eine Klasse↑ `A` ist Superklasse einer Klasse `B`, wenn `A` entweder eine Oberklasse↑oder eine Elternklasse↑ von `B` ist.

Typ
Ein Typ ist der Name einer Schnittstelle↑.[Dob96] Von einem Typ können Instanzen↑ erzeugt werden, die diese Schnittstelle unterstützen. [Weg87]

Unterklasse
Eine Klasse↑ `A` ist Unterklasse einer Klasse `B`, wenn `A` von `B` über eine Vererbungsbeziehung (nicht über eine Delegationsbeziehung) erbt. Jede Klasse ist Unterklasse von sich selbst [Mey97]. Bei Delegationsbeziehungen wird statt Unterklasse der Begriff Kindklasse↑ benutzt.

Vererbung
siehe Beschreibung in Abschnitt 3.1.3 oder [Boo94, Mey97].

Literaturverzeichnis

[AG98] Ken Arnold und James Gosling. *The Java Programming Language.* Addison-Wesley, 2 Auflage, 1998.

[ASU86] Alfred Aho, Ravi Sethi, und Jeffrey Ullman. *Compilers - principles, techniques, and tools.* Addison-Wesley, 1986.

[BD96] Daniel Bardou und Christophe Dony. Split Objects: A Disciplined Use of Delegation within Objects. *ACM SIGPLAN Notices*, 31(10):122 ff., oct 1996.

[Boo94] Grady Booch. *Objektorientierte Analyse und Design.* Addison Wesley, 2 Auflage, 1994.

[Bur97] Rainer Burkhardt. *UML - Unified Modeling Language.* Addison-Wesley, 1997.

[Cos98] Pascal Costanza. Lava - Delegation in einer streng typisierten Programmiersprache - Sprachdesign und Compiler. Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelm-Universität Bonn, 1998.

[CS95] James O. Coplien und Douglas C. Schmidt. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[CVK96] James O. Coplien, John Vlissides, und Norman L. Kerth. *Pattern Languages of Program Design 2.* Addison-Wesley, 1996.

[DD96] Ernst-Erich Doberkat und Stefan Dißmann. *Einführung in die objektorientiere Programmierung mit BETA.* Addison-Wesley, 1996.

[DG95] Stefan Dißmann und Volker Gruhn. Delegation als Konzept der objekt-orientierten Modellierung. In *GISI 95 Herausforderungen eines globalen Informationsverbundes für die Informatik*, 25. GI-Jahrestagung und 13. Schweizer Informatiktag, Seiten 459–470, Zürich, 18.-20.9.1995 1995.

[Dob96] Ernst Erich Doberkat. Folien zur Vorlesung Software-Technologie II - Entwurfsmuster im SS 96. Universität Dortmund, 1996.

[FH97] Ulrich Frank und Sören Halter. Enhancing Objekt-Oriented Software Development with Delegation. Technischer Bericht 2, Institut für Wirtschaftsinformatik - Universität Koblenz-Landau, 1997.

[GJHV95] Erich Gamma, Ralph Johnson, Ralph Helm, und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[GR89] Adele Goldberg und David Robson. *Smalltalk-80. The Language.* Addison-Wesley, 1989.

[Has95] Wilhelm Hasselbring. Folien zur Stammvorlesung Software-Technologie im WS 95/96. Universität Dortmund, 1995.

[Heu97] Andreas Heuer. *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme.* Addison-Wesley, 2 Auflage, 1997.

[HZ97] W. Hasselbring und P. Ziesche. Einsatz von Entwurfsmustern bei der Entwicklung eines föderierten Krankenhausinformationssystems mit CORBA. In *Tagungsband 'Verteilte Objekte in Organisationen' (Mobis '97)*, Band 4, Seiten 21–25, Bamberg, September 1997. Rundbrief Informationssystem-Architekturen.

[Inc98] Interactive Software Engineering Inc. Object-Oriented Languages: A Comparison. www.eiffel.com/doc/manuals/technology/oo-comparison, 1998.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering : A Use Case Driven Approach*. Addison-Wesley, 1992.

[JZ91] Ralph E. Johnson und Jonathan M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, 4(7):31–34, November/Dezember 1991.

[Kat97] Ravi Kathuria. Improved design of classes and associations using assimilation. *JOOP-Journal of Object-Oriented Programming*, 4(3):32–37, June 1997.

[Kni96] Günter Kniesel. Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems. Technischer bericht, Institut für Informatik II - Universität Bonn, 1996.

[KS96] Ravi Kathuria und Venkar Subramniam. Assimilation: A New and Necessary Concept for an Object Model. *Report on Object Analysis and Design (ROAD)*, Seiten 36–39, 1 1996.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systms. In *OOPSLA '86*, SIGPLAN Notices, Seiten 214–223. ACM Press, September 1986.

[Mey89] Bertrand Meyer. You Can Write, But Can You Type? *JOOP - Journal of object-oriented programming*, 1(6):58–67, 1989.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mey95] Scott Meyers. *Effektiv C++ programmieren: 50 Möglichkeiten zur Verbesserung Ihrer Programme*. Addison Wesley, 2 Auflage, 1995.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 2 Auflage, 1997.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, und Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[New] www.newton.com.

[Pap95] Michael Papathomas. Concurrency in Object-Oriented Programming. In *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[Pet96] Charles Petzold. *Windows 95 Programmierung*. Microsoft Press, 1996.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, und William Lorensen. *Objekt-oriented Modeling and Design*. Prentice Hall, 1991.

[Rog98] Dale Rogerson. *Inside COM*. Microsoft Press, 1998.

[Sau89] J. Saunders. A Survey of Object-Oriented Programming Lnaguages. *JOOP - Journal of object-oriented programming*, 1(6):5–11, 1989.

[Sch98] Matthias Schickel. Lava - Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung. Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelm-Universität Bonn, 1998.

[SM95] Patrick Steyaert und Wolfgang De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Seiten 127–144. Springer-Verlag, 1995.

[Smi95] Walter R. Smith. Using a Prototype-Based Language for User Interfaces: The Newton Project's Experiences. In *OOPSLA '95 Conference Proceedings, SIGPLAN Notices*, Seiten 61–72. ACM Press, 1995.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. *ACM SIGPLAN Notices*, 22(12):138–146, dec 1987.

[Str94] Bjarne Stroustrup. *Design und Entwicklung von C++*. Addison-Wesley, 1994.

[Str97] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley, 3 Auflage, 1997.

[SU95] Randall B. Smith und David Ungar. Programming as an Experience: The Inspiration for Self. In *ECOOP 95*. Springer, 1995.

[Ull94] Jeffrey D. Ullman. *Elements of ML programming*. Prentice Hall, 1994.

[US87] David Ungar und Randall B. Smith. SELF: The Power of Simplicity. In *OOPSLA '87*, Seiten 227–242. ACM Press, 1987.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *OOPSLA '87*, Seiten 168–182. ACM Press, 12 1987.

[Weg93] Ingo Wegener. *Theoretische Informatik*. Teubner, 1993.

[You93] Edward Yourdon. *Decline & Fall of the American Programmer*. Prentice Hall, 1993.