Objektorientierte Konzepte in Smalltalk, C++, Objective-C, Eiffel und Modula-3

Harald Gall, Manfred Hauswirth und René Klösch

Institut für Informationssysteme, Abteilung für Verteilte Systeme, Technische Universität Wien

Zusammenfassung. Dieser Artikel untersucht einige weit verbreitete objektorientierte Programmiersprachen hinsichtlich ihrer Umsetzung der zugrundeliegenden objektorientierten Konzepte. Als Basis für einen konsistenten Vergleich wird eine einheitliche (objektorientierte) Terminologie definiert.

Schlagwörter: Objektorientierte Programmierung und Programmiersprachen, Objekt- und Klassendefinition, Abstraktion, Datenkapselung, Polymorphismus, Vererbung, dynamisches Binden.

Summary. This article surveys several popular object-oriented programming languages. A consistent terminology is defined and important concepts of object-orientation are explained. We then classify the languages according to the terminology and concepts.

Key words: Object-oriented programming and programming languages, object definition, class definition, abstraction, encapsulation, polymorphism, inheritance, dynamic binding.

Computing Reviews Classification: D.1.5, D.2.7, D.3.2, D.3.3

---

1. Motivation

Die zunehmende Verbreitung des objektorientierten Paradigmas führte zur Entwicklung einer Reihe objektorientierter Programmiersprachen und zur Erweiterung bisher nicht objektorientierter Sprachen um derartige Konzepte. Dieser Bereich ist gegenwärtig einer hohen Dynamik unterworfen, sodaß laufend eine Fülle neuer Vorschläge und deren praktische Umsetzungen in Programmiersprachen auftauchen.

Der Anwender sieht sich einer Unmenge an praktischen Realisierungen gegenüber und ist gefordert, dieselben nach ihren Möglichkeiten bzw. ihrer Verwendbarkeit zu beurteilen. Durch seine rasche Verbreitung und die dem Bereich des Software-Engineering leider derzeit noch anhaftende "Unschärfe" gibt es im Gebiet der Objektorientierung eine Unzahl von Konzepten und Begriffen, welche sich überlappen oder sogar widersprechen. Dies ist verwirrend für Anwender, die heute bereits häufig beachtliches Fachwissen im Bereich der Objektorientierung besitzen. In diesem Artikel wollen wir auf Basis einer knappen, präzisen Begriffsdefinition dem versierten Anwender einen raschen Überblick über einige der populärsten objektorientierten Programmiersprachen und ihrer Umsetzung der objektorientierten Konzepte geben.

2. Konzepte

Nicht jede Programmiersprache, die Objekte unterstützt, ist deshalb auch als objektorientierte Programmiersprache zu bezeichnen. Laut Wegner [19] ist eine Sprache dann objektorientiert, wenn sie folgende Sprachmittel zur Verfügung stellt: Objekte, Klassen und Vererbung. Sprachen, die lediglich Objekte unterstützen (z.B. Ada83) werden als *objekt-basiert* bezeichnet.

Zusätzlich zu diesen für objektorientierte Sprachen "verpflichtenden" Basiskonzepten werden dem Anwender häufig noch weitere objektorientierte Konzepte, wie Polymorphismus und dynamisches Binden zur Verfügung gestellt, die von Programmiersprachen in unterschiedlichem Umfang realisiert werden.

Neben diesen klassisch objektorientierten Konzepten werden die in diesem Artikel betrachteten Sprachen auch hinsichtlich der allgemeinen programmiersprachlichen Kriterien Typkonzept und Kapselung untersucht.

2.1. Klassen und Objekte

Ein *Objekt* ist als Abstraktion eines "realen Dinges" zu verstehen, das einen inneren Zustand besitzt, der nur mittels der Methoden (i.e. Operationen) des Objektes manipuliert werden kann (vgl. [20]). Der Zustand eines Objektes wird durch seine Attribute (z.B. Eckpunktkoordinaten einer geometrischen Figur) repräsentiert. Die Methoden legen fest, auf welche *Nachrichten* (Methoden-Invokationen) ein Objekt reagieren kann.

Die Dynamik eines objektorientierten Systems (i.e. der Kontrollfluß) wird durch den Austausch von Nachrichten zwischen Objekten (i.e. Aufruf von Methoden) realisiert.

Mit dem Konzept des Objekts eng verbunden ist der Begriff der Klasse. Sie beschreibt eine Menge von Objekten hinsichtlich Struktur, Hierarchie in bezug auf Vererbung und Verhalten (vgl. [1]). Ein Objekt ist eine Instanz, ein konkretes Vorkommnis einer Klasse. Die Klasse stellt also eine Schablone dar, nach der Objekte (dieser Klasse) erzeugt werden können.

Wie Objekte besitzen auch Klassen interne Daten (Klassenvariablen – Platzhalter für Informationen für alle Objekte dieser Klasse) und Methoden (Klassenmethoden – z.B. zum Erzeugen und Löschen von Objekten dieser Klasse).

2.2. Vererbung

Unter Vererbung ist jener Mechanismus zu verstehen, der es ermöglicht, Attribute und Methoden von Klassen an andere Klassen weiterzugeben. Eine vererbende Klasse (Superklasse) gibt Attribute und Methoden an eine erbende Klasse (Subklasse) weiter, die von dieser erweitert (um zusätzliche Attribute und Methoden) bzw. modifiziert (Redefinition von Methoden) werden können.

Durch Vererbung wird eine Relation definiert, die eine Hierarchie von Sub- und Superklassen bestimmt (Klassenhierarchie). Je nachdem, ob von genau einer oder mehreren Superklassen geerbt wird, spricht man von *einfacher* bzw. *mehrfacher Vererbung*.

2.3. Typkonzept

In objektorientierten Sprachen muß grundsätzlich unterschieden werden zwischen dem *statischen Typ* (bzw. der *statischen Klasse*) eines Objektes, womit der Typ gemeint ist, den der Compiler herleiten kann, und dem *dynamischen Typ* (bzw. der *dynamischen Klasse*), welcher den Typ eines Objektes angibt, den das Objekt zur Laufzeit eines Programms tatsächlich besitzt. Hinsichtlich des Typkonzeptes wird folgende Klassifikation verwendet(vgl. [5, 10]):

Static typing. Der Typ jedes Ausdrucks in einem Programm kann durch statische Analyse (Compiler) bestimmt werden. Typfehler werden bei der Übersetzung erkannt und können somit zur Laufzeit nicht auftreten.

Strong typing. Werden Vererbung bzw. Polymorphismus ausgenützt, kann der Typ davon betroffener Ausdrücke nicht mittels statischer Analyse festgelegt werden, d.h. der dynamische Typ unterscheidet sich vom statischen Typ. Dies ist beispielsweise der Fall, wenn Objekte aus Subklassen in einem Superklassenkontext verwendet werden oder Subklassen Methoden von Superklassen redefinieren. Daher muß durch zusätzliche Laufzeitüberprüfungen sichergestellt werden, daß ein Programm, welches sich fehlerlos übersetzen läßt, auch ohne Typfehler abläuft (*Typkonsistenz*).

Untyped. Es existiert kein explizites Typkonzept und es wird daher keine Typüberprüfung durchgeführt. Durch "Typinkonsistenzen" (z.B. Verwendung unbekannter Methoden) verursachte Laufzeitfehler können also auftreten.

2.4. Polymorphismus

Unter Polymorphismus ist die Fähigkeit einer Größe zu verstehen, zur Laufzeit unterschiedliche Ausprägungen annehmen zu können. Während in monomorphen Programmiersprachen Funktionen bzw. Prozeduren und ihre Parameter, Operatoren und deren Operanden, etc. einen eindeutigen Typ besitzen, erlauben polymorphe Sprachen dafür mehr als einen Typ (vgl. [5, 8]). Prinzipiell ist zwischen *ad hoc* und *universal* Polymorphismus zu unterscheiden[1] (vgl. Abb. 1).

[1] Die unterschiedlichen Arten von Polymorphismus werden zum besseren Verständnis anhand von Methoden bzw. Operatoren erläutert.

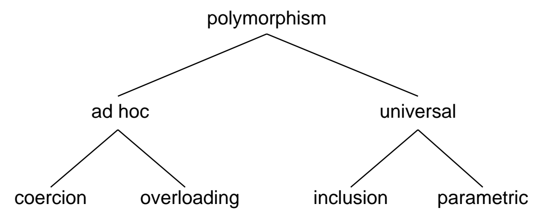

Abbildung 1: Arten von Polymorphismus [5]

Ad hoc Polymorphismus liegt vor, wenn z.B. eine Funktion auf einer endlichen Menge von Typen in einer unter Umständen vom Typ abhängigen Weise operieren kann. Dabei können zwei Ausprägungen unterschieden werden: Das (syntaktische) Konzept *Overloading* ermöglicht es, *einen* Namen für *unterschiedliche* semantische Objekte (z.B. Funktionen, Operatoren) zu vergeben und aufgrund des Verwendungskontextes statisch zu entscheiden, welche Ausprägung zu benutzen ist.

Coercion hingegen stellt ein semantisches Konzept dar. Statt wie bei *overloading* eine "passende" Funktion zu verwenden, wird jedes Funktionsargument in einen vorgegebenen (formalen) Typ umgewandelt (implizite oder explizite Typkonversion, zur Übersetzungs- oder Laufzeit).

Arbeiten Funktionen auf einer potentiell unbeschränkten Anzahl von Typen, so spricht man von *universal* Polymorphismus. Im Unterschied zu *ad hoc* Polymorphismus wird hier der gleiche Code für Argumente beliebigen Typs ausgeführt. Auch hier können zwei Arten unterschieden werden:

Inclusion Polymorphismus modelliert die Konzepte Vererbung bzw. Klassenhierarchie. Ein Objekt kann im Rahmen der Vererbungshierarchie als zu unterschiedlichen Klassen gehörig betrachtet werden. Somit kann ein Objekt aus einer Subklasse in einem Superklassenkontext verwendet werden.

Parametric Polymorphismus ermöglicht *Generizität*. Eine, in dieser Art polymorphe Funktion besitzt explizite oder implizite Typparameter, die die Typen der Argumente für eine konkrete Verwendung bestimmen (Template, Schablone). Eine solcherart generische Funktion kann ihre Aufgabe mit Argumenten eines erst später zu konkretisierenden Typs durchführen.

2.5. Dynamisches Binden

Wie in Abschnitt 2.4. beschrieben, erlauben polymorphe Programmiersprachen, die Verwendung von Konstrukten, bei denen zur Übersetzungszeit nicht festgelegt werden kann, welcher Klasse ein Objekt (zur Laufzeit) angehört bzw. welche Version (Ausprägung) einer Methode aufgerufen werden soll.

Dynamisches (spätes) Binden bedeutet, daß ein Name mit einer Klasse oder Methode erst dann assoziiert (verbunden) wird, wenn das durch den Namen bezeichnete Objekt erzeugt bzw. die durch den Namen referenzierte Methode verwendet wird. Das heißt, daß die Bindung erst zur Laufzeit hergestellt wird (vgl. [1]).

3. Programmiersprachliche Realisierungen

3.1. Smalltalk

Smalltalk [7, 16] ist (abgesehen von Simula-67) eine der ersten objektorientierten Programmiersprachen, die jedoch bereits alle wichtigen Konzepte des objektorientierten Paradigmas enthält. Aufgrund der notwendigen (teuren) Hardware-Voraussetzungen verbreitete sich Smalltalk ursprünglich nur langsam, erfreut sich inzwischen aber wegen seiner mächtigen Entwicklungsumgebung, seiner puristisch objektorientierten Konzepte und seiner besonderen Eignung für das Prototyping steigender Beliebtheit in der professionellen Programmentwicklung.

3.1.1. Klassen und Objekte

Smalltalk stellt eine puristische, vollständig objektorientierte Programmiersprache dar: In Smalltalk gibt es ausschließlich Objekte. Es werden veränderliche (i.e. Objekte im herkömmlichen Sinne) und unveränderliche Objekte (i.e. Zahlen, Zeichen, Zeichenketten, etc.) unterschieden. Aufgrund seiner reinen objektorientierten Konzeption werden in Smalltalk auch Klassen als Objekte bezeichnet. Objekte selbst sind Instanzen einer Klasse. Das sogenannte *class description protocol* definiert alle Attribute und Methoden jedes Objekts, das Instanz derselben Klasse ist. Neue Objekte einer Klasse (Objekterzeugung) werden durch entsprechende Nachrichten (i.e. Methodeninvokationen) an die Klasse selbst erzeugt (Klassenmethoden, z.B. `new`).

Jedes Objekt besitzt *private data*, das sind die Instanzvariablen (Attribute) des Objekts, welche nur der jeweiligen Instanz und ihren Instanzmethoden zugänglich sind. *Shared data* (Klassenvariablen) sind allen Objekten einer Klasse frei zugänglich und können von Instanzmethoden und Klassenmethoden verwendet werden. *Pool data* können von Objekten unterschiedlicher Klassen verwendet werden und sind über das sogenannte *pool dictionary* sowohl von Instanzmethoden als auch von Klassenmethoden zugreifbar. Obwohl diese Daten von verschiedenen Klassen verwendet werden können, dürfen sie ausschließlich über Methoden dieser Klassen, welche im *class description protocol* definiert werden müssen, verwendet werden.

Weiters werden bei der Klassendefinition im *class description protocol* die Klassen- und Instanzmethoden definiert. Ein Methodenaufruf ist in Smalltalk aus verschiedenen Blöcken aufgebaut: dem Empfänger (*receiver*) der Nachricht, einem Identifikationsmerkmal (*selector*) und den möglichen Argumenten.

3.1.2. Vererbung

Smalltalk in seiner ursprünglichen Form unterstützt nur Einfachvererbung, aber es gibt (wenig verbreitete) Implementierungen von Smalltalk, die auch Mehrfachvererbung unterstützen. Jede neue Klasse muß in die Klassenhierarchie des Smalltalk-Systems, das die Vererbungsstruktur definiert, integriert werden.

Eine Subklasse erbt von ihrer unmittelbaren Superklasse und allen hierarchisch darüber liegenden Superklassen bis zu der an der Spitze dieser Hierarchie liegenden "Klasse aller Klassen", genannt `object`. Jede Klasse, ausgenommen `object` selbst, ist Subklasse der Klasse `object`.

Wird von einem Objekt eine Nachricht empfangen, so sucht es in seinen Methoden nach einer geeigneten. Wird in der Menge der Instanzmethoden keine passende gefunden, so wird in der Superklasse weitergesucht. Dieses Prinzip setzt sich nach oben in der Klassenhierarchie fort, bis entweder eine entsprechende Methode gefunden oder selbst in der Klasse `object` keine gefunden wird, was einen Laufzeitfehler (`message not understood`) zur Folge hat.

3.1.3. Typkonzept

Smalltalk ist grundsätzlich eine ungetypte Sprache, d.h. Variablen besitzen keinen expliziten (bzw. deklarierten) Typ. Es gibt daher im Smalltalk-System keine Typen, sondern ausschließlich Klassen. Klassen können jedoch entsprechend der Klassenhierarchie konvertiert werden.

3.1.4. Polymorphismus und dynamisches Binden

Da in Smalltalk grundsätzlich alle Objekte potentiell polymorph sind, werden Methoden zur Laufzeit, abhängig von der jeweiligen Ausprägung eines Objekts, dynamisch ausgewählt (*inclusion* Polymorphismus). In Smalltalk kann dieselbe Nachricht an verschiedene Objekte gesendet werden, wobei ihre Bedeutung durch das *class description protocol* der jeweiligen Klasse des diese Nachricht empfangenden Objekts festgelegt wird.

Um die Klassenkonsistenz zu gewährleisten, implementiert Smalltalk *coercion*: Werden in einem Ausdruck Zahlen unterschiedlichen Typs (d.h. unterschiedlicher Klassen) identifiziert, so werden diese in die allgemeinere der identifizierten Klassen konvertiert und erst danach die spezifizierte Operation durchgeführt.

Dynamisches Binden (in Smalltalk "spätes Binden" genannt) wird durch die sogenannte Subklassen-Verantwortung (*subclass responsibility*) ermöglicht: In der Superklasse wird die spezielle Methode lediglich definiert, nicht aber implementiert. Die verschiedenen Implementierungen müssen in den Subklassen erfolgen, daher bezeichnet man dies als die Verantwortung der Subklassen.

3.1.5. Kapselung

Die Datenkapselung wird in Smalltalk bei der Definition einer Klasse durch das *class description protocol* realisiert. Der Zugriff auf Instanzvariablen ist nur der Instanz selbst erlaubt. Objekte derselben Klasse können hingegen auf die ihnen gemeinsamen Klassenvariablen oder aber auch auf Pool-Variablen zugreifen. Pool-Variablen, die für Objekte unterschiedlicher Klassen zugreifbar sind, weichen das strenge

Datenkapselungskonzept auf. Da der Zugriff auf solche Variablen aber im *class description protocol* speziell festgelegt werden muß und Pool-Variablen typischerweise Variablen von Objekten einer anderen Klasse sind, wird der Zugriff darauf durch die von dieser Klasse zur Verfügung gestellten Methoden beschränkt.

Daneben können in Smalltalk auch global sichtbare Variablen, die fix im Smalltalk-System verankert bleiben, definiert werden. Es handelt sich dabei um System-Variablen (z.B. Drucker, etc.), die für einen effizienten Betrieb des Smalltalk-Systems erforderlich sind.

3.2. C++

Beginnend in den frühen achtziger Jahren wurde bei AT&T die Programmiersprache C++ entwickelt [18]. C++ ist eine hybride Sprache, die in sich die Ausdruckskraft einer objektorientierten Sprache mit den Möglichkeiten einer traditionellen, imperativen Programmiersprache vereinigt. Diese Zwitterrolle wurde bewußt in Kauf genommen, um effiziente, systemnahe (prozedurale) Programmierung zu erlauben, die in großer Menge existierenden C-Programme weiterverwenden zu können und Programmierer ohne großen Umschulungsaufwand an das objektorientierte Paradigma heranzuführen.

3.2.1. Klassen und Objekte

Es ist die grundlegende Philosophie von C++, nur eine kleine Menge an Basisklassen mit dazugehörigen, einfachen Operationen zur Verfügung zu stellen. Darüber hinaus werden keine höheren Klassen und Operatoren (z.B. Zeichenketten mit Verkettungsoperator, etc.) angeboten. Diese müssen vom Benutzer definiert werden, können dann allerdings gleich wie die in der Sprache vorhandenen einfachen Klassen und Operatoren verwendet werden.

Jedes Objekt ist Instanz einer bestimmten Klasse. Im Gegensatz zu Smalltalk sind allerdings Klassen in C++ keine Objekte. Eine Klasse wird definiert durch eine Menge von Datenelementen (*data members*) und Methoden (*member functions*). Zugriffsrechte auf Datenelemente und Methoden werden bei der Deklaration explizit festgelegt: *private* (nur innerhalb der Klasse zugänglich), *protected* (auch aus Subklassen zugreifbar) oder *public* (öffentliche Schnittstelle der Klasse). Zur Steigerung der Laufzeiteffizienz können Methoden `inline` deklariert werden, wodurch der Code einer Methode direkt an der Stelle ihrer Verwendung eingefügt wird (kein Funktionsaufruf).

Neben Objektvariablen und -methoden können auch Klassenvariablen (klassenglobale Variablen) bzw. -methoden (Konstruktoren/Destruktoren für die (De-)Instanzierung, *overloaded operator functions*, Konversionsfunktionen, etc.) definiert werden.

3.2.2. Vererbung

Das Konzept der Vererbung wird in C++ als *derivation*, eine Subklasse somit als *derived class* bezeichnet. Superklassen werden *base classes* genannt. Subklassen erben die *data members* und *member functions* aller in der Vererbungshierarchie über ihnen liegenden Klassen. Ererbte Methoden können dabei von der erbenden Klasse überladen, d.h. redefiniert werden. Im Unterschied zu Smalltalk existiert keine gemeinsame "Wurzel" der Vererbungshierarchie, welche die Superklasse aller Objekte und Klassen darstellt.

C++ bietet sowohl die Möglichkeit einfacher als auch mehrfacher Vererbung (vgl. [17]). Mehrdeutigkeitsprobleme (*name clashes*), wenn von gleichen *base classes* von mehreren Seiten geerbt wird, werden durch das Konzept der *virtual base classes* (*sharing* von *base classes*, vgl. [11]) vermieden. Müssen unterschiedliche Klassen von einer gemeinsamen Superklasse erben (z.B. bei *inclusion* Polymorphismus), ohne dabei aber eine Instanzierung der Superklasse zu erlauben, so unterstützt dies C++ durch *abstract base classes* (vgl. [11]).

3.2.3. Typkonzept

C++ ist *strongly typed*, d.h. erfolgreiche Übersetzung garantiert Typkonsistenz zur Laufzeit, auch wenn dabei nicht alle Typen eindeutig feststellbar sind (z.B. bei *virtual member functions*).

In C++ weist der Compiler allen Ausdrücken, unter Zuhilfenahme impliziter und expliziter Typkonversion, Typen zu und überprüft ihre Konsistenz. Dieses Schema kann der Programmierer um eigene Konversionsfunktionen erweitern, die gleichberechigt zu den vordefinierten angewendet werden (vgl. [11, 18]).

3.2.4. Polymorphismus und dynamisches Binden

C++ bietet alle der in Abschnitt 2.4. beschriebenen Formen von Polymorphismus. Um z.B. Methoden zu überladen, genügt es in C++, gleichnahmige Methoden mit unterschiedlichen Signaturen zu definieren. *Coercion* wird mittels vordefinierter Konversionen für einfache Typen und der zuvor bereits erwähnten Möglichkeit, eigene Konversionsfunktionen zu definieren, unterstützt.

Wie jede objektorientierte Sprache unterstützt C++ *inclusion* Polymorphismus. Solcherart polymorphe Methoden sind in C++ *virtual member functions*. Der Methodendefinition ist dabei das Schlüsselwort `virtual` voranzustellen, um sie so von *overloading* zu unterscheiden. Für diese Form des Polymorphismus ist in C++ dynamisches Binden notwendig.

Als weitere Form unterstützt C++ noch *parametric* Polymorphismus über den Mechanismus der *templates* (vgl. [11, 18]). Im Zusammenspiel mit Vererbung stellt dies eines der mächtigsten Ausdrucksmittel von C++ dar.

3.2.5. Kapselung

Eine Klasse in C++ definiert eine Datenkapsel mit abgestuften Zugriffsmöglichkeiten, was auch im Zuge der Vererbung beibehalten wird. Dabei existieren allerdings Lücken (vgl. [12]). Des weiteren kann dieses strikte Konzept durch sogenannte *friends* aufgeweicht werden: Wird eine Klasse

oder eine Methode als *friend* einer Klasse deklariert, so hat diese Klasse bzw. Methode Zugriff auf den privaten Teil der Klasse, in welcher diese Definition steht. Klassen und Methoden können über diesen Mechanismus also auf private Daten einer anderen Klasse zugreifen. Damit wird die Methodenschnittstelle einer Klasse umgangen, was zwar zu einer Erhöhung der Laufzeiteffizienz, aber auch zu einer groben Verletzung des Konzepts der Kapselung führt.

3.3. Objective-C

Objective-C wurde von Brad J. Cox [6] entwickelt und ist eine Erweiterung von ANSI C um objektorientierte Konzepte. Wie C++ ist Objective-C eine hybride Sprache, mit sowohl imperativen als auch objektorientierten Sprachmitteln. Im Unterschied zu C++, welches in der Tradition von Simula-67 steht, übernimmt Objective-C jedoch große Teile des Objekt-, Klassen- und *message passing*-Konzepts von Smalltalk (vgl. [6, 3]).

3.3.1. Klassen und Objekte

Eine Klasse wird durch Instanzvariablen, die den Zustand eines Objektes repräsentieren, und Methoden für den Zugriff auf diese Attribute definiert. Diese Definition zerfällt in einen Interface- und Implementationsteil. Ersterer deklariert Name, Position in der Vererbungshiererachie, Instanzvariablen und die öffentliche Schnittstelle der Klasse (Methoden). Zweiterer liefert die Implementierung der im Deklarationsteil definierten Methoden. Ähnlich wie in C++ können Zugriffsniveaus auf Instanzvariablen definiert werden (nicht aber auf Methoden), mit dem Unterschied, daß die Bedeutung von *private* und *protected* vertauscht ist. Im Unterschied zu C++ kennt Objective-C keine Klassenvariablen und unterscheidet auch syntaktisch explizit zwischen Klassen- und Instanzmethoden.

Ein Methodenaufruf ist wie in Smalltalk aus mehreren Teilen aufgebaut (vgl. Abschnitt 3.1.1.): dem Empfänger (*receiver*) der Nachricht, der Identifikation einer Methode (*selector*) und möglichen Argumenten, anhand derer zwischen unären bzw. binären Methoden und Methoden mit Schlüsselwörtern (*keyword message expressions*) unterschieden wird. Ein Methodenaufruf (*message passing*) ist nur innerhalb sogenannter *message expressions* erlaubt, wobei Laufzeitfehler (unbekannte Methoden) auftreten können. Methoden können zur Laufzeit dynamisch hinzugefügt bzw. entfernt werden (in C++ nicht möglich).

3.3.2. Vererbung

Das Vererbungskonzept von Objective-C ist an Smalltalk angelehnt und unterstützt daher nur Einfachvererbung. Es existiert, im Unterschied zu C++, genau eine Vererbungshierarchie mit der Klasse `Object` als Wurzel. Klassen erben von ihren Vorfahren alle Methoden und Instanzvariablen, die nicht als geschützt deklariert wurden. Die Auflösung von Methodenaufrufen wird ebenfalls wie in Smalltalk durchgeführt: Enthält der *receiver* einer Nachricht keinen passenden *selector* (i.e. es existiert keine passende Methode), so wird die Nachricht entlang der Vererbungshierarchie nach oben weitergereicht, um eine passende Methode zu finden. Scheitert dies, tritt ein Laufzeitfehler auf.

Neben den bekannten Möglichkeiten, ererbte Methoden zu redefinieren und neue Instanzvariablen hinzuzufügen, bietet Objective-C noch das Konzept des *posing*. Dabei kann eine Klasse unter Umgehung der strikten Vererbungshierarchie die Position einer anderen Klasse einnehmen (ganz bzw. nur einzelne Methoden), d.h. anstelle der adressierten Klasse wird transparent eine beliebige andere Klasse angesprochen.

3.3.3. Typkonzept

Objective-C besitzt auf Grund seiner hybriden Sprachnatur zwischen ANSI C (*statically typed*) und Smalltalk (*untyped*) kein homogenes Typkonzept (C++ ist im Gegensatz dazu eindeutig *strongly typed*). Grundsätzlich übernimmt Objective-C das von ANSI C definierte statische Typkonzept. Dadurch besteht die Möglichkeit, zur Übersetzungszeit Typinkonsistenzen festzustellen. Parallel und gleichberechtigt dazu besitzt Objective-C einen speziellen Typ für Objekte (`id`). Variablen dieses Typs können Objekte aus beliebigen Klassen referenzieren. Es existiert also nur ein “Objekttyp” ähnlich wie in Smalltalk. Aus diesem Blickwinkel ist Objective-C also als ungetypte Sprache zu betrachten, da sich Variablen des Typs `id` einer statischen Typprüfung entziehen. Sie können nur zur Laufzeit des Programmes klassifiziert werden.

3.3.4. Polymorphismus und dynamisches Binden

Im Gegensatz zu C++ erlaubt Objective-C einige Formen von Polymorphismus nicht bzw. nur eingeschränkt: *Overloading* wird nur für Methoden, nicht aber für Operatoren unterstützt. *Coercion* ist nur eingeschränkt, aus Kompatibilität zu ANSI C für einfache Typen möglich bzw. notwendig, da wie in Smalltalk nur Nachrichten an Objekte geschickt werden, die von diesen entweder korrekt behandelt werden können oder Laufzeitfehler verursachen. Objective-C kennt nur *inclusion* Polymorphismus als Ausprägung von *universal* Polymorphismus. Im Unterschied zu C++ wird diese Art (wie in Smalltalk) implizit angenommen und muß nicht speziell gekennzeichnet werden (vgl. Abschnitt 3.2.4.). *Parametric* Polymorphismus wird nicht unterstützt.

Dem Mechanismus des dynamischen Bindens kommt zentrale Bedeutung zu, da Objective-C in weiten Bereichen wie Smalltalk ungetypt ist (Objekttyp `id`) und Objekte als potentiell polymorph angesehen werden. Daher wird in der Regel dynamisches Binden verwendet, obwohl Objective-C auch statisches Binden unterstützt. Der beim dynamischen Binden verwendete Algorithmus entspricht jenem von Smalltalk.

3.3.5. Kapselung

Mit der Definition einer Klasse werden ihre Instanzvariablen gekapselt und die zugehörigen Zugiffsniveaus festgeschrie-

ben. Auf sie kann nur über die im Interface-Teil der Klasse angeführten Methoden zugegriffen werden (vgl. *class description protocol* von Smalltalk). Methoden selbst hingegen sind immer öffentlich zugänglich, d.h. es werden keine sprachseitigen Mittel zur Einschränkung des Benutzerkreises von Methoden geboten.

Wie in C++ kann das Konzept der Kapselung aufgrund der Manipulationsmöglichkeiten, die Objective-C als Obermenge von ANSI C gezwungenermaßen enthält, umgangen werden.

3.4. Eiffel

Eiffel ist eine Entwicklung von B. Meyer [14] und stellt eine streng objektorientierte Programmiersprache dar, die sich auf die Implementierung möglichst wiederverwendbarer Software konzentriert. Eiffel erleichtert weiters die Anwendung von *design by contract* (vgl. [13]) und verbindet damit Implementierung und Design.

3.4.1. Klassen und Objekte

Eine Klasse in Eiffel beschreibt eine Menge von Laufzeit-Objekten, die durch *features*, das sind ihre Attribute und Methoden, charakterisiert werden. Die Sichtbarkeit (und somit Verwendbarkeit) von *features* kann in Eiffel explizit bei der Klassendefinition festgelegt werden: *Generally available features* sind allen Klassen zugänglich. *Selectively available features* sind beschränkt für zugelassene Klassen sichtbar. *Secret Features* sind für keine andere Klasse zugreifbar.

Objekte, als Instanzierungen von Klassen, müssen durch Konstruktoren explizit zur Laufzeit erzeugt werden und werden dann Variablen (in Eiffel *entities* genannt) zugewiesen. Eiffel bietet die Möglichkeit, Eigenschaften von Klassen durch Zusicherungen (*assertions*) formal zu definieren. Es können Vor- und Nachbedingungen von Methoden sowie Klasseninvarianten definiert werden. Klasseninvarianten müssen von allen Instanzen einer Klasse erfüllt werden, wann immer sie von "außen" zugreifbar sind und repräsentieren somit allgemeine Konsistenzbedingungen für eine Klasse.

3.4.2. Vererbung

Eiffel unterstützt Mehrfachvererbung. Grundsätzlich erbt eine Subklasse alle *features* ihrer Superklassen, wobei jedoch einige Möglichkeiten zur Adaptierung der Subklasse geboten werden: Das syntaktische Konzept des *renaming* ermöglicht es, Namenskonflikte bei Mehrfachvererbung (z.B. namensgleiche Methoden in unterschiedlichen Superklassen) aufzulösen und ererbte *features* an lokale Gegebenheiten anzupassen.

Methoden können von Subklassen redefiniert werden (Zuweisung einer neuen Implementierung). Im Unterschied zu vielen anderen objektorientierten Sprachen, bietet Eiffel auch hier Möglichkeiten zur Erhaltung der semantischen Konsistenz von Methoden durch zusätzliche Zusicherungen (vgl. Abschnitt 3.4.1.). Eine redefinierte Version einer Methode muß eine schwächere oder gleiche Vorbedingung erfüllen und eine strengere oder gleiche Nachbedingung zusichern als das Original.

Im Zuge der Vererbung können sogenannten *deferred features* (diese werden in der ursprünglichen Klasse nur deklariert, nicht jedoch implementiert) mit dementsprechenden Implementierungen versehen werden. Eiffel bietet auch einen interessanten *join*-Mechanismus, der es ermöglicht, zwei *deferred features* mit kompatibler Signatur und Spezifikation, in der Subklasse mit einer Implementierung zu versehen.

3.4.3. Typkonzept

Eiffel ist ein Beispiel für eine *strongly typed* Sprache. Eiffel ist so in der Lage, Laufzeitfehler aufgrund von Typinkonsistenzen zu verhindern, bietet aber eine wesentlich bessere Laufzeit-Effizienz als dynamisch typüberprüfende Sprachen.

3.4.4. Polymorphismus und dynamisches Binden

Aufgrund seines klaren und puristischen Designs, existiert in Eiffel kaum *ad hoc* Polymorphismus. *Overloading* wird nicht unterstützt und *coercion* nur insoweit, als es für (implizite) Typkonversionen notwendig ist.

Parametric Polymorphismus wird mittels generischer Klassen realisiert, die mit formalen, generischen Parametern spezifiziert werden. Jede dieser Klassen beschreibt ein sogenanntes *type template*, von dem man durch Übergabe konkreter Typen (der aktuellen generischen Parameter) eine direkt verwendbare Klasse ableiten kann.

Eiffel bietet *inclusion* Polymorphismus durch die Möglichkeit, Eigenschaften von Klassen teilweise oder auch ganz zu redefinieren (vgl. Abschnitt 3.4.2.). Durch dynamisches Binden wird dann zur Laufzeit ermittelt, welche der redefinierten Versionen (d.h. welche konkrete Implementierung) tatsächlich ausgeführt werden soll. Eine Zuweisung der Form `a := b` ist in Eiffel zulässig, nicht nur wenn `a` und `b` vom selben Typ sind, sondern auch, wenn `b` Instanz einer Subklasse der Klasse von `a` ist (*inclusion* Polymorphismus). Die Fähigkeit einer Größe zur Laufzeit auf Instanzen unterschiedlicher Klassen verweisen zu können, wird in der getypten Umgebung von Eiffel durch das Vererbungsmodell eingeschränkt: Einer Variablen dürfen nur Variablen derselben Klasse oder einer ihrer Subklassen zugewiesen werden (Typverträglichkeitsregel von Eiffel), d.h. daß Variablen einer Klasse nur Variablen derselben Klasse oder einer spezielleren Klasse als Werte übernehmen dürfen.

In Eiffel werden dynamisches Binden und statische Typüberprüfung kombiniert, um einerseits zu garantieren, daß jeweils die richtige aktuelle Version verwendet wird und andererseits der Compiler garantieren kann, daß zumindest eine solche Version existiert. Damit können aufwendige Suchvorgänge zur Laufzeit vermieden werden, die Flexibilität andererseits aber wird eingeschränkt.

Durch die Möglichkeit, die Semantik schon bei der Spezifikation einer Methode (Vor- und Nachbedingungen) präzise festlegen zu können, eignet sich Eiffel auch als Entwurfssprache. Danach kann der Designer in der Implementierung der

Subklassen eine schrittweise Verfeinerung der Methodendefinitionen vornehmen (vgl. [14]).

3.4.5. Kapselung

Eiffel unterstützt die Datenkapselung explizit, d.h. es stehen Sprachmittel zur Verfügung, die eine Einschränkung des Zugriffs auf *features* einer Klasse ermöglichen. Wird keine Einschränkung (*export restriction*) angegeben, so sind die *features* einer Klasse allen anderen Klassen zugänglich. Die Sichtbarkeit der *features* kann durch Angabe der gewünschten Klassen in der Klassendefinition explizit definiert und somit eingeschränkt werden (i.e. *information hiding*).

3.5. Modula-3

Modula-3 wurde vom DEC Systems Research Center und Olivetti Research Center in den Jahren 1986 bis 1990, basierend auf Modula-2 (N. Wirth) bzw. Modula-2+ (DEC), mit welchen es jedoch nicht kompatibel ist, entwickelt [4, 15, 9]. Die Einfachheit und Typsicherheit seiner Vorgängersprachen wird erhalten und um wichtige Konzepte wie *exception handling*, *garbage collection*, *concurrency* und objektorientierte Programmkonstrukte erweitert.

3.5.1. Klassen und Objekte

Das strenge Typkonzept von Modula-3 definiert auch für Objekte jeweils einen bestimmten Typ, den sogenannten Objekttyp. In einer Klassendefinition findet sich die Definition für ihre Attribute und ihre Methoden, und die Postition der Klasse in der Vererbungshierarchie. Methoden werden durch ihre Signaturen in der Objekttypdefinition festgelegt. Die Implementierung einer jeden Methode findet sich dann außerhalb der Objektdefinition in Prozedurform. Es müssen sowohl die durch die Klassenhierarchie ererbten Methoden als auch die eigenen Methoden definiert werden, die zusammen die Methodenliste (*method suite*) eines Objekts darstellen.

Objekte sind in Modula-3 Referenzen auf einen Daten-Record gemeinsam mit einer Methodenliste. Auf Objekttypen als solche kann nicht referenziert werden, sondern nur auf einzelne Teile von Objekten (Attribute oder Methoden). Konstruktoren, die bei der Objekterzeugung benutzerdefinierten Initialisierungscode aufrufen, und Destruktoren zur benutzerdefinierten Speicherbereinigung werden von Modula-3 nicht unterstützt. Die Freigabe von Speicherbereichen ist die Aufgabe des *garbage collectors* (Teil des Laufzeitsystems).

3.5.2. Vererbung

Das Vererbungskonzept von Modula-3 erlaubt einer Klasse (Objekttyp), nur von genau einer Superklasse zu erben (i.e. Einfachvererbung). Subtypen erben sämtliche Attribute und Methoden aller ihrer Supertypen entlang der Vererbungshierarchie. Genauso ist es möglich, in einer Subklasse zusätzliches Verhalten und weitere Eigenschaften zu definieren bzw. zu redefinieren (vgl. dazu Abschnitt 3.5.4.).

Modula-3 definiert eine allgemeine Subtyprelation "<:", die auf unterschiedliche Typen, Objekte eingeschlossen, angewendet werden kann: Ist S ein Subtyp von T ($S <: T$), dann ist ein jeder Wert vom Typ S ebenso ein Wert von Typ T. Diese reflexive und transitive Relation ist auf Prozeduren, Arrays, Referenzen, Unterbereichstypen, gepackte Typen und Objekte anwendbar.

3.5.3. Typkonzept

Ähnlich wie Modula-2 definiert auch Modula-3 ein strenges Typkonzept. Um Polymorphismus zu ermöglichen, mußte Modula-3 allerdings vom statischen Typkonzept seines Vorgängers zu *strongly typed* übergehen.

3.5.4. Polymorphismus und dynamisches Binden

Modula-3 verwendet strukturelle Äquivalenz von Typen anstelle von Namensäquivalenz, wie z.B. Modula-2. Strukturelle Äquivalenz bedeutet, daß zwei Typen dann gleich sind, wenn ihre Definitionen nach Expandierung gleich sind. Indem Prozeduren, die einen bestimmten Typ T als Eingabetyp erwarten, auch Subtypen von T als Eingabeparameter akzeptieren, bietet Modula-3 *inclusion* Polymorphismus. Gleiches gilt auch für Objekte: Da Objekte einen Objektyp besitzen, kann auch auf sie diese Subtyprelation angewendet werden.

Wird eine neue Subklasse gebildet, so finden sich in der Methodenliste die Einträge der Superklasse und zusätzlich die neu definierten Methoden. Subklassen können ererbte Methoden redefinieren (*overriding*), wodurch in der *method suite* der Zeiger für die ursprünglich ererbte Methode auf eine andere Prozedur "umdirigiert" wird. Durch dynamisches Binden wird erst zur Laufzeit entschieden, welche konkrete Implementierung zur Ausführung gelangt. Darüberhinaus gibt es in Modula-3 die Möglichkeit, einen *supercall* durchzuführen: Dabei handelt es sich um den Aufruf einer Methode der Superklasse, die in ihrer Subklasse redefiniert wurde (vgl. [9, 15]).

Weiters bietet Modula-3 die Möglichkeit, generische *interfaces* (vgl. Abschnitt 3.5.5.) zu bilden (*parametric* Polymorphismus). Ein solches generisches *interface* ist eine Schablone, die Variablen, Typen oder auch Prozeduren definiert, die auf formalen Parametern des generischen *interface* operieren. Durch entsprechende Instanzierung werden dann die aktuellen Parameter im *interface* verwendet.

Während Modula-3 *overloading* für Operatoren realisiert, kennt es – wie sein Vorgänger Modula-2 – keine *coercion*. Eine notwendige Typkonversion muß vom Programmierer explizit durch dafür zur Verfügung gestellte Funktionen durchgeführt werden.

3.5.5. Kapselung

Modula-3 unterstützt das Konzept des *information hiding* durch die Notation von Modulen als Übersetzungseinheit. Während in anderen Sprachen *information hiding* durch das Klassenkonzept verwirklicht wird, unterscheidet Modula-3

hier die Definition von Klassen (Objekttyp) zur Abstraktion und das Konzept des Moduls für das *information hiding*. Ähnlich wie Objekte haben auch Module private Daten, die für Benutzer unzugänglich sind. Im Modul kann durch Exportieren festgelegt werden, welche Daten und Prozeduren öffentlich zugreifbar sind und somit von Benutzern verwendet werden können. Ein *interface* legt fest, welche Elemente eines Moduls allgemein zugänglich sind, wobei mehrere Implementierungsmodule zu einem *interface* möglich sind. Dies kann dazu verwendet werden, um einerseits eine allgemeine Schnittstelle und andererseits eine Schnittstelle für spezielle, besonders "vertrauenswürdige" Klienten anbieten zu können [2].

In Modula-3 gibt es auch die Möglichkeit, Objekte mittels *opaker* Typen zu abstrahieren und somit für Benutzer unnötige Details (z.B. Realisierung von Methoden, interne Daten, etc.) zu verbergen. Solche opaken Typen können in einem *interface* für das entsprechende Objekt gemeinsam mit weiteren Typen und Methoden den Benutzern angeboten werden. Es wird eine Superklasse gebildet, die nur die allgemeine (i.e. *public*) Information der opaken Klasse definiert und allgemein zugänglich ist. Die opake Klasse selbst ist für Benutzer nicht zugänglich, legt aber die eigentliche Implementierung der Klasse fest. In dieser opaken Klassendefinition werden die Attribute, die Methodenimplementierungen und ggf. private Prozeduren für die Realisierung solcher definiert.

4. Schlußbemerkungen

Eine vergleichende Betrachtung verschiedener objektorientierter Programmiersprachen bedarf aufgrund der teilweise in diesem Bereich herrschenden Begriffsvielfalt und unklaren Begriffsverwendung einer einheitlichen Definition der grundlegenden Konzepte und Begriffe. Der versierte Anwender, der mit den Grundkonzepten des objektorientierten Paradigmas bereits vertraut ist, soll sich, aufbauend auf den Begriffsdefinitionen im Kapitel 2., einen raschen Überblick über die Umsetzung dieser Konzepte in einigen der zur Zeit bedeutendsten objektorientierten Programmiersprachen verschaffen können.

Eine vergleichende Bewertung, welche Programmiersprache für eine bestimmte Anwendung am besten geeignet ist, hängt von vielen verschiedenen Einflußfaktoren (z.B. Eignung für den konkreten Anwendungsfall, Implementationsumfeld, etc.) ab und kann in einem solchen Artikel nicht gegeben werden. Dieser Artikel kann und soll jedoch dem Anwender einen Überblick als Grundlage für weitere Bewertungen geben und gegebenenfalls auch als Nachschlagewerk dienen.

Danksagung. Die Autoren danken Robert (ρ) Barta und Georg Trausmuth für ihre kritischen und hilfreichen Anmerkungen.

Literatur

[1] BOOCH, G.: *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway, Redwood City, California 94065, 2. Auflage, 1994.

[2] BÖSZÖRMENYI, L.: *A Comparison of Modula-3 and Oberon-2*. Structured Programming, 14:15–22, 1993.

[3] BUDD, T.: *An Introduction to Object Oriented Programming*. Addison-Wesley, 1991.

[4] CARDELLI, L., J. DONAHUE, L. GLASSMAN, M. JORDAN, B. KALSOW und G. NELSON: *Modula-3 Language Definition*. ACM SIGPLAN Notices, 27(8), August 1992.

[5] CARDELLI, L. und P. WEGNER: *On Understanding Types, Data Abstraction and Polymorphism*. ACM Computing Surveys, 17(4), Dezember 1985.

[6] COX, B.J.: *Object Oriented Programming. An Evolutionary Approach*. Addison-Wesley, 1986.

[7] GOLDBERG, A. und D. ROBSON: *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[8] GROGONO, P. und A. BENNETT: *Polymorphism and Type Checking in Object-Oriented Languages*. ACM SIGPLAN Notices, 24(11), November 1989.

[9] HARBISON, S.P.: *Modula-3*. Prentice-Hall, 1992.

[10] KORSON, T. und J.D. MCGREGOR: *Understanding Object-Oriented: A Unifying Paradigm*. Communications of the ACM, 33(9), September 1990.

[11] LIPPMAN, S.B.: *C++ Primer*. Addison-Wesley, zweite Auflage, 1992.

[12] LIU, C.-S.: *On the Object-Orientedness of C++*. ACM SIGPLAN Notices, 26(3), März 1991.

[13] MANDRIOLI, D. und B. MEYER: *Advances in Object-Oriented Software Engineering*. Prentice-Hall, 1994.

[14] MEYER, B.: *Eiffel: The Language*. Prentice-Hall, 1992.

[15] NELSON, G.: *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, 1991.

[16] PINSON, L.J. und R.S. WIENER: *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, 1988.

[17] SHAPIRO, J.E.: *An Example of Multiple Inheritance in C++: A Model of the Iostream Library*. ACM SIGPLAN Notices, 24(12), Dezember 1989.

[18] STROUSTRUP, B.: *Die C++ Programmiersprache: erweitert um Entwürfe zur ANSI-/ISO-Standardisierung*. Addison-Wesley, 1994.

[19] WEGNER, P.: *Dimensions of Object-Based Language Design*. In: *Proceedings of OOPSLA '87*, Seiten 168 – 182, 1987. Veröffentlicht in *ACM SIGPLAN Notices* 22(12).

[20] WEGNER, P.: *Concepts And Paradigms Of Object-Oriented Programming*. In: *OOPS Messenger*, Seiten 7 – 87. ACM Press, 1990.