Technical University of Vienna
Information Systems Institute
Distributed Systems Group

Object-oriented concepts in Smalltalk, C++, Objective-C, Eiffel and Modula-3

Harald Gall, Manfred Hauswirth and
René Klösch
H.Gall@infosys.tuwien.ac.at
M.Hauswirth@infosys.tuwien.ac.at
R.Kloesch@infosys.tuwien.ac.at

TUV-1841-95-03 March 17, 1995

This paper introduces the reader to object-oriented concepts and surveys several object-oriented programming languages. The object-oriented paradigm has gained wide-spread acceptance in the last decade. Numerous object-oriented languages have been developed and object-oriented features have been added to conventional programming languages. We define a consistent terminology and explain important concepts of object-orientation. We then classify prominent languages (Smalltalk, C++, Objective-C, Eiffel, Modula-3) according to the terminology and concepts.

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

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-4470
fax: +43 1 5058453
URL: http://www.infosys.tuwien.ac.at/

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

Harald Gall, Manfred Hauswirth, René Klösch

Technische Universität Wien

Institut für Informationssysteme,

Abteilung für verteilte Systeme

Email: {gall, hauswirth, kloesch}@infosys.tuwien.ac.at

Zusammenfassung

Die zunehmende Verbreitung des objekt-orientierten Paradigmas führte auch zur Entwicklung einer Reihe objekt-orientierter Programmiersprachen und zur Adaptierung bisher nicht objekt-orientierter Sprachen um dementsprechende Sprachkonzepte.

Dieser Artikel untersucht einige der verbreitesten objekt-orientierten Programmiersprachen hinsichtlich der ihnen jeweils zugrundeliegenden objekt-orientierten Konzepte. Die Programmiersprachen werden bezüglich der charakteristischen objekt-orientierten Konzepte analysiert und gegenübergestellt.

Dabei werden zuerst die grundlegenden objekt-orientierten Konzepte vorgestellt und beschrieben, sowie eine einheitliche Terminologie eingeführt, um einen konsistenten Vergleich der ausgewählten Programmiersprachen, die für ähnliche Konzepte zum Teil eine Vielfalt unterschiedlicher Bezeichnungen verwenden, zu ermöglichen.

In diesem Artikel wird versucht, speziell dem Anwender einen Überblick über objekt-orientierte Konzepte und deren Realisierung in verbreiteten objekt-orientierten Programmiersprachen zu geben.

Schlüsselwörter

Objekt-orientierte Programmierung und Programmiersprachen, Objekt- und Klassendefinition, Abstraktion, Datenkapselung, Polymorphismus, Vererbung, dynamisches Binden.

1 Motivation

Seit einigen Jahren ist der Begriff der Objektorientierung eines der gebräuchlichsten Schlagwörter der Informatik, wenn von innovativen Produkten, Software-Engineering, -design bzw. Wiederverwendung, moderneren, "besseren" Programmiersprachen, leistungsfähigeren Datenbank-Architekturen, usw. die Rede ist. Objektorientierung ist Mode geworden, mit allen positiven wie auch negativen Implikationen, die sich daraus ergeben.

Vorteile, die sich aus dieser Entwicklung ergeben, sind die Verbreitung der Konzepte nicht nur in der wissenschaftlichen Welt, sondern auch unter Anwendern[1], die durch positive Resonanz geförderte Weiterenwicklung der Theorie, das Einfließen des objektorientierten Paradigmas in neue kommerzielle Produkte bzw. den Software-Entwicklungsprozeß im allgemeinen.

Andererseits wird des öfteren ein etwas verklärtes bzw. verfälschtes Bild gezeichnet, indem das objekt-orientierte Paradigma als "Allheilmittel" für viele schwer lösbare Probleme präsentiert wird, obwohl noch wesentliche, für eine durchgehende und allgemeine Anwendung notwendige Teile bzw. Fragestellungen, wie ein kompletter objektorientierter Softwareentwicklungszyklus oder die Probleme, die sich bei der Verwendung objekt-orientierter Konzepte im Bereich verteilter Systeme auftreten (z.B. *intra-* bzw. *interobject-concurrency*, Spezifikation temporaler und funktionaler Abhängigkeiten etc.), nicht definiert bzw. erforscht sind.

Objektorientierung ist ein Bereich, der einer besonders hohen Dynamik unterworfen ist, sodaß laufend eine Fülle neuer Vorschläge und deren praktische Umsetzungen auftauchen. Dadurch erweitert sich jedoch auch die Kluft zwischen Theorie und Praxis, zwischen Wissenschaft (mit den als mögliche Verwirklichungen angebotenen Implementierungen) und Anwendern, im Sinne von Softwareproduzenten bzw. Systemhäusern, die nach geeigneten Werkzeugen für neue Projekte suchen.

Grundsätzlich läßt sich daher feststellen:

1. Es existieren fundierte theoretische Grundlagen, welche allgemein akzeptierte Basiskonzepte definieren und als Grundstock für darauf aufbauende, weitere Entwicklungen dienen können.

2. Die Umsetzung der klaren theoretischen Basiskonzepte geschieht auf verschiedenste Art und Weise, wobei sie des öfteren aus unterschiedlichsten Gründen auch verwaschen oder abgeschwächt werden (Beispiel: C++).

3. Der Anwender sieht sich einer Unmenge an Umsetzungen bzw. Realisierungen gegenüber[2] und ist gefordert, dieselben nach ihren Möglichkeiten bzw. ihrer Verwendbarkeit zu beurteilen. Darunter ist in diesem Zusammenhang die Bewertung

[1] Das Erreichen einer "kritischen Masse", die die allgemeine Akzeptanz eines neuen Paradigmas erst ermöglicht, wird begünstigt.

[2] Man betrachte bloß den Bereich der objekt-orientierten Programmiersprachen oder Entwicklungsumgebungen inklusive derer, die vorgeben, objekt-orientiert zu sein.

der verwendeten objekt-orientierten Konzepte und deren Verwirklichung zu verstehen. Das heißt natürlich, daß ein fundiertes Wissen objektorientierter Konzepte vorhanden sein muß, um unterschiedliche Ansätze, Unschärfen bei der Umsetzung derselben oder Ausmaß und Vollständigkeit von Implementierungen beurteilen zu können.

In diesem Artikel wollen wir den Anwender bei dieser Beurteilung unterstützen, indem wir einen klaren und einfachen Überblick über die Konzepte des objekt-orientierten Paradigmas geben und einige objekt-orientierte Programmiersprachen hinsichtlich der Vollständigkeit ihrer Umsetzung der Konzepte, einschließlich ihrer "Auffassung" derselben, sowie der jeweiligen Umsetzung betrachten.

2 Konzepte objektorientierter Programmierung

In diesem Abschnitt soll der Rahmen in bezug auf die objekt-orientierten Konzepte abgesteckt werden, anhand derer in den folgenden Abschnitten die Programmiersprachen hinsichtlich deren Unterstützung und Umsetzung betrachtet werden.

Die Konzepte sind dabei folgendermaßen kategorisiert:

1. Basiskonzepte, die eine Programmiersprache *erfüllen muß*, um überhaupt als objekt-orientiert gelten zu können.
2. Zusätzliche Konzepte, die eine Sprache zwar erst zu einem praktisch verwendbaren Werkzeug werden lassen, aber für die Klassifikation als objekt-orientiert nicht relevant sind.

2.1 Grundlegende Konzepte

Nicht jede Programmiersprache, die Objekte unterstützt, ist deshalb auch als objekt-orientierte Programmiersprache zu bezeichnen. Vielmehr muß eine Programmiersprache über das Objekt-Konzept hinaus weitere Mechanismen unterstützen, um als eine objekt-orientierte Sprache bezeichnet werden zu können.

Nach der Definition von Wegner in [Weg87][3] ist eine Sprache dann objekt-orientiert, wenn sie folgende Sprachmittel zur Verfügung stellt:

- Objekte
- Klassen

[3] *object-oriented = objects + classes + inheritance*

- Vererbung

Neben den tatsächlich objekt-orientierten Programmiersprachen gibt es jedoch auch noch Programmiersprachen, die lediglich Objekte unterstützen (z.B. Ada, Actors, etc.) und von Wegner [Weg87] als sogenannte *objekt-basierte* Sprachen bezeichnet werden.[4]

In der Folge wenden wir uns jedem dieser grundlegenden Konzepte objekt-orientierter Programmiersprachen im Detail zu.

2.1.1 Objekte

Ein *Objekt* ist als Abstraktion eines "realen Dinges" zu verstehen, das einen inneren Zustand besitzt, auf den nur mit Hilfe von Operationen (*Methoden*) zugegriffen werden kann. Als "reales Ding" ist in diesem Zusammenhang alles, was intellektuell erfaß- bzw. formulierbar ist, zu verstehen[5]. Für unsere weiteren Ausführungen wollen wir uns daher auf folgende Objektdefinition beziehen:

> *Objekte sind Mengen von Operationen, die einen Zustand teilen* [Weg90].
>
> *Ein Objekt hat einen Zustand, zeigt ein genau definiertes Verhalten und besitzt eine eindeutige Identität* [Boo93].

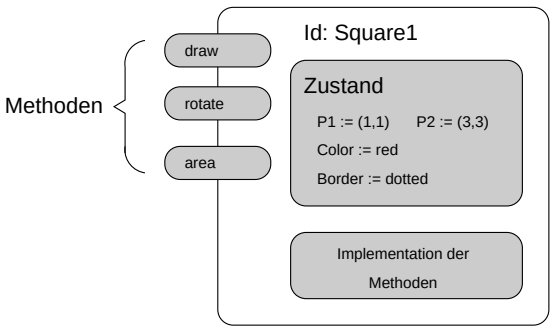

Abbildung 1: Ein einfaches Objekt

Der innere Zustand eines Objektes wird durch seine Attribute (vgl. `P1, P2, Color, Border` in Abb. 1) repräsentiert. Diese Attribute können nur mittels der Methoden des Objektes manipuliert werden (Datenkapselung).

[4] *object-based = objects*

[5] Der Leser möge über diesen kurzen Ausflug in den Bereich der Philosophie hinwegsehen. Wir verlassen dieses Schlachtfeld der Meinungen, ohne uns weiter mit den – in unserem Kontext irrelevanten – unterschiedlichen Objektbegriffen klassischer und moderner Denker auseinanderzusetzen.

Die Menge der Methoden (z.B. `draw, rotate, area` aus Abb. 1) legt fest, auf welche *Nachrichten* (Methoden-Invokationen) ein Objekt reagieren kann. Dadurch wird implizit seine Schnittstelle nach außen festgelegt, wohingegen sein innerer Zustand vor der Umgebung verborgen bleibt und nur über die zur Verfügung gestellten Methoden zugreifbar bzw. veränderbar ist. Die zulässigen Folgen von Methodenaufrufen zwischen Objekten werden als Protokoll bezeichnet.

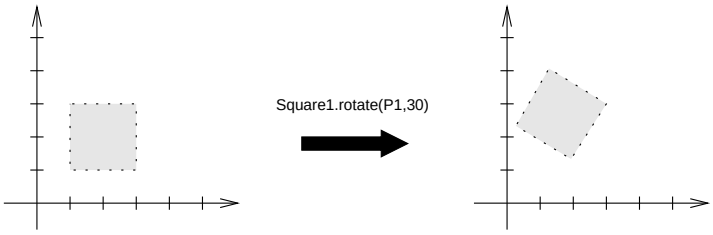

Abbildung 2: Anwendung einer Methode

Die Dynamik eines objekt-orientierten Systems (i.e. der Kontrollfluß) wird durch den Austausch von Nachrichten zwischen Objekten (i.e. Aufruf von Methoden) realisiert. Soll beispielsweise das Objekt `Square1` um 30 Grad gedreht werden, so erfolgt dies durch Senden der Nachricht `Square1.rotate(P1,30)` (vgl. Abb. 2).

Der erste Eindruck, Methoden seien – im theoretischen Sinn – das gleiche wie Funktionen bzw. Prozeduren trügt. Vergleicht man die drei Konzepte so zeigt sich folgendes:

- Eine “reine” Funktion (im Sinne von [Weg90]) hat kein “Gedächtnis” (i.e. inneren Zustand). Unabhängig von vorangegangenen Aufrufen, liefert eine Funktion ein Ergebnis, das nur von den Funktionsparametern abhängig ist.
- Prozeduren liefern ein Ergebnis, abhängig von ihrer Umgebung (globale Variablen, Zeigervariablen, o.ä.) und ihren Parametern. Sie können ihre Umgebung – implizit oder explizit – verändern.
- Eine Methode liefert ein Resultat (im streng objekt-orientierten Sinn handelt es sich um ein Objekt), das in Abhängigkeit von den Aufrufparametern und dem innerem Zustand des Objekts berechnet wurde.

Objekte werden vielfach als Instanzen einer Klasse (vlg. Abschnitt 2.1.2), ihre Attribute dementsprechend als Instanzvariablen und ihre Methoden als Instanzmethoden bezeichnet. Die Instanzvariablen sind Platzhalter für den jeweiligen Zustand der Instanz und Instanzmethoden verwalten den internen Zustand eines Objekts.

Objektidentität

Ein Objekt besitzt als spezielle Eigenschaft eine (systemweit) eindeutige Kennung (Objektidentität), die es von allen anderen Objekten unterscheidet. Die Objektidentität wird vom Laufzeitsystem der jeweiligen Programmiersprache vergeben, ist nicht änderbar und bestimmt eindeutig Zustand und Verhalten eines Objekts[6].

In der objekt-orientierten Programmierung ist es wichtig, den Unterschied zwischen dem Namen eines Objektes und dem Objekt selbst genau abzugrenzen. Wird ein Objekt deklariert, so erhält es vom Programmierer einen *Objektnamen*. Dahinter verbirgt sich nichts anderes als ein Speicherbereich, der eine *Referenz* auf ein Objekt aufnehmen kann, nicht aber ein Objekt selbst. Das Objekt selbst wird erst nach Verwendung einer Erzeugungsmethode (in den meisten Sprachen heißt diese einfach `new`) erzeugt. Beim Erzeugen wird dem Objekt eine eindeutige, vom Laufzeitsystem vergebene und verwaltete, *Objektidentität* zugewiesen und das Objekt über eine Referenz (z.B. die Objektidentität) mit dem Objektnamen verbunden. Es sei an dieser Stelle nochmals betont, daß die Identität eines Objektes sowohl von seinen Werten (zwei idente Quadrate können unterschiedliche Objekte repräsentieren), als auch von seinem Namen, der vom Programmierer vergeben wurde, als auch vom Speicherplatz, an dem es abgelegt ist, unabhängig und verschieden ist (vergleiche dazu [Weg90]).

Dieser Mechanismus kann jedoch auch zu problematischen Situationen führen, falls z.B. nach einer Zuweisungsoperation zwei Objektnamen (Variable) auf ein und dasselbe Objekt verweisen (*structural sharing*). Diese Situation ist deshalb gefährlich, da sie die Möglichkeit bietet, den Zustand eines Objektes über einen Namen zu ändern, ohne daß dies von Verwendern des zweiten Namens bemerkt wird.

Objekt-orientierte Programmiersprachen stellen daher unterschiedliche Formen von Kopier-, Zuweisungs- und Äquivalenzoperationen zur Verfügung, die sich aus der oben beschriebenen Verwendung der Objektidentität ergeben. Dies heißt z.B. *structural sharing* über einfache Zuweisungsoperatoren, Kopieroperatoren zum Vervielfältigen eines Objektes (mit/ohne *state sharing*), Operatoren zur Feststellung der unterschiedlichen Arten von Äquivalenz (Äquivalenz der Objektidentität[7], des Objektzustandes, Objektverhaltens, etc.). In jedem Fall ist wichtig, für jede Programmiersprache bei der Verwendung von Zuweisungs- bzw. Vergleichsoperatoren deren zugrundeliegende Semantik in bezug auf die Objektidentität genau zu berücksichtigen.

[6] Die umgekehrte Annahme, aus dem Zustand und dem Verhalten eines Objektes auf dessen Identität schließen zu können, ist nicht richtig!

[7] Der Vergleich eindeutiger Kennungen reicht hier nicht unbedingt aus. Die Semantik der Äquivalenz spielt eine zentrale Rolle. Wenn beipielsweise eine Person aus unterschiedlicher Sicht betrachtet wird – einmal als Autobesitzer, ein anderes mal als Steuerzahler – ist eine Verknüpfung der Identitäten in vielen Fällen notwendig und sinnvoll (vergleiche dazu [Weg90]).

2.1.2 Klassen

Mit dem Konzept des Objekts eng verbunden ist der Begriff der Klasse. Eine Klasse beschreibt das gemeinsame Verhalten, die Struktur und die Hierarchie (in bezug auf Vererbung) einer Menge von Objekten.

> *Eine Klasse ist eine Menge von Objekten, die eine gemeinsame Struktur und ein gemeinsames Verhalten besitzen* [Boo93].

Wie in der realen Welt dient also das Konzept der Klassen dazu, Objekte nach ihren Fähigkeiten, Attributen und ihrer Struktur einzuteilen. Eine Klasse beschreibt die Abstraktion eines Begriffes, wohingegen ein Objekt eine Instanzierung, ein konkretes Vorkommnis dieser Abstraktion darstellt.

Man definiert z.B. eine Klasse `Square`, in der alle Eigenschaften eines Vierecks (z.B. Eckpunkte, Farbe, Art der Umrandung, etc.) und die darauf anwendbaren Methoden (Anzeigen, Berechnen des Flächeninhalts, Drehen um einen Punkt, etc.) beschrieben sind (vgl. Abb. 3). Diese Abstraktion wird dann zur Laufzeit des sie beinhaltenden Programms in eines oder mehrere konkrete Objekte (Instanzen) umgesetzt.

```
class Square : public Polygon
{
    private:                // accessible only within the class

        VectorType    P1, P2;
        ColorType     Color;
        BorderType    Border;


    public:                 // accessible outside the class

        Square();           // the constructor: a class method

        void  draw(void);   // instance methods
        float area(void);
        void  rotate(VectorType center, float angle);
};
```

Abbildung 3: Eine einfache Klassendefinition in C++

Eine Klasse ist also eine Schablone, nach der Objekte (dieser Klasse) erzeugt werden können. Ein Objekt selbst ist keine Klasse, obwohl umgekehrt im streng objektorientierten Sinn (jedoch abhängig von der verwendeten objekt-orientierten Programmiersprache) eine Klasse ebenfalls ein Objekt ist.

Klassen besitzen ebenso wie Objekte interne Daten und Methoden. Interne Daten einer Klasse werden als Klassenvariablen, die Methoden einer Klasse als Klassenmethoden bezeichnet. Klassenvariablen stellen Platzhalter für allgemeine Informationen dar, die alle

Objekte dieser Klasse betreffen. Klassenmethoden beziehen sich ebenfalls auf alle Objekte der Klasse und sind Methoden, die beispielsweise zum Erzeugen und Initialisieren (z.B. `Square` in Abb. 3) oder zum Löschen von Objekten (Instanzen, Vorkommnissen) dieser Klasse dienen.

Klasse = Typ ?

Eine wichtige Frage, die sich im Zusammenhang mit Klassen stellt, ist, ob Klassen mit Typen klassischer Programmiersprachen gleichgesetzt werden können. Obwohl es sich dabei um eine von Sprache zu Sprache variierende Entscheidung handelt, die keinen Einfluß, im Sinne der Definition, auf die Objektorientiertheit einer Sprache hat, ergeben sich daraus gravierende Konsequenzen in Bezug auf Prüfmechanismen bei der Übersetzung bzw. zur Laufzeit von Programmen.

Typen werden als Prädikate über Ausdrücke definiert, die der Typüberprüfung eines Compilers dienen. Klassen hingegen spezifizieren Schablonen für die Erzeugung von Objekten mit gleichen Attributen und gemeinsamen Verhalten im Zuge der Applikationsentwicklung, dienen also dem Entwickler. Aus dieser unterschiedlichen Sichtweise und dem unterschiedlichen Zweck ergibt sich eine ebenso verschiedene "Verwandtschaftsrelation" zwischen Typen und Subtypen bzw. Klassen und Subklassen:

- Subtypen werden durch zusätzliche Prädikate, die die Struktur von Ausdrücken beschränken, definiert ([Weg90]). Dies bedeutet, daß ein Subtyp eine Untermenge (z.B. einen eingeschränkten Wertebereich) eines Typs spezifiziert (i.e. intensionale Sichtweise).
- Subklassen hingegen dienen zur (beliebigen) Modifikation des Verhaltens einer ("Eltern-") Klasse, wodurch sich eine viel losere, verhaltenserweiternde Verwandtschaftsrelation ergibt ([Weg90]). Subklassen können im Prinzip ein von ihrer/ihren Elternklasse(n) vollkommen verschiedenes Verhalten definieren[8].

Allgemein kann gesagt werden, daß zwar jede Klasse ein Typ ist – durch ein Prädikat, welches die Schablone spezifiziert – der Umkehrschluß allerdings nicht gültig ist, weil Prädikate nicht notwendigerweise Schablonen definieren.

Darauf näher einzugehen, würde den Rahmen dieses Artikels bei weitem sprengen, daher sei hier nur diese einfache, unvollständige Beschreibung des Problems gegeben und der interessierte Leser auf [Weg90] bzw. [Liu91] verwiesen.

2.1.3 Vererbung

Das Konzept der Vererbung, das bereits in Abschnitt 2.1.2 erwähnt wurde, soll nun präzisiert werden. Unter Vererbung ist jener Mechanismus zu verstehen, der es ermöglicht, Eigenschaften und Fähigkeiten von Klassen an andere Klassen weiterzugeben.

[8] Davon sollte im Sinne eines guten Programmierstils nur diszipliniert Gebrauch gemacht werden.

Vererbung definiert ein “natürliches” Ordnungsprinzip von Klassen, bzw. in weiterer Folge von Objekten, durch eine Verwandtschaftsrelation: Eine vererbende Klasse (Superklasse, Vorfahre, Basisklasse) gibt Eigenschaften an eine erbende Klasse (Subklasse, abgeleitete Klasse) weiter, die von dieser erweitert bzw. modifiziert werden können. Dies bedeutet, daß die Subklasse *zusätzliche* Definitionen (Attribute und Methoden) aufweisen kann, um die Eigenschaften der Superklasse zu verändern. Die Subklasse kann die Eigenschaften bzw. Methoden der Superklasse jedoch auch für sich modifizieren (i.e. Redefinition).

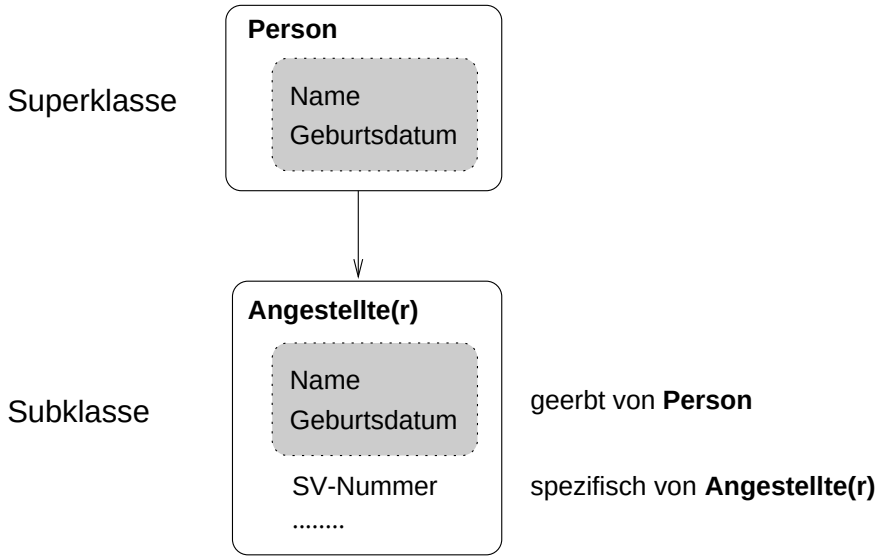

Abbildung 4: Vererbung von Attributen

Vererbung bietet also die Möglichkeit von *Programmierung durch Erweiterung* und ermöglicht somit implizit die Weiterverwendung von Verhaltensmustern und Zustandsräumen einer Klasse durch die Definition neuer Subklassen (*Software-Reuse*). Durch Vererbung wird eine *is-a*-Relation definiert, aus der sich eine Hierarchie von Sub- und Superklassen ableitet, die als Klassenhierarchie bezeichnet wird.

Die in Abb. 5 dargestellte Vererbungshierarchie zeigt die sogenannte *einfache Vererbung*: Eine Subklasse erbt von *genau einer* Superklasse. Vererbung muß allerdings nicht darauf beschränkt sein. Eine Subklasse kann auch von mehreren Superklassen erben. Man spricht dann von *mehrfacher Vererbung*. Eine Klasse `CompFigur` könnte beispielsweise von den Klassen `Polygon` und `Ellipse` erben. In diesem Fall präsentiert sich der Vererbungsgraph nicht als Baumstruktur, wie bei einfacher Vererbung, sondern – allgemeiner – als azyklischer, gerichteter Graph (Gitterstruktur) mit $n : m$ anstelle von *n:1 is-a*-Relationen.

Mehrfache Vererbung entspricht eher der Modellierung der realen Welt, kann jedoch

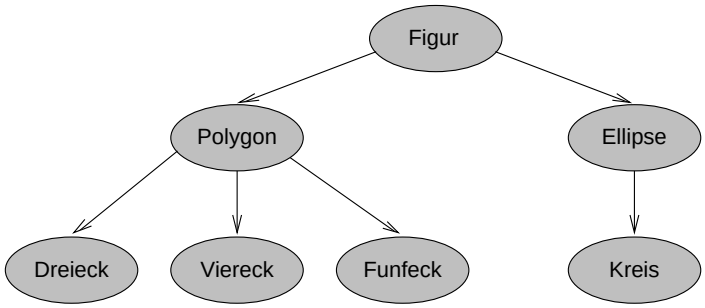

Abbildung 5: Beispiel für eine Vererbungshierarchie [KM90]

aufgrund der entstehenden Gitterstruktur bei Wartungsoperationen zu unangenehmen “Fernwirkungen” führen. Der Mächtigkeit dieses Modellierungskonstrukts steht die große Gefahr der Unüberschaubarkeit und daraus resultierender unerwünschter Nebeneffekte gegenüber.

2.2 Weitere objektorientierte Konzepte

Neben den im vorigen Abschnitt erwähnten Basiskonzepten gibt es noch weitere objektorientierte Konzepte, die von den existierenden Programmiersprachen in unterschiedlichem Umfang realisiert bzw. zur Verfügung gestellt werden.

- Datenabstraktion
- Kapselung
- Typkonzept
- Polymorphismus
- Dynamisches Binden

Diese Konzepte sollen im folgenden kurz erklärt werden.

2.2.1 Datenabstraktion

Eine allgemeine Definition dieses Begriffes im objekt-orientierten Sinne gibt Wegner in [Weg87] folgendermaßen:

> *Eine Datenabstraktion ist ein Objekt, dessen Zustand nur durch seine Operationen zugreifbar ist.*

Datenabstraktion wird in den meisten Programmiersprachen durch *abstrakte Datentypen* umgesetzt. In frühen Phasen der Software-Entwicklung (z.B. Spezifikation) ist man oft nicht daran interessiert (oder auch nicht in der Lage), die Datentypen eindeutig und vollständig zu definieren. Man beschränkt sich dann auf die Beschreibung der wesentlichen Eigenschaften, die ein Datentyp haben soll [EGL89]. Ein abstrakter Datentyp ist ein Modell, mit Hilfe dessen die gültigen Datenwerte und die darauf möglichen Operationen definiert werden. Wie man sofort sieht, erfüllt das objektorientierte Konzept der Klasse natürlich diese Bedingungen. Objektorientierte Sprachen stellen somit implizit Sprachmittel für die Datenabstraktion zur Verfügung.

2.2.2 Kapselung

Während Abstraktion darauf abzielt, die äußere Ansicht von Objekten zu definieren, wird mit Hilfe des Konzepts der Kapselung (*Encapsulation, Information Hiding*) versucht, eben diese Sicht als einzig zulässige für andere zu manifestieren. Kapselung soll das "Innenleben" von Objekten vor der "Außenwelt" verbergen.

Booch definiert Kapselung in [Boo93] wie folgt:

> *Kapselung ist der Prozeß des Versteckens aller Details eines Objekts, die nicht zu seinen essentiellen Charakteristiken beitragen*[9].

Kapselung bedeutet also nichts anderes, als daß die Umsetzung einer Abstraktion, also die Implementierung, die als eigentliche Repräsentation der Abstraktion zu verstehen ist, nach außen hin nicht sichtbar ist. Ein Objekt deklariert mithin nur seine Schnittstelle (*Interface*) zur Umgebung, ohne Auskunft auf die im Objekt vorhandenen Daten bzw. die tatsächliche Realisierung der auf das Objekt anwendbaren Methoden zu geben.

Um beispielsweise die in vorangegangenen Beipielen präsentierte Klasse `Square` verwenden zu können, müssen nur das Verhalten und die Möglichkeiten der Verwendung bekannt sein, nicht aber deren interne Struktur. Für einen Programmierer, der eine Klasse `Stack` verwendet, ist es irrelevant, ob der Stack intern als verkettete Liste oder als Array realisiert ist – das Interface wird davon nicht betroffen.

Für das Design und die Programmentwicklung ergibt sich durch das Konzept der Kapselung eine Reihe von Vorteilen:

- Eine abstrakte Sicht des Systems bleibt gewahrt, die das Verständnis eines Gesamtsystems wesentlich erleichtert bzw. oft erst ermöglicht.
- Kein Teil eines komplexen Systems hängt direkt von der internen Struktur eines anderen Teils des Systems ab.

[9] Booch [Boo93] zitiert dazu auch Liskow: "*Damit Abstraktion funktionieren kann, müssen die Implementierungen gekapselt sein.*"

- Änderungung an den Interna eines Objekts bleiben im Idealfall ohne Auswirkung auf die Umgebung des Objekts oder verursachen nur limitiert Anpassungen des Umfeldes (*Lokalität*).

Obwohl die meisten objektorientierten Programmiersprachen Kapselung unterstützen, ist dieses Konzept durchaus kein selbstverständlicher Bestandteil derselben: *Simula*, als klassische objekt-orientierte Sprache, erlaubt den Zugriff auf Instanzvariablen, in *CLOS* können Methoden außerhalb von Klassen definiert werden, usw.

2.2.3 Typkonzept

Typen dienen in Programmiersprachen unterschiedlichsten Zwecken, abhängig von welchem Standpunkt aus (Entwickler von Anwendungen, Compilern, etc.) sie betrachtet werden:

- Typen erhöhen die Lesbarkeit und Verständlichkeit von Programmen.
- Das verwendete Typkonzept beeinflußt die Effizienz und Geschwindigkeit der ausführbaren Programme, indem sie festlegen, in welchem Umfang Laufzeitüberprüfungen durchgeführt werden.
- Die Ablaufsicherheit von Programmen wird erhöht. Indem jedem Ausdruck ein Typ zugewiesen wird, kann durch (statische oder dynamische) Analyse festgestellt werden, ob ein Programm in bezug auf das Typkonzept korrekt bzw. konsistent ist.
- Mit Hilfe von Typen können Abstraktionen und Designentscheidungen ausgedrückt werden.
- Das Typkonzept einer Sprache bestimmt ihre Flexibilität und Eignung für die Lösung von Problemen aus unterschiedlichen Problemkreisen.

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.

Um Programmiersprachen nach dem verwendeten Typkonzept klassifizieren zu können wurden folgende Kategorien von Typkonzepten eingeführt (vgl. dazu [CW85], [ASU88], [KM90]):

- *Static typing*

  Der Typ jedes Ausdrucks in einem Programm kann durch statische Analyse (i.e. Analyse des Programmcodes durch den Compiler) bestimmt werden. Dadurch können

Inkonsistenzen und Fehler in bezug auf Typen schon bei der Übersetzung eines Programms erkannt werden und auf diese Weise dadurch möglicherweise verursachte Laufzeitfehler ausgeschlossen werden. Diese Forderung kann allerdings für etliche Anwendungsfälle zu restriktiv sein.

Beispiele für Sprachen dieses Typs sind: Ada, Modula-2, Pascal.

- *Strong typing*

  Darunter ist die Garantie zu verstehen, daß alle Ausdrücke innerhalb eines Programmes *typkonsistent* sind, auch wenn der Typ von (einigen oder allen) Ausdrücken im Zuge einer statischen Analyse nicht festgestellt bzw. festgelegt werden kann. In objektorientierten Sprachen kann nämlich – bei Ausnützung der Konzepte Vererbung bzw. Polymorphismus – der Fall eintreten, daß der dynamische Typ eines Objektes zur Laufzeit sich vom statisch im Programmtext verwendeten Typ unterscheidet. Dies ist beispielsweise der Fall, wenn Objekte aus Subklassen in einem Superklassenkontext verwendet werden oder Subklassen Methoden von Superklassen redefinieren (siehe dazu die Beipiele in Abschnitt 2.2.4). Die Feststellung des tatsächlichen Typs wird durch zusätzliche Laufzeitüberprüfungen bewerkstelligt. In Sprachen mit *strong typing* ist also sichergestellt, daß ein Programm, wenn es sich fehlerlos übersetzen läßt, ohne Typfehler abläuft.

  Beispiele für Sprachen dieses Typs sind: C++, Modula-3, Eiffel.

- *Untyped*

  Es existiert kein explizites Typkonzept und es wird daher keine Typüberprüfung durchgeführt. Das bedeutet, daß zur Laufzeit durch "Typinkonsistenzen" verursachte Fehler auftreten können. Es kann also passieren, daß ein Objekt eine Nachricht (Methodeninvokation) erhält, die in der Klasse, der das Objekts angehört bzw. in deren Superklassen nicht definiert wurde. In dieser Situation kommt es zu einem Laufzeitfehler, da das Objekt nicht "weiß", wie auf die Nachricht zu reagieren ist.

  Durch das Fehlen eines Typkonzepts wird allerdings auch eine hohe Flexibilität erreicht, die vor allem bei der Erstellung von Prototypen (*rapid prototyping*) und im Bereich der künstlichen Intelligenz wichtig bzw. notwendig ist, da hier sehr oft, anstelle eines spezifischen Problems, ganze Problemklassen zu lösen sind.

  Bespiele für Sprachen dieses Typs sind: Smalltalk, Lisp, CLOS.

Objektorientierte Sprachen variieren sehr stark im verwendeten Typkonzept und bieten jede Art der hier angeführten Typkonzepte.

2.2.4 Polymorphismus

Polymorphismus stellt eines jener Konzepte des objektorientierten Paradigmas dar, welches die sehr gute Eignung und große Flexibilität objektorientierter Programmiersprachen

für Entwurf und Implementierung von Softwaresystemen begründet. Unter Polymorphismus ist die Fähigkeit einer Größe (Objekte und somit implizit die mit ihnen verbundenen Methoden, Funktionen, Variablen, etc.) zu verstehen, zur Laufzeit unterschiedliche Ausprägungen bzw. Formen annehmen zu können.

Anhand von Beispielen sei diese Definition näher erläutert[10]:

Beispiel 1: Gegeben sei ein Array mit planaren Figuren aus der in Abb. 5 gezeigten Vererbungshierarchie als Elementtyp. Um alle Figuren auf einer graphischen Oberfläche anzeigen zu lassen, kann aufgrund des Konzepts des Polymorphismus eine einfache Schleife programmiert werden, unabhängig davon, ob ein Dreieck, Viereck oder Fünfeck dargestellt werden soll. Die einzelnen Figuren werden einfach durch Anwendung der polymorphen Methode `draw` gezeichnet:

```
for (Item = 0; Item < MAX_ITEM; Item++)
{
    ArrayOfFigures[Item].draw();
}
```

In einer konventionellen, *monomorphen* Sprache hätten die einzelnen Figuren als Teile eines Variantenrecords und die Ausgaberoutine als `case`-Statement (für jede Figur separat) implementiert werden müssen. Bei jeder Hinzunahme eines neuen Figurentyps hätten außerdem sowohl der Record als auch die Routine geändert werden müssen, was man sich in objektorientierten Programmiersprachen aufgrund des Klassenkonzeptes und der "verteilten" Implementierung von `draw` in den verschiedenen Objekten ersparen kann. Die polymorphe Methode `draw` liefert in Abhängigkeit vom jeweiligen Objekt das entsprechende, richtige Ergebnis.

Beispiel 2: Es besteht die Möglichkeit, Funktionen zu schreiben, die in ihren formalen Parametern polymorph sind. Um beispielsweise Figuren nach ihrer Fläche sortieren zu können, kann folgende Funktion implementiert werden:

```
Figure &CompareArea (const Figure &fig1, const Figure &fig2)
{
    if (fig1.area() < fig2.area())
    {
        return fig1;
    }
    else
    {
        return fig2;
    }
}
```

Aufgrund des polymorphen Charakters der formalen Parameter sind daher Aufrufe möglich, die unterschiedliche Figuren aus der Vererbungshierachie miteinander vergleichen:

[10] Das Konzept des Polymorphismus interagiert stark mit anderen objekt-orientierten Konzepten. Darauf und auf die sich daraus ergebenden Einschränkungen wird in den Beispielen nicht explizit eingegangen. Für ausführliche Beschreibungen dieses Problemkreises sei auf [CW85], [Boo93] und [Mey93] verwiesen.

```
Square      S1;
Triangle    T1;
Circle      C1;
Figure      BiggerOne;
Figure      BiggestOne;

BiggerOne  = CompareArea (S1, T1);
BiggestOne = CompareArea (BiggerOne, C1);
```

Beispiel 3: Variablen können zur Laufzeit Objekte unterschiedlicher Klassen referenzieren:

```
Square      S1;
Circle      C1;
Figure      *TheFigure;

TheFigure = &S1;

// Some computations

TheFigure = &C1;
```

Polymorphie stellt also, wie an obigen Beispielen deutlich wird, eine Aufweichung des von *monomorphen* Programmiersprachen bekannten Typkonzeptes dar. Während herkömmliche monomorphe Programmiersprachen (C, Modula-2, etc.) davon ausgehen, daß Funktionen bzw. Prozeduren und ihre Parameter, Operatoren und deren Operanden, etc. einen eindeutigen Typ besitzen[11], erlauben polymorphe Sprachen für bestimmte Größen mehr als einen Typ. Um dennoch typkonsistente Programme zu erhalten, müssen natürlich Compiler und/oder Laufzeitsystem der jeweiligen Sprache entsprechende Konsistenzprüfungen vorsehen.

Zum genaueren Verständnis des Konzeptes der Polymorphie geben Cardelli und Wegner (vgl. [CW85]) die in Abb. 6 dargestellte Klassifikation:

Prinzipiell ist zwischen *ad hoc* und *universal* Polymorphismus zu unterscheiden[12]. *Ad hoc* Polymorphismus bedeutet, daß eine Funktion auf einer endlichen Menge von unterschiedlichen und möglicherweise nicht miteinander verwandten Typen operieren kann (vgl. Bsp. 1, Bsp. 2). Man könnte also eine *ad hoc* polymorphe Funktion als eine Menge monomorpher Funktionen betrachten, aus denen die richtige vom Compiler bzw. Laufzeitsystem ausgewählt wird. Für die Funktion existieren also mehrere Code-Teile, die die unterschiedlichen Ausprägungen implementieren.

Arbeiten Funktionen auf einer potentiell unbeschränkten Anzahl von Typen, so spricht man von *universal* Polymorphismus. In bezug auf die Implementierung bedeutet dies – im

[11] Man beachte, daß gerade die in Bsp. 3 dargestellte Referenzierung von Objekten unterschiedlichen Typs zur Laufzeit durch eine Variable in diesen monomorphen Sprachen unmöglich ist.

[12] Die unterschiedlichen Arten von Polymorphismus werden im folgenden der leichteren Verständlichkeit wegen anhand von Funktionen (Methoden) bzw. Operatoren erläutert. Es bleibt dem Leser überlassen, die Beschreibungen – soweit sinnvoll bzw. möglich – auf andere Programmiersprachenkonzepte zu übertragen.

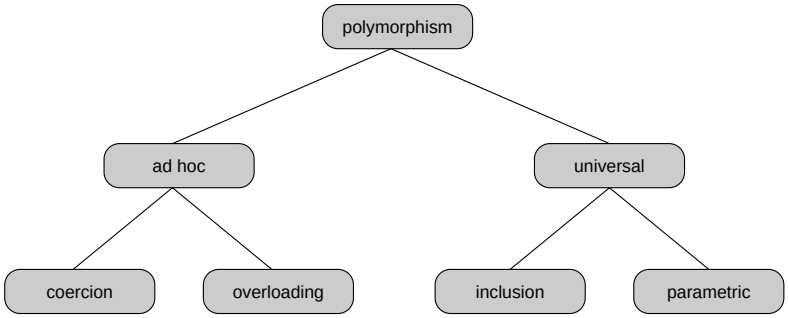

Abbildung 6: Arten von Polymorphismus [CW85]

Unterschied zu *ad hoc* Polymorphismus – daß *ein und derselbe* Code für Argumente beliebigen Typs ausgeführt wird. Aus diesem Grund wird *universal* Polymorphismus auch als *true polymorphism* und *ad hoc* Polymorphismus als *apparent polymorphism* klassifiziert. Diese beiden großen Klassen zerfallen in weitere Gruppen.

Die bekannteste Form von *ad hoc* Polymorphismus ist wohl *overloading*, da diese Art auch in nicht objekt-orientierten Programmiersprachen (z.B. Ada) verwendet wird. *Overloading* ist ein rein syntaktisches Konzept, das besagt, daß *ein* Name für *unterschiedliche* semantische Objekte (z.B. Funktionen, Operatoren) verwendet wird und aufgrund des Kontextes der Verwendung vom Compiler entschieden wird, welche Ausprägung zu verwenden ist. Der Operator `'+'` kann z.B. verwendet werden, um Integer- bzw. Real-Addition und Stringkonkatenation zu bezeichnen. Aus dem Kontext der Verwendung sollte klar sein, welche Funktionalität von `'+'` gemeint ist. Diese Form von Polymorphismus ergibt sich auch implizit aus dem Konzept der Vererbung, wenn z.B. Subklassen Methoden von Superklassen redefinieren, erweitern oder einschränken.

Im Unterschied zu *overloading* handelt es sich bei *coercion* um ein semantisches Konzept, das besagt, daß Argumente von Funktionen in jene Typen umgewandelt werden, die die Funktion erwartet. Dies bedeutet, daß man eine Funktion hat, die *einen* bestimmten Typ pro formalen Parameter erwartet, und daher vor Verwendung explizite oder implizite Typkonversionen der Argumente vom Compiler bzw. Laufzeitsystem durchgeführt werden müssen. Folgendes Beispiel aus [CW85] soll dies illustrieren:

```
3   +  4
3.0 +  4
3   +  4.0
3.0 +  4.0
```

Diese vier Additionen können mit *coercion* folgendermaßen interpretiert werden[13]: Der Operator `'+'` ist nur für reelle Zahlen definiert und ganze Zahlen werden vor Durchführung

[13]Wie Cardelli und Wegner anmerken, sind auch durchaus andere Interpretationsmöglichkeiten (siehe [CW85]), wie verschiedene Arten von *overloading*, möglich. Die Trennung zwischen *overloading* und

der Addition in reelle Zahlen umgewandelt. Mit Hilfe von *coercion* können also semantisch notwendige Typkonversionsoperationen entfallen, indem die Verantwortung dafür dem Compiler/Laufzeitsystem übertragen wird (implizite Typkonversion). Dies dient der Abkürzung von Programmtext und der Entlastung des Programmierers von solchen Überprüfungen.

Inclusion Polymorphismus als Untergruppe von *universal* Polymorphismus wurde zur Modellierung der Konzepte Vererbung bzw. Klassenhierarchie eingeführt. Dieses Modell besagt, daß ein Objekt als zu verschiedenen Klassen gehörig betrachtet werden kann, was sich implizit aus der Vererbungshierarchie ergibt (vgl. Abb. 5 und Bsp. 3). Jedes Objekt aus einer Subklasse (Subtyp) kann also in einem Superklassen- (Supertypen-) Kontext verwendet werden.

Mit Hilfe von *parametric* Polymorphismus wird das Prinzip der *Generizität* in eine Programmiersprache eingeführt. Eine, in dieser Art polymorphe Funktion besitzt explizite oder implizite Typparameter, die die Typen der Argumente für jede Anwendung bestimmen. Eine solcherart generische Funktion kann also ihre Aufgabe unabhängig vom Typ der Argumente durchführen. Man beschreibt in diesem Fall das Verhalten durch eine Schablone. So kann beispielsweise eine generische Klasse `Stack` definiert werden und über *parametric* Polymorphismus sowohl für ganze, als auch reelle Zahlen, Strings oder Kundenrecords usw. verwendet werden. Beispiele dafür sind *generic packages* in Ada bzw. *templates* in C++.

2.2.5 Dynamisches Binden

In den vorangegangenen Abschnitten wurde *dynamisches Binden*, das eng mit Polymorphismus und dem Konzept der Vererbung verbunden ist, bereits implizit verwendet. Um Polymorphismus überhaupt zu ermöglichen – zumindest in jenen Variationen, bei denen zur Übersetzungszeit vom Compiler noch nicht alle Referenzen aufgelöst werden können – wird dieses Konzept benötigt und soll hier näher beschrieben werden.

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 des Programms hergestellt wird (vgl. [Boo93]).

Wie in Abschnitt 2.2.4 beschrieben, erlauben Programmiersprachen, die Polymorphismus unterstützen, die Verwendung von Konstrukten, bei denen der Compiler zur Übersetzungszeit zwar unter Umständen noch Typkonsistenzprüfungen durchführen kann, aber nicht mehr in der Lage ist festzustellen, welcher Klasse ein Objekt (zur Laufzeit) angehört bzw. welche Version (Ausprägung) einer Methode aufgerufen werden soll (Bsp. 1, Bsp. 2).

coercion Polymorphismus hängt, wie in [CW85] gezeigt, stark von der Sichtweise bzw. der konkreten Implementierung ab.

In Bsp. 1 aus Abschnitt 2.2.4 kann der Compiler keine Aussagen über die Klassen der im Array zur Laufzeit enthaltenen Figuren treffen und kann somit auch nicht den tatsächlich auszuführenden Programmcode für die verwendete Methode `draw` einsetzen, da jede Klasse (`Square`, `Circle`, etc.) eine eigene Methode mit diesem Namen besitzt, die nur für ein Objekt dieser speziellen Klasse anwendbar ist. Die Auswahl der Methode hängt also vom dynamischen Typ eines Objektes ab und das Laufzeitsystem muß abhängig von diesem darüber entscheiden, welche konkrete Version (Ausprägung) der Methode in der aktuellen (dynamisch unterschiedlichen) Situation zu verwenden ist. Ähnliches gilt auch für Bsp. 2 in Abschnitt 2.2.4.

Im Unterschied zu Sprachen mit *statischem (frühem) Binden* (z.B. Ada), bei denen sämtliche Namen-Typ-Bindungen bei der Programmübersetzung festlegbar und somit überprüfbar sind, muß in Sprachen mit dynamischem Binden ein Teil dieser Aufgabe vom Laufzeitsystem übernommen werden. Dies bedeutet zwar leichte Einbußen in bezug auf die Laufzeiteffizienz von Programmen, die Einfachheit von Programmkonstrukten, die Übersichtlichkeit, leichte Erweiterbarkeit und Wartbarkeit kompensiert diesen Nachteil jedoch bei weitem.[14]

3 Programmiersprachliche Realisierungen

3.1 Smalltalk

Smalltalk stellt eine der frühesten[15], jedoch bereits alle wichtigen Konzepte des objekt-orientierten Paradigmas beinhaltende Programmiersprache dar.

Die langsame Verbreitung von Smalltalk ist nicht zuletzt auf dessen komplexe, ressourcen-intensive Entwicklungsumgebung und die daher lange Zeit nicht verfügbaren (bzw. teuren) Hardware-Voraussetzungen zurückzuführen.

In der Zwischenzeit erfreut sich Smalltalk gerade durch diese mächtige Entwicklungsumgebung, seine puristisch objekt-orientierten Konzepte und seiner besonderen Eignung für das Prototyping steigender Beliebtheit in der professionellen Programmentwicklung.

3.1.1 Klassen und Objekte

Smalltalk [GR83], [PW88], [Gol84] stellt eine puristische, vollständig objekt-orientierte Programmiersprache dar. Demzufolge gibt es in Smalltalk ausschließlich Objekte. Es wer-

[14] Die Effizienzunterschiede werden, wie jüngste Implementierungen beweisen, immer geringer. Etliche Sprachen, die dynamisches Binden unterstützen, bieten dem Programmierer außerdem zusätzlich die Möglichkeit, über die zu verwendende Art des Bindens zu entscheiden, wodurch eine weitere Verbesserung erreicht wird, da dynamisches Binden nur mehr dort verwendet wird, wo es der Programmierer für notwendig bzw. sinnvoll erachtet.

[15] abgesehen von Simula-67

den veränderliche (i.e. Objekte im herkömmlichen Sinne) und unveränderliche Objekte (i.e. Zahlen, Zeichen, Zeichenketten, Symbole und Felder) unterschieden.

Aufgrund seiner puristischen objekt-orientierten Konzeption (“alles sind Objekte”) werden in Smalltalk selbst Klassen als Objekte bezeichnet. Objekte selbst sind Instanzen einer Klasse; Klassen definieren das Verhalten von Objekten durch deren Spezifikation. Das sogenannte *class description protocol* definiert alle Fähigkeiten, Eigenschaften und insbesondere Methoden jedes Objekts, das Instanz derselben Klasse ist.

Alle Arten von erzeugbaren Objekten einer Klasse müssen im *class description protocol* definiert werden. Neue Objekte einer Klasse (Objekterzeugung) werden durch entsprechende Nachrichten an die Klasse selbst erzeugt (Klassenmethoden, z.B. `new`).

Eine Klassenbeschreibung besteht aus dem Klassennamen, der Deklaration von Klassen- und Instanzvariablen, sowie den Klassen- und Instanzmethoden. Diese Beschreibung wird im *class description protocol* einer jeden Klasse definiert. Jede neue Klasse muß in die Klassenhierarchie des Smalltalk-Systems (definiert die Vererbungsstruktur, vgl. Abschnitt 3.1.3) integriert werden (vgl. Abb. 7).

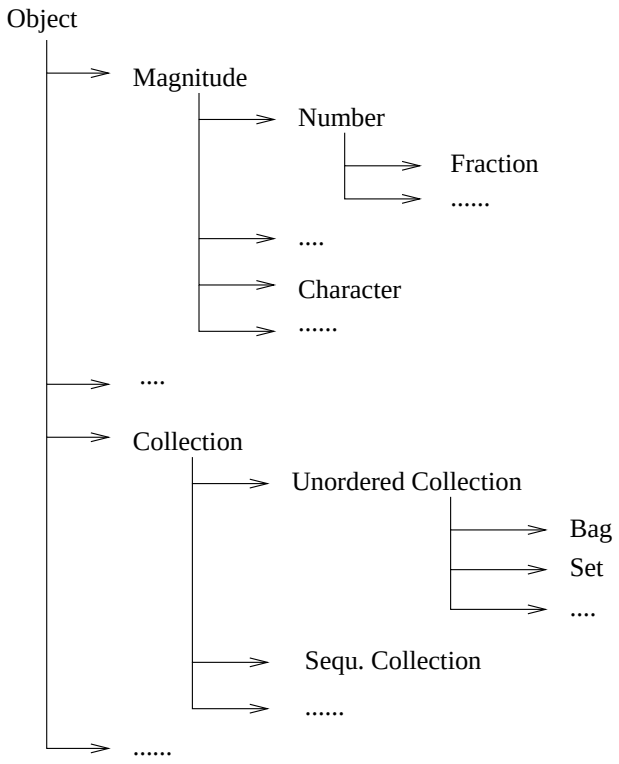

Abbildung 7: Klassenhierarchie von Smalltalk

Bei der Definition einer Klasse wird festgelegt, wo diese in der Klassenhierarchie ein-

gereiht wird und wie die Variablen für *private, shared* und *pool data* bezeichnet werden:

- Jedes Objekt besitzt für sich *private data*, die Instanzvariablen (Attribute) des Objekts. Solche privaten Daten können nur durch Instanzmethoden verwendet werden und sind lediglich für die jeweilige Instanz der Klasse selbst zugänglich.
- *shared data*: Diese Daten (= Klassenvariablen) sind allen Objekten dieser 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 Methoden definiert:

- Instanzmethoden implementieren die möglichen Aktionen von Objekten auf bestimmte Nachrichten.
- Klassenmethoden implementieren jene Aktionen, auf die die Klasse antworten kann (z.B. Erzeugen von Objekten).

Smalltalk-Objekte sind Kapselungen von Abstraktionen (Datenabstraktion und funktionelle Abstraktion, vgl. Abschnitt 3.1.4). Alle Daten- und Funktionsabstraktionen eines Objekts werden im *class description protocol* definiert.

3.1.2 Methoden

Wie im vorigen Abschnitt bereits ausgeführt, werden bei der Klassendefinition im *class description protocol* auch jene Methoden definiert, auf die ein Objekt in Smalltalk reagieren kann.

Ein Methodenaufruf ist in Smalltalk aus verschiedenen Blöcken aufgebaut: dem Empfänger (*receiver*) der Nachricht, einem Identifikationsmerkmal (*selector*) und den möglichen Argumenten. Je nach Anzahl der Argumente werden in Smalltalk unäre, binäre und Methoden mit Schlüsselwörtern unterschieden:

```
| unaer |
9 sqrt
| binaer |
3 + 5
| Methode mit Schluesselwoertern |
'das ist ein String' at: 12 put: $T
```

Der Empfänger einer Nachricht liefert (soferne er die aufgerufene Methode anbietet) als Ergebnis wiederum ein Objekt zurück.[16] Diese Tatsache muß auch bei zusammengesetzten Nachrichten berücksichtigt werden; in diesem Fall werden unäre vor binären, und diese wiederum vor Methoden mit Schlüsselwörtern ausgewertet. Im Zweifelsfall bedient man sich der Klammerung von Ausdrücken, um die gewünschte Reihenfolge der Auswertung zu gewährleisten.

3.1.3 Vererbung

Smalltalk in seiner ursprünglichen Form unterstützt nur Einfachvererbung, aber es gibt (wenig verbreitete) Implementierungen von Smalltalk, die auch Mehrfachvererbung unterstützen.

Allgemein gelten in Smalltalk folgende Regeln für die Vererbung:

1. Subklassen erben nur von im Vererbungsbaum hierarchisch über ihnen liegenden Superklassen, welche durch das *class description protocol* definiert 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* (vgl. Abb. 7).
2. Jede Klasse, ausgenommen *object* selbst, ist Subklasse der Klasse *object*.
3. Es gibt keine Vererbung von Subklassen an Superklassen.
4. Eine Subklasse erbt sowohl *private data*, *shared data*, *instance methods* und *class methods* von ihrer Superklasse (bzw. von allen ihren Superklassen).
5. Die Regeln für den Zugriff auf *private data* und *shared data* entsprechen jenen, die im *class description protocol* der Superklasse definiert sind.
6. Subklassen können zusätzlich zu den ererbten Daten und Methoden neue hinzufügen.
7. Subklassen können ererbte Instanz- und Klassenmethoden redefinieren (*method overriding*, eine spezielle Form von Polymorphismus).

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 Vererbungshierarchie fort, bis entweder eine entsprechende Methode gefunden oder selbst in der Klasse *object* keine gefunden wird. Ist letzteres der Fall, so antwortet das Smalltalk-System mit einer Fehlermeldung (*message not understood*), da es diese Nachricht nicht bearbeiten kann.

[16] Man beachte, daß Smalltalk das strenge Objekt-Konzept auch für Ergebnisse von Methodenaufrufen durchhält.

3.1.4 Kapselung

Die Datenkapselung wird in Smalltalk bei der Definition einer Klasse durch das *class description protocol* realisiert (vgl. Abschnitt 3.1.1). Dabei werden sowohl Klassenvariable als auch Instanzvariable definiert. Der Zugriff auf Instanzvariablen ist nur dem Objekt (= Instanz) selbst erlaubt. Objekte derselben Klasse können hingegen auf die ihnen gemeinsamen Klassenvariablen oder aber auch auf Pool Variablen zugreifen.

Solche Pool Variablen werden von Objekten unterschiedlicher Klassen geteilt, was einer Aufweichung des strengen Datenkapselungskonzepts entspricht; da der Zugriff auf solche Variablen aber speziell im *class description protocol* 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.

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

Die Datenkapselung wird demzufolge streng vorgegeben, sodaß der Zugriff auf private Daten eines Objekts ausschließlich durch das Objekt selbst erfolgen kann.

3.1.5 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. Die Konvertierung zwischen Klassen erfolgt aber dem Typkonzept (bzw. der Klassenhierarchie) entsprechend, sodaß letztendlich in Smalltalk eine Klasse einem Typ entspricht (Klasse = Typ).

Um die Typ- (d.h. eigentlich die Klassen-) Konsistenz zu gewährleisten, implementiert Smalltalk ad-hoc Polymorphismus: Werden in einem Ausdruck Zahlen unterschiedlichen Typs (Klassen) identifiziert, so konvertiert das Smalltalk-System diese in den allgemeineren der identifizierten Typen (Klassen) und führt damit anschließend die spezifizierte Operation durch (*coercion*, vgl. Abschnitt 2.2.4).

Beispielsweise gilt für die drei Klassen `float, integer` und `fraction`, daß `float` die allgemeinste Klasse dieser ist, und die beiden anderen Klassen lediglich Spezialfälle von `float` sind: Alle rationalen Zahlen können durch Floating-Point Zahlen repräsentiert werden. `integer`-Zahlen sind ein Spezialfall von `fraction`. Ein Ausdruck, in der diese Klassen auftreten, wird generalisiert, sodaß alle Klassen schließlich auf `float` konvertiert und anschließend der Ausdruck ausgewertet wird.

3.1.6 Polymorphismus und dynamisches Binden

Neben dem bereits im vorigen Abschnitt erwähnten Arten des Polymorphismus kann in Smalltalk dieselbe Nachricht an verschiedene Objekte gesendet werden, wobei die Bedeutung der Nachricht durch das *class description protocol* der jeweiligen Klasse eines Objekts festgelegt wird.

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.

Da in Smalltalk grundsätzlich alle Objekte potentiell polymorph sind, werden Methoden zur Laufzeit, abhängig von der jeweilgen Ausprägung eines Objekts, dynamisch ausgewählt.

3.1.7 Klassenbibliothek und Programmierumgebung

Die in Smalltalk-Systemen üblicherweise vorhandene, umfangreiche Klassenbibliothek bietet dem Entwickler die Möglichkeit, bereits implementierte Klassen und Methoden wiederzuverwenden, und sich somit auf die Komposition eines neuen Programmes aus zum großen Teil existierenden Software-Komponenten zu konzentrieren. Dies stellt ein äußerst mächtiges Programmierwerkzeug dar.

Die meisten Smalltalk-Systeme stellen für die Programmentwicklung verschiedene Tools zur Verfügung: Für die Navigation in der Klassenbibliothek gibt es in vielen Systemen einen *Class Hierarchy Browser*, um das Auffinden von Klassen und deren Methoden sowie die Erweiterung um neue, vom Benutzer definierte Klassen, zu unterstützen. Zumeist finden sich in solchen Klassenbibliotheken Graphikklassen, die eine einfache und rasche Erstellung von Benutzerschnittstellen für die jeweilige Systemplattform (z.B. IBM Presentation Manager, Microsoft Windows, X-Windows, etc.) erlauben.

Da Smalltalk-Systeme interpretativ arbeiten, wird darüberhinaus die Software-Entwick–lung mittels Prototyping und evolutionärer Programmierung unterstützt, da der Entwickler seine Erweiterungen und Neuerungen sofort austesten und gegebenenfalls frühzeitig korrigierend eingreifen kann.

3.2 C++

C++ wurde beginnend in den frühen achtziger Jahren von Bjarne Stroustrup ([Str94]) bei AT&T ursprünglich als Sprache für ereignisgesteuerte Simulationen entwickelt. Eigentlich war Simula 67 als Implementierungssprache vorgesehen gewesen. Dies wurde allerdings aus Effizienzgründen wieder verworfen und stattdessen, aufbauend auf die Sprache

C, eine neue, effiziente objekt-orientierte Programmiersprache – C++ – entwickelt. Durch einen wechselseitigen Austausch zwischen Sprachdesignern und Benutzern wurde C++, über seinen eigentlichen Verwendungszweck im Rahmen des erwähnten Projektes hinaus, im Laufe der Jahre zu einer allgemein verwendbaren objekt-orientierten Sprache weiterentwickelt, die sich nun soweit konsolidiert hat, um von ANSI und in Folge sicher auch von ISO genormt zu werden. In weiterer Folge sind massive Bestrebungen im Gange C als *die* Systemsprache von UNIX durch C++ abzulösen.

C++ ist eine *hybride* Sprache. Sie vereinigt in sich die Ausdruckskraft einer objekt-orientierten Sprache mit den Möglichkeiten einer traditionellen, effizienten, strukturierten Programmiersprache. Damit wird dem Entwickler sowohl die Möglichkeit geboten, maschinennahe, effizient und einfach an den Rechner anpaßbar zu arbeiten als auch problemnahe auf hohem Abstraktionsniveau, indem die Möglichkeit für eine direkte und prägnante Formulierung der Lösung geboten wird. Diese Zwitterrolle wurde aus folgenden Gründen bewußt in Kauf genommen:

- C++ wurde als Sprache zur Systemprogrammierung im weitesten Sinne entwickelt ([Str94]). Da gerade in diesem Bereich Effizienz eine herausragende Rolle spielt, wird die Möglichkeit geboten, *prozedural* und systemnah zu programmieren.
- Es existieren riesige Mengen Code in C, die nicht ohne Probleme wiederverwendet werden könnten. Aus diesem Grund ist C++ bis auf minimale Ausnahmen vollständig mit ANSI C kompatibel.
- C-Programmierer müssen nicht eine vollständig neue Sprache bzw. ein unbekanntes Paradigma von Grund auf erlernen, sondern können sich sukzessive die neu zur Verfügung gestellten Möglichkeiten aneignen, wodurch der Umschulungsaufwand stark reduziert wird.

Da, wie bereits erwähnt, jedes ANSI C Programm auch ein – wenngleich nur die prozeduralen Konzepte ausnutzendes – C++-Programm ist, stellt C++ eine Übermenge von C dar. Daraus erklärt sich auch der Name C++: `'++'` ist in C der Increment-Operator, C++ also ein “inkrementiertes”, verbessertes C. Sehr oft wird C++ daher auch als “C mit Klassen” bzw. “besseres C” bezeichnet. Diese umgangssprachliche Klassifizierung unterschätzt allerdings die Fähigkeiten von C++, da es sich tatsäschlich um eine mächtige, moderne objekt-orientierte Programmiersprache handelt.

3.2.1 Klassen und Objekte

Die grundlegende Philosophie von C++ ist es, nur eine kleine Menge an Basisdatentypen (Basisklassen) mit dazugehörigen (einfachen) Operationen zur Verfügung zu stellen. C++ enthält darüber hinaus keine höheren Datentypen und keine höheren primitiven Operationen (z.B. Zeichenketten mit Verkettungsoperator). Solche Datentypen und Operationen müssen vom Benutzer definiert werden, können dann allerdings (bei entsprechender

Deklaration und Implementierung) *gleich* wie die in der Sprache vorhandenen einfachen Klassen und Operationen verwendet werden. Dies erlaubt also eine benutzer- bzw. anwendungsspezifische "Spracherweiterung". Die grundlegende Aktion beim Programmieren in C++ ist somit die "Definition eines neuen allgemein verwendbaren oder anwendungsspezifischen Typs" [Str94].

Eine Klasse in C++ ist ein benutzerdefinierter Typ. Die objektorientierten Konzepte der Datenkapselung und Vererbung werden durch das Modell der Klasse realisiert (vgl. Abschnitte 3.2.3 und 3.2.4). Alle Objekte sind Exemplare (Instanzen) einer bestimmten Klasse. Die Deklaration einer Klasse bestimmt das Verhalten aller Objekte dieser Klasse. Klassen selbst sind in C++, im Gegensatz zu Smalltalk, allerdings *keine* Objekte.

Eine Klasse in C++ wird durch folgende Komponenten definiert:

- Eine Menge von Datenelementen (*data members*), die die Klasse und in weiterer Folge den Zustand der Instanzen (Objekte) der Klasse repräsentieren.
- Eine Menge von Methoden (*member functions*), wodurch die Menge der Operationen festgelegt wird, die auf die Klasse angewendet werden können.
- Verschiedene Ebenen des Zugriffs auf die Klasse bzw. ihre Repräsentation können angegeben werden. Datenelemente bzw. Methoden der Klasse (*members*) können entweder als privat (`private`), geschützt (`protected`) oder öffentlich (`public`) definiert werden. Diese Zugriffsebenen bestimmen den Zugriff auf *members* der Klasse innerhalb eines Programmes. In einer Klasse können alle diese Zugriffsebenen gemischt verwendet werden. Unterschiedlichen Teilen der Klasse werden somit unterschiedliche Zugriffsrechte zugeordnet:
  - `private`
    Daten und Methoden sind nur innerhalb der Klasse zugänglich.
  - `protected`
    Nur innerhalb der Klasse selbst und aus Subklassen dieser Klasse (*derived classes*) kann auf die in diesem Teil definierten *members* zugegriffen werden.
  - `public`
    Dieser Teil definiert die von außen zugängliche Schnittstelle der Klasse.

Abb. 3 zeigt ein Beispiel einer einfachen Klassendefinition in C++.

Jede Klasse definiert Konstruktoren und Destruktoren, die zur Erzeugung (Instanzierung) von Exemplaren einer Klasse (entspricht `new` in Smalltalk) und ihrer Zerstörung (Deinstanzierung) dienen. Die Namen der Konstruktoren/Destruktoren sind gleich dem Namen der jeweiligen Klasse (z.B. `Square` in Abb. 3). Die Instanzierung bzw. Deinstanzierung erfolgt durch impliziten (durch das Laufzeitsystem) oder expliziten Aufruf (durch den Programmierer) der Konstruktoren bzw. Destruktoren. Eine implizite Instanzierung

findet statt, wenn der Sichtbarkeitsbereich (*scope*) der Deklaration einer Variablen betreten wird. Im umgekehrten Fall erfolgt analog die Deinstanzierung. Der explizite Aufruf der Konstruktoren bzw. Destruktoren wird benötigt, wenn Zeiger auf Objekte verwendet werden. Dabei muß das Schlüsselwort `new` dem Konstruktor bzw. `delete` der Zeigervariable, die auf ein Objekt zeigt, vorangestellt werden.

Klassenvariable sind in C++ als statische Variable (`static`) zu deklarieren, werden dadurch nicht bei jeder Instanzierung neu angelegt, sondern unter allen Instanzen einer Klasse geteilt und existieren bereits vor der ersten Instanzierung. Sie können von allen Objekten dieser Klasse verwendet werden (klassen-globale Variable).

3.2.2 Methoden

In der Klassendefinition werden auch die Methoden festgelegt, über die eine Instanz einer Klasse (i.e. Objekt) manipuliert werden kann. Je nachdem auf welchem Zugriffsniveau eine Methode deklariert wurde, kann sie nur von bestimmten Benutzern, wie im vorigen Abschnitt beschrieben, verwendet werden. Die generelle Schnittstelle nach außen bilden die Methoden des öffentlichen Teils der Klassendefinition (*public interface*).

Wie im objekt-orientierten Paradigma üblich, unterscheidet auch C++ zwei Arten von Methoden:

- Objektmethoden, die die Aktionen implementieren, um ein Objekt zu manipulieren.
- Klassenmethoden für Aufgaben, die von der Klasse selbst und nicht von einem bestimmten Objekt durchzuführen sind. Dazu zählen:
  - Konstruktoren und Destruktoren (siehe vorangegangenen Abschnitt).
  - Überladene Operatoren (*overloaded operator functions*), die in Operator-Notation auf Objekte angewendet werden können (z.B. Stringkonkatenation in der Form `s1 + s2`).
  - Konversionsoperatoren (system- oder benutzerdefiniert), die eine Menge erlaubter Typkonversionen für eine Klasse definieren.
  - *Static member functions*, die Methoden definieren, die *unabhängig* von der Instanzierung von Objekten der Klasse existieren (z.B. Methoden zum Zugriff auf Klassenvariable).

Zur Steigerung der Effizienz können Methoden in C++ als `inline`-Code definiert werden. Eine derart deklarierte Methode wird vom Compiler nicht als Funktionsaufruf umgesetzt, sondern der Code der Methode wird an jeder Stelle ihrer Verwendung in den Programmcode eingefügt. Dies kann als spezielle Ausprägung von Makroexpansion interpretiert werden.[17]

[17] Die Bewertung der Sinnhaftigkeit solcher Vorgangsweisen im Sinne objekt-orientierter Programmierung bleibt dem Leser überlassen.

Die Funktionalität von Methoden muß in C++ nicht innerhalb der Klassendefinition implementiert werden. Es genügt vielmehr die Methode durch Angabe ihrer Signatur (entspricht einem Funktionsprototypen) in der Klassendefinition zu deklarieren und an anderer Stelle im Quellcode die tatsächliche Implementierung nachzuliefern. Als Beispiel dazu dient die Definition der Methode `area` aus Abb. 3:

```
float Square::area(void)
{
    return abs((P1.x - P2.x) * (P1.y - P2.y));
}
```

3.2.3 Vererbung

Als objektorientierte Sprache unterstützt C++ natürlich auch das Konzept der Vererbung (in C++ *derivation* genannt). Ein Subklasse wird in C++ als *derived class* bezeichnet, eine Superklasse als *base class*. Subklassen erben alle *data members* und *member functions* aller in der Vererbungshierarchie über ihnen liegenden Superklassen. Ererbte Methoden können dabei von der erbenden Klasse überladen d.h. redefiniert werden. Eine Vererbung von Subklassen an Superklassen ist nicht möglich.

Im Unterschied zu Sprachen wie z.B. Smalltalk existiert keine gemeinsame "Wurzel" der Vererbungshierarchie, welche die Superklasse aller Objekte und Klassen darstellt. Vielmehr können in C++ beliebig viele, parallele Vererbungshierarchien mit verschiedenen Wurzelklassen definiert werden.

C++ bietet sowohl die Möglichkeit von *single inheritance* als auch von *multiple inheritance*, i.e. Klassen können entweder von genau einer Superklasse oder von mehreren Superklassen erben. Bei *multiple inheritance* existiert zusätzlich noch das Konzept der *virtual base classes*, um Mehrdeutigkeitsprobleme (*name clashes*) bei von mehreren Seiten geerbten gleichen Klassen zu umgehen.

In der in Abb. 8 gezeigten Vererbungshierarchie erbt die Klasse `CompFigur` beispielsweise indirekt zweimal von `Figur`. Damit in `CompFigur` eine Methode oder Instanzvariable aus `Figur` angesprochen werden kann, muß, um die Eindeutigkeit von Namen zu gewährleisten, der genaue Weg im Vererbungsgraphen (mit Hilfe des *scoping operators* `'::'`) bis zur gewünschten Methode bzw. Instanzvariable angegeben werden. Um solche Mehrdeutigkeitsprobleme zu umgehen und zusätzlich noch Speicherplatz zu sparen (für jede Klasse in der Vererbungshierarchie wird Speicherplatz angelegt), wird in C++ das oben erwähnte Konzept der *virtual base classes* (*sharing* von *base classes*, siehe [Lip92]) verwendet. Der rechte Teil von Abb. 8 zeigt die gleiche Vererbungshierarchie bei Verwendung von *virtual base classes*.

Des öfteren kann es notwendig sein, daß Klassen von einer gemeinsamen Superklasse erben, wobei die Superklasse nur diesem Zweck dient und nicht beabsichtigt ist, Instanzen dieser Superklasse zu erzeugen (z.B. bei Verwendung von Subklassen in einem Superklassenkontext unter Verwendung von Polymorphismus). C++ bietet eine Lösung für dieses

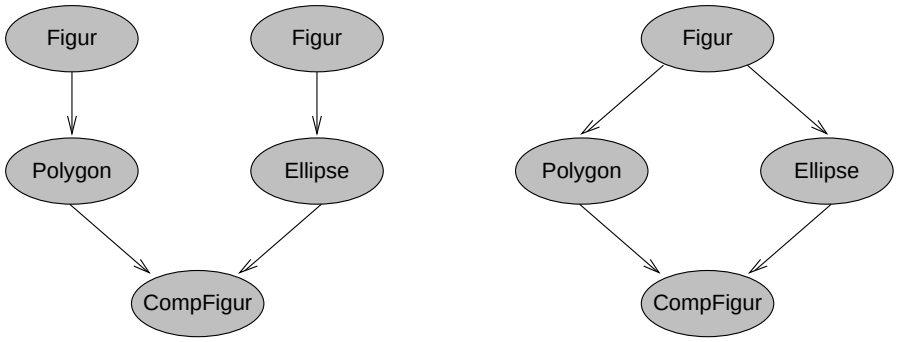

Abbildung 8: *Base class* vs. *virtual base class*

Problem durch das Konzept der *abstract base classes*. `Figur` aus Abb. 5 beispielweise wird sinnvollerweise als *abstract base class* implementiert werden.

Auch im Zusammenhang mit Vererbung besteht in C++ die Möglichkeit, spezielle Zugriffsniveaus auf die Daten und Methoden einer Superklasse seitens der erbenden Klasse zu definieren. Diese ergeben sich ja bereits aus der Definition der Zugriffrechte in der Superklasse. Wie in einer einfachen Klassendefinition können Subklassen bei den Klassen, von denen sie erben, auch die Art des Zugriffsniveaus auf die Superklasse – unter Berücksichtigung der in den Superklassen selbst definierten Zugriffsrechte – mit den Schlüsselwörtern `private`, `protected` und `public` angeben. Dabei gelten folgende Regeln:

- Auf den privaten Teil einer vererbenden Klasse (Superklasse), kann aus einer Subklasse *nicht* zugegriffen werden. Subklassen haben grundsätzlich nur Zugriff auf den öffentlichen und geschützten Teil einer Superklasse[18].
- Eine erbende Klasse hat Zugriff auf alle *members* (Attribute und Methoden) in den öffentlichen und geschützten Teilen aller in der Vererbungshierarchie über ihr stehenden Klassen. Dabei gilt, daß *public* bzw. *protected members* aus Superklassen so verwendet werden können, als wären sie im entsprechenden Teil der erbenden Klasse definiert. Für diese Art des Zugriffs muß das Schlüsselwort `public` definiert werden.
- Wird bei der Angabe der Klassen von denen zu erben ist, das Schlüsselwort `protected` verwendet, so werden die öffentlichen und geschützten Teile der vererbenden Klasse als *protected members* der erbenden Klasse betrachtet.
- Für `private` gilt analoges zu `protected`.

[18] Ausnahmen dazu sind allerdings möglich. Siehe dazu Abschnitt 3.2.4.

3.2.4 Kapselung

Kapselung wird in C++ implizit durch den Mechanismus der Klasse erreicht. Nur ein Objekt selbst darf auf die in der Klassendefinition im privaten Teil angegebenen *members* zugreifen. Auf den geschützten Teil einer Klasse wiederum haben nur Subklassen dieser Klasse Zugriff. Nur der als öffentlich deklarierte Teil der Klasse ist von außen zugänglich. Eine Klasse in C++ definiert also eine Datenkapsel mit abgestuften Zugriffsmöglichkeiten. Dieses Konzept der Kapselung wird – wie im vorangegangenen Abschnitt beschrieben – im Zuge der Vererbung beibehalten und um zusätzliche Möglichkeiten erweitert.

Abgesehen von dieser strikten Kapselung existieren zwei Formen von globalen Variablen. Einerseits sind dies globale Variablen, wie sie von den meisten prozeduralen Programiersprachen her bekannt sind, und in C++ schon allein aus Kompatibilitätsgründen zu ANSI C, das diesen Mechanismus beinhaltet, vorhanden sein müssen. Andererseits verfügt C++ über Klassenvariable bzw. Klassenmethoden (*static members*), die innerhalb einer Klasse global sind und auf die alle Instanzen einer Klasse Zugriff haben.

Neben dieser strikten Form der Kapselung beinhaltet C++ allerdings auch Mechanismen, die dieses strenge Konzept aufweichen und hauptsächlich aus Gründen der Effizienz und einfachen Verwendbarkeit Eingang in die Sprachdefinition gefunden haben. Das in C++ vorhandene Konzept der *friends* durchbricht das Konzept der Datenkapselung. Wird eine ganze Klasse oder eine einzelne Methode als *friend* einer Klasse deklariert, so hat diese 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 ein Klasse umgangen, was zu einer Erhöhung der Laufzeiteffizienz führt. Wie dem programmiererfahrenen Leser allerdings auch sofort klar wird, bietet dieser Mechnismus breite Möglichkeiten des Mißbrauchs und sollte daher nur mit Vorsicht verwendet werden.

Doch in C++ existieren noch weitere Probleme in bezug auf Datenkapselung. Die Schilderung derselben würde allerdings genaue Kenntnisse der Programmierung in C++ voraussetzen und den Rahmen dieses Artikels sprengen. Der interessierte Leser sei auf [Liu91] verwiesen, worin eines dieser Probleme ausführlich geschildert wird.

3.2.5 Typkonzept

Vom Typkonzept her ist C++ ein typischer Vertreter der *strongly typed* Sprachen: Die statische Analyse eines Programmes durch den Compiler garantiert die Typkonsistenz des Programmes, auch wenn zu diesem Zeitpunkt noch nicht alle dynamischen Typen eindeutig feststellbar sind, wie dies in C++ z.B. bei der Verwendung von *virtual member functions* der Fall ist. Ein derart überprüftes Programm kann dann mit Sicherheit ohne Typfehler ablaufen.

Die Typüberprüfung läuft in C++ so ab, daß der Compiler allen Ausdrücken Typen zuweist und ihre Konsistenz überprüft. Wird dabei keine Übereinstimmung erreicht, so

können noch implizite (vom Compiler) oder explizite (vom Programmierer) Typkonversionen angewandt werden. Der Programmierer hat die Möglichkeit über sogenannte *casts* eine explizite Typkonversion vorzuschreiben. Dieser Mechanismus ist bereits in C bzw. ANSI C vorhanden. Im Unterschied zu C bzw. ANSI C, die nur eine bestimmte Menge von Konversionsfunktionen, vor allem für Zahlen-, Zeichen- und Zeigertypen, verfügen, bietet C++ die darüber hinausgehende Möglichkeit, *eigene* Konversionsfunktionen zu definieren, und zwar für beliebige Typen. Diese benutzerdefinierten Konversionen werden vom Compiler *gleich* wie die vordefinierten behandelt und können sowohl implizit als auch explizit zur Anwendung kommen (vgl. [Lip92] und [Str94]).

3.2.6 Polymorphismus und dynamisches Binden

C++ bietet alle der in Abschnitt 2.2.4 beschriebenen Formen von Polymorphismus und bietet dem Entwickler in dieser Hinsicht eine außerordentlich hohe Flexibilität bzw. Bandbreite an Mechanismen, derer er sich in unterschiedlichen Anwendungsformen bedienen kann.

Overloading ist in C++ sowohl für Methoden bzw. Funktionen als auch Operatoren auf einfache Weise möglich. Um Methoden bzw. Funktionen zu überladen, genügt es in C++ eine Funktionen mit gleichem Namen und unterschiedlichen Signaturen zu definieren. Bis auf wenige Ausnahmen können in C++ auch alle Operatoren überladen werden. Dazu ein einfaches Beispiel: Der Ausgabe-Operator `'<<'` soll überladen werden, um ihn auf die in Abb. 3 definierte Klasse `Square` anwenden zu können:

```
class Square
{
    friend ostream&
    operator << (ostream& OutputStream, Square& OutputSquare);
            ...
}

ostream& operator << (ostream& OutputStream, Square& OutputSquare)
{
    // Implementierung, wie Square ausgegeben werden soll
}
```

Danach kann `'<<'` auf Objekte der Klasse `Square` wie auf vordefinierte Typen von C++ angewendet werden:

```
            ...
Square Square1;
            ...
cout << "Ausgabe von Square1:" << endl << Square1 << endl
            ...
```

Auch die zweite Form von *ad hoc* Polymorphismus, *coercion* wird in C++ unterstützt. Neben vordefinierten Konversionen für Zahlen-, String- und Zeigertypen, bietet C++ auch

die Möglichkeit, wie bereits im vorangegangenen Abschnitt beschrieben, eigene Konversionsfunktionen zu definieren und so die Grenzen und Möglichkeiten dieser Form von Polymorphismus an das jeweilige zu bewältigende Problem anzupassen. Beschränkungen über die Art der Konversionen bestehen dabei nicht. Die hinzugefügten Konversionen werden den vordefinierten gleichgestellt und können sowohl implizit als auch explizit verwendet werden.

Wie jede objektorientierte Sprache unterstützt C++ natürlich *inclusion* Polymorphismus. Diese Form des Polymorphismus modelliert die Konzepte der Klassenhierarchie, wie sie natürlich auch in C++ existiert, und der Vererbung. In C++ kann jedes Objekt einer Subklasse auch im Kontext einer ihrer Superklassen (bei `public`-Vererbung) verwendet werden. Beipiele dafür wurden bereits in Bsp. 1 und Bsp. 2 aus Abschnitt 2.2.4 gegeben. In diesen Beipielen wird abhängig vom dynamischen Typ der Objekte die richtige Ausprägung der Methoden `draw` bzw. `area` aufgerufen.

Um dieserart polymorphe Methoden zu definieren, kennt C++ den Mechanismus der *virtual member functions*. Eine Klasse kann eine Methode durch Voranstellen des Schlüsselwortes `virtual` als virtuelle Methode definieren. Dabei kann auch eine Defaultimplementierung der Methode angegeben werden. Wird dies unterlassen und die Methode nur deklariert, so spricht man von *pure virtual member functions*. Eine von dieser Klasse erbende Klasse kann nun diese Funktion an ihre eigenen Notwendigkeiten anpassen und die gewünschte Funktion implementieren. Vernüftigerweise also würde die Klasse `Figur` aus Abb. 5 die Methoden `draw`, `area` und `rotate` als virtuelle Methoden deklarieren und die eigentliche Implementierungen dieser Methoden den jeweiligen von ihr erbenden Klassen (z.B. `Square`) überlassen.

Es sei an dieser Stelle betont, daß es für die Verwendung von Subklassen in einem Superklassenkontext notwendig ist, virtuelle Methoden zu verwenden. Würde dies nicht gemacht, so würde die Redefinition einer Methode in einer Subklasse zu *overloading* führen und bei Verwendung der Subklasse im Kontext einer Superklasse, die Superklassenmethode aufgerufen (anstelle der beabsichtigten Subklassenmethode).

Wie aus diesen Ausführungen ersichtlich ist, benötigt *inclusion* Polymorphismus als einzige Form von Polymorphismus in C++ das zusätzliche Konzept des dynamischen Bindens. Alle anderen Arten von Polymorphismus in C++ können im Unterschied dazu bereits zur Übersetzungszeit aufgelöst werden. Da bei virtuellen Funktionen aber erst der dynamische Typ über die Auswahl der zu verwendenden Ausprägung entscheidet, muß diese zur Laufzeit dynamisch gebunden werden. Die Implementierung von C++ erlaubt sehr effizientes dynamisches Binden, sodaß der Entwickler Effizienzüberlegungen größtenteils unberücksichtigt lassen kann.

Als letzte Form unterstützt C++ noch *parametric* Polymorphismus. Dies geschieht in C++ über den Mechanismus der *templates*. Auf diese Art können in C++ generische Klassen und Methoden bzw. Funktionen definiert werden. Mit diesem Konzept kann die Funktionalität und Repräsentation einer Klasse bzw. Funktion einmal festgelegt werden und derselbe Programmcode für unterschiedlichste Typen (wieder-) verwendet werden, ohne spezielle Anpassungen vornehmen zu müssen. Im Zusammenspiel mit Vererbung

stellt dies eines der mächtigsten Ausdrucksmittel von C++ dar. Weitergehende Betrachtungen über *templates* finden sich in [Lip92] und [Str94].

3.2.7 Klassenbibliothek und Programmierumgebung

C++ entstand wie sein Vorfahre C in einer UNIX-Umgebung und bietet somit als einfachste Form der Programmierumgebung, jene, welche standardmäßig unter UNIX zur Verfügung gestellt wird. Dies bedeutet, daß zwar z.T. sehr mächtige Werkzeuge (Steuerung des Übersetzungsablaufes, sprachsensitive Editoren, etc.) zur Verfügung stehen können, aber nicht müssen. Philosphie von UNIX ist es, ein allgemeines Grundgerüst zur Verfügung zu stellen, aber keine spezielle, für eine Sprache optimierte Entwicklungsumgebung.

Auch ist eine Programmierumgebung bei C++ im Unterschied zu Sprachen wie Smalltalk, Oberon-2, etc. nie integraler Bestandteil der Sprachentwicklung gewesen. Dies bedeutet, daß die einfachste Programmierumgebung für C++ nichts anderes als der Compiler und das Betriebssystem sind. Im Laufe der Zeit wurden jedoch von Herstellern und Universitäten Umgebungen für die Programmentwicklung unter C++ geschaffen, die dieses ursprünglich vorhandene Manko schon längst ausgemerzt haben. Moderne Programmierumgebungen für C++ stellen integrierte graphische Oberflächen zur Verfügung und sind mit einer Vielzahl von Entwicklungswerkzeugen, wie Debugger, Classbrowser, etc. ausgestattet, die eine effiziente Softwareentwicklung ermöglichen. Die Art und Vollständigkeit dieser Umgebungen differiert von Hersteller zu Hersteller.

Neben einer guten Programmierumgebung ist für die Entwicklung von Software mit C++ allerdings auch eine gute Klassenbibliothek notwendig. Um effizient programmieren zu können, sollte der Entwickler ja nicht unbedingt jede Klasse selbst implementieren müssen, sondern ihm sollte bereits ein reicher Fundus an allgemein verwendbaren Klassen zur Verfügung stehen. Auch in diesem Bereich existiert für C++ bereits ein reichhaltiges Angebot. Oft sind Klassenbibliotheken bereits integraler Bestandteil von Programmierumgebungen.

Neben diesen herstellerspezifischen Klassenbibliotheken existieren Bestrebungen, C++ selbst bereits mit einer gut ausgestatteten (genormten) Standardbibliothek auszurüsten. Dies geschieht sowohl von seiten der Hersteller also auch durch Universitäten und Free Software Organisationen.

3.3 Objective-C

Objective-C wurde von Brad J. Cox ([Cox86]) entwickelt und ist eine Erweiterung von ANSI C um objektorientierte Konzepte. Wie bei C++ handelt es sich bei Objective-C um eine Obermenge von ANSI C, sodaß in Objective-C auch konventionelle, imperative Programmierung möglich ist. Dies bedeutet, daß eine fließende Umstellung von ANSI C auf Objective-C für existierende Softwaremodule möglich ist und das Erlernen der Sprache für den Programmierer durch sukzessive Erweiterung seines “Sprachschatzes” möglich

ist. Die in Abschnitt 3.2 für C++ angeführten Vorteile einer solchen Obermengensprache gelten auch für Objective-C.

Wie C++ ist Objective-C eine *hybride* Sprache, mit sowohl imperativen als auch objekt-orientierten Sprachmitteln. Im Unterschied zu C++, welches in der Tradition von Simula 67 steht, bietet Objective-C einen Ansatz, der stark von Smalltalk herrührt. Objective-C übernimmt weite Teile des Objekt-, Klassen- und *message passing* Konzepts von Smalltalk (vgl. [Cox86], [Bud91]). Daraus ergeben sich interessante Unterschiede zwischen C++ und Objective-C, auf die auch im folgenden an geeigneter Stelle hingewiesen werden wird (vgl. auch [Hui90], [Sch94]).

3.3.1 Klassen und Objekte

Das auch in Objective-C zur Verfügung gestellte Konzept der Klasse erlaubt es dem Entwickler, die Repräsentation und das Verhalten von Objekten zu spezifizieren. Durch diese Definition werden auch die Zugriffsmöglichlichkeiten auf das Objekt bzw. seinen Zustand sowie die Manipulationsmöglichkeiten, die ein Objekt bietet, festgelegt (vgl. *class description protocol* in Abschnitt 3.1.1).

Eine Klasse in Objective-C besteht aus der Definition von Instanzvariablen, die den Zustand eines Objektes repräsentieren, und Methoden für den Zugriff auf diese Instanzdaten. Diese Definition zerfällt grundsätzlich in zwei Teile:

- `@interface`

  Deklaration einer neuen Klasse. Hier werden der Name der Klasse, ihre Position in der Vererbungshierarchie, Instanzvariablen und Methoden deklariert. In diesem Teil wird die öffentliche Schnittstelle der Klasse beschrieben. Die Definition wird mit dem Schlüsselwort `@end` abgeschlossen.

- `@implementation`

  In diesem Teil wird die Klasse näher spezifiziert. Dies bedeutet, daß hier die Implementierung der im Deklarationsteil definierten Methoden gegeben wird. Auch diese Definition wird mit dem Schlüsselwort `@end` abgeschlossen.

Folgendes Beispiel zeigt das bereits bekannte Polygon-Beispiel:

```
@interface Square : Polygon
{
    Vector        P1, P2;
    Color         TheColor;
    Border        TheBorder;
}

- (void)  draw;
- (float) area;
- (void)  rotate: center angle: (float) angle);

@end
```

Bei der Definition der Instanzvariablen erlaubt Objective-C zusätzlich die Angabe von Zugriffsniveaus. Ähnlich wie in C++ werden dadurch die möglichen Zugriffe auf Instanzvariablen beschränkt, indem Gruppen von Instanzvariablen unterschiedliche Zugriffrechte gegeben werden:

- `@public`

  Freier Zugriff auf Instanzvariablen ist möglich.

- `@private`

  Nur Methoden dieser Klasse und ihrer Subklassen haben Zugriff auf die Instanzvariablen[19].

- `@protected`

  Nur Methoden dieser Klasse haben Zugriff auf die Instanzvariablen.

Wenn kein Zugriffsniveau angegeben wurde wird implizit `@protected` angenommen. Im Unterschied zu C++ gibt es keine Möglichkeit, diesen Mechanismus auch auf Methoden anzuwenden. Methoden einer Klasse sind immer öffentlich.

Im Unterschied zu C++ besitzt Objective-C keine klassenspezifischen Konstruktoren bzw. Destruktoren. Es besteht allerdings die Möglichkeit, durch Redefinition der Methoden `new` (Instantiierung) bzw. `free` (Deinstantiierung) ein ähnliches Verhalten zu erreichen. Des weiteren existiert zur Instanzgenerierung die Möglichkeit der Definition von *factory objects* (vgl. [Cox86], [Bud91]).

In Objective-C existiert keine sprachseitige Unterstützung von Klassenvariablen (im Unterschied zu C++). Es besteht jedoch die Möglichkeit, dies durch Definition von `static` Variablen in Implementationsteilen zu umgehen, um eine ähnliche Funktionalität zu erreichen.

3.3.2 Methoden

Methoden bilden in Objective-C die Schnittstelle von Instanzen (i.e. Objekten) nach außen. Die Funktionalität von Methoden wird in Objective-C im `@implementation` Teil der Klassendefinition festgelegt. Es wird dabei zwischen Klassen- und Instanzmethoden explizit unterschieden. Klassenmethoden werden durch ein dem Namen der Methode vorangestelltes `'+'` Zeichen gekennzeichnet, Instanzmethoden hingegen durch ein vorausgehendes `'-'` Zeichen. Alle für eine Klasse definierten Methoden sind öffentlich zugänglich. Die Semantik von Klassen- bzw. Instanzmethoden entspricht der in objektorientierten Programmiersprachen üblichen.

[19] Vgl. dazu die Definition von Zugriffsniveaus in C++ in Abschnitt 3.2.1.

```
@implementation Square
- (float) area;
  {
      return abs(([P1 getx] - [P1 gety]) * ([P2 getx] - [P2 gety]));
  }
...
@end
```

Ein Methodenaufruf ist wie in Smalltalk aus mehreren Teilen aufgebaut: dem Empfänger (*receiver*) der Nachricht, einer Identifikation der Methode (*selector*) und möglichen Argumenten. Wie Smalltalk unterscheidet auch Objective-C je nach Anzahl der Argumente zwischen unären Methoden, binären Methoden und Methoden mit Schlüsselwörtern (*keyword message expressions*). Das eigentliche *message passing* ist Smalltalk entlehnt und kann nur innerhalb sogenannter *message expressions* auftreten. Dabei handelt es sich um in eckige Klammern eingeschlossene Ausdrücke (in Smalltalk Blöcke genannt). Folgendes Beispiel zeigt solche Methodenaufrufe:

```
...
id Square1 = [Square new];          /* unary method invocation   */
...
[Square1 rotate: P3 angle: 3.14];   /* keyword method invocation */
...
```

Neben diesen vom Benutzer definierten Methoden kennt jedes Objekt in Objective-C auch noch eine vordefinierte Auswahl an Nachrichten zur Feststellung von Klassen- bzw. Objektzugehörigkeit, Feststellung, ob ein Objekt eine bestimmte Nachricht versteht, etc[20]. Wird eine Nachricht an ein Objekt geschickt, die vom Objekt nicht “verstanden” wird, so kommt es zu einem Laufzeitfehler (mit entsprechender Fehlermeldung, ähnlich wie in Smalltalk). Des weiteren existieren Mechanismen, um zur Laufzeit Methoden zu Klassen hinzuzufügen bzw. zu entfernen[21].

3.3.3 Vererbung

Auch im Bereich der Vererbung ist Objective-C sehr eng an Smalltalk angelehnt. Wie Smalltalk bietet Objective-C Einfachvererbung (*single inheritance*). Dies bedeutet, daß jede Klasse genau eine Superklasse besitzt, von der sie erbt. Wie in Smalltalk existiert in Objective-C *genau* eine baumartige Vererbungshierarchie (im Unterschied zu C++) mit der speziellen Klasse `Object` als Wurzel. Jede Klasse in Objective-C außer `Object` selbst ist Subklasse dieser “Klasse aller Klassen”.

Eine Subklasse erbt in Objective-C von ihrer direkt übergeordneten Superklasse und den hierarchisch darüberliegenden Superklassen alle Instanzvariablen, die als öffentlich bzw. privat definiert wurden und alle Methoden. Instanzvariablen, die als geschützt deklariert wurden, sind Subklassenmethoden nicht zugänglich.

[20] Diese Möglichkeiten bietet C++ nicht, da die dafür notwendigen Informationen zur Laufzeit nicht mehr vorhanden sind.

[21] Dies ist in C++ nicht möglich.

Beim Aufruf einer Methode wird zuerst der *receiver* der Nachricht nach einem passenden *selector* durchsucht. Existiert keine geeignete Methode wird die Nachricht entlang der Vererbungshierarchie solange nach oben weitergereicht, bis die Methode gefunden wird. Scheitert dies bis inklusive `Objekt`, so reagiert das Laufzeitsystem mit der speziellen Nachricht `doesNotRecognize` und meldet einen Laufzeitfehler. Diese Vorgangsweise der Auflösung eines Methodenaufrufs entspricht jener von Smalltalk.

Subklassen können zu ererbten Instanzvariablen und Methoden neue hinzufügen, um das Verhalten der Klasse zu erweitern. Darüber hinaus besteht die Möglichkeit, ererbte Methoden zu redefinieren (*overriding*). Dies ist eine der möglichen Formen von Polymorphismus in Objective-C. Dabei wird auch der Mechanismus der *deferred methods* angeboten. Dies bedeutet, daß eine Methode von einer Klasse nur definiert wird und erst ihre Subklassen die Funktionaliät der Methode implementieren (*subclass responsibility*).[22]

Neben diesen von anderen objekt-orientierten Sprachen ebenfalls bekannten Formen der Vererbung bietet Objective-C noch das zusätzliche, in dieser Sprachfamilie relativ einzigartige Konzept des *posing*. Üblicherweise wird streng hierarchisch von Superklasse zu Subklasse vererbt. *Posing* bietet nun die Möglichkeit in entgegengesetzter Richtung zu "vererben"[23]: Eine Klasse kann die Position einer anderen Klasse einnehmen (ganz bzw. nur einzelne Methoden). Dies gilt im speziellen auch für Sub- und Superklassen, wodurch die hierarchische Vererbung durchbrochen wird. Es ist also möglich, daß eine Subklasse durch *posing* die Stelle einer Superklasse einnimmt. Dies bedeutet, eine Nachricht an die Superklasse geht zuerst an die Subklasse und *kann* von dieser nach entsprechender Bearbeitung an die Superklasse weitergleitet werden. Da es sich hier um ein ausgesprochen mächtiges Instrument handelt, durch welches etliche Restriktionen umgangen werden können, sollte es nur sparsam und vorsichtig eingesetzt werden!

3.3.4 Kapselung

Mit der Definition einer Klasse wird automatisch auch die Kapselung der Daten einer Klasse festgeschrieben. Durch die Deklaration der Methoden der Klasse wird die Schnittstelle der Klasse nach außen festgelegt. Auf die Instanzvariablen einer Klasse kann nur über die im `@interface` Teil angeführten Methoden zugegriffen werden. Ungewollte Veränderungen der Instanzvariablen sind somit ausgeschlossen. Dies ist ein dem *class description protocol* von Smalltalk sehr verwandter Mechanismus.

Da Objective-C keine Klassenvariablen unterstützt, existiert natürlich keine entsprechende Kapselungsmöglichkeit. Es besteht allerdings, wie bereits in Abschnitt 3.3.1 erwähnt, die Möglichkeit Klassenvariablen zu simulieren. Dazu wird die bereits in C vorhandene Möglichkeit ausgenutzt, daß Variablen, die als `static` definiert sind nur innerhalb der Datei sichtbar sind, in der sie deklariert wurden und auch ihren Wert auch beim Verlassen eines *scopes* behalten. Klassenvariablen werden in Objective-C durch globale `static`

[22] Man vgl. dazu *virtual member functions* in C++.

[23] Die Bezeichnung 'Vererbung' ist nicht ganz berechtigt. Ebenso gut könnte man *posing* auch als spezielle Form von Polymorphismus interpretieren.

Variablen in den Implementationsdateien von Klassen (`@implementation` Teile) simuliert. Auf diese Variablen kann von Subklassen und Verwendern der Klasse nicht zugegriffen werden. Da es sich dabei aber um ein nicht von der Sprache selbst explizit unterstütztes Konzept handelt, liegt die Verantwortung für richtige Verwaltung, Initialisierung, etc. vollständig beim Programmierer.

Wie bereits in Abschnitt 3.3.1 beschrieben, besteht noch die Möglichkeit, Zugriffsrechte auf Instanzvariablen zu definieren (`@public`, `@private`, `@protected`), wodurch eine weitere Ebene der Kapselung eingeführt werden kann, indem nicht nur der Zugriff auf die Klasse von außen abgeschirmt wird, sondern dies auch innerhalb einer Vererbungshierarchie ermöglicht wird (vgl. Abschnitt 3.3.3.

Im Unterschied zu Instanzvariablen sind Methoden einer Klasse immer öffentlich zugänglich. Objective-C bietet keine sprachlichen Mittel, um den Benutzerkreis von Methoden – sowohl Instanz- als auch Klassenmethoden – einzuschränken. Es besteht zwar die Möglichkeit mit bestimmten Programmiertechniken eine teilweise Kapselung zu simulieren, doch kann dies leicht vom Verwender eine Klasse umgangen werden und bietet so nur beschränkten Schutz.

Es sei an dieser Stelle auch darauf hingewiesen, daß in Objective-C – ähnlich wie in C++ – Möglichkeiten bestehen, das Konzept der Kapselung zu umgehen. Dies ergibt sich durch die vielfältigen und mächtigen Manipulationsmöglichkeiten, die Objective-C als Obermenge von ANSI C gezwungenermaßen enthält. Aus Gründen der Abwärtskompatibilität muß Objective-C mit diesem durchbrochenen Kapselungskonzept existieren.

3.3.5 Typkonzept

Die Klassifikation des Typkonzepts von Objective-C gestaltet sich auf Grund der hybriden Sprachnatur ziemlich schwierig. Durch die Kompatibiltät zu ANSI C (*statically typed*) und der konzeptuellen Verwandtschaft mit Smalltalk (*untyped* bzw. *dynamically typed*) ist das Typkonzept von Objective-C relativ inhomogen. Teile der Sprache sind eindeutig als getypt zu klassifizieren, andere als ungetypt[24].

Grundsätzlich übernimmt Objective-C das von ANSI C definierte statische Typkonzept. Variable können elementare Typen wie z.B. `int`, `float`, `char`, usw. besitzen. Ebenso können komplexere Typen (`typedef`) definiert und Variablen mit diesen benutzerdefinierten Typen versehen werden. Zusätzlich besteht natürlich wie in ANSI C die Möglichkeit Zeiger zu verwenden. Durch dieses statische Typkonzept wird dem Compiler die Möglichkeit einer strengen Typprüfung gegeben und Typinkonsistenzen können zur Übersetzungszeit festgestellt werden.

Parallel und gleichberechtigt neben diesem statischen Typkonzept besteht allerdings auch noch ein dynamisches bzw. ungetyptes. Objective-C fügt zu ANSI C den Typ des Objektes hinzu. Dieser neue Datentyp heißt `id` (*object identifier*). Variablen dieses Typs

[24] C++ ist im Gegensatz dazu eindeutig *strongly typed* (vgl. 3.2.5).

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` natürlich einer statischen Typprüfung entziehen. Sie können nur zur Laufzeit des Programmes klassifiziert werden (*dynamic typing* wie in Smalltalk).

3.3.6 Polymorphismus und dynamisches Binden

Im Gegensatz zu C++, das alle Arten von Polymorphismus unterstützt, besitzt Objective-C Polymorphismus in eingeschränkterer Form, indem gewisse Formen von Polymorphismus gar nicht und andere nicht durchgängig zur Verfügung gestellt werden. Eleganter als C++ löst Objective-C das allgemeine Konzept polymorpher Objekte. Da ähnlich wie in Smalltalk grundsätzlich nur ein Typ von Objekten (`id`) existiert, ist jedes Objekt potentiell polymorph. Jede Variable vom Typ `id` kann beliebige Objekte referenzieren[25].

Etwas differenzierter muß Polymorphismus in bezug auf Methoden betrachtet werden. Objective-C bietet die Möglichkeit von *overloading*. Dies bedeutet, mehrere Methodenimplementierungen an einen – dadurch polymorphen – Namen gebunden werden können. Welche Ausprägung der Methode verwendet wird, hängt in diesem Fall von den Argumenten des Methodenaufrufs ab. Allerdings kann in Objective-C im Unterschied zu C++ nicht alles überladen werden. Objective-C stellt kein *operator overloading* zur Verfügung.

Coercion als zweite Form von *ad hoc* Polymorphismus ist in Objective-C nur eingeschränkt notwendig und daher nicht voll in die Sprache integriert. Aus Kompatibilität zu ANSI C existieren natürlich implizite und explizite (*casts*) Konversionsfunktionen für Basistypen. Da aber nur eine Art von Objekten existiert und wie in Smalltalk nur Nachrichten an Objekte geschickt werden, auf die das Objekt antwortet, oder gegebenenfalls ein Laufzeitfehler auftritt, existiert keine über die Basisfunktionalität hinausgehende Form von *coercion*[26].

Inclusion Polymorphismus modelliert das Konzept der Vererbungshierarchie und wird daher auch in Objective-C unterstützt. Damit können Subklassen im Kontext von Superklassen verwendet werden. Desweiteren haben Subklassen somit die Möglichkeit, Methoden von Superklassen zu redefinieren (*overriding*). Wird also eine Subklasse in einem Superklassenkontext verwendet und eine Methode aufgerufen, die in der Subklasse redefiniert wurde, so wird anstelle der Superklassenmethode die redefinierte Methode ausgeführt. Im Unterschied zu C++, wo derart polymorphe Methoden explizit als `virtual` deklariert werden müssen, wird dies in Objective-C (wie in Smalltalk) implizit angenommen und es bedarf keiner speziellen Angaben.

[25] Komplexere Formen von polymorphen Objekten, bei denen, wie in C++, zwischen statischem und dynamischen Typ unterschieden werden muß, treten nur als Ausnahme bei der Verwendung von `static` Deklaration in Zusammenhang mit Zeigern und Referenzen auf (vgl. [Bud91]).

[26] *Coercion* im Zusammenhang mit Objekten kann aber sehr wohl über systemdefinierte Nachrichten `isKindOf` simuliert werden (vgl. [Bud91]).

Neben der Redefinition von Methoden besteht auch die Möglichkeit *deferred methods* zu definieren. In diesem Fall wird eine Methode von einer Superklasse nur deklariert, aber keine Implementierung der Funktionalität beigefügt. Es ist Aufgabe der Subklassen (*subclass responsibility*) eine an ihre Notwendigkeiten angepaßte Funktionalität zur Verfügung zu stellen (vgl. *pure virtual member functions* in C++).

Im Unterschied zu C++ kennt Objective-C nur *inclusion* Polymorphismus als Form von *universal* Polymorphismus. *Parametric* Polymorphismus als zweite Ausprägung dieser Art von Polymorphismus wird nicht unterstützt.

Da Objective-C in weiten Bereichen wie Smalltalk ungetypt ist, kommt dem Mechanismus des dynamischen Bindens zentrale Bedeutung zu. Objekte gelten in Objective-C als potentiell polymorph und zur Übersetzungszeit stehen daher keine Informationen über die Art von Objekten zur Verfügung, sodaß Methoden dynamisch zur Laufzeit, abhängig von der Klasse des Objekts gebunden werden müssen. Objective-C unterstützt zwar auch statisches Binden – bei entsprechender Deklaration von Objekten – doch ist dynamisches Binden eher die Regel als die Ausnahme. Der beim dynamischen Binden verwendete Algorithmus entspricht jenem von Smalltalk (vgl. Abschnitt 3.1.3).

3.3.7 Klassenbibliothek und Programmierumgebung

In bezug auf Klassenbibliotheken und Programmierumgebung gilt für Objective-C dasselbe wie bereits in Abschnitt 3.2.7 für C++ beschrieben. Zusätzlich dazu sei an dieser Stelle aber noch NeXTStep erwähnt. NeXTStep ist das objekt-orientierte Betriebssystem für Computer der Firma NeXT sowie PCs (Mach-Kernel mit 4.3BSD UNIX-Schale). Objective-C hat unter NeXTStep C als Systemsprache ersetzt. Demzufolge bietet NeXTStep eine vollständig integrierte Objective-C Entwicklungsumgebung mit umfangreichen Entwicklungswerkzeugen und Klassenbibliotheken (vgl. [Hui90]).

3.4 Eiffel

Eiffel wurde von B. Meyer [Mey92], [Mey93] entwickelt und stellt eine streng objektorientierte Programmiersprache dar, die sich auf die Implementierung möglichst wiederverwendbarer Software konzentriert. Eiffel ist keine Erweiterung einer prozeduralen Programmiersprache, sondern bietet eine strenge Unterstützung der objektorientierten Konzepte.

Die Modularität von Eiffel basiert auf dem Klassenkonzept. Eiffel erleichtert weiters die Anwendung von *design by contract* (vgl. [MM94], [Ner92]) und verbindet somit Implementierung und Design auf seine Weise.

3.4.1 Klassen und Objekte

Das zentrale Konzept in Eiffel ist die Klasse. Eine Klasse stellt die Implementierung eines abstrakten Datentyps dar (vgl. Abschnitt 2.1.2). Eine Klasse beschreibt somit eine Menge von Laufzeit-Objekten, die durch ihre Attribute (Daten) sowie ihre Methoden (in Eiffel Routinen genannt) charakterisiert werden.

Die Beschreibung einer Klasse besteht aus der Definition der sogenannten *features*, das sind die Attribute (repräsentieren die Daten der Objekte), sowie die Routinen (Methoden). Bei den Routinen wird in Eiffel zwischen Prozeduren (Methoden, die keine Ergebnisse zurückliefern) und Funktionen (Methoden, die Ergebnisse zurückliefern) unterschieden.

Die Sichtbarkeit (und somit Verwendbarkeit) von *features* kann in Eiffel explizit bei der Klassendefinition festgelegt werden. Für *features* können in Eiffel folgende Zugriffsniveaus angegeben werden:

- *Generally available features* sind *features*, die allen Klassen zugänglich sind.
- Die Sichtbarkeit von *features* kann auf einzelne Klassen beschränkt werden. Solche *features* werden als *selectively available features* bezeichnet. Diese *features* sind auch allen Subklassen der zugelassenen Klassen zugänglich.
- *Features* können auch keinen Klassen zugänglich sein. Solche *features* werden als *secret* bezeichnet.

Das folgende Beispiel zeigt die Definition einer einfache Klasse `POLYGON`:

```
class POLYGON feature
   -- Attributes
   side:REAL;
   -- Routines
   perimeter:REAL is
     do
       -- implementation of perimeter
     end; -- perimeter
...
```

Während es sich bei der Klasse also um eine Compile-Zeit Notation handelt, sind Objekte Instanziierungen solcher Klassen zur Laufzeit. Objekte müssen durch Konstruktoren (`'!!'`) explizit zur Laufzeit erzeugt werden und werden dann Variablen (in Eiffel *entities* genannt) zugewiesen, z.B.:

```
p:POLYGON;
...
!!p  -- creation of instance p
...
print(p.perimeter)
...
```

Eiffel bietet darüberhinaus noch die Möglichkeit, Eigenschaften von Klassen durch sogenannte *assertions* formal zu definieren. Solche *assertions* können in Eiffel in verschiedener Form auftreten:

- *Assertions* können zur Definition und Beschreibung von *pre-* und *postconditions* von Routinen (Methoden) verwendet werden.
- Klasseninvarianten (*class invariants*) müssen von jeder Instanz einer Klasse (i.e. Objekt) erfüllt werden, wann immer das Objekt von "außen" zugreifbar ist. Diese Invarianten repräsentieren also allgemeine Konsistenzbedingungen, die für alle Routinen der Klasse gelten.

3.4.2 Vererbung

Eiffel unterstützt das Konzept der Mehrfach-Vererbung (*multiple inheritance*) und unterscheidet sich dadurch von vielen anderen objekt-orientierten Programmiersprachen.

Grundsätzlich erbt eine Subklasse alle *features* der Superklasse. Eiffel bietet jedoch einige Möglichkeiten zur Adaptierung der Subklasse:

- Das Konzept des *renaming* ermöglicht die Auflösung von Namenskonflikten, die im Zuge der Mehrfachvererbung auftreten können (z.B. eine in zwei Superklassen gleichbezeichnete Routine). Darüberhinaus ermöglicht *renaming* das Anpassen von ererbten *features* an die lokale Umgebung der Subklasse, was in manchen Anwendungsfällen recht nützlich sein kann. Man beachte, daß das Konzept des *renaming* nur eine syntaktische Adaption, jedoch keine semantische Veränderung realisiert.

```
class C inherit
    A rename x as x1, y as y1 end;
    B rename x as x2, y as y2 end
feature ...
```

- Eiffel bietet natürlich das Konzept der Redefinition von *features* in der bereits von anderen Programmiersprachen bekannten Weise (Zuweisung einer neuen Implementierung, auch unter Änderung der Signatur).

  Zur Erhaltung der semantischen Konsistenz von Methoden über die Redefinition hinweg werden zusätzlich Zusicherungen (*assertions*) verwendet. Eine redefinierte Version einer Methode muß eine schwächere oder gleiche Vorbedingung (*precondition*) erfüllen und eine strengere oder gleiche Nachbedingung (*postcondition*) 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.

Das nachfolgende Beispiel zeigt eine Vererbungstruktur mit einer Redefinition der Routine `perimeter` für `RECTANGLE` ([Mey92]):

```
class RECTANGLE inherit
   POLYGON redefine perimeter end
 feature
    -- specific features of rectangles, such as:
     side1:REAL; side2:REAL;

     perimeter:REAL is
         -- rectangle-specific version
       do
         Result:= 2*(side1+side2)
       end; -- perimeter

     ... other RECTANGLE features ....
```

3.4.3 Typkonzept

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

Die Programmierumgebung von Eiffel stellt darüberhinaus eine automatische Initialisierung von Variablen sicher.

3.4.4 Polymorphismus und dynamisches Binden

Eiffel realisiert die meisten der in den vorigen Abschnitten eingeführten Arten von Polymorphismus.

Parametric Polymorphismus wird mittels generischer Klassen realisiert. Eine solche generische Klasse wird mit einem formalen, generischen Parameter spezifiziert. Jede dieser Klassen beschreibt ein *type template*, von dem man durch Übergabe eines konkreten Typs (eines aktuellen generischen Parameters) eine direkt verwendbare Klasse ableiten kann.

Das folgende Beispiel zeigt solche generische Klassen und deren konkrete Ableitungsmöglichkeiten [Mey92]:

```
ARRAY[G]
LIST[G]
-- generische Ableitung davon:
il: LIST[INTEGER];
aa: ARRAY[ACCOUNT];
aal: LIST[ARRAY[ACCOUNT]];
```

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 (vgl. dazu 3).

Wie in Abschnitt 3.4.2 bereits erwähnt, bietet Eiffel die Möglichkeit, Eigenschaften von Klassen teilweise oder auch ganz zu redefinieren. Durch dynamisches Binden wird dann zur Laufzeit ermittelt, welche der redefinierten Versionen (d.h. welche konkrete Implementierung) tatsächlich ausgeführt werden soll.

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 eingeschränkt.

In Eiffel können sich Programm-Elemente auf Objekte aus mehr als einer Klasse beziehen. Diese Art des Polymorphismus entspricht der Fähigkeit einer Größe, zur Laufzeit auf Instanzen unterschiedlicher Klassen zu verweisen. In der getypten Umgebung von Eiffel wird die Möglichkeit solcher Verweise durch das Vererbungsmodell eingeschränkt: In Eiffel dürfen einer Variablen einer Klasse nur Variablen derselben Klasse oder einer Subklasse zugewiesen werden (Typverträglichkeitsregel in Eiffel), d.h. daß Variablen einer Klasse nur Variablen derselben Klasse oder einer spezielleren Klasse als Werte übernehmen dürfen (z.B. einer Variablen vom Typ `Polygon` dürfen Werte des spezielleren Typs `Rechteck` zugewiesen werden, aber nicht umgekehrt, da Rechteck eine Subklasse von Polygon ist).

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

3.5 Modula-3

Modula-3 wurde von Digital Equipment Corporation's Systems Research Center (SRC) und Olivetti Research Center in den Jahren 1986 bis 1990 entwickelt [CDG+92], [Nel91], [Har92] und ist der Pascal-Sprachfamilie zuzuordnen. Modula-3 basiert auf Modula-2 von Wirth und auf Modula-2+, das von Digital selbst entwickelt wurde, ist jedoch zu keiner dieser Sprachen kompatibel.

Modula-3 erhält die Einfachheit und Typ-Sicherheit ihrer Vorgängersprachen und erweitert diese um wichtige Konzepte wie *exception handling*, automatische *garbage collection*, *concurrency* und objekt-orientierte Programmkonstrukte.

Modula-3 unterstützt objektorientierte Programmierung durch die in die Sprache eingebetteten Konzepte. Objektorientierung ist aber nicht das zentrale Konzept dieser Sprache, obwohl Konzepte wie Subtyping oder auch opake Typen unterstützt werden.

3.5.1 Klassen und Objekte

Durch das strenge Typkonzept von Modula-3 sind auch Objekte jeweils von einem bestimmten Typ, dem Objekttyp. 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.

In einer Klassendefinition findet sich die Definition für ihre Attribute und ihre Methoden, wobei angegeben werden kann, wo in der Vererbungshierarchie die neue Klasse eingeordnet werden soll. Die Methoden werden durch ihre Signaturen in der Objekttypdefinition festgelegt; deren implementierungsspezifische Realisierung erfolgt außerhalb der Deklaration mittels Prozeduren.

Objekte sind dabei stets Referenzen auf ein Daten-Record gemeinsam mit einer Methodenliste (*method suite*). Auf Objekttypen als solche kann nicht referenziert werden, sondern nur auf einzelne Teile von Objekten (Attribute oder Methoden).

Folgendes Beispiel zeigt eine Klassendefinition für einen Punkt (*Point*), der eine x- und eine y-Koordinate als Attribute und eine Methode `move` zur Verschiebung um einen Vektor (`deltaX, deltaY`) aufweist.

```
TYPE Point = OBJECT
       x, y: REAL := 0.0;
     METHODS
       move(deltaX, deltaY: REAL):= Move; (* Methodensignatur *)
     END;

     PROCEDURE Move(self: Point; deltaX, deltaY: REAL) =
       BEGIN
         self.x := self.x + deltaX;
         self.y := self.y + deltaY;
       END Move;
```

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), sodaß diese Funktionalität nicht vom Programmierer explizit definiert werden muß.

3.5.2 Vererbung

Das Vererbungskonzept von Modula-3 erlaubt einer Klasse (d.h. einem Objekttyp), nur von genau einer Superklasse zu erben (i.e. Einfachvererbung). Subtypen erben – wie im Vererbungskonzept des objektorientierten Paradigmas üblich – sämtliche Attribute und Methoden aller ihrer (direkten und indirekten) Supertypen entlang der Vererbungshierarchie. Genauso ist es möglich, in einer Subklasse zusätzliches Verhalten und weitere

Eigenschaften zu definieren bzw. zu redefinieren (vgl. dazu Abschnitt 3.5.3). Im folgenden Beispiel wird die Klasse `Point` um das Attribut `size` erweitert und somit eine neue Klasse `BigPoint` definiert:

```
TYPE BigPoint = Point OBJECT
       size: REAL := 1.0;
     END;
```

3.5.3 Methoden und dynamisches Binden

Die Methoden eines jeden Objekts werden in der Objekttypdefinition festgelegt. Dabei müssen einerseits die durch die Klassenhierarchie ererbten Methoden und andererseits zusätzliche Methoden definiert werden. In der Objekttypdefinition selbst wird nur die Signatur einer jeden Methode angeführt (vgl. dazu die obige Definition der Klasse `Point`).

Die zuvor definierten Objekte `Point` und `BigPoint` haben hinsichtlich der Implementierung die in Abb. 9 dargestellte Repräsentation. Es gibt für jedes Objekt eine *method suite*, worin alle Methoden des Objekts angeführt sind (dies entspricht den Signaturen, die bei der Objektdefinition angegeben werden). Die Implementierung einer jeden Methode findet sich dann außerhalb der Objektdefinition in Prozedurform; dies wird beispielsweise durch den Pointer von `move` auf `Move procedure` in Abb. 9 verdeutlicht.

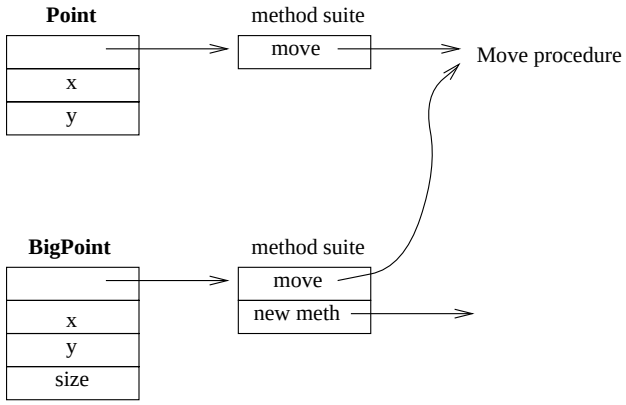

Abbildung 9: Objektimplementierung in Modula-3 [Har92]

Wird nun eine neue Subklasse (in unserem Beispiel `BigPoint`) gebildet, so finden sich in der *method suite* von `BigPoint` die Einträge der Superklasse `Point` und zusätzlich die neu definierten Methoden von `BigPoint`.

Die Redefinition von Methoden (*overriding*) hat letztendlich in der *method suite* zur Folge, daß der Zeiger für die normalerweise ererbte Methode auf eine andere Prozedur "umgehängt" wird. So kann dann ein `BigPoint` mittels `move` bewegt werden:

```
VAR point: Point;
...
point := NEW(Point);
point.move(1.0,2.0); (* wird zu Move(point,1.0,2.0) *)
(* oder: *)
point := NEW(BigPoint);
point.move(1.0,2.0); (* wird zu MoveBigPoint(point,1.0,2.0) *)
```

Durch dynamisches Binden wird erst zur Laufzeit entschieden, welche konkrete Implementierung zur Ausführung gelangt.

Ein Subtyp kann somit Methoden, die in einem Supertyp definiert sind, durch eine an seine spezifischen Anforderungen angepaßte ersetzen (*overriding*). Darüberhinaus gibt es in Modula-3 auch die Möglichkeit, einen *supercall* durchzuführen: Dabei handelt es sich um den Aufruf einer Methode der Superklasse, die in ihrer Subklasse überschrieben wurde. Für nähere Ausführungen sei der Leser auf [Har92], [Nel91] verwiesen.

3.5.4 Kapselung

Modula-3 unterstützt das Konzept des *information hiding* durch die Notation von Modulen als Compilierungseinheit. In vielen Sprachen wird für das *information hiding* auch das Klassenkonzept verwendet. Modula-3 unterscheidet 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; dabei kann es aber zu einem *interface* durchaus auch mehrere Implementierungsmodule geben. Die kann dazu verwendet werden, um einerseits eine allgemeine Schnittstelle und andererseits eine Schnittstelle für spezielle, besonders “vertrauenswürdige” Klienten anbieten zu können [Bös93].

Weiters bietet Modula-3 die Möglichkeit, generische *interfaces* zu bilden. Ein solches generisches *interface* stellt eine Art *template* dar, das 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.

In Modula-3 gibt es auch die Möglichkeit, Objekte zu abstrahieren und somit (für Benutzer) unnötige Details (z.B. Prozeduren für die Realisierung von Services, interne Daten, etc.) zu verbergen.

Um eine solche Abstraktion zu erreichen, werden in Modula-3 sogenannte *opaque types* unterstützt: 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 ggfs. private Prozeduren für die Realisierung solcher definiert.

Nachfolgendes Beispiel (aus [Har92]) zeigt eine solche opake Klasse *Positioned-Object*. Das Interface weist darin lediglich die für den Zugriff von außen notwendige Information (i.e. `init`).

```
INTERFACE PositionedObject;
   TYPE Point = RECORD x,y: REAL END;
        T <: Public;

        Public = OBJECT
        METHODS
          init(location: Point): Public;
        END; (* Public *)

        Rectangle <: PublicRectangle;

        PublicRectangle = T OBJECT
        METHODS
          init(topLeft: Point; height,width: REAL): PublicRectangle;
        END; (* PublicRectangle *)
END PositionedObject.
```

3.5.5 Polymorphismus

Modula-3 verwendet strukturelle Äquivalenz von Typen anstatt von Namensäquivalenz, die u.a. von Modula-2 verwendet wird. Strukturelle Äquivalenz bedeutet, daß zwei Typen dann gleich sind, wenn ihre Definitionen nach Expandierung gleich sind. Dies kann jedoch u.U. zu unerwünschten Effekten bei Prozeduraufrufen (strukturell gleiche Typen in unterschiedlichen Kontexten) führen.

Modula-3 bietet durch sein Typkonzept die Möglichkeit, daß Prozeduren, die einen bestimmten Typ `T` als Eingabetyp erwarten, auch Subtypen von `T` als Eingabeparameter akzeptieren und entsprechend verarbeiten können (*universal polymorphism*). Handelt es sich beispielsweise bei `T` um einen `INTEGER`, so werden auch Subtypen davon (z.B. `[1..100]`) problemlos akzeptiert.

Ähnliches gilt auch für Objekte: Da Objekte einen speziellen Typ, den Objekttyp, aufweisen, kann auch auf diese die Subtyprelation und die damit verbundene Ersetzungseigenschaft von Supertypen angewendet werden. Dies bedeutet, daß anstelle eines Objekttyps auch ein von ihm abgeleiteter Subtyp eingesetzt werden kann.

Da Methoden der Superklasse wie der Subklasse in der *method suite* die gleiche relative Position haben, kann somit der polymorphe Methodenaufruf zur Laufzeit einfach durchgeführt werden, da eine Subklasse seiner Superklasse weitgehend ähnlich ist.

Während Modula-3 *overloading* für Operatoren (z.B. `+, *,` etc.) realisiert, kennt es – wie sein Vorgänger Modula-2 – keine explizite Typkonversion (*coercion*). Eine notwendige Typkonversion muß vom Programmierer explizit durch dafür zur Verfügung gestellte Funktionen (z.B. `FLOAT(x,REAL)` konvertiert `INTEGER x` nach `REAL`) durchgeführt werden.

3.5.6 Zusammenfassung

Modula-3 ist eine streng getypte Sprache, die es dem Programmierer erlaubt, neben dem Konzept der Klasse, Module und *interfaces* für das *information hiding* zu verwenden. Damit kann eine gute Strukturierung von Modula-3 Programmen realisiert werden.

Da Objekte getypt sind, läßt sich die Subtyprelation auch auf Objekte anwenden: Dadurch kann relativ einfach ein Vererbungsmodell realisiert werden; die Regeln für eine Zuweisung folgen direkt aus dieser Subtyprelation. Die Ersetzungseigenschaft, die für Typen gilt, ist daher aufgrund dieser Relation auch auf Objekte anwendbar.

4 Schlußbemerkungen

Das Verständnis objekt-orientierter Programmierung bedarf der Klärung der grundlegenden Konzepte des objekt-orientierten Paradigmas. Dazu wurden in diesem Artikel zuerst die allgemeinen Konzepte dieses Paradigmas vorgestellt und einige weit verbreitete objekt-orientierte Programmiersprachen dahingehend betrachtet. Damit wird dem Anwender eine Möglichkeit geboten, sich im Bereich objekt-orientierter Programmiersprachen zu orientieren und somit einen Einstieg für eine weitere, detailliertere Beurteilung im konkreten Anwendungsfall zu finden.

Als Kriterien dieser Untersuchung wurden insbesondere die Objekt- und Klassendefinition, die Konzepte der Vererbung und des Polymorphismus, sowie des dynamischen Bindens betrachtet.

Da diese Konzepte in den einzelnen Programmiersprachen sehr unterschiedlich (syntaktisch wie semantisch) realisiert sind, wurde versucht, dem Leser durch die Ausführung mehrerer Beispiele ein Gefühl für die verschiedenen Sprachen zu vermitteln.

Obwohl die objekt-orientierten Konzepte am Beispiel der einzelnen Programmiersprachen kritisch beleuchtet wurden, kann dieser Artikel jedoch keine vollständige und vergleichende Bewertung der untersuchten Sprachen darstellen. Eine repräsentative Bibliographie soll dem Leser daher ermöglichen, sich einen weiterführenden Einblick in verschiedene objekt-orientierte Programmiersprachen zu verschaffen.

Danksagung Die Autoren danken Robert Barta und Georg Trausmuth für ihre konstruktiven und hilfreichen Anmerkungen.

Literatur

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilerbau*. Addison-Wesley Publishing Company, 1988.

[BL94] M. Beaudouin-Lafon. *Object-oriented Languages. Basic principles and programming techniques*. Chapman & Hall, 2–6 Boundary Row, London SE1 8HN, UK, 1994.

[Boo93] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway, Redwood City, California 94065, 1993.

[Bös93] L. Böszörmenyi. A comparison of modula-3 and oberon-2. *Structured Programming*, 14:15–22, 1993.

[Bud91] T. Budd. *An Introduction to Object Oriented Programming*. Addison-Wesley, 1991.

[CDG+92] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 language definition. *ACM SIGPLAN Notices*, 27(8), August 1992.

[Cox86] B.J. Cox. *Object Oriented Programming. An Evolutionary Approach*. Addison-Wesley, 1986.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4), December 1985.

[EGL89] H.-D. Ehrich, M. Gogolla, and U. W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen*. B. G. Teubner, Stuttgart, 1989.

[GB89] P. Grogono and A. Bennett. Polymorphism and type checking in object-oriented languages. *ACM SIGPLAN Notices*, 24(11), November 1989.

[Gol84] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Har92] S.P. Harbison. *Modula-3*. Prentice-Hall, 1992.

[Hat94] B. Hathaway, editor. *comp.object FAQ*. Usenet, June 1994.

[Hui90] G. Huizenga. Intro to objective c on the next machine. Technical report, Purdue University Computing Center, Purdue Research Foundation, 1990.

[Jor90] D. Jordan. Implementation benefits of c++ language mechanisms. *Communications of the ACM*, 33(9), September 1990.

[Kee89] S.E. Keene. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, 1989.

[KM90] T. Korson and J.D. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9), September 1990.

[LD94] X.-M. Lu and T. S. Dillon. An algebraic theory of object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):412–419, June 1994.

[Lip92] S.B. Lippman. *C++ Primer*. Addison-Wesley, zweite edition, 1992.

[Liu91] C.-S. Liu. On the object-orientedness of c++. *ACM SIGPLAN Notices*, 26(3), March 1991.

[Mey87a] B. Meyer. Eiffel: Programming for reusability and extendibility. *ACM SIGPLAN Notices*, 22(2), 1987.

[Mey87b] B. Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

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

[Mey93] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1993.

[Mit89] J. Mittendorfer. *Objektorientierte Programmierung mit C++ und Smalltalk*. Addison-Wesley, 1989.

[MM94] D. Mandrioli and B. Meyer. *Advances in Object-Oriented Software Engineering*. Prentice-Hall, 1994.

[Nel91] G. Nelson. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, 1991.

[Ner92] J.-M. Nerson. Applying object-oriented analysis and design. *Communications of the ACM*, 35(9):63–74, Septermber 1992.

[Pok89] B.P. Pokkunuri. Object oriented programming. *ACM SIGPLAN Notices*, 24(11), November 1989.

[PW88] L.J. Pinson and R.S. Wiener. *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, 1988.

[Sch94] P. Schoenmakers, editor. *Comp.Lang.Objective-C FAQ*. Usenet, June 1994.

[Sha89] J.E. Shapiro. An example of multiple inheritance in c++: A model of the iostream library. *ACM SIGPLAN Notices*, 24(12), December 1989.

[Str94] B. Stroustrup. *Die C++ Programmiersprache: erweitert um Entwürfe zur ANSI-/ISO-Standardisierung*. Addison-Wesley, 1994.

[Weg87] P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA '87*, pages 168 – 182, 1987. Veröffentlicht in *ACM SIGPLAN Notices* 22(12).

[Weg90] P. Wegner. Concepts and paradigms of object-oriented programming. In *OOPS Messenger*, pages 7 – 87. ACM Press, 1990.

[WP88] R.S. Wiener and L.J. Pinson. *An Introduction to Object-Oriented Programming and C++*. Addison-Wesley, 1988.

[WP89] R.S. Wiener and L.J. Pinson. A practical example of multiple inheritance in c++. *ACM SIGPLAN Notices*, 24(9), September 1989.