Kapitel 5

Übersetzung objektorientierter Sprachen

Softwaresysteme werden zunehmend komplexer und größer. Damit wächst die Notwendigkeit, die Entwicklung solcher Systeme effizienter und transparenter zu machen. Fernziel ist es, Softwaresysteme – wie heute schon Hardwaresysteme (und die allermeisten Produkte des täglichen Lebens, z.B. Autos, Waschmaschinen usw.) – aus vorgefertigten Standardbausteinen zusammenzusetzen. Diesem Ziel versucht man durch Fortschritte u.a. in folgenden Bereichen näherzukommen:

- Modularisierung
- Wiederverwendbarkeit von Modulen
- Erweiterbarkeit von Modulen
- Abstraktion.

Objektorientierte Sprachen bringen neue Möglichkeiten in diesen Bereichen. Objektorientierung wird deshalb heute als ein wesentliches Paradigma angesehen, um die Komplexität von Softwaresystemen zu beherrschen.

In diesem Kapitel skizzieren wir die wichtigsten neuen Konzepte objektorientierter Sprachen und zu ihrer Implementierung verwendbare Übersetzungsschemata.

5.1 Konzepte objektorientierter Sprachen

Objektorientierte Sprachen sind imperativen Sprachen weit näher verwandt als funktionale oder gar logische Programmiersprachen. So benutzen sie (typischerweise) dasselbe Ausführungsmodell: eine Variante des von-Neumann-Rechners; dabei wird unter expliziter Programmkontrolle ein komplexer *strukturierter* Zustand verändert. Man kann objektorientierte Sprachen ansehen als imperative Sprachen, die zu deren bekannten Konzepten wie Variablen, Felder, Strukturen, Prozeduren und Funktionen einige neue Konzepte hinzufügen. Nur auf einige dieser „*neuen*" Konzepte[1] werden wir in diesem Kapitel eingehen. Im Zentrum steht dabei die Festlegung einer angemessenen Modularisierungseinheit.

[1] Tatsächlich sind diese Konzepte nicht alle ganz neu, sondern wurden zum Teil bereits früher in moderneren imperativen oder im Bereich der künstlichen Intelligenz verwendeten Sprachen benutzt.

5.1.1 Objekte

Die Hauptmodularisierungseinheit imperativer Sprachen ist die Funktion (bzw. die Prozedur[2]). Funktionen können sehr komplexe Abläufe kapseln und abstrahieren, etwa das Lösen von Optimierungsaufgaben, linearen oder nichtlinearen Gleichungssystemen und Differentialgleichungen. Solange die Komplexität der Daten gegenüber der der Abläufe vernachlässigbar ist, ist dies eine geeignete Abstraktions- und Modularisierungsstufe. Für Aufgaben, deren Beschreibung und effiziente Lösung die Verwendung komplexer Datenstrukturen erfordert, sind Funktionen allein als Modularisierungseinheit nicht ausreichend. Eine angemessene Abstraktionsstufe für die effiziente Bearbeitung solcher Aufgaben muß erlauben, sowohl die Datenstrukturen als auch die zugehörigen auf diesen Strukturen arbeitenden Funktionen in einer Einheit zu kapseln und als Modularisierungseinheit zu verwenden.

Das Grundkonzept objektorientierter Sprachen ist das **Objekt**. Ein Objekt besteht aus einem Objektzustand, ausgedrückt durch die aktuellen Werte eines Satzes von **Attributen**, und Funktionen, die auf diesem Zustand arbeiten, **Objekt-Methoden** genannt. Attribute und Methoden eines Objektes zusammen nennen wir seine **Merkmale**. Ein Objekt kapselt damit in seinem Zustand sowohl Daten als auch in seinen Methoden die auf diesen Daten durchführbaren Operationen. Zu den wichtigsten Basisoperationen objektorientierter Sprachen gehört die Aktivierung einer Methode `m` zu einem Objekt `o`, etwa geschrieben als `o.m`. Im Vordergrund steht dabei das Objekt; die Methode ist als sein Bestandteil diesem untergeordnet. Diese Objektorientierung hat der Sprachklasse ihren Namen gegeben.

Wir wollen den Nutzen der Objektorientierung an einem ersten Beispiel verdeutlichen: an Bausteinen zur Bearbeitung (zweidimensionaler) graphischer Objekte. Es gibt viele Typen graphischer Objekte: Kreise, Ellipsen, Rechtecke, Dreiecke, Polygone, Punkte, Linien, Linienzüge und viele weitere mehr. Durch Gruppierung graphischer Objekte erhält man einen weiteren Typ: zusammengesetzte graphische Objekte. Jeder dieser verschiedenen Typen hat seine Besonderheiten, aber es gibt auch Gemeinsamkeiten. So wird ein brauchbarer Baukasten für die Arbeit mit graphischen Objekten für jeden Typ mindestens folgende Operationen zulassen:

- Kopieren,
- Verschieben,
- Löschen
- und eventuell Skalieren.

Für viele Abläufe ist der genaue Typ nicht relevant und sollte deshalb auch nicht angegeben werden müssen. So ist etwa die angemessene Beschreibung für das Kopieren eines zusammengesetzten Objektes: „Kopiere die Unterobjekte des zusammengesetzten Objektes und gruppiere die Kopien zu einem neuen zusammengesetzten Objekt". Die genauen Typen der Unterobjekte sind für die Beschreibung des Ablaufs irrelevant. In einer objektorientierten Implementierung dieses Problems[3] enthält jedes graphische Objekt seine eigene Kopierfunktion, die ohne Kenntnis des genauen Objekttyps aktiviert werden kann. Um dies mit einer funktionsorientierten Implementierung zu erreichen, muß eine Kopierfunktion definiert werden, die in der Lage ist, graphische Objekte *jeden* Typs zu kopieren. Diese Funktion wird den Typ des zu kopierenden Objektes ermitteln und die entsprechende typspezifische Kopierfunktion aktivieren. Nachteilig daran ist, daß über diese Funktion und die von ihr interpretierte Datenstruktur zur Repräsentation graphischer Objekte *alle* Typen graphischer Objekte miteinander gekoppelt sind. Als eine Folge davon enthält ein Programm mit der allgemeinen Kopierfunktion auch sämtliche typspezifischen Kopierfunktionen, selbst wenn es viele Typen graphischer Objekte überhaupt nicht benötigt. Es wird unnötig groß. Darüber hinaus werden Erweiterungen erschwert. Wird in dem Baukasten ein weiterer Basistyp benötigt, dann muß die allgemeine Kopierfunktion ebenso erweitert werden wie die von ihr interpretierte Datenstruktur. Im schlimmsten Fall müssen alle Bausteine neu übersetzt werden, dann nämlich, wenn sie von dieser Datenstruktur abhängen. Bei einer objektorientierten Implementierung wird der neue Basistyp einfach hinzugefügt. Der gesamte Rest des Baukastens kann unverändert und ohne Neuübersetzung weiterbenutzt werden – eine wesentliche Verbesserung hinsichtlich Modularisierbarkeit und Erweiterbarkeit.

[2] Wir werden im folgenden stets von Funktion sprechen, obwohl es sich eigentlich nicht um Funktionen im mathematischen Sinne handelt, sondern um Prozeduren mit oder ohne Rückgabewert.

5.1.2 Objektklassen

Um die Programmentwicklung sicherer und effizienter zu machen, ist es wünschenswert, daß Inkonsistenzen und Fehler in Programmen möglichst frühzeitig und möglichst zuverlässig erkannt werden. Übersetzer können hierzu einen Beitrag leisten, da sie Programmteile zum Teil eingehend analysieren. Ihre Prüfung auf Konsistenz und Fehlerfreiheit ist dabei notwendigerweise nicht vollständig, da sie nur mit partiellen Spezifikationen arbeiten. Typinformation spielt dabei eine wesentliche Rolle.

(Statische) Typinformation besteht aus Angaben zu den in einem Programm verwendeten Namen. Sie legen fest, daß die an diese Namen gebundenen Laufzeitobjekte einen angegebenen Typ besitzen werden bzw. müssen. Ein Typ steht dabei für eine Menge zulässiger Werte; der Typ bestimmt in hohem Maße, wie das in dem Laufzeitobjekt abgelegte Bitmuster dekodiert werden muß. Übersetzer können Typinformation nutzen, um

- effizienteren Code zu erzeugen,
- Mehrdeutigkeiten in der Operatorverwendung aufzulösen,
- automatische Typkonversionen einzufügen,
- Inkonsistenzen zu erkennen.

[3] Eine objektorientierte Implementierung kann u.U. auch mit einer imperativen Sprache realisiert werden. Das gilt etwa für C; dabei muß allerdings das Typsystem dieser Sprache an mindestens einer Stelle durch explizite Typkonversion umgangen werden.

Statische, das heißt dem Übersetzer bekannte oder von ihm ableitbare Typinformation ist eine wichtige Voraussetzung dafür, daß er effiziente und einigermaßen zuverlässige Programme erzeugen kann.

Objektorientierte Sprachen erweitern das von imperativen Sprachen wie etwa `Pascal` oder `C` bekannte Typkonzept.[4] Ihre Typen heißen üblicherweise **Objektklassen**. Eine Objektklasse legt Attribute und Methoden fest, die ein Objekt haben muß, um zu dieser Klasse gehören zu können. Für Attribute wird ihr Typ und für Methoden ihr Prototyp (Typen für Rückgabewert und Parameter) vorgegeben. Einige objektorientierte Sprachen, so etwa Eiffel, erlauben weitere Festlegungen für die Methoden, z.B. Vor- und Nachbedingungen. Auf diese Weise kann die Bedeutung der Methode (Semantik) eingeschränkt werden. Häufig definiert die Klasse auch die Methoden; diese Definitionen können jedoch unter gewissen Bedingungen überschrieben werden. Objekte, die neben den geforderten weitere Merkmale besitzen, können u.U. ebenfalls der Klasse angehören.

Die Objektklasse bildet die eigentliche Modularisierungseinheit objektorientierter Sprachen. So ist unser Baukasten zur Handhabung graphischer Objekte aus dem vorigen Abschnitt als Klassenbibliothek mit Objektklassen für die verschiedenen Typen graphischer Objekte realisiert.

Wir werden im folgenden die Begriffe „*Klasse*“ und „*Typ*“ synonym gebrauchen.

5.1.3 Vererbung

Unter **Vererbung** versteht man die Übernahme aller Merkmale einer Klasse `A` in eine neue Klasse `B`. `B` kann zusätzlich Merkmale definieren und unter gewissen Voraussetzungen von `A` geerbte Methoden überschreiben. Einige Sprachen erlauben, daß geerbte Merkmale umbenannt werden, um Namenskonflikte zu vermeiden oder einfach einen im neuen Kontext aussagekräftigeren Namen verwenden zu können.

Erbt `B` von `A`, dann heißt `B` eine von `A` **abgeleitete Klasse**; `A` heißt **Basisklasse** von `B`.

Die Vererbung gehört zu den wichtigsten Konzepten objektorientierter Sprachen. Sie vereinfacht Erweiterungen und Variantenbildung entscheidend. Durch Bildung von Vererbungshierarchien erlaubt sie ferner eine Strukturierung von Klassenbibliotheken und das Einziehen verschiedener Abstraktionsstufen.

Wir veranschaulichen dies wieder an unserem Beispiel mit graphischen Objekten. Abb. 5.1 zeigt einen Ausschnitt aus der Vererbungshierarchie dieser Klassenbibliothek mit den Objektklassen `graphical object` (allgemeines graphisches Objekt), `closed graphical` (geschlossenes graphisches Objekt), `ellipse` (Ellipse), `polyline` (Linienzug), `polygon` (Polygon), `rectangle` (Rechteck) und `triangle` (Dreieck).

In dieser Abbildung sind Objektklassen durch Ellipsen veranschaulicht. In den Ellipsen stehen der Name der Objektklasse sowie eine Auswahl der von dieser

[4] Nicht alle objektorientierten Sprachen benutzen statische Typisierung, das gilt etwa für Smalltalk-80.

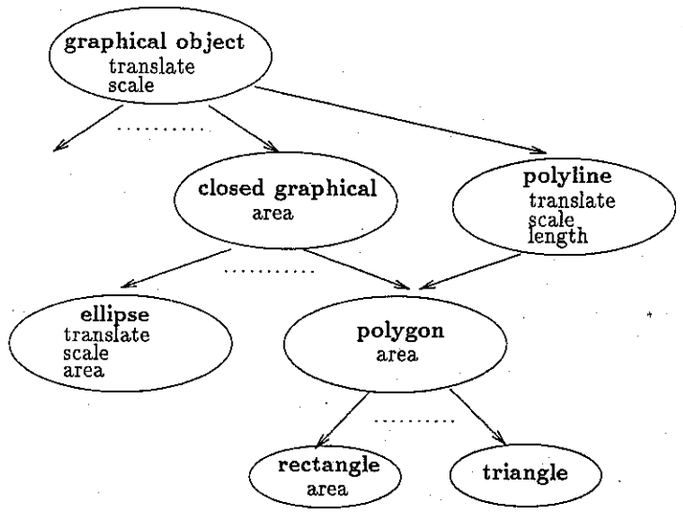

Abb. 5.1: Vererbungshierarchie graphischer Objekte

Klasse eingeführten Methoden. Vererbung wird durch einen Pfeil repräsentiert. So führt die Klasse `graphical object` die Methoden `translate` und `scale` ein: Alle graphischen Objekte können verschoben und skaliert werden. Die Klassen `closed graphical` und `polyline` erben diese Methoden von `graphical object`. In `polyline` werden die geerbten Methoden `translate` und `scale` überschrieben und die Methode `length`, die Länge des Linienzugs, neu aufgenommen. `Closed graphical` führt die neue Methode `area` ein, die die von dem geschlossenen graphischen Objekt eingeschlossene Fläche liefert. Die Klasse `polygon` erbt sowohl von `closed graphical` als auch von `polyline`; `area` wird überschrieben. `rectangle` schließlich erbt von `polygon` und überschreibt `area`.

Die Methoden `translate` und `scale` werden in der Klasse `graphical object` zwar eingeführt, können dort aber noch nicht definiert werden. Erst, wenn graphische Objekte erstmals konkret realisiert werden – in unserem Beispiel in den Klassen `ellipse` und `polyline`, können die Methoden für Verschieben und Skalieren eine Definition erhalten. Daß die Methoden trotzdem schon in `graphical object` aufgenommen wurden, soll zum Ausdruck bringen, daß jedes graphische Objekt diese Methoden besitzen muß, ohne daß etwas über ihre Implementierung ausgesagt werden kann. Klassen, die undefinierte Methoden enthalten, werden **abstrakt** genannt. Sie enthalten keine eigenen Objekte, das heißt Objekte, die nicht aus einer echten Unterklasse stammen.

Analog dazu führt die Klasse `closed graphical` die Methode `area` ein, ohne sie definieren zu können. `area` kann erst in den Klassen `ellipse` und `polygon` definiert werden. Die Flächenberechnung für allgemeine Polygone ist jedoch kom-

plex und basiert auf einer Triangulierung der von dem Polygon eingeschlossenen Fläche und einer Summation der entstehenden Dreiecksflächen. Im Vergleich dazu ist die Flächenberechnung für ein Rechteck trivial. Sofern Rechtecke in dem anvisierten Anwendungsbereich häufig verwendet werden (und öfter ihre Fläche bestimmt werden muß), ist es günstig, `area` in `rectangle` neu und effizienter zu realisieren, wie wir dies in unserem Beispiel getan haben. Demgegenüber ist es naheliegend, für Polygone alle Methoden von `polyline` unverändert zu übernehmen.

Wir sehen, daß das Vererbungskonzept uns eine Möglichkeit schafft, in einfacher Weise auf Teile einer bestehenden Implementierung zurückzugreifen, sie zu erweitern und sie bei Bedarf lokal – durch Überschreiben einzelner Methoden – an spezielle Anforderungen oder Gegebenheiten anzupassen.

Ferner erhalten wir die Möglichkeit, abstrakte Klassen zu definieren. Dies führt eine ähnliche Flexibilität in Programmiersprachen ein, wie wir sie in der natürlichen Sprache durch abstrakte Begriffe gewinnen: Wir erhalten unterschiedliche Abstraktionsstufen. Was auf einer hohen Abstraktionsstufe formuliert werden kann, hat einen sehr weiten Anwendungsbereich und damit ein hohes Maß an Wiederverwendbarkeit. Wir sind daher bestrebt, eine möglichst hohe Abstraktionsstufe zu verwenden. Andererseits werden wir gelegentlich gezwungen, auf eine konkretere Stufe überzugehen, weil dort mehr Struktur für die Lösung spezifischer Aufgaben zur Verfügung steht. Nehmen wir an, eine Transformation eines graphischen Objektes läßt sich durch eine Folge von Translationen, Skalierungen und eventuell weiterer für alle graphischen Objekte definierter Operationen beschreiben, dann kann diese Transformation durch eine einzige Funktion realisiert werden – anwendbar auf jedes graphische Objekt, unabhängig von seinem Typ. Hätten wir keine abstrakten Klassen (aber eine Typprüfung durch den Übersetzer), dann bräuchten wir für jede konkrete Klasse eine eigene Funktion, obwohl ihre Implementierung immer gleich aussehen würde.

Typisierte objektorientierte Sprachen berücksichtigen deshalb die Vererbungshierachie in ihrem Typsystem. Erbt eine Klasse `B` von der Klasse `A`, dann ist der `B` zugeordnete Typ ein Teiltyp des Typs zu `A`. Jedes Objekt eines Teiltyps ist automatisch auch Element des Obertyps; eine erbende Klasse wird Teilklasse der beerbten Klasse. Dies hat folgende Auswirkung:

Teiltypregel: Wird auf einer Eingabeposition (Funktionseingabeparameter, rechte Seiten von Zuweisungen) oder als Funktionsrückgabewert ein Objekt eines bestimmten Typs verlangt, so dürfen Objekte eines beliebigen Teiltyps übergeben werden.

Wir nennen die Objekte von `B`, die nicht auch Objekte einer echten Teilklasse sind, die **eigenen Objekte** von `B` oder auch die **eigentlichen `B`-Objekte**. Wir nennen `B` den **eigentlichen Typ** der eigentlichen `B`-Objekte. Damit besitzt jedes Objekt einen eindeutig bestimmten eigentlichen Typ: den kleinsten Typ, zu dem das Objekt gehört. Es ist darüber hinaus Element jedes Obertyps seines eigentlichen Typs.

Aufgrund der Teiltypregel können Methoden und Funktionen in objektorientierten Sprachen etwa auf einer Parameterposition Objekte mit verschiedenen eigentlichen Typen und damit unterschiedlichem Aufbau akzeptieren. Dies ist eine Form von **Polymorphismus**.

Die Teiltypregel der Vererbung zusammen mit der Möglichkeit, daß erbende Klassen eine geerbte Methode überschreiben dürfen, hat eine interessante und für Übersetzer wichtige Konsequenz. Betrachten wir beispielhaft eine Funktion `f`, die als Parameter Objekte der Klasse `closed graphical` zuläßt, und nehmen an, daß sie die Methode `area` dieses Parameters aufruft. Da `closed graphical` die Methode `area` nicht definieren kann, ist offensichtlich, daß in `f` notwendigerweise die Methode `area` des Parameters und nicht die der Klasse `closed graphical` aufgerufen werden muß. Dies gilt allgemein:

Methodenauswahlregel: Überschreibt eine Klasse `B` eine Methode `m` einer beerbten Klasse `A`, dann *muß* für ein Objekt `b` aus `B` auch dann die von `B` (oder u.U. einer seiner Teilklassen) definierte Methode aufgerufen werden, wenn `b` als Element von `A` benutzt wird; die eventuell von `A` definierte Methode darf für `b` nicht benutzt werden.

Diese Regel stellt einen Übersetzer vor ein kleines Problem: Er muß Code für die Aktivierung einer Methode erzeugen, die er zu diesem Zeitpunkt u.U. noch nicht kennt. Als Folge davon kann beispielsweise der Übersetzer bei der Codeerzeugung für `f` den Methodennamen `area` nicht an eine konkrete Methode binden. Diese Bindung kann erst bei Kenntnis des aktuellen Parameters und damit im allgemeinen zur Programmlaufzeit durchgeführt werden: Wir sprechen dann von **dynamischer Bindung** im Gegensatz zur **statischen Bindung**, bei der der Übersetzer die Bindung selbst vornimmt. Wir können die Methodenauswahlregel damit auch folgendermaßen formulieren:

Dynamische-Bindungs-Regel: Eine Methode eines Objektes `o`, die potentiell in einer Teilklasse überschrieben wird, muß dynamisch gebunden werden, wenn der Compiler den eigentlichen Typ von `o` nicht ermitteln kann.

Der Hauptteil dieses Kapitels wird sich mit einer effizienten Implementierung der Vererbung beschäftigen.

5.1.4 Generizität

Streng typisierte Sprachen zwingen häufig zu einer Reimplementierung der gleichen Funktion für verschiedene Typinstantiierungen. Oft unterscheiden sich die Funktionen ausschließlich im Typ ihrer Parameter und – notwendigerweise – in ihrem Namen. Diese mehrfachen Funktionsinstanzen erschweren die Implementierung, machen Programme unübersichtlicher und schwerer wartbar.

Wir haben gesehen, daß das auf Vererbung beruhende Typkonzept objektorientierter Sprachen uns in einigen Fällen eine Duplikation von Funktionsimplementierungen erspart. Für eine wichtige Klasse von Problemstellungen führt Vererbung allein aber nicht zu eleganten Lösungen: für die Implementierung allgemeiner Behälterdatenstrukturen wie Listen, Mengen, Keller, Wartenschlangen...

Diese Behälterdatenstrukturen haben einen natürlichen Parameter: den Typ ihres Inhalts. **Generizität** (engl. **genericity**) ermöglicht uns, für solche Datenstrukturen und ihre Methoden eine Mehrfachimplementierung zu vermeiden. Sie erlaubt, Typdefinitionen (und Funktionsdefinitionen) zu parametrisieren; als Parameter sind selbst wieder Typen zugelassen. Eine einzige Definition für die parametrisierte Klasse `List<T>` kann beispielsweise Listen mit beliebigem Basistyp beschreiben. Listen mit spezifischem Basistyp werden durch **Instantiierung** der generischen Klasse erzeugt: `List<int>`, `List<List<int>>` beschreiben etwa Listen ganzer Zahlen oder Listen solcher Listen. Wie Prozeduren nicht eine einzelne Berechnung, sondern eine ganze Klasse von Berechnungen beschreiben, so wird durch einen generischen Typ nicht ein einzelner Typ, sondern eine ganze Klasse von Typen beschrieben. Wie durch Aufruf einer Prozedur mit aktuellen Parametern eine Berechnung realisiert wird, so wird durch die Instantiierung einer generischen Klasse mit aktuellen Parametern ein spezieller Typ realisiert: Generische Typen sind die „*Prozeduren der Typwelt*".

Generizität wird von einigen objektorientierten Sprachen unterstützt. Sie ist allerdings keine Erfindung objektorientierter Sprachen. Die imperative Sprache `Ada` hatte bereits früher ein sehr ausgefeiltes Konzept von Generizität.

5.1.5 Informationskapselung

Die meisten objektorientierten Sprachen stellen Konstrukte zur Verfügung, mit denen die Merkmale einer Klasse als **privat** oder **öffentlich** klassifiziert werden können. Private Merkmale sind in gewissen Kontexten entweder ganz unsichtbar oder zumindest nicht zugreifbar. Manche objektorientierten Sprachen unterscheiden verschiedene Sichtbarkeitskontexte: etwa „*innerhalb der Klasse*", „*in abgeleiteten Klassen*", „*in fremden Klassen*", „*in bestimmten Klassen*" usw. Sprachkonstrukte oder allgemeine Regeln können festlegen, in welchen Kontexten welche Merkmale sichtbar bzw. lesbar/schreibbar oder aufrufbar sind.

Die Realisierung solcher Konstrukte durch einen Übersetzer ist einfach und naheliegend. Wir werden daher in diesem Kapitel nicht weiter darauf eingehen, obwohl die Informationskapselung von großer Bedeutung ist – für eine klare Trennung zwischen der abstrakten Sicht der Klassenbedeutung und der konkreten Sicht ihrer Implementierung.

5.1.6 Zusammenfassung

Wir fassen die wichtigsten Ergebnisse unserer bisherigen Diskussion zusammen:

- Objektorientierte Sprachen bringen eine neue Modularisierungseinheit: **Objektklassen**. Objektklassen können sowohl Daten als auch auf diesen Daten arbeitende Funktionen kapseln. Die Berücksichtigung von Daten *und* Funktionen in einer Einheit ermöglicht die Realisierung natürlicher, abgeschlossener und leicht zu integrierender und zu erweiternder Module.

- Das **Vererbungskonzept** ist ein übersichtliches, mächtiges und bequemes Hilfsmittel zur Erweiterung und Variantenbildung bestehender Module (Objektklassen).
- Das Typsystem objektorientierter Sprachen nutzt das Vererbungskonzept: Erbende Klassen werden Teiltypen der Basisklassen; ihre Objekte können an fast allen Stellen benutzt werden, an denen Objekte der Basisklasse zulässig sind. Rein von ihrer Funktionalität her betrachtet, erben sie so auch den Anwendungsbereich ihrer Eltern und sind daher sehr leicht in bestehende Systeme integrierbar.
- **Vererbungshierarchien** führen unterschiedliche Abstraktionsebenen in Programme ein. Dies erlaubt, an verschiedenen Stellen innerhalb eines Programms oder Systems je nach Zweckmäßigkeit auf unterschiedlichen Abstraktionsstufen zu arbeiten: etwa auf der Stufe `graphical object` für die Implementierung allgemeiner graphischer Funktionen wie `zoom` (Vergrößern und Ausrichten) oder `group` (Gruppieren), aber auf der Stufe `closed graphical` für die Implementierung von Funktionen, in die Flächenberechnungen eingehen.
- **Abstrakte** Klassen können in Spezifikationen verwendet und durch schrittweise Vererbung verfeinert und schließlich realisiert werden. Der Übergang zwischen Spezifikation, Entwurf und Realisierung wird fließend. Unterschiedliche Modulvarianten, z.B. Prototyp, Produktionsversion oder speziell optimierte Realisierungen, können je nach Entwicklungsphase und Anforderungen leicht in ein Gesamtsystem integriert und gegeneinander ausgetauscht werden.
- **Generizität** erlaubt, Klassendefinitionen zu parametrisieren. So können etwa allgemeine Algorithmen und zugehörige Datenstrukturen wie Listen, Stacks, Warteschlangen, Mengen... unabhängig vom Elementdatentyp implementiert werden.

Beispiele für objektorientierte Sprachen sind etwa `C++` und `Eiffel`. In diesem Kapitel werden wir hauptsächlich diese beiden Sprachen als Beispiel heranziehen. Einige neuere `C++`-Implementierungen unterstützen alle skizzierten Konzepte – jedoch nicht immer in vollem Umfang. Von `Eiffel` werden sie uneingeschränkt unterstützt. Darüber hinaus bietet `Eiffel` eine Reihe weiterer interessanter Konstrukte, die die Entwicklung sicherer, wiederverwendbarer Module fördern: etwa Klasseninvarianten, Vor- und Nachbedingungen für Methoden, Invarianten (Assertions) und Schleifeninvarianten, Ausnahmebehandlung. Zur Zeit gewinnt die objektorientierte Sprache `Java` stark an Bedeutung – als Basis plattformunabhängiger, über das WWW ladbarer und im WWW-Browser ausführbarer Module. Als Urvater objektorientierter Sprachen gilt `Simula 67`, eine Erweiterung von `Algol 60` um Objektklassen, einfache Vererbung, Coroutinen und Primitive zur Unterstützung diskreter Simulationen. Ein weiterer bekannter Vertreter ist `Smalltalk-80`, eine normalerweise interpretierte Sprache ohne statische Typisierung, mit einfacher Vererbung, in der Klassen selbst wieder Objekte sind, die unter Programmkontrolle verändert werden können. Weitere Vertreter sind

Objective-C, eine Erweiterung von C um Smalltalk-80-Konzepte, und die objektorientierten Erweiterungen von Lisp, Loops, Flavors sowie Ceyx.

5.2 Die Übersetzung von Methoden

Wir beginnen diesen Abschnitt mit einen konkreten Beispiel. Hierzu realisieren wir einen Teilausschnitt aus der Klassenhierarchie graphical object des vorigen Abschnitts in der objektorientierten Sprache C++.[5]

Wir definieren zunächst die Klasse graphical object:

```
struct graphical_object {
  virtual void translate(double x_offset, double y_offset);
  virtual void scale(double factor);
  // evtl. weitere allgemeine Methoden der Klasse
};
```

Die Klasse graphical object

Wir beachten, daß die Methoden translate und scale als virtuell deklariert wurden; in C++ ist dies Voraussetzung dafür, daß eine Methode von einer abgeleiteten Klasse überschrieben werden kann.

Eine besonders wichtige Teilklasse der graphischen Objekte sind die Punkte. Punkte werden in der Implementierung fast jeder konkreten Klasse graphischer Objekte in der einen oder anderen Weise benutzt. Die Klasse der Punkte wird folgendermaßen definiert:

```
class point : public graphical_object {
  double        xc, yc;
public:
  void translate(double x_offset, double y_offset) {
    xc+=                x_offset;
    yc+=                y_offset;
  }
  void scale(double factor) {
    xc*=                factor;
    yc*=                factor;
  }
  point(double x0=0, double y0=0) { xc= x0; yc= y0; }
  void set(double x0, double y0) { xc= x0; yc= y0; }
  double x(void) { return xc; }
  double y(void) { return yc; }
  double dist(point &);
};
```

Die Klasse point

[5] Der Ausschnitt und alle weiteren Beispiele aus diesem Kapitel wurden mit dem GNU C++ Compiler Version 2.5.6 bearbeitet. Dieser Compiler ist „*freie Software*". Er ist lauffähig auf einer Vielzahl von Unix-Plattformen, mit einem speziellen Extender auch unter DOS und Windows. Er kann – einschließlich Quellen – beispielsweise über Internet bezogen werden. Die GNU GENERAL PUBLIC LICENCE regelt die Nutzungsrechte. Sie ist jeder legalen Distribution beigefügt.

Punkte enthalten ihre x- und y-Koordinate, xc und yc. Diese beiden Koordinaten legen die Lage des Punktes innerhalb des zweidimensionalen Raums fest. Sie sind private Daten des Punktes: Auf sie kann nur innerhalb von Methoden der Klasse (oder von sog. Freundfunktionen) zugegriffen werden. Die von point definierten Methoden sind öffentlich – benutzbar von jedem, der einen Punkt „*kennt*". Ist p ein Punkt, dann wird beispielsweise durch p.x() die Methode x von p aktiviert und liefert die x-Koordinate von p zurück. p.translate(1,2) verschiebt p um eine Einheit in x- und um zwei Einheiten in y-Richtung, und zwar durch entsprechende Änderung der in p vermerkten Koordinaten.

Allgemein wird für ein Objekt o durch o.m(arg1,arg2,...) die Methode m von o mit Argumenten arg1,arg2,... aktiviert.

Methoden werden im wesentlichen wie Funktionen in imperativen Sprachen übersetzt. Der Unterschied ergibt sich daraus, daß Methoden unmittelbar auf die Merkmale *ihres* Objektes zugreifen können. Wir zeigen nun, wie Methoden durch äquivalente Funktionen implementiert werden können. Die so entstehenden Funktionen können – wie im Kapitel über die Übersetzung imperativer Sprachen gezeigt – in die Sprache einer abstrakten Maschine oder einer realen Maschine übersetzt werden.

Wenn m eine Methode aus der Klasse C mit dem Prototyp <returntype> m(<Arguments>) ist, dann hat die zu m äquivalente Funktion fm den Prototyp <returntype> fm(C &this,<Arguments>).[6] Die Information über o wird an fm also als zusätzliches neues erstes Argument übergeben. Entsprechend wird ein Aufruf o.m(<args>) übersetzt in fm(o,<args>), und ein Zugriff innerhalb von m auf ein Merkmal k von o wird in fm realisiert durch this.k. Diese Übersetzungsregeln sind in der Tabelle 5.1 zusammengefaßt.

Tabelle 5.1: Übersetzung der Methode m von Klasse C in eine Funktion fm.

	Methode	Funktion
Prototyp	<returntype>m(<args>)	<returntype>fm(C &this,<args>)
Aufruf	o.m(<args>)	fm(o,<args>)
Merkmal-zugriff	k	this.k

Die Methode x aus der Klasse point wird übersetzt in die äquivalente Funktion x__5point mit der Definition double x__5point(point &this) { return this.xc; }. Die Methodenaktivierung p.x() wird übersetzt in x__5point(p). Die Methode translate aus point hat die Funktion translate__5pointdd als Übersetzung:

[6] & kennzeichnet in C++ Referenzen; &this bezeichnet einen „*by reference*" übergebenen formalen Parameter mit Namen this.

```
void translate__5pointdd(point &this, double x_offset,
                         double y_offset) {
  this.xc+= x_offset;  this.yc+= y_offset;
}
```

Definition von `translate__5pointdd`

Als Name für die implementierenden Funktionen wird nicht der Methodenname selbst verwendet. Statt dessen erhält der Methodenname eine Erweiterung, in der die zugehörige Klasse und eventuell die Parametertypen kodiert werden. Die Kodierung der Klasse in den Funktionsnamen ist erforderlich, weil verschiedene Klassen Methoden mit demselben Namen enthalten können – jede Klasse besitzt ihren eigenen Namensraum: Es gibt keine Konflikte *zwischen* diesen Namensräumen. Der Namensraum für die Funktionen ist demgegenüber global: Im ganzen Programm darf es keine verschiedenen Funktionen mit demselben Namen geben. Durch Kodierung des Klassennamens im Funktionsnamen sichert man die Eindeutigkeit der erzeugten Funktionsnamen.[7] In `C++` dürfen darüber hinaus sogar verschiedene Methoden innerhalb derselben Klasse denselben Namen besitzen, sofern sie durch ihre Parametertypen unterschieden werden können. Für die Implementierung von `C++` werden deshalb auch die Parametertypen im Namen kodiert. So hat der Name `translate__5pointdd` die folgenden fünf Bestandteile:

1. `translate` – Methodenname
2. `__` – Trenner
3. `5point` – Kodierung des Methodennamens
   Die führende Zahl gibt an, wie viele der folgenden Zeichen zu dieser Angabe gehören. Allen Namen, die unmittelbar aus dem Programm übernommen werden, wird ihre Länge vorangestellt.
4. `d` – Kodierung für `double` (Typ des ersten Parameters)
5. `d` – Kodierung für `double` (Typ des zweiten Parameters)

5.3 Schemata zur Übersetzung von Vererbung

Vererbung ist das wichtigste von objektorientierten Sprachen eingeführte Konzept. Mehrfache Vererbung ist darüber hinaus eine Herausforderung: in erster Linie in bezug auf einen sauberen Sprachentwurf, in zweiter Linie in bezug auf eine effiziente Implementierung. Wir werden uns in diesem Abschnitt eingehend damit beschäftigen.

Beerbt `B` direkt oder indirekt `A`, dann ist `A` eine Oberklasse von `B` und an fast allen Stellen, an denen `A`-Objekte benutzt werden können, dürfen auch `B`-Objekte benutzt werden. Aus Effizienzgründen setzt der Übersetzer einen bestimmten Aufbau von Objekten eines Typs voraus. Um die Verwendung von `B`-Objekten

[7] Die Eindeutigkeit wird durch das vorgestellte (in `C++` verwendete) Kodierungsschema nicht ganz erreicht: `m__5a` in Klasse `a` und `m` in Klasse `a__1a` führen etwa zum selben Funktionsnamen. Das Problem ist leicht zu beseitigen: Man muß nur als Trenner zwischen Methodenname und Erweiterung ein in Namen ungültiges Zeichen verwenden.

als `A`-Objekte zu ermöglichen, muß der Übersetzer deshalb in der Lage sein, effizient eine **A-Sicht** auf `B`-Objekte zu erzeugen. Unter dieser Sicht müssen die vom Übersetzer gemachten Voraussetzungen für den Aufbau von `A`-Objekten erfüllt sein.

Die Dynamische-Bindungs-Regel (s.S. 179) fordert u.a., daß eine Methode eines `A`-Objektes `o`, die in einer abgeleiteten Klasse `B` überschrieben wird, dynamisch gebunden werden muß, wenn der Übersetzer den eigentlichen Typ von `o` nicht ermitteln kann. Hat nämlich `o` den eigentlichen Typ `B`, dann muß die in `B` und nicht die in `A` eingeführte Methode verwendet werden. Für solche Fälle muß der Übersetzer in der Lage sein, Vorbereitungen dafür zu treffen, daß zur Programmlaufzeit die von der zu aktivierenden Methode erwartete Sicht, etwa die ursprüngliche `B`-Sicht, effizient aus der `A`-Sicht erzeugt werden kann.

Wie bereits erwähnt unterstützen viele, insbesondere ältere objektorientierte Sprachen ausschließlich einfache Vererbung. Im nächsten Abschnitt beschäftigen wir uns mit einem geeigneten Schema für die Übersetzung dieses Konzeptes. Im übernächsten Abschnitt wenden wir uns dann dem wesentlich komplexeren Problem der Übersetzung mehrfacher Vererbung zu.

5.3.1 Ein Übersetzungsschema für einfache Vererbung

In einer Sprache mit einfacher Vererbung kann jede Klasse höchstens eine Klasse beerben – sie hat höchstens eine direkte Oberklasse. Die Vererbungshierarchien solcher Sprachen sind Bäume oder Wälder.

Die in diesem Abschnitt skizzierten Mechanismen zur Übersetzung einfacher Vererbung lassen sich verhältnismäßig einfach auch direkt in imperativen Sprachen wie etwa `C` nutzen. Dies ermöglicht, objektorientierte Strukturierungsarten einschließlich der einfachen Vererbung auch in imperativen Programmen zu benutzen.

Wir beginnen mit einem Beispiel: Der folgende Programmcode beschreibt die Klassen `polyline` und, davon abgeleitet, `rectangle`.

```
#include "graphical_object.h" /* "graphical_object" */
#include "list.h"             /* Listen */
#include "point.h"            /* Punkte */

class polyline : public graphical_object {
  List<point> points;
public:
  void translate(double x_offset, double y_offset);
  virtual void scale(double factor);
  virtual double length(void);
};
```

Die Klasse `polyline`

```
#include "polyline.h"

class rectangle : public polyline {
  double side1_length, side2_length;
public:
  rectangle(double s1_len, double s2_len, double x_angle = 0);
  void scale(double factor);
  void length(void);
};
```

Die Klasse `rectangle`

Diese Definition von `rectangle` geht davon aus, daß die beiden Seitenlängen eines Rechtecks nicht erst bei Bedarf aus den Eckpunkten ermittelt, sondern im Rechteck abgespeichert werden sollen. Dies ermöglicht einerseits eine effiziente Redefinition von `length`, macht andererseits aber auch eine Redefinition von `scale` notwendig, denn die Seitenlängen werden durch Skalierung verändert. Da eine Verschiebung den zusätzlichen Zustand nicht verändert, kann `translate` von `polyline` übernommen werden.

Der Gesamtzustand eines Objektes der Klasse `rectangle` ergibt sich aus dem Zustand eines Objektes der Klasse `polyline` sowie dem zusätzlich in `rectangle` definierten Zustand, in unserem Fall den Attributen `side1_length` und `side2_length`. Der allgemeine Fall ist in Abb. 5.2 veranschaulicht. Der Gesamtzustand von Objekten einer erbenden Klasse enthält die Attribute von Objekten der Oberklasse und die neu in der erbenden Klasse definierten Attribute.

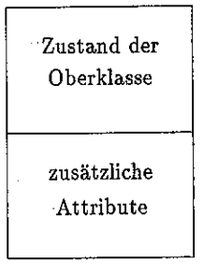

Abb. 5.2: Gesamtzustand einer erbenden Klasse

Wir müssen noch erklären, wie der Übersetzer die dynamische Bindung effizient realisieren kann. In unserem Beispiel integriert die Klasse `rectangle` aus Effizienzgründen die beiden Rechteck-Seitenlängen unmittelbar in ihren Objekten. Als Folge davon darf für die Skalierung eines Rechtecks nicht die Methode `scale` von `polyline` verwendet werden – sie weiß nichts von den zusätzlichen Attributen, die bei der Skalierung mitgeändert werden müssen –, sondern es muß

die von `rectangle` definierte Methode verwendet werden.[8] Nun darf ein Objekt aus der Klasse `rectangle` überall als Argument an eine Funktion/Methode übergeben werden, wenn dort ein Objekt aus einer Oberklasse zulässig ist. Insbesondere gilt dies etwa für die Funktion `zoom`, die ein graphisches Objekt zunächst so verschiebt, daß der Punkt `center` in den Nullpunkt zu liegen kommt, und anschließend das Ergebnis mit dem Wert `zoom_factor` skaliert.

```
void zoom(graphical_object &obj, double zoom_factor,
          point &center){
  obj.translate(-center.x,-center.y);  // "center" to "(0,0)"
  obj.scale(zoom_factor);              // scale
}
```

Zoom: Zentrieren und Skalieren

In diesem Fall muß im Rumpf von `zoom` die Skalierungsfunktion von `rectangle` und nicht die von `polyline` oder gar von `graphical_object` aufgerufen werden. Andererseits kann `zoom` bereits übersetzt und in einer Bibliothek abgelegt worden sein, bevor die Klasse `rectangle` überhaupt definiert wurde. Der Übersetzer kennt also die im Rumpf von `zoom` zu aktivierende Methode bei der Übersetzung von `zoom` unter Umständen noch überhaupt nicht. Daraus ergibt sich, daß er die Bindung von `scale` an eine konkrete Methode während der Übersetzung von `zoom` nicht durchführen kann. `scale` muß innerhalb von `zoom` dynamisch an eine Methode gebunden werden.

Zur effizienten Handhabung dieser dynamischen Bindung können Übersetzer folgendes Schema benutzen: Pro Klasse legt der Übersetzer eine Methodentabelle an, in der alle in der Klasse definierten Methoden, die möglicherweise dynamisch gebunden werden müssen, zusammengefaßt sind. Für `C++` heißen diese Methodentabellen aus naheliegenden Gründen **virtual function tables**. Sie enthalten Einträge für alle als `virtual` definierten Methoden einer Klasse bzw. deren Oberklassen. Jedes Objekt einer Klasse enthält als erste Komponente einen Zeiger auf die zu der Klasse gehörende Methodentabelle. Der Übersetzer bindet Methodennamen an Indizes innerhalb der Methodentabelle. Zum Aufruf der Methode aktiviert er die unter dem zugehörigen Index in der Methodentabelle abgelegte Funktion. Die Methodentabelle einer erbenden Klasse wird folgendermaßen erstellt: Wir beginnen mit einer Kopie der Methodentabelle der beerbten Klasse; in dieser Kopie werden redefinierte Methoden durch die neue Definition überschrieben. Anschließend werden die erstmals neu eingeführten Methoden an die Tabelle angehängt. Hierdurch wird sichergestellt, daß Methodennamen, die bereits in der beerbten Klasse definiert waren, in der neuen Klasse denselben Index zugeordnet bekommen.

Ist `B` eine Klasse und `A` eine Oberklasse von `B`, dann besteht eine `A`-Sicht auf ein Objekt `b` von `B` aus einem Anfangsabschnitt von `b` und einem Anfangsabschnitt der von `b` referenzierten Methodentabelle. Im zur `A`-Sicht gehörenden Anfangs-

[8] Wir sehen hier an einem konkreten Beispiel, weshalb die Methodenauswahlregel (s.S. 179) formuliert wurde.

abschnitt von `b` liegen genau der Verweis auf die Methodentabelle und die von `A` geerbten Attribute. Der zur `A`-Sicht gehörende Abschnitt der Methodentabelle deckt die Indizes für die in `A` oder einer Oberklasse eingeführten Methoden ab. Alle Sichten auf `b` werden einheitlich durch einen Zeiger auf `b` repräsentiert. Der Übergang zwischen verschiedenen Sichten wird damit trivial.

Wir verdeutlichen das Schema für unser Beispiel in Tabelle 5.2 und Abb. 5.3.

Tabelle 5.2: Methodentabellen für verschiedene Teilklassen graphischer Objekte

`graphical object`	`polyline`	`rectangle`
`translate_GO`	`translate_PL`	`translate_PL`
`scale_GO`	`scale_PL`	`scale_RA`
	`length_PL`	`length_RA`

Tabelle 5.2 zeigt die Methodentabellen für `graphical object`, `polyline` und `rectangle`. Die Methodentabelle für `polyline` geht aus der von `graphical object` hervor, indem zunächst die in `polyline` redefinierten Methoden `translate_PL` und `scale_PL` die entsprechenden Methoden von `graphical object` ersetzen und anschließend die neu definierte (virtuelle) Methode `length_PL` angehängt wird. Die Methodentabelle von `rectangle` ihrerseits ergibt sich aus der von `polygon`, indem die in `rectangle` redefinierten Methoden die entsprechenden Methoden von `Polyline` ersetzen. Die nicht redefinierte Methode `translate_PL` wird übernommen. Der Übersetzer bindet `translate` an den Index **0**, `scale` an den Index **1** und `length` an den Index **2**.

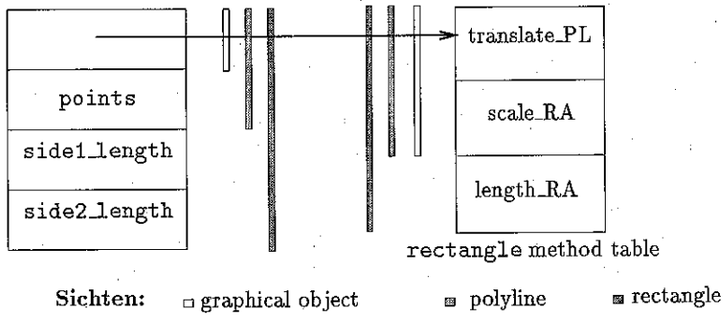

Abb. 5.3: Repräsentation von `rectangle`-Objekten

Abb. 5.3 zeigt die Repräsentation von `rectangle`-Objekten. Jedes derartige Objekt enthält neben seinem eigentlichen Zustand einen Verweis auf die einmal abgelegte Methodentabelle der Klasse `rectangle`.

Die Realisierung einfacher Vererbung mit dynamischer Bindung ist daher mit einem Speicheroverhead von einem Verweis pro Objekt verbunden. Hinzu kommt pro Klasse die Ablage der zugehörigen Methodentabelle. Durch die dynamische

Bindung erhöht sich die Laufzeit für einen Methodenaufruf um eine Indirektion zum Auffinden der Methodentabelle und eine Indizierung zur Lokalisierung der zu aktivierenden Methode.

5.3.2 Übersetzungsschemata für mehrfache Vererbung

Im letzten Abschnitt wurde gezeigt, daß einfache Vererbung leicht übersetzbar ist. Mehrfache Vererbung stellt demgegenüber eine größere Herausforderung dar, und zwar mehr noch in bezug auf eine saubere Sprachdefinition als auf den Bau eines Übersetzers.

Sprachen mit mehrfacher Vererbung erlauben einer Klasse, nicht nur eine, sondern beliebig viele Klassen zu beerben. Eine Klasse kann also mehrere direkte Oberklassen besitzen. Die sich ergebenen Vererbungshierarchien sind nicht mehr Wälder, sondern gerichtete azyklische Graphen.

Uns ist mehrfache Vererbung bereits bei der Einführung der Klasse `polygon` begegnet (Abb. 5.1, S. 177). Die mehrfache Vererbung bringt dort zum Ausdruck, da"3 Polygone sowohl geschlossene graphische Objekte als auch Linienzüge sind. Immer dann, wenn Objekte zu mehreren Klassen gehören, läßt sich dies durch mehrfache Vererbung[9] in natürlicher Weise modellieren. Mehrfache Vererbung unterstützt darüber hinaus einen Programmierstil, in dem kleine Basisklassen elementare Fähigkeiten implementieren und komplexere Klassen diese durch Beerbung erwerben. Beispiele für Basisklassen sind `persistent` und `queueable`. Die Klasse `persistent` kann Objekte der Klasse im Dateisystem ablegen und von dort laden; die Infrastruktur der Klasse `queueable` erlaubt ihren Objekten, sich in eine Warteschlange einzuhängen. Eine Klasse, die beide Fähigkeiten benötigt, beerbt einfach die beiden Basisklassen. Eine abstrakte Klasse entspricht einer Spezifikation: der Festlegung, da"3 die Objekte der Klasse vorgegebene Mindestmerkmale besitzen müssen. Müssen mehrere dieser Spezifikationen gleichzeitig erfüllt sein, so lä"3t sich dies durch eine Klasse modellieren, die durch Beerben der zugehörigen abstrakten Klassen entsteht. Mit Hilfe mehrfacher Vererbung kann also das „*Und*“[10] mehrere Spezifikationen realisiert werden. Schließlich ist mehrfache Vererbung verwendbar, um eine Spezifikation, gegeben durch eine abstrakte Klasse, und ihre Implementierung, gegeben durch eine konkrete Klasse, zu integrieren. So kann z.B. ein Keller mit beschränkter Grö"3e durch Beerben der abstrakten Klasse `stack` und der konkreten Klasse `array` realisiert werden. Mehrfache Vererbung ist ein sehr vielseitiges und mächtiges Konzept!

Alle Probleme und mögliche Lösungsansätze lassen sich am Fall zweifacher Beerbung veranschaulichen. Im gesamten folgenden Abschnitt gehen wir deshalb von dem Fall aus, daß eine Klasse `C` durch Beerben der beiden Klassen `B1` und `B2` entsteht.

Probleme für die Sprachdefinition und zum Teil den Übersetzerbau ergeben sich vor allem aus:

[9] Eigentlich ist es eine mehrfache Beerbung.
[10] die gleichzeitige Gültigkeit

1. Konflikten und Widersprüchen zwischen `B1` und `B2`.
   Wird etwa in den beiden Basisklassen derselbe Name für Methoden oder Attribute benutzt, kann es bei der gemeinsamen Beerbung zu einem Konflikt kommen.

2. Wiederholter Beerbung.
   Sind etwa `B1` und `B2` direkte oder indirekte Erben von `A`, dann wird `A` von `C` wiederholt beerbt – ein ausgesprochen interessanter Konfliktherd, wie wir sehen werden.

Das Problem 1 ist in erster Linie ein Sprachdefinitionsproblem. Folgende Lösungsansätze, die auch kombiniert werden können, stehen beim Sprachentwurf zur Verfügung:

1. `B1` wird als Haupterblasser festgelegt; Konflikte werden zugunsten von `B1` aufgelöst.
   Diese Möglichkeit wird vor allem von interpretierten objektorientierten Erweiterungen von `Lisp` benutzt. Sie binden Namen typischerweise dynamisch, indem die Klassenhierarchie in einer vorgegebenen Ordnung[11] nach einer geeigneten Definition durchsucht wird; die erste passende Definition gewinnt.
   Der Ansatz ist nicht ganz ungefährlich, da potentielle Widersprüche nicht offensichtlich sein müssen und für den Programmentwickler damit nicht immer deutlich ist, wann obige Konfliktauflösungsstrategie zum Einsatz kommt – u.U. entgegen seiner Intention.

2. Die Sprache ermöglicht Umbenennungen bei der Beerbung und gibt dem Programmierer damit die Möglichkeit, potentielle Konflikte durch expliziten Eingriff aufzulösen. Dieser Ansatz wird etwa von `Eiffel` verwendet.

3. Die Sprache stellt Konstrukte zur Verfügung, mit deren Hilfe beim Zugriff ein Konflikt explizit aufgelöst werden kann.
   Stehen beispielsweise Definitionen in `B1` und `B2` für einen Namen `n` zueinander im Widerspruch, dann kann durch Konstrukte der Form `B1::n` bzw. `B2::n` eindeutig festgelegt werden, daß die Definition von `B1` bzw. `B2` zu verwenden ist. Dieser Ansatz wird etwa von `C++` verwendet.

Der von der Sprache vorgegebene Ansatz hat Auswirkungen auf die Symboltabellenverwaltung und -organisation des Übersetzers. Alle lassen sich jedoch verhältnismäßig einfach implementieren. Wir werden nicht weiter darauf eingehen.

Bezüglich des Problems 2, skizziert in Abb. 5.4, gibt es zwei einander diametral entgegengesetzte Ansätze:
1. Das wiederholte Erbe wird mehrfach instantiiert, siehe Abb. 5.5,
2. das wiederholte Erbe wird ein einziges Mal instantiiert, siehe Abb. 5.6.

Beide Ansätze haben Vor- und Nachteile.

[11] In unserem Fall: `C`, dann `B1`, dann `B2`.

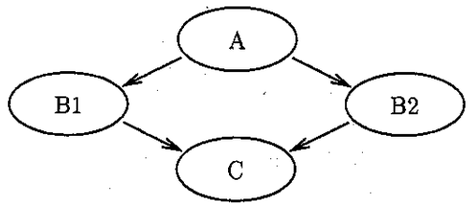

Abb. 5.4: Wiederholte Beerbung

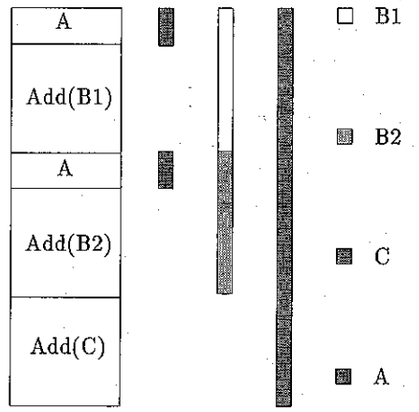

Abb. 5.5: Mehrfache Instantiierung wiederholten Erbes

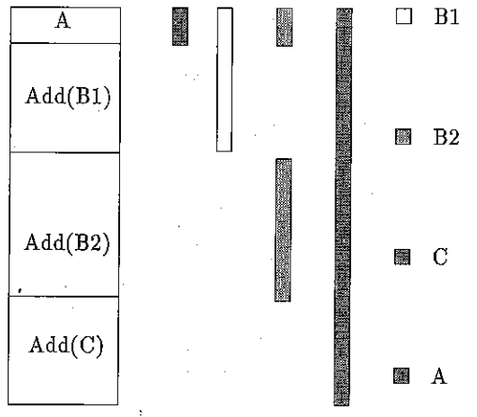

Abb. 5.6: Einmalige Instantiierung wiederholten Erbes

Betrachten wir unsere Klassenbibliothek graphischer Objekte als erstes Beispiel. Wie wir in Abb. 5.1 (s.S. 177) sehen, beerbt die Klasse `polygon` sowohl `closed graphical` als auch `polyline`, da Polygone geschlossene Linienzüge sind. Auf beiden Wegen beerbt `polygon` die Klasse `graphical object`. Dies bedeutet jedoch nicht, daß Polygone zwei unabhängige Objektinstanzen aus `graphical object` als Teilobjekte enthalten sollen, wie dies nach Abb. 5.5 der Fall wäre. Wir wollen dies durch eine Erweiterung unseres Beispiels verdeutlichen. Aus Effizienzgründen wird für viele Visualisierungen von Objektbewegungen nicht das eigentliche Objekt in seiner Bewegung visualisiert, sondern an seiner Stelle eine vereinfachte Form, etwa sein umschließendes Rechteck. Wir legen mit diesem Hintergrund fest, daß jedes graphische Objekt einen Verweis auf sein umschließendes Rechteck enthalten soll. Dies wird ausgedrückt durch Aufnahme eines entsprechenden Attributes in `graphical object`. Wird wiederholtes Erbe mehrfach instantiiert, dann enthalten Polygone zwei verschiedene Verweise auf das umschließende Rechteck, die konsistent gehalten werden müssen. In diesem Beispiel kann es mit etwas Glück gelingen, beide Verweise sogar konstant zu halten: Sie zeigen immer auf dasselbe Rechteck; die Verweise bleiben unverändert, das Rechteck wird gemäß den Operationen auf dem Hauptobjekt verändert. Verloren wären wir, wenn wir statt eines *Verweises* auf das umschließende Rechteck, seine Eckkoordinaten in graphische Objekte integriert hätten. In diesem Fall müßten wir in `polygon` sämtliche von `polyline` geerbten Methoden redefinieren, um sicherzustellen, daß die Angaben in der über `closed graphical` geerbten Kopie von `graphical object` konsistent zu der über `polyline` geerbten Kopie sind. Ferner müßten alle in `polyline` oder Teilklassen definierten Methoden die beiden Kopien konsistent halten – eine unnatürliche Komplikation. Wir sehen: Eine mehrfache Instantiierung des gemeinsamen Erbes ist für unser Beispiel graphischer Objekte unnatürlich und nur bedingt geeignet.

Wir konstruieren nun ein Beispiel, für das die mehrfache Instantiierung genau das gewünschte Ergebnis bringt. Die GNU-C++-Klassenbibliothek enthält zwei Klassen für statistische Auswertungen: `SampleStatistics` und `SampleHistogramm`. `SampleStatistics` enthält unter anderem Dienste zur Bestimmung des Mittelwertes, der Varianz und der Standardabweichung von Meßreihen. `SampleHistogramm` enthält unter anderem Dienste zur Bestimmung von Histogrammen. Nehmen wir an, wir müssen eine Klasse definieren, die für eine Temperaturmeßreihe Mittelwerte und Varianzen ermitteln und für eine Druckmeßreihe Histogramme bestimmen muß. Ein naheliegender Ansatz ist, die beiden Klassen `SampleHistogramm` und `SampleStatistics` zu beerben und ihre Methoden so umzubenennen[12], daß aus dem Namen hervorgeht, ob sie sich auf die Temperatur- oder Druckmeßreihe beziehen. Nun ist `SampleHistogramm` in der `C++`-Klassenbibliothek als Erbe von `SampleStatistics` definiert. Unsere Druck/Temperatur-Klasse beerbt `SampleStatistics` also zweimal. Für dieses Beispiel ist es unbedingt erforderlich, daß das gemeinsame Erbe mehrfach instantiiert wird, da ansonsten die beiden Meßreihen für die statistischen Auswertungen nicht

[12] Umbenennung ist in `C++` nicht möglich, jedoch beispielsweise in `Eiffel`.

auseinandergehalten würden.

Wir sehen: In gewissen Fällen wünschen wir, daß gemeinsames Erbe mehrfach instantiiert wird, während wir in anderen Fällen eine einzige Instanz des gemeinsamen Erbes benötigen. Es kann in manchen Fällen sogar wünschenswert sein, für einige Merkmale des wiederholten Erbes eine einzige Instanz zu haben, während andere Merkmale mehrfach instantiiert werden. `Eiffel` bietet diese Flexibilität.

Wir betrachten zunächst Übersetzungsschemata, die ausschließlich eine Mehrfachinstantiierung wiederholten Erbes erlauben. Diese Schemata sind einfacher und erzeugen effizienteren Code als Schemata, die darüber hinaus eine einmalige Instantiierung gemeinsamen Erbes erlauben.

Übersetzung unabhängiger mehrfacher Beerbung

Bei unabhängiger mehrfacher Beerbung ist das Erbe aus den verschiedenen direkten Oberklassen unabhängig voneinander. Dementsprechend enthalten Objekte der erbenden Klasse `C` vollständige Kopien der beerbten Klassen `B1` und `B2`, wie in Abb. 5.7 veranschaulicht. Die Sprache `C++` hat diese Vorstellung von mehrfacher Vererbung.

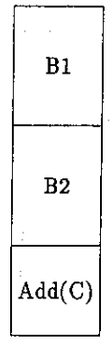

Abb. 5.7: Objektaufbau bei unabhängiger mehrfacher Beerbung (Programmsicht)

Wiederholtes Erbe tritt dadurch in mehreren Instanzen auf – wie in Abb. 5.5 skizziert – und führt zu Konflikten und Mehrdeutigkeiten:

- bei Referenz der mehrfach instantiierten Merkmale zwecks Zugriffs, Aufrufs oder Überschreibens;
- bei der Erzeugung einer `A`-Sicht auf `C`-Objekte, da `C`-Objekte mehrere `A`-Teilobjekte enthalten.

Sichtbarkeitsregeln können in gewissen Fällen helfen, die Schwierigkeiten zu vermeiden. So erlaubt `C++` etwa einer Klasse, ihr eigenes Erbe vor ihren Erben zu verbergen: `B1` kann z.B. `A` privat beerben,[13] ohne daß dies in `C` sichtbar wird; es

[13] In diesem Fall wird `B1` keine Teilklasse von `A`.

entstehen dann in C keine Mehrdeutigkeiten wegen der mehrfachen Instantiierung von A.[14] Wo Sichtbarkeitsregeln nicht ausreichen, die Mehrdeutigkeiten zu beseitigen, werden weitere Sprachkonstrukte notwendig: In C++ sind dies etwa der Qualifikationsoperator ::, benutzt in der Form B1::k, um das Merkmal k der Klasse B1 zu bezeichnen, und Typkonversionen, mit denen etwa ein C-Objekt explizit in ein B1-Objekt konvertiert werden kann.

Wir versuchen nun, das Übersetzungsschema für einfache Beerbung auf den Fall unabhängiger zweifacher Beerbung zu erweitern. Zur effizienten Implementierung der dynamischen Methodenbindung haben wir (s. Abb. 5.3, S. 188) in jedes Objekt als erste Komponente einen Zeiger auf die Methodentabelle seiner Klasse integriert. Gleiches versuchen wir auch jetzt: Jedes Objekt einer Klasse enthält als erstes Element einen Verweis auf die Methodentabelle der Klasse.

Zweite Aufgabe: Der Übersetzer muß für jede Oberklasse B von C eine B-Sicht auf C-Objekte erzeugen können. Da das B1-Teilobjekt am Anfang des C-Objektes liegt, können wir für B1 den Ansatz für einfache Vererbung benutzen: Die B1-Sicht auf C-Objekte ist ein Anfangsabschnitt der C-Sicht.[15] Als B2-Sicht können wir leider keinen Anfangsabschnitt der C-Sicht verwenden, denn eine B2-Sicht eines Objektes muß als erste Komponente einen Zeiger auf eine Methodentabelle mit einem von B2 vorgegebenen Aufbau und anschließend die Attributwerte von B2 enthalten. Dies führt zu folgendem Ansatz: Wir integrieren in C-Objekte vor den B2-Attributwerten einen Zeiger auf eine weitere Methodentabelle. Diese Methodentabelle ergibt sich aus einer Kopie der B2-Methodentabelle durch Überschreiben von Methoden, die in C redefiniert wurden. Repräsentiert wird die B2-Sicht durch eine sog. B2-Referenz: Sie zeigt auf den Anfang des B2-Teilobjektes. Dessen erste Komponente ist der Zeiger auf die Methodentabelle des B2-Teilobjektes. Für jede Instanz einer Oberklasse A kennt der Übersetzer den Offset des entsprechenden Teilobjektes innerhalb des C- Objektes. Er erzeugt die entsprechende A-Referenz, indem er diesen Offset zur C-Referenz addiert. Wir erhalten den Aufbau aus Abb. 5.8.

Dritte Aufgabe: C-Methoden, d.h. in C definierte Methoden, erwarten eine C-Sicht auf die Objekte, für die sie aktiviert werden. Überschreibt eine solche Methode einen Eintrag in der Methodentabelle für einen Obertyp A,[16] dann ist bei der Methodenaktivierung nur eine A-Sicht verfügbar. Es muß zur Laufzeit möglich sein, daraus die geforderte C-Sicht zu berechnen. Für den Übersetzer, der jetzt sowohl C wie A kennt, wäre dies ein leichtes: Sichten werden durch die entsprechenden Referenzen repräsentiert; für jedes C-Objekt ist aber die Differenz d der A-Referenz und der C-Referenz eine Konstante, nämlich der Offset des A-Teilobjektes innerhalb des C-Objektes. Die C-Referenz ergibt sich also aus der A-Referenz durch Subtraktion von d. Sowohl bei der Übersetzung des Methodenaufrufs als auch zur Laufzeit ist C aber unbekannt. Aus diesem Grund legt der

[14] In C++ werden Sichtbarkeitsregeln jedoch explizit nicht benutzt, um Mehrdeutigkeiten aufzulösen. Obwohl C das Erbe A in B1 nicht sieht, kommt es durch die beiden Instanzen von A zu Mehrdeutigkeiten – eine zweifelhafte Entscheidung beim C++-Entwurf.

[15] Sowohl bezüglich der Objektkomponenten als auch der Methodentabelle.

[16] Genauer: eine Instanz eines Obertyps; C kann mehrere Instanzen von A enthalten.

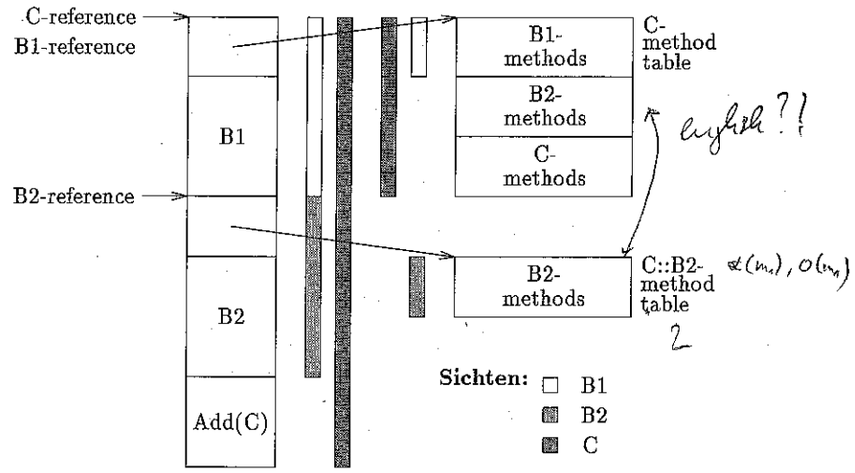

Abb. 5.8: Objektaufbau bei unabhängiger mehrfacher Beerbung (Implementierungssicht)

Übersetzer den für die Bestimmung der erforderlichen Sicht notwendigen Offset neben dem Methodenzeiger mit in der Methodentabelle ab. Jeder Eintrag in einer Methodentabelle hat also zwei Komponenten: die zu aktivierende Methode und den Offset, um aus der bei der Methodenaktivierung verfügbaren Sicht die von der Methode erwartete Sicht zu erzeugen – beide Sichten repräsentiert durch die jeweiligen Referenzen.

Der oben skizzierte Ansatz hat einen kleinen aber wesentlichen Schönheitsfehler: Die beiden für die Klasse C angelegten Methodentabellen enthalten zwei (modifizierte) Kopien der B2-Methodentabelle. Folge: Der Speicherbedarf für die Ablage der Methodentabellen einer Klasse C kann exponentiell mit der Komplexität der Definition von C[17] wachsen!

Um exponentielles Wachstum zu vermeiden, müssen wir mit einer einzigen Kopie der B2-Methodentabelle auskommen. Wir erreichen dies, indem wir die C-Methodentabelle *verteilt* ablegen, wie in Abb. 5.9 veranschaulicht. Diese Abbildung bringt ein interessantes Detail nicht voll zum Ausdruck: Selbstverständlich sind auch die Kopien der Methodentabellen für B1 und B2 verteilt abgelegt.

Als nächstes wenden wir uns einer halbformalen Beschreibung dieses Übersetzungsschemas zu. Wir werden uns dabei das Leben vereinfachen, indem wir davon ausgehen, daß alle Mehrdeutigkeiten bereits aufgelöst sind. Die Auflösung von Mehrdeutigkeiten ist keinesfalls trivial, aber die hierzu notwendigen Techniken sind an anderer Stelle, in den Kapiteln über syntaktische und semanti-

[17] Unter der Komplexität einer Klassendefinition verstehen wir die Größe der aufgefalteten Definition der Klasse. Die aufgefaltete Definition einer Klasse ergibt sich aus ihrer Definition, indem die beerbten Klassen durch ihre aufgefalteten Definitionen ersetzt werden.