Diplomarbeit

Übersetzung und Optimierung objektorientierter Programmiersprachen unter besonderer Berücksichtigung eingebetteter Systeme

Frank Jagla

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

17. Mai 2000

Betreuer:
Dipl.-Inform. Steven Bashford
Prof. Dr. Peter Marwedel

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Eingebette Systeme sind ein bedeutender europäischer Wirtschaftsfaktor mit einem Marktwachstum, das dasjenige des klassischen PC-Marktes noch übertreffen wird. Da dieser Bereich auch nicht die durch die amerikanischen Chipgiganten wie Intel oder Motorola dominiert wird, bietet sich hier die Chance für eine eigenständige europäische Hard- und Softwareindustrie. Beispiele sind ARM und Siemens, die extrem konkurrenzfähige Designs etablieren konnten.

An eingebettete Systeme werden je nach Einsatzbereich hohe Anforderungen an Sicherheit und Zuverlässigkeit gestellt, die im Entwicklungsprozess von vorneherein zu berücksichtigen sind. Hierbei gilt wie überall in der Informatik, daß die Methoden zur Herstellung komplexer Software bei weitem nicht so fortgeschritten sind wie die für die Hardware. Eigentlich ist die Lage bei eingebetteten Systemen sogar noch prekärer, da hier mehrheitlich noch in Assembler (gefolgt von C) codiert wird. Assembler erfüllt zweifellos nicht die Anforderungen der modernen Softwaretechnologie.

In der allgemeinen Informatik hat sich jedoch schon länger die Auffassung durchgesetzt, daß zur Entwicklung grosser und komplexer Softwaresysteme die *objektorientierte Programmierung* besonders geeignet ist. Diese zeichnet sich aus durch problemadäquate Abstraktionsmechanismen und durch die Möglichkeit, einmal erzielte Software-Lösungen in Bausteinbibliotheken zur späteren Wiederverwendung ablegen zu können.

Diese Arbeit soll dazu beitragen, daß zur Programmierung eingebetteter Systeme zunehmend auch höhere Programmiersprachen eingesetzt werden können.

Meine persönliche Motivation begründet sich in meiner Faszination für portable digitale Assistenten wie den Apple Newton PDA und die EPOC32-Maschinen von Psion/Symbian und in meinem starken Interesse an objektorientierten Programmiersprachen.

1.2 Ziele der Arbeit

Das übergeordnete Ziel wurde bereits erwähnt: Wie kann objektorientierte Technologie für die Programmierung eingebetteter Systeme nutzbar gemacht werden? Welche Anforderungen an eingebettete Systeme sind zu berücksichtigen? Wie können speziell objektorientierte Sprachen diese Anforderungen erfüllen?

Im einzelnen werden dazu folgende Themen bearbeitet:

- **Kapitel 2**. Es werden Anforderungen an eingebettete Systeme ermittelt. Diese sind Sicherheit, Echtzeitbetrieb und Berücksichtigung beschränkter Zeit- und Platz-Ressourcen. Es werden konkrete Anforderungen an Programmiersprachen abgeleitet. Ziel ist die Erstellung hochverläßlicher Software.
- **Kapitel 3**. Es wird kompakt und dem aktuellen Stand der Forschung entsprechend herausgestellt, was die wesentlichen Merkmale und Vorteile der objektorientierten Programmierung sind. Insbesondere werden auf den wichtigen Unterschied zwischen Klasse und Typ und die Nachteile der Mehrfachvererbung hingewiesen.
- **Kapitel 4**. Es wird ein Gestaltungsraum für das Design konkreter objektorientierter Programmiersprachen vorgestellt. Es soll ein Bewußtsein für Möglichkeiten, Unterschiede, Gemeinsamkeiten geweckt werden. Verschiedene Optionen werden aufgezeigt und bezüglich Ausdruckskraft und Sicherheit bewertet. Insbesondere wird auf leistungsfähige statische Typsysteme hingewiesen, da diese unbedingte Voraussetzung für die Erstellung sicherer Software sind.
- **Kapitel 5**. Es werden grundlegende Techniken zur Übersetzung objektorientierter Programmiersprachen vorgestellt. Die bestehende Verwandtschaft zu imperativen Programmiersprachen wird ausgenutzt, in dem die objektorientierten Merkmale auf bekannte Datenstrukturen wie Verbund, Reihung und Zeiger reduziert werden. Nach dieser Reduktion kann der entstandene Code mit Standard- und maschinennahen Techniken weiter optimiert werden, die ursprünglich für imperative Sprachen entwickelt worden sind.
- **Kapitel 6**. Es werden high-level Optimierungen diskutiert, die den Zeit-Bedarf für die Methodensuche und den Platz-Bedarf für die Methodentabellen verkleinern können. Tabellenorientierte Verfahren haben deterministisch konstante Suchzeiten. Es werden Möglichkeiten zur statischen Bindung von Nachrichtenausdrücken aufgezeigt.
- **Kapitel 7**. Es wird aufgezeigt, wie der Zeit-Bedarf für die Speicherverwaltung von Objekten durch statische und beschränkt polymorph statische Allokation optimiert werden kann. Als Nebeneffekt können Nachrichtenausdrücke statisch gebunden werden.
- **Kapitel 8**. Es werden Zwischendarstellungen für objektorientierte Sprachen vorgestellt.

- **Kapitel 9**. In der Konklusion wird insbesondere eine Empfehlung für das Design einer hybriden objektorientierten Programmiersprache ausgesprochen.

Folgende Themen werden in der Arbeit gar nicht oder nur am Rande erwähnt:

- Ablaufplanung und Ressourcenzuteilung für objektorientierte Programme in Echtzeit-Betriebssystemen,
- die für objektorientierte Programmiersprachen wichtige effiziente und realzeitfähige automatische Speicherbereinigung (Garbage Collection).

Der ideale Leser dieser Diplomarbeit kennt sich aus mit der Übersetzung prozeduraler Sprachen und beherrscht die Grundlagen der objektorientierten Programmierung.

Kapitel 2

Eingebettete Systeme

In diesem Kapitel wird zunächst eine Charakterisierung eingebetteter Systeme versucht. Im Anschluss daran werden Anforderungen aufgelistet, denen eingebettete Systeme entsprechen müssen. Daraus werden Konsequenzen für die Software-Entwicklung für eingebettete Systeme abgeleitet. Insbesondere wird auch auf den Ansatz der reaktiven Systeme und der korrespondierenden synchronen Sprachen eingegangen.

2.1 Einführung

Der Begriff *Eingebettetes System* bezeichnet ein weites Spektrum an informationsverarbeitenden Computersystemen, denen allen gemeinsam ihre enge Einbindung in eine physikalische Umgebung ist. Der Leistungsbereich reicht dabei von einfachen Mikrokontrollern bis hin zu ausgewachsenen Rechnern mit RISC-Prozessor und mehreren Megabytes Hauptspeicher. Sie finden sich in Chipkarten, Waschmaschinen, Toastern ebenso wie in komplexen Anwendungen der Luft- und Raumfahrt, der Kfz-Elektronik oder zur Steuerung komplexer Produktionsanlagen.

Eingebettete Systeme sind Kombinationen aus Hard- und Softwarekomponenten, die über Sensoren und Aktuatoren mit einer realen Umgebung interagieren. Da die Regelung bzw. Steuerung einer physikalischen Umgebung im Vordergrund steht, werden eingebettete Systeme auch als *reaktive* Systeme bezeichnet, im starken Gegensatz zu den uns geläufigeren *interaktiven* Systemen wie den heimischen PC.

Die neu entstehenden Geräte der mobilen Telekommunikation, die Persönlichen Digitalen Assistenten (PDAs) oder die Navigationssysteme für AutofahrerInnen bilden eine Art Zwischenform, da sie zwar eine deutliche Benutzerorientierung aufweisen, aber aufgrund der speziellen Hardwarekomponenten und der hard- und softwareseitigen Beschränkungen zu den eingebetteten Systemen gezählt werden können.

2.2 Charakterisierung

Eingebettete Systeme können durch folgende Eigenschaften beschrieben werden:

Anwendungsbereich: Messen, Steuern, Regeln; digitale Signalverarbeitung, kontinuierliche Regelung, diskrete Steuerungen, Datenaufnahme- und Speicherung;

Informationsverarbeitung: datenflußorientiert, zustandorientiert, regelbasiert (z. B. Fuzzy-Logik), transaktionsbasiert;

Verläßlichkeit: viele eingebettete Systeme werden in sicherheitskritischen Bereichen eingesetzt, eine Fehlfunktion hat im schlimmsten Fall den Verlust von Menschenleben zur Folge;

Echtzeitbetrieb: Rechtzeitigkeit der Systemanwort, Einhaltung von Zeitschranken;

Langlebigkeit: kontinuierlicher Betrieb über lange Zeiträume;

Ressourcenbeschränkung: knappe Ressourcen sind beispielsweise Zeit, Speicher, Energie, Chipfläche;

Benutzerorientierung: meistens für den Benutzer vollständig unsichtbar, aber auch einfache Benutzerinteraktion bis hin zum vollen GUI;

Hardware: ASICs, Prozessorcores, Mikrokontroller, aber auch sogenannte Industrie-PCs;

Betriebssystem: Echtzeit-Betriebssysteme wie VxWorks, OS-9, QNX, Lynx, RTOS-UH, aber auch MS-DOS, Windows NT, Unix, RT-Linux. Der Trend geht zu kleinen, modularen und konfigurierbaren Mikrokernen;

Programmierung: Assembler oder C, PEARL, seltener objektorientierte oder funktionale Sprachen.

Das wesentliche Merkmal eingebetteter Systeme ist ihre enge Einbindung in eine physikalische Umgebung. Diese enge Einbindung kann sich auf mehreren Ebenen auswirken. Einerseits erfüllen eingebettete Systeme in Bezug auf die Umgebung eine bestimmte Aufgabe. Diese ist über einen langen Zeitraum *verläßlich* und *rechtzeitig* zu erledigen. Andererseits stehen zur Realisierung eingebetteter Systeme in der Regel hard- und softwareseitig nur *beschränkte Ressourcen* zur Verfügung.

Diese hohen, sich oft auch widersprechenden Anforderungen machen die Entwicklung eingebetteter Systeme zu einer anspruchsvollen Aufgabe, wobei festgestellt werden kann, daß inzwischen die besondere Herausforderung sehr viel mehr in der Entwicklung *hochverläßlicher Software* besteht, und weniger in der Hardwareentwicklung.

Insbesondere die nicht nachrichtentechnisch motivierten Anwendungen mit zustandorientierter oder transaktionsbasierter Informationsverarbeitung sollten einer Programmierung in höheren Sprachen zugänglich gemacht werden. In der allgemeinen Informatik hat sich schon länger die Auffassung durchgesetzt, daß zur Entwicklung grosser und komplexer Softwaresysteme die *objektorientierte Programmierung* besonders geeignet ist. Diese zeichnet sich aus durch problemadäquate Abstraktionsmechanismen und durch die Möglichkeit, einmal erzielte Software-Lösungen in Bausteinbibliotheken zur späteren Wiederverwendung ablegen zu können. In [OSN99] wird allerdings argumentiert, daß gegenwärtig die Anwendung objektorientierter Technologie nur für eine bestimmte Klasse eingebetteter Systeme sinnvoll ist, die sich insbesondere durch eine hohe Komplexität der Steuerungsaufgabe und durch ausreichende Ressourcen charakterisieren lassen.

Es ist ein wesentliches Ziel dieser Arbeit, den Einsatz objektorientierter Sprachen zur Programmierung eingebetteter Systeme zu befördern, indem aufgezeigt wird, wie die besonderen Merkmale dieser Sprachklasse übersetzt und reduziert werden können auf bekannte und ausführlich untersuchte Datenstrukturen wie den Verbund, die Reihung und den Zeiger. Im Anschluss können hochoptimierende und retargierbare Compiler, wie sie beispielsweise am Lehrstuhl XII entwickelt werden, qualitativ hohen, effizienten und die speziellen Merkmale der Zielarchitektur ausnutzenden Code generieren.

2.3 Anforderungen an eingebettete Systeme

Da vom einwandfreien Funktionieren eingebetteter Systeme Menschenleben direkt oder indirekt abhängig sein können, müssen an eingebettete Systeme hohe Anforderungen bezüglich der Verläßlichkeit und der Echtzeitfähigkeit gestellt werden. Weitere Anforderungen ergeben sich aus

2.3.1 Verläßlichkeit

Nach Halang et.al. [HFL+98] stellen *hochverläßliche programmierbare elektronische Systeme* für sicherheitskritische Steuer- und Regelanwendungen ein ganz neues Gebiet dar, das erst am Anfang seiner wissenschaftlichen Bearbeitung steht. Die Autoren schreiben:

> "Mehr und mehr dringen die ureigenen, mit Rechnerprogrammen verbundenen Sicherheitsprobleme auch in das Bewußsein der Öffentlichkeit. Sicherheitstechniken klassischer Ingenieursdisziplinen, die sich vorwiegend mit zufällig verteilten Ausfällen technischer (Hardware-) Systeme befassen, sind auf programmintensive Systeme nicht übertragbar, denn Programme unterliegen weder Verschleißerscheinungen noch Umwelteinflüssen, sind dafür aber – mit Ausnahme trivialer Fälle – immer fehlerbehaftet. Alle Fehler werden bereits beim Entwurf gemacht, sind also systematischer Natur, weshalb ihre Ursachen mithin immer latent vorhanden

sind. Programme weisen eine extrem große und nicht mehr handhabbare Zahl diskreter Systemzustände auf, die zudem selten reguläre Strukturen aufweisen."

Der neue Begriff *Verläßlichkeit (dependability)* faßt alle Eigenschaften eines Systems zusammen, welche sicherstellen, daß sich ein System wie erwartet verhält.

Verläßlichkeit meint im einzelnen *Zuverlässigkeit (reliability)*, *Sicherheit (safety)* und *Geschütztheit (security)* [BW97, Mar99]:

Zuverlässigkeit: *Continuity of service.* Berücksichtigt insbesondere die reinen Hardwarefehler, aber auch Fehler in der Software. Fehlertoleranz.

Sicherheit: *Non-occurence of catastrophic failures.* Gewährleistet den Schutz von Menschenleben. Beispielsweise darf ein Air-Bag auch dann nicht versehentlich ausgelöst werden, wenn ein Prozessor einen Hardware-Fehler hat.

Geschütztheit: *Protection against intentional faults.* Schutz gegen böswillige Angreifer, Hacker, Sabotage.

Sicherheitsklasse	Gefährdung
SIL 4 (Safety-critical)	catastrophic loss of life
SIL 3 (Safety-critical)	loss of multiple lives
SIL 2 (High-Integrity)	potential loss of one live
SIL 1 (High-Integrity)	injuries
SIL 0 (Not safety-related)	the rest

Tabelle 2.1: Sicherheitsklassen nach IEC 61508

Die internationale Norm IEC 61508 (VDE 0801) "Funktionale Sicherheit – Sicherheitssysteme" behandelt in Teil 1 allgemeine Anforderungen und in Teil 3 Anforderungen an Software. Es werden fünf *Sicherheitsklassen (Safety Integrity Levels SIL)* definiert[1], die sich an der Gefährdung von Menschenleben orientieren. Die Norm gibt insbesondere auch für jede Sicherheitsklasse methodische Empfehlungen bzgl. der geeigneten Entwicklungsstrategien. Anforderungen an Programmiersprachen für die einzelnen Sicherheitsklassen werden in Abschnitt 2.4 vorgestellt.

2.3.2 Echtzeitbetrieb

Der Echtzeitbetrieb wird in der DIN 44 300 wie folgt definiert:

[1] siehe auch http://www.oakcomp.demon.co.uk/Levels.html

> Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, daß die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.

Nach Halang et.al. [HFL+98] unterscheidet sich der Echtzeitbetrieb von der allgemeinen Datenverarbeitung durch das explizite Hinzutreten der Dimension *Zeit*:

> "Dieses drückt sich in der fundamentalen Benutzeranforderung nach *Rechtzeitigkeit* aus, die auch unter extremen Lastbedingungen erfüllt sein muß. Auf Anforderung durch externe Prozesse müssen Erfassung und Auswertung von Prozeßdaten sowie geeignete Reaktionen pünktlich ausgeführt werden. Dabei steht nicht die Schnelligkeit der Bearbeitung im Vordergrund, sondern die Rechtzeitigkeit der Reaktionen innerhalb vorgegebener und *vorhersehbarer* Zeitschranken – obwohl der derzeitige Stand der Technik noch weit davon entfernt ist, die Einhaltung solcher Zeitschranken garantieren zu können. Echtzeitsysteme sind mithin dadurch charakterisiert, daß die funktionale Korrektheit eines Systems nicht nur vom Resultat einer Berechnung bzw. einer Verarbeitung, sondern auch von der Zeit abhängt, wann dieses Resultat produziert wird. Korrekte Zeitpunkte werden von der Umwelt der Echtzeitsysteme vorgegeben, d.h. diese Umwelt kann nicht wie die von Stapelverarbeitungs- und Teilnehmersystemen dazu gezwungen werden, sich der Verarbeitungsgeschwindigkeit von Rechnern unterzuordnen."

In Bezug auf die Einhaltung von Zeitschranken kann man Echtzeitsysteme folgendermaßen unterscheiden:

harte Echtzeitsysteme (hard real-time systems): eine Überschreitung von Zeitschranken kann katastrophale Folgen haben und muß auf jeden Fall ausgeschlossen werden (z.B. Flugzeug im Landeanflug, Auslösen eines Air-Bag),

weiche Echtzeitsysteme (soft real-time systems): eine gelegentliche Überschreitung von Zeitschranken kann toleriert werden,

Die Einführung der Zeit in die Software kann erfolgen durch [BW97, Mar99]:

- Zugriff auf eine Uhr zur Zeitmessung,
- Verzögerung von Prozessen um eine definierte Zeit,
- Programmierung von timeouts, damit das Ausbleiben eines Ereignisses erkannt und behandelt werden kann,

- Spezifikation von Zeitschranken (deadlines) und Ablaufplanung (scheduling), damit die notwendigen Zeitbedingungen spezifiziert und eingehalten werden können.

Zeitschranken werden formuliert als zeitbehaftete Anweisungen und Anweisungsfolgen, sogenannte *Temporal Scopes (TS)*, für die fünf Zeitbedingungen unterschieden werden können:

1. die Deadline bis zur Beendigung von TS,
2. die minimale Verzögerung bis zum Start von TS,
3. die maximale Verzögerung bis zum Start von TS,
4. die maximale Ausführungszeit von TS,
5. die maximal während der Ausführung von TS verstrichene Zeit.

Programmiersprachen, die explizit Temporal Scopes unterstützen, gibt es nur wenige. Zu ihnen gehören PEARL und Real Time Euclid. Es ist Aufgabe eines Echtzeit-Betriebssystems, durch geeignete Ablaufplanung die Einhaltung von Zeitschranken zu gewährleisten.

2.3.3 Ressourcenbeschränkung

Eingebettete Systeme müssen aus verschiedenen Gründen oft mit in der Leistung limitierter Hardware oder mit speziellen DSPs realisiert werden. Wird in hohen Stückzahlen gefertigt, so kostet jede zusätzliche Chipfläche auch mehr Geld. Insbesondere für mobile akkubetriebene Geräte sind lange Betriebszeiten anzustreben. Bei der Softwareentwicklung ist daher mit Beschränkungen bezüglich Arbeitsspeicher und Prozessorleistung zu rechnen.

Diese Beschränkungen erschweren den Einsatz höherer Programmiersprachen zur Programmierung eingebetteter Systeme. Es sind Compiler erforderlich, die hochoptimierten und kompakten Code erzeugen können.

2.4 Anforderungen an Programmiersprachen

Halang et al. [HFL+98] geben für jede der in Abschnitt 2.3.1 vorgestellten Sicherheitsklassen typische Programmiermethoden, Verifikationsmethoden, Testmethoden und sichere Sprachkonstrukte an.

Die Autoren nennen folgende Anforderungen an Programmiersprachen für die Sicherheitsklasse SIL 1:

Sicherheitsklasse	Typische Programmiermethode
SIL 4	Ursache- / Wirkungstabellen
SIL 3	Funktionsplan mit verifizierten Bibliotheken
SIL 2	SafePEARL
SIL 1	HI-PEARL

Tabelle 2.2: Sicherheitsklassen und Programmiermethoden

1. in einer Sprache ausgedrücktes Verhalten soll immer zeitdeterministisch sein,
2. eine Sprache soll Konzepte zur Behandlung von Zeitbedingungen und Ereignissen bereitstellen (z.B. sollen alle Schleifenausführungen zeitbeschränkt sein),
3. eine Sprache soll modular und selbstdokumentierend sein und über einfache Programmierkonzepte verfügen,
4. eine Sprache soll keine unstrukturierten Sprachmittel enthalten (z. B. Sprunganweisungen, Semaphore und Bolts),
5. eine Sprache soll inhärent Systemverklemmungen verhindern (z. B. durch den Einsatz zeitbegrenzter Synchronisationsanweisungen),
6. eine Sprache soll keine Möglichkeit bieten, dynamische Elemente zu deklarieren,
7. eine Sprache soll standardisiert sein und eine eindeutige Semantik besitzen,
8. eine Sprache soll streng typisiert sein bzw. über adäquate Schnittstellenkonzepte verfügen,
9. eine Sprache soll strukturierte Sprachmittel zur Behandlung von Ausnahmesituationen bereitstellen.

Man kann wohl deutlich erkennen, daß bereits diese Anforderungen für SIL 1 von der gängigen Industriepraxis (Stichwort Assembler) schwerlich erfüllt werden. Von der Sprache C wird gänzlich abgeraten, allerdings existiert von anderer Seite her ein Vorschlag für eine sichere Untermenge *Safer-C*, siehe [Hat95]. Den Einsatz von C++ halten die Autoren unter Verzicht auf bestimmte Eigenschaften von C schon eher für denkbar. Ein Vorschlag dazu wird an anderer Stelle unter dem Namen *Embedded C++* diskutiert [Pla98].

Objektorientierte Sprachen können die oben genannten Anforderungen weitgehend erfüllen. In dieser Arbeit werden Konzepte und Übersetzungstechniken vorgestellt, die sich insbesondere auf die erste, sechste und siebte Empfehlung beziehen:

- in Kapitel 6 werden Techniken zur Implementierung der Methodensuche vorgestellt, die konstante Suchzeiten bei gleichzeitig minimiertem Platzbedarf garantieren,

- in Kapitel 7 werden Möglichkeiten zur statischen Allokation von Objekten aufgezeigt,
- in Kapitel 3.2.6 und Kapitel 4.2.4 ff. werden leistungsfähige statische Typsysteme mit Einfachvererbung und Schnittstellen diskutiert,
- bezüglich der letzten Empfehlung sei beispielsweise auf die Arbeit [HMP97] verwiesen.

Die Autoren fahren fort mit der Auflistung von Anforderungen für die Sicherheitsklasse SIL 2:

- eine Sprache soll in einfacher Weise und (halb-) automatisch formal überprüfbar sein,
- eine Sprache soll ausschließlich Schleifen zulassen, deren Durchlaufanzahl begrenzt ist,
- alle benutzten Variablen müssen explizit deklariert werden.

Sprachen, die diese Anforderungen erfüllen können, sind die in Abschnitt 2.5 vorgestellten synchronen reaktiven Sprachen.

Abschließend werden für die Sicherheitsklasse SIL 3 weitere Einschränkungen gefordert:

- eine Sprache soll in einfacher Weise und vollautomatisch formal überprüfbar sein,
- eine Sprache soll keine Schleifenanweisungen mehr enthalten,
- eine Sprache soll auf den Gebrauch (Einsatz und Verknüpfung) vorgefertigter, schon verifizierter (bzw. bereits zugelassener) Programmkomponenten eingeschränkt sein,
- die Benutzung von Variablen beschränkt sich auf solche, die innerhalb einer verifizierten Komponente schon definiert sind,
- die Schnittstellen eines verifizierten Moduls sollen so definiert sein, daß keine Möglichkeit besteht, den Zustand einer solchen Komponente ohne Überwachung zu ändern,
- eine Sprache soll eine graphische Darstellung unterstützen.

Eine Sprache für SIL 3 sind Funktionspläne mit verifizierten Bibliotheken. Die Anforderungen der Sicherheitsklasse SIL 4 werden nur von Ursache-/Wirkungstabellen erfüllt.

2.5 Synchrone Sprachen

Eingebettete Systeme werden auch als reaktive Systeme bezeichnet. Ein *reaktives System* reagiert kontinuierlich auf Aktionen (Eingaben) der Umgebung gemäß des aktuellen internen Zustands mit Reaktionen (Ausgaben) (EVA-Prinzip). Zu beachten ist, daß

1. die Zeitpunkte, zu denen eine Eingabe erfolgt, ausschließlich von der Umgebung bestimmt werden und
2. die Berechnung der Reaktion abgeschlossen sein muß, bevor die nächste Eingabe erfolgt.

Es liegen also auch hier Echtzeitanforderungen vor. Reaktive Systeme sind einer formalen Beschreibung und Verifikation durch *synchrone Sprachen* zugänglich. Nach [SSH+99] kann man synchrone Sprachen unterteilen in

Graphische synchrone Sprachen: StateCharts, Argos, SyncCharts;

Datenflußsprachen: Lustre, Signal;

Imperative synchrone Sprachen: Esterel, Reactive-C, SML, synchronousEifel.

Die Anweisungen synchroner Programmier- und Modellierungssprachen unterteilen sich in zeitverbrauchende und zeitlose. Jede *zeitverbrauchende Anweisung* benötigt ein vielfaches einer festgewählten logischen "Zeiteinheit". Synchrone Sprachen werden "getaktet" wie Hardware. Da logische Zeitmodelle in einem engen Zusammenhang mit dem realen Zeitverbrauch stehen, lassen sich wie bei der Hardware-Synthese werkzeugunterstützt automatisch Rückschlüsse auf das reale Echtzeitverhalten gewinnen.

Eine weitere Eigenschaft synchroner Sprachen ist die Möglichkeit, Parallelität auf der Ebene leichtgewichtiger Prozesse zu beschreiben, wobei die Kommunikation in der Regel durch global sichtbare Signale erfolgt, die auch Daten tragen können. Speziell Esterel bietet auch Konstrukte zur Unterbrechungsbehandlung an. In [SSH+99] wird erwähnt, daß Esterel-Beschreibungen (auch mit sehr vielen Prozessen) in ein sequentielles C-Programm übersetzt werden können, wobei als besonders vorteilhaft herausgestellt wird, daß dann ein (Echtzeit-) Betriebssystemkern zur (aufwendigen) Prozeßverwaltung nicht mehr benötigt wird.

Eine objektorientierte synchrone Sprache ist *synchronousEifel* [BMPS98, BM98, Bud98]. Nach einer persönlichen Mitteilung von Herrn Budde werden synchronous-Eifel-Programme recht einfach und direkt in äquivalenten C-Code übersetzt. Die Anwendung der in dieser Arbeit beschriebenen Techniken wäre sicherlich eine interessante näher zu prüfende Option.

2.6 Literaturhinweise

Eine Methodenlehre sicherheitsgerichteter Echtzeitprogrammierung findet sich in [HFL+98]. Dort werden auch Ursache-/Wirkungstabellen, Funktionspläne, die sicherheitsgerichtete Echtzeitprogrammiersprache HI_ PEARL und die aus PEARL abgeleitete objektorientierte Sprache PEARL* vorgestellt. Zu PEARL* [HF97, FH98] ist anzumerken, daß die Sprachdefinition zwar sehr viele interessante und moderne Konzepte aufgreift, grundlegende objektorientierte Konzepte wie polymorphe Nachrichtenausdrücke und dynamische Speicherverwaltung aber gar nicht erwähnt werden. Auch ist dem Kapitel nichts über eine eventuell begonnene oder bereits vorliegende reale Implementierung zu entnehmen.

Eine Diskussion synchroner Sprachen, der neuen Sprache PURR und des Verifikationssystems C@S bringt [SSH+99]. Dem Artikel können auch Literaturhinweise auf die oben erwähnten synchronen Sprachen entnommen werden.

Die objektorientierte Spezifikation und Simulation eingebetteter Realzeitsysteme wird in [OSN99] beschrieben. Die Modellierung auf Systemebene erfolgt dabei durch Anwendung der Designmethode HRT-HOOD [BW94, BW95], Softwarekomponenten werden in Ada95 und Hardwarekomponenten in Objective VHDL implementiert. Ada95 [BW97] ist eine objektorientierte Weiterentwicklung von Ada83, Objective VHDL ist eine objektorientierte Erweiterung von VHDL.

Kapitel 3

Objektorientierte Programmierung

Die Grundprinzipien der objektorientierten Programmierung sind unabhängig von ihrer Realisation in einer konkreten Programmiersprache. Es erleichtert daher die Darstellung und Verständlichkeit diese in einem eigenen Kapitel vorzustellen. Besondere Aufmerksamkeit wird dem Unterschied zwischen Klasse und Typ bzw. Implementierungs- und Schnittstellenvererbung gewidmet.

3.1 Einführung und Motivation

Definition 3.1.1 (Imperative Programmierung) [1]

$$Programme = Algorithmen + Datenstrukturen$$

Im konventionellen imperativen Stil geschriebene Programme bestehen aus Datenstrukturen und Anweisungen, die diese Daten manipulieren. Diese Konzeption orientiert sich an dem nach John von Neumann benannten Maschinenmodell. Ein Programm ist in dieser Sicht zu verstehen als die schrittweise Ausführung von (aktiven) Anweisungen, die (passive) Daten manipulieren und transformieren.

Klassische prozedurale Programmiersprachen wie Pascal oder C abstrahieren dieses Modell ohne es aber grundsätzlich in Frage zu stellen. Auf der Basis vordefinierter elementarer Datentypen können benutzerdefinierte Typen durch Aufzählung, Reihung, Verbund und Zeiger gebildet werden. Daten werden transformiert durch Prozeduren und Funktionen. Kontrollflußanweisungen sind elementarer Bestandteil der Sprache. Die Erzeugung und Verwaltung dynamischer Datenstrukturen erfolgt manuell.

[1] nach Niklaus Wirth

Definition 3.1.2 (Objektorientierte Programmierung) [2]

OO-Programme = abstrakte Datentypen (Klassen) + Vererbung +
dynamische Bindung

Die objektorientierte Programmierung basiert auf einem kommunikationsorientierten Modell. Objekte erscheinen als "kleine Computer", die ihren eigenen Speicher haben und durch Nachrichtenaustausch miteinander kommunizieren. Objekte repräsentieren physische oder abstrakte Gegenstände einer realen oder gedachten Welt. Die Ausführung eines objektorientierten Programms kann daher auch als Simulation dieser modellierten Welt betrachtet werden.

Die Innovation der objektorientierten Programmierung besteht in der Bereitstellung neuer und leistungsfähiger Sprachmittel zur Beschreibung von Objekten. Im Aufbau gleichartige Objekte werden beschrieben durch Klassen. Klassen sind abstrakte Datentypen, in denen die Daten und die sie manipulierenden Operationen zu einer Einheit zusammengefasst werden. Klassen können als vorgefertigte Bausteine in Klassenbibliotheken abgelegt werden. Eine Klasse kann allgemein nützliche Eigenschaften über die Vererbung an andere neue Klassen weitergeben, so daß diese Eigenschaften nur einmal beschrieben werden müssen. Der Kontrollfluß innerhalb eines Programms wird realisiert durch Nachrichtenaustausch mit dynamischer Bindung von Nachrichten an Methoden. Das Speichermodell ist abstrakt, die Verwaltung des Speichers erfolgt in der Regel automatisch.

Beispiel 3.1.1 (Binärer Berechnungsbaum)

Ein Berechnungsbaum besteht aus Knoten. Ein Wertknoten hält eine Zahl und hat keine Nachfolger. Die Auswertung eines Wertknotens liefert die Zahl selbst. Ein Plusknoten repräsentiert die Addition, er hat einen linken und rechten Nachfolger. Zur Auswertung eines Plusknotens werden rekursiv der linke und der rechte Nachfolger ausgewertet und die Ergebnisse addiert.

Beispiel 3.1.2 *Prozedurales Programm zur Auswertung binärer Berechnungsbäume*

```
type KnotenArt is enumeration(wert, plus);

type Knoten is record
    art:    KnotenArt;
    wert:   int;
    links:  pointer to Knoten;
    rechts: pointer to Knoten;
end
```

[2] nach Hanspeter Mössenböck

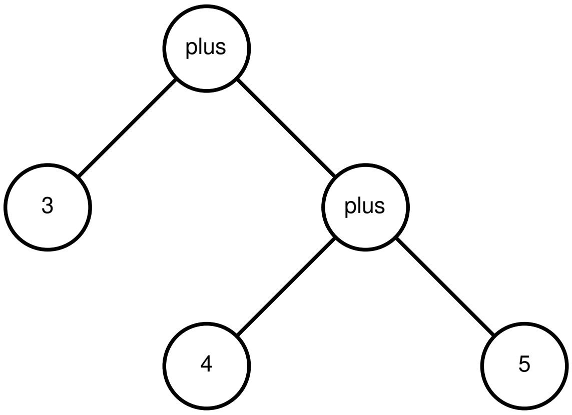

Abbildung 3.1: Ein Berechnungsbaum für 3+4+5

```
procedure init(k: Knoten, a: KnotenArt)
begin
    k.art   := a;
    k.wert  := 0;
    k.links := nil;
    k.rechts:= nil;
end

function evaluate(k: Knoten): int
begin
    case k.art of
        wert: return k.wert;
        plus: return evaluate(k.links^) + evaluate(k.rechts^);
    end
end
```

Die wesentlichen Merkmale dieses Programmierstils sind:

- Man muß sich von Anfang an Gedanken über Art und Anzahl der verschiedenen Knotentypen machen;
- Ein Knoten trägt immer einen Wert, auch wenn er ein Additionsknoten ist. Dementsprechend muß bei der Initialisierung dieser Wert immer gesetzt werden;
- Die Funktion evaluate muß eine explizite Fallunterscheidung zwischen den Knotenarten machen;
- Bei einer eventuellen Erweiterung um einen *Multiplikationsknoten* sind sowohl der Typ KnotenArt als auch die besagte Fallunterscheidung zu berücksichtigen, die Erweiterung ist also nicht lokal begrenzt.

Beispiel 3.1.3 *Objektorientiertes Programm zur Auswertung*[3]

```
abstract class Knoten
{
    Knoten  links;
    Knoten  rechts;

    method  (Knoten) init
    {
       links  := null;
       rechts := null;  // leere Referenzen auf Nachfolger
       return self;
    }

    abstract method  (int) evaluate
    {
       // leere Methode ohne Implementierung
    }
}

class WertKnoten extends Knoten
{
    int  wert := 0;

    method  (WertKnoten) initWith:(int )i
    {
       [super init];   // zuerst geerbte inst vars initialisieren
       wert := i;
       return self;
    }

    method  (int) evaluate
    {
       return wert;
    }
}

class PlusKnoten extends Knoten
{
    method  (int) evaluate
    {
       return [links evaluate] + [rechts evaluate];  // zwei Nachrichten
    }
}
```

[3]Das Programm ist in einer fiktiven Sprache notiert, die Elemente aus Java und Objective-C kombiniert. Insbesondere werden Nachrichtenausdrücke in der Schlüsselwort-Syntax von Objective-C formuliert. Die Sprache ist hybrid, hat ein statisches Typsystem und unterstützt späte Bindung.

Beispiel 3.1.4 *Berechnung des Ausdrucks 3 + 4*

```
PlusKnoten plus := [[PlusKnoten new] init]; // ererbte init Methode
WertKnoten drei := [[WertKnoten new] initWith: 3];
WertKnoten vier := [[WertKnoten new] initWith: 4];

plus.links := drei; plus.rechts := vier;

int ergebnis := [plus evaluate];
```

Die wesentlichen Merkmale dieses Programmierstils sind:

- Top-Down Entwurf beginnend mit einer *abstrakten Klasse* Knoten, welche nur die Struktur und die beabsichtigte Funktionalität beschreibt;
- Schrittweise Erweiterung durch Hinzufügung bzw. *Unterklassenbildung* von Wert- und Additionsknoten. Der Code zur Basisinitialisierung eines Knotens wird *wiederverwendet*;
- Jeder Knoten beschreibt in der Methode evaluate nur die ihm gemäße Funktionalität. Die Fallunterscheidung aus Beispiel 3.1.2 wird ersetzt durch *dynamische Bindung* zur Laufzeit;
- Dadurch ist einfache, lokal begrenzte und für die anderen Klassen transparente Erweiterung um einen Multiplikationsknoten möglich.

Im Beispiel wird in den Methoden init und initWith: Gebrauch gemacht von den wichtigen Pseudovariablen *self* und *super*. Wenn eine Methode sich auf self bezieht, dann ist das aktuelle Objekt gemeint, in dessen Kontext (im Sinne von Zustandsraum) die Methode ausgeführt wird. Mit super wird Bezug genommen auf die Vererbungshierarchie, gemeint ist dann immer die Methode in der direkten Oberklasse. Auf diese Weise kann elegant ererbtes Verhalten wiederverwendet und dann erweitert bzw. spezialisiert werden. Da in der Klasse PlusKnoten eine Beschreibung für die Methode init fehlt, wird bei entsprechender Nachricht an ein Objekt dieser Klasse die ererbte Methode aus der Oberklasse Knoten unverändert ausgeführt.

3.2 Grundlagen

Dieser Abschnitt soll in die Begriffswelt der Objektorientierten Programmierung einführen. Insbesondere wird auf den Unterschied zwischen Klasse und Typ und den Unterschied zwischen Implementierung und Schnittstelle eingegangen [GHJV95].

3.2.1 Objekte

Objektorientierte Programme bestehen aus *Objekten.* Ein Objekt faßt sowohl Daten als auch Prozeduren zusammen, die auf diesen Daten arbeiten. Die Prozeduren heißen üblicherweise *Methoden* oder *Operationen.* Ein Objekt führt eine Operation aus, wenn es eine Anfrage bzw. eine Nachricht von einem Klienten erhält.

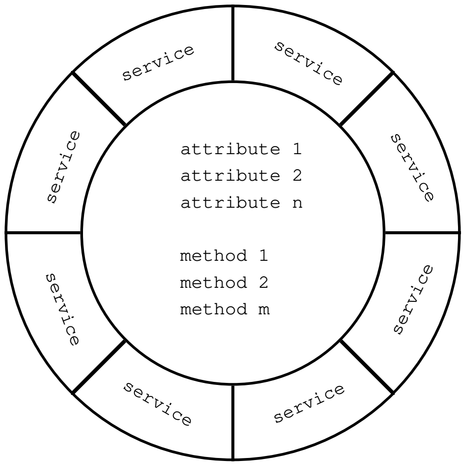

Abbildung 3.2: Ein Objekt ist ein Dienstanbieter

Anfragen bzw. *Nachrichten* sind der einzige Weg, ein Objekt dazu zu bringen, eine Operation auszuführen. Operationen sind der einzige Weg, um die internen Daten eines Objekts zu ändern. Man sagt aufgrund dieser Einschränkungen, daß der interne Zustand des Objekts gekapselt ist. Es kann nicht direkt auf ihn zugegriffen werden; seine Repräsentation ist außerhalb des Objekts nicht sichtbar (*Information Hiding*).

Für den Programmier kann es sehr hilfreich sein, sich ein ein Objekt als einen *Dienstanbieter* vorzustellen, der bestimmte genau festgelegte *Dienstleistungen (services)* erbringen kann. In der Regel wird es die Operation eines anderen Objekts sein, das diese Dienstleistung anfordert. Dieses andere Objekt wird auch *Klient* oder *Dienstnehmer* genannt (Abbildung 3.2).

Die Zerlegung eines Systems in Objekte ist Aufgabe und Ziel der objektorientierten Entwurfsmethoden. Objekte können dabei in Grösse und Anzahl drastisch variieren. Sie können alles von der Hardware bis hin zu vollständigen Anwendungen repräsentieren. Die repräsentierten Konzepte können sich auch von der realen Welt lösen, sie müssen keine physische Entsprechung haben.

3.2.2 Objektschnittstellen und Typen

Jede von einem Objekt deklarierte Operation spezifiziert den Operationsnamen, die Objekte, die sie als Parameter erhält, und den Rückgabewert der Operation. Dies nennt man die *Signatur* der Operation. Die Menge aller durch die Operationen eines Objekts definierten Signaturen nennt man die *Schnittstelle* des Objekts. Die Schnittstelle eines Objekts bestimmt die vollständige Menge von Anfragen, die an ein Objekt gerichtet werden können. Jede Nachricht, die zu einer Signatur in der Objektschnittstelle paßt, kann an dieses Objekt geschickt werden.

Ein *Typ* ist ein Name, der eine bestimmte Schnittstelle bezeichnet. Wir sagen von einem Objekt, daß es vom Typ "Knoten" ist, wenn es alle Anfragen akzeptiert, die in der Schnittstelle namens "Knoten" definiert sind. Ein Objekt kann viele Typen haben, und sehr unterschiedliche Objekte können sich einen Typ teilen. Ein Teil einer Objektschnittstelle mag durch einen Typ bestimmt sein, und andere Teile mögen durch andere Typen bestimmt sein. Zwei Objekte desselben Typs brauchen nur Ausschnitte ihrer Schnittstellen zu teilen.

Schnittstellen können andere Schnittstellen als Untermengen enthalten. Man spricht davon, daß ein Typ ein *Subtyp* eines anderen Typs ist, wenn seine Schnittstelle die Schnittstelle seines Supertyps enthält. Man sagt auch, daß der Subtyp die Schnittstelle seines Supertyps erbt, auch *Schnittstellenvererbung* genannt. Daraus folgt weiterhin, daß Subtypen kompatibel mit ihren Supertypen sind (aber nicht umgekehrt). WertKnoten und PlusKnoten sind z.B. Subtypen des Typs Knoten. Überall, wo ein Objekt vom Typ Knoten erwartet wird, kann auch ein Objekt vom Subtyp WertKnoten oder PlusKnoten eingesetzt werden.

Schnittstellen sind grundlegend für objektorientierte Systeme und das objektorientierte Denken. Will man etwas über ein Objekt herausfinden oder es zu bestimmten Diensten veranlassen, so muss man seine Schnittstellen kennen und verwenden. Eine Schnittstelle sagt nichts über ihre Implementierung, verschiedene Objekte können dieselben Nachrichten unterschiedlich implementieren. Das bedeutet insbesondere, daß zwei Objekte mit völlig unterschiedlichen Implementierungen dieselbe Schnittstelle haben können.

3.2.3 Dynamische Bindung und Polymorphismus

Wenn eine Nachricht an ein Objekt geschickt wird, hängt die genaue Operation sowohl von der Nachricht als auch von dem Empfängerobjekt ab. Verschiedene Objekte, die identische Nachrichten unterstützen, können unterschiedliche Implementierungen der Anfragen besitzen, d.h. unterschiedliche Operationen ausführen. Die zur Laufzeit hergestellte Verbindung einer Anfrage an ein Objekt mit einer seiner Operationen wird allgemein als *dynamisches Binden* bezeichnet.

Dynamisches Binden bedeutet, daß man beim Stellen einer Anfrage vor ihrer Ausführung nicht auf eine bestimmte Implementierung festgelegt ist. Somit kann man Programme schreiben, die ein Objekt mit einer bestimmten Schnittstelle erwarten, und man kann sicher sein, daß jedes Objekt mit einer passenden Schnittstelle die Nachricht akzeptieren wird. Weiterhin ermöglicht dynamisches Binden, Objekte mit identischen Schnittstellen zur Laufzeit gegenseitig zu ersetzen. Diese Austauschbarkeit von Objekten ist auch als *Polymorphismus* bekannt. Polymorphismus (Vielgestaltigkeit) ist ein weiteres zentrales Konzept objektorientierter Systeme. Ein Klient, der eine bestimmte Dienstleistung in Anspruch nehmen möchte, muß nur voraussetzen, daß das andere Objekt die entsprechende Schnittstelle erfüllt bzw. vom entsprechenden Typ oder Subtyp ist.

3.2.4 Objektimplementierungen und Klassen

Bis jetzt wurde von Objekten und ihren Schnittstellen gesprochen, aber nicht erklärt, wie man Objekte konkret definiert bzw. programmiert. Die Implementierung eines Objekts wird durch seine *Klasse* beschrieben. Die Klasse spezifiziert Zustand und Verhalten eines Objekts, d.h. Art und Anzahl der internen Daten und wie welche Operationen auszuführen sind. Klassen sind Vorlagen, Schablonen, Bauanleitungen für Objekte.

Objekte werden von einer Klasse erzeugt (instantiiert). Ein Objekt ist *Instanz* bzw. Exemplar seiner Klasse. Der Prozeß des Erzeugens eines Objekts durch seine Klasse alloziert Speicher für die internen Daten des Objekts, die Instanz- oder Exemplarvariablen, und verbindet die Operationen mit den Daten. Eine Klasse kann viele Objekte erzeugen. Jedes dieser Objekte wird durch eine eigene eindeutige *Objektreferenz* bezeichnet und besitzt einen eigenen internen Zustand. Zwei Objekte einer Klasse, die (meistens nur temporär) denselben Zustand besitzen, sind dennoch verschieden.

Neue Klassen können mittels existierender Klassen durch die Verwendung von *Klassenvererbung* gebildet werden. Wenn eine *Unterklasse* von einer Oberklasse erbt, bindet sie die Definitionen aller Daten und Operationen ein, welche von der Oberklasse definiert werden. Objekte einer Unterklasse enthalten alle von dieser Unterklasse und ihren Oberklassen definierten Daten und sind in der Lage, alle von dieser Unterklasse und ihren Oberklassen definierten Operationen auszuführen.

Unterklassen können das Verhalten ihrer Oberklassen verfeinern und verändern. Eine Klasse kann eine von seiner Oberklasse definierte Operation *überschreiben* und so das Verhalten modifizieren. Es ist möglich, daß die neue Implementierung der Operation die alte Implementierung aus der Oberklasse aufruft und auf diese Weise Code wiederverwendet. Unterklassen können auch neue Daten und Operationen definieren und implementieren, die die Oberklasse nicht kennt.

Eine *abstrakte Klasse* ist eine Klasse, deren Hauptzweck darin besteht, eine für alle Unterklassen gemeinsame Schnittstelle zu definieren. Von einer abstrakten Klasse können keine Exemplare gebildet werden, da eine abstrakte Klasse mindestens eine Operation nur deklariert, aber nicht implementiert. Diese heißen abstrakte Operationen oder abstrakte Methoden. Die nichtabstrakten Klassen, von denen Exemplare gebildet werden können, heißen auch *konkrete Klassen.*

3.2.5 Einfach- versus Mehrfachvererbung

Bis jetzt wurde gesagt, daß eine neue Klasse als Unterklasse einer einzigen Oberklasse gebildet werden kann. Dieses nennt man *Einfachvererbung (single inheritance)*. Es ist aber auch möglich, eine Unterklasse von mehreren Oberklassen zu bilden, dieses Vorgehen heißt *Mehrfachvererbung (multiple inheritance)*. Mehrfachvererbung ist problematisch und wird daher nur von wenigen objektorientierten Sprachen unterstützt.

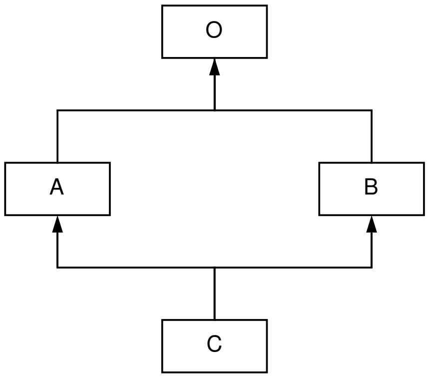

Abbildung 3.3: Rautenproblem bei Mehrfachvererbung [Mös98]

Man betrachte die rautenförmige Vererbungsbeziehung in Abbildung 3.3. Die in Klasse O definierten Daten und Operationen werden an Klasse A und Klasse B vererbt. Es ist weiterhin möglich, daß in A und B Daten und Operationen mit identischen Namen und Signaturen definiert werden. Da Klasse C alle Daten und Operationen sowohl von A als auch von B erbt, kommt es dann zwangsläufig zu Namenskollisionen. Welche Operation hat Vorrang, die aus A oder die aus B? Das gleiche Problem entsteht für die indirekt geerbten Daten und Operationen aus O. Sollen sie in C doppelt vorkommen oder nur einmal?

Mehrfachvererbung verkompliziert die Sprachdefinition und führt bei der Übersetzung zu stark erhöhtem Zeit- und Platzbedarf für Objekte und Methodensuche, siehe dazu insbesondere [WM96, Kapitel 5.3.2].

Die Erfahrungen mit der Smalltalk-Klassenbibliothek und anderen Bibliotheken haben gezeigt, daß Einfachvererbung ausreichend mächtig ist. Zudem wird Mehrfachvererbung oft zur Deklaration von Schnittstellen benutzt (über abstrakte Klassen), was sich durch entsprechende Sprachkonstrukte viel direkter und unproblematisch erledigen läßt (siehe unten).

In dieser Arbeit wird Mehrfachvererbung daher nicht weiter berücksichtigt werden.

3.2.6 Implementierungs- versus Schnittstellenvererbung

An dieser Stelle soll nochmal auf Klasse und Typ eines Objekts Bezug genommen und der Unterschied verdeutlicht werden.

Die Klasse eines Objekts beschreibt, wie das Objekt implementiert ist. Die Klasse definiert den internen Zustand des Objekts und die Implementierung seiner Operationen. Im Gegensatz dazu bezieht sich der Typ eines Objekts (bzw. die Typen) lediglich auf seine Schnittstelle, also auf die Menge von Anfragen, die es beantworten kann. Ein Objekt ist immer Exemplar genau einer Klasse, aber ein Objekt kann viele Typen haben, und die Objekte verschiedener Klassen können von demselben Typ sein.

Neben Unterschieden existieren auch Gemeinsamkeiten. Da eine Klasse die Operationen definiert, die ein Objekt ausführen kann, bestimmt sie ebenfalls einen Typ. Wenn also ein Objekt Exemplar einer Klasse ist, so folgt daraus, daß das Objekt auch die von der Klasse definierte Schnittstelle unterstützt.

In diesem Sinne setzen die meisten objektorientierten Sprachen Klasse und Typ einfach identisch, was das Verständnis für den Unterschied natürlich erschwert. Wenn man in C++, Oberon-2 oder Java eine Klasse programmiert, so hat man gleichzeitig einen Typ definiert[4]. Klassen erfüllen in diesen Sprachen eine Doppelfunktion. Objective-C, Java, Sather-K sind mir bekannten Sprachen, die es erlauben, den Unterschied explizit zu machen: Objective-C kennt *Protokolle* und Java die *Interfaces* als eigenes Sprachkonstrukt zur Spezifikation von Schnittstellen, d.h. von benannten Mengen von Methoden-Signaturen. Man spricht in diesen Sprachen davon, daß eine Klasse *ein Protokoll befolgt* bzw. *ein Interface implementiert.* Darüberhinaus ist es in Java sogar möglich, eine Variable mit einem Interface zu typisieren. Sather-K kann unterscheiden zwischen Vererbung zur Untertypbildung und reiner Code-Wiederverwendung.

Wichtig ist auch der Unterschied zwischen *Implementierungsvererbung (Unterklassenbildung, Klassenvererbung)* und *Schnittstellenvererbung (Subtyping, Untertypbildung).* Klassenvererbung definiert die Implementierung von Objekten mittels der Implementierung von anderen Klassen. Es handelt sich also um einen Mechanismus zur Wiederverwendung von Code und interner Repräsentation. Im Gegensatz dazu beschreibt die Schnittstellenvererbung, wann ein Objekt anstelle eines anderen Objekts verwendet werden kann.

Da die meisten Sprachen wie gesagt Klasse und Typ identisch setzen, müssen sie zwangsläufig auch die beiden genannten Vererbungsarten mit einem Konzept bedienen. Reine Schnittstellenvererbung realisiert man dann durch öffentliches Erben von abstrakten Klassen, reine Implementierungsvererbung durch privates Erben von konkreten Klassen[5].

Schnittstellen als Sprachkonstrukt wie in Java und Objective-C sind zudem eine elegante Möglichkeit, die Nachteile der Mehrfachvererbung bei Klassen zu vermeiden, ohne auf die meisten Vorteile derselben verzichten zu müssen.

---

[4] Besonders deutlich bei Oberon-2, wo gar nicht erst von Klassen und Klassenvererbung, sondern nur von Typen und Typerweiterung gesprochen wird [Wir88].

[5] Es gibt natürlich auch Mischformen, z.B. die (abstrakten) Reader und Writer Klassen in Java, die sowohl eine Schnittstelle deklarieren als auch eine Teilimplementierung liefern.

Gamma et al. [GHJV95] geben in ihrem Buch über Entwurfsmuster abschließend die Empfehlung:

> Programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung.

Diese Empfehlung gewinnt im Zusammenhang mit *komponentenbasierter Software-Entwicklung* zunehmend an Bedeutung. Eine exzellente Einführung in dieses neue Gebiet ist [Szy98].

3.3 Kontext und Bewertung

Bei der Programmierung eingebetteter Systeme haben *Einfachheit* und *Sicherheit* absoluten Vorrang. In diesem Kapitel wurde daher besonders betont, wie sich die Komplexitäten der Mehrfachvererbung bei Klassen durch explizite Unterscheidung von Klassen- und Schnittstellenvererbung vermeiden lassen. Eine objektorientierte Programmiersprache für eingebettete Systeme sollte daher nur einfache Klassenvererbung (single inheritance) unterstützen, aber zur Erhöhung der Ausdruckskraft Schnittstellen und Schnittstellenvererbung auf Sprachebene kennen. Beispiele für solche Sprachen sind Java und Objective-C.

Zur Sicherheit können insbesondere Mechanismen zur Kapselung von Merkmalen und statische Typsysteme mit später Bindung beitragen, siehe dazu das Kapitel 4.2.2 und die Kapitel 4.2.4 bis 4.2.7.

3.4 Literaturhinweise

Die Vorteile der objektorientierten Programmierung und Empfehlungen zur Durchführung entsprechender Projekte werden in [GR95] dargestellt. In [RBP+93] wird die Object-Modeling Technique (OMT) vorgestellt, welche die objektorientierte Denkweise auf alle Phasen der Softwareentwicklung von der Analyse über den Entwurf bis zur Implementierung anwendet. Speziell auf die Entwurfsphase ausgerichtet ist das einflußreiche Buch von Gamma et al. [GHJV95], das für spezifische Probleme des objektorientierten Softwareentwurfs einfache und elegante Lösungen beschreibt, die sogenannten Entwurfsmuster.

Für weitere Literaturhinweise sei auf Kapitel 4.5 verwiesen.

Kapitel 4

Objektorientierte Programmiersprachen

Die im letzten Kapitel vorgestellten Konzepte der objektorientierten Programmierung finden ihre konkrete Ausprägung in den diversen bekannten und weniger bekannten objektorientierten Programmiersprachen. Dabei fördert eine genauere Betrachtung durchaus signifikante Unterschiede zu Tage, die sich auf die Ausdruckskraft und Performanz der Sprachen auswirken.

4.1 Einführung

Allen objektorientierten Sprachen gemeinsam ist die Implementierung der in Kapitel 3 genannten Konzepte Objekt, Klasse, Vererbung und Methode. Dennoch existieren weitreichende Unterschiede beispielsweise im Typsystem, in der Bindung von Methoden und im Grad der Reinheit in der Umsetzung der genannten Konzepte.

Anmerkung: Es wird in dieser Arbeit sehr oft von der Unterscheidung *statisch / dynamisch* gebrauch gemacht werden. Diese meint im Kontext objektorientierter Programmiersprachen mehr oder weniger dasselbe wie die Unterscheidungen *Übersetzungszeit / Laufzeit* bzw. *compile-time / run-time*.

Die von der Xerox PARC Software Concepts Group entwickelte Sprache Smalltalk-80 [GR83, Gol84] ist *die* klassische reine objektorientierte Sprache. Sie zeichnet sich aus durch dynamische Typisierung und dynamische Bindung der Methodenaufrufe. Weiterhin verfügt die Sprache über ein Block-Konzept, welches den Lambda-Ausdrücken funktionaler Sprachen wie Scheme oder ML vergleichbar ist. Smalltalk Programme werden in einen Byte-Code übersetzt, der dann von einer virtuellen Maschine ausgeführt wird.

Die hohe Ausdrucksfähigkeit dieser Sprache zusammen mit der interpretierten Ausführung brachten der Sprache von Anfang an Performanzprobleme ein, deren Behebung aber schon früh Gegenstand zahlreicher wissenschaftlicher Publikationen war

[DS84, Kra83]. Die dort entwickelten Optimierungstechniken wie die dynamische Compilierung des Byte-Codes und das Caching von Methodenaufrufen verweisen die immer noch hartnäckig behauptete Langsamkeit von Smalltalk schon lange in den Bereich des Mythos.

Die inzwischen extrem populäre Sprache Java [AG98] ist eine (fast) reine objektorientierte Sprache mit statischer Typisierung und hauptsächlich spät gebundenen Methodenaufrufen. Dynamische Bindung kennt Java nur bei Interfaces. Auch Java wird in einen Byte-Code übersetzt, der von einer virtuellen Maschine ausgeführt wird. Im Unterschied zu Smalltalk existiert kein Block-Konzept. Viele der im Zusammenhang mit Smalltalk entwickelten Optimierungstechniken werden unter anderem Namen auch für Java eingesetzt. Die kürzlich von Sun vorgestellte hochoptimierende virtuelle Maschine HotSpot greift zusätzlich noch Ideen auf, die dem Self-Projekt der Stanford University und Sun entstammen.

Die bekanntesten hybriden objektorientierten Sprachen sind Objective-C, C++, Eiffel, Oberon-2, CLOS. Es existiert praktisch für jede Sprache eine objektorientierte Erweiterung, selbst für Cobol.

Objective-C [CN91] und C++ sind beide Erweiterungen der imperativen Sprache C. Erstere kann charakterisiert werden als Einbettung von Smalltalk in C unter Verzicht auf das Block-Konzept und die virtuelle Maschine. Objective-C kennt daher dynamische Typisierung und dynamische Bindung und erreicht annähernd die Mächtigkeit von Smalltalk. Objective-C ist Grundlage der Frameworks von NeXT-Step und jetzt Apple MacOS X.

Wesentlich bekannter ist allerdings C++ [ES92]. Ähnlich wie Oberon basiert C++ auf der Erweiterung des Verbundes (record, struct) zur Klasse (class). Die Sprache ist statisch typisiert und kennt nur späte Bindung. Ein Template-Konzept für generische Typen soll für mehr Flexibilität sorgen. C++ zeichnet sich aus durch eine ständig expandierende Sprachdefinition mit entsprechend verwirrender Syntax und Semantik.

Oberon und Oberon-2 sind Weiterentwicklungen von Pascal bzw. Modula-2. Hervorzuheben sind die bekannt klare Syntax und das ausgefeilte Modulkonzept. Oberon-2 ist streng statisch typisiert mit später Bindung.

4.2 Charakterisierung von OO-Sprachen

Zur näheren Charakterisierung objektorientierter Programmiersprachen eignet sich ein Gestaltungsraum mit folgenden Dimensionen [GR95, Man97]:

1. Objektabstraktion
2. Kapselung
3. Objektlebensdauer
4. Typisierung
5. Typprüfung
6. Vererbung
7. Bindung

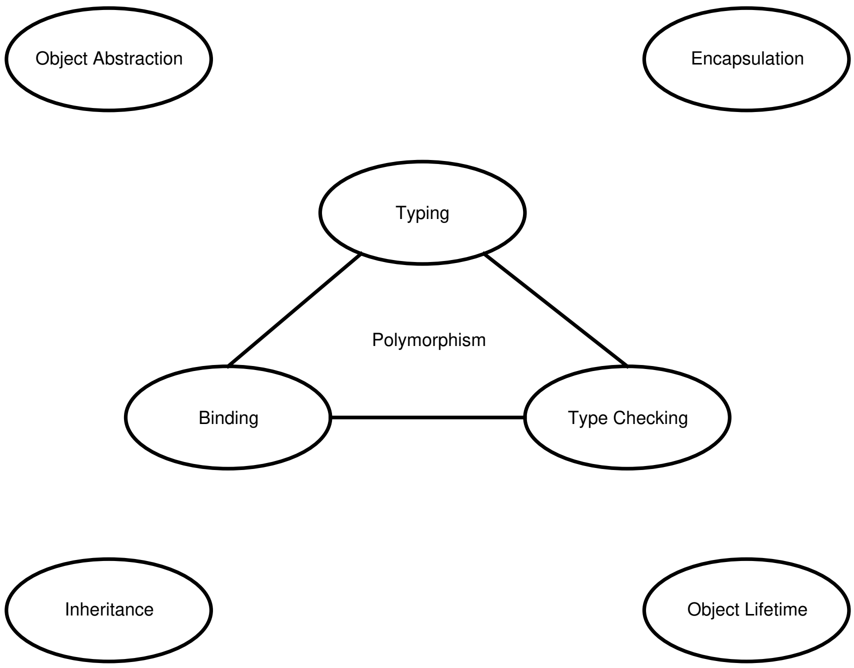

Abbildung 4.1: Dimensionen des Gestaltungsraumes

Die sieben Dimensionen dieses Raumes liefern einen allgemein anwendbaren Rahmen für die Klassifikation der objektorientierten Programmiersprachen. Eine konkrete Ausprägung dieser vielen Designmöglichkeiten in einer Sprache bezeichnet man auch als das *Objektmodell* der Sprache.

Die vorliegende Arbeit konzentriert sich hauptsächlich auf die Beschreibung des *klassischen Objektmodells*, wie es durch die Sprachen Smalltalk-80 und Simula-67 kanonisiert worden ist. Die entscheidenden Merkmale dieses Modells sind die Klasse als Abstraktionsmechanismus für Objekte, die Unterklassenbildung als Mittel der Vererbung und die späte bzw. dynamische Bindung von Nachrichten an Methoden.

4.2.1 Objektabstraktion

Nach Goos [Goo96] ist ein Objekt ein elementares Teilsystem. Es repräsentiert einen beliebigen Gegenstand (Person, Ding, Sachverhalt) der realen Welt und besitzt meßbare, durch Werte erfassbare Eigenschaften. Diese heissen *Attribute* des Objekts und bilden seinen *Zustand*. Ein Objekt kann Tätigkeiten ausführen, auf äussere Ereignisse reagieren und selbst wieder Ereignisse auslösen, also ein bestimmtes *Verhalten* zeigen. Die Handlungsvorschriften heissen *Methoden* und bilden zusammen mit den Attributen die *Merkmale* des Objekts.

Die Identifikation elementarer Teilsysteme im Rahmen der Modellierung eines Problembereichs ist eine aktive Abstraktionsleistung des menschlichen Beobachters. Das Resultat eines solchen Abstraktionsvorgangs ist die Bildung von Objekt-*Klassen*.

Klassen erfüllen in objektorientierten Progammiersprachen [Bla94, CN91] vielfältige Aufgaben:

- Klassen sind ein Mittel zur Beschreibung von Objekten mit gleichen Eigenschaften;
- Klassen helfen, Programme zu strukturieren;
- Klassen erzeugen neue Objekte, sie werden daher oft auch als *Objektfabriken* bezeichnet;
- Klassen erleichtern die Implementierung von Objekten; im Sinne einer Arbeitsteilung formuliert kann man sagen: Die Objekte haben ihren jeweiligen individuellen Zustand, die Klasse liefert dazu das allen gemeinsame Verhalten.

Nach der herrschenden Meinung sind Objekte ohne Klassen nicht denkbar. In der Forschung wird jedoch schon seit einiger Zeit in Gestalt der *prototypenbasierten objektorientierten Sprachen* ein anderer Ansatz untersucht. Die Grundidee ist hier, daß die Vorlage für ein neues Objekt nicht eine Klasse ist, sondern schlicht ein bereits bestehendes Objekt, welches kopiert wird. Das bestehende Objekt ist der *Prototyp* für das neue Objekt. Dem neuen Objekt können dann zur Laufzeit neue Merkmale hinzugefügt werden. Vererbungsbeziehungen zwischen Objekten sind möglich und können zur Laufzeit sogar geändert werden. Bekannte Sprachen dieser Art sind Self, NewtonScript, Omega, Kevo. Prototypenbasierte Sprachen werden meines Wissens derzeit nicht kommerziell eingesetzt[1].

Interessante philosphische und historische Anmerkungen zu Klassen und Prototypen macht A. Taivalsaari [NT96]. Die philosphischen Grundlagen der *klassenbasierten Sprachen* sieht er in Platos Untescheidung von Klassen und Exemplaren und in Aristoteles Interesse an hierarchischer Klassifikation. Er argumentiert weiter,

[1] Eine bedeutende Ausnahme war der Apple Newton PDA, der aber nicht mehr produziert wird. Der Newton und NewtonScript sind meiner Meinung nach eine der herausragenden Innovationen des letzten Jahrzehnts.

daß *prototypenbasierte Sprachen* durch Wittgensteins Kritik der Klassifikation inspiriert wurden. Es sei schwierig im Vorhinein zu sagen, welche Eigenschaften und Hierarchien benötigt werden, so daß Konzepte besser zu definieren sind über Familienähnlichkeiten zwischen verwandten Objekten. Diese Ideen wurden in den siebziger Jahren von Eleanor Rosch zur Prototypentheorie weiterentwickelt, die besagt, daß bestimmte Mitglieder einer Familie (Prototypen) bessere Repräsentaten der Familie sind als andere.

Der Autor beschreibt folgende Konsequenzen dieser Überlegungen. Da Klassifikationen im Vorhinein nicht zu bestimmen sind, gibt es keine optimalen Klassenhierarchien und die Gestaltung von Klassenbibliotheken muß als ein zeitlicher und evolutionärer Prozeß gesehen werden. Klassenhierarchien entwickeln sich "aus der Mitte" heraus; und insbesondere Klassen, die hoch in der Hierarchie angesiedelt sind, sind notwendigerweise sehr allgemein und werden erst nach den einfachen Klassen entdeckt. Unglücklicherweise erfordern klassenbasierte Sprachen, daß Klassenhierarchien top-down zu definieren sind, wobei die abstraktesten Klassen zuerst beschrieben werden müssen. Dem Autor zufolge können prototypenbasierte Sprachen diese Nachteile vermeiden.

Die vorliegende Arbeit konzentriert sich auf klassenbasierte Sprachen, viele der beschriebenen Techniken lassen sich jedoch auf prototypenbasierte Sprachen übertragen.

4.2.2 Kapselung

Es ist im Sinne des *Information Hiding* wichtig, daß Klienten eines Objekts nicht den vollen Zugriff auf alle Merkmale des Objekts haben, sondern sich an die öffentliche Schnittstelle halten. Die Einhaltung einer solchen Schnittstelle wird in vielen Sprachen durch Mechanismen der Kapselung auf Objekt-, Klassen- und Modulebene erzwungen.

Schutz von Merkmalen

Die mit C++ eingeführten Sichtbarkeitsmodifikatoren *public*, *private* und *protected* sind ein sehr feinkörniger Schutzmechanismus für Merkmale auf Objekt- und Klassenebene. Bezüglich des Objekts wird geregelt, ob ein Merkmal für einen Klienten sichtbar ist, bezüglich der Klasse wird geregelt, ob ein Merkmal in Unterklassen sichtbar ist (siehe Tabelle 4.1).

Beispiel 4.2.1

```
public class Point
{
    protected int x;
    protected int y;
}
```

Wird beispielsweise ein Merkmal der Klasse Point als protected markiert, so ist das Merkmal zwar in Unterklassen von Point sichtbar, nicht jedoch für Klienten eines Objekts dieser Klasse. Smalltalk behandelt einheitlich alle Attribute als private und alle Methoden als public.

		Klasse	
		vererbt sichtbar	vererbt unsichtbar
Objekt	sichtbar	public	—
	nicht sichtbar	protected	private

Tabelle 4.1: Sichtbarkeitsmodifikatoren für Merkmale

Module und Pakete

Oft arbeiten mehrere Klassen eng zusammen zur Erfüllung einer Aufgabe oder sie lassen sich aus inhaltlichen Gründen zu einer Gruppe formieren. Es wäre sinnvoll, wenn man solche Klassen zu einer höheren Einheit zusammenfassen könnte.

Java implementiert ein datei- und verzeichnisbasiertes Paketkonzept. Jede Datei nimmt eine Klassendefinition auf, mehrere Dateien in einem Verzeichnis bilden ein *Paket (package)* und Unterpakete entsprechen Unterverzeichnissen. Klassen innerhalb eines Pakets sind füreinander sichtbar. Die oben genannten Sichtbarkeitsmodifikatoren regeln den Export ausgewählter Klassen nach außen.

Das Modulkonzept von Oberon-2 basiert auf Modula-2. Klassen (Typen), benutzerdefinierte Typen, Methoden etc. können in Form einer Datei zu einem *Modul* zusammengefasst werden. Ferner kann bestimmt werden, welche Klassen mit welchen Merkmalen exportiert werden und es ist im Modulkopf anzugeben, welche Klassen aus welchen Modulen man zu importieren wünscht.

Einige Sprachen wie beispielsweise Modula-3, PEARL* und Dylan erlauben es überdies dem Programmier, ein Modul mit *mehreren öffentlichen Schnittstellen* zu versehen (multiple interfacing), so daß der Export der im Modul realisierten Funktionalität auf die Bedürfnisse verschiedener Klienten hin spezialisiert werden kann.

Ein Übersetzer kann und muß zur Übersetzungszeit prüfen, ob die Sichtbarkeitsregeln eingehalten werden. Dazu sind seine Symboltabellen durch entsprechende Attribute für jede Klasse und jedes Merkmal einer Klasse zu erweitern.

4.2.3 Objektlebensdauer und Status von Klassen

Alle Objekte unterliegen dem folgenden *Lebenszyklus:*

1. Jedes Objekt muß zuerst erzeugt werden. Diese Phase nennt man auch Instanzerzeugung (instance creation). Es findet sofort danach eine Initialisierung statt, entweder durch Konstruktoren (C++, Java) oder per Konvention durch Aufruf speziell ausgezeichneter Methoden.

2. In der zweiten Phase ist das Objekt für eine bestimmte Zeit in Gebrauch. Im Hinblick auf effiziente Speicherbereinigung kann diese Phase weiter unterteilt werden [Ung84].

3. Die letzte Phase ist die Löschung des Objekts, wenn es nicht länger benötigt wird. Diese Phase muß bei automatischer Speicherbereinigung nicht explizit programmiert werden. Manche Programmiersprachen erlauben eine letzte Intervention durch Destruktoren (C++) oder Finalizer (Java, Dylan), z.B. für Aufräumarbeiten bei verketteten Datenstrukturen.

Alle Phasen dieses Zyklus finden zur Laufzeit des Programms statt. Dies impliziert zwangsläufig das Vorhandensein eines Laufzeitsystems, welches Aufgaben der Speicherverwaltung übernehmen muß. Man kann unterscheiden manuelle, halbautomatische und automatische Speicherbereinigung (garbage collection). Weiterhin kennen viele Sprachen Vorrichtungen zur Serialisierung von Objekten zwecks Speicherung auf Festplatten oder Transfers über Netzwerke. Objekte können also durchaus die Beendigung des Programms, welches sie erzeugt hat, überleben (Persistenz).

Klassen haben abhängig vom Design der betrachteten Programmiersprache einen unterschiedlichen *Status*. Man kann exemplarisch folgende Möglichkeiten unterscheiden:

C++: Klassen existieren nur zur Übersetzungszeit, zur Laufzeit ist keinerlei Information mehr verfügbar.

Oberon-2: Klassen existieren im wesentlichen nur zur Übersetzungszeit, gewisse Informationen sind aber zur Laufzeit über einen sogenannten *Typdeskriptor* abfragbar.

Java: Klassen sind spezielle Datenstrukturen, alle Eigenschaften (Attribute, Methoden etc.) der Klasse sind zur Laufzeit über die sogenannte *Reflektion* abfragbar. Klassen können Klassenmethoden und Klassenvariablen besitzen. Die virtuelle Javamaschine JVM kennt einen eigenen Byte-Code zum Aufruf von Klassenmethoden (invokestatic).

Smalltalk, Objective-C: Klassen sind Objekte, als solche betrachtet sind Klassen dann Instanzen von sogenannten *Metaklassen* (Abbildung 4.2). Die normale Klassenhierarchie spiegelt sich in einer zusätzlichen Metaklassenhierarchie.

Metaklassen sind eine allgemeine Möglichkeit der Bereitstellung von *Klassenmethoden* und *Klassenvariablen* einschließlich Vererbung derselben. Der unendliche Regress, der sich dadaurch ergibt, daß Metaklassen eigentlich auch wieder Instanzen von Metametaklassen sind usf., wird in Smalltalk durch eine Anomalie in der Klassenhierarchie aufgelöst, die letztlich alle Klassen auf die Wurzelklasse Object zurückführt. Eine ähnliche Anomalie existiert bei Objective-C und den Frameworks von NeXT/Apple. Abbildung 4.3 zeigt den entsprechenden Ausschnitt aus der Smalltalk-Klassenhierarchie.

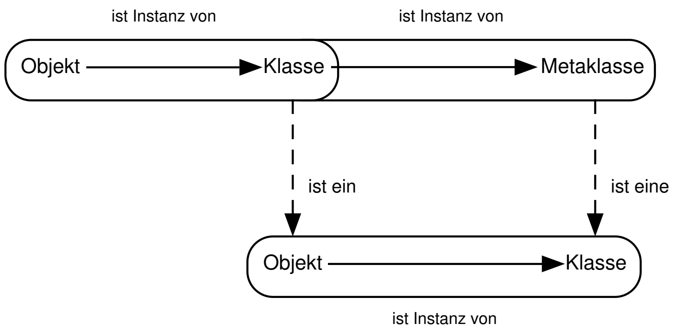

Abbildung 4.2: Klassen sind Instanzen von Metaklassen

Der Aufwand für die Implementierung und Übersetzung von Klassen ist für die erste Variante am geringsten und für die letzten beiden am höchsten.

Metaklassen können die Ausdruckskraft und Flexibilität einer objektorientierten Sprache erheblich steigern, erhöhen aber auch insbesondere den Platzbedarf für übersetzte Programme, da zusätzlich zu jeder regulären Klasse der Klassendeskriptor und die Methodentabelle der Metaklasse zu speichern sind (siehe Kapitel 5.6.5). Zur näheren Erläuterung und Motivation von Metaklassen sei auf die Smalltalk und Objective-C Literatur verwiesen.

4.2.4 Typisierung

Man kennt von den klassischen imperativen Sprachen die Datentypen int, float, struct, pointer etc. Datentypen haben eine Repräsentation im Speicher und für jeden Datentyp legt die Sprachdefinition eine Menge erlaubter Operationen (mit Syntax und Semantik) fest. So darf man zwei int's mit "+" addieren, struct's aber nicht. Verkettete Listen realisiere ich am besten mit Zeigern, denn nur diese bieten geeignete Operationen an. Für den Programmierer ist es daher letztlich wichtiger zu wissen: Was kann dieser Datentyp, ist sein Verhalten für meine Zwecke geeignet? Seine konkrete Repräsentation ist dagegen für den Programmierer zweitrangig.

Man kann also sagen: Typisierung ist eine Form der Verhaltensfestlegung, bestimmte Operationen sind erlaubt, andere dagegen verboten. Gleiches gilt für den Zugriff auf Instanzvariablen. Objektorientierte Sprachen kennen im wesentlichen zwei Formen der Typisierung:

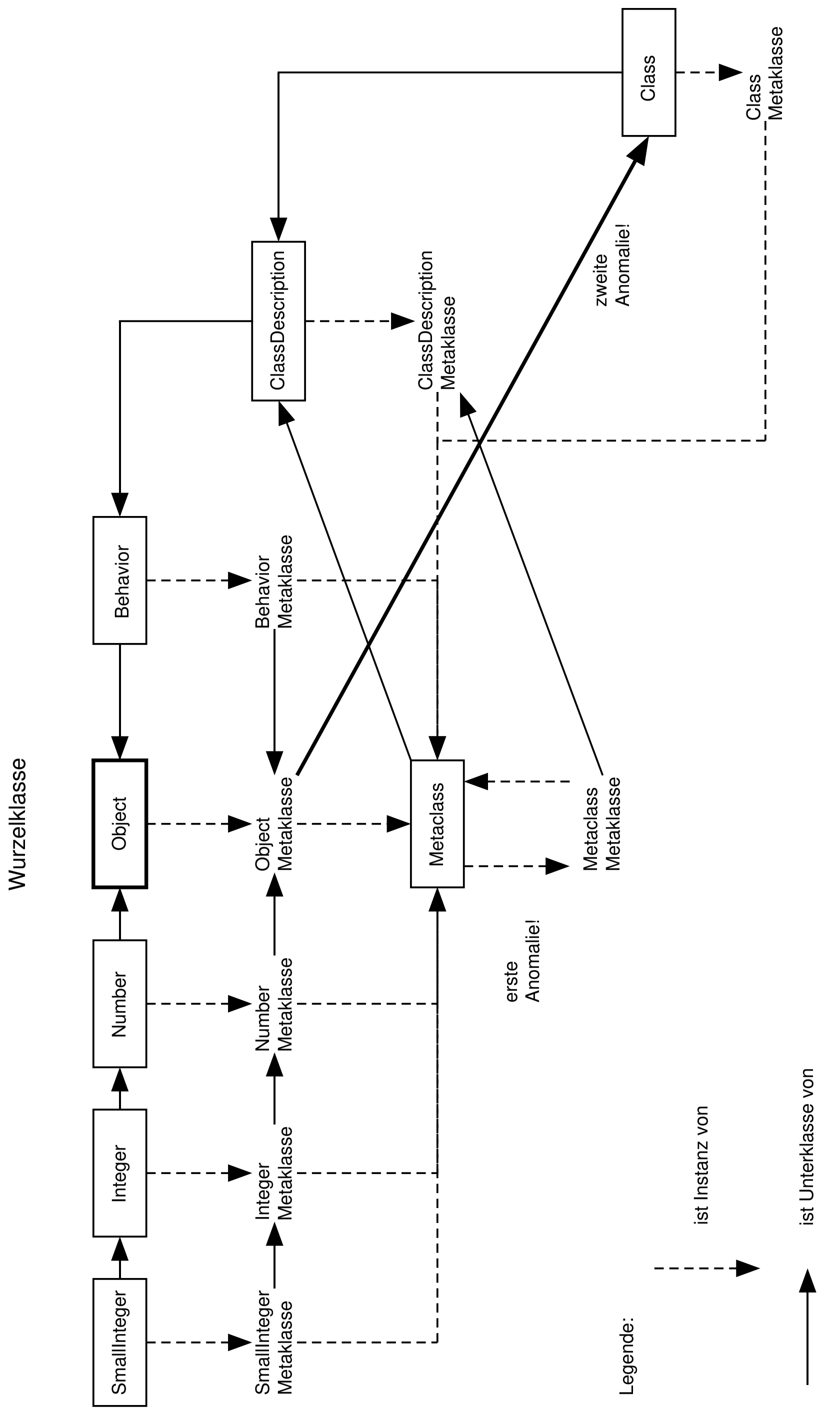

Abbildung 4.3: Metaklassen und ihre Einbindung in die Smalltalk-Klassenhierarchie

Statische Typisierung: Im Quelltext des Programms müssen alle Variablen und Parameter von Methoden explizit mit einer Typangabe versehen werden, auch als formale Deklaration bezeichnet. Der Typ ist also bereits zur Übersetzungszeit bekannt.

Dynamische Typisierung: Variablen und Parameter werden ohne explizite Typangabe (informell) deklariert. Variablen können immer auf Objekte beliebigen Typs verweisen. Der Typ wird erst zur Laufzeit des Programms durch Instantiierung einer Klasse festgelegt. Nicht die Variable ist Trägerin der Typinformation, sondern das Objekt selbst durch einen internen Verweis auf seine Klasse.

Übrigens sollte man die *Deklaration* einer Variablen, die auf ein Objekt verweist, nicht mit der *Allokation* des Objekts selbst verwechseln. Objekte werden in der Regel erst zur Laufzeit des Programms dynamisch auf dem Heap alloziert, dies gilt auch für statisch typisierte Sprachen. Allerdings gestatten einige Sprachen eine Ausnahme von dieser Regel (C++, Objective-C, Oberon), diese kann vom Programmierer oder einem optimierenden Übersetzer zur Effizienzsteigerung genutzt werden. Auch bezüglich der Allokation ist also eine Unterscheidung nach statisch / dynamisch möglich und sinnvoll (siehe auch Kapitel 7).

Ein Übersetzer nutzt die Typinformation zur

- Typprüfung,
- Festlegung einer internen Repräsentation der Objekte einer bestimmten Klasse,
- Dekodierung der in einem Laufzeitobjekt abgelegten Bitmuster zwecks Zugriff auf Instanzvariablen,
- Generierung von Laufzeitdatenstrukturen und Code zum Finden und Ausführen von Methoden.

Beispiele für statisch typisierte Sprachen sind Eiffel, Sather-K, C++, Oberon-2, Java. Dynamisch typisierte Sprachen finden sich seltener in der Praxis, im wesentlichen ist hier Smalltalk zu nennen. Eine interessante (und wohl einzigartige) Zwitterposition nimmt Objective-C ein, da in dieser Sprache beide Varianten möglich und erlaubt sind. Neben der statischen Typisierung nach Art von C++ kennt Objective-C die dynamische Typisierung nach Art von Smalltalk über den generischen Objektreferenztyp `id`[2].

Die Art und Weise der Typisierung einer Sprache ist eng verflochten mit der Typprüfung und der Bindung von Methodenaufrufen. Diese Abhängigkeiten werden im weiteren Verlauf des Textes noch deutlich werden.

[2] id ist die Abkürzung für: Zeiger auf ein beliebiges Objekt

4.2.5 Typprüfung

Typisierung ist die Voraussetzung für die Typprüfung. Die Art und Weise der Typisierung bildet zusammen mit der Art und Weise der Typprüfung das *Typsystem* der Sprache.

Wie bereits angedeutet, kann man die Typisierung als Verhaltensfestlegung interpretieren. Die *Typprüfung* hat nun die Aufgabe festzustellen, ob das Programm gemäß dieser Festlegungen *korrekt* ist. Der Sinn und Zweck eines Typsystems ist die Erkennung fehlerfreier und die Ablehnung fehlerhafter Programme. Was mit den Begriffen Fehler und Korrektheit genau gemeint ist, wird weiter unten erläutert werden.

Auch bei der Typprüfung kann man im wesentlichen zwei Varianten unterscheiden:

Statische Typprüfung: Das Programm wird während oder vor der Übersetzung einer Typprüfung unterzogen. Inkorrekte Programme werden zurückgewiesen, die Übersetzung wird mit entsprechenden Fehlermeldungen abgebrochen.

Dynamische Typprüfung: Sämtliche Prüfung findet erst zur Laufzeit des Programms statt. Dies bedeutet, daß auch inkorrekte Programme übersetzt werden und zur Ausführung gelangen können.

Typsysteme können unterschieden werden nach Zeitpunkt (Tabelle 4.2) und Strenge der Prüfung (Tabelle 4.3) [Vit95].

Zeitpunkt der Typprüfung	Art des Typsystems
compile-time	statisches Typsystem
run-time	dynamisches Typsystem

Tabelle 4.2: Klassifikation nach Zeitpunkt

Bei der Strenge des Typsystems interessiert man sich dafür, wieviele Fehler im Rahmen des Typprüfung gefunden werden. Die Sprache C ist ein Beispiel für eine Sprache mit einem schwachen statischen Typsystem, Maschinensprache ist ein Beispiel für eine Sprache ohne Typsystem. Smalltalk besitzt ein streng dynamisches Typsystem.

Was ist nun ein von der Typprüfung erkennbarer Fehler bzw. ein inkorrektes Programm? Für objektorientierte Sprachen gilt:

Definition 4.2.1 *Es ist ein* Fehler*, einem Objekt eine Nachricht zu schicken, die es (unter Einbeziehung der Vererbung) nicht versteht. Weiterhin ist der versuchte Zugriff auf eine nichtexistente oder geschützte Instanzvariable ein Fehler. Ein* inkorrektes Programm *ist ein Programm, das einen in diesem Sinne fehlerhaften Nachrichtenausdruck enthält.*

Strenge des Typsystems	gefundene Fehler
strong	alle
weak	manche
untyped	keine

Tabelle 4.3: Klassifikation nach Strenge

Nicht gemeint sind zur Laufzeit auftretende Ausnahmen wie die Division durch Null oder die Überschreitung von Feldgrenzen. Übersetzer für hybride objektorientierte Sprachen wie C++ oder Oberon müssen darüberhinaus auch weiterhin die von imperativen Sprachen her bekannte Typprüfung durchführen.

Der wesentliche Unterschied zwischen statischer und dynamischer Typprüfung ist nun der, daß ein statisches Typsystem für jeden Nachrichtenausdruck *die Existenz mindestens einer adäquaten Implementierung garantiert* [Mey89], während ein dynamisches Typsystem dies nicht tut. Das Laufzeitsystem eines statisch typgeprüften Programms kann sich also darauf verlassen, daß *alle* Nachrichtenausdrücke auswertbar sind. Es existiert immer mindestens eine Implementierung. Das Laufzeitsystem einer dynamisch typgeprüften Sprache muß dagegen jeden einzelnen Nachrichtenausdruck zur Laufzeit prüfen. Es kann immer der Fall eintreten, daß einem Objekt eine Nachricht geschickt wird, die es nicht versteht. Oder anders ausgedrückt: Die Nachricht fordert die Ausführung einer nichtexistenten Methode an. Das Laufzeitsystem von Smalltalk signalisiert diesen Fall mit der Standardnachricht doesNotUnderstand: theMessage an das betroffene Objekt.

Der Berücksichtigung der Vererbung in der Typprüfung wird ausführlich in Abschnitt 4.2.6 diskutiert.

Nach Ravi Sethi [Set90] ist für eine Sprache eine statische Typprüfung durch ein strenges und leistungsfähiges Typsystem anzustreben. Dies gilt insbesondere für eingebette Systeme, an die ja besonders hohe Sicherheits- und Zuverlässigkeitsanforderungen gestellt werden.

4.2.6 Vererbung

Die *Vererbung* ist eine der zentralen Innovationen der objektorientierten Programmierung. Zusammen mit der Bindung (siehe Abschnitt 4.2.7) bildet sie die Grundlage für eine andere zentrale Eigenschaft, den Polymorphismus. Die überwiegende Mehrheit der objektorientierten Sprachen ist klassenbasiert und folgt letztlich dem klassischen Vererbungsmodell von Smalltalk. In diesem Modell werden vererbt:

- Variablen (Instanz- bzw. Klassenvariablen)
- Methoden (Instanz- bzw. Klassenmethoden)

Das Sprachkonstrukt für die Vererbung ist die *Unterklassenbildung.* Besonders zu beachten ist in diesem Modell, daß Variablen und Methoden auf unterschiedliche Weise behandelt werden: Während geerbte Variablen kopiert werden, erfolgt der Zugriff auf geerbte Methoden durch interne Verweise auf die Oberklasse. Der Code geerbter Methoden wird also *nicht* kopiert. Diese Verweisstruktur erst ermöglicht den objektorientierten Sprachen die effiziente *Wiederverwendung* von bestehendem Code.

Man kann bei der Bildung einer neuen Unterklasse folgende Phasen unterscheiden:

1. Die neue Klasse bekommt einen internen Verweis auf ihre Oberklasse (Superclass-Link).
2. Die Deklarationen der geerbten Variablen werden in die Definition der neuen Klasse einkopiert.
3. Neue Variablendeklarationen werden hinter die einkopierten Deklarationen angefügt. Eine Umdeklaration (Typänderung) oder gar Löschung bestehender Variablen ist nicht erlaubt[3].
4. Die Deklarationen der geerbten Methoden werden in die Definition der neuen Klasse einkopiert. Der eigentliche Code allerdings wird nicht kopiert, er kann über den Superclass-Link gefunden werden. Das Finden des Codes zur Laufzeit ist die Aufgabe der dynamischen Bindung (siehe Abschnitt 4.2.7).
5. Neue Methoden können deklariert und implementiert werden. Hat eine neue Methode dieselbe Signatur wie eine geerbte, so spricht man davon, daß die neue Methode die alte *überschreibt*[4]. Eine überschreibende Methode kann den überschriebenen Code aus der Oberklasse aufrufen. Dieses Konstrukt ermöglicht die Erweiterung bestehender Methoden um neue Funktionalität. Eine Löschung von Methoden ist nicht erlaubt.

Statisch typisierte Sprachen müssen nun Vererbung im Regelwerk ihres Typsystems berücksichtigen. Diese Sprachen beruhen auf der Gleichsetzung von Klasse und Typ. Eine Klasse T0 definiert den Typ T0. Ein Typ ist eine benannte Schnittstelle, eine Menge von Methodensignaturen. Zur Schnittstelle gehören ebenfalls die als öffentlich deklarierten Instanzvariablen. Bildet man nun eine Unterklasse T1 von T0, so ist T1 ein *Untertyp* von T0. T1 ist eine Spezialisierung des *Basistypen* T0. Man spricht daher auch von *Typerweiterung* [Wir88]. Es gilt die

Typerweiterungsregel: T1 ist ein Untertyp von T0, wenn die Schnittstelle von T1 eine Obermenge der Schnittstelle von T0 ist. Das Überschreiben von Methoden

[3] Eine gewisse Ausnahme findet sich in der Sprachdefinition von Java, hier ist eine Maskierung von Instanzvariablen möglich, wird aber nicht empfohlen.

[4] Sprachen wie C++ und Java kennen darüberhinaus das *Überladen* von Methoden. Dabei werden Methoden gleichen Namens in derselben Klasse anhand ihrer Signatur unterschieden. Das Überladen von Methoden hat nichts mit Vererbung zu tun.

ist erlaubt. Eine überschreibende Methode in T1 hat dieselbe Signatur wie die überschriebene aus T0, die Parametertypen dürfen also nicht geändert werden[5].

Vergleicht man nun Objekte vom Typ T1 mit Objekten vom Typ T0, so stellt man fest: Objekte vom Typ T1 verstehen alle Nachrichten, die auch T0-Objekte verstehen, und weiterhin existieren in T1-Objekten auch alle Instanzvariablen der T0-Objekte. Dies führt in Anlehnung an [WM96] zur

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

Man sagt auch, daß Objekte vom Typ T1 *kompatibel* zum Typ T0 sind. Entsprechendes gilt für Untertypen von T1. Ist z.B. T2 ein Untertyp von T1, so sind Objekte von T2 sowohl kompatibel zu T1 als auch zu T0. Wird auf der rechten Seite einer Zuweisung ein Objekt vom Basistyp T0 erwartet, so sind nach der Untertypregel Objekte der kompatiblen Typen T0, T1 und T2 erlaubt. In der Oberon-Sprechweise hätte ein T2-Objekt an dieser Stelle der Zuweisung den *statischen Typ* T0 und den *dynamischen Typ* T2 [Mös98]. Der dynamische Typ ist der tatsächliche Laufzeit-Typ des Objekts.

Oberon und andere statisch typisierte Sprachen begleiten die Untertypregel durch Typtest und Typzusicherung. Der *Typtest* beantwortet zur Laufzeit die Frage, ob ein Objekt o vom dynamischen Typ T (oder einer Erweiterung davon) ist. Der boolesche Ausdruck hat z.B. in Oberon die Form (o IS T) und in Java die Form (o instanceof T). Die *Typzusicherung (type guard)*, z.B. geschrieben (T)o, erfüllt eine Doppelfunktion: sie wandelt den statischen Typ von o in T um und prüft zur Laufzeit, ob der dynamische Typ von Objekt o tatsächlich T ist. Die Typzusicherung ist also eine Typumwandlung mit Laufzeit-Typprüfung. Die Typumwandlung zur Übersetzungszeit ist nur erlaubt, wenn T ein Untertyp des eigentlichen statischen Typs von o ist. Für die Laufzeit-Typprüfung ist vom Übersetzer entsprechender Code zu generieren, der eine Ausnahme auslöst, falls o nicht den dynamischen Typ T hat.

Die Typerweiterungsregel erlaubt das Überschreiben von Methoden. Objekte eines nach dieser Regel gebildeteten Untertyps dürfen nach der Untertypregel den Platz des Basistypen einnehmen, so daß zur Laufzeit der statische und der dynamische Typ durchaus verschieden sein können. Dieses muß bei der Aktivierung von Methoden beachtet werden:

[5] Eine Abschwächung dieser letzten Bedingung wird in der Literatur unter den Stichworten Kovarianz bzw. Kontravarianz diskutiert, siehe z.B. [Mös98, Kapitel 7.3].

Methodenauswahlregel: T sei eine direkte oder indirekte Unterklasse der Basisklasse T0. T überschreibt mit der Methode m' die geerbte Methode m aus T0. Wird nun gemäß der Untertypregel ein Objekt von T an einer Stelle übergeben, die ein Objekt vom Typ T0 erwartet, und wird die Ausführung der Methode angefordert, so *muß* die überschreibende Methode m' aus T aktiviert werden. Die in T0 implementierte Methode m darf nicht benutzt werden.

Das Problem für den Übersetzer ist hier, daß er zwar den statischen Typ T0 des Objekts kennt, der dynamische Typ des Objekts sich aber von Programmlauf zu Programmlauf ändern kann. Der Übersetzer muß also Code für die Aktivierung einer Methode erzeugen, die er zur Übersetzungszeit noch nicht genau kennt. Die auszuführende Methode ist abhängig vom dynamischen Typ des Objekts, nicht vom statischen Typ. Der einfache und direkt übersetzbare Prozeduraufruf muß daher ersetzt werden durch einen Vorgang, der zur Laufzeit und unter Berücksichtigung des dynamischen Typs des Objekts die richtige Methode findet (späte Bindung, siehe auch Abschnitt 4.2.7). Wie sich noch zeigen wird, hält sich dieser Aufwand für statisch typisierte Sprachen aber in Grenzen.

Zwischenbewertung

Die beschriebenen Regeln für statisch typisierte und typgeprüfte Sprachen schränken den Freiheitsgrad ein, den der Programmierer bei der Vererbung hat. Dafür kann die Typprüfung aber auch garantieren, daß zur Laufzeit des Programms für jede angeforderte Methode (mindestens) eine Implementierung existiert[6].

Dynamisch typisierte und typgeprüfte Sprachen wie Smalltalk kennen solche Einschränkungen nicht. Aufgrund der dynamischen Typisierung müssen Typerweiterungs- und Untertypregeln nicht aufgestellt werden. Die Methodenauswahlregel hingegen bleibt gültig. Der Programmierer kann prinzipiell jederzeit an jeder Stelle des Programms ein beliebiges Objekt übergeben. Dies kann allerdings zur Folge haben, daß Nachrichten an Objekte verschickt werden, für die es keine Implementierung gibt, mit entsprechenden Laufzeitfehlern bis hin zum vorzeitigen Programmabbruch.

Ein Vergleich der beiden Ansätze liefert folgende Aussagen:

Statisches Typsystem: Programme sind sicher im oben erklärten Sinn, sie sind effizient übersetzbar; aber eingeschränkte Flexibilität und Ausdruckskraft

Dynamisches Typsystem: Programme sind nicht sicher, eine effiziente Übersetzung ist u.U. nur eingeschränkt möglich; maximale Flexibilität und hohe Ausdruckskraft

Vitek [Vit95] bemerkt dazu: "The advantage of dynamically typed languages is that they will only reject programs which *are* incorrect and not programs which *may be* incorrect. This makes a real difference in expressive power."

[6]Die Typzusicherung stellt in diesem Sinne allerdings eine Sicherheitslücke dar, der man sich bewußt sein sollte. Macht man die Typzusicherung abhängig von einem erfolgreichen vorangehenden Typtest, kann die Sicherheit wiederhergestellt werden.

4.2.7 Bindung

Unter Bindung versteht man abstrakt die Zuordnung eines symbolischen Namens zu einer konkreten Speicherzelle im Kontext einer Umgebung. Eine *Umgebung* ist eine Funktion u, die Namen auf Speicherzellen abbildet. Man sagt, die Speicherzelle wird an den Namen *gebunden*. Speicherzellen[7] halten Werte. Der *Zustand* ist eine Funktion z, die Speicherzellen auf den in ihnen gespeicherten Wert abbildet [ASU86, Kapitel 7.1].

Beispiel 4.2.2 *Zuweisung, l-value, r-value*

```
i := i + 1;
```

Das Vorkommen der Variablen `i` auf der linken Seite der Zuweisung wird als *l-value* bezeichnet und verweist auf die Speicherzelle $u(\texttt{i})$, das Vorkommen auf der rechten Seite heißt *r-value* und meint den Wert $z(u(\texttt{i}))$.

Abhängig von der Umgebung kann derselbe Name auf unterschiedliche Speicherzellen verweisen. Die Verwendung von Namen wird geregelt durch Gültigkeits- und Sichtbarkeitsregeln (Scope und Extent) [WM96]. Diese Regeln sind wesentlicher Bestandteil der Sprachdefinition. Um die Verwendung von Namen zu erleichtern und um Namenskonflikte zu vermeiden, existieren oft getrennte Namensräume für globale und lokale Variablen, benutzerdefinierte Typen (Klassen), Prozeduren.

Ein wesentliches Merkmal objektorientierter Sprachen ist die dynamische Speicherverwaltung für Objekte. Objekte werden in der Regel erst zur Laufzeit erzeugt. Variablen können daher nicht direkt das Objekt bezeichnen (im Sinne einer Wertsemantik), sondern sie sind als *Verweis (Zeiger)* auf das Objekt zu interpretieren (Referenzsemantik). Jedes Objekt besitzt eine eindeutige Identität und ist durch eine eindeutige *Referenz* erreichbar. Die genaue Implementierung dieser Referenz ist abhängig von der jeweiligen Speicherverwaltung, letztlich handelt es sich aber um eine Speicheradresse. Eine Folge dieser Referenzsemantik ist, daß verschiedene (Objekt-) Variablen ein und dasselbe Objekt referenzieren können.

Die Mehrheit der objektorientierten Sprachen (z.B. C++, Objective-C, Oberon, Java) verwendet den Umgebungsbegriff der klassischen prozeduralen Sprachen wie C und Pascal. Die in der Literatur für prozedurale Sprachen beschriebenen Techniken zur Übersetzung von Umgebungen und Bindungen [ASU86, Wir96] sind daher grundsätzlich auch auf die genannten objektorientierten Sprachen anwendbar. Seltener orientieren sich die Designer objektorientierter Sprachen an den elaborierteren Umgebungskonzepten funktionaler Sprachen wie Scheme [AS85]. Beispiele sind Smalltalk mit seinem Block-Konzept[8] und das auf CommonLisp basierende CLOS.

[7] Speicherzelle meint hier jeden inhaltlich zusammengehörigen Speicherbereich, also auch Verbünde, Objekte, Code von Methoden etc.

[8] Ein Block ist vergleichbar mit einem lambda-Ausdruck.

Das in Abschnitt 4.2.6 beschriebene Vererbungsmodell hat nun zur Folge, daß bei Objekten die Bindung von Instanzvariablen anders zu behandeln ist als die Bindung von Nachrichten an Methoden.

Die *Bindung von Instanzvariablen* kann bereits zur Übersetzungszeit erfolgen, da die Struktur des Objekts hier bereits vollständig bekannt ist. Das Wissen um den statischen Typ des Objekts genügt.

Anders verhält es sich mit der *Bindung von Nachrichten an Methoden.* Diese Form der Bindung muß wesentlich den dynamischen Typ des Empfängers berücksichtigen, ist also ein Laufzeit-Vorgang. Der Übersetzer bereitet ihn jedoch vor durch Generierung von Code, der auf entsprechenden Laufzeit-Datenstrukturen operiert. Die statische Bindung der prozeduralen Sprachen wird ersetzt durch späte bzw. dynamische Bindung:

Statische Bindung: entspricht dem direkten Prozeduraufruf der imperativen Sprachen,

Späte Bindung: implementiert die eingeschränkte Methodenauswahlregel für statisch typisierte und typgeprüfte OO-Sprachen,

Dynamische Bindung: implementiert die uneingeschränkte Methodenauswahlregel für dynamisch typisierte und typgeprüfte OO-Sprachen.

Die Literatur verzichtet in ihrer Begriffsbildung oft auf eine Ausdifferenzierung der späten Bindung. Die Autoren unterscheiden nur zwischen statischer und dynamischer Bindung oder benutzen das Begriffspaar *early binding* und *late binding.* Alternative Bezeichnungen sind *Methodensuche*, *Message Passing*, *Message Dispatching*, *Method Lookup* oder *Messaging.*

Der für die Methodensuche zuständige Teil des Laufzeitsystems wird auch *Dispatcher* oder *Messager* genannt.

Syntax von Nachrichtenausdrücken

Auf der Ebene der Programmiersprache wird das Verschicken von Nachrichten an Objekte formuliert durch *Nachrichtenausdrücke (message expressions).* Ein Nachrichtenausdruck besteht immer aus dem *Empfänger (receiver, target)* und der eigentlichen *Nachricht (message).* Die Nachricht wiederum wird gebildet aus dem *Selektor (selector)* und den *Argumenten.*

Schreibweisen für Nachrichtenausdrücke sind die Schlüsselwort-Syntax (Smalltalk, Objective-C, Self, Omega) und die bekanntere Punkt-Syntax (C++, Java, Oberon).

Die Schreibweisen unterscheiden sich insbesondere bei Nachrichten mit mehreren Argumenten. Man vergleiche dazu die letzten Zeilen der Tabellen 4.4 und 4.5. Die Schlüsselwort-Schreibweise kann zu sehr gut lesbarem Code beitragen.

message expression	selector	arguments
`[anObject doThis]`	doThis	–
`[anObject doThis: arg1]`	doThis:	arg1
`[anObject do: arg1 with: arg2]`	do:with:	arg1, arg2

Tabelle 4.4: Schlüsselwort-Syntax (keyword message expression)[CN91]

message expression	selector	arguments
`anObject.doThis()`	doThis	–
`anObject.doThis(arg1)`	doThis_	arg1
`anObject.dowith(arg1, arg2)`	dowith__	arg1, arg2

Tabelle 4.5: Punkt-Syntax für Nachrichtenausdrücke (dot message expression)

Nachrichtenausdrücke können Werte zurückliefern und haben einen Typ. Der Wert ist das Ergebnis der Methode, die nach der Bindung der Nachricht ausgeführt wurde. Der Typ des gesamten Nachrichtenausdrucks ist der Ergebnistyp der Methode. Er kann zur Übersetzungszeit durch Zuordnung des Selektors zu der passenden Methode im statischen Typ des Empfänger-Objekts ermittelt werden.

4.3 Nichtklassische Objektmodelle

Nichtklassische Objektmodelle können im Raster der genannten Gestaltungsdimensionen folgendermaßen charakterisiert werden:

Objektabstraktion: Prototypen statt Klassen;

Vererbung: Die Vererbung von Instanzvariablen erfolgt uniform zu der der Instanzmethoden;

Vererbungsbeziehung: Im klassischen Objektmodell wird die Klassenhierarchie zur Übersetzungszeit eingefroren, nichtklassische Objektmodelle hingegen unterstützen *dynamische Vererbung*;

Bindung: Instanzvariablen werden wie Methoden vollständig zur Laufzeit gebunden.

Self ist die wichtigste nichtklassische objektorientierte Sprache, ein prototypenbasierter Alternativentwurf zu Smalltalk.

4.4 Kontext und Bewertung

In diesem Kapitel wurde ein Gestaltungsraum für das Design objektorientierter Programmiersprachen innerhalb des klassischen Objektmodells vorgestellt. Eine speziell für eingebettete Systeme zu entwickelnde Sprache sollte sich bei der Auswahl bestimmter Optionen an Gesichtspunkten wie Einfachheit, Sicherheit und effiziente Übersetzbarkeit orientieren. Für die Sprache ist dabei nicht eine hohe oder maximale Ausdruckskraft anzustreben, sondern eine ausreichende.

Insbesondere der Sicherheitsaspekt erzwingt eine Sprache mit statischem Typsystem und später Bindung, da Laufzeitfehler während der Methodensuche bereits zur Übersetzungszeit ausgeschlossen werden können. Späte Bindung ermöglicht eine effiziente Übersetzung durch VTBLs (siehe Kapitel 5.6.3). Die Methodensuche auch für späte Bindung kann hinsichtlich des Platzbedarfs optimiert werden, siehe dazu Kapitel 6.1. Die Sicherheit kann allgemein durch gezielten Einsatz der in Abschnitt 4.2.2 beschriebenen Kapselungsmechanismen gesteigert werden, deren Einhaltung ebenfalls schon zur Übersetzungszeit geprüft werden kann.

Auf die Nachteile der Mehrfachvererbung und Alternativen dazu wurde bereits in Kapitel 3 hingewiesen.

Echte Metaklassen erhöhen zwar die Flexibilität und Ausdruckskraft einer objektorientierten Sprache, was insbesondere die Programmierung stark benutzerorientierter und interaktiver Programme erleichtern kann, benötigen aber auch zusätzlichen Platz für Deskriptoren und Methodentabellen. Eingebettete Systeme sind in der Regel nicht interaktiv und Ressourcen sind beschränkt, so daß nach der Einschätzung des Autors auf Metaklassen verzichtet werden sollte. Bei Bedarf lassen sich Klassenmethoden- und variablen auch durch (modul-)globale Prozeduren und Variablen, wie sie beispielsweise in C vorhanden sind, nachbilden.

Praktische Erwägungen wie Verbreitungsgrad und hardwarenahe Programmierung lassen eine Erweiterung der Sprache C um objektorientierte Elemente sinnvoll erscheinen. Der Autor bevorzugt dabei eine Vorgehensweise, wie sie bei der Definition von Objective-C gewählt wurde, da die resultierende Sprachdefinition wesentlich kompakter und einfacher als die von C++ ist. Auf die in Objective-C mögliche dynamische Typisierung und dynamische Bindung sollte allerdings aus den schon genannten Gründen verzichtet werden.

Prototypenbasierte Sprachen wie Self oder NewtonScript sind für einen sicheren echtzeitfähigen Betrieb gänzlich ungeeignet, da sie alle in der Objektorientierung überhaupt denkbaren Entscheidungen in die Laufzeit des Programms verlagern. Dynamische Vererbung und dynamische Bindung sowohl von Nachrichten als auch von Variablen ist extrem zeitaufwendig und eben auch unsicher. Allerdings hat der Apple Newton PDA eindrucksvoll demonstriert, daß eine prototypenbasierte objektorientierte Sprache für diese interaktive benutzerorientierte Anwendung einen unerreicht hohen Grad an Komfort und Flexibilität in Programmierung und Gebrauch bieten kann.

4.5 Literaturhinweise

Grundlegende Konzepte von Programmiersprachen werden dargestellt in [Set90] und [AS85]. Einführungen in die objektorientierte Denkweise bringen [GR95, Goo96, Mey97]. Vergleichende Betrachtungen verschiedener objektorientierter Programmiersprachen finden sich in [Bud91, SO91, GHK95, Man97].

C++ wird beschrieben in [ES92], *Objective-C*[9] in [CN91, NeX93, Tho98].

Eine Objective-C nachgebildete Sprache mit Übersetzer, Laufzeitsystem, verschiedenen numerischen Lösern, einem Optimierungsframework und einer Komponentenbibliothek ist Bestandteil der Simulationsumgebung SMILE[10]. *SMILE* ist eine an der Technischen Universität Berlin und der GMD Forschungszentrum Informationstechnik GmbH entwickelte dynamische Simulationsumgebung zur Simulation komplexer energiewandelnder Systeme.

Oberon wird beschrieben in [Wir88] und *Oberon-2* in [Mös98]. Franz [Fra95] hat Oberon um Protokolle erweitert.

Eine Darstellung von *Java* findet sich in [AG98] und in [Fla97].

Sather-K ist didaktische Grundlage der Vorlesungen über Informatik [Goo96].

Die klassische Literatur zu *Smalltalk* sind die Bücher von Goldberg et al. [GR83, Gol84]. Ingalls et al. beschreiben in [IKM+97] eine freie Implementierung namens Squeak[11] von Smalltalk-80.

Prototypenbasierte Sprachen sind *Self*[12] [US91, SU95, ABC+95], *NewtonScript* [App96], *Omega*[13] [Bla94] und *Kevo* [Tai93].

Ausführliche und tiefgründige Anmerkungen zur *Vererbung* finden sich in [Tai93, Tai96].

Verschiedene Formen von *Polymorphismus* werden unterschieden in [CW85].

Leistungsfähige statische Typsysteme für objektorientierte Sprachen sind ein aktueller Gegenstand der Forschung [PS94]. Meyer gibt in [Mey89, Mey97] eine Einführung in das Typsystem von *Eiffel*. Interessierte Leser sollten sich auch die Typsysteme von Sather-K [Goo96] und Omega [Bla94] anschauen. Thorup macht in [Tho97] einen Vorschlag zur Erweiterung von Java um generische Typen. Zur Unterscheidung von Klasse und Typ siehe [CHC90, GHJV95]. *Cecil*[14] ist eine innovative objektorientierte Sprache mit getrennten Typ- und Klassenhierarchien[Cha95].

[9] http://developer.apple.com/techpubs/macosx/ObjectiveC/index.html
[10] http://buran.fb10.tu-berlin.de/Energietechnik/EVT_KT/smile/
[11] http://www.squeak.org
[12] http://www.cs.ucsb.edu/oocsb/self/papers/papers.html
[13] http://infosoft.soft.uni-linz.ac.at/Info/OmegaReport/OmegaReport.html
[14] http://www.cs.washington.edu/research/projects/cecil/www/Papers/papers.html

Kapitel 5

Grundlagen der Übersetzung

Dieses Kapitel widmet sich den Grundlagen der Übersetzung objektorientierter Sprachen. Auf technischer Ebene besteht eine enge Verwandtschaft zu imperativen Programmiersprachen. Diese wird durch Reduktion objektorientierter Sprachmerkmale auf imperative Merkmale ausgenutzt. Nach einer ersten Einführung werden primitive und Referenzdatentypen erläutert. Es folgen Techniken zur Repräsentation von Objekten, Methoden und Klassen durch Verbünde, Reihungen, Zeiger und Prozeduren. Die Vererbung von Instanzvariablen und -methoden ist ein weiteres Thema des Kapitels.

5.1 Einführung

Diese Einführung möchte einen ersten Eindruck davon vermitteln, wie objektorientierte Programme übersetzt werden können. Die Beispiele sind in der aus Kapitel 3.1 bekannten hybriden Sprache notiert.

Beispiel 5.1.1 *Eine Klasse für zweidimensionale Punkte*

```
class Point2D extends Object
{
    int x := 0;
    int y := 0;

    method (Point2D) setX:(int )x andY:(int )y
    {
        self.x := x;
        self.y := y;
        return self;
    }
}
```

```
Point2D  p1 := new Point2D;

[p1 setX: 1 andY: 2];
```

In Beispiel 5.1.1 wird die Klasse Point2D mit zwei Instanzvariablen x, y für die Koordinaten und einer Instanzmethode setX:andY: definiert. Die Methode nimmt zwei Integer-Argumente als neue Koordinaten entgegen und liefert eine Referenz auf das modifizierte Objekt zurück.

Das sich anschließende Programmfragment erzeugt zuerst ein Objekt der Klasse mit den Standardwerten $(0, 0)$ und setzt danach die Koordinaten des Punkt-Objekts auf $(1, 2)$.

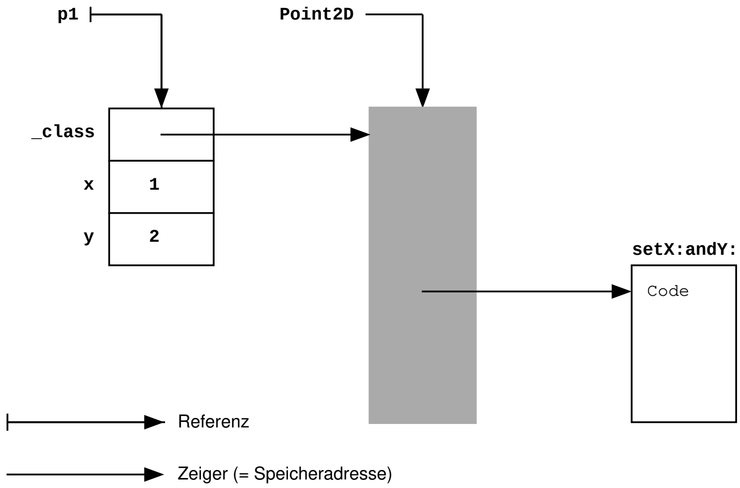

Abbildung 5.1: Objekt und Klasse Point2D

Die Übersetzung und Ausführung des Beispiels führt zu der in Abbildung 5.1 dargestellten Laufzeitsituation, die wie folgt charakterisiert werden kann:

Speichermodell. Klassen und Methoden werden zur Übersetzungszeit in eine interne Darstellung überführt und an einer festen Stelle im Speicher hinterlegt. Auf sie kann durch *Zeiger* im Sinne von Speicheradressen zugegriffen werden. Objekte hingegen werden zur Laufzeit dynamisch auf dem Heap alloziert. Da Objekte unter Umständen durch die automatische Speicherverwaltung innerhalb des Heaps verschoben werden können, erfolgt die Adressierung von Objekten konzeptionell durch *Referenzen.* Ein Stack dient zur Übergabe von Argumenten und zur Ablage der Rücksprungadresse beim Aufruf von Methoden.

Das Objekt. Die Referenzvariable p1 verweist auf das Punkt-Objekt. Das Objekt ist ein reiner Zustandsspeicher. Es besitzt an erster Stelle eine interne zusätzliche Instanzvariable _class, einen Zeiger auf die Klassenbeschreibung. Es folgen die Instanzvariablen x und y. Wenn man eine Wortbreite von 32 Bit für Zeiger und Integer voraussetzt, belegt das Objekt genau 12 Byte im Speicher.

Die Klasse. Die textuelle Beschreibung der Klasse wird in eine interne Darstellung überführt, die hier nicht im Detail besprochen werden soll und daher als graue Box gezeichnet ist. Wesentlicher Bestandteil einer Klassenrepräsentation ist eine Methodentabelle, die Zeiger auf den Code der definierten Methoden enthält. Im Beispiel hat diese Tabelle einen Eintrag, den Zeiger auf den Code für die Methode setX:andY:.

Die Methode. Eine Methode ist vergleichbar mit einer Prozedur oder Funktion in imperativen Programmiersprachen.

In der Literatur wird oft allgemein davon gesprochen, daß Objekte Zustand und Verhalten kapseln. Diese Aussage kann nun dahingehend konkretisiert werden, daß Objekte reine Zustandsspeicher sind, während Klassen das gemeinsame Verhalten bereitstellen. Die Verbindung zwischen Objekt und Klasse wird hergestellt über den internen Zeiger _class.

Das nächste Beispiel demonstriert Vererbung von Instanzvariablen und -methoden.

Beispiel 5.1.2 *Eine Unterklasse für dreidimensionale Punkte*

```
class Point3D extends Point2D
{
    int z := 0;

    method (Point3D) setX:(int )x andY:(int )y andZ:(int )z
    {
        [self setX: x andY: y];
        self.z := z;
        return self;
    }
}

Point3D  p2 := new Point3D;

[p2 setX: 1 andY: 2 andZ: 3];
```

Die Unterklasse Point3D erbt alle Merkmale von Point2D und ergänzt eine Instanzvariable z für die dritte Dimension und eine Instanzmethode setX:andY:andZ: zum Setzen der dreidimensionalen Koordinaten. Den Prinzipien des Information Hiding folgend werden die x und y Koordinaten nicht direkt modifiziert, sondern durch Aufruf der ererbten Methode setX:andY:. Übersetzung und Ausführung des Beispiels führen zu der in Abbildung 5.2 dargestellten Laufzeitsituation.

Folgende Aspekte sind hervorzuheben:

Das Objekt. Das durch p2 referenzierte Objekt ist strukturell eine echte Erweiterung des Point2D-Objekts. Im Point3D-Objekt erscheinen zuerst der interne Zeiger

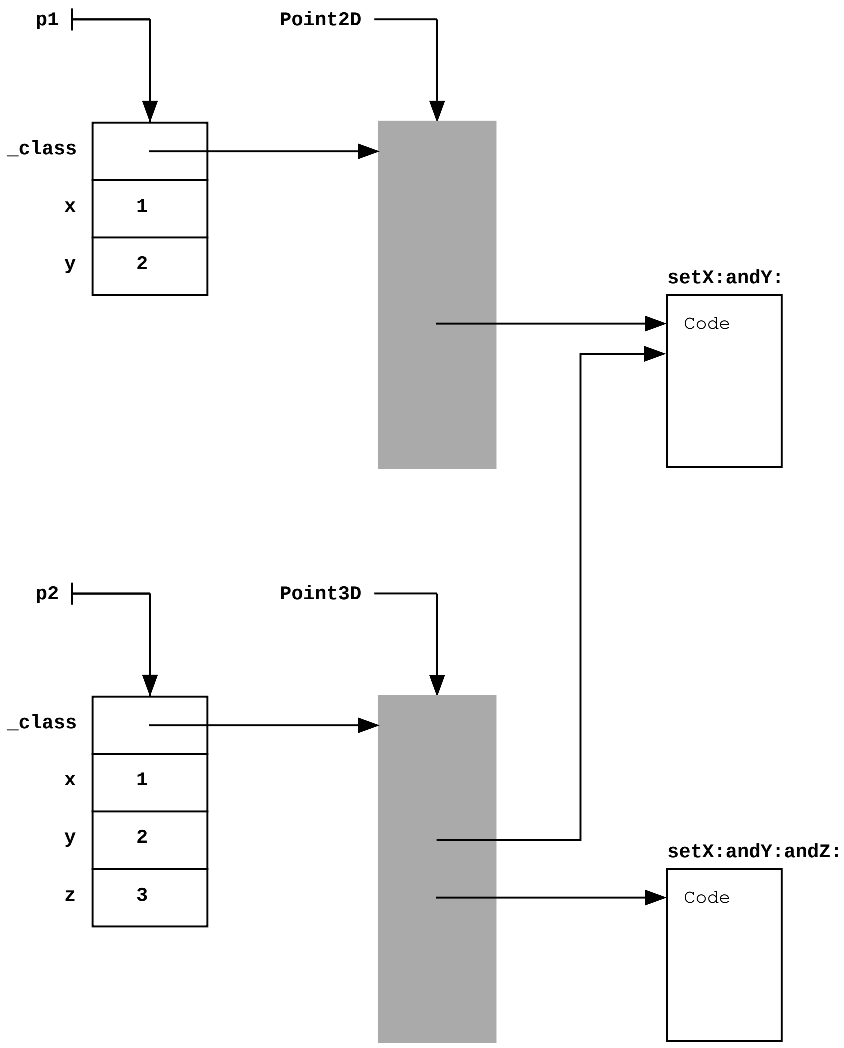

Abbildung 5.2: Point3D als Unterklasse von Point2D

_class, gefolgt von den ererbten Instanzvariablen x und y. Die Objekte sind bis dahin im Aufbau identisch. Es folgt dann die neu deklarierte Instanzvariable z. Das Objekt belegt 16 Bytes im Speicher.

Klasse und Methoden. Gleiches gilt für die interne Repräsentation der Klasse, insbesondere für die Methodentabelle. Die ererbte Methode setX:andY: wird nicht neu übersetzt, es wird nur der entsprechende Zeiger übernommen. Die Methodentabelle wird dann ergänzt um einen Zeiger auf den Code der neuen Methode setX:andY:andZ:.

Das Beispiel verdeutlicht, wie sich die Vererbung von Instanzvariablen im strukturellen Aufbau der Objekte spiegelt und entsprechend die Vererbung von Instanzmethoden in der Zeigerstruktur der Methodentabellen.

Das letzte Beispiel dieser Einführung beleuchtet den Unterschied zwischen statischem und dynamischem Typ einer Referenzvariablen.

Beispiel 5.1.3 *Statischer und dynamischer Typ*

```
Point2D  p2;
Point3D  p3 := new Point3D;

p2 := p3;    // erlaubt

[p2 setX: 10 andY: 20 andZ: 30];    // verboten!
```

Die Variable p2 wird deklariert als Referenz auf ein Point2D-Objekt. Die Variable p3 wird deklariert als Referenz auf ein Point3D-Objekt, welches auch sofort erzeugt wird. Der statische Typ von p2 ist Point2D, der von p3 ist Point3D.

Die folgende Zuweisung ist aufgrund der Untertypregel erlaubt. Sie läßt sich jetzt anschaulich mit Abbildung 5.2 begründen. Ein Point3D-Objekt kann strukturell gesehen als ein Point2D-Objekt erscheinen. Die Zuweisung kopiert nur die Referenz, nicht das Objekt. Der statische Typ von p2 ist weiterhin Point2D, der dynamische Typ ist aber nun Point3D, da p2 und p3 jetzt auf dasselbe Point3D-Objekt verweisen. Der dynamische Typ kann zur Laufzeit über den Zeiger _class ermittelt werden.

Der Nachrichtenausdruck wird aber dennoch vom Typsystem zurückgewiesen, da bei der Überprüfung des Ausdrucks nur der statische Typ von p2 betrachtet werden kann. Point2D kennt die Nachricht setX:andY:andZ: nicht. Eine dynamisch typisierte Sprache hingegen erledigt die Typprüfung zur Laufzeit und könnte daher den Nachrichtenausdruck erfolgreich auswerten.

5.2 Primitive Datentypen

Auch objektorientierte Sprachen müssen mit numerischen Werten und Zeichen umgehen können. Hybride Sprachen besitzen daher einen Satz primitiver Datentypen, die exemplarisch in Tabelle 5.1 aufgelistet sind. Reine objektorientierte Sprachen kapseln primitive Datentypen in entsprechenden Klassen.

Typ	Inhalt	Standard	Größe
boolean	`true` oder `false`	`false`	1 Bit
char	Unicode-Zeichen	\u0000	16 Bit
byte	Integer mit Vorzeichen	0	8 Bit
short	Integer mit Vorzeichen	0	16 Bit
int	Integer mit Vorzeichen	0	32 Bit
long	Integer mit Vorzeichen	0	64 Bit
float	IEEE 754 Fließkommazahl	0.0	32 Bit
double	IEEE 754 Fließkommazahl	0.0	64 Bit

Tabelle 5.1: Primitive Datentypen der Sprache Java

5.3 Referenztypen

Variablen, die in irgendeiner Form Objekte bezeichnen sollen, speichern nicht das Objekt selbst, sondern eine *Referenz* auf das Objekt. Dafür gibt es im wesentlichen drei Gründe:

1. Objekte werden erst zur Laufzeit auf dem Heap alloziert;
2. Es kommt häufig vor, daß mehrere Variablen ein und dasselbe Objekt referenzieren müssen;
3. Objektvariablen müssen aufgrund der Untertypregel in der Lage sein, Objekte verschiedener physischer Größe zu bezeichnen (siehe Beispiel 5.1.3).

Typ	Inhalt	Standard	Größe
id	Referenz auf Objekt	`nil`	32 Bit

Tabelle 5.2: Objective-C Referenztyp

Der Referenzdatentyp hat eine konstante Grösse, die abhängig von den Adressierungsmöglichkeiten der Zielmaschine gewählt werden kann (Tabelle 5.2). Er ist insofern einem Zeiger nicht unähnlich.

Beispiel 5.3.1 *Verwendung von Referenzvariablen in Objekten*

```
class Rectangle extends Object
{
    Point2D  origin := new Point2D;
    Point2D  corner := new Point2D;

    method (Point2D) getCorner
    {
       return corner;
    }
}

Rectangle  r := new Rectangle;

[[r getCorner] setX: 10 andY: 10];
```

Übersetzung und Ausführung des Beispiels führen zu der in Abbildung 5.3 dargestellten Laufzeitsituation. Besonders zu beachten ist, daß nicht nur ein Rectangle-Objekt erzeugt wird, sondern auch zwei Point2D-Objekte.

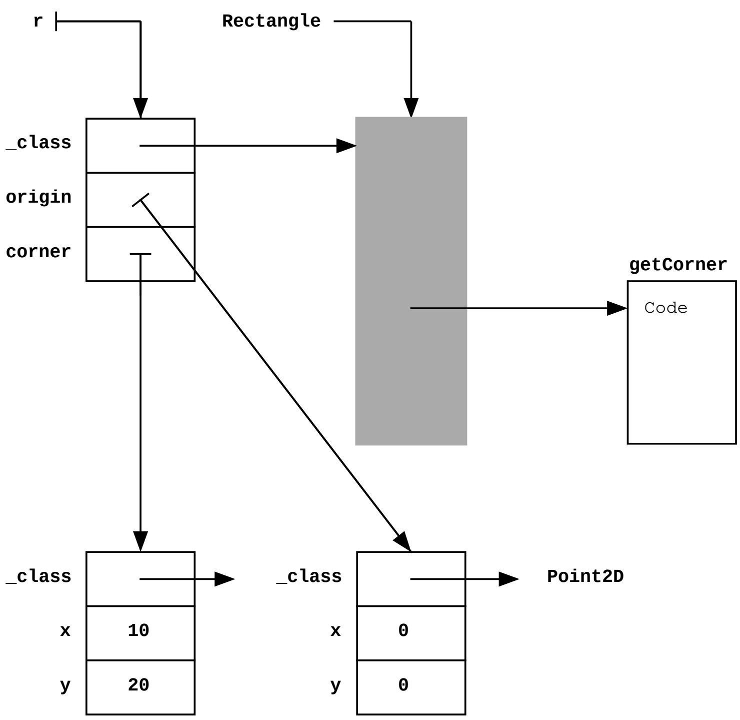

Abbildung 5.3: Objektreferenzen als Instanzvariablen

Der konkrete Inhalt einer Referenzvariablen ist je nach gewählter Implementierungstechnik (Abbildung 5.4):

- ein direkter Zeiger (direct pointer),
- ein Handle (pointer to master pointer),
- ein Index in eine Objekttabelle bestehend aus Zeigern.

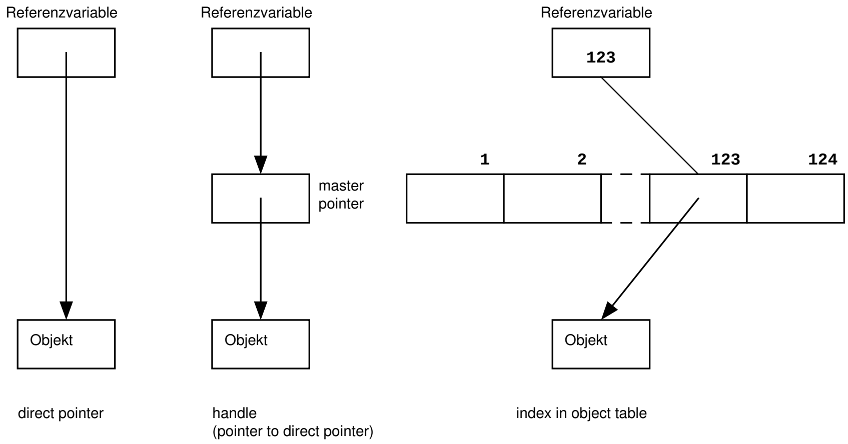

Abbildung 5.4: Implementierungen von Objektreferenzen [Bla94]

Mit einem direkten Zeiger benötigt man nur eine Indirektion zum Zugriff auf das Objekt, die anderen Techniken benötigen zwei Indirektionen. Daher verwenden moderne Implementierungen bevorzugt den direkten Zeiger. Frühe Smalltalk-Implementierungen basierten auf einer Objekttabelle mit 15 Bit breitem Index, so daß die Anzahl möglicher Objekte auf 32768 beschränkt war.

Die Festlegung von Objektvariablen als Referenzen hat nach [Bla94] folgende Konsequenzen:

- Alle Referenzen haben dieselbe Grösse, damit haben auch alle Objektvariablen dieselbe Grösse. Auf diese Weise können Objekte verschiedener Klassen an dieselbe Variable zugewiesen werden, die Grösse des Objekts spielt keine Rolle.
- Eine Zuweisung $x := y$ plaziert nicht eine Kopie des Objekts y in die Variable x. Nur die Referenz wird kopiert. Danach verweisen x und y auf dasselbe Objekt.
- Objekte haben keine Namen. Sie sind anonym. Es ist daher eigentlich nicht korrekt von dem "Objekt x" zu sprechen, da die Variable x zu einem späteren Zeitpunkt ein anderes Objekt referenzieren kann.
- Objekte können selbst Objektvariablen enthalten, siehe Beispiel 5.3.1. Auf diese Weise können zur Laufzeit beliebige und komplexe Netzwerke aus Objektbeziehungen aufgebaut werden.

In Java und Smalltalk sind Objektvariablen immer Referenzen, andere Sprachen wie C++, Oberon, Objective-C bieten mehr Freiheiten an. Hier hat der Programmierer zusätzlich die Möglichkeit, eine Objektvariable zu deklarieren, die das Objekt wirklich enthält. Dann ändert sich die Semantik der Zuweisung zu einer Kopiersemantik (statische Allokation, siehe Abschnitt 7.2).

5.4 Objekte

Objekte sind reine Zustandsspeicher. Technisch gesehen entsprechen sie direkt den Verbünden (struct, record) der imperativen Sprachen und können in derselben Weise übersetzt werden.

5.4.1 Instanzvariablen

Ein Objekt belegt Speicher entsprechend der Anzahl und Bitbreite seiner Instanzvariablen. Diese können allgemein in drei Gruppen (Abbildung 5.5) geteilt werden:

- die systemdefinierten Instanzvariablen,
- die benutzerdefinierten Instanzvariablen,
- die indizierten Instanzvariablen.

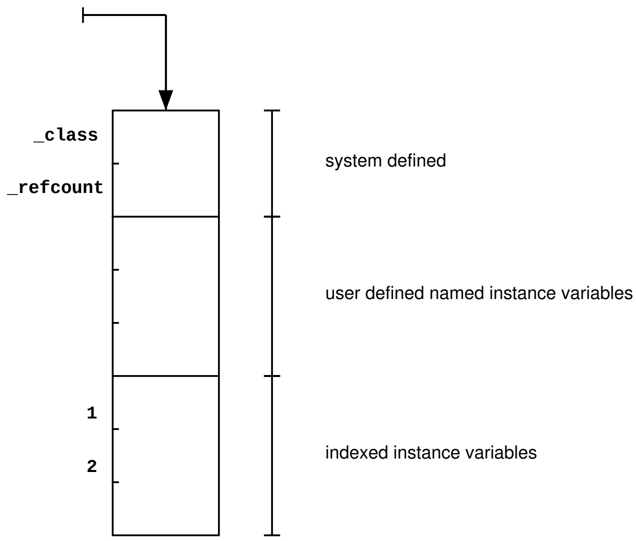

Abbildung 5.5: Instanzvariablen eines Objekts

Jedes Objekt besitzt mindestens die *systemdefinierten Instanzvariablen*. Diese werden entweder in der Wurzelklasse aller Klassen Object deklariert und vererbt oder direkt vom Übersetzer erzeugt, wie es in C++ der Fall ist. Unverzichtbares Mitglied dieser Gruppe ist der Zeiger _class, der das Objekt mit der internen Darstellung seiner Klasse verbindet. Die Methodensuche zur Laufzeit bedient sich dieses Zeigers um Zugriff auf die Methodentabelle zu erlangen. Abhängig von den systemspezifischen Erfordernissen können weitere Variablen deklariert werden. Ein Beispiel wäre der Zähler _refcount, der im Dienste einer automatischen Speicherverwaltung die Anzahl der Referenzen auf sein Objekt zählt.

Die *benutzerdefinierten Instanzvariablen* werden vom Programmierer in der Klasse des Objekts deklariert. Zu ihnen zählen natürlich auch die aus direkten und indirekten Oberklassen ererbten Variablen.

Die *indizierten Instanzvariablen* bilden die optionale letzte Gruppe. Sprachen wie Smalltalk und Objective-C verwenden sie zur Implementierung von Reihungen (Arrays). Der Zugriff erfolgt über spezielle Methoden wie **at:** und **at:put:** unter Angabe eines Index.

5.4.2 Vererbung von Instanzvariablen

Ein Schema für die Vererbung von Instanzvariablen im klassischen Objektmodell wurde bereits in Abschnitt 4.2.6 beschrieben. Die folgenden Ausführungen sollen verdeutlichen, warum dieses Schema die technische Voraussetzung für die Untertypregel in statisch typisierten Sprachen ist.

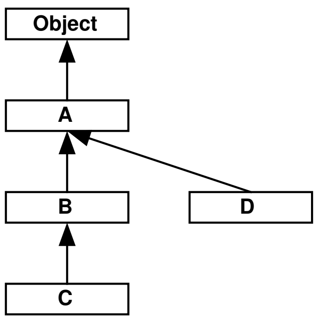

Abbildung 5.6: Klassenhierarchie für Beispiel 5.4.1 und 5.4.2

Beispiel 5.4.1

```
class A extends Object
{
    int a1;
    int a2;
}
```

```
class B extends A
{
    int b1;
    int b2;
}

class C extends B
{
    int c1;
    int c2;
}
```

Das Beispiel ist eine programmatische Umsetzung der Klassenhierarchie aus Abbildung 5.6. Klasse A ist eine Unterklasse der Wurzelklasse Object, B ist Unterklasse von A, C ist Unterklasse von B. Jede Klasse deklariert zwei Instanzvariablen. Abbildung 5.7 zeigt von links nach rechts den Aufbau dreier Objekte der Klassen A, B und C.

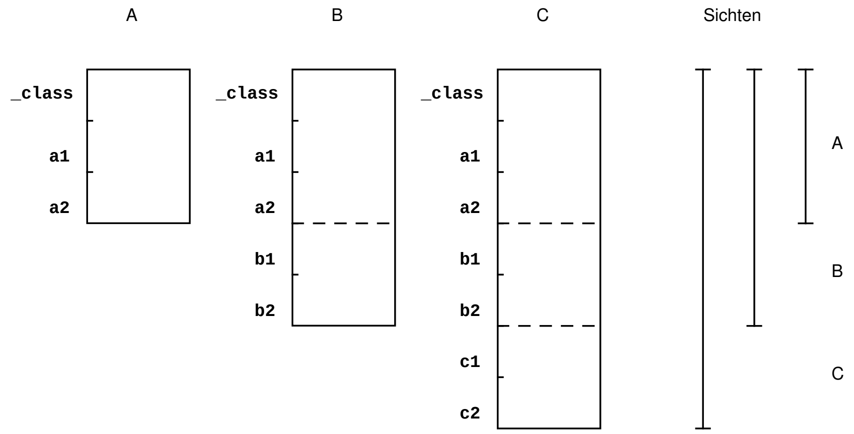

Abbildung 5.7: Objekte der Klassen A, B, C

In der Struktur der Objekte spiegelt sich die Klassenhierarchie:

Klasse	Hierarchie	Aufbau des Objekts
A	$O \rightarrow A$	zuerst die systemdefinierte Instanzvariable, dann die in A deklarierten Variablen a1, a2
B	$O \rightarrow A \rightarrow B$	die systemdefinierte Instanzvariable und die in A deklarierten, erst dann die in B deklarierten Variablen b1, b2
C	$O \rightarrow A \rightarrow B \rightarrow C$	Übernahme der Struktur von B, gefolgt von den eigenen Instanzvariablen c1, c2

Als direkte Konsequenz dieses additiven Aufbaus entlang der Klassenhierarchie *(prefixing)* ist es nun möglich, verschiedene *Sichten* auf ein Objekt zu erlauben. Ein B-Objekt kann als ein A-Objekt erscheinen, und ein C-Objekt kann auf ein A-Objekt oder auf ein B-Objekt reduziert werden.

Beispiel 5.4.2 *Ein Gegenbeispiel*

```
class D extends A
{
    int d1;
    int d2;
    int d3;
}
```

Auch die Klasse D ist eine Unterklasse von A, deklariert aber drei neue Instanzvariablen (Abbildung 5.8).

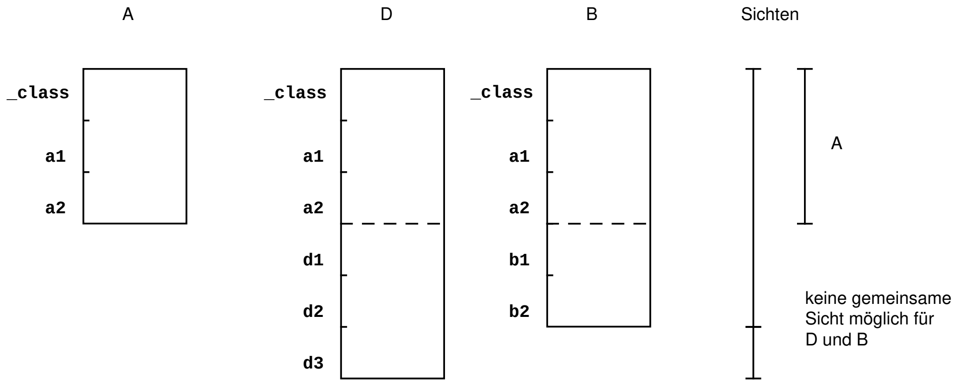

Abbildung 5.8: D-Objekt im Vergleich zu B-Objekt

Die Klassen D und B befinden sich in der Hierarchie auf derselben Ebene. Ihre strukturelle Ähnlichkeit beschränkt sich aber auf die von A geerbte Struktur. Objekte der Klassen B und D können daher als A-Objekte erscheinen, eine weitere gemeinsame Sicht ist aber nicht möglich. Ein D-Objekt kann nicht als B-Objekt gesehen werden und umgekehrt.

Die beschriebene Möglichkeit, eine reduzierte Sicht auf ein Objekt gemäß der Klassenhierarchie zu erlauben, ist die Grundvoraussetzung der Untertypregel für Sprachen mit statischem Typsystem.

5.4.3 Zugriff auf Instanzvariablen

Ein Übersetzer wird den hierarchischen Aufbau der Objekte durch geeignete Generierung von Offsets ausnutzen. Diese Offsets sind in der Symboltabelle den Einträgen für die jeweiligen Instanzvariablen beizufügen.

Man beachte in den Tabellen 5.3 und 5.4 insbesondere auch die identischen Offsets für b1 ff. und d1 ff.

Beispiel 5.4.3

```
C   einObjekt := new C;

int dummy := einObjekt.c1;
```

Im obigen Fragment wäre der Zugriff auf c1 zu übersetzen als:

> dereferenziere die Objektreferenz *einObjekt*;
>
> addiere auf die Speicheradresse den Offset für c1 in Klasse C;
> lade den Inhalt der Adresse.

Klasse	Instanzvariable	Offset
O, A, B, C	_class	0
A, B, C	a1	4
	a2	8
B, C	b1	12
	b2	16
C	c1	20
	c2	24

Tabelle 5.3: Offsets für Klassen A, B und C

Klasse	Instanzvariable	Offset
O, A, D	_class	0
A, D	a1	4
	a2	8
D	d1	12
	d2	16
	d3	20

Tabelle 5.4: Offsets für Klassen A und D

Kosten: Eine Indirektion[1] zur Auflösung der Referenz und eine Addition. Die Kosten sind konstant.

5.5 Methoden

Methoden sind Bestandteil einer Klassendefinition. Sie implementieren das gemeinsame Verhalten aller Objekte der Klasse und modifizieren dabei insbesondere den Zustand. Methoden sind die Prozeduren und Funktionen der objektorientierten Programmierung.

Methoden werden aktiviert durch die Auswertung eines Nachrichtenausdrucks. Der Kontext der Methode ist dabei das Objekt, welches die Nachricht empfängt. Es wurde allerdings bereits ausführlich dargestellt, daß Objekte nur Zustandsspeicher sind. Damit ergeben sich zwei Fragen:

1. Wenn Methoden nicht physikalischer Bestandteil des Objekts sind, wo und wie sind sie dann zu finden?
2. Wie bekommt die Methode Zugang zu dem konkreten Objekt, welches die Nachricht empfangen hat?

Die Beantwortung der ersten Frage bleibt dem Abschnitt 5.6 vorbehalten. Hier sind enge Verflechtungen zwischen der Repräsentation von Klassen und ihren Methodentabellen und der Methodensuche zu berücksichtigen.

Die zweite Frage soll in diesem Abschnitt geklärt werden. Methoden werden in der gleichen Weise übersetzt wie Prozeduren und Funktionen in imperativen Sprachen. Während der Übersetzung werden die Signaturen jedoch ergänzt um einen für den Programmierer unsichtbaren Parameter **self**. self ist eine Referenzvariable und stellt in der Methode die Verbindung zu dem Objekt her[2].

Beispiel 5.5.1

```
class A extends Object
{
    int a1;
    int a2;

    method (int) summe
    {
```

[1] Je nach Implementierung der Referenz auch zwei Indirektionen.

[2] Diese Variable heißt self bei Smalltalk und Objective-C, *this* bei Java und C++. Oberon-Programmierer müssen sie per Hand deklarieren.

```
            return a1 + a2;
        }
    }
```

Die obige Methode summe ohne Parameter wird übersetzt in die folgende Funktion A_summe mit einem Parameter:

```
        function A_summe(self: A) : int
        {
            return self.a1 + self.a2;
        }
```

Bei der Transformation einer Methode in eine Funktion sind also folgende Schritte erforderlich:

- Die Liste der Parameter ist zu ergänzen um den Parameter `self`. Der Typ ist die Klasse, in der die Methode definiert wird.
- Sofern noch nicht durch den Programmierer geschehen, ist der Zugriff auf eine Instanzvariable `instvar` zu ersetzen durch `self.instvar`.
- Ist der Namensraum für Funktionen flach, so ist im Funktionsnamen auf geeignete Weise die Klasse zu codieren, da Methoden in verschiedenen Klassen ja denselben Namen haben können.

Nach dieser rein textuellen Transformation kann die Funktion in Code für die Zielmaschine übersetzt werden. Ein Zeiger auf diesen Code wird in eine ebenfalls durch den Übersetzer zu generierende Methodentabelle eingetragen.

Wilhelm und Maurer beschreiben in [WM96, Kapitel 5.2] die Übersetzung von C++ Methoden in äquivalente Funktionen.

5.6 Klassen

In den obigen Abschnitten wurde erläutert, wie Methoden in äquivalente Funktionen übersetzt werden können. Weiterhin wurde gezeigt, wie ein Übersetzer aus der Klassendefinition eine physische Repräsentation der Objekte der Klasse gewinnen kann. Die Darstellung soll nun abgerundet werden durch die Beantwortung der ersten Frage aus Abschnitt 5.5.

Die durch den Programmierer vorgegebene textuelle Klassendefinition wird vom Übersetzer in einen *Klassendeskriptor* transformiert und im Speicher hinterlegt. Der interne Zeiger _class eines Objekts verweist zur Laufzeit auf genau diesen Deskriptor der Klasse. Im folgenden wird angenommen, daß Klassendeskriptoren statisch alloziert werden und daher nicht durch eine automatische Speicherbereinigung verschoben oder gar gelöscht werden können.

Der wesentliche Bestandteil eines Klassendeskriptors ist eine *Methodentabelle (Dispatch Table)*, die für jede Instanzmethode einen Zeiger auf den übersetzen Code der Methode enthält. Nachrichtenausdrücke werden in Code übersetzt, der sich zur Laufzeit dieser Methodentabelle bedient, um gemäß des dynamischen Typs des Empfängerobjekts die richtige Implementierung für die Nachricht zu finden. Deskriptoren für Sprachen mit statischem Typsystem sind dabei einfacher im Aufbau als Deskriptoren für Sprachen mit dynamischem Typsystem.

Der Klassendeskriptor unterstützt das Laufzeitsystem weiterhin durch *Metainformation* über die Objekte der Klasse und die Klasse selbst.

5.6.1 Metainformation

Der Umfang der im Klassendeskriptor hinterlegten Metainformation ist abhängig von den Bedürfnissen der konkreten Programmiersprache. Die C++ Deskriptoren heissen *Virtual Function Table (VTBL)*, bestehen nur aus der Methodentabelle und speichern überhaupt keine Metainformation. Java, Objective-C, Smalltalk hingegen besitzen sehr ausführliche Deskriptoren, denen zur Laufzeit viele Eigenschaften der Objekte und der Klasse entnommen werden können.

Ein Deskriptor kann beispielsweise folgende Information bereitstellen:

Grösse eines Objekts: Soll zur Laufzeit ein neues Objekt der Klasse erzeugt werden, kann von der Speicherverwaltung ein entsprechend grosser Bereich angefordert werden.

Typen der Instanzvariablen: Diese Angaben sind nützlich für die automatische Speicherbereinigung. Sie erfährt hier, welche Komponenten eines Objekts Referenzen auf andere Objekte beinhalten.

Verweis auf die Oberklasse: Ordnet die Klasse in die Klassenhierarchie ein. Grundlage für Typtest und Typzusicherung. Kann durch ein *display* weiter optimiert werden [App98, Kapitel 14.4].

Name der Klasse: Ermöglicht beispielsweise einem Debugger die Ausgabe einer lesbaren textuellen Repräsentation eines Objekts der Klasse.

5.6.2 Methodentabellen und Methodensuche

Die Methodentabelle stellt die Verbindung her zwischen dem Objekt und seinen Instanzmethoden. Während ein polymorpher Nachrichtenausdruck bei später Bindung direkt am Ort des Aufrufs *(call-site)* in Code übersetzt werden kann, wird für die Methodensuche der dynamischen Bindung ein *Dispatcher* benötigt, der Bestandteil des Laufzeitsystems der Sprache ist. An der call-site wird dann ein Aufruf des Dispatchers eincompiliert.

5.6.3 Deskriptor für späte Bindung

Die in Abschnitt 4.2.6 beschriebene eingeschränkte Methodenauswahlregel für statische Typsysteme ermöglicht einen additiven Aufbau der Methodentabellen, der analog zum Aufbau der Objekte erfolgt (Abschnitt 5.4.2). Der Übersetzer muß nun allerdings berücksichtigen, daß Methoden einer Klasse geerbte Methoden gleicher Signatur überschreiben dürfen.

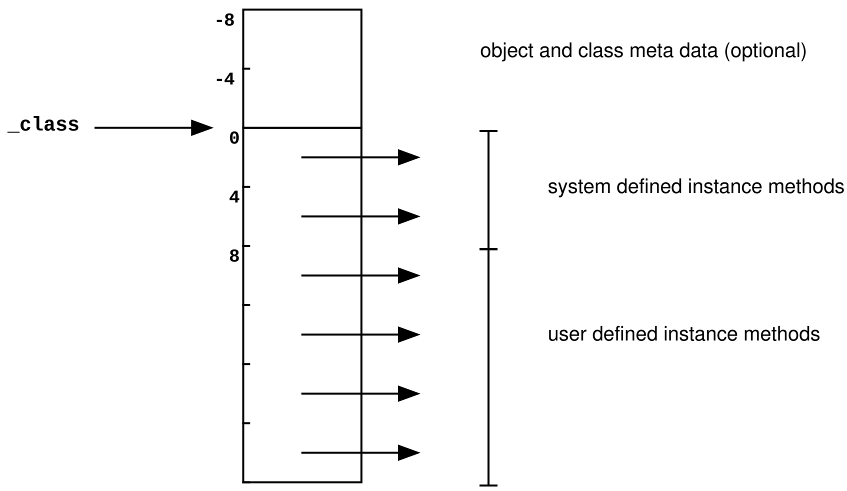

Abbildung 5.9: Klassendeskriptor = Metainformation + Methodentabelle

Abbildung 5.9 zeigt den grundsätzlichen Aufbau eines Klassendeskriptors. Der Zeiger _class eines Objekts verweist direkt auf Offset 0 der Methodentabelle. Die eventuell vorhandene Metainformation kann über negative Offsets erreicht werden. Die Methodentabelle beginnt mit Zeigern auf die systemdefinierten Instanzmethoden, die jedes Objekt aus der Wurzelklasse Object erbt. Es folgen Zeiger auf die vom Programmierer definierten Methoden. Genauer gesagt handelt es sich um *Zeiger auf den Code von Funktionen*, in welche die Methoden gemäß des Verfahrens aus Abschnitt 5.5 übersetzt worden sind.

Das folgende Beispiel illustriert den additiven Aufbau dreier Methodentabellen für die Klassen A, B und C, wobei Methoden nicht überschrieben werden.

Beispiel 5.6.1

```
class A extends Object
{
    method (int) f  { };
}
```

```
class B extends A
{
    method (int) g  { };
}

class C extends B
{
    method (int) h  { };
}
```

Klasse A definiert die Instanzmethode f, B erbt von A und definiert g, C erbt von B und definiert h. Die Instanzmethoden werden übersetzt in die Funktionen `A_f(self: A), B_g(self: B) und C_h(self: C).`

Die sich ergebenden Methodentabellen mit den Zeigern auf die in Funktionen übersetzten Methoden sind in Abbildung 5.10 dargestellt. Die Zeiger auf die systemdefinierten Methoden wurden ausgeblendet um die Übersichtlichkeit zu erhöhen.

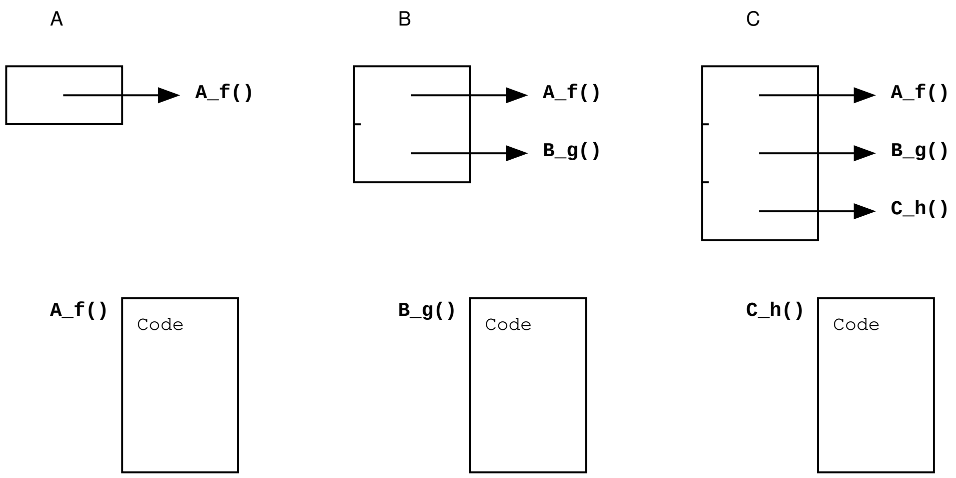

Abbildung 5.10: Methodentabellen für Klassen A, B, C

Die Reihenfolge der Zeiger entspricht der Reihenfolge der Methodendefinitionen. In B und C erscheinen zuerst die Zeiger auf die geerbten Funktionen, erst dann folgt die in der Klasse selbst definierte Funktion. Die Darstellung macht auch deutlich, wie diese verzeigerte Implementierung der Verbung von Methoden Platz spart: Obwohl rein theoretisch in den drei Klassen insgesamt sechs Methoden bekannt sind, wird tatsächlich nur Code für drei Funktionen erzeugt.

Beispiel 5.6.2 *Überschreiben von Methoden*

```
class D extends C
{
    method (int) g  { int dummy := [super g]; };
    method (int) h  { };
}
```

Klasse D erbt alle Merkmale der Klasse C, *überschreibt* aber die geerbten Methoden g und h mit eigenen Definitionen. Die sich für D ergebende Methodentabelle ist daher nicht länger als die für C, besitzt aber andere Zeiger für g und h (siehe Abbildung 5.11). Die in D überschriebenen Methoden sind neu zu übersetzen in die Funktionen `D_g(self: D)` und `D_h(self: D)`.

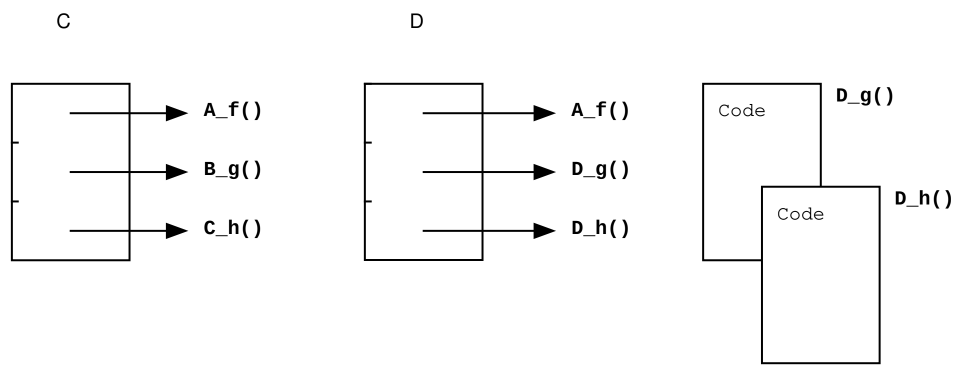

Abbildung 5.11: Methodentabellen für Klassen C und D

Analog zur Behandlung der Instanzvariablen wird ein Übersetzer den hierarchischen Aufbau der Methodentabellen durch geeignete Generierung von Offsets für die Methodennamen ausnutzen. Diese Offsets sind in der Symboltabelle den Einträgen für die jeweiligen Instanzmethoden beizufügen. Der Übersetzer darf aber für überschreibende Methoden keine neuen Offsets vergeben, sondern muß die Offsets der überschriebenen Methoden weiterverwenden. Neue Offsets sind nur für in der Klasse wirklich neu definierte Methoden erlaubt.

Bei der Bildung einer neuen Unterklasse ist zuerst die Methodentabelle der direkten Oberklasse zu kopieren. Werden Methoden überschrieben, so sind die entsprechenden Zeiger auf die neuen Implementierungen auszurichten. Werden neue Methoden definiert, so ist die Methodentabelle entsprechend zu verlängern. Die Länge der zu generierenden Methodentabelle entspricht also immer der Gesamtanzahl der in der Klasse bekannten Methoden.

Übersetzung von Nachrichtenausdrücken (späte Bindung)

Ein beschränkt polymorpher Nachrichtenausdruck der Form

```
[receiver selector: arguments];
```

kann – nach erfolgreicher statischer Typprüfung – übersetzt werden in:

> dereferenziere die Objektreferenz *receiver*;
>
> bestimme die Basisadresse der Methodentabelle durch den Zeiger _class an Offset 0 im Objekt;
> addiere auf die Adresse den generierten Offset für den Methodennamen *selector:*;
> führe einen indirekten Funktionsaufruf über die Adresse mit Übergabe der Argumente *arguments* und der Referenz *receiver* aus.

Zeit-Kosten: Eine Indirektion[3] zur Auflösung der Referenz, eine weitere Indirektion zur Ermittlung der Methodentabelle, eine Addition, ein indirekter Funktionsaufruf. Die Zeit-Kosten sind konstant.

Platz-Kosten: Insbesondere für tiefe Klassenhierarchien können die beschriebenen Methodentabellen unökonomisch werden, da sie viel redundante Information beinhalten.

Im Beispiel 5.6.2 wurde in der Methode g mit [super g] die Implementierung der überschriebenen Methode aufgerufen. In einem Nachrichtenausdruck der Form

```
[super selector: arguments];
```

ist super ein sogenannter *Pseudo-Empfänger*. Diese Nachrichtenausdrücke sind vom Übersetzer zu erkennen und gesondert zu behandeln. Sie sind nur in Methodendefinitionen innerhalb einer Klassendefinition erlaubt. Aus dem Kontext dieses speziellen Nachrichtenausdrucks, nämlich der Klasse und dem Methodennamen, kann der Übersetzer ermitteln, welche Methode einer direkten oder indirekten Oberklasse gemeint ist. Abweichend vom obigen Schema kann hier dann ein direkter Funktionsaufruf in den Code eincompiliert werden.

Der Nachrichtenausdruck des Beispiels 5.6.2 `[super g]` wird statisch gebunden und übersetzt in `call B_g(self)` .

[3] Je nach Implementierung der Referenz auch zwei Indirektionen.

5.6.4 Deskriptor für dynamische Bindung

Smalltalk und Objective-C verschieben die Typprüfung und die Bindung von Nachrichtenausdrücken vollständig auf die Laufzeit des Programms. Insbesondere kann dies zur Folge haben, daß Nachrichten nicht zugestellt werden können. Zur Implementierung dieser *dynamischen Bindung* benötigt man aufwendigere Klassendeskriptoren und ein Subsystem der Laufzeitumgebung, den *Dispatcher* oder *Messager*. Als Grundlage der folgenden Darstellung dient die Objective-C Implementierung der dynamischen Bindung [CN91]. Die Konzepte sind ohne weiteres auf Smalltalk oder andere Sprachen übertragbar.

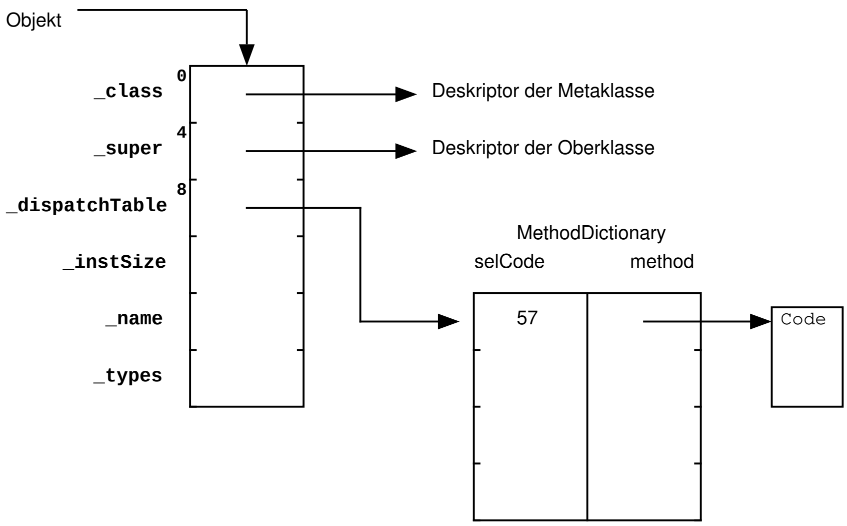

Abbildung 5.12: Klassendeskriptor = Metainformation + Zeiger auf Methodenwörterbuch

Der in Abbildung 5.12 dargestellte Klassendeskriptor besteht aus einem Verbund zur Speicherung der Metainformation. An einem festgelegten Offset innerhalb dieses Verbundes findet sich ein Zeiger _dispatchTable, welcher auf das *Methodenwörterbuch (MethodDictionary)* der Klasse verweist. Eine für die Methodensuche ebenfalls wichtige Komponente ist der Zeiger _super, der auf den Deskriptor der direkten Oberklasse verweist. Der Übersetzer für Objective-C erzeugt weiterhin automatisch zu jeder Klasse eine interne Metaklasse, deren Deskriptor über den Zeiger _class gefunden werden kann. Metaklassen werden in Abschnitt 5.6.5 erläutert.

Objective-C adaptiert für Nachrichtenausdrücke die gut lesbare Smalltalk-Schlüsselwortsyntax (siehe Abschnitt 4.2.7). Die eigentliche Nachricht besteht dabei aus dem *Selektor* und den Argumenten. Wenn verschiedene Methoden in unterschiedlichen Klassen denselben Namen haben, so werden sie auch durch denselben Selektor ausgewählt. Der Begriff Selektor eröffnet eine Abstraktionsebene, die über den einzelnen Implementierungen liegt und diese unter dem Gesichtspunkt eines ähnlichen Verhaltens zusammenfaßt. Es ist Aufgabe des Dispatchers, *abhängig vom Empfänger der Nachricht* die richtige und konkrete Implementierung für den Selektor zu finden.

Der Objective-C Übersetzer führt eine *Selektortabelle*, die jedem distinkten Methodennamen eine Nummer zuordnet. Diese Tabelle wird parallel zur Abarbeitung der Klassendefinitionen erzeugt. Für jeden bisher in der Tabelle nicht verzeichneten Selektor oder Methodennamen wird ein neuer Eintrag angelegt. Intern arbeitet Objective-C also nicht mit den vollen Selektornamen, sondern mit sogenannten *Selektorcodes (selCode)*. Das spart Zeit und Platz. Die Selektortabelle wird geeigneterweise als Dictionary (key-value Paarung) unter Verwendung einer Hash-Funktion zur Beschleunigung des Zugriffs implementiert werden.

Der Übersetzer muß nun für jede Klassendefinition einen Deskriptor mit Methodenwörterbuch generieren. Speziell für den Dispatcher wichtig sind die Zeiger _class und _super. In das Methodenwörterbuch werden *nur* Methoden eingetragen, die entweder geerbte Methoden überschreiben oder die in der Klasse ganz neu definiert werden. Nicht eingetragen werden unverändert geerbte Methoden. Ein Methodenwörterbuch wird daher einen deutlich geringeren Umfang haben als die Methodentabellen für späte Bindung. Ein Eintrag besteht aus dem Selektorcode für den Methodennamen gepaart mit einem Zeiger auf die übersetzte Funktion.

Übersetzung von Nachrichtenausdrücken (dynamische Bindung)

Ein unbeschränkt polymorpher Nachrichtenausdruck der Form

```
[receiver selector: arguments];
```

wird übersetzt in einen Aufruf des Dispatchers:

```
call _dispatch(receiver, selCode, arg1, ... , argn)
```

Der Dispatcher realisiert die sogenannte *Dispatch Table Search (DTS)*:

> dereferenziere die Objektreferenz *receiver* und bestimme den Klassendeskriptor;
>
> finde das Methodenwörterbuch unter _dispatchTable;
> finde im Wörterbuch den Eintrag für *selCode*;
>
> falls gefunden: führe die zugeordnete Funktion unter Übergabe der Argumente und der Referenz *receiver* aus;
> falls nicht gefunden: wiederhole die Suche rekursiv in allen Oberklassen (zu finden über _super);
>
> wird der selCode in keinem Methodenwörterbuch gefunden, signalisiere einen Laufzeitfehler.

Für den Nachrichtenausdruck mit dem Pseudo-Empfänger **super** gelten dieselben Beschränkungen wie bei der späten Bindung. Der Dispatcher besitzt einen speziellen Einstiegspunkt, wo die Suche nach dem selCode sofort in der direkten Oberklasse begonnen wird.

Zeit-Kosten: Die beschriebene Implementierung des Dispatchers als DTS ist für die Praxis viel zu langsam. Die Zeit-Kosten sind nicht konstant. Im worst-case muß die gesamte Klassenhierarchie durchsucht werden. Die DTS wird in der Literatur hauptsächlich aus didaktischen Gründen erwähnt, um das Prinzip der dynamischen Bindung zu verdeutlichen. Dynamisch optimierende Übersetzer für Smalltalk oder Self benutzen sie nur als Notfall-Strategie.

Platz-Kosten: Methodenwörterbücher sind die kompakteste bekannte Darstellungsform, da es für jede übersetzte Methode nur genau einen Eintrag gibt.

5.6.5 Metaklassen

Smalltalk, Objective-C und Java kennen neben Instanzmethoden die sogenannten *Klassenmethoden*. Diese beschreiben Verhalten, welches nicht sinnvollerweise einem einzelnen Objekt zugeordnet werden kann. Das Äquivalent zu Instanzvariablen sind die *Klassenvariablen*.

Klassenmethoden werden aktiviert, wenn der Empfänger der Nachricht nicht ein Objekt ist, sondern die Klasse selbst. Einsatzgebiete sind z.B. die Erzeugung von neuen Instanzen, die Erzeugung speziell initialisierter Instanzen, das Abfragen der Metainformation. Java verwendet Klassenmethoden als Ersatz für globale Funktionen (z.B. im Math-Paket), da diese in der Sprachdefinition nicht vorgesehen sind.

Wie bereits erwähnt, gibt es zu jeder Objective-C Klasse eine automatisch generierte Metaklasse. Wird die Klasse selber als Objekt betrachtet, so ist jede Klasse (die einzige) Instanz ihrer Metaklasse[4].

Die Abbildung 5.12 deutet dies bereits an: Die Metainformation des Klassendeskriptors ist ein Objekt. Der _class Zeiger verweist auf die Klasse des Objekts, die Metaklasse. In reinen objektorientierten Sprachen wie Smalltalk ist auch jedes Methodenwörterbuch eine Instanz einer Klasse MethodDictionary, dies ist aber in Objective-C nicht der Fall. Hier handelt es sich um eine vom Übersetzer für den Dispatcher erzeugte Laufzeit-Datenstruktur.

Die Klassenhierarchie der normalen Klassen spiegelt sich in einer Metaklassenhierarchie. Ist B Unterklasse von A, so ist die Metaklasse von B Unterklasse der Metaklasse von A. Auf diese Weise können auch Klassenvariablen und -Methoden vererbt

[4] Aus der Sicht der Metaklasse sind daher Klassenvariablen nichts anderes als Instanzvariablen. Gleiches gilt für Klassenmethoden

werden. Die Wurzel der normalen Hierarchie ist Object, die Wurzel der Metaklassenhierarchie ist die Metaklasse von Object. In den Methodenwörterbüchern von Object und seiner Metaklasse sind die systemdefinierten Methoden hinterlegt, die u.a. die Verbindung zum Laufzeitsystem mit der Speicherverwaltung herstellen. Die Konstruktion einer Wurzelklasse ist eine sehr anspruchsvolle Aufgabe, da hier die Grundfunktionen des gesamten Objekt-Netzwerks definiert werden. In C++ gibt es weder Metaklassen noch eine vordefinierte Wurzelklasse.

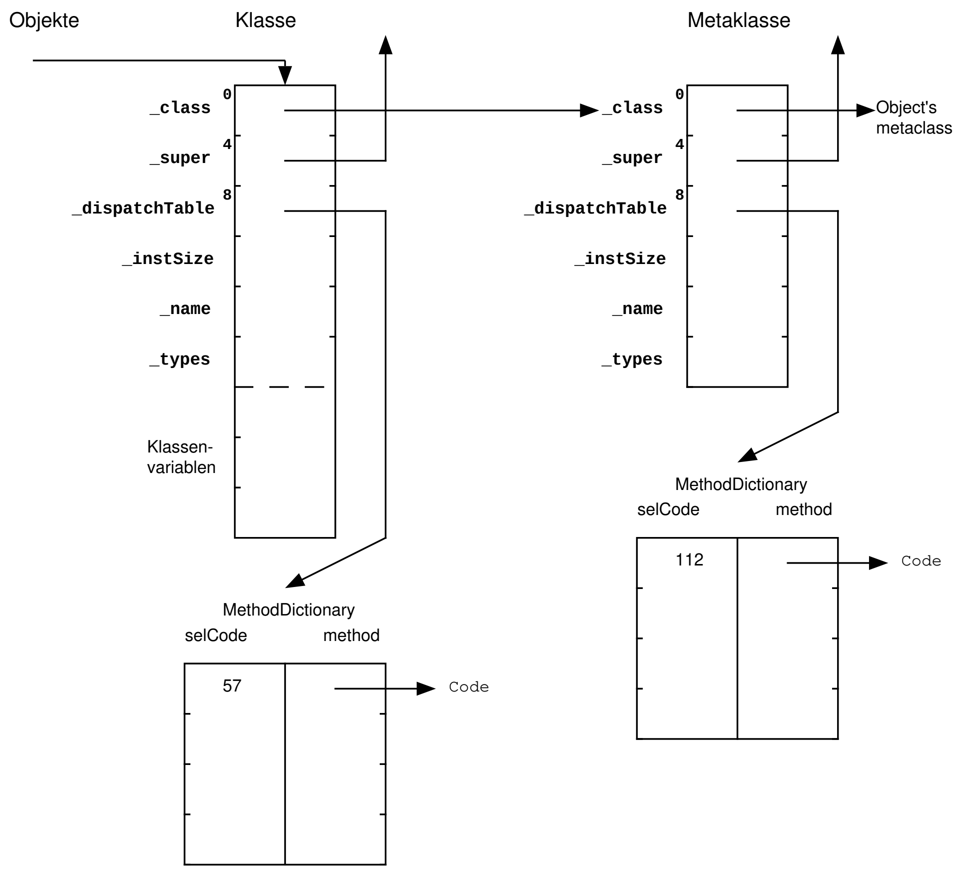

Abbildung 5.13: Klassen- und Metaklassendeskriptor

Abbildung 5.13 zeigt nebeneinander die Deskriptoren einer Klasse und ihrer Metaklasse. Zu jedem Deskriptor gehört ein Methodenwörterbuch, das linke beschreibt die Instanzmethoden der Objekte, das rechte die Klassenmethoden der Klasse.

Interessanterweise ist bei dieser Organisation keine Änderung am Dispatcher erforderlich. Dieser durchsucht einfach die Methodenwörterbücher entlang der Klassenhierarchie. Klassen und Metaklassen sehen für ihn gleich aus. Da Objective-C alle Referenzen als Zeiger implementiert, gibt es auch bei der Auflösung derselben keinen Unterschied. Der Objective-C Literatur ist zu entnehmen, daß die Oberklasse aller Metaklassen letztlich auch die Wurzelklasse Object ist[5]. Object selbst hat keine Oberklasse, im Klassendeskriptor ist _super auf nil gesetzt. Hier beendet der

[5]Die Metaklasse von Object ist also gleichzeitig Unterklasse von Object!

Dispatcher für beide Hierarchien seine Methodensuche.

Beispiel 5.6.3

```
Point2D  p1;           // Klasse als Typ

p1 := [Point2D new];  // Aufruf der Klassenmethode new
```

Die Referenzvariable p1 wird deklariert. Die Klasse Point2D dient dem Übersetzer dabei als Typinformation. Der Nachrichtenausdruck [Point2D new] erzeugt eine Instanz der Klasse und liefert eine Referenz auf das neue Objekt zurück: Der Name Point2D wird interpretiert als Zeiger auf den Klassendeskriptor. Dieser sieht für den Dispatcher aus wie ein normales Objekt. Der Dispatcher sucht nun im Methodenwörterbuch des Metaklassendeskriptors von Point2D nach dem selCode für new. Finden wird er ihn erst in der Metaklasse von Object, da new zu den systemdefinierten Klassenmethoden gehört. Die Implementierung von new ruft den Speicherallokator des Laufzeitsystems auf, die Information über die gewünschte Grösse des Objekts wird dabei dem Eintrag _instSize des Klassendeskriptors von Point2D entnommen.

Beispiel 5.6.4 *Implementierung der systemdefinierten Klassenmethode* new

```
function MetaObject_new(self: MetaObject) : id
{
    id  newObject;

    newObject := allocMemoryBlock(self->_instSize, 0);
    newObject->_class := self;
    return newObject;
}
```

self ist vom Typ MetaObject, d.h. self ist ein Zeiger auf den Klassendeskriptor von Object bzw. auf den Klassendeskriptor einer Unterklasse von Object. Im vorangehenden Beispiel zeigt self auf den Deskriptor für Point2D. Die Systemfunktion allocMemoryBlock reserviert einen Speicherbereich und liefert einen Zeiger auf das erste freie Element zurück.

5.7 Zusammenfassung und Kontext

In diesem Kapitel wurde die Übersetzung der objektorientierten Sprachmerkmale in imperative Datenstrukturen und Prozeduren erläutert:

OO-Merkmal	imperatives Sprachmerkmal
Objekt	Verbund mit Zeiger auf Klassendeskiptor
Objektreferenz	Zeiger
Objektallokation	Laufzeitsystem
Methode	Prozedur mit unsichtbarem Parameter self
Klasse	Verbund für Metainformation und Reihung von Zeigern auf Prozeduren für Methodentabelle
Bindung	indirekter Prozeduraufruf über die Methodentabelle
Vererbung	additive Erweiterung von Verbund und Methodentabelle

Tabelle 5.5: Übersetzung objektorientierter in imperative Merkmale

Der Übersetzer hat im wesentlichen Datenstrukturen für Klassendeskriptoren und Methodentabellen, Code für jeden Nachrichtenausdruck und Aufrufe des Laufzeitsystems für jede dynamische Objektallokation zu generieren. Das Resultat einer solchen Übersetzung ist ein imperatives Programm, das grundsätzlich mit den bekannten Techniken optimiert werden kann. Techniken zur Optimierung der Speicher- und Array-Zugriffe in eingebetteten Systemen werden beispielsweise in [Fra99] beschrieben.

5.8 Literaturhinweise

Appel [App98] bringt anhand der Sprache Object-Tiger eine Einführung in die Übersetzung objektorientierter Sprachen.

Die Übersetzung von C++ wird von Wilhelm und Mauerer [WM96] beschrieben. Die Autoren widmen sich besonders ausführlich der Mehrfachvererbung von Instanzvariablen und -methoden. Die Virtual Function Tables (VTBLs) von C++ werden in [Koe89] erklärt.

Cox beschreibt in [CN91], wie C durch wenige klug gewählte Erweiterungen in eine Smalltalk-ähnliche Sprache verwandelt werden kann. Das Buch erläutert anschaulich die Konzepte von Objective-C und die Rückübersetzung nach C. Eine weitere umfangreiche Informationsquelle zu Objective-C sind die Webseiten von Apple[6], dort wird auch ausführlich auf die Eigenschaften der Wurzelklasse NSObject und das Laufzeitsystem eingegangen.

Das Projekt Oberon wird beschrieben in [WG93]. Auch im Lehrbuch [Mös98] wird im Anhang knapp auf die Übersetzung von Oberon-2 eingegangen. Insbesondere findet man dort ein Beispiel für einen Typdeskriptor.

[6] http://developer.apple.com/techpubs/macosx/macosx.html

Bauer et al. [BH98] haben ein Springer-Lehrbuch zur Übersetzung objektorientierter Sprachen verfasst, das aber dem Autor zur Erhellung des Themas wenig geeignet erschien.

Java ist eine sehr gut dokumentierte Sprache. In [MD97] wird u.a. das Java Class File Format beschrieben.

Die Referenz-Implementierung von Smalltalk ist in frühen Auflagen von [GR83] dokumentiert. Erfahrungsberichte dazu finden sich in [Kra83].

Kapitel 6

Optimierung der Methodensuche

Die gegenüber imperativen Sprachen erheblich höhere Ausdruckskraft objektorientierter Sprachen hat ihren Preis. Die Bindung polymorpher Nachrichtenausdrücke an konkrete Implementierungen kostet Zeit für die Methodensuche und Platz für die Methodentabellen. Eine Optimierung dieses Vorgangs ist daher wünschenswert. Die wichtigsten Techniken sollen in diesem Kapitel vorgestellt werden.

Die bekannten Optimierungstechniken unterscheiden sich dabei in Vorgehensweise und Zielrichtung. *Tabellenorientierte Verfahren* komprimieren die Methodentabellen teilweise erheblich bei gleichzeitig konstanten Suchzeiten. *Caching-Techniken* nutzen die Beobachtung aus, daß Nachrichtenausdrücke in der Praxis nicht oder nur wenig polymorph sind, die Klasse der Empfängerobjekte sich also nicht oder nur selten ändert. Die Ergebnisse erfolgreich durchgeführter Methodensuchen werden als (Klasse, Selektor, Methodenadresse)-Tripel in globalen oder lokalen Caches zwischengespeichert und stehen im Wiederholungsfall schnell zur Verfügung. *Statische Analysetechniken* versuchen die genannte Beobachtung bereits zur Übersetzungszeit auszunutzen. Nachrichtenausdrücke, die nachweislich im betrachteten Programm nicht polymorph sind, d.h. immer dieselbe Methode aktivieren, werden statisch an diese Methode gebunden und in einen normalen Prozeduraufruf übersetzt.

Die ersten beiden Techniken werden schwerpunktmäßig zur Optimierung dynamisch typisierter Sprachen eingesetzt, die dritte Technik ist auf Typinformation angewiesen und eignet sich daher eher für Sprachen mit statischem Typsystem.

6.1 Komprimierung von Methodentabellen

6.1.1 Dispatch Table Search

Die *Dispatch Table Search (DTS)* zur Realisierung der dynamischen Bindung wurde bereits in Abschnitt 5.6.4 beschrieben. Im Spektrum der möglichen Implementierungen stellt diese das eine Extrem dar: kompakteste Methodentabellen, im worst-case muß aber die gesamte Klassenhierarchie nach dem Selektor durchsucht werden. Der Zugriff auf die Methodenwörterbücher kann durch eine Hash-Funktion beschleunigt werden, für eine ausführliche Diskussion siehe [Dri93a].

Vitek et al. [VH96] geben für die Referenzklassenbibliothek ObjectWorks Smalltalk (776 Klassen, 5325 Selektoren, 8780 Methoden, 50696 Call-sites) einen Platzbedarf von 93KB für die Methodenwörterbücher und 405KB für call-sites an.

6.1.2 Selector Table Indexing

Das andere Extrem ist *Selector Table Indexing (STI)*. Dieses Verfahren ist aufgrund seines enormen Platzbedarfs nicht praxistauglich, bildet aber die konzeptionelle Basis für die folgenden Kompressionstechniken. Die Idee hinter STI ist einfach. Für eine Klassenhierarchie mit C Klassen, S Selektoren und M Methodenimplementierungen wird eine zweidimensionale Tabelle konstruiert. Klassen und Selektoren werden fortlaufend durchnummeriert, die Zeilen werden mit den Klassen, die Spalten mit den Selektoren beschriftet. Die Tabelle wird zeilenweise gefüllt. Für jede Klasse und jeden Selektor wird quasi eine DTS durchgeführt und die gefundene Methodenadresse an entsprechender Stelle eingetragen. Kennt die Klasse den Selektor nicht, wird die Adresse des Exception-Handlers eingetragen.

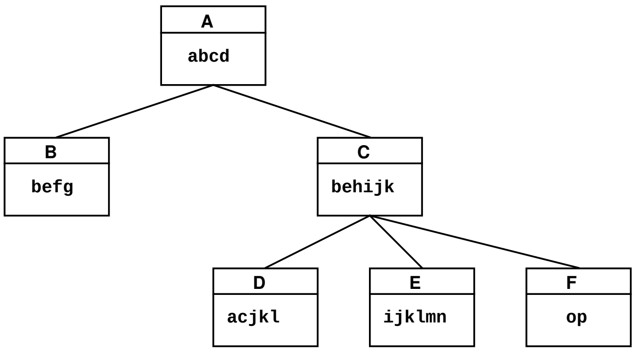

Abbildung 6.1: Die Beispiel-Klassenhierarchie [VH96]

Abbildung 6.1 zeigt eine Klassenhierarchie mit 6 Klassen (A–F), 16 Selektoren (a–p) und 27 Methoden. Tabelle 6.1 ordnet den übersetzten Methoden symbolische Adressen zu. Die zugehörige *Selector Indexed Dispatch Table (SIDT)* zeigt Abbildung 6.2. Die grau hinterlegten Einträge markieren die von den Klassen nicht akzeptierten Selektoren, im Beispiel sind das 44% aller möglichen Einträge. Zur Bildung von Klassendeskriptoren nach Abschnitt 5.6.3 kann man die Tabelle in ihre Zeilen zerlegen. Die Methodensuche reduziert sich auf eine Index-Operation:

```
lookup(Object o, SelectorOffset s)
      row = class(o)     // row = Zeile in der Tabelle
      method = row[s]
      call method(o)
```

Methode	Adr.	Methode	Adr.	Methode	Adr.	Methode	Adr.
A::a	1	C::b	9	D::j	17	E::n	25
A::b	2	C::e	10	D::k	18	F::o	26
A::c	3	C::h	11	D::l	19	F::p	27
A::d	4	C::i	12	E::i	20	msg not under-stood	**0**
B::b	5	C::j	13	E::j	21		
B::e	6	C::k	14	E::k	22		
B::f	7	D::a	15	E::l	23		
B::g	8	D::c	16	E::m	24		

Tabelle 6.1: Methoden und ihre symbolischen Adressen [VH96]

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
A	01	02	03	04	0	0	0	0	0	0	0	0	0	0	0	0
B	01	05	03	04	06	07	08	0	0	0	0	0	0	0	0	0
C	01	09	03	04	10	0	0	11	12	13	14	0	0	0	0	0
D	15	09	26	04	10	0	0	11	12	17	18	19	0	0	0	0
E	01	09	03	04	10	0	0	11	20	21	22	23	24	25	0	0
F	01	09	03	04	10	0	0	11	12	13	14	0	0	0	26	27

Abbildung 6.2: Selector Indexed Dispatch Table

Vitek et al. [VH96] geben für die Referenzklassenbibliothek ObjectWorks Smalltalk einen Platzbedarf von 16.5MB für die Tabelle und 811KB für call-sites an. Der Zeitbedarf liegt bei 6 SPARC-Zyklen.

6.1.3 Virtual Function Tables

Für statisch typgeprüfte Sprachen kann die SIDT zu Klassendeskriptoren verdichtet werden, wie sie bereits in Abschnitt 5.6.3 beschrieben worden sind (Abbildung 6.3).

	a	b	c	d	e	f	g
A	01	02	03	04			
B	01	05	03	04	06	07	08

	a	b	c	d	e	h	i	j	k
C	01	09	03	04	10	11	12	13	14

	a	b	c	d	e	h	i	j	k	l
D	15	09	26	04	10	11	12	17	18	19

	a	b	c	d	e	h	i	j	k	l	m	n
E	01	09	03	04	10	11	20	21	22	23	24	25

	a	b	c	d	e	h	i	j	k	o	p
F	01	09	03	04	10	11	12	13	14	26	27

Abbildung 6.3: VTBL Darstellung der SIDT

Die Methodensuche reduziert sich auf wieder auf eine Index-Operation:

```
lookup(Object o, SelectorOffset s)
      dt = class(o)       // dt = DispatchTable
      method = dt[s]
      call method(o)
```

Der übersetzte Code für einen Nachrichtenausdruck entspricht dem für STI. Eine (hypothetische) Implementierung der Referenzklassenbibliothek mit VTBLs würde nach [VH96] für 8780 Methoden 217062 Einträge beanspruchen. Das wäre ein Platzbedarf von 868KB für die Tabellen.

6.1.4 Selector Colouring

Selector Colouring (SC) [DMSV89, AR92, Dri93b] basiert auf graphentheoretischen Überlegungen zur Kompression der SIDT. Die Idee ist folgende. Dem Intervall $[1, S]$ der Selektorcodes wird eine Funktion $colour(selector)$ zugeordnet, welche das Intervall $[1, S]$ auf ein Intervall $[1, K]$ von *Farben* abbildet. Dabei ist anzustreben $K \ll S$, die Anzahl der Farben soll deutlich geringer sein als die Anzahl der Selektoren. Die Funktion *colour* muß dabei folgende einschränkende Bedingung einhalten. Sei CS die Menge aller Selektoren, die von der Klasse C verstanden werden. Für alle Paare von Selektoren aus CS muß gelten:

$$\forall s1, s2 \in CS : \quad s1 \neq s2 \implies colour(s1) \neq colour(s2) \tag{6.1}$$

Werden zwei Selektoren von einer Klasse verstanden, so sollen sie also verschiedene Farben bekommen.

						h	i				o	p
	a	b	c	d	e	f	g	j	k	l	m	n
A	01	02	03	04	0	0	0	0	0	0	0	0
B	01	05	03	04	06	07	08	0	0	0	0	0
C	01	09	03	04	10	11	12	13	14	0	0	0
D	15	09	26	04	10	11	12	17	18	19	0	0
E	01	09	03	04	10	11	20	21	22	23	24	25
F	01	09	03	04	10	11	12	13	14	0	26	27

Abbildung 6.4: SC Darstellung der SIDT

Die Bestimmung einer guten Färbungsfunktion *colour* ist äquivalent zu einem graphentheoretischen Problem, der Graphfärbung. Jeder Knoten des Graphen repräsentiert einen Selektor. Kanten verbinden zwei Knoten, wenn die entsprechenden Selektoren beide von einer beliebigen Klasse gleichzeitig verstanden werden. Zwei verbundenen Knoten darf nicht dieselbe Farbe zugeordnet werden. Das Problem ist die Bestimmung einer minimalen Färbung des Graphen, so daß verbundene Knoten verschiedene Farben besitzen. Es kann graphentheoretisch bewiesen werden, daß die Anzahl der minimal erforderlichen Farben mindestens so groß ist wie die Knotenzahl der größten Clique des Graphen. Eine Clique ist ein vollständiger Teilgraph. Alle Selektoren einer Klasse unter Einbeziehung der geerbten Selektoren formen eine

Clique. Es ist einleuchtend, daß die Klasse mit der größten Anzahl Selektoren eine untere Schranke für die chromatische Zahl K festlegt.

Das Färbungsproblem ist NP-vollständig, aber die Eigenschaften von Klassenhierarchien ermöglichen einen polynomialen Algorithmus, der eine Färbung mit guter Approximation der minimalen Farbzahl K liefert. Die Selektoren werden absteigend nach der Anzahl der Klassen sortiert, in denen sie verstanden werden. Die Färbung beginnt mit dem Selektor, der am häufigsten verstanden wird. Dem nächsten Selektor in der Folge wird dann die kleinste Farbe zugeteilt, die die Färbungsbedingung noch einhält. Der Algorithmus endet, wenn dem letzten Selektor eine Farbe zugeordnet wurde.

Die Anzahl der Spalten der SIDT kann nun auf K Spalten reduziert werden (Abbildung 6.4). In den Nachrichtenausdrücken ist der Selektorcode s durch die Farbe $colour(s)$ zu ersetzen. Man könnte vermuten, daß die Methodensuche sich wieder auf eine Index-Operation reduziert (c = colour(s)):

```
lookup(Object o, Colour c)
      dt = class(o)
      method = dt[c]
      call method(o)
```

Das Problem ist aber, daß die Bedingung 6.1 nicht immer umkehrbar ist. Vielen Selektoren wird dieselbe Farbe zugeteilt werden *(Selektor-Aliasing)*. Im Beispiel gilt dies für $\{h, f\}$, $\{i, g\}$, $\{o, p\}$ und $\{p, n\}$.

Sei CC die Menge der Farben, die den Selektoren einer Klasse C zugeordnet wurde. Es kann nun folgender Fall eintreten:

$\exists c \in CC$ und $\exists s1, s2 \in S$ so daß zwar gilt $colour(s1) = colour(s2) = c$ aber $s1 \notin CS$ oder $s2 \notin CS$.

Die Konsequenz aus diesem *Selektor-Aliasing-Problem* ist, daß die nach einer Farbe aufgerufene Methodenimplementierung in ihrem *Methoden-Prolog* überprüfen muß, ob der Selektor stimmt. Ein Nachrichtenausdruck wird übersetzt zu:

```
lookup(Object o, Colour c, SelectorNum s)
      dt = class(o)
      method = dt[c]
      call method(o, s)
```

Der Methoden-Prolog lautet:

```
if (s != #mySelectorNumber)
    goto messageNotUnderstood
```

Ein Methoden-Prolog macht deswegen Sinn, weil es in grossen Systemen deutlich weniger Methoden als Call-sites gibt (8780 Methoden stehen 50696 Call-sites gegenüber). Die Auslagerung der Prüfung von der Call-site in den Prolog spart also den Platz für redundanten Dispatch-Code.

Vitek et al. geben für die Referenzklassenbibliothek einen Platzbedarf von 1.15MB für die Tabelle und 916KB für call-sites an. Der Zeitbedarf liegt bei 9 SPARC-Zyklen.

6.1.5 Row Displacement

Die *Row Displacement (RD)*-Technik [Dri93b] betrachtet die SIDT zeilenweise und bildet die Zeilen nacheinander auf ein eindimensionales MasterArray ab, so daß sich echte Einträge nur mit leeren Einträgen (die grauen Boxen mit einer 0) überlappen[1].

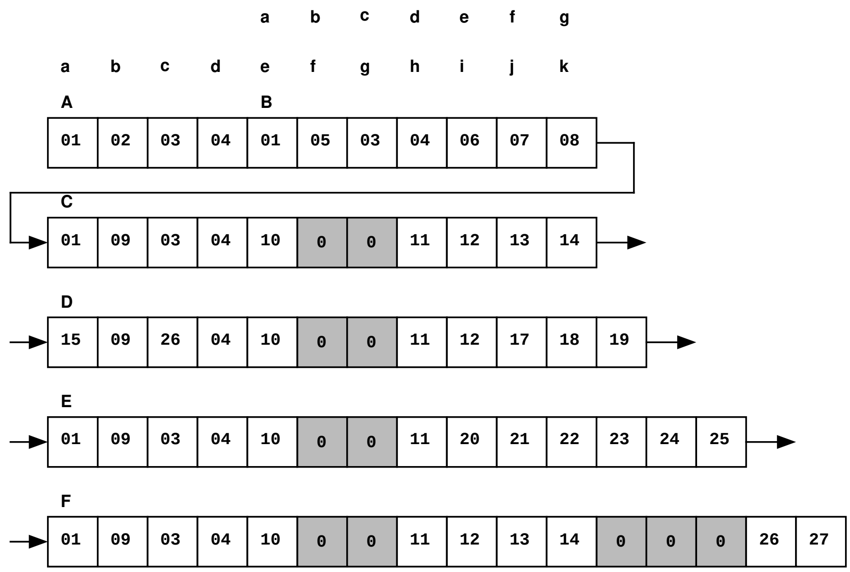

Abbildung 6.5: RD-Darstellung der SIDT

Der von Driesen [Dri93b, Dri93a] beschriebene Algorithmus basiert auf einer geschickten Sortierung der Selektoren mit anschließender Vergabe von Selektoroffsets und einer entsprechenden Neuanordnung der Einträge. Der Autor kann für die Smalltalk-Klassenhierarchie ein Array mit nur 33% leeren Einträgen berechnen. Ein weiter verbesserter Algorithmus [DHV95], der spaltenweise arbeitet, füllt das Array zu 95%.

Die Anwendung einer einfachen RD-Transformation auf das Beispiel zeigt Abbildung 6.5. Innerhalb des gesamten Arrays gibt es für jede Klasse einen bestimmten

[1] Anschaulich beschrieben, werden die Zeilen am Ende abgeschnitten, wenn nur noch Nullen folgen, und dann hintereinandergelegt)

Einstiegspunkt (Klassenoffset) in die eigene Tabelle. Die Einträge werden dann indiziert über die Selektoroffsets. Wie man an den Klassen A und B bereits erkennen kann, gibt es ähnlich wie bei Selector Colouring die Gefahr von Fehlinterpretationen, die aber mit der für SC beschriebenen Prolog-Technik auch ausgeschlossen werden können.

Ein Nachrichtenausdruck wird übersetzt zu:

```
lookup(Object o, SelectorOffset s)
      dt = classoffset(o)     // Offset in MasterArray
      method = dt[s]
      call method(o, s)
```

Der Methoden-Prolog lautet:

```
if (s != #mySelectorOffset)
    goto messageNotUnderstood
```

Vitek stellt in [Vit95] eine verbesserte Technik zur Vermeidung von falschen Methodenaktivierungen vor, die auf sogenannten *confusable sets* basiert. Zu jedem Selektor wird die Menge der im MasterArray fehlinterpretierten Selektoren ermittelt. Diesen Mengen wird ein Code zugewiesen, der dann im Methoden-Prolog statt des SelectorOffsets überprüft wird. Auf diese Weise kann der Umfang des an den call-sites erforderlichen Maschinencodes weiter reduziert werden. Vitek et al. geben für die Referenzklassenbibliothek einen Platzbedarf von 819KB für das MasterArray und 916KB für call-sites an. Der Zeitbedarf liegt bei 9 SPARC-Zyklen. Die Kosten sind damit denen für SC vergleichbar.

6.1.6 Compact Dispatch Tables

Vitek beschreibt in seiner Diplomarbeit [Vit95] die *Compact Dispatch Tables (CT-95)*. Ausgehend von der vollen SIDT werden folgende Optimierungen durchgeführt. Die SIDT wird in zwei Tabellen zerlegt. Die eine Tabelle beinhaltet normale Selektoren, die andere sogenannte konfligierende Selektoren (conflict selectors)[2]. Konflikt-Selektoren sind solche Selektoren, die von Klassen implementiert werden, die sich nicht in einer Vererbungsbeziehung zueinander befinden. Auf die normale Dispatch Tabelle wird ein Selektor-Aliasing Algorithmus zur ersten Verdichtung angewandt. Die zentrale Idee des Verfahrens ist die geeignete *Partitionierung* dieser Tabellen, so daß identische Partitionen von verschiedenen Klassen geteilt werden können.

> "The idea of partitioning is to improve sharing of dispatch tables by allowing the sharing of portions of tables. Consider a subclass which

[2] conflict: Konflikt; in Widerspruch stehen, widersprechen

> inherits one hundred methods from its superclass and only redefines five of them. Would it not be simpler to have one table with the ninety-five common entries, and two separate tables with the five methods that actually differ?"

Der Partitionsgrösse ist ein Parameter des Algorithmus, der aber nur sehr allgemein beschrieben wird. Der Klassendeskriptor besteht aus einem Array von Zeigern auf Methodentabellen, die den einzelnen Partitionen entsprechen. Diese Struktur ist für alle Klassen identisch. Jede Klasse hat die gleiche Anzahl von Zeigern auf Partitionen, jede Partition hat die gleiche Anzahl von Zeigern auf Methoden. Der Übersetzer ordnet jedem Selektor einen Offset innerhalb einer Partition zu.

Der experimentelle Teil der Arbeit bestimmt für die Referenzbibliothek eine optimale Partitionsgrösse von 14 Einträgen. Nachrichtenausdrücke werden übersetzt wie bei SC und RD mit einer zusätzlichen Indirektion. Es ergibt sich ein Platzbedarf von 221KB für 23 Partitionen und 1MB für call-sites. Der Zeitbedarf liegt bei 11 SPARC-Zyklen. Damit sind zumindest die Platz-Kosten deutlich reduziert worden.

Der Algorithmus stellt auch für statisch typisierte Sprachen eine interessante Alternative dar, da er dem Autor zufolge Tabellen erzeugt, die regelmäßig deutlich kleiner sind als VTBLs, bei konstanter Suchzeit. Im Anhang der Arbeit finden sich Übersetzungen der lookup-Prozeduren und Methoden-Prologe nach SPARC-Assembler.

6.1.7 Weitere Verfahren

Huang und Chen [HC92][3] beschreiben *Modified Two-Way Colouring (MTWC)*, eine Weiterentwicklung von Selector Colouring. Das Verfahren nutzt die strukturelle Ähnlichkeit von Klassen und Unterklassen aus, in dem es die volle SIDT nicht nur spaltenweise nach den Selektoren, sondern auch zeilenweise nach den Klassen färbt. Mögliche Fehlinterpretationen werden wie bei SC durch einen Methoden-Prolog abgefangen, der allerdings zusätzlich zum Selektor auch noch die Klasse überprüfen muß. Der Artikel beschreibt detailliert die Färbungs- und Dispatchalgorithmen. Den Autoren zufolge hat MTWC ungefähr die sechsfachen Platzkosten der DTS bei konstanter Dispatch-Zeit.

Thorup [Tho93] hat für den GNU Objective-C Compiler ein *Sparse-Array* Verfahren implementiert, das ähnlich wie CT-95 arbeitet und ebenfalls die strukturellen Ähnlichkeiten zwischen Ober- und Unterklassen ausnutzt. Die Zeilen der SIDT werden in Partitionen gleicher Grösse zerlegt, identische Partitionen werden nur einmal gespeichert. Klassendeskriptoren bestehen aus Zeigern auf Partitionen.

Muthukrishnan und Müller [MM96] bzw. Ferragina und Muthukrishnan [FM96] abstrahieren die Methodensuche zu einem sogenannten *colored-ancestors* Problem

[3] siehe http://www.iis.sinica.edu.tw/˜skhuang/publication.html für eine revidierte Fassung des Artikels

und unterscheiden eine statische und eine dynamische Variante. Gegeben sei ein gewurzelter Baum mit n Knoten und eine Menge von Farben $\{1, 2, \ldots, C\}$. Jedem Knoten u wird eine bestimmte Teilmenge von Farben der Grösse d_u zugeordnet, D sei die Summe $\sum_u d_u$. Die statische Variante des Problems besteht in der Beantwortung von Anfragen $FIND(p, c)$. Gesucht ist der *nächste Vorgänger* von Knoten p (einschließlich p selbst), der die Farbe c hat. Existiert kein solcher Vorgänger, liefert $FIND(p, c)$ `null` zurück. Die dynamische Variante des Problems kennt zusätzlich noch Operationen $INSERT(p, c)$ und $DELETE(p, c)$, die Farben zu Knoten hinzufügen bzw. entfernen. Zu beachten ist ferner, daß der Baum selbst nicht geändert wird. Knoten entsprechen Klassen, Farben entsprechen Selektoren und D bezeichnet die Anzahl der Methoden.

Der erste Artikel [MM96] präsentiert für die statische Variante eine Datenstruktur, welche mit linearen Zeit- und Platzkosten in $\Theta(n + D)$ erzeugt werden kann und Anfragen deterministisch in Zeit $O(\log \log n)$ beantwortet. Im theoretischen Teil der Arbeit wird das colored-ancestors Problem reduziert auf das Finden des kleinsten Integer-Intervalls innerhalb einer Menge von geschachtelten Intervallen, welches eine gesuchte natürliche Zahl enthält. Als praktisch anwendbares Verfahren wird die sogenannte *(x, y)-selective Strategie* vorgestellt, welche im Kostenvergleich besser abschneidet als das Verfahren von Thorup. Andere Vergleiche werden in der Arbeit selbst nicht durchgeführt. Als Grundlage der Bewertung dient aber die weiter oben erwähnte Referenzklassenbibliothek ObjectWorks Smalltalk. Die im Artikel genannten Zahlen deuten darauf hin, daß das beschriebene Verfahren eine mit CT-95 vergleichbare Performanz hat. Die theoretische Arbeit [FM96] verallgemeinert die gefundenen Ergebnisse auf die dynamische Variante des colored-ancestors Problem. Die genannten Arbeiten sind den Autoren zufolge die ersten nichttrivialen theoretischen Untersuchungen zum Problem der dynamischen Methodensuche.

6.1.8 Zwischenbewertung

Insbesondere die Arbeiten von Vitek et al. machen deutlich, daß bei der Bewertung eines Kompressionsverfahrens nicht nur die Grösse der resultierenden Tabelle entscheidend ist, sondern auch die Gesamtgrösse des an den call-sites erforderlichen Dispatch-Codes. Die Messungen von Vitek et al. beziehen sich dabei auf ein sehr grosses System. Es wäre zu überprüfen, ob sich diese Ergebnisse für eingebettete Systeme bestätigen lassen, da hier von kleineren Klassenbibliotheken und Programmen mit deutlich weniger call-sites auszugehen ist.

VTBLs sind die bevorzugte Implementierungstechnik für späte Bindung in statisch typisierten Sprachen. Der im Vergleich immer noch hohe Platzbedarf dieser Standardtechnik läßt auch für späte Bindung den Einsatz von Alternativen wie beispielsweise CT-95 sinnvoll erscheinen.

6.2 Caching

Caching-Verfahren sind zuerst für *interaktive Programmierumgebungen* wie Smalltalk und Self entwickelt worden. Diese zeichnen sich dadurch aus, daß ein Programmierer immer vollen Zugriff auf sämtliche Klassendefinitionen hat und diese jederzeit, auch während des "laufenden Betriebes" modifizieren kann. Die Anwendung eines tabellenorientierten Verfahrens kann unter diesen Umständen zu erheblichen Wartezeiten führen, da jede Änderung einer bestehenden Klasse oder der Klassenhierarchie eine zeitaufwendige Neuberechnung der Dispatch-Tabellen und der call-sites erforderlich macht.

In der Regel wird daher nur die DTS mit einem Cache beschleunigt. Ergebnisse erfolgreich durchgeführter Methodensuchen werden zwischengespeichert. Ein Cache kann nur dann funktionieren, wenn sich nach Klasse und Selektor identische Methodensuchen innerhalb eines begrenzten Zeitraumes auch oft genug wiederholen.

Man unterscheidet im wesentlichen zwei Ansätze: Das System besitzt entweder einen grossen globalen Cache oder viele kleine, einen an jeder call-site (inline cache).

6.2.1 Globales Caching

Ein globaler Cache speichert Tripel der Form (Klasse, Selektor, Methode). Der Zugriff wird in der Regel durch Hashing beschleunigt [Dri93a]. Zur Auswertung eines Nachrichtenausdrucks wird aus Klasse und Selektor ein Hash-Code berechnet, mit dem dann der Cache sondiert wird. Bei einem Treffer wird die entsprechende Methode sofort aktiviert, ansonsten muß auf die DTS zurückgegriffen werden. Der Cache wird mit dem neuen Tripel aktualisiert. In der Literatur [Kra83] wird von Trefferraten von 85-90% berichtet.

Alternativ kann man auch jeder Klasse einen eigenen Cache zuteilen (class caching). Dieses Verfahren kommt im Objective-C Laufzeitsystem von NeXT/Apple zur Anwendung.

6.2.2 Inline Caching

Inline Caching (IC) wurde erstmalig zur Beschleunigung von Smalltalk-80 eingesetzt [DS84]. Das Verfahren basiert auf der Beobachtung, daß sich oft bei zeitlich aufeinanderfolgenden Auswertungen eines bestimmten Nachrichtenausdrucks die Klasse des Empfängers nicht ändert. Inline Caching wird realisiert als eine Form von selbstmodifizierendem Code. Ein Nachrichtenausdruck wird zuerst übersetzt in einen normalen Aufruf des Dispatchers. Kommt ein Nachrichtenausdruck zur Auswertung, wird die Klasse des Empfängers an der call-site zwischengespeichert und die Methodenadresse in der DTS gesucht. Der Aufruf des Dispatchers wird an der call-site dann durch einen direkten Aufruf der Methode ersetzt. Da sich die Klasse des Empfängers auch wieder ändern kann, prüft jede Methode in ihrem Prolog, ob die gespeicherte Klasse mit der aktuellen Klasse des Empfängers übereinstimmt. Deutsch et al. berichten von Trefferraten von 95% für einen solchen Cache, wobei in der Implementierung nicht mit Prologen, sondern mit verbundenen (letzte Klasse testen) und unverbundenen (DTS) Dispatch-Routinen gearbeitet wird:

```
push receiver
push arg1,...,argn
call unlinkedDispatch
<selector>
<unused>
```

```
push receiver
push arg1,...,argn
call linkedDispatch
<last class>
<method>
```

Inline Caching und globales Caching können kombiniert werden. Polymorphes Inline Caching (PIC) ist eine direkte Erweiterung der obigen Idee [HCU91], in dem an der call-site mehrere Klassen und Methodenadressen erinnert werden.

Die Performanz von IC und PIC hängt stark vom Grad des Polymorphismus des einzelnen Nachrichtenausdrucks ab. Hölzle et al. unterscheiden *monomorphe*, *polymorphe* und *megamorphe* Nachrichtenausdrücke mit entsprechend nur einer konstanten Empfängerklasse, wenigen Empfängerklassen oder sehr vielen Empfängerklassen.

Untersuchungen von Driesen et al. [DHV95] deuten an, daß Inline Caching schneller sein kann als eine VTBL-Suche, womit das Verfahren auch für statisch typisierte Sprachen interessant wird. Die Autoren begründen dies mit dem für VTBLs erforderlichen indirekten Funktionsaufruf, der die Befehlspipeline moderner Prozessoren invalidiert, was für einen Treffer im IC nicht der Fall ist. Allerdings kombiniert Vitek IC mit CT-95 ohne eine Verbesserung der Performanz des Referenzsystem zu erzielen.

6.3 Statische Analysetechniken

Insbesondere Programme in Sprachen mit einem statischen Typsystem sind einer Analyse zur Übersetzungszeit zugänglich. Eine Analyse kann besonders wirkungsvoll durchgeführt werden, wenn der Übersetzer unter der Closed-World Annahme (siehe Kapitel 8.3) arbeitet. Weiterhin soll davon ausgegangen werden, daß Klassen nur einfach vererbt werden und die Klassenhierarchie ein Baum mit genau einer Wurzel ist.

Eine statische Analyse kann zur Optimierung eines konkreten Quellprogramms folgendermaßen vorgehen:

1. Selektoren, die in keinem Nachrichtenausdruck vorkommen, werden ermittelt und müssen in Methodentabellen nicht durch Einträge (Spalten der SIDT) berücksichtigt werden. Für alle Methoden, die diese Selektoren implementieren, braucht kein Code generiert zu werden;

2. für Klassen, die nicht instantiiert werden, kann auf die Generierung von Deskriptoren[4] und Einträgen in Methodentabellen (Zeilen der SIDT) verzichtet werden;

3. gibt es für einen Selektor nur genau eine Implementierung, so muß dieser Selektor nicht in Methodentabellen (Spalten der SIDT) berücksichtigt werden;

4. Nachrichtenausdrücke sind nach Möglichkeit statisch zu binden;

5. es werden alle Methoden ermittelt, auf die keine Verweise mehr in Methodentabellen und Prozeduraufrufen existieren. Für diese Methoden muß kein Code generiert werden.

Bis auf Schritt 4 lassen sich die oben genannten Untersuchen recht einfach durch eine statische Inspektion des Quelltexts realisieren. Diese Optimierungen reduzieren insbesondere die Grösse der Methodentabellen und den Platzbedarf für übersetzte Methoden.

Ein Übersetzer kann unter folgenden Umständen Nachrichtenausdrücke[5] statisch binden und in direkte Prozeduraufrufe übersetzen:

1. es gibt in der gesamten Klassenhierarchie nur genau eine Implementierung des Selektors;

2. Nachrichtenausdrücke mit dem Pseudo-Empfänger super, siehe Kapitel 5.6.3, können statisch gebunden werden, in dem die erste Implementierung durch Aufwärtssuche in einer Oberklasse gefunden wird;

3. der statische Typ des Empfängerobjekts ist ein *Blatt* des Klassenbaums. Der Nachrichtenausdruck kann daher nicht polymorph sein. Der Selektor wird entweder in der Klasse selbst implementiert oder die geerbte Implementierung kann durch Aufwärtssuche in einer Oberklasse gefunden werden;

4. der statische Typ des Empfängerobjekts ist *Wurzel eines Teilbaums* der Klassenhierarchie. Der Nachrichtenausdruck ist potentiell polymorph. Der Selektor wird aber in den Unterklassen nicht neu implementiert sondern nur geerbt. Die Implementierung des Selektors kann entweder im statischen Typ selbst oder in einer Oberklasse gefunden werden;

5. der Nachrichtenausdruck ist potentiell polymorph. Der Übersetzer kann aber die möglichen Empfängerklassen eingrenzen auf eine bestimmte *Teilmenge* von Klassen. In dieser Teilmenge wird immer dieselbe Implementierung des Selektors aktiviert.

[4]bezüglich der Deskriptoren ist die Implementierung von Typtest und Typzusicherung zu berücksichtigen, diese benötigen eventuell die _super-Zeiger aus der Metainformation.

[5]zur Erinnerung: Ein Nachrichtenausdruck besteht aus dem Empfängerobjekt *und* dem Selektor. Das Empfängerobjekt kann innerhalb einer Methodendefinition auch durch den Parameter self bezeichnet werden, in diesem Fall ist der statische Typ von self die Klasse, in der die Methode definiert wird.

Die Fälle 1 und 2 sind ebenfalls durch eine statische Inspektion des Quelltexts problemlos realisierbar. Zur Bewältigung der Fälle 3, 4 und teilweise 5 ist die sogenannte *Class Hierarchy Analysis CHA* [DGC95, Dea96] entwickelt worden. Diese immer noch leicht zu implementierende aber effektive Technik betrachtet die gesamte Klassenhierarchie, konzentriert sich dabei aber nur auf Definitionen von Klassen und die von ihnen implementierten Selektoren. Das Aufrufverhalten der Methoden untereinander muß nicht weiter analysiert werden. Das Verfahren basiert auf intraprozeduraler Datenflußanalyse und ordnet jedem Ausdruck eine spezifische Liste von konkreten Empfängerklassen zu. Diese Information unterscheidet sich von statischer Typinformation, die nur eine Schnittstelle spezifiziert, aber keine genauen Angaben über die tatsächlichen Klassen macht. In [DGC95] wird von Beschleunigungen von 23% bis 89% und von Platzersparnissen von 12% bis 21% berichtet, dies allerdings für Programme in der Sprache Cecil. Aigner et al. [AH96] beschreiben einen optimierenden Source-to-Source Compiler für C++, der Laufzeit-Typrückmeldungen mit Klassenhierarchie-Analyse kombiniert. Die dort für CHA ermittelten Ergebnisse sind allerdings deutlich schlechter. Ein Grund dafür kann sein, daß in der reinen objektorientierten Sprache Cecil *per se* sehr viel mehr Gegelegenheiten für wirkungsvolle statische Bindung vorhanden sind als in der hybriden Sprache C++. Beispielsweise wird in Cecil der Zugriff auf Instanzvariablen indirekt durch dynamische Bindung an Zugriffsmethoden realisiert. Klassenhierarchie-Analyse kann die in Abschnitt 6.4 erläuterten Direktiven überflüssig machen.

Eine weitere Verbesserung der CHA wird durch *Interprocedural Class Analysis* [Gro95, Gro98b, DGC98] erzielt, die feinkörnigere Analysen durchführt. Es wird ein Datenflußgraph konstruiert, wobei es für jede Variablendeklaration, Methodendeklaration, Klassen-Instantiierung und jeden Nachrichtenausdruck des Quellprogramms einen Knoten gibt. Durch den Graphen werden konkrete Mengen von Klassen propagiert. Das Ziel ist wie bei CHA die mögliche Menge von Empfängerklassen in polymorphen Nachrichtenausdrücken so weit wie möglich einzuschränken. DeFouw et al. [DGC98] beschreiben einen schnellen Algorithmus zur Konstruktion eines Datenflußgraphen, der viele wichtige ältere Verfahren subsumiert und um neue ergänzt.

6.4 Spezielle Quelltextdirektiven

Der Programmierer kann die statische Bindung von Nachrichtenausdrücken erleichtern, wenn dem Übersetzer im Quelltext entsprechende Hinweise gegeben werden können. Bekannte Konstrukte sind exemplarisch:

C++: Statische Bindung ist die Regel, sollen Nachrichtenausdrücke zur Laufzeit gebunden werden, so sind die entsprechenden Methodendefinitionen als `virtual` zu deklarieren. Diese Vorgehensweise hat entscheidende Nachteile und wird nicht mehr empfohlen. Das Problem besteht darin, daß man insbesondere beim Design von Klassenbibliotheken nicht immer im voraus wissen kann, welche Klassen ihre Merkmale sinnvollerweise polymorph weitervererben sollen.

Java: Klassen können als final deklariert werden. Diese Deklaration entspricht einem Verbot weiterer Unterklassenbildung, so daß Nachrichtenausdrücke mit diesen Empfängerklassen ohne weitere Umstände vom Übersetzer statisch gebunden werden können. Die final-Deklaration ist die wesentlich sinnvollere Umkehrung der virtual-Deklaration.

Dylan: Während die final-Deklaration nur einzelne Blätter des Klassenbaums fixiert, kann die sealed class-Direktive von Dylan ganze Zweige mit Blättern versiegeln. Eine versiegelte Klasse darf keine Unterklassen haben, es sei denn die Unterklassen befinden sich in *derselben* Bibliothek wie die versiegelte Klasse. Die Menge der im übrigen ebenfalls versiegelten Unterklassen einer versiegelten Klasse ist also zur Übersetzungszeit bekannt. Diese Information kann zur statischen Bindung genutzt werden.

Java: Klassenmethoden werden durch das Schlüsselwort static als solche bekannt gemacht und können statisch gebunden werden.

Objective-C, C++, Oberon-2 Objekte können in Objektvariablen statisch alloziert werden. Auf diese Weise kann der dynamische Typ nicht vom statischen Typ abweichen. Nachrichtenausdrücke mit solchen Empfängerobjekten können daher statisch gebunden werden.

Insbesondere die final- und sealed class-Direktiven können auch zur Sicherheit in der Programmierung beitragen. Den Designern von Modulen oder Klassenbibliotheken ist es nun möglich, vom Übersetzer nachprüfbar bestimmte Klassen, die dafür nicht geeignet oder vorgesehen sind, von der weiteren Unterklassenbildung auszuschliessen[6].

6.5 Kontext und Bewertung

Die in diesem Kapitel vorgestellten Techniken sind ursprünglich zur Optimierung teilweise extrem anspruchsvoller dynamisch typisierter Sprachen wie Smalltalk, Self, Cecil entwickelt worden. Erst in der Folge hat man versucht, diese Techniken auf statisch typisierte Sprachen wie C++ oder Modula-3 zu übertragen. Abschließende eindeutige Bewertungen für die Anwendbarkeit in eingebetteten Systemen können daher nicht gemacht werden.

Die Ergebnisse bezüglich der tabellenorientierten Verfahren deuten aber darauf hin, daß auch für statisch typisierte Sprachen die Anwendung von beispielsweise CT-95 sinnvoll sein kann. Weiterhin kann ein Übersetzer durch teilweise recht problemlose Quelltext-Inspektionen zur Verdichtung der Tabellen und zur statischen Bindung von Nachrichtenausdrücken beitragen, so daß mindestens der Platzbedarf des ausführbaren Programms reduziert wird.

[6] Ein Klient kann eine Klasse auf zwei Weisen nutzen: durch Erzeugung von Objekten der Klasse und durch Unterklassenbildung

6.6 Literaturhinweise

Einen sehr guten Überblick über tabellenorientierte und caching-Verfahren findet man in der Diplomarbeit von Jan Vitek [Vit95]. Leistungsfähige statische und dynamische Optimierungstechniken wurden insbesondere für die Sprachen Self[7] und Cecil[8] entwickelt und in der Folge dann auf C++, Modula-3, Java etc. angewendet. Hölzle entwickelt in seiner Dissertation [Höl94] extrem wirkungsvolle Optimierungstechniken im Kontext der dynamischen prototypenbasierten Sprache Self. Mit diesen Techniken konnte die Ausführungsgeschwindigkeit von Self auf das doppelte der besten Smalltalk-Implementierung gesteigert werden.

Einen guten Überblick über statische Analysetechniken geben [Cha96, CDG97]. Aigner et al. [AH96] beschreiben einen optimierenden Source-to-Source Compiler für C++, der Laufzeit-Typrückmeldungen mit Klassenhierarchie-Analyse kombiniert.

Ein Lehrbuch zu *Dylan*, der Apple Dynamic Language, ist [FKMW97]. Die Sprache wird ebenfalls auf den Webseiten der Harlequin Group[9] vorgestellt. Dylan ist eine moderne hybride objektorientierte Sprache mit Scheme-ähnlicher Syntax und Lexical Scoping, unterstützt Multiple Inheritance, Multiple Dispatching, ein leistungsfähiges Modulsystem und einiges mehr.

[7] http://www.cs.ucsb.edu/oocsb/self/papers/papers.html
[8] http://www.cs.washington.edu/research/projects/cecil/www/Papers/papers.html
[9] http://www.harlequin.com/products/products_dylan.html

Kapitel 7

Optimierung der Objektspeicherung

Die objektorientierte Programmierung setzt konzeptionell ein dynamisches Speichermodell voraus. Die dynamische Allokation von Objekten erst zur Laufzeit ist wesentlicher Bestandteil der Semantik objektorientierter Sprachen. Diese Grundannahmen haben folgende konkreten Auswirkungen. Objektvariablen speichern nur eine Referenz auf ihr Objekt (Abschnitt 5.3). Alle Objekte besitzen als erste Komponente den _class Zeiger, der zur Laufzeit den Klassendeskriptor und damit die Methodentabelle zugänglich macht (Abschnitt 5.4.1). Es wird ein Laufzeitsubsystem benötigt, welches dynamische Speicheranforderungen bearbeitet und insbesondere den Speicher um nicht mehr benötigte Objekte bereinigt (Abschnitt A).

In diesem Kapitel werden Optimierungen diskutiert, die zu einer Zeit- und Platzkostenersparnis in den oben genannten drei Bereichen beitragen können.

7.1 Dereferenzierung

Vor jedem Zugriff auf eine Instanzvariable und jeder Methodenaktivierung ist die entsprechende Objekt-Referenz aufzulösen. Wird die Referenz nicht als direkter Zeiger implementiert, kostet dieser Vorgang Zeit für die Indirektion über den Master-Pointer oder die Objekttabelle. Für eine Methodenaktivierung ist zusätzlich die Adresse der Methodentabelle dem _class Zeiger des Objekts zu entnehmen. Dieser Vorgang kostet eine weitere Indirektion. Referenzen sind durch den unsichtbaren Parameter self und die eventuell deklarierten formalen Parameter Bestandteil jeder Methodendefinition. Innerhalb einer Methode wird in der Regel mehrfach auf self und andere Objekte zugegriffen, so daß sich der oben beschriebene Zeitaufwand multiplizieren kann.

Eine erste Optimierung kann darin bestehen, die Objektadressen und Methodentabellen für fragliche Objekte nur einmal zu bestimmen und die gefundenen Adressen in Registern der Zielmaschine zu speichern. Ein Übersetzer kann den Code einer Methode auf Mehrfachzugriffe hin analysieren und diese bei der Registerzuteilung und bei der Dispatch-Code Generierung berücksichtigen. Damit dieses Verfahren sicher funktionieren kann, darf eine automatische Speicherbereinigung während der Ausführung einer dergestalt optimierten Methode die betroffenen Objekte nicht im Speicher verschieben, da mindestens die Objektadressen sonst ungültig würden. Die Adressen der Methodentabellen sind nur betroffen, wenn auch Klassendeskriptoren einer automatischen Speicherbereinigung und -verdichtung unterliegen, was aber in den meisten Sprachen nicht der Fall ist.

Eine weitere Optimierung könnte Standardtechniken aus der LISP-Welt aufgreifen. Bei der Implementierung von LISP werden bestimmte Bits der Maschinenwörter oft für einen sogenannten *tag* reserviert, der den Datentyp des Maschinenworts (Integer, Cons-Zelle etc.) unmittelbar identifiziert [Ros88]. LISP-Maschinen hatten spezielle Hardware, die diese Tags bei jeder Operation automatisch abgeprüft haben. Auch der SPARC-Prozessor beherrscht eine tagged Integer-Arithmetik. Eine Übertragung dieser Ideen in die Welt der objektorientierten Programmiersprachen könnte darin bestehen, daß man den _class Zeiger (den dynamischen Typ) aus dem Objekt in die Referenz selbst verlegt. Eine 32 Bit breite Referenz könnte beispielsweise in 16 Bit für die Adresse des Klassendeskriptors und 16 Bit für die Objektadresse aufgeteilt werden. Ob eine solche gepackte Darstellung einer Referenz Sinn macht, hängt von Faktoren wie Anzahl der Klassen und Objekte im System, Hardwareunterstützung, Registerbreite, begrenzter Adressraum des Prozessors ab. Nach [Fra99] besitzt beispielsweise der DSP TI TMS320C2x nur einen 16 Bit breiten externen Adressbus. Dies mag für den Einsatz objektorientierter Software als nicht ausreichend erscheinen, aber es hat auch schon erfolgreiche Versuche gegeben, Smalltalk auf 8 Bit Prozessoren wie den Z80 oder 6502 zu portieren (TinyTalk [Kra83, Kapitel 1]).

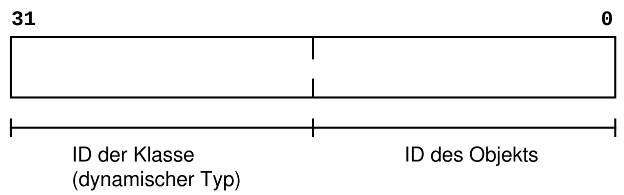

Abbildung 7.1: Beispiel für eine gepackte Referenz

Die in diesem Abschnitt beschriebenen Techniken lassen sich auch kombinieren. Eine gepackte Referenz kann beispielsweise eine direkte Objektadresse beinhalten, während die ID der Klasse als Index in eine Klassentabelle mit Zeigern auf Deskriptoren interpretiert werden könnte. Der Vorteil einer gepackten Referenz ist die Ersparnis des _class Zeigers in allen Objekten.

7.2 Statische Allokation

Die dynamische Allokation von Objekten zur Laufzeit ist immer mit einem Overhead für die Speicheranforderung und -bereinigung verbunden. Insbesondere für temporäre Objekte, die innerhalb von Methoden lokal begrenzt zum Einsatz kommen und danach nicht mehr referenziert werden, kann dieser Overhead durch *statische Allokation* eingespart werden. Ein Objekt wird beispielsweise in Oberon-2 statisch alloziert durch eine Deklaration der Form

```
var c : C;
var store : array 100 of C;
```

In dieser Form definierte Variablen *beinhalten* das Objekt. Sie sind *nicht polymorph*, die Untertypregel kommt nicht zur Anwendung. Der statische und der dynamische Typ des Objekts sind immer identisch. Eine willkommene Konsequenz ist, daß Nachrichtenausdrücke mit statisch allozierten Empfängerobjekten statisch gebunden werden können, sie können also in direkte Prozeduraufrufe übersetzt werden. Weiterhin besteht theoretisch die Möglichkeit, in statisch allozierten Objekten den _class Zeiger einzusparen. Allerdings kann dies in Methoden zu Mißverständnissen beim Zugriff auf Instanzvariablen führen, da deren Offsets den Zeiger ja miteinschliessen. Abhilfe könnte durch einmaliges Dekrementieren der Objekt-Speicheradresse um die Grösse des _class Zeigers vor dem ersten Zugriff geschaffen werden.

Die Semantik der Zuweisung wandelt sich zu einer Kopiersemantik, wenn mindestens eine statisch allozierte Variable beteiligt ist. Referenzen auf statisch allozierte Objekte sollten kein Problem sein, wenn die Sprache diese generell als Zeiger implementiert. Auf der rechten Seite einer Zuweisung an eine statisch allozierte Variable darf keine Referenz stehen, es sei denn man sichert sich durch eine Typzusicherung ab.

Die statische Allokation normalerweise dynamisch allozierter Objekte wird in der Literatur auch als *"unboxing"* bezeichnet. Es ist eine naheliegende Idee, das unboxing von Objekten dem Übersetzer zu überlassen. Ich habe zu diesem Thema keine Arbeiten gefunden, lediglich Hinweise auf Pointer-Alias Analyse. In der Dissertation von Martin Trapp [Tra00] sollen Verfahren entwickelt worden sein, die Arbeit ist bis jetzt aus patentrechtlichen Gründen aber nicht einsehbar. Zur Zeit bearbeitet ein Mitarbeiter der Gruppe um Prof. Dr. Goos in Karlsruhe[1], Florian Liekweg, das Thema in seiner Dissertation[2].

7.3 Beschränkt polymorphe statische Allokation

In diesem Abschnitt wird eine Erweiterung des Typsystems von Oberon-2 vorgestellt. Die originale Sprachdefinition von Oberon-2 findet sich in [Mös98, Anhang A]. Neu definiert wird der *beschränkt polymorphe Typ*. Ziel der Erweiterung ist es, die Deklaration von *statisch allozierten* Variablen und Arrays zu ermöglichen, die Objekte einer eingeschränkten Menge von Klassen aufnehmen können, die also

[1] http://i44s11.info.uni-karlsruhe.de:80/

[2] persönliche Mitteilung

beschränkt polymorph sind. Die jeweiligen Vorteile von statischer Allokation und Polymorphismus können so kombiniert werden. Die aus Pascal bekannten Varianten Records können dabei als eine nicht-objektorientierte Vorform interpretiert werden.

7.3.1 Syntax

Oberon-2 kennt neben den Standardtypen den Array-Typ, den Record-Typ, den Pointer-Typ und den Prozedur-Typ:

Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.

Als neuen Grundtypen führe ich den *beschränkt polymorphen Typ* ein[3]:

BoundedType = "(" RecordType {"," RecordType} ")".

Da anscheinend der Aufzählungstyp aus Oberon-2 gestrichen worden ist, kann dessen Pascal-Syntax in der obigen Definition gefahrlos reaktiviert werden. Der beschränkt polymorphe Typ soll nur in Variablen-Deklarationen und als Bestandteil von Array-Typen erlaubt sein:

BPType = Qualident | ArrayType | RecordType | ... | ProcedureType | BoundedType.
VariableDeclaration = IdentList " : " BPType.
ArrayType = ARRAY [Length {"," Length}] OF BPType.

Die Feldliste von Record-Typen wird modifiziert zu:

FieldList = [IdentList ":" BPType].

Auch formale Parameter von Prozeduren lassen sich mit einem BoundedType typisieren:

FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
FPSection = [VAR] ident {"," ident} ":" BPType.

Beispiel 7.3.1 *Anwendung der neuen Syntax*

```
var store : array 100 of C;             // 100 C-Objekte, alte Syntax
var store : array 100 of (C);           // C und alle Unterklassen
var store : array 100 of (C, D, E, F);  // C und bestimmte Unterklassen von C
var temp : (C, D);                      // nur C- und D-Objekte
```

[3] In Oberon-2 werden Klassen durch Record-Typen deklariert.

7.3.2 Semantik

Ein Übersetzer muß den BoundedType in der Typprüfung und in der statischen Speicherzuteilung berücksichtigen.

Die Bildung eines BoundedTypes soll den folgenden Regeln genügen:

- Die erste Klasse in der Aufzählung ist die Basisklasse. Diese legt den *primären statischen Typ* des BoundedTypes fest. Die in der Aufzählung folgenden Klassen sind Unterklassen der Basisklasse *(sekundäre statische Typen).*
- Besteht die Aufzählung nur aus einer Klasse, wird diese als die Basisklasse betrachtet und alle im System bekannten Unterklassen werden automatisch mit eingeschlossen.

Die Regeln bedeuten, daß Variablen und Arrays nun mehr als einen statischen Typ besitzen können. Die statischen Typen müssen sich nur in einer Vererbungsbeziehung zueinander befinden. Die allgemeine Untertypregel aus Abschnitt 4.2.6 wird eingeschränkt auf eine genau definierte Menge von Typen. Die Symboltabellen des Übersetzers sind entsprechend zu erweitern. Bei der Typprüfung von Nachrichtenausdrücken mit statisch allozierten Empfängerobjekten ist entscheidend, ob die Selektoren vom *primären* statischen Typ verstanden werden. Die Methodenauswahlregel bleibt unverändert. Es darf nur auf die im primären statischen Typ deklarierten Instanzvariablen zugegriffen werden.

Zusätzlich zur normalen Untertypregel gilt eine eingeschränkte Variante:

Eingeschränkte Untertypregel: Wird auf einer Eingabeposition (Funktionseingabeparameter, rechte Seiten von Zuweisungen) ein Objekt eines *beschränkten Typs* verlangt, so dürfen nur statisch allozierte Objekte übergeben werden, die mit ihrem statischen Typ zu einem der im BoundedType deklarierten primären oder sekundären Typen identisch sind.

Die Speicherzuteilung gestaltet sich recht einfach. Für alle in einem BoundedType aufgezählten Klassen ist die Grösse der Objekte bekannt. Der Speicherbedarf des BoundedTypes bestimmt sich als das Maximum dieser Grössen. Auf diese Weise ist für jedes erlaubte Objekt genügend Platz vorhanden, unter Umständen kommt es aber auch zu Leerstellen. Die Variable (das Array) wird mit einem Objekt (Objekten) der Basisklasse vorinitialisiert. Für Zuweisungen gilt das in Abschnitt 7.2 gesagte.

Beispiel 7.3.2 *Die beschränkt polymorph statisch allozierte Variable temp*

```
var temp : (C, D);
```

wird mit einem C-Objekt vorinitialisiert, kann aber auch D-Objekte aufnehmen:

```
var obj : D;
temp := obj;
```

Da es sich nicht um Referenzvariablen handelt, wird das Objekt obj nach temp kopiert.

Beispiel 7.3.3 *Array für 7 Objekte der Klasse B*

```
var store : array 7 of B;
```

Das Array wird mit sieben B-Objekten vorinitialisiert. Im Array können nur B-Objekte gespeichert werden.

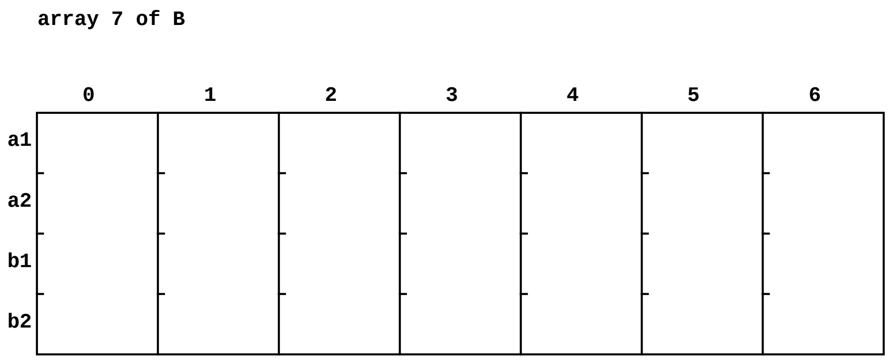

Abbildung 7.2: Speicherbild eines normalen Arrays

Beispiel 7.3.4 *Beschränkt polymorphes Array für 7 Objekte der Klassen A, B, C, D*

```
var store : array 7 of (A, B, C, D);
var     c : C;

store[2] := c;     // usw.
```

Im Array können Objekte der Klassen A, B, C, D gespeichert werden. Das Array wird mit sieben A-Objekten vorinitialisiert.

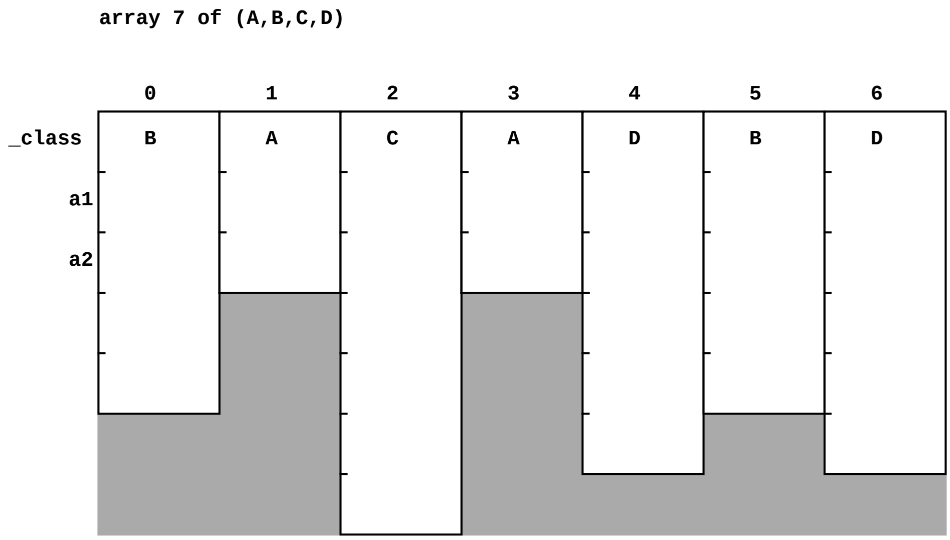

Abbildung 7.3: Speicherbild eines beschränkt polymorphen Arrays

Die Abbildungen 7.2 und 7.3 zeigen im Vergleich ein statisch alloziertes Array für sieben B-Objekte und ein beschränkt polymorph statisch alloziertes Array für sieben Objekte der Klassen A, B, C oder D.

Das Array aus Beispiel 7.3.3 belegt 28 Bytes im Speicher. Die _ class Zeiger der Objekte können eingespart werden, da bereits zur Übersetzungszeit der statische und dynamische Typ als identisch bekannt sind. Nachrichtenausdrücke können statisch gebunden werden.

Das Array aus Beispiel 7.3.4 belegt 7*7=49 Bytes im Speicher. Das potentiell grösste Objekt ist eines der Klasse C, welches 7 Bytes benötigt. Die _ class Zeiger können nicht eingespart werden, da zur Laufzeit die dynamischen Typen der Objekte vom statischen Typ A abweichen können. Nachrichtenausdrücke sind spät zu binden. Der für das Array reservierte Speicher wird in der Regel nicht voll genutzt, es kommt für A, B und D-Objekte zu Leerstellen.

7.3.3 Zwischenbewertung

Insbesondere das Beispiel 7.3.4 mit Abbildung 7.3 macht deutlich, daß die gesparten Zeitkosten für die dynamische Allokation und Speicherbereinigung durch erhöhte Platzkosten bezahlt werden müssen. Der Vorteil liegt darin, daß diese worst-case Platzkosten aber bereits zur Übersetzungszeit bekannt sind.

Die beschränkt polymorphe statische Allokation eignet sich hauptsächlich zur Definition von globalen Datenstrukturen und zur Definition lokaler temporärer Objekte innerhalb von Methoden.

7.4 Kontext und Bewertung

Die statische Allokation von Objekten spart insbesondere Zeit für Methodensuche und dynamische Speicherzuteilung. Sie sollte daher in Sprachen für eingebettete Systeme möglich sein. Die beschränkt polymorph statische Allokation ist flexibler, aber auch hier wird die Zeit für dynamische Speicherzuteilung eingespart. Letztere muß noch auf ihre Praxistauglichkeit hin untersucht werden. Auch sind noch weitere Studien bezüglich einer sicheren und flexiblen Semantik erforderlich.

7.5 Literaturhinweise

Rose [Ros88] zeigt Designmöglichkeiten für Dispatch-Code Sequenzen auf, wobei die Anforderungen von Sprachen wie CommonLISP im Vordergrund stehen.

Die statische Allokation von Objekten in Objective-C wird am Rande in [CN91] erwähnt. Die beschränkt polymorphe statische Allokation ist meine eigene Erfindung, Literatur vergleichbaren Inhalts ist mir nicht bekannt.

Kapitel 8

Übersetzerinfrastrukturen

In diesem Kapitel wird die Grundstruktur eines Übersetzers wiederholt, danach werden Zwischendarstellungen für objektorientierte Sprachen und Übersetzungsmodelle diskutiert.

8.1 Grundstruktur eines Übersetzers

Ein Übersetzer für objektorientierte Sprachen wird im wesentlichen die gleiche Struktur aufweisen wie ein Übersetzer für imperative Sprachen.

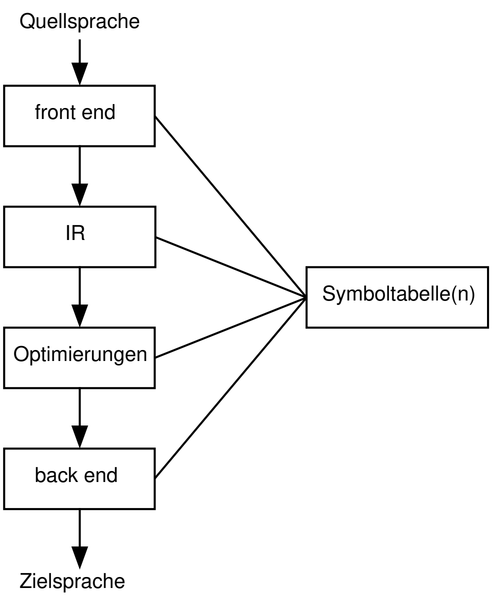

Abbildung 8.1: Grundstruktur eines Übersetzers

Das *front end* des Übersetzers übersetzt den Quelltext nach lexikalischer, syntaktischer und semantischer Analyse in eine Zwischendarstellung (intermediate representation). Die IR wird durch optionale *Optimierungsphasen* weiter transformiert und schließlich vom *back end* des Übersetzers in die Zielsprache übersetzt [ASU86].

Die Optimierungsphase kann unterteilt werden in Phasen für

- globale und high-level Optimierungen, die die objektorientierten Sprachmerkmale auf imperative Merkmale reduzieren, dabei Methodentabellen verdichten und Gelegenheiten zur statischen Bindung erkennen und nutzen;
- Standardoptimierungen;
- zielmaschinenspezifische Optimierungen beispielsweise der Speicherzugriffe in DSPs.

Alle Phasen erweitern und nutzen eine Symboltabelle, die alle im Quelltext benutzten Bezeichner aufsammelt und diesen weitere Attribute zuordnet, wie beispielsweise Information über Typ, Geltungsbereich und zugeordneten Speicher. Eine Symboltabelle sollte objektorientierte Sprachmerkmale wie Klassen und ihre Instanzvariablen und -Methoden repräsentieren können, in dem beispielsweise für jede Klasse eine eigene Untertabelle angelegt wird.

8.2 Zwischendarstellungen

Einen guten Überblick über wichtige Zwischendarstellungen gibt die Diplomarbeit von Holger Kienle [Kie98]. In diesem Abschnitt sollen besonders die Zwischendarstellungen OSUIF, Vortex IL und Firm diskutiert werden.

8.2.1 OSUIF

OSUIF ist eine Erweiterung des Stanford University Intermediate Format SUIF in Version 2.0 zur Unterstützung objektorientierter Programmiersprachen.

Der SUIF 1.0 Kernel definiert eine Zwischendarstellung, die vergleichbar zu einem Abstract Syntax Tree (AST) ist. Jedes Element dieser IR bzw. jeder Knoten eines AST ist implementiert als eine C++ Klasse innerhalb einer objektorientierten Hierarchie von Datenstrukturen. Die IR hat sprachunabhängige high-level Darstellungen für Schleifen, bedingte Anweisungen und Array-Zugriffe. Diese können in eine äquivalente low-level Darstellung transformiert werden, die RISC-ähnliche Instruktionen durch ein Quadrupel-Format repräsentiert. Die Übersetzung erfolgt in getrennten Phasen, die einzelnen Phasen (SUIF passes) kommunizieren ihre Ergebnisse über Dateien.

Das zu übersetzende Programm liegt vor als eine Sammlung mehrerer Quelltextdateien. Es existiert eine entsprechende Hierarchie von Symboltabellen. Eine globale Symboltabelle speichert in allen Quelldateien global sichtbare Bezeichner. Jeder Quelltextdatei ist eine file symbol table zugeordnet, die nur in dieser Datei sichtbare Bezeichner aufnimmt. Die ausschließlich innerhalb einer Prozedur sichtbaren Bezeichner werden in procedure symbol tables gesammelt. Jeder Prozedurkörper wird repräsentiert als ein eigener AST.

Symboltabellen nehmen drei verschiedene Arten von Einträgen auf: Typen, Bezeichner und Variablendefinitionen. SUIF Typen beschreiben die Typen von Variablen und die Ergebnistypen von Instruktionen. Sie kennen insbesondere ihre Speichergrösse in Bits. Bezeichner sind Prozedurbezeichner, Labelbezeichner und Variablenbezeichner. Variablendefinitionen allozieren Speicher für Variablen, die nicht auf dem Stack abgelegt werden.

SUIF 2.0 ist eine komplette objektorientierte Neuimplementierung des SUIF 1.0 Kernels in C++, wobei die grundlegenden Prinzipien sich nicht geändert haben, aber um neue Funktionalität ergänzt wurden. Der neue SUIF 2.0 Kernel führt fünf Abstraktionsebenen über die Klassenbibliotheken sty, wallow, zot, pyg und suif ein. *sty* definiert generische Datenstrukturen wie Arrays, Listen, Hashtabellen. *wallow* implementiert architekturunabhängie Ein-/Ausgabe für verschiedene primitive und Fließkommatypen. *zot* definiert die Basisklasse für alle SUIF-Objekte. *pyg* beschreibt (abstrakte) Basisklassen der SUIF IR für Typen, Symboltabellen, Instruktionen, Anweisungen etc. *suif* konkretisiert diese Basisklassen in Klassen für Integer-Typen, arithmetische Instruktionen, Kontrollflußanweisungen usw. *suif* beinhaltet alle IR Konstrukte, die zur Darstellung imperativer Sprachen benötigt werden.

SUIF 2.0 beinhaltet insbesondere einen neuen Mechanismus zur Erweiterung der IR. Diese Erweiterungen heissen Dialekte.

Eine solcher Dialekt ist Object SUIF (OSUIF). OSUIF implementiert high-level Darstellungen für Klassen mit Klassen- und Instanzmerkmalen, Beziehungen zwischen Klassen, Nachrichtenausdrücke (dynamische Methodenaktivierungen), Objekterzeugung und Löschung, ein erweitertes Typsystem und Ausnahmebehandlung. Klassen werden repräsentiert durch die OSUIF-Klasse class_type. Diese hat zwei Symboltabellen, eine für Instanzmerkmale und eine für Klassenmerkmale. Zu einem class_type gehören inheritance_links, die auf Ober- und Unterklassen verweisen können. Die Symboltabellen eines class_types beinhalten anfänglich nur die Merkmale, die in der entsprechenden Klasse neu deklariert werden. Erst ein generischer OSUIF lowering pass propagiert Instanzmerkmale entlang der Klassenhierarchie und komplettiert so die Symboltabellen um die ererbten Merkmale. Andere OSUIF lowering Phasen erzeugen Virtual Function Tables und den entsprechenden Dispatch Code.

Nach [Kie98] befindet OSUIF sich noch in der aktiven Entwicklung. Von der geplanten Funktionalität fehlt noch viel. OSUIF hat demonstriert, daß SUIF 2.0 sich erweitern läßt, wobei diese Erweiterungen aber hohe Ansprüche an die Programmierung stellen.

8.2.2 Vortex

Vortex [DDG+96] ist ein an der University of Washington, Seattle, USA entwickelter experimenteller Übersetzer, der speziell zur Erforschung von Techniken zur Optimierung objektorientierter Programmiersprachen gedacht ist. Verschiedene Front-Ends von Vortex übersetzen Cecil, C++, Modula-3 und Java Byte-Code in die Vortex Intermediate Language. Die Optimierungsphase nutzt diverse high-level Optimierungen zur statischen Bindung von Nachrichtenausdrücken (interprocedural class analysis, class hierarchy analysis, receiver class prediction, customization and specialization). Das Back-End generiert entweder C- oder SPARC-Code. Vortex selbst ist in Cecil programmiert. Zentraler Bestandteil der Vortex Infrastruktur ist ein Iterative Dataflow Analysis Framework.

Zur textuellen Vortex Intermediate Language [Gro98a] gehören neben traditionellen low-level Operationen ausgewählte high-level Operationen, die statische Typinformation, Klassen, Nachrichtenausdrücke, Zugriffe auf Instanzvariablen, Typtests und Typzusicherungen und Objekterzeugungen explizit machen. Diese high-level Konstrukte werden in den Übersetzungs- und Optimierungsphasen fortschreitend in low-level Konstrukte transformiert. Die Vortex IL unterstützt insbesondere die anspruchsvolle Sprache Cecil durch Mehrfachvererbung von Klassen, Multiple Dispatching, Generische Funktionen und Exceptions. Ausführbarer Code wird repräsentiert als eine Folge von Drei-Adress-Code Anweisungen, wobei die übliche Sammlung an arithmetischen, logischen, Zeiger-, Verzweigungs- und direkten und indirekten Prozeduraufruf-Anweisungen zur Verfügung steht. Nachrichtenausdrücke werden in eine send-Anweisung transformiert:

C++	**Vortex IL**
`Shape* s = ...;` `s->draw(color);`	`send draw__5Shape_int(s, color);`

Nachrichtenausdrücke werden abhängig von der Quellsprache implementiert durch Methodentabellen (table_send) oder Polymorphe Inline Caches (send).

8.2.3 Firm

Firm ist von Martin Trapp im Rahmen seiner Dissertation [Tra00] entwickelt worden. Diese Dissertation konnte für die vorliegende Diplomarbeit nicht ausgewertet werden, da der Autor sie aus patentrechtlichen Gründen noch nicht freigegeben hatte. Die folgenden Angaben sind der Homepage von Martin Trapp[1] entnommen.

Firm nutzt die Static Single Assignment Form (SSA) zur Darstellung *skalaren* Datenflusses. Objektorientierte Sprachen hingegen repräsentieren Objekte als zusammengehörige Zustandsspeicher, die durch Speicherzugriffe modifiziert werden. Dies ist in SSA-Form nicht darstellbar. Firm führt das Konzept des Speicherzustands

[1] http://i44s11.info.uni-karlsruhe.de:80/~trapp/

(store) ein. Abhängigkeiten zwischen Speicherzugriffen werden als Datenabhängigkeiten zwischen stores modelliert. Stores können (beinahe) wie skalare Werte behandelt werden.

Kontrollfluß wird auch in Firm durch Kontrollflußgraphen (CFG) repräsentiert, wobei allerdings die streng sequentielle Anordnung von Operationen innerhalb eines Basisblocks durch eine partielle Ordnung gemäß der Datenabhängigkeiten ersetzt wird.

Der besondere Beitrag der Arbeit sind neben der innovativen IR leistungsfähige interprozedurale Analysetechniken für objektorientierte Sprachen, welche Ausdrücke, Zeiger, Typen und Speicherinhalte statisch eingrenzen können. Diese Analysen ermöglichen u.A. die statische Bindung von Nachrichtenausdrücken, das Inlining von Methoden und die statische Allokation von Objekten (unboxing).

8.3 Übersetzungsmodelle

Ein Übersetzer operiert unter bestimmten *Annahmen (world assumptions)* bezüglich der äußeren Welt. Diese Annahmen können die Effizienz des generierten Codes und die für einen Übersetzerdurchlauf benötigte Zeit entscheidend beeinflussen.

Interaktive benutzerorientierte Umgebungen wie Smalltalk benötigen Übersetzer, die schnelle Antwortzeiten garantieren und lokal beschränkte Modifikationen erlauben, ohne das gleich sämtliche Klassen der Klassenhierarchie neu übersetzt werden müssen. Im übersetzten Byte-Code bleiben Bezüge auf andere Klassen symbolisch, die erst zur Laufzeit durch dynamische Bindung aufgelöst werden. Optimierungen, die Information über die gesamte Klassenhierarchie benötigen, können nicht durchgeführt werden.

Im Gegensatz dazu stehen ergebnisorientierte Übersetzer, die den Quelltext in ein möglichst kompaktes und schnell ausführbares Programm transformieren sollen. Im optimalen Fall stehen dem Übersetzer ausführliche Angaben über die gesamte Klassenhierarchie zur Verfügung, so daß symbolische Bezüge auf Klassen, Instanzvariablen, Methoden durch konkrete Offsets, Speicher-Adressen von Deskriptoren und Prozeduren, indirekte und direkte Prozeduraufrufe ersetzt werden können. Wenn leistungsfähige Optimierungen lange Übersetzungszeiten zur Folge haben, wird dies gerne toleriert. Allerdings kann im worst-case jede kleinste Änderung im Quelltext eine komplette Neuübersetzung erforderlich machen.

Nach [PS94, Vit95] können folgende Übersetzungsmodelle unterschieden werden:

Open World: Das Programm ist unterteilt in getrennte Einheiten, die unabhängig voneinander übersetzt werden können;

Upward Closed: Das Programm ist unterteilt in Einheiten wie Klassen oder Pakete. Die Übersetzung einer Einheit ist abhängig von Oberklassen und anderen Klassen, die bei der Implementierung der Einheit verwendet wurden. Eine Neuübersetzung der Einheit wird nur bei einer Änderung der Schnittstellen der genannten Klassen erforderlich. Die getrennt übersetzten Einheiten werden erst durch einen Linker zum kompletten Programm zusammengefügt. Dieses Modell wird von Sprachen wie C++ oder Oberon-2 realisiert;

Downward Closed: Die Übersetzung einer Einheit bezieht Information über Unterklassen mit ein. Die in Kapitel 6.4 beschriebenen Sprachkonstrukte geben dem Übersetzer Hinweise über Unterklassen, die dieser zur Optimierung der Methodensuche nutzen kann, ohne das Semantik und Sicherheit beeinträchtigt werden.

Closed World: Der Übersetzer bearbeitet immer den kompletten Quelltext inklusive des Quelltexts der Bibliotheken (whole-program compilation). Globale Optimierungen können sämtliche verfügbare Information zur Generierung effizienten Codes nutzen.

Smalltalk arbeitet mit dem Open-World Modell, Oberon-2 und Objective-C mit dem Upward-Closed Modell. Java, C++ und Dylan berücksichtigen ein Upward- und Downward-Closed Modell. Vortex ist ein Closed-World Compiler.

Eingebettete Systeme sind in der Regel geschlossen und nicht interaktiv. Es sollte daher letztlich ein Closed-World Compiler zum Einsatz kommen, der durch globale Optimierungen die beschränkten Zeit- und Platzressourcen am besten berücksichtigen kann. In früheren Phasen des Softwareentwicklungsprozesses hingegen ist auch der Einsatz offenerer Übersetzer denkbar und sinnvoll, da es während Design, Entwurf und Test mehr auf die Einhaltung der Spezifikation, Fehlerfreiheit des Programms und Bequemlichkeit durch kurze Übersetzungszeiten ankommt.

8.4 Literaturhinweise

Eine ausführliche Darstellung von SUIF 2.0 und OSUIF findet sich in [KH97, Kie98]. Vortex wird in [DDG+96] vorgestellt, die Grammatik der Vortex IL ausführlich in [Gro98a]. Zu Firm konsultiere man die Dissertation von Trapp [Tra00], sobald diese verfügbar ist. Brandis [Bra95] beschreibt die Guarded Static Single Assignment Form GSA zur Darstellung imperativer Sprachen.

Kapitel 9

Konklusion

Es ist eine grundsätzliche Entscheidung, ob man ein eingebettetes System mit objektorientierter Technologie realisiert. Es ist abzuwägen, ob die erhöhten Anforderungen durch dynamische Speicherverwaltung und Methodensuche oder die Vorteile durch Abstraktion und Wiederverwendung schwerer wiegen. Auch ist objektorientierte Technologie nicht für jeden Anwendungsfall geeignet.

Der Anwendungsbereich eingebetteter Systeme wurde in Kapitel 2 in Sicherheitsklassen eingeteilt. Objektorientierte Sprachen können die Anforderungen der Ebene SIL 1 erfüllen.

Die vorliegende Arbeit hat in Kapitel 4 zwei Grundformen von objektorientierten Sprachen identifiziert, solche mit statischem Typsystem und solche mit dynamischem Typsystem. Zur Programmierung eingebetteter Systeme sind Sprachen mit einem leistungsfähigen und strengen statischem Typsystem eindeutig zu bevorzugen, da diese sicher und effizient übersetzbar sind. Die Methodensuche zur Laufzeit kann nicht fehlschlagen und lässt sich auf einen indirekten Sprung über eine Tabelle reduzieren. Metaklassen erhöhen die Ausdruckskraft von objektorientierten Sprachen, sind aber auch mit erhöhtem Zeit- und Platzaufwand verbunden, so daß allenfalls Typdeskriptoren wie in Oberon-2 sinnvoll erscheinen. Wesentlich mehr Aufmerksamkeit sollte den generischen Typen und ihrer effizienten Übersetzung gewidmet werden, hier besteht noch hoher Forschungsbedarf.

Auf den wichtigen Unterschied zwischen Klasse und Typ wurde in Kapitel 3 ausführlich hingewiesen. Es wurde erläutert, daß moderne Sprachen wie Objective-C und Java diesen Unterschied vorteilhaft durch neue Sprachkonstrukte ausnutzen und dadurch auch die Nachteile der Mehrfachvererbung vermeiden.

In Kapitel 5 wurde beschrieben, wie Objekte, Klassen, Methoden, Vererbung und Methodensuche sich auf imperative Datenstrukturen wie den Verbund, die Reihung und den Zeiger reduzieren lassen. Diese Aspekte objektorientierter Sprachen lassen sich also mit Standardtechniken bewältigen. Für die komplexe Problematik der effizienten und realzeitfähigen dynamischen Speicherallokation- und bereinigung wurden in Anhang A Literaturhinweise gegeben.

Insbesondere der Berücksichtigung beschränkter Zeit- und Platzressourcen und des Echtzeitbetriebs dienen die in Kapitel 6 und 7 beschriebenen Optimierungen der Methodensuche und der Objektspeicherung. Es wurde die vom Autor entwickelte beschränkt polymorph statische Allokation skizziert, die aber noch auf ihre Praxistauglichkeit hin untersucht werden müsste. Auch sind noch weitere Studien bezüglich einer sicheren und flexiblen Semantik erforderlich. Besonders effektiv zur Optimierung statisch typisierter Sprachen sind die statischen Analysetechniken (Kapitel 6.3) und die Verfahren zur Verdichtung von Methodentabellen (Kapitel 6.1).

Eine objektorientierte Sprache für eingebettete Systeme sollte folgende Eigenschaften haben:

- sie sollte hybrid sein, praktische Erwägungen lassen eine Erweiterung der Sprache C sinnvoll erscheinen;
- die Sprache sollte ein leistungsfähiges strenges statisches Typsystem besitzen und auch generische Typen unterstützen;
- die Sprache sollte nur einfache Vererbung, aber dafür Schnittstellen wie in Objective-C und Java kennen;
- Nachrichtenausdrücke sind spät zu binden, aus Gründen der Lesbarkeit ist die Objective-C Syntax zu bevorzugen. Bei der Implementierung der Methodensuche sind neben VTBL auch Verfahren wie CT-95 in Erwägung zu ziehen;
- Referenzen sollten als Zeiger implementiert werden, was durch die Speicherverwaltung zu berücksichtigen ist;
- spezielle Sprachkonstrukte wie final- und sealing-Direktiven sollten die statische Bindung und die statische Allokation erleichtern;
- von Metaklassen ist abzuraten, insbondere die Erzeugung neuer Objekte sollte über ein Schlüsselwort wie new fester Bestandteil der Sprache sein;
- von Konstruktoren und Destrukturen wird ebenfalls abgeraten, da sie eine Sprache unnötig kompliziert machen können (siehe Java und C++). Die Initialisierung von Objekten kann auf zwei Arten erfolgen: Standardwerte für Instanzvariablen werden in der Klassendefinition angegeben, weitere Initialisierung erfolgt per Konvention über Instanzmethoden, die beispielsweise init, initWith: heissen können (siehe Objective-C);
- der Übersetzer sollte ein closed-world compiler sein, dem sämtlicher Quelltext inklusive Klassenbibliotheken für globale Optimierungen vorliegt.

Existierende Sprachen, die als Vorbild dienen können, sind Oberon-2, Sather-K, PEARL und Objective-C.

Insbesondere für eingebettete Systeme mit harten Echtzeitanforderungen könnte das nähere Studium der in Kapitel 2.5 beschriebenen synchronen (objektorientierten) Sprachen sinnvoll sein. Möglicherweise sind diese Sprachen, die ja auch einer formalen Verifikation zugänglich sind, für Anforderungen bis Sicherheitsklasse SIL 2 zielführender als die herkömmliche Kombination aus Echtzeit-Betriebssystem und Programmen.

Anhang A

Speicherverwaltung

Objektorientierte Programme haben ein sehr dynamisches Laufzeitverhalten. Es werden in der Regel sehr viele Objekte erzeugt, die aber nicht während der gesamten Laufzeit des Programms verwendet werden. Da die Ressource Speicher beschränkt ist, müssen nicht mehr benötigte Objekte aus dem Speicher auch wieder entfernt werden. Eine Speicherverwaltung benötigt aber auch zusätzliche Rechenzeit, die dem eigentlichen Programm nicht zur Verfügung steht. Insbesondere für eingebettete Systeme ist also eine effiziente Speicherverwaltung und -bereinigung von besonderem Interesse.

A.1 Anforderungen

Eine Speicherverwaltung für objektorientierte Sprachen muß grundsätzlich zwei Aufgaben erledigen:

1. die effiziente *Allokation* von Speicher für Objekte, dabei ist eine Fragmentierung des Objektspeichers möglichst zu vermeiden,
2. die effiziente *Speicherbereinigung (garbage collection)*, die sich insbesondere unter Echtzeitbedingungen deterministisch verhalten sollte.

Für die Speicherbereinigung kann man folgende Ansätze unterscheiden:

Manuell: Der Programmierer ist selbst für die Löschung nicht benötigter Objekte zuständig. Dieses Vorgehen ist durchaus fehleranfällig und kann Memory Leaks zur Folge haben.

Automatisch: Eine automatische Speicherbereinigung untersucht in gewissen Abständen und mit bestimmten Verfahren den Objektspeicher und entfernt automatisch unnötige Objekte.

Halbautomatisch: Der Programmierer ist letztlich verantwortlich, wird aber bis zu einem gewissen Grad vom Laufzeitsystem unterstützt. Dieser Ansatz könnte für eingebettete Systeme eine interessante Alternative darstellen.

A.2 Literaturhinweise

Das Thema Speicherverwaltung ist umfangreich genug, um mehrere Diplomarbeiten füllen zu können. Es soll daher nur auf Literatur verwiesen werden, die mögliche Einstiegspunkte für weitere Forschungen bietet.

Ein Lehrbuch zum Thema ist [Jon96].

Boehm und Weiser [BW88, Boe93] beschreiben ihren sehr bekannten Garbage Collector, mit dem man Sprachen wie C und C++ nachträglich mit einer automatischen Speicherbereinigung ausstatten kann.

Die besonderen Anforderungen einer Realzeitumgebung an die Speicherbereinigung werden in [PB98] diskutiert. Weiterhin wird dort ein Garbage Collector für eine freie Java VM beschrieben.

Nilsen [Nil96, Nil98] beschreibt Konzepte und Modifikationen, die Java und ihre Speicherverwaltung echtzeitfähig machen sollen.

Embedded Eiffel [Int98] kontrolliert die Realzeiteigenschaften der Garbage Collection durch eine per-thread GC, die sich in zeitkritischen Abschnitten auch abschalten läßt.

Für das auf C++ basierende Betriebssystem EPOC[1] werden in [TDS+00] detaillierte Richtlinien für eine sichere manuelle Speicherverwaltung beschrieben.

Die halbautomatische Speicherverwaltung ist ein besonderes Merkmal der Apple Objective-C Frameworks. Es sei daher für weitere Einzelheiten auf die entsprechenden Webseiten verwiesen[2] [3].

[1] http://www.symbian.com
[2] http://www.stepwise.com/Articles/Technical/MemoryManagement.html
[3] http://developer.apple.com/techpubs/macosx/macosx.html

Anhang B

Existierende Systeme

In folgenden Produkten oder Forschungsprototypen werden objektorientierte Sprachen eingesetzt:

EPOC32: EPOC [TDS$^+$00] ist ein von Psion/Symbian[1] entwickeltes Betriebssystem für PDAs. EPOC wird mit einer sorgfältig definierten Untermenge von C++ programmiert und beinhaltet auch leistungsfähige Standardapplikationen zur Textverarbeitung und Terminverwaltung etc. Die Konzepte von EPOC sind sehr modern und genügen höchsten Anforderungen der Software-Technologie.

Java: Die Beurteilung des tatsächlichen Stands der Dinge bezüglich Java in eingebetteten Systemen fällt mir schwer, siehe aber die Webseiten von SUN zur eingebetteten virtuellen Maschine KVM, Hewlett-Packard [Hew98] und Nilsen zu PERC [Nil98].

Eiffel: Die Firma von Bertrand Meyer, Interactive Software Engineering, hat Embedded Eiffel entwickelt. In [Int98] werden Erfahrungen berichtet, die man bei Hewlett-Packard damit gemacht hat.

Oberon-2: Die ETH Zürich hat eine Spin-Off Firma namens Oberon Microsystems[2] gegründet, die ein Echtzeitbetriebssystem JBED mit innovativer Ablaufplanung und eine auf Oberon-2 basierende Sprache namens Component Pascal vertreibt. Weiterhin ist man in der Lage, Java direkt in Maschinensprache zu übersetzen.

Elate RTOS: Elate ist das Echtzeitbetriebssystem der Tao Group[3]. Programme werden in VPCode geschrieben. VP ist ein Virtual Processor, ein imaginärer 32 bit RISC Prozessor. Bekannt geworden ist man durch die Produktfamilie *intent* Multimedia Toolkit und *intent* Java Technology Edition.

[1] http://www.symbian.com
[2] http://www.oberon.ch
[3] http://www.tao-group.com

Smalltalk: Auch Smalltalk kommt in eingebetteten Systemen zur Anwendung! Siehe dazu [Tho95] für eine allgemeine Darstellung mit Hinweisen auf Texas Instruments und Tektronix; [EMP+96] beschreiben die Realzeitsprache RealTimeTalk und Object Technology International [Obj98] liefern ENVY/Embedded Smalltalk unter anderem für IBM PowerPC Systeme.

Literaturverzeichnis

[ABC$^+$95] O. Agesen, L. Bak, C. Chambers, B. W. Chang, U. Hölzle, J. Maloney, R. B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual.* Sun Microsystems, Inc. and Stanford University, 1995.

[AG98] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, second edition, 1998.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating Virtual Function Calls in C++ Programs. In *ECOOP'96*, Lecture Notes in Computer Science 1098, pages 142–166. Springer-Verlag, 1996.

[App96] Apple Computer Inc. *The NewtonScript Programming Language.* Apple Computer Inc, 1996.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK, 1998.

[AR92] P. André and J. C. Royer. Optimizing Method Search with Lookup Caches and Incremental Coloring. In *ACM SIGPLAN Notices OOPSLA'92 Proceedings*, 1992.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BH98] Bernhard Bauer and Rita Höllerer. *Übersetzung objektorientierter Programmiersprachen – Konzepte, abstrakte Maschinen und Praktikum Java-Compiler.* Springer, Berlin, 1998.

[Bla94] Günther Blaschek. *Object-Oriented Programming with Prototypes.* Springer-Verlag, 1994.

[BM98] R. Budde and M. Muellerburg. Validation und Verifikation in Synchronie: synchronousEifel. In *GMA-Kongress 98 Mess- und Automatisierungstechnik*, VDI-Berichte 1397, pages 299–308, Düsseldorf, 1998. VDI-Verlag.

[BMPS98] R. Budde, M. Muellerburg, A. Poigne, and K.-H. Sylla. Verlässliche eingebettete Systeme durch synchrone objektorientierte Technologie. *GMD-Spiegel*, 1:29–30, 1998.

[Boe93] Hans-Juergen Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28 of *ACM SIGPLAN Notices*, pages 197–206, June 1993.

[Bra95] Marc Michael Brandis. *Optimizing Compilers for Structured Programming Languages.* PhD thesis, ETH Zürich, 1995.

[Bud91] Timothy A. Budd. *An Introduction to Object-Oriented Programming.* Addison-Wesley, Reading, Mass., 1991.

[Bud98] Reinhard Budde. The Design and Programming Language synchronousEifel (sE-version 1.1). Technical report, GMD-SET-EES, Schloß Birlinghoven, D-53754 Sankt Augustin, Oktober 1998.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, 18(9):807–820, 1988.

[BW94] Alan Burns and Andrew J. Wellings. HRT-HOOD: A Design Method for Hard Real-time Ada. *Real-Time Systems*, 6:73–114, 1994.

[BW95] Alan Burns and Andrew J. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* Elsevier, 1995.

[BW97] Alan Burns and Andrew J. Wellings. *Real-time systems and their programming languages.* Addison-Wesley, second edition, 1997.

[CDG97] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. Technical Report TR-96-06-02, University of Washington, January 1997.

[Cha95] Craig Chambers. The Cecil Language Specification and Rationale: Version 2.0. Technical report, Department of Computer Science and Engineering, University of Washington, December 1995.

[Cha96] Craig Chambers. Synergies Between Object-Oriented Programming Language Design and Implementation Research. In *ISOTAS'96 Conference*, Kanazawa, Japan, March 1996.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, January 1990.

[CN91] Brad J. Cox and Andrew J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, second edition, 1991.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys*, 17(4):471–521, December 1985.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. VORTEX: An Optimizing Compiler for Object-Oriented Languages. In *ACM SIGPLAN Notices OOPSLA '96*, number 10, pages 83–100, October 1996.

[Dea96] Jeffrey Dean. *Whole-Program Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, 1996.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *ECOOP'95*, 1995.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast Interprocedural Class Analysis. In *ACM Symposium on Principles of Programming Languages POPL'98*, 1998.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message Dispatch on Pipelined Processors. In *ECOOP'95*, 1995.

[DMSV89] R. Dixon, T. McKee, P. Schweizer, and M. Vaughan. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *ACM SIGPLAN Notices OOPSLA'89 Proceedings*, 1989.

[Dri93a] Karel Driesen. Method Lookup Strategies in Dynamically Typed Object-Oriented Programming Languages. Master's thesis, Vrije Universiteit Brussel, 1993.

[Dri93b] Karel Driesen. Selector Table Indexing and Sparse Arrays. *ACM SIGPLAN Notices OOPSLA'93 Proceedings*, 28(10):259–270, 1993.

[DS84] L. Peter Deutsch and Alan M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, January 1984.

[EMP+96] Christer Eriksson, Jukka Maki-Turja, Kjell Post, Mikael Gustafsson, Jan Gustafsson, Kristian Sandstrom, and Ellus Brorsson. An Overview of RealTimeTalk, a Design Framework for Real-Time Systems. *Journal of Parallel and Distributed Computing*, 36(1):66–80, July 1996.

[ES92] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, 1992.

[FH98] A. H. Frigeri and W. A. Halang. Erhöhung inhärenter Sicherheit in der Echtzeitprogrammierung durch Objektorientierung. Technical report, FernUni Hagen, Lehrgebiet Informationstechnik, 1998.

[FKMW97] Neal Feinberg, Sonya E. Keene, Robert O. Mathews, and P. Tucker Withington. *The Dylan Programming Book*. Addison-Wesley Longman, 1997.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly and Associates, Inc., second edition, 1997.

[FM96] Paolo Ferragina and S. Muthukrishnan. Efficient Dynamic Method-Lookup in Object-Oriented Languages. In *Lecture Notes in Computer Science 1136*. Springer-Verlag, 1996.

[Fra95] Michael Franz. Protocol extension: A technique for structuring large extensible software-systems. *Software - Concepts and Tools*, 16:2:86–94, July 1995.

[Fra99] Björn Franke. Analysen und Methoden optimierender Compiler zur Steigerung der Effizienz von Speicherzugriffen in eingebetten Systemen. Master's thesis, Universität Dortmund, Fachbereich Informatik, Lehrstuhl XII, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[GHK95] Harald Gall, Manfred Hauswirth, and Rene Klösch. Objektorientierte Konzepte in Smalltalk, C++, Objective-C, Eiffel und Modula-3. *GI Informatik Spektrum*, 18(4):195–202, August 1995.

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

[Goo96] Gerhard Goos. *Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen*. Springer-Verlag, 1996.

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

[GR95] Adele Goldberg and Kenneth S. Rubin. *Succeeding with Objects, Decision Frameworks for Project Management*. Addison-Wesley, 1995.

[Gro95] David Grove. The Impact of Interprocedural Class Analysis on Optimization. In *CASCOM'95*, 1995.

[Gro98a] The Cecil Group. Vortex RTL Textual Description Grammar. Technical report, Department of Computer Science, University of Washington, December 1998.

[Gro98b] David Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[Hat95] Les Hatton. *Safer C : Developing Software for High-integrity and Safety-critical Systems*. McGraw-Hill, 1995.

[HC92] Shih-Kun Huang and Deng-Jyi Chen. Efficient algorithms for method dispatch in object-oriented programming systems. *Journal of Object-Oriented Programming*, 5(5):43–54, September 1992.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of ECOOP '91*, Lecture Notes in Computer Science 512, pages 21–38. Springer-Verlag, July 1991.

[Hew98] Hewlett-Packard. HP Embedded Virtual Machine. WWW, 1998. http://www.hpconnect.com/embeddedvm/evm.htm.

[HF97] W. A. Halang and A. H. Frigeri. Eine objektorientierte Erweiterung von PEARL 90. In P. Holleczek, editor, *PEARL 97 – Workshop über Realzeitsysteme*, Informatik aktuell, pages 31–40. Springer-Verlag, 1997.

[HFL+98] W. A. Halang, A. H. Frigeri, R. Lichtenecker, U. Steinmann, and K. Wendland. *Methodenlehre sicherheitsgerichteter Echtzeitprogrammierung*, volume FB 813 of *Schriftenreihe der Bundesanstalt für Arbeitsschutz Dortmund / Berlin.* Wirtschaftsverlag NW, 1998.

[HMP97] M. Hof, H. Mössenböck, and P. Pirkelbauer. Zero-Overhead Exception Handling Using Metaprogramming. *Lecture Notes in Computer Science 1338*, 1997.

[Höl94] Urs Hölzle. *Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming.* PhD thesis, Department of Computer Science, Stanford University, August 1994.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak - A Usable Smalltalk Written in Itself. In *Proceedings of OOPSLA '97*, volume 32 of *ACM SIGPLAN Notices*, pages 318–326, 1997.

[Int98] Interactive Software Engineering. Embedded Eiffel: High-performance object technology for the embedded world. WWW, February 1998. http://www.eiffel.com/announcements/97/embedded.html.

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley and Sons, 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted February 1997.

[KH97] Holger Kienle and Urs Hölzle. Introduction to the SUIF 2.0 Compiler System. Technical Report TRCS97-22, University of California, Santa Barbara., December 1997.

[Kie98] Holger Michael Kienle. A SUIF Java Compiler. Diplomarbeit, Universität Stuttgart, Fakultät Informatik, Germany, June 1998.

[Koe89] Andrew Koenig. How Virtual Functions Work. *Journal of Object-Oriented Programming*, 2(1):73–74, 1989.

[Kra83] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

[Man97] Frank Manola. Object Model Features Matrix. Technical report, NCITS (National Committee for Information Technology Standards) Technical Committee H7 (Object Information Management), May 1997. http://www.objs.com/x3h7/h7home.htm.

[Mar99] Prof. Dr. Peter Marwedel. Begleitmaterial zur Vorlesung Prozessrechnertechnik (Eingebettete Realzeit-Systeme). Skript, Universität Dortmund, Fachbereich Informatik XII, April 1999.

[MD97] Jon Meyer and Troy Downing. *The Java Virtual Machine.* O'Reilly and Associates, 1997.

[Mey89] Bertrand Meyer. You Can Write, But Can You Type? *JOOP*, 1(6):58–67, March/April 1989.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Englewood Cliffs, second edition, 1997.

[MM96] S. Muthukrishnan and Martin Müller. Time and Space Efficient Method-Lookup for Object-oriented Programs. In *7th ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[Mös98] Hanspeter Mössenböck. *Objektorientierte Programmierung in Oberon-2.* Springer Verlag, third edition, 1998.

[NeX93] NeXT Software Inc. *Object-oriented Programming and the Objective-C Language.* Addison-Wesley, 1993.

[Nil96] Kelvin Nilsen. Issues in the Design and Implementation of Real-Time Java. Technical report, Iowa State University, 1996. http://www.newmonics.com.

[Nil98] Kelvin Nilsen. picoPERC: A Small-Footprint Dialect of Java. *Dr. Dobb's Journal of Software Tools*, 23(3):50–54, March 1998.

[NT96] James Noble and Antero Taivalsaari. ECOOP'96 Workshop on Prototype Based Object Oriented Programming. Publiziert im WWW, 1996. http://www.mri.mq.edu.au/~kjx/proto/wkshop96/node19.html.

[Obj98] Object Technology International. ENVY / Embedded Smalltalk. WWW, 1998. http://www.stic.org/IttyBitty/Intro.htm.

[OSN99] Frank Oppenheimer, Guido Schumacher, and Wolfgang Nebel. OO-COSIM – objektorientierte Spezifikation und Simulation eingebetteter Realzeitsysteme. *it + ti – Informationstechnik und Technische Informatik*, (2), 1999.

[PB98] Alexandre Petit-Bianco. Java Garbage Collection For Real-Time Systems. *Dr. Dobb's Journal of Software Tools*, 23(10):20–22, 24, 26–29, October 1998.

[Pla98] P. J. Plauger. Embedded C++: An Overview. WWW, 1998. http://www.embedded.com.

[PS94] J. Palsberg and M. I. Schwartzbach. *Object-oriented Type Systems.* John Wiley and Sons, 1994.

[RBP+93] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Objektorientiertes Modellieren und Entwerfen.* Carl Hanser Verlag und Prentice Hall, 1993.

[Ros88] John R. Rose. Fast Dispatch Mechanisms for Stock Hardware. *ACM SIGPLAN Notices OOPSLA '88 Proceedings*, 23(11), 1988.

[Set90] Ravi Sethi. *Programming Languages, Concepts and Constructs.* Addison-Wesley, Reading, Massachusetts, 1989, reprinted 1990.

[SO91] Heinz W. Schmidt and Stephen M. Omohundro. CLOS, Eiffel and Sather: A Comparison. Technical Report TR-91-047, International Computer Science Institute, September 1991.

[SSH+99] Deflef Schmid, Klaus Schneider, Michaela Huhn, George Logothetis, and Viktor Sabelfeld. Formale Verifikation eingebetteter Systeme. *it + ti – Informationstechnik und Technische Informatik*, (2), 1999.

[SU95] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In Walter G. Olthoff, editor, *ECOOP'95*, Lecture Notes in Computer Science 952, pages 303–330. Springer-Verlag, 1995.

[Szy98] Clemens A. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Tai93] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming.* PhD thesis, University of Jyväskylä, Finland, 1993.

[Tai96] Antero Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[TDS+00] Martin Tasker, Jonathan Dixon, Mark Shackman, Tim Richardson, and John Forrest. *Professional Symbian Programming: Mobile Solutions on the EPOC Platform.* Wrox Press, Februar 2000.

[Tho93] Kresten Krab Thorup. Optimizing Method Dispatch using Sparse Arrays. In *FreeSoft'93, Moscow, Russia*, March 1993. http://www.daimi.aau.dk/~krab.

[Tho95] Dave Thomas. Ubiquitous Applications: Embedded Systems to Mainframe. *Communications of the ACM*, 38(10):112–114, October 1995.

[Tho97] Kresten Krab Thorup. Genericity in Java with Virtual Types. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97 – 11th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science 1241, pages 444–471. Springer-Verlag, 1997.

[Tho98] Kresten Krab Thorup. *Saba Zamir (ed.): Handbook of Object Technology*, chapter Objective-C. CRC Press, 1998.

[Tra00] Martin Trapp. *Optimierung objektorientierter Programme*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, 2000.

[Ung84] David Ungar. Generation Scavenging: A non-disruptive High-Performance Storage Reclamation Algorithm. In *ACM SIGPLAN Symposium on Practical Software Development Environments*, April 1984.

[US91] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *OOPSLA 1987*, 227–241.

[VH96] Jan Vitek and R. Nigel Horspool. Compact Dispatch Tables for Dynamically Typed Object Oriented Languages. In *Proceedings of Compiler Construction CC'96*. Springer-Verlag, 1996.

[Vit95] Jan Vitek. Compact Dispatch Tables for Dynamically Typed Programming Languages. Master's thesis, University of Victoria, British Columbia, Canada, 1995.

[WG93] Niklaus Wirth and Jurg Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1993.

[Wir88] Niklaus Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.

[Wir96] Niklaus Wirth. *Grundlagen und Techniken des Compilerbaus*. Addison-Wesley, 1996.

[WM96] R. Wilhelm and D. Maurer. *Übersetzerbau: Theorie, Konstruktion, Generierung*. Springer-Verlag, Berlin, second edition, 1996.