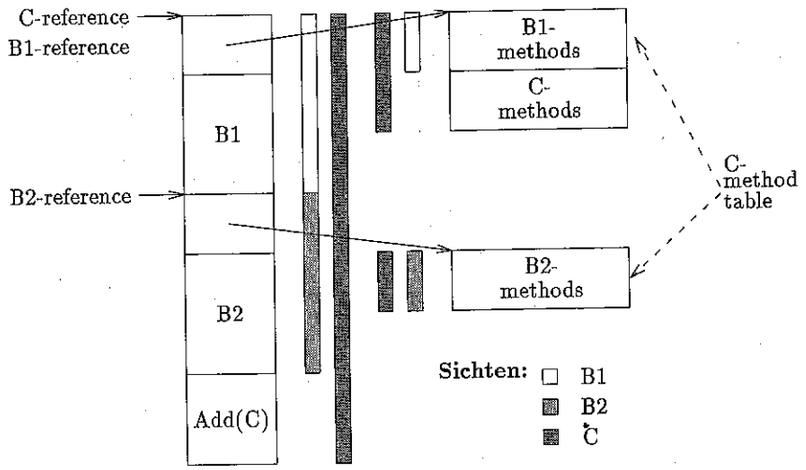

Abb. 5.9: Objektaufbau bei unabhängiger mehrfacher Beerbung (tatsächliche Implementierung)

sche Analyse, beschrieben. Einen saubereren Sprachentwurf vorausgesetzt, ist die Auflösung von Mehrdeutigkeiten kein spezifisches Problem der Übersetzung mehrfacher Vererbung.

Für die Beschreibung benötigen wir einen Satz von Symbolen, der im folgenden eingeführt wird.

N: die Menge der natürlichen Zahlen,

$\mathcal{N}$: eine Menge von Namen(srepräsentationen),

$\mathcal{F}$: eine Menge von speziell kodierten Funktionsnamen (siehe Abschnitt 5.2), die entsprechenden Funktionen implementieren Methoden,

$\mathcal{T}$: eine Menge von Repräsentationen für Typen.

Auf der von uns gewählten Abstraktionsstufe ist eine Klassendefinition für C unter Beerbung von B1 und B2 gegeben durch:

B1, B2: die Basisklassen,

$\mathtt{DM}_1 \subset \mathtt{M}_{\mathtt{B1}} \times \mathcal{F}$: die überschriebenen B1-Methoden zusammen mit den neuen die Methoden implementierenden Funktionen,

$\mathtt{DM}_2 \subset \mathtt{M}_{\mathtt{B2}} \times \mathcal{F}$: die überschriebenen B2-Methoden zusammen mit den neuen die Methoden implementierenden Funktionen,

$\mathtt{DM} \subset \mathcal{N}$: die Liste der von C erstmals eingeführten Methoden,

$\mathtt{DMF} : \mathtt{DM} \to \mathcal{F}$: Diese Funktion ordnet den neu eingeführten Methoden die sie implementierenden Funktionen zu,

$\mathtt{DA} \subset \mathcal{N}$: die Liste der von C erstmals eingeführten Attribute,

$\mathtt{DAT} : \mathtt{DA} \to \mathcal{T}$: Diese Funktion ordnet den neu eingeführten Attributen ihren Typ

zu; aus ihm kann u.a. die Größe des von ihnen belegten Speicherbereichs bestimmt werden.

Auf der Basis dieser Information bestimmt der Übersetzer folgende Größen für C – entsprechende Größen liegen schon für B1 und B2 vor:

$\mathtt{OK_C}$: die Objektkomponenten von C,
dies sind die Komponenten, die zur Laufzeit in C-Objekten abgelegt werden; neben den Attributen gehören dazu auch die Verweise auf die Methodentabellen, jedoch nicht die Methoden selbst.

$\mathtt{M_C}$: die Methoden von C, einschließlich ererbter Methoden,

$\mathtt{O_C}$: die Oberklassen von C,

$\mathtt{MT_C}$: die Methodentabelle von C,
repräsentiert als eine Folge von Methodentabellenabschnitten $\mathtt{MT_C^1} \ldots \mathtt{MT_C^{n_C}}$, jeder Einzelabschnitt ist dabei eine Tabelle, indiziert mit kleinen Zahlen, mit Paaren bestehend aus Funktionsnamen und Offsets als Komponenten: $\mathtt{MT_C^i} : [0 \ldots n_C^i] \longrightarrow \mathcal{F} \times \mathbf{N}$,

$\mathtt{BK_C} : \mathtt{OK_C} \to \mathbf{N}$: die Bindungsfunktion für Objektkomponenten,
sie ordnet Objektkomponenten von C ihren Offset innerhalb von C-Objekten zu,

$\mathtt{BM_C} : \mathtt{M_C} \to \mathbf{N} \times \mathbf{N}$: die Bindungsfunktion für Methoden,
sie ordnet Methoden m von C Paare (i, j) zu – Bedeutung: m ist im i-ten Methodentabellenabschnitt unter Index j beschrieben,

$\mathtt{BMT_C} : [1 \ldots n_C] \to \mathbf{N}$: die Bindungsfunktion für Methodentabellenabschnitte,
sie ordnet den Methodentabellenabschnitten von C den Offset innerhalb von C-Objekten zu, an dem der Zeiger auf den Methodentabellenabschnitt abgelegt wird,

$\mathtt{BO_C} : \mathtt{O_C} \to [1 \ldots n_C]$: die Bindungsfunktion für Oberklassen,
sie ordnet Oberklassen A von C den Methodentabellenabschnitt zu, in dem die Methodentabelle von A beginnt,

$\mathtt{Size_C}$: die Größe von C-Objekten.

Diese Größen werden folgendermaßen bestimmt:

Objektkomponenten:

$\mathtt{OK_C} := \{\mathtt{C}\} \cup (\mathtt{OK_{B1}} - \{\mathtt{B1}\}) \cup \mathtt{OK_{B2}} \cup \mathtt{DA}$

Die Objektkomponenten von C umfassen also:

- C als Komponente für den Verweis auf die C-Methodentabelle,
- die Objektkomponenten von B1, jedoch ohne B1 selbst, da der Verweis auf die B1-Methodentabelle mit dem auf die C-Methodentabelle zusammenfällt,
- die Objektkomponenten von B2,
- die von C neu eingeführten Attribute.

Methoden:

$\mathtt{M_C} := \mathtt{M_{B1}} \cup \mathtt{M_{B2}} \cup \mathtt{DM}$

Oberklassen:
$O_C := \{C\} \cup O_{B1} \cup O_{B2}$
Die Oberklassen von C sind C selbst sowie die Oberklassen von B1 und B2.[18]

Methodentabelle:
$MT_C := MT^1_C, MT'^2_{B1}, \ldots, MT'^{n_{B_1}}_{B1}, MT'^1_{B2}, \ldots, MT'^{n_{B_2}}_{B2}$
mit:
$MT^1_C := MT'^1_{B1} \cdot DMF(DM) \times \{0\}$
$MT'^i_{B1}(j) := MT^i_{B1}(j)$, falls es kein (m,f) in DM_1 gibt mit $BM_{B1}(m) = (i,j)$
$MT'^i_{B1}(j) := (f, BMT_C(i))$, sonst
$MT'^i_{B2}(j) := MT^i_{B2}(j)$, falls es kein (m,f) in DM_2 gibt mit $BM_{B2}(m) = (i,j)$
$MT'^i_{B2}(j) := (f, BMT_C(i + n_{B1}))$, sonst

Die Methodentabelle von C ergibt sich also im wesentlichen als Folge modifizierter Kopien MT'^i_{Bx} der Methodentabellenabschnitte MT^i_{Bx} von B1 und B2; MT'^1_{B1}, der erste so entstandene Abschnitt, wird am Ende um Einträge für die von C erstmals eingeführten Methoden erweitert. Diese neuen Methoden benutzen den Offset 0. Ein Eintrag $MT^i_{Bx}(j)$ wird unverändert nach $MT'^i_{Bx}(j)$ übernommen, wenn C für die zugehörige Methode m keine Redefinition (m,f) enthält; ansonsten wird der Eintrag (in der Kopie) durch (f, o) überschrieben, wobei o der Offset des Zeigers auf den entsprechenden Methodentabellenabschnitt innerhalb von C-Objekten ist: Das ist $BMT_C(i)$ für Bx = B1 und $BMT_C(i + n_{B1})$ für Bx = B2. Dieser Offset muß bei der Methodenaktivierung von der zu diesem Zeitpunkt verfügbaren Sicht, nämlich der zu dem Methodentabellenabschnitt gehörendenden Sicht, abgezogen werden, um die von der Methode benötigte C-Sicht zu erhalten.

Bindung von Objektkomponenten an ihren Offset in C-Objekten:
$BK_C(C) := 0$
für $k \in OK_{B1}$: $BK_C(k) := BK_{B1}(k)$
für $k \in OK_{B2}$: $BK_C(k) := BK_{B2}(k) + Size_{B1}$
für $k \in DA$: $BK_C(k) := o(k) + Size_{B1} + Size_{B2}$
$o(k)$ ist dabei der Offset von k innerhalb des für die neuen Attribute von C bereitgestellten Speicherbereichs; er wird in der Art bestimmt, wie wir dies bei der Ablage von Verbunden im Kapitel über die Übersetzung imperativer Sprachen gesehen haben.

Bindung von Methoden an ihre Position in der Methodentabelle:
für $m \in M_{B1}$: $BM_C(m) := BM_{B1}(m)$
für $m \in M_{B2}$: $BM_C(m) := (i + n_{B1}, j)$ mit $(i,j) = BM_{B2}(m)$
für $m \in DM$: $BM_C(m) := (1, j + |MT^1_{B1}|)$;
dabei gibt j die Position von m innerhalb von DM und $|MT^1_{B1}|$ die Anzahl der Einträge in diesem Abschnitt an.

Bindung von Oberklassen an Methodentabellenabschnitte:
$BO_C(C) := 1$

[18] Sie enthalten ihrerseits B1 und B2.

$BO_C(A) := BO_{B1}(A)$ für $A \in O_{B1}$,
$BO_C(A) := BO_{B2}(A) + n_{B1}$ für $A \in O_{B2}$.

Bindung von Methodentabellenabschnitten an Offsets:
für $1 \le i \le n_{B1}$: $BMT_C(i) := BMT_{B1}(i)$,
für $i > n_{B1}$: $BMT_C(i) := BMT_{B2}(i - n_{B1}) + Size_{B1}$

Größe von C-Objekten:
$Size_C := Size_{B1} + Size_{B2} + Size(C)$;
dabei bezeichnet Size(C) die Größe des für die Ablage der in C neu eingeführten Attribute benötigten Speicherbereichs.

Die verschiedenen Bindungsfunktionen und die Methodentabellenabschnitte sind die wichtigsten der oben bestimmten Größen. Die übrigen sind Hilfsgrößen, um diese bestimmen zu können.

Wir schauen uns nun an, wie der Übersetzer diese Information nutzt.

Methodentabellen: Die Methodentabellenabschnitte werden im erzeugten Zielprogramm einmalig als Daten abgelegt.

Objektinitialisierung: Für die Initialisierung eines C-Objektes wird BMT_C benutzt: Es wird Code erzeugt, um am Offset $BMT_C(i)$ einen Zeiger auf den Methodentabellenabschnitt MT^i_C abzulegen.

Methodenaufruf: Für die Erzeugung eines Methodenaufrufs c.m werden $BM_C(m)$ und BMT_C benutzt: Ist $BM_C(m) = (i,j)$, dann ist m im j-ten Eintrag des i-ten Methodentabellenabschnitts beschrieben. Der Zeiger auf diesen Methodentabellenabschnitt liegt am Offset $oi := BMT_C(i)$ im zugehörigen Objekt, gegeben durch seine C-Referenz c, also an der Adresse mtv := c+oi. mtv ist gleichzeitig die Objektsicht auf c, auf die sich die Offsetangaben innerhalb des Methodentabellenabschnitts als Ausgangspunkt beziehen. mta := *mtv[19] ist also die Anfangsadresse des Methodentabellenabschnitts. f := mta[j].f ist die Funktion, die die Methode m des Objektes c implementiert, und o := mta[j].o der Offset, mit dessen Hilfe die von f benötigte Objektsicht aus der Sicht mtv des Methodentabellenabschnitts bestimmt werden kann. Der Übersetzer erzeugt demnach folgenden Code für den Methodenaufruf m(args):

```
mtv= c + oi;  // Sicht i.ter Methodentabellenabschnitt
mta= *mtv;          // Anfangsadresse dieses Abschnittes
f=   mta[j].f;      // die 'm' implementierende Funktion
v=   mtv - mta[j].o;// die von 'f' benötigte Objektsicht
(*f)(v,args);       // der Methodenaufruf
```

Übersetzung der Methodenaktivierung m(args)

[19] * ist in den Sprachen C und C++ der Dereferenzierungsoperator, bezeichnet also den Inhalt einer Speicherstelle gegebener Adresse (und gegebenen Typs) und wird hier in dieser Bedeutung verwendet.

Die Größen *oi* und *j* sind dem Übersetzer zum Zeitpunkt der Übersetzung der Methodenaktivierung bekannt: Es sind Konstanten im erzeugten Code. `mtv`, `mta`, `b` und `f` sind temporär benutzte Hilfsvariablen.

Sichten: Um für einen Obertyp `A` von `C` eine `A`-Sicht aus einer `C`-Sicht zu erzeugen, benutzt der Übersetzer $\texttt{BO}_\texttt{C}$ und $\texttt{BMT}_\texttt{C}$. Die `A`-Referenz ergibt sich aus der `C`-Referenz durch Addition von $\texttt{BMT}_\texttt{C}(\texttt{BO}_\texttt{C}(\texttt{A}))$ zur `C`-Referenz.

Wir schließen die Darstellung ab mit einer kurzen Betrachtung des durch das Schema erzeugten Overheads:

Laufzeitaufwand für Methodenaktivierung: Die sehr wichtige Methodenaktivierung verursacht einen Laufzeitoverhead von einer Addition, einer Dereferenzierung, zwei indizierten Zugriffen und einer Subtraktion. Dies sind für viele Zielmaschinen sechs Maschineninstruktionen.

Speicheroverhead pro Objekt: Jedes Objekt enthält als Overhead die Zeiger auf die Methodentabellenabschnitte. Nennen wir **Urklasse** eine Klasse, die nicht durch Beerbung aus einer anderen Klasse hervorgegangen ist, also keinen (echten) Vorfahren in der Vererbungshierachie hat, dann ist die Anzahl der Urklasseninstanzen, die in der Definition von `C` vorkommen, genau die Anzahl der `C`-Methodentabellenabschnitte. Diese Anzahl wächst höchstens linear mit der Komplexität der Definition von `C` und wird darüber hinaus in der Praxis fast immer klein sein, verglichen mit der Gesamtgröße des Objektes.

Speicheroverhead pro Klasse: Pro Klasse muß einmal die Methodentabelle abgelegt werden. Die Größe dieser Tabelle wächst linear mit der Größe der Klassendefinition.

Laufzeitoverhead für Objektinitialisierung: Bei der Neuinitialisierung bzw. Erzeugung von Objekten müssen die Zeiger auf die Methodentabellenabschnitte initialisiert werden. Der Overhead ist linear im Speicheroverhead pro Objekt. In vielen praktischen Fällen wird er nicht wesentlich ins Gewicht fallen. Trotzdem wird man sich wünschen, daß die Sprache sparsam mit automatisch angelegten und initialisierten temporären Objektinstanzen umgeht und der Programmierer explizit die Initialisierung seiner Objekte kontrollieren kann. Für `C++` ist das derzeit nicht vollständig der Fall.

Übersetzung abhängiger mehrfacher Beerbung

Bei diesem Verständnis mehrfacher Beerbung wird zugelassen, daß es Abhängigkeiten gibt zwischen dem Erbe von `B1` und dem von `B2`. Beschränkt sind diese Abhängigkeiten auf gemeinsames Erbe von `B1` und `B2`, das als wiederholtes Erbe an `C` übergeben wird. Der wichtigste Spezialfall ist gegeben, wenn das wiederholte Erbe in `C` nur einmal instantiiert wird und sich die `B1`- und `B2`-Teilobjekte diese Instanz teilen, wie in Abb. 5.6 (s.S. 191) veranschaulicht.

Viele der Konflikte und Widersprüche, die bei der unabhängigen Beerbung auftreten, etwa die Mehrdeutigkeiten beim Zugriff auf das wiederholte Erbe oder das Auftreten von mehreren Instanzen derselben Oberklasse in der abgeleiteten Klasse, treten bei diesem Spezialfall nicht auf: Das wiederholte Erbe ist nur einmal instantiiert. Statt dessen treten andere Mehrdeutigkeiten und Probleme auf:

1. Wird eine Methode `m` des gemeinsames Erbes `A` sowohl von `B1` als auch von `B2` überschrieben, dann ist unklar, welche dieser Methoden in die `A`-Sicht von `C`-Objekten übernommen werden soll. Umbenennen der überschreibenden Methoden ist keine Lösung; die `A`-Sicht kennt nur eine einzige Methode `m`, wir aber haben zwei Konkurrenten, unabhängig von ihrem Namen. Statt dessen müssen wir fordern, daß `C` seinerseits die beiden überschreibenden Methoden in `B1` und `B2` durch dieselbe Methode überschreibt.

2. Ein weiteres Problem tritt mit Klasseninvarianten auf. Eine Klasseninvariante für eine Klasse `C` ist eine Aussage, die für alle `C`-Objekte zutrifft. Nehmen wir beispielsweise an, wir haben eine Klasse `Liste`, die in herkömmlicher Weise eine Liste durch einen Listenkopf realisiert, an dem die verketteten Listenelemente beginnen. Nehmen wir weiter an, wir haben zusätzlich ein Attribut `ElementAnzahl` in `Liste`, dann wird die Bedeutung von `ElementAnzahl` formal durch eine Klasseninvariante beschrieben, die zum Ausdruck bringt, daß in jedem `Listen`-Objekt `ElementAnzahl` die Anzahl der am Listenkopf hängenden Elemente enthält. Klasseninvarianten sind ein wichtiges Hilfsmittel, um die Bedeutung von Klassen und ihrer Attribute formal zu beschreiben. Sie sind nutzbar für Korrektheitsbeweise. Ihre Überprüfung zur Laufzeit ist ein wesentliches Hilfsmittel beim Test und erhöht die Zuverlässigkeit der erzeugten Module. Selbstverständlich können die Invarianten nicht zu jedem Zeitpunkt erfüllt sein; während wir beispielsweise ein Element in eine `Liste` einfügen, muß die Invariante notwendigerweise kurzzeitig verletzt sein. Die Invariante einer Klasse muß aber wieder erfüllt sein, wenn eine von außerhalb der Klasse aufgerufene Methode der Klasse zurückkehrt. Das ist eine geeignete Stelle, ihre Gültigkeit zu prüfen.

   Das Problem mit Klasseninvarianten bei einmaliger Instantiierung gemeinsamen Erbes tritt dadurch auf, daß etwa `B2`-Methoden durch Modifikation der einzigen Instanz des gemeinsamen Erbes Teile von `B1`-Objekten verändern und dadurch unbemerkt ihre `B1`-Invariante verletzen können.

Das Schema, das wir in diesem Abschnitt entwicklen, erlaubt nicht nur, den Spezialfall einer einmaligen Instantiierung des gemeinsamen Erbes zu behandeln. Statt dessen kann für jedes einzelne Merkmal des gemeinsamen Erbes festgelegt werden, ob es einfach oder mehrfach instantiiert wird. Neben den Extremfällen: mehrfache Instantiierung des gesamten wiederholten Erbes wie bei unabhängiger Beerbung auf der einen Seite und einmalige Instantiierung des gesamten wiederholten Erbes auf der anderen Seite können also auch Mischformen behandelt werden. Dies ist die Flexibilität, die `Eiffel` dem Programmierer bietet. In

Eiffel ist die Entscheidung über eine einfache oder mehrfache Instantiierung eines Merkmals des gemeinsamen Erbes (Attribut oder Methode) an Umbenennungen gekoppelt: Wurde das Merkmal auf mindestens einem Erbpfad umbenannt, wird es mehrfach instantiiert, sonst nur einmal.

Selbstverständlich erben wir für alle Mischformen die Probleme beider Extremfälle.[20] Ich bin deshalb nicht überzeugt, daß die Möglichkeit, für jedes einzelne Merkmal über einfache oder mehrfache Instantiierung zu entscheiden, von praktischer Bedeutung ist. Vielmehr erwarte ich, daß eine Entscheidungsmöglichkeit auf der Stufe vollständiger Oberklassen ausreichend wäre; aber das allgemeinere Übersetzungsschema ist nur geringfügig komplexer und kann sogar zu kürzeren Programmlaufzeiten führen. Deshalb stellen wir dieses allgemeinere Übersetzungsschema vor. Abb. 5.10 veranschaulicht einen möglichen Objektaufbau, der von diesem Schema unterstützt wird. Das B2-Teilobjekt eines C-Objektes be-

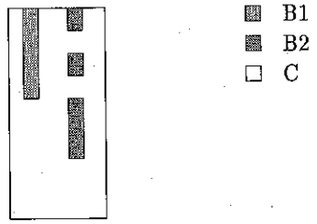

Abb. 5.10: Objektaufbau bei allgemeiner abhängiger mehrfacher Beerbung (Programmsicht)

nutzt einige Abschnitte gemeinsam mit dem B1-Teilobjekt. Wesentlich daran ist, daß das B2-Teilobjekt keinen zusammenhängenden Speicherbereich mehr belegt: Es enthält Lücken. Schlimmer: Die Lücken entstehen erst durch Beerben von B2 durch C, lange nachdem der Übersetzer festlegen mußte, wie eine B2-Sicht aussieht. Weiter: Je nachdem, in welchem Kontext B2 auftaucht, sind die Lücken unterschiedlich groß; so sind sie etwa in den eigentlichen B2-Objekten nicht vorhanden. Als Folge ergibt sich, daß die Offsets von Komponenten innerhalb von Objekten nicht mehr konstant sind und deshalb nicht mehr statisch gebunden werden können.

Ein ähnliches Problem hatten wir auch schon bei der Übersetzung einfacher Vererbung: Der Übersetzer mußte dort Code für den Aufruf von Funktionen erzeugen, die ihm noch nicht bekannt waren; jetzt muß er Code für den Zugriff auf Komponenten erzeugen, deren Offset er nicht kennt. Tatsächlich kann der Übersetzer für dieses (und viele ähnliche Probleme) dieselbe Technik anwenden, die bei der dynamischen Bindung von Funktionen geholfen hat: das Einziehen einer Indirektionsstufe mit Hilfe einer Tabelle. Der Übersetzer bindet die Komponenten einer Klasse an feste Indizes in einer Indextabelle; für jeden verschiedenen

[20] Auch das sehr weit entwickelte Eiffel enthält nicht alle Sprachmittel, um die entstehenden Probleme unter Programmkontrolle vollständig auflösen zu können.

Kontext wird eine eigene Indextabelle angelegt mit den für diesen Kontext gültigen Offsets. Die Objekte enthalten einen Verweis auf die für sie gültige Indextabelle. So realisieren wir die dynamische Bindung von Komponentenoffsets, ganz analog zur dynamischen Bindung von Methoden. Die entstehende Struktur wird in Abb. 5.11 veranschaulicht. Sie zeigt den Aufbau eines C-Objektes mit sei-

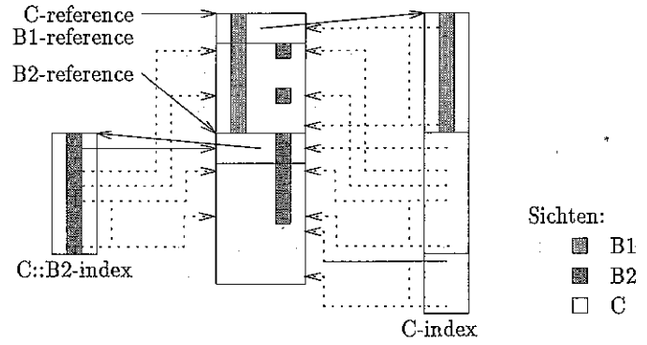

Abb. 5.11: Objektaufbau bei allgemeiner abhängiger mehrfacher Beerbung (Implementierungssicht)

nen beiden Verweisen auf die ein einziges Mal abgelegten Indextabellen C-Index und C::B2-Index. C-Index ordnet jeder Komponente von C ihren Offset relativ zu einer C-Referenz zu; analog ordnet C::B2-Index jeder Komponente des B2-Teilobjektes ihren Offset relativ zu einer B2-Referenz zu. In der Abbildung sind die Offsets in den Indextabellen durch punktierte Pfeile veranschaulicht.

Ebenso wie der erste Ansatz für die Implementierung unabhängiger mehrfacher Beerbung hat auch dieser Ansatz einen entscheidenden Schönheitsfehler: Die beiden Abbildungstabellen für C enthalten zwei Abbilder der Indextabelle von B2 – Folge: potentiell exponentiell wachsender Speicherbedarf in der Größe der Klassendefinition. Um dies zu vermeiden, muß es uns gelingen, eine der beiden Kopien einzusparen. Wie die Abbildung korrekt zeigt, sind beide Kopien fast identisch. Einziger Unterschied: Die Offsets in C::B2-Index beziehen sich auf B2-Referenzen, während die Kopie innerhalb von C-Index Offsets bezogen auf C-Referenzen enthält. Die Differenz beider Referenzen ist für alle eigentlichen C-Objekte konstant und bereits in der C-Indextabelle abgelegt: am Anfang seiner Kopie der B2-Indextabelle. Sofern wir diesen Offset bei der Interpretation einer Indextabelle stets berücksichtigen, können wir die Kopie der B2-Indextabelle innerhalb der C-Indextabelle als Indextabelle für B2-Teilobjekte von C-Objekten mitbenutzen. Die resultierende Objektstruktur ist in Abb. 5.12 veranschaulicht. In die Indextabelle werden wir neben Offsets auch die aus dem vorigen Abschnitt bekannte Methodeninformation integrieren.

Wie im vorigen Abschnitt werden wir unser Schema jetzt halbformal beschreiben. Wiederum machen wir vereinfachende Annahmen: Genau die Merkmale, die

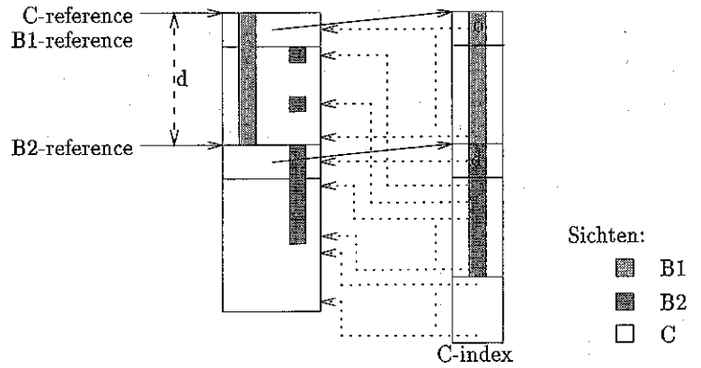

Abb. 5.12: Objektstruktur bei allgemeiner abhängiger mehrfacher Beerbung (tatsächliche Implementierung)

B1 und B2 in einer Instanz gemeinsam benutzen sollen, haben identische Namen, alle übrigen Namen sind paarweise verschieden. Es ist sichergestellt, daß keine der oben beschriebenen potentiellen Mehrdeutigkeiten oder Widersprüche auftreten, also etwa für ein einmal instantiiertes Merkmal zwei widersprüchliche Definitionen vorliegen.

Auf der von uns gewählten Abstraktionsstufe ist eine Klassendefinition für C unter Beerbung von B1 und B2 (genau wie bei unabhängiger Beerbung) gegeben durch:

B1, B2: die Basisklassen

$DM_1 \subset M_{B1} \times \mathcal{F}$: die überschriebenen B1-Methoden zusammen mit den neuen die Methoden implementierenden Funktionen,

$DM_2 \subset M_{B2} \times \mathcal{F}$: die überschriebenen B2-Methoden zusammen mit den neuen die Methoden implementierenden Funktionen,

$DM \subset \mathcal{N}$: die Liste der von C erstmals eingeführten Methoden,

$DMF : DM \to \mathcal{F}$: Diese Abbildung ordnet den neu eingeführten Methoden die sie implementierenden Funktionen zu,

$DA \subset \mathcal{N}$: die Liste der von C erstmals eingeführten Attribute,

$DAT : DA \to \mathcal{T}$: Diese Abbildung ordnet den neu eingeführten Attributen ihren Typ zu; aus ihm kann u.a. die Größe des von ihnen belegten Speicherbereichs bestimmt werden.

Auf der Basis dieser Information bestimmt der Übersetzer folgende Größen für C – entsprechende Gößen liegen schon für B1 und B2 vor:

K_C: die Merkmale von C, d.h. ihre Methoden und Attribute,

O_C: die Oberklassen von C,

OK_C: die Objektkomponenten; das sind die Komponenten, die in jedem C-Objekt

zur Laufzeit abgelegt sind; neben den Attributen gehören dazu auch die Verweise auf die Indexabschnitte, jedoch nicht die Methoden,

IK_C: die Komponenten in C-Indizes: Attribute, Methoden, Oberklassen – mehrfach genannt, wenn sie sowohl in B1 als auch in B2 auftreten,

IMK_C: die Methoden unter den Indexkomponenten,

IOK_C: die Objektkomponenten (Attribute, Oberklassen) unter den Indexkomponenten; IK_C zerfällt damit in die beiden Teilmengen IMK_C und IOK_C,

$IA_C \subset O_C$: die Abschnitte der C-Indextabelle, repräsentiert durch sie benutzende Klassen,

I_C: die Indextabelle von C; eine Tabelle mit Indizes aus $0 \ldots |IK_C| - 1$ und Funktion-Offset-Paaren für Methoden bzw. Offsets für Objektkomponenten als Werten,

$BK_C : K_C \to IK_C$: die Bindungsfunktion für Merkmale von C; sie ordnet jedem Merkmal von C seine Indexkomponente zu,

$BO_C : O_C \to IK_C$: die Bindungsfunktion für Oberklassen; sie ordnet jeder Oberklasse von C ihre Indexkomponente zu,

$BOK_C : OK_C \to \mathbf{N}$: die Bindungsfunktion für Objektkomponenten; sie ordnet den Elementen aus OK_C ihre Offsets innerhalb von eigentlichen C-Objekten[21] zu,

$BIK_C : IK_C \to \mathbf{N}$: die Bindungsfunktion für Indexkomponenten; sie ordnet Indexkomponenten ihre Position innerhalb der Indextabelle zu,

$BIOK_C : IOK_C \to OK_C$: die Bindungsfunktion für Objektkomponenten in C-Indizes; sie ordnet den Objektkomponenten repräsentierenden Indexkomponenten die entsprechenden Objektkomponenten in den C-Laufzeitobjekten zu,

$BIMK_C : IMK_C \to \mathcal{F} \times O_C$: die Bindungsfunktion für Methoden unter den C-Indexkomponenten; sie ordnet den Methodenkomponenten Paare zu, bestehend aus der Funktionsimplementierung der Methode und der die Methode definierenden Klasse.

Diese Größen werden folgendermaßen bestimmt:

Merkmale von C:

$K_C := K_{B1} \cup (K_{B2} - K_{B1}) \cup DA \cup DM$

Oberklassen:

$O_C := \{C\} \cup O_{B1} \cup (O_{B2} - O_{B1})$

Objektkomponenten und ihre Bindung:

$OK_C := \{C\} \cup (OK_{B1} - \{B1\}) \cup (OK_{B2} - OK_{B1}) \cup DA$

die Objektkomponenten von C umfassen also:

- C als Komponente für den Verweis auf die C-Indextabelle,
- die Objektkomponenten von B1, jedoch ohne B1 selbst, da der Verweis auf die B1-Methodentabelle mit dem auf die C-Indextabelle zusammenfällt,
- die Objektkomponenten von B2, soweit sie nicht auch in B1 auftreten,
- die von C neu eingeführten Attribute.

[21] Also C-Objekten, die nicht einer echten Teilklasse angehören.

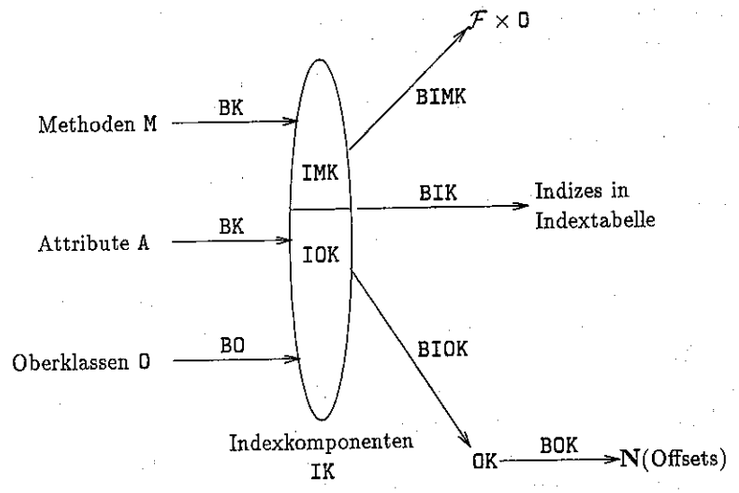

Abb. 5.13: Zusammenhang zwischen den bei der Klassenübersetzung definierten Größen

Zur Bestimmung von $\texttt{BOK}_\texttt{C}$ werden die Elemente von $\texttt{OK}_\texttt{C}$ als Komponenten eines Verbundes abgelegt, wie wir dies im Kapitel über die Übersetzung imperativer Sprachen gesehen haben. Die Klassen in $\texttt{OK}_\texttt{C}$ repräsentieren dabei die Verweise auf die zugehörigen Indextabellenabschnitte. Die Komponente für C muß ab Offset 0 abgelegt werden. $\texttt{BOK}_\texttt{C}(k)$ wird definiert als der Offset der Komponente k innerhalb des Verbundes.
Beachten Sie, daß $\texttt{BOK}_\texttt{C}$ nur für eigentliche C-Objekte gültig ist.

Indexkomponenten und ihre Bindung:

$\texttt{IK}_\texttt{C} := \{\texttt{C}\} \cup \texttt{B1::}(\texttt{IK}_{\texttt{B1}} - \{\texttt{B1}\}) \cup \texttt{B2::}\texttt{IK}_{\texttt{B2}} \cup \{\texttt{B1::B1}\} \cup \texttt{DA} \cup \texttt{DM}$
$\texttt{BIK}_\texttt{C}$ ordnet den Elementen von $\texttt{IK}_\texttt{C}$ fortlaufend Nummern zu – in der in der Definition von $\texttt{IK}_\texttt{C}$ angegebenen Reihenfolge. Die relevanten Zuordnungen sind:

$\texttt{BIK}_\texttt{C}(C) := 0,$
$\texttt{BIK}_\texttt{C}(\texttt{B1::}k) := \texttt{BIK}_{\texttt{B1}}(k)$ für $k \neq \texttt{B1},$
$\texttt{BIK}_\texttt{C}(\texttt{B2::}k) := \texttt{BIK}_{\texttt{B2}}(k) + |\texttt{IK}_{\texttt{B1}}|,$
$\texttt{BIK}_\texttt{C}(\texttt{B1::B1}) := |\texttt{IK}_{\texttt{B1}}| + |\texttt{IK}_{\texttt{B1}}|.$

Die genaue Indexvergabe für die neuen Merkmale von C ist von untergeordneter Bedeutung.

Diese beiden Definitionen legen den Aufbau der C-Indextabelle fest. Die Indextabelle von C muß je eine (modifizierte) Kopie der Indextabelle von B1 und B2 enthalten. Wir würden also erwarten, daß $\texttt{IK}_\texttt{C}$ die Vereinigung von $\texttt{IK}_{\texttt{B1}}$ und $\texttt{IK}_{\texttt{B2}}$ umfaßt. Aber diese beiden Mengen sind nicht notwendig disjunkt. Dies ist der Grund, weshalb wir die beiden Operanden der Vereinigung durch B1:: bzw. B2:: kennzeichnen und ihre Elemente dadurch voneinander unterscheidbar machen.

Beachten Sie ferner, daß wir B1 *nicht* den Index 0 zuordnen, sondern statt dessen einen Index hinter den übrigen Komponenten von $\texttt{IK}_{\texttt{B1}}$ gewählt haben. Begründet ist dies durch folgende Überlegungen: Der erste Eintrag in einem Indextabellenabschnitt *muß* immer die zugehörige Klasse beschreiben; er enthält den Offset des zur Klasse gehörenden Teilobjektes innerhalb des Hauptobjektes. Der erste Eintrag in einer C-Indextabelle muß also C beschreiben. Hätten wir den Elementen von $\texttt{IK}_{\texttt{B1}}$ von 1 beginnend fortlaufende Nummern zugeordnet, hätten wir die Indextabelle von C nicht für das B1-Teilobjekt mitbenutzen können. Dies hätte einen weiteren Verweis in der Laufzeitdarstellung von C-Objekten auf diesen Indexabschnitt erforderlich gemacht. Indem wir B1 ausnehmen und den übrigen Elementen denselben Index zuordnen, den sie auch von $\texttt{BIK}_{\texttt{B1}}$ erhalten, können wir einen Anfangsabschnitt der C-Indextabelle als B1-Indextabelle benutzen. Dies ist so lange möglich, wie das B1-Objekt tatsächlich innerhalb des C-Objektes instantiiert ist. Ist das nicht der Fall – dies kann nur für C-Teilobjekte in Objekten abgeleiteter Klassen auftreten –; dann gibt es an anderer Stelle in der zu dieser abgeleiteten Klasse gehörenden Indextabelle eine weitere Kopie der B1-Indextabelle und in den Objekten dieser Klasse einen Verweis auf diese Indextabelle; unser B1-Eintrag in der C-Indextabelle wird stets so besetzt, daß wir über ihn den gültigen Verweis auf die zu benutzende B1-Indextabelle finden. Deshalb darf die B1-Indexkomponente nicht wie bei der B1-Objektkomponente mit der entsprechenden C-Komponente zusammengefaßt werden.

Indexabschnitte:

$\texttt{IA}_\texttt{C} := \texttt{C} \cup (\texttt{IA}_{\texttt{B1}} - \{\texttt{B1}\}) \cup (\texttt{IA}_{\texttt{B2}} - \texttt{IA}_{\texttt{B1}})$

Bindung der Merkmale von C an ihre Indexkomponenten:

$\texttt{BK}_\texttt{C}(\texttt{k}) := \texttt{B1::k}$ für $\texttt{k} \in \texttt{K}_{\texttt{B1}}$
$\texttt{BK}_\texttt{C}(\texttt{k}) := \texttt{B2::k}$ für $\texttt{k} \in \texttt{K}_{\texttt{B2}} - \texttt{K}_{\texttt{B1}}$
$\texttt{BK}_\texttt{C}(\texttt{k}) := \texttt{k}$ sonst

Bindung der Oberklassen an ihre Indexkomponenten:

$\texttt{BO}_\texttt{C}(\texttt{C}) := \texttt{C}$
$\texttt{BO}_\texttt{C}(\texttt{A}) := \texttt{B1::A}$ für $\texttt{A} \in \texttt{O}_{\texttt{B1}}$
$\texttt{BO}_\texttt{C}(\texttt{A}) := \texttt{B2::A}$ für $\texttt{A} \in \texttt{O}_{\texttt{B2}} - \texttt{O}_{\texttt{B1}}$

Bindung der Index-Objektkomponenten an Objektkomponenten:

$\texttt{BIOK}_\texttt{C}(\texttt{k}) := \texttt{BIOK}'_\texttt{C}(\texttt{k})$, falls $\texttt{BIOK}'_\texttt{C}(\texttt{k}) \neq \texttt{B1}$
$\texttt{BIOK}_\texttt{C}(\texttt{k}) := \texttt{C}$, sonst

$\texttt{BIOK}'_\texttt{C}(\texttt{C}) := \texttt{C}$
$\texttt{BIOK}'_\texttt{C}(\texttt{B1::k}) := \texttt{BIOK}_{\texttt{B1}}(\texttt{k})$
$\texttt{BIOK}'_\texttt{C}(\texttt{B2::k}) := \texttt{BIOK}_{\texttt{B1}}(\texttt{k})$, falls $\texttt{k} \in \texttt{IOK}_{\texttt{B1}}$

$\texttt{BIOK}'_{\texttt{C}}(\texttt{B2::k}) := \texttt{BIOK}_{\texttt{B2}}(\texttt{k})$, falls $\texttt{k} \notin \texttt{IOK}_{\texttt{B1}}$
$\texttt{BIOK}'_{\texttt{C}}(\texttt{k}) := \texttt{k}$, sonst

In diese Definition geht folgende Überlegung ein: Sofern k zu B1 gehört, soll auch für B2::k die zu B1 gehörende Bindungsfunktion $\texttt{BIOK}_{\texttt{B1}}$ und nicht die zu B2 gehörende Bindungsfunktion $\texttt{BIOK}_{\texttt{B2}}$ benutzt werden. Nur falls k zu B2, aber nicht zu B1 gehört, wird $\texttt{BIOK}_{\texttt{B2}}$ zur Bindung von B2::k benutzt. Dies hat Auswirkungen vor allem auf gemeinsame Oberklassen von B1 und B2. Durch diesen Mechanismus wird es möglich, daß eine Oberklasse A von B2, die bisher den Verweis auf ihren Indextabellenabschnitt etwa mit B2 teilte, nun ihren Verweis in einer B1-Komponente besitzt.
Die Hilfsfunktion $\texttt{BIOK}'_{\texttt{C}}$ setzt dies um. Allerdings kann sie Komponenten auf B1 abbilden, das in den C-Objektkomponenten durch C ersetzt wurde; die gegebene Definition für $\texttt{BIOK}_{\texttt{C}}$ korrigiert dies wieder.

Bindung von Methodenkomponenten an Funktion-Klasse-Paare:
$\texttt{BIMK}_{\texttt{C}}(\texttt{B1::m}) := \texttt{BIMK}_{\texttt{B1}}(\texttt{m})$, falls es kein $(\texttt{m},\texttt{f})$ in $\texttt{DM}_1$ gibt
$\texttt{BIMK}_{\texttt{C}}(\texttt{B1::m}) := (\texttt{f},\texttt{C})$, sonst
$\texttt{BIMK}_{\texttt{C}}(\texttt{B2::m}) := \texttt{BIMK}_{\texttt{B2}}(\texttt{m})$, falls es kein $(\texttt{m},\texttt{f})$ in $\texttt{DM}_2$ gibt
$\texttt{BIMK}_{\texttt{C}}(\texttt{B2::m}) := (\texttt{f},\texttt{C})$, sonst
$\texttt{BIMK}_{\texttt{C}}(\texttt{m}) := (\texttt{f},\texttt{C})$ für $(\texttt{m},\texttt{f})$ in DM.

Indextabelle:
Jede Komponente j von $\texttt{I}_{\texttt{C}}$ entspricht unter $\texttt{BIK}_{\texttt{C}}$ einer Komponenten k aus $\texttt{IK}_{\texttt{C}}$.
Ist k eine Objektkomponente, dann ist

$$\texttt{I}_{\texttt{C}}(j) := \texttt{BOK}_{\texttt{C}}(\texttt{BIOK}_{\texttt{C}}(k))$$

der Offset von k innerhalb von (eigentlichen) C-Objekten.
Ist k eine Methode, dann wird mit $(\texttt{f},\texttt{A}) := \texttt{BIMK}_{\texttt{C}}(k)$ die k implementierende Funtkion und der Offset der sie definierenden Klasse gegeben durch

$$\texttt{I}_{\texttt{C}}(j) := (\texttt{f}, \texttt{BOK}_{\texttt{C}}(\texttt{BIOK}_{\texttt{C}}(\texttt{BO}_{\texttt{C}}(\texttt{A})))).$$

Die verschiedenen Bindungsfunktionen, die Menge der Indexabschnitte und die Indextabelle sind die wichtigsten der oben bestimmten Größen. Die übrigen sind Hilfsgrößen, um diese bestimmen zu können.
Wir schauen uns nun an, wie der Übersetzer diese Information nutzt.

Indextabelle: Die Indextabelle wird im erzeugten Zielprogramm einmalig als Daten abgelegt.

Objektinitialisierung: Zur Initialisierung eines eigentlichen C-Objektes wird Code erzeugt, um für jedes Element A von $\texttt{IA}_{\texttt{C}}$ am Offset $\texttt{BOK}_{\texttt{C}}(\texttt{BIOK}_{\texttt{C}}(\texttt{BO}_{\texttt{C}}(\texttt{A})))$ des zu initialisierenden Objektes einen Verweis auf den $\texttt{BIK}_{\texttt{C}}(\texttt{BO}_{\texttt{C}}(\texttt{A}))$-ten Eintrag in der Indextabelle abzulegen.

Methodenaufruf: Ist m eine Methode zu einem C-Objekt, gegeben durch seine C-Referenz c, und $i := \texttt{BIK}_{\texttt{C}}(\texttt{BK}_{\texttt{C}}(\texttt{m}))$, dann verweist der Zeiger an der Adresse

c auf eine Indextabelle mit dem Aufbau einer C-Indextabelle. Der erste Eintrag dieser Indextabelle enthält den Offset co des betrachteten C-Teilobjektes innerhalb seines Hauptobjektes o:= c - co. Der i-te Eintrag der Indextabelle enthält ein Paar (f,fo), das die m implementierende Funktion und die von ihr erwartete Objektsicht beschreibt. Diese Sicht ergibt sich aus o durch Addition von fo. Zur Aktivierung von c.m(args) erzeugt der Übersetzer also folgenden Code:

```
it=  *c;              // Anfang der Indextabelle
o=   c - *it;         // Referenz auf das Hauptobjekt
f=   it[i].f;         // die 'm' implementierende Funktion
v=   o + it[i].fo;    // die von 'f' benötigte Objektsicht
(*f)(v,args);         // der Methodenaufruf
```

Übersetzung der Methodenaktivierung c.m(args)

i ist dem Übersetzer bekannt, es ist eine Konstante im erzeugten Code. it, o, f und v sind temporär benutzte Hilfsvariablen.

Attributzugriff: Ist a ein Attribut eines C-Objektes, gegeben durch seine C-Referenz c, und $i := \texttt{BIK}_{\texttt{C}}(\texttt{BK}_{\texttt{C}}(\texttt{a}))$, dann verweist der Zeiger an der Adresse c auf eine Indextabelle mit dem Aufbau einer C-Indextabelle. Der erste Eintrag dieser Indextabelle enthält den Offset co des betrachteten C-Teilobjektes innerhalb seines Hauptobjektes o:= c - co. Der i-te Eintrag der Indextabelle enthält den Offset ao von a innerhalb von o. Zum Zugriff auf c.a erzeugt der Übersetzer also folgenden Code:

```
it=  *c;              // Anfang der Indextabelle
o=   c - *it;         // Referenz auf das Hauptobjekt
ao=  it[i];           // Offset von 'a' in 'o'
...o[ao]...;          // Zugriff
```

Übersetzung des Attributzugriffs c.a

i ist dem Übersetzer bekannt, es ist eine Konstante im erzeugten Code. it, o und ao sind temporär benutzte Hilfsvariablen.

Sichten: Um für einen Obertyp A von C eine A-Sicht auf ein C-Objekt, gegeben durch seine C-Referenz c, zu erhalten, erzeugt der Übersetzer folgenden Code:

```
it=  *c;              // Anfang der Indextabelle
o=   c - *it;         // Referenz auf das Hauptobjekt
ao=  it[i];           // Offset des 'A'-Teilobjektes in 'o'
a=   o + ao;          // 'A'-Sicht auf 'c'
```

Berechnen einer A-Sicht auf c

i := $\texttt{BIK}_C(\texttt{BO}_C(\texttt{A}))$ ist dem Übersetzer bekannt, es ist eine Konstante im erzeugten Code. `it`, `o`, `ao` und `a` sind temporär benutzte Hilfsvariablen.

Wir schließen die Darstellung ab mit einer kurzen Betrachtung des durch das Schema erzeugten Overheads:

Laufzeitoverhead für Attributzugriff: Der wichtige Attributzugriff verursacht einen Laufzeitoverhead von zwei Dereferenzierungen, einer Zuweisung und einer Substraktion zur Ermittlung von `it` und `o` – das sind 4 Instruktionen bei vielen Zielmaschinen. Darüber hinaus fällt zusätzlicher Overhead in Form eines indizierten Zugriffs mit variablem Index an – das sind weitere ein oder zwei Instruktionen. Attributzugriffe erfolgen meist nur auf das aktuelle Objekt.[22] Für diesen Fall kann man erwarten, daß ein optimierender Übersetzer die zugehörigen häufig benutzten Größen `it` und `o` bereits in Registern hält. Im Normalfall rechnen wir daher nur mit einem Overhead von einem indizierten Zugriff,[23] also mit ein bis zwei Maschinenbefehlen, pro Attributzugriff.

Laufzeitoverhead für Methodenaktivierung: Die Methodenaktivierung verursacht einen Laufzeitoverhead von einer Addition, zwei Dereferenzierungen, zwei indizierten Zugriffen und einer Subtraktion. Dies sind für viele Zielmaschinen sieben Maschineninstruktionen.

Speicheroverhead pro Objekt: Jedes Objekt enthält als Overhead die Zeiger auf die Indexabschnitte. Die Anzahl dieser Zeiger entspricht wie bei unabhängiger Beerbung der Anzahl der Urklasseninstanzen, die in die Definition von `C` eingehen. Diese Anzahl wächst höchstens linear mit der Komplexität der Definition von `C` und wird typischerweise vernachlässigbar sein.

Speicheroverhead pro Klasse: Pro Klasse muß einmal die Indextabelle abgelegt werden. Die Größe dieser Tabelle wächst linear mit der Größe der Klassendefinition.

Laufzeitoverhead für Objektinitialisierung: Bei der Neuinitialisierung bzw. Erzeugung von Objekten müssen die Zeiger auf die Indexabschnitte initialisiert werden. Der Overhead ist linear im Speicheroverhead pro Objekt.

5.4 Generizität

Streng typisierte Sprachen mit Typprüfung durch einen Übersetzer haben gegenüber Sprachen mit dynamischer Typprüfung sehr große Vorteile hinsichtlich Sicherheit und Effizienz. Denn auf Basis von Typinformation kann der Übersetzer viele potentielle Inkonsistenzen zu einem Zeitpunkt erkennen, zu dem sie noch keine schwerwiegenden Auswirkungen auf unsere menschliche Welt haben.

[22] Einige Sprachen erlauben sogar ausschließlich solche Attributzugriffe.
[23] mit variablem Index

Ferner kann der Übersetzer das durch Typen ausgedrückte Wissens über den Aufbau der Laufzeitobjekte ausnutzen, um effiziente Zielprogramme zu erzeugen. So haben wir etwa im vorherigen Abschnitt gesehen, wie der Übersetzer auf der Basis des Wissen über die Objekttypen und damit ihren Aufbau sehr effizient auf die Methoden und Attribute eines Objektes zugreifen kann – mit einem Overhead von 5-7 Maschineninstruktionen. Ohne Typinformation müßte eine Methode dynamisch gesucht werden – mit einem Overhead von Hunderten von Instruktionen.

Andererseits legen streng typisierte Sprachen ihren Programmen Fesseln an. Diese Fesseln zwingen häufig dazu, denselben Algorithmus in vielen Funktionsinstanzen zu realisieren. Eines der einfachsten Beispiel ist der Vertauschungsoperator, der seine beiden Argumente miteinander vertauscht. In einer typlosen Sprache kann er etwa folgendermaßen implementiert werden:

```
exchange(inout x, inout y)
  local t;
  t:= x; x:= y; y:= t
```

Implementierung von `exchange`

Er vertauscht für beliebiges `x` und `y` ihre Werte. In einer streng typisierten Sprache muß der Vertauschungsoperator für jeden Typ durch eine eigene Funktion realisiert werden, z.B. für ganze Zahlen

```
exchange_integer(inout integer x, inout integer y)
  local integer t;
  t:= x; x:= y; y:= t
```

Implementierung von `exchange_integer`

und für Fließpunktzahlen

```
exchange_real(inout real x, inout real y)
  local real t;
  t:= x; x:= y; y:= t
```

Implementierung von `exchange_real`

Die auszuführenden Operationen sind gleich;[24] trotzdem verlangen streng typisierte Sprachen zwei verschiedene Funktionsdefinitionen.

Auf den ersten Blick mag es scheinen, als könne die früher eingeführte Vererbung das Problem lösen: Wir definieren eine Klasse `object`, die Oberklasse einer jeden anderen Klasse ist. Diese Klasse definiert allgemeine für alle Objekte verfügbare Operationen: Zuweisung, Test auf Identität. Die Funktion `exchange` kann damit bereits über `object` definiert und von allen Teilklassen geerbt werden.

[24] zumindest auf der Abstraktionsstufe der Algorithmusbeschreibung

```
exchange_object(inout object x, inout object y)
  local object t;
  t:= x; x:= y; y:= t
```

Implementierung von exchange_object

Leider ist diese Funktion völlig unbrauchbar. Dies liegt an einer Eigenschaft des mit Vererbung zusammenhängenden Typsystems, die wir bisher nur angedeutet haben: Es *gibt* Stellen, an denen ein Objekt *nicht* durch ein Objekt einer Teilklasse ersetzt werden darf. Wichtigstes Beispiel ist das Ziel einer Zuweisung. Hierfür gilt genau die umgekehrte Regel:

Zuweisungstypregel: Eine Zuweisung `b:=...` bleibt typkorrekt, wenn die Variable vom Typ `B` durch eine Variable von einem *Ober*typ `A` ersetzt wird.
Eine Zuweisung `...:=a` bleibt typkorrekt, wenn das Objekt `a` vom Typ `A` durch ein Objekt `b` von einem *Teil*typ `B` ersetzt wird.

Betrachten wir kurz die Begründung für diese Regel: Eine Zuweisung an eine Variable vom Typ `B` ist zulässig, wenn sichergestellt ist, daß das zugewiesene Objekt tatsächlich ein `B`-Objekt ist, das heißt alle von `B` geforderten Merkmale besitzt. Ersetzen wir das Zuweisungsziel durch eine Variable eines Obertyps, schwächen wir die Anforderungen ab – Oberklassen fordern weniger Merkmale – die Zuweisung bleibt zulässig. Ein Übergang zu Variablen eines Teiltyps würde demgegenüber die Anforderungen verschärfen und kann die Zuweisung unzulässig machen. Deshalb wird dieser Übergang vom Typsystem verboten. Aus der Zuweisungstypregel leiten wir unmittelbar Regeln für die verschiedenen Parameterarten ab.

Typverträglichkeit zwischen aktuellen und formalen Parametern: An einen Eingabeparameter mit deklariertem Typ `A` können nur Objekte eines Teiltyps von `A` übergeben werden. An einen Ausgabeparameter mit deklariertem Typ `A` können nur Variablen eines Obertyps von `A` übergeben werden. An einen Ein/Ausgabeparameter mit deklariertem Typ `A` können nur Variablen mit deklarierten Typ `A` übergeben werden.

Diese Regeln ergeben sich aus der Zuweisungsregel, da die Parameterübergabe an einen Eingabeparameter als eine Zuweisung des aktuellen Parameters an den formalen Parameter beim Funktionseintritt gesehen werden kann, die Parameterübergabe an einen Ausgabeparameter dagegen als Zuweisung des formalen Parameters an den aktuellen Parameter beim Verlassen der Funktion. Daher ergibt sich, daß der oben definierte Operator nur auf Objekte mit deklariertem Typ `object` angewandt werden darf – wir haben nichts gewonnen.[25]

[25]Das Typsystem von `Eiffel` unterstützt ein Konzept, mit dem sich Abhängigkeiten zwischen den Typen unterschiedlicher Variablen ausdrücken lassen. Solche Abhängigkeiten in der Form „`y` hat denselben Typ wie `x`" und zugehörige Typverträglichkeitsregeln erlauben in gewissen Fällen, nachzuweisen, daß eine Zuweisung auch beim Übergang zu einem Teiltyp für die Zielvariable gültig bleibt (weil sich nachweislich der Typ des zugewiesenen Objektes in gleicher Weise verschärft). Hätte `Eiffel` Ein/Ausgabeparameter, wäre der Vertauschungsoperator für alle Nachfahren von `object` mit einer einzigen Funktion implementierbar. Die Möglichkeit, Typen durch Bezugnahme auf die Typen anderer Objekte festzulegen und dadurch Abhängigkeiten zwischen den Typen verschiedener Objekte auszudrücken, bringt dem an sich schon mächtigen Vererbungsmechanismus noch zusätzliche Flexibilität.

Wäre das Problem auf so einfache Fälle wie den Vertauschungsoperator beschränkt, würde niemand sich um eine Lösung Gedanken machen. Aber der Vertauschungsoperator ist nur ein Beispiel aus einer riesigen Klasse von Algorithmen, zu denen unter anderem meistgebrauchte Basisalgorithmen gehören: etwa die Manipulation von Listen, Mengen, Warteschlangen, Inhaltsverzeichnissen, Bäumen. Betroffen sind also unter anderem Datentypen, die man in einem weiten Sinn als „*Behälter*" bezeichnen kann. Solche Behälter haben in typisierten Sprachen einen natürlichen Parameter: den Typ ihres Inhalts. **Generizität** erlaubt die Definition in dieser Art parametrierter Typen.

Ein **generischer Datentyp** bzw. eine **generische Klasse** ist ein Prototyp mit formalen Parametern, der bei Versorgung mit geeigneten aktuellen Parametern einen Datentyp bzw. eine Klasse definiert. Die Anwendung einer generischen Klasse auf aktuelle Argumente wird meist **Instantiierung** genannt: Dadurch entsteht eine Klasseninstanz. Die Abstraktion durch generische Klassen ist im Bereich der Typen analog zur Abstraktion durch Prozeduren im Bereich von Berechnungen zu sehen. Der Instantiierung auf der einen Seite entspricht die Prozeduraktivierung auf der anderen Seite.

Wir schauen uns beispielhaft einen Ausschnitt aus der Definition einer generischen Listenklasse in `C++` an:

```
// Listlinks -- the links to form lists
template <class T>
struct Listlink {
    T             value;   // value of this element
    Listlink      *next;   // link to next element
  // Constructor
    Listlink(T val, Listlink *link) { value= val; next= link; }
};

// Lists  -- implemented as head to linked listlinks
template <class T>
class List {
    Listlink<T> *head;    // list head

  public:
    // Constructor
      List(void) { head= 0; }
    // check for emptyness
      int empty(void) { return head == 0; };
    // get an element (returns the first one)
      Listlink<T> *elem(void) { return head; };
```

```
    // push a value
      void push(T value) {
        Listlink<T>     *elem= new Listlink<T>(value,head);
        head=           elem;
      }

    // pop first element and return its value
    //   requires the list to be nonempty (not checked!)
      T pop(void) {
        Listlink<T> *elem= head;
        T value=        elem->value;
        head=           elem->next;
        delete elem;
        return value;
      }

    // .....
};
```

Die generische Klasse `List`

Hier werden zwei generische Klassen definiert: zunächst die Klasse `Listlink<T>`. Sie definiert die Kettelemente zum Aufbau von Listen. Jedes Kettelement enthält zwei Attribute: den gespeicherten Wert `value` und den Verweis `next` auf das nächste Kettelement. Der Typ von `value` ist `T`, der formale Parameter der Klasse.

Die generische Klasse `List` definiert Listen, die Objekte vom Typ `T` als Elemente enthalten. Sie werden implementiert als ein Listenkopf `head`, der auf den Anfang einer Kette aus Kettelementen für den Typ `T` zeigt. In der Klassendefinition wird an mehreren Stellen auf den formalen Parameter `T` und die entsprechende Instanz `Listlink<T>` Bezug genommen.

Diese eine Definition kann für alle Basistypen benutzt werden. Bei Bedarf wird sie geeignet instantiiert: `List<int>` sind etwa die Listen ganzer Zahlen, `List<List<int>>` die Listen, deren Elemente Listen ganzer Zahlen sind, ... Die Definition von Linienzügen nutzt Listen von Punkten:

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

Linienzüge

Generizität kann auch benutzt werden, um das Problem unseres Vertauschungsoperators zu lösen.


```
template <class T> void exchange(T &x, T &y) {
  T  t;
  t= x; x= y; y= t;
}
```

Generische Implementierung des Vertauschungsoperators in `C++`

Hiermit wird eine generische Funktion definiert, die auf beliebige Argumente angewandt werden kann, sofern ihre Typen gleich sind und sie als Ziel von Zuweisungen benutzt werden dürfen. Folgende Nutzungen sind beispielsweise zulässig und führen zum erwarteten Ergebnis:

```
struct S {int i; int f[20];
}      s1, s2;
int    i1, i2;

void f(void) {
  // ...
  exchange(s1,s2);      // s1 <-> s2
  exchange(i1,i2);      // i1 <-> i2
  // ...
}
```

Nutzung von `exchange`

Der Übersetzer instantiiert die generische Funktion bei Bedarf. Er nutzt dabei ein von `C++` unterstütztes Konzept: **Überladung**. `C++` erlaubt, daß verschiedene Funktionen denselben Namen haben, solange sie aufgrund des Typs ihrer Argumente voneinander unterschieden werden können. Funktionsnamen können also eine überladene, eine mehrfache Bedeutung besitzen. Konzeptionell definiert die generische Funktion `exchange` eine unendliche Familie von Funktionen, die alle `exchange` heißen, sich aber durch ihren Argumenttyp voneinander unterscheiden.

Die typische `C++`-Implementierung von Generizität ist aus Sicht des Übersetzerbaus nicht allzu interessant: Werden Instanzen einer generischen Definition referenziert, etwa durch explizite Instantiierung wie in `List<int>` oder indirekt wie etwa durch einen Aufruf von `exchange`, wird die generische Definition als Muster benutzt, um mit den aktuellen Parametern eine konkrete Klasse oder eine konkrete Funktion zu erzeugen: Die generische Definition wird *expandiert*. Das Ergebnis ist eine Klassen- bzw. Funktionsdefinition, wie sie auch vom Programmierer selbst hätte definiert werden können. Der Übersetzer nimmt dem Programmierer nur die stupide Kopierarbeit ab, erspart ihm, neue künstliche Namen zu erfinden und die neuen Namen an richtiger Stelle in die Kopie einzufügen. Wird eine generische Definition mehrmals mit *verschiedenen* Parametern instantiiert, dann enthält das erzeugte Programm mehrere Instanzen der Klasse bzw. Funktion. Jede Klasseninstanz kommt mit ihren eigenen Methodenimplementierungen. Dies führt dazu, daß bei intensivem Gebrauch großer generischer Klassen die Programmgröße dramatisch anwachsen kann.

Bevor wir uns Generizität in Eiffel anschauen, wo eine *generische* Klasse unmittelbar in Zielcode übersetzt und die Implementierung ihrer Methoden für *alle* Instantiierungen der generischen Klasse mitbenutzt werden kann, wollen wir begründen, daß ein C++-Übersetzer in vielen Fällen tatsächlich verschiedene Instantiierungen derselben Funktion/Methode *erzeugen muß*. Der einfache exchange-Operator liefert wieder ein gutes Beispiel. Obwohl exchange<int> und exchange<S> denselben Funktionsrumpf haben, handelt es sich *tatsächlich* um verschiedene Funktionen: In exchange<int> wird 1, in exchange<S> werden 21 ganze Zahlen zwischen den beiden Argumenten ausgetauscht. Sind für zwei Instantiierungen einer generischen Definition die Größen jeweils sich entsprechender aktueller Parameter[26] gleich, dann kann eine einzige Instanz beide Aufgaben erfüllen.[27]

Genau diese Eigenschaft ermöglicht Eiffel, mit einer einzigen Instanz einer generischen Definition jede ihrer Instantiierungen zu implementieren. In Eiffel haben nämlich alle Variablen, Parameter, Rückgabewerte und Objektattribute einen der folgenden *Laufzeit*typen: INTEGER, CHARACTER, BOOLEAN, REAL oder REFERENCE. Die entsprechenden Werte werden einheitlich durch typischerweise jeweils ein Maschinenwort dargestellt. Objekte werden nie direkt manipuliert, sondern stets indirekt über Referenzen. Für Objekte, deren Typ durch einen formalen Parameter einer generischen Definition festgelegt ist, stehen innerhalb der generischen Definition ausschließlich die Operationen Zuweisung und Test auf Identität zur Verfügung. Sie arbeiten nur auf Laufzeitwerten und nicht auf den eigentlichen Objekten und können für alle Laufzeitwerte einheitlich realisiert werden. Deshalb kann eine einzige Instanz einer generischen Klasse die Implementierung sämlicher Instantiierungen übernehmen.

Das Generizitätskonzept ist in C++ viel weiter gefaßt, als wir das bisher veranschaulicht haben: Als formale Parameter einer generischen Definition sind neben Typen auch alle von Funktionsdefinitionen her bekannten Parameterarten zugelassen.
Das folgende Beispiel zeigt einen Ausschnitt aus der Definition **assoziativer Felder**. Das sind Felder, deren Indizes nicht auf ganze Zahlen beschränkt sind, sondern einen beliebigen Typ haben können.

```
template <class Indextype, class Valuetype>
class assoc {                   // abstrakte Definition eines
                                // 'assoziativen Feldes',
                                // mit Werten aus 'Valuetype' und
                                // Indizes aus 'Indextype'

 public:
   // Zugriff auf Komponente mit gegebenem Index
```

[26] Unter der Größe eines Typs wollen wir hier die Größe der Objekte mit diesem Typ verstehen.

[27] Gleiches würde gelten, wenn entweder die Objektgröße im Laufzeitobjekt abgelegt wäre oder die Größeninformation über zusätzliche Parameter an die Funktionen bzw. Methoden übergeben würden. Beide Ansätze lassen sich aber nur schwer mit anderen Merkmalen von C++ vereinbaren.


```
     virtual
     Valuetype operator [] (Indextype const &index);
   // Nachfrage, ob die Komponente 'index' existiert
     virtual
     bool exists(Indextype const &index);
   // Entfernen der Komponente 'index'
     virtual
     void remove(Indextype const &index);
   // ....
};

template <class Indextype, class Valuetype, int maxsize>
class fixed_assoc: public assoc<Indextype,Valuetype> {
                              // assoziatives Feld
                              // mit maximal 'maxsize' Einträgen
                              // implementiert als normales Feld
  pair<Indextype,Valuetype>  vec[maxsize];
                              // Vektor mit 'maxsize' Elementen
  int                 used;   // Anzahl belegter Elemente
};
```

Ausschnitt aus der Definition assoziativer Felder

Das Bespiel definiert zunächst die abstrakte Klasse assoc, die die jedem assoziativen Feld gemeinsamen Methoden festlegt. Die nächste Definition zeigt einen Ausschnitt aus einer Implementierung dieser abstrakten Klasse durch Felder vorgegebener Maximalgröße maxsize, die als Parameter in die generische Definition eingeht. Das zur Implementierung verwendete Feld enthält Paare der Form (index,wert). Soll auf eine Komponente index zugegriffen werden, wird das Feld nach einer entsprechenden Komponente durchsucht und eine Referenz auf den zugehörigen Wert zurückgeliefert. Sollte sie noch nicht existieren, wird sie automatisch angelegt. Eine Instantiierung fixed_assoc<STRING,STRING,100> definiert eine Klasse, in der für die Implementierung assoziativer Felder mit Strings als Indizes und Werten gewöhnliche Felder der Größe 100 verwendet werden.

Wir haben damit jedoch noch immer nicht alle Möglichkeiten der von C++ unterstützten Generizität kennengelernt. Das folgende Beispiel fügt eine weitere hinzu: Wir können für Vektoren gleicher Länge eine Addition definieren, wenn wir ihre Komponenten addieren können. Die folgende Definition kann also nicht beliebig instantiiert werden:

```
template <class component: group, int size>
class vectorgroup: public vector<component,size> {
  // Addition zweier Vektoren
  vectorgroup operator+ (vectorgroup y) {
    int         i;
    vectorgroup result;
```

```
    for (i=0; i<size; i++) result[i]= this[i] + y[i];

    return result;
  }
  // ....
}
```

Ausschnitt aus der Definition `vectorgroup`

Voraussetzung für eine Instantiierung der Form `vectorgroup<comptype,size>` ist, das der Komponententyp `comptype` eine Operation `+` definiert. Die oben verwendete Angabe `class component: group` bringt eine solche Einschränkung zum Ausdruck: Der aktuelle Parameter einer Instantiierung für `vectorgroup` muß eine Teilklasse von `group` sein.[28] Wir haben hier eine Form eingeschränkter Generizität vor uns. Die Einschränkung besteht darin, daß die formalen Typparameter Teilklassen vorgegebener Klassen sein müssen. Die früheren Formen werden demgegenüber als uneingeschränkte Generizität bezeichnet.

`Eiffel` unterstützt durch spezielle Sprachkonstrukte nur die uneingeschränkte Generizität, wobei als formale Parameter generischer Klassendefinitionen nur Typparameter verwendet werden dürfen. Das sehr flexible Vererbungs- und Typkonzept von `Eiffel` erlaubt jedoch die *Emulation* auch eingeschränkter Generizität.

5.5 Übungen

2.1 Das folgende Beispiel zeigt einen weiteren Ausschnitt der Klassenbibliothek graphischer Objekte aus Abb. 5.1 (s.S. 177):

```
class closed_graphical: public graphical_object {
 public:
  // vom Objekt eingeschlossene Fläche
    virtual double area(void);
};

class ellipse: public closed_graphical {
  point  _center;        // Zentrum der Ellipse
  double _x_radius, _y_radius;
                         // Radien der Ellipse
  double _angle;         // Drehung gegen x-Achse

 public:
  // Konstruktor
    ellipse(point &center,
```

[28] Unter Gruppe versteht man in der Mathematik eine Menge mit Operationen +, - und einem Element 0, wenn zusätzlich die sog. Gruppenaxiome erfüllt sind.


```
            double x_radius, double y_radius,
            double angle= 0) {
    _center= center;
    _x_radius= x_radius;  _y_radius= y_radius;
    _angel= angle;
  }

// Ellipsenfläche -- überschreibt 'closed_graphical::area'
  double area(void) { return PI * _x_radius * _y_radius; }

// Distanz zu einen Punkt -- aufwendig!
  virtual double dist(point &);

// Zentrum
  const point& center(void) { return _center; }

// Verschieben -- überschreibt 'graphical_object::translate'
  void translate(double x_offset, double y_offset) {
    _center.translate(x_offset, double y_offset);
  }

// Skalieren -- überschreibt 'graphical_object::scale'
  void scale(double scale_factor) {
    _x_radius *= scale_factor;
    _y_radius *= scale_factor;
  }

  // ....
};

class circle: public ellipse {
 public:
  // Konstruktor
    circle(point &center, double radius){
      ellipse(center,radius,radius);
    }

  // Distanz zu einem Punkt -- überschreibt 'ellipse::dist'
    virtual double dist(point &p) {
      double center_dist= _center.dist(p);
      if (center_dist <= radius) return 0;
      else return center_dist - radius;
    }

  // ....
```

```
};
```

Die Klassen `closed graphical`, `ellipse` und `circle`

Übersetzen sie die Methode `ellipse::translate` in die sie implementierende Funktion.

3.1 Bestimmen Sie die Methodentabellen und die Methodenindizes gemäß Abschnitt 5.3.1 für die in der vorherigen Aufgabe definierten Klassen und ihre Methoden.

3.2 Bestimmen Sie die Methodentabelle und die Methodenindizes gemäß dem Übersetzungsschema für unabhängige mehrfache Beerbung (s.S. 193) von `circle` und übersetzen sie einen Methodenaufruf der Form `c.dist(p)`, wobei `c` ein Kreis und `p` ein Punkt ist.

3.3 Bestimmen sie die Indextabelle und die Bindungsfunktionen gemäß dem Übersetzungsschema für abhängige mehrfache Beerbung für die Klasse `polygon`.

```
class polygon: public polyline, public closed_graphical {};
```

Polygone

`polyline` ist auf Seite 214 definiert.

3.4 Die Neudefinition einer Klasse soll keine Auswirkungen auf das bestehende Klassengefüge verursachen. Aufgrund der Teiltypregel müssen jedoch auch Methoden schon bestehender Klassen mit Objekten der neuen Klasse arbeiten können. Durch Überschreiben von Merkmalen beerbter Klassen kann deren Sicht auf die neuen Objekte inkonsitent werden. Redefinitionen sind deshalb nur eingeschränkt erlaubt. Drei unterschiedliche Aspekte müssen beachtet werden:

1. die Bedeutung (Semantik) der Merkmale,
   jedes Attribut und jede Methode hat eine Bedeutung oder Aufgabe, die durch das Überschreiben nicht beeinträchtigt werden darf. So hat etwa die Methode `scale` die Aufgabe, ein graphisches Objekt zu skalieren. Eine Redefinition muß dieselbe Aufgabe erfüllen, sowohl auf der aktuellen Ebene wie auf der Ebene einer Oberklasse betrachtet.
2. die Einschränkungen durch das Typsystem,
   Redefinitionen dürfen nicht zu Typinkonsistenzen führen.
3. die Einschränkungen des Übersetzungsschemas,
   Redefinitionen können die bei der Übersetzung gemachten Annahmen ungültig werden lassen und dürfen in diesem Fall nicht zugelassen werden.

Der erste Aspekt ist sehr wesentlich, aber ein Übersetzer verfügt normalerweise nicht über die notwendige Spezifikation, um die semantische Zulässigkeit zu verifizieren. In dieser Aufgabe betrachten wir Einschränkungen durch das Typsystem.

Der Übergang von einem Typ zu einem Teiltyp heißt **Typverschärfung**, der Übergang von einem Typ zu einem Obertyp **Typabschwächung**. Ein Prototyp (einer Methode) wird verschärft, wenn die Typen des Rückgabewertes und der Ausgabeparameter verschärft und die Typen der Eingabeparameter abgeschwächt werden.

(a) Begründen Sie, daß aus Sicht des Typsystems keine Einwände gegen eine Verschärfung des Prototyps einer redefinierten Methode bestehen. Zeigen Sie an Hand von Beispielen, daß der Prototyp öffentlicher, also von beliebigen Klassen aktivierbarer Methoden höchstens verschärft werden kann; jede andere Abänderung kann zu Typfehlern bei bisher korrekten Aufrufen führen. Begründen Sie, daß eine nicht verschärfende Prototypänderung auch für redefinierte private Methoden nur unter sehr eingeschränkten Bedingungen zulässig ist. Zur Verifikation der Zulässigkeit ist Information über die Oberklassen notwendig, die über die Kenntnis der Typen ihrer Attribute und Prototypen ihrer Methoden hinausgeht – welche?

(b) Die Sprache `Eiffel` erlaubt in einer abgeleiteten Klasse, den Typ eines geerbten Attributes zu verschärfen. Begründen Sie, daß damit Attribute notwendigerweise von einer fremden Klasse nur gelesen werden können. Zeigen Sie anhand eines Beispiels, daß die Verschärfung des Typs eines Attributes nur unter sehr eingeschränkten Bedingungen zulässig ist. Zur Verifikation der Zulässigkeit ist Information über die Oberklassen notwendig, die über die Kenntnis der Typen ihrer Attribute und Prototypen ihrer Methoden hinausgeht – welche?

3.5 Diese Aufgabe untersucht Einschränkungen der Redefinitionsmöglichkeiten auf Grund des verwendeten Übersetzungsschemas.

(a) Unser Übersetzungsschema für einfache Vererbung läßt die Verschärfung des Prototypen bei Redefinition von Methoden zu, nicht so unsere Übersetzungsschemata für die mehrfache Vererbung. Ermitteln Sie, welche Voraussetzungen dieser beiden Schemata durch eine Prototypverschärfung verletzt werden können.
Suchen Sie nach einer Erweiterung unserer Schemata, die die Verschärfung von Prototypen redefinierter Methoden uneingeschränkt erlaubt.
Hinweis: Die redefinierte Methode wird in zwei Varianten in die Methodentabelle aufgenommen. Die neue Definition wird unmittelbar für Sichten ab der neu definierten Klasse verwendet; Sichten von Oberklassen hingegen verwenden die neue Definition nicht direkt, sondern eingehüllt in einen Rahmen, der alte und neue Typen für Parameter und Rückgabewert aneinander anpaßt.

(b) Unser Übersetzungsschema für einfache Vererbung erlaubt die Verschärfung von Attributtypen (sofern durch die Verschärfung keine Typinkonsistenzen auftreten), die Schemata für die mehrfache Vererbung tun dies nicht. Können Sie den Grund dafür angeben?

(c) Die Sprache `Eiffel` erlaubt einer Klasse, eine geerbte parameterlose Methode als ein Attribut umzudefinieren. Geben Sie an, wie unsere Übersetzungsschemata eine solche Redefiniton behandeln müßten.

3.6 Unsere Übersetzungsschemata für mehrfache Vererbung verursachen einen verhältnismäßig hohen Speicheroverhead pro Objekt. In dieser nicht ganz einfachen Aufgabe entwickeln wir ein Übersetzungsschema, das diesen Overhead drastisch reduziert.

(a) Für jede Oberklasse A einer Klasse C gibt es eine A-Sicht auf C-Objekte. Unsere Schemata müssen jedoch nicht jede Sicht getrennt repräsentieren; häufig fällt die Repräsentation für verschiedene Sichten zusammen. Für jede verschiedene Sichtenrepräsentation enthalten die Objekte je einen Zeiger als Overhead. Um den Overhead zu reduzieren, müssen wir also die Anzahl verschieden repräsentierter Sichten verringern.

Wir gehen jetzt vorläufig davon aus, daß wiederholtes Erbe stets nur ein einziges Mal instantiiert wird. In diesem Fall enthält ein C-Objekt für jeden Obertyp A genau ein A-Teilobjekt. Dies erlaubt uns, *alle* Sichten einheitlich durch einen Zeiger auf das Objekt zu repräsentieren, sofern wir beim Merkmalszugriff neben dem Attribut bzw. der Methode auch die entsprechende Klasse mit angeben. Der Übersetzer bindet also Attribute und Methoden an Paare (Klassenid, Offset) bzw. (Klassenid, Methodenindex). Klassenid ist dabei eine eindeutige Identifikation der Klasse, die das Merkmal erstmals[29] eingeführt hat. Der Zeiger am Anfang eines Objektes zeigt auf einen zweistufigen Index. Die Indextabelle erster Stufe wird indiziert mit der Klassenid und liefert den Offset des zu dieser Klasse gehörenden Teilobjektes sowie den Verweis auf ihre Methodentabelle. Abb. 5.14 veranschaulicht die sich ergebende Struktur für die in Abb. 5.6 (s.S. 191) dargestellte Situation. Für die Effizienz dieses Verfahrens ist von entscheidender

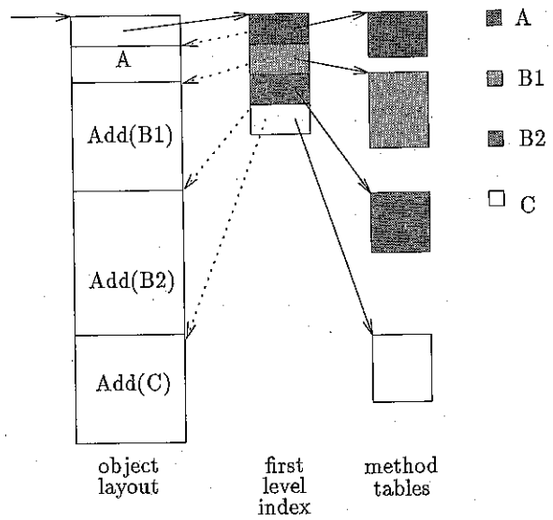

Abb. 5.14: Objekt und Indexstruktur bei klassenbezogenem Merkmalszugriff

Bedeutung, daß die Indizierung des Indexes erster Stufe mit den Klassenids effizient durchgeführt werden kann. Wünschenswert wäre die Verwendung kleiner ganzer Zahlen als Klassenids, da dann der Index erster Stufe durch gewöhnliche

[29] innerhalb der Vererbungshierarchie

Tabellen realisiert werden kann. Allerdings müssen Klassenids folgende Eigenschaften haben:

- Innerhalb eines Programms muß eine Klasse überall dieselbe Klassenid erhalten,
- enthält ein Programm die Klasse C, dann müssen alle Oberklassen von C verschiedene Klassenids erhalten.

Ein Übersetzer hat üblicherweise nur ein beschränktes Wissen über die Klassen, die in ein Programm integriert werden. Aus diesem Grund kann er die Klassenids nicht vergeben. Der Programmbinder bindet vorübersetzte Klassen zu einem Programm zusammen und kennt damit alle für ein Programm relevanten Klassen. Er ist eine geeignete Instanz, die vom Übersetzer verwendeten Klassenid-Namen an die eigentlichen Klassenids, kleine ganze Zahlen, zu binden. Ebenfalls zu seinen Aufgaben gehört der Aufbau der Indextabellen erster Stufe auf Basis der Information, die der Übersetzer in Form von Tripellisten (Klassenid-Name, Offset, Methodentabelle) für jede Klasse vorbereitet hat.
Führen Sie die Details aus.

(b) Wenn wiederholtes Erbe A (partiell oder vollständig) mehrfach instantiiert wird, ist obiges Verfahren nicht mehr anwendbar, da die Angabe von A allein beim Merkmalzugriff nicht ausreicht, um zwischen den verschiedenen Instanzen von A zu unterscheiden.
Zeigen Sie für die in Abb. 5.5 (s.S. 191) dargestellte Situation, daß die Sichten auf B1 und B2 notwendigerweise verschieden repräsentiert werden müssen.

(c) Kombinieren Sie den im ersten Teil der Übung skizzierten Ansatz mit unseren früheren Übersetzungsschemata. Sie sollten dadurch neue Übersetzungsschemata für unabhängige bzw. allgemeine abhängige mehrfache Beerbung erhalten, deren Anzahl verschieden repräsentierter Sichten auf C-Objekte durch die maximale Anzahl zu unterscheidender Instanzen *desselben* Obertyps gegeben wird.[30]

4.1 Entwerfen Sie eine generische Definition für Warteschlangen, das sind first-in-first-out Queues.

5.6 Literaturhinweise

Simula67, der Urvater objektorientierter Sprachen, ist in [DMN84] und in [Sim87] beschrieben. [GR83] beschreibt Smalltalk80. C++ wird in [Str91] definiert. Die Grundlage für die Standardisierung von C++ durch ANSI ist [C++90]. [Mey88] und [Mey92] geben eine gute Einführung in Eiffel. [Cox86] beschreibt Objective-C. Objektorientierte Spracherweiterungen von Lisp sind in [BS82, Can80, Hul84] beschrieben. Java wird in [JGS96] definiert; [Har] ist empfehlenswert, wenn man mit Java programmieren möchte.

[30] Muß C etwa zwei A-Instanzen und drei B-Instanzen unterscheiden, werden drei verschieden repräsentierte Sichten benötigt.