Perspektiven: Persistente Objekte mit anwendungsspezifischer Struktur und Funktionalität

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Wolfgang Reddig

aus

Heidenheim/BadenWürttemberg

Bonn 2001

Vorwort

Die vorliegende Arbeit entstand aus meiner Forschungsarbeit in mehreren Projekten am Institut für Informatik III der Rheinischen Friedrich-Wilhelm-Universität Bonn, der ich bis Ende 1998 als wissenschaftlicher Mitarbeiter unter der Leitung von Professor Cremers angehörte. Die schriftliche Ausarbeitung erfolgte jedoch nach über einjähriger wissenschaftlicher Pause, nachdem ich einen Wechsel in die Software-Industrie gut überstanden hatte und mich trotz der beruflichen Trennung von der Universität dennoch der fortgesetzten Ermutigung und Motivation seitens Herrn Cremers ausgesetzt sah, die in jahrelanger Arbeit erbrachten Ergebnisse doch noch in schriftlicher Form festzuhalten.

Dass diese Aufgabe neben dem Berufsleben nun dennoch abgeschlossen wurde ist nicht nur der mehr oder weniger wohlwollenden Duldung, sondern der tatkräftigen Unterstützung meiner Frau zu verdanken, die wo immer es ging versuchte, mir den unerlässlichen Freiraum zu verschaffen. Ich kann deshalb nur hoffen, dass ich den häufigen Verzicht meiner Familie auf Vater und Ehemann in der kommenden Zeit wenigstens teilweise ausgleichen können werde.

Der Inhalt dieser Arbeit stützt sich fast ausschließlich auf Ergebnisse meiner Arbeit im Institut für Informatik III. Und diese Ergebnisse wären ohne die angenehme und sehr fruchtbare Arbeit am Institut und die vielen Ideen-gebenden Diskussionen mit anderen Wissenschaftlern unmöglich gewesen. Neben Herrn Cremers selbst möchte ich deshalb meine Kollegen Andreas Bergmann, der lange Jahre im gleichen Projekt arbeitete und dadurch die Umsetzung vieler Konzepte erst möglich gemacht hat, sowie Dirk Flachbart, Thomas Bode und Günter Kniesel namentlich erwähnen. Vor allem diese vier vermochten durch ihren Sachverstand so manchen Irrweg in meinen Ideen zu vermeiden und standen immer wieder gerne und umfassend zu Hilfe, wenn ein Problem scheinbar keine Lösung hatte.

Vielleicht der wichtigste Mitstreiter war und ist jedoch mein Freund und Kollege Oleg Balovnev, der nicht nur viele Jahre lang jeden Tag durch das stoische Ertragen des Rauchs von mindestens zwanzig selbstgedrehten Zigaretten in unserem Büro ein leuchtendes Beispiel der sprichwörtlichen Leidensfähigkeit des russischen Volkes gab, der nicht nur bei jeder Verhedderung in Implementierungsdetails und endlosen Sitzungen vor Compiler und Debugger immer der letzte Rettungsanker war, der nicht nur mit sehr trockenem Humor mir dazu verhalf, die wissenschaftliche Arbeit nicht ständig allzu ernst zu nehmen, sondern der auch durch sein tiefes Verständnis für

Software-Technologie und seine den meinigen sehr ähnlichen beruflichen Interessen der bestmögliche Diskussionspartner war und ist.

1 Einleitung

In komplexen objektorientierten Systemen stellt sich sehr häufig das Problem, Objekte für unterschiedliche Anforderungen oder gar unterschiedliche Anwendungen in verschiedenen Ausprägungen darzustellen bzw. nutzbar zu machen. Klassische objektorientierte Programmiersprachen bieten jedoch nicht die Möglichkeit, Objekte je nach Anwendungskontext als Instanzen unterschiedlicher Typen bzw. Klassen darzustellen. Eine derartige Funktionalität wird jedoch häufig benötigt, um Anwendungen unterschiedlicher Disziplinen eine ihnen angemessene Arbeit mit üblicherweise persistenten Objekte zu ermöglichen. Als Beispielszenario betrachten wir in dieser Arbeit immer wieder Bauingenieuranwendungen, die die unterschiedlichen Aufgaben im Bereich Planung und Konstruktion von Gebäuden unterstützen und idealerweise auf identischen persistenten Objekten operieren. Um zu verdeutlichen, dass auch in völlig anderen Arten von Systemen die Notwendigkeit von der Kontextspezifischen Nutzung von Objekten besteht, stellen wir an dieser Stelle kurz ein Beispiel aus dem Compilerbau vor, das in ähnlichem Zusammenhang häufig zitiert wurde [GHJV94].

Ein Compiler erzeugt aus einem Quelltext einen abstrakten Syntaxbaum, der seinerseits aus einer Reihe von Knoten besteht. Auf diesen Koten sind eine Reihe von Aufgaben auszuführen: Syntaxprüfung, semantische Analyse, Codegenerierung, Generierung von Debugging Informationen, Pretty Printing, um nur einige der Möglichkeiten zu nennen. Typisch ist an diesem Beispiel, dass die Anwendung über eine komplexe Objektstruktur iteriert und auf den einzelnen in diesem Netzwerk bzw. Baum enthaltenen Instanzen „ähnliche“ Operationen oder Operationen mit gleichem Namen und gleicher Signatur ausführt. Der Ablauf bzw. der Kontrollfluss ist also stets gleich oder zumindest sehr ähnlich. Auch die Instanzen (hier: Knoten des Baumes), auf denen diese Operationen auszuführen sind, sind identisch. Die Operationen selbst differieren jedoch sehr stark und benötigen auch ganz unterschiedlichen Daten in den Objekten, auf denen sie aufgerufen werden.

Objektorientiertes Design erhebt oft den Anspruch, bestehende Systeme nicht durch Manipulation existierenden Codes, sondern allein oder zumindest ganz überwiegend durch Hinzufügung neuer (Unter-) Klassen an neue oder veränderte Anforderungen anpassen zu können. Für unser Compiler-Beispiel bedeutet dies, dass eine Hinzufügung von optimierter Codegenerierung möglich sein sollte, ohne die Klassen für die Knoten im Baum zu erweitern oder den Code sonst wie umstellen zu müssen. Im Rahmen dieser Arbeit werden wir untersuchen, wie weit derartige Ansprüche mit den Mitteln traditioneller objektorientierter Programmiersprachen zu erfüllen sind

bzw. welche Eigenschaften Sprachen aufweisen müssen, um diese Ziele zu erreichen.

Oft werden objektorientierte Programmiersprachen nach der Eigenschaft der sogenannten Typsicherheit klassifiziert. Als typsicher gelten Sprachen, bei denen zur Laufzeit Referenzen, die zur Übersetzungszeit mit einem Typ versehen waren, niemals auf Objekte eines anderen, zum zur Übersetzungszeit vereinbarten Typ nicht kompatiblen anderen Typ verweisen. Ohne in eine – zwar sinnvolle und notwendige, aber an dieser Stelle nicht angebrachte – Diskussion über die einzelne Abstufungen der Typsicherheit, wie sie in verbreiteten objektorientierte Programmiersprachen realisiert ist abzurutschen, nennen wir vereinfacht die Sprachen C++, Java, und Eiffel als die prominentesten der typsicheren Sprachen[1]. Demgegenüber stehen Smalltalk als zwar klassenbasierte, jedoch nicht typsichere und Self als ganz und gar untypisierte Sprache.

Die Vor- und Nachteile dieser beiden Richtungen sind in den letzten Jahren und Jahrzehnten ausführlich diskutiert worden. Klar ist, dass ein strenges und sicheres Typsystem ein Design nahe legt, indem einzelne Bereiche oft sehr leicht voneinander isoliert werden können. Zusätzlich ist die Anzahl der erst zur Laufzeit erkennbaren Fehler durch die Typsicherheit sehr stark eingeschränkt. Nicht zuletzt kann ein Compiler für typsichere Sprachen die Annahmen über die Typisierung der Objekte, die in einem solchen System manipuliert werden, ausnutzen, um sehr effizient über Offsets auf Interna von Datenstrukturen zuzugreifen. Auf der anderen Seite erlauben ungetypte Sprachen eine wesentlich höhere Flexibilität und damit potentiell eine höhere Wiederverwendungsrate bereits implementierter Komponenten. Denn diese Komponenten und ihre Schnittstellen werden nicht durch strenge Typisierungen unnötig eingeschränkt und lassen sich deshalb in mehr Kontexten nutzen als ihre getypten Verwandten.

Diese Arbeit bewegt sich wegen der genannten Vorteile im Umfeld typsicherer Sprachen. Ganz offensichtlich sind jedoch Konzepte notwendig, welche die Typsysteme objektorientierter Sprachen ohne Verlust an Sicherheit flexibler machen. Flexibler bedeutet bei Objektorientierung meist, dass Klassen bzw. ihre Instanzen in möglichst vielen Zusammenhängen und Anwendungssituationen nutzbar sind. Und dies hat als Grundvoraussetzung, dass das Typsystem der Sprache die Ersetzbarkeit von Objekten durch andere Objekte sehr weitgehend unterstützt. Die

---

[1] Natürlich enthalten alle drei genannten Sprachen mehr oder weniger bekannte „Löcher“ in ihren Typsystemen, auf die hier jedoch nicht eingegangen wird.

vorliegende Arbeit versucht, die Ersetzbarkeit in Programmiersprachen um eine Dimension zu erweitern, ohne an Typsicherheit Einbußen hinnehmen zu müssen.

Abschnitt 2 zeigt, dass traditionelle objektorientierte Programmiersprachen und auch objektorientierte Datenbank Management Systeme keine Unterstützung für derartige Szenarien bieten. Wir stellen deshalb in Abschnitt 3 ein Typkonzept vor, welches Anpassungen und Erweiterungen besser unterstützt und somit objektorientierte Sprachen flexibler und objektorientierte Systeme anpassbarer macht. In Abschnitt 4 stellen wir Realisierungstechniken für Sprachen vor, die ein derartig flexibles Typsystem unterstützen. Dabei gehen wir auch ausführlich auf Effizienzfragen ein und betrachten Optimierungstechniken, die helfen sollen, die entworfene Sprache in ihrer Performanz vergleichbar zu klassischen, streng getypten objektorientierten Sprachen zu machen. Im Rahmen des DFG Schwerpunktprogramms „Objektorientierte Modellierung in Planung und Konstruktion“ wurde im Teilprojekt „Definition und Implementierung eines Bauwerkmodellkerns“ am Institut für Informatik III der Universität Bonn das System CEMENT (**C**ivil **E**ngineering **M**odeling **E**nvironme**nt**) realisiert. Die dazu gehörige Programmiersprache CPL (**C**EMENT **P**rogramming **L**anguage) ist eine vollständige Implementierung aller in dieser Arbeit vorgestellten Konzepte. Anschauliche Beispiele im Abschnitt 4 lehnen sich deshalb an diese Sprache an. CPL orientiert sich in seiner Syntax an C++, soweit nicht Eigenschaften der Sprache betroffen sind, die durch C++ oder andere traditionelle objektorientierte Sprachen nicht abgebildet werden können. In Abschnitt 5 schließen wir die Arbeit mit einer Zusammenfassung und einer Betrachtung weiterer Anwendungsfelder ab.

2 Motivation

Die Einhaltung von Datenkonsistenz ist eine der wichtigsten Aufgaben in Anwendungsszenarien, bei denen unterschiedliche Werkzeuge mit Objektmodellen arbeiten, die letztlich dieselben „real world“ Objekte beschreiben. In objektorientierten Datenbanksystemen fehlen jedoch bislang Möglichkeiten, Klassen eines globalen Datenbankschemas für die Bedürfnisse spezieller Anwendungen anzupassen. Bei genauerem Hinsehen stellt sich aber heraus, dass schon auf der Ebene der Programmiersprachen die Modellierung von Objekten in unterschiedlichen Ausprägungen nur sehr unzureichend unterstützt wird. Da heutige ODBMS üblicherweise sehr eng an objektorientierte Programmiersprachen gekoppelt sind ist es verständlich, dass diese Systeme in dieser Hinsicht keine Ansätze bieten.

Im Folgenden stellen wir eine typische, wenn auch vereinfachte Problemstellung aus dem Bereich der Bauingenieuranwendungen vor und diskutieren an Hand dieses Beispiels die Möglichkeiten klassischer objektorientierter Sprachen, die Modellierung anwendungsspezifischer Eigenschaften zu unterstützen. Anschließend stellen wir in Kapitel 3 unseren Ansatz vor, der die in diesem Abschnitt beschrieben Schwächen vermeidet und eine sehr flexible und einfache Beschreibung von Objekten, die Struktur und Verhalten ihrem jeweiligen Kontext ihrer Nutzung anpassen, ermöglicht.

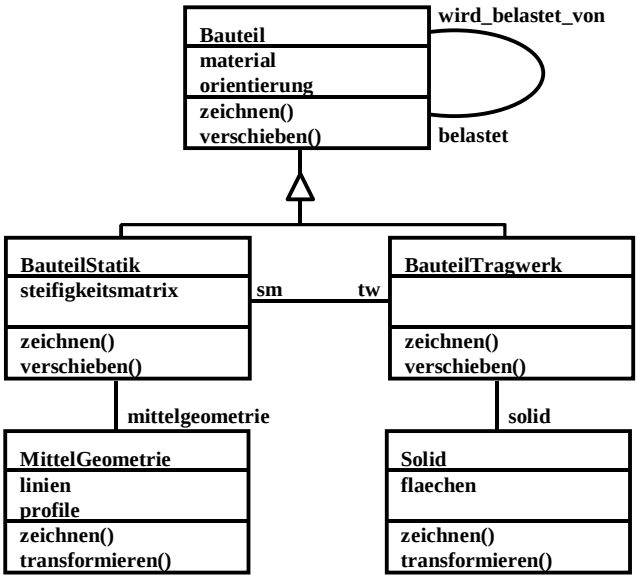

Abbildung 1

Als Beispiel betrachten wir das Schema in Abbildung 1 in OMT-Darstellung [RBPE93]. Die Klassen *BauteilTragwerk* und *BauteilStatik* beschreiben Bauteile aus der Sicht verschiedener Bauingenieurdisziplinen. Beide Disziplinen nutzen

geometrische Repräsentationen, die ihren spezifischen Anforderungen entsprechen - eine 3D-Körper Darstellung *Solid* für die Tragwerkmodellierung und eine Systemmittelgeometrie *MittelGeometrie* für die Statik.

2.1 Redundante Objekte

Das Schema aus Abbildung 1 erzwingt die Erzeugung unterschiedlicher Softwareobjekte für die Darstellung ein und desselben konkreten Bauteils. Diese Vorgehensweise bringt durch die inhärente Redundanz offensichtlich eine Vielzahl von Problemen mit sich: Änderungen an einem Objekt der Klasse *BauteilTragwerk* müssen im korrespondierenden *BauteilStatik* Objekt semantisch äquivalent nachgezogen werden. Zur Veranschaulichung dieser Problematik haben wir den Code für die *verschieben*()-Methoden in Abbildung 2 dargestellt.

```
void
BauteilTragwerk::verschieben(double x, double y, double z)
{
  solid->transformieren(x, y, z);
  sm->verschieben(x, y, z);
  // weitere Teilmodelle?
}
```

Abbildung 2

Besonders problematisch ist bei diesem Design, dass das Hinzufügen weiterer Teilmodelle – z. B. das numerische Modell oder das architektonische Modell – die Veränderung der Implementierung der Methode *BauteilTragwerk::verschieben* sowie aller korrespondierenden Methoden in den übrigen Ableitungen der Klasse *Bauteil* erzwingt.

Weitere Probleme entstehen durch die Verbindungen zwischen Bauteilen. In Abbildung 1 ist die Beziehung *Belastung* als Beziehung zwischen Objekten der Klasse *Bauteil* dargestellt. Diese Modellierung erlaubt den Unterklassen *BauteilTragwerk* und *BauteilStatik* jedoch nicht, die spezielleren Eigenschaften dieser Klassen auch bei der Navigation über die Beziehung hinweg zu nutzen. Um dies zu ermöglichen, wäre die kovariante Redefinition des Typs der Beziehung *Belastung* notwendig. Sprachen, die eine solche Vorgehensweise erlauben, sind jedoch nicht typsicher, wie man sich an dem einfachen Beispiel aus Abbildung 3 leicht klarmachen kann. In diesem Beispiel entsteht sogar über den Typfehler hinaus ein Fall, in dem Instanzen der Klasse *BauteilStatik* mit Instanzen der Klasse *BauteilTragwerk* verbunden sind – eine Inkonsistenz, Compiler bzw. Typsystem der Sprache nicht verhindern können.

```
class Bauteil {
   ...
   void belasteDurch(Bauteil* b) {
     belastet->insert(b);
   }
   Set<Bauteil*>* belastet;
};
class BauteilTragwerk extends Bauteil {
   ...
   Set<BauteilTragwerk*>* belastet;
};

Bauteil* tw = new BauteilTragwerk;
Bauteil* ts = new BauteilStatik;
tw->belasteDurch(ts);
// Bauteil::belastet enthält ein Objekt falschen Typs
```

Abbildung 3

Die Modellierung der anwendungsspezifischen Eigenschaften und Verhaltensweisen von Objekten durch einfache Unterklassenbildung ist durch ihre Fehleranfälligkeit und Redundanz offensichtlich wenig geeignet, große Datenmengen komplex strukturierter Objekte anwendungsübergreifend zu beschreiben.

2.2 Multiple Vererbung

Bevor wir als zweite Designalternative den Einsatz von Mehrfachvererbung diskutieren, gehen wir kurz darauf ein, warum Vererbung in diesem Szenario überhaupt attraktiv ist. Die Möglichkeit, Funktionsaufrufe dynamisch, d. h. zur Laufzeit und nicht zur Übersetzungszeit zu binden, erlaubt die Formulierung von Algorithmen auf sehr abstraktem Niveau. Als Beispiel betrachten wir das Zeichnen eines Gebäudes in Abbildung 4: ein Gebäude wird im Wesentlichen als eine Aggregation von Bauteilen beschrieben. Das grafischen Zeichnen eines solchen Gebäudes geschieht deshalb durch Zeichnen der einzelnen Bauteile. Zu diesem Zweck wird die Methode *Gebaeude::zeichnen* durch eine einfache Schleife[2] implementiert, die als Iteration über eine Menge von Referenzen auf Objekte der Klasse *Bauteil* agiert und jeweils die Methode *zeichnen* aufruft. Die tatsächliche Funktion, die sich hinter diesem Aufruf von *zeichnen* verbirgt, wird erst zur Laufzeit, abhängig vom konkreten Typ der variablen *b* gewählt. In unserem Beispiel gehen wir

[2] Die syntax `collection.apply<variable>(expression(variable))` bewirkt einen Aufruf des Ausdrucks `expression(variable)` auf sämtlichen Elementen der Kollektion `collection`. Bei dieser Iteration wird das aktuelle Element der Kollektion mit der zu diesem Zweck in spitzen Klammern deklarierten Hilfsvariablen `variable` bezeichnet.

davon aus, dass sich unter der Klasse *Bauteil* sehr viele Unterklassen einerseits für unterschiedliche Typen von Bauteilen wie Träger, Balken, Platten etc. und andererseits die schon erwähnten anwendungsspezifischen Ausprägungen *BauteilTragwerk* und *BauteilStatik* finden. Die Formulierung der Methode *Gebaeude::zeichnen* braucht diese Spezialfälle jedoch in keiner Weise zu berücksichtigen – ihre Implementierung abstrahiert vom konkreten Typ des Bauteils und verlässt sich auf das dynamischen Binden. Ein ebenso wichtiger Vorteil dieser Technik ist die Erweiterbarkeit: das Hinzufügen neuer Ableitungen der Klasse *Bauteil* beeinflußt in keiner Weise die Implementierung der Client-Klasse *Gebaeude* – die Methodenimplementierung von *Gebaeude::zeichnen* paßt sich sozusagen „automatisch“ an.

```
class Gebaeude {
  ...
  virtual void zeichnen() const {
    bauteile->apply<b>(b->zeichnen());
  }
  virtual void fuegeEin(Bauteil* b) {
    bauteile->insert(b);
  }
  ...
  Set<Bauteil*>* bauteile;
};
```

Abbildung 4

Diese Art des Polymorphismus, wegen der Inklusionsbeziehung zwischen den Instanzmengen der Oberklasse und ihren Unterklassen häufig *Inklusionspolymorphismus* genannt, unterscheidet sich in seiner Mächtigkeit vom sogenannten *parametrischen Polymorphismus*, der in Programmiersprachen üblicherweise mit parametrisierten Typen bzw. Klassen realisiert wird. Bei parametrischem Polymorphismus, in Abbildung 5 mit dem C++-Konstrukt *template* dargestellt, wird zur Übersetzungszeit des Quellcodes eindeutig entschieden, welcher Typ den Variablen im Rumpf der Methode *zeichnen* zugewiesen wird. Eine Abstraktion von der speziellen Ausprägung einer Bauteilklasse wird deshalb von dieser auch als *generische Programmierung* bezeichneten Technik nicht ohne dynamisches Binden unterstützt.

```
template <class T> class BauteilMenge {
   ...
   void zeichnen() const {
      set->apply<b>(b->zeichnen());
   }
   ...
   Set<T*>* set;
};

BauteilMenge<Platte>* p = new BauteilMenge<Platte>;
p->zeichnen();
BauteilMenge<Traeger>* t = new BauteilMenge<Traeger>;
t.zeichnen();
```

Abbildung 5: parametrischer Polymorphismus/generische Programmierung

Um die Vorteile des Inklusionspolymorphismus zu erhalten und dennoch persistente Objekte redundanzfrei beschreiben zu können kann mehrfache Vererbung, wie in Abbildung 6 skizziert, verwendet werden. In dieser Modellierung müssen alle Bauteil-Objekte als Instanzen der alle Eigenschaften umfassenden Klasse *BauteilTragwerkStatik* instanziiert werden – die beiden direkten Oberklassen *BauteilTragwerk* und *BauteilStatik* dienen lediglich als Schittstellenspezifikation. Zwar vermeidet dieser Ansatz das Redundanz-Problem, impliziert jedoch eine Reihe anderer, nicht minder schwerwiegender Nachteile:

- Es gibt keine klare Trennung zwischen den einzelnen Applikationen, da die Klassen, welche die Objekte erzeugen, die Vereinigung aller Eigenschaften der unterschiedlichen Anwendungen beinhalten,
- das Hinzufügen einer weiteren Applikation mit neuen Anforderungen an Struktur und Verhalten für bereits existierende Klassen und ihrer persistenten Objekte erfordert Schemaevolution der Datenbank sowie die Anpassung der bereits existierenden Anwendungen,
- der Aufruf von dynamisch gebundenen Methoden liefert immer die Implementierung der alle Eigenschaften umfassenden Klasse *BauteilTragwerkStatik* (vgl. 2.2.3).

Diese Punkte werden in den folgenden Abschnitten ausführlich diskutiert.

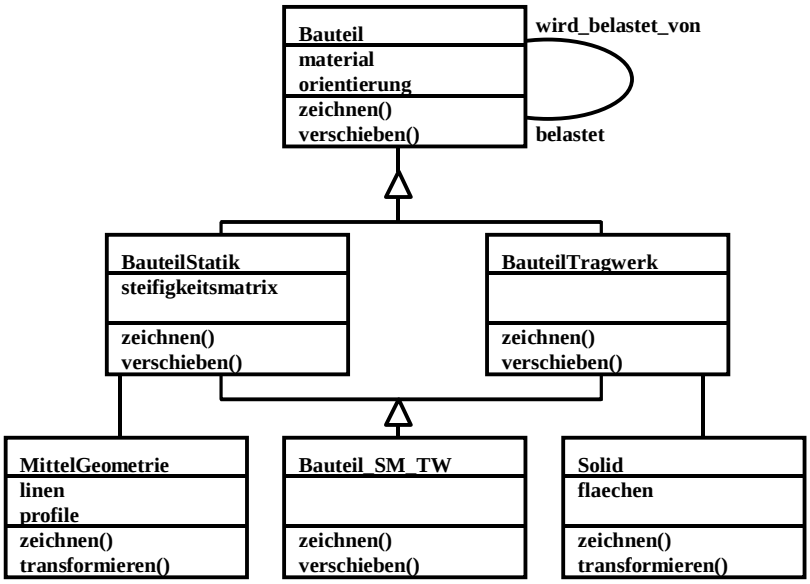

Abbildung 6: Redundanzfreie Modellierung mittels multipler Vererbung

2.2.1 Erweiterbarkeit

Werden weitere Teilproduktmodelle in das System integriert, so ergibt dich üblicherweise die Notwendigkeit, neue Attribute in den teilmodellspezifischen einzufügen. So würde z. B. das numerische Modell eines Bauwerkes eine diskretisierte Darstellung von Bauteilen für Finite Elemente Berechnungen ablegen. Diese physische Erweiterung der speziellsten Klasse hat in vielen objektorientierten Datenbanksystemen eine Schemaevolution zur Folge, die mit der physischen Migration von Objekten verbunden ist. Einige Systeme [ODI99] bieten keine Unterstützung für Schemaevolution im laufenden Betrieb an, so dass für die Migration von Objekten die Datenbank vorübergehend für alle Anwendungen gesperrt werden muss. Abschnitt 3.4 diskutiert diese Problematik ausführlich.

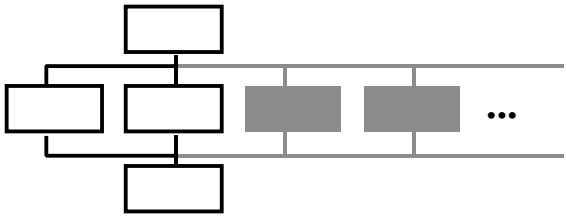

Abbildung 7: Erweiterbarkeit bei multipler Vererbung I

Abbildung 7 zeigt einen möglichen Ansatzpunkt für die Erweiterung um zusätzliche Teilproduktmodelle. Die schwarz umrandeten Boxen stehen für die in Abbildung 6 dargestellten vier Bauteilklassen, die grauen Boxen auf der Ebene von *BauteilTragwerk* und *BauteilStatik* deuten weitere Ableitungen von *Bauteil* an. Bei diesem Vorgehen führt jede Integration eines neuen Teilmodells auch zu einer Änderung der alle Eigenschaften umfassenden Klasse, die sich in der Skizze ganz unten findet. Diese Klasse erhält jeweils eine zusätzliche Oberklasse und muss

dementsprechend in ihren Methodenimplementierungen angepaßt werden, wie in Abbildung 8 gezeigt. Dies notwendige Modifikation des bereits bestehenden Codes steht jedoch in Widerspruch zum eigentlichen Anspruch der objektorientierten Modellierung und Programmierung: Anpassung durch Hinzufügen neuer Klassen anstatt Änderung des Codes. Dieses auch als *open-closed-principle* [Meye98] bekannte Konzept geht davon aus, dass notwendige Veränderungen im Verhalten eines objektorientierten Systems am besten ausschließlich durch Integration neuer (Unter-) Klassen ins System erreicht werden. Das System ist damit *offen* für Veränderungen, bereits implementierten Klassen sind jedoch *abgeschlossen* gegen Modifikationen.

```
void
Bauteil_alle_Modelle::verschieben(double x, double y, double z)
{
  BauteilTragwerk::verschieben(x, y, z);
  BauteilStatik::verschieben(x, y, z);
  BauteilNumerischesModell::verschieben(x, y, z);
  // weitere Teilproduktmodelle?
}
```

Abbildung 8

Um die Verletzung des open-closed-principles zu vermeiden, können wir die Modifikation der speziellsten Klasse durch Bildung von jeweils zwei Unterklassen je neu zu integrierendem Teilproduktmodell ersetzen. Abbildung 9 zeigt diesen Ansatz grafisch. Die Implementierung der jeweils speziellsten *verschieben*-Methode setzt sich in diesem Fall immer aus Aufrufen der beiden direkten Oberklassen-Implementierungen zusammen (Abbildung 10).

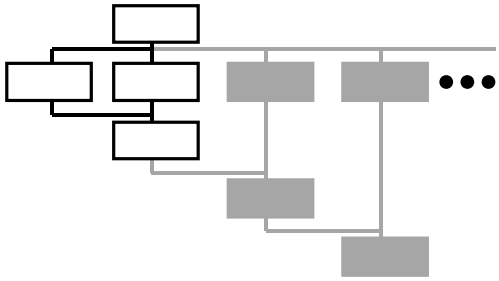

Abbildung 9: Erweiterbarkeit bei multipler Vererbung II

Diese Modellierung macht auf den ersten Blick einen recht flexiblen Eindruck und genügt scheinbar dem open-closed-principle, weil einmal fertiggestellte Klassen nicht mehr verändert werden müssen. Im Zusammenhang mit persistenten Objekten entsteht jedoch erneut das Problem der notwendigen physischen Objektmigration, wenn auch diesmal nicht in der Form, dass sich das physische Layout einer Klasse bzw. ihrer Instanzen ändert, sondern dass alle Instanzen der bisherigen speziellsten Klasse nun als Instanzen der neuen speziellsten Klasse neu angelegt werden

müssen. Darüber hinaus entsteht im Allgemeinen das Problem, dass an allen Stellen, an denen bisher Bauteile der Klasse *BauteilTragwerkStatik* erzeugt wurden, in Zukunft Objekte der Klasse *Bauteil_T_S_N* erzeugt werden müssen. Diese Problematik wird im folgenden Abschnitt diskutiert.

```
class Bauteil_T_S_N :   public BauteilTragwerkStatik,
            public BauteilNumerischesModell {
  ...
  virtual void verschieben(double x, double y, double z);
  ...
};

void
Bauteil_T_S_N::verschieben(double x, double y, double z)
{
  BauteilTragwerkStatik::verschieben(x, y, z);
  BauteilNumerischesModell::verschieben(x, y, z);
}
```

Abbildung 10

2.2.2 Objektinstanziierung

Die Erzeugung von Objekten ist in der objektorientierten Programmierung häufig ein Punkt, der über die Flexibilität und Erweiterbarkeit des Systems, in dem diese Objekte instanziiert werden, mitentscheidet. Durch das dynamische Binden werden prinzipiell an allen Stellen, an denen eine Methode aufgerufen wird, Einstiegspunkte für zukünftige Unterklassenbildung gebildet. Das dynamische Binden richtet sich in traditionellen objektorientierten Sprachen jedoch immer ausschließlich nach dem Typ des Empfängerobjekts, der zur Zeit seiner Erzeugung festgelegt wurde. Damit werden Codefragmente der Art

```
Gebaeude* g = ...;
Bauteil* b = new BauteilTragwerkStatik;
g->fuegeEin(b);
g->zeichnen();
```

sehr anfällig gegen Erweiterungen des Systems – im Falle der Hinzunahme der Klasse *Bauteil_T_S_N* werden alle derartigen Codestellen zu ändern sein.

Mit diesen Problemen setzten sich die Design Pattern für die Objekterzeugung auseinander [GHJV94]. Ihr Ziel ist es, die Instanziierung konkreter Objekte durch Unterklassenbildung ebenso erweiterbar zu halten wie andere Methodenaufrufe auch. Das in diesem Zusammenhang bekannteste und am häufigsten verwendete Design Pattern ist die *Abstract Factory*, deren Einsatz am Beispiel der Bauteilklassen in Abbildung 11 skizziert ist.

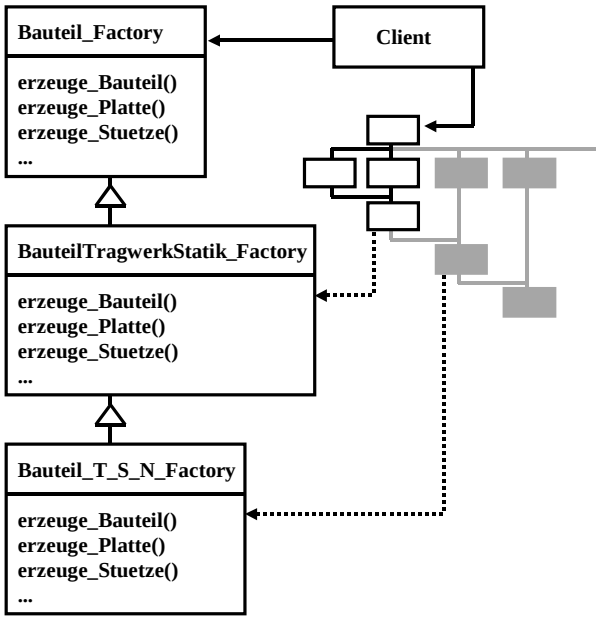

Abbildung 11: Abstract Factory Pattern

Die diesem Muster zu Grunde liegende Idee verlegt die tatsächliche Erzeugung eines konkreten Objekts in eine dynamisch gebundene Methode einer abstrakten Klasse, in Abbildung 11 zum Beispiel die Methoden *erzeuge_Bauteil*(), *erzeuge_Platte*(), *erzeuge_Stuetze*() etc. Das oben dargestellte Codefragment für die Erzeugung von Bauteilen würde dementsprechend so formuliert:

```
Gebaeude* g = ...;
Bauteil* b = the_global_factory->erzeuge_Bauteil();
g->fuege_ein(b);
g->zeichnen();
```

Bei konsequenter Nutzung des Abstract Factory Patterns werden sämtliche Instanziierungen von Bauteilen durch Factories realisiert. Durch Erzeugung der geeigneten Factory besteht nun die Möglichkeit, durch Änderung einer einzigen Anweisung die Generierung von Bauteilen systemweit auf die gewünschte neue Klasse umzustellen.

```
Bauteil_Factory* the_global_factory =
  new Bauteil_T_S_N_Factory;
```

Die hier geschilderte Vorgehensweise erlaubt, durch geschickte Ausnutzung des dynamischen Bindens ein System so flexibel zu beschreiben, dass sich eine gewünschte Änderung des Systemverhaltens durch Hinzunahme neuer Klassen und fast völlig ohne Änderung bestehenden Codes formulieren lässt. Allerdings bleibt das Problem der notwendigen physischen Migration von persistenten Objekten. Gravierender ist jedoch die Frage nach den angemessenen Methodenimplementierungen in der speziellsten Klasse *Bauteil_T_S_N*. Der folgende Abschnitt beschäftigt sich ausführlich mit diesem Aspekt.

2.2.3 Method Dispatch

Wie in Abschnitt 2.2 bereits erwähnt liefert der Aufruf dynamisch gebundenen Methoden bei den Mehrfachvererbungs-Ansätzen immer die Implementierung der speziellsten, die Eigenschaften aller Teilproduktmodelle umfassenden Klasse. Funktionen, die eine anwendungsspezifische Realisierung verlangen (z.B. *zeichnen*()) können in der Klasse *BauteilTragwerkStatik* oder gar in der Klasse *Bauteil_T_S_N* nicht sinnvoll implementiert werden – sowohl Aufrufe in der Tragwerkmodellierung als auch in der Statik müssen aber in diesem Szenario letztlich diese Implementierung ausführen. Ein Ausweg wäre die Einführung unterschiedlicher Methodennamen wie *zeichnen_Tragwerk*(), *zeichnen_Statik*() etc. Eine solche „Methodenexplosion“ schließt jedoch die Möglichkeit aus, Algorithmen schon auf sehr abstrakter Ebene und damit wiederverwendbar zu spezifizieren, da schon zur Übersetzungszeit eine Auswahl zwischen den teilmodellspezifischen Methodennamen getroffen werden muss.

2.2.4 Probleme bei wiederholter Mehrfachvererbung

```
class Base {
  virtual void do_something() {
    i++;
  }
  int i;
};

class Derived1 : public Base {
  virtual void do_something() {
    Base::do_something();
    j++;
  }
  int j;
};
```

```
class Derived2 : public Base {
  virtual void do_something() {
    Base::do_something();
    k++;
  }
  int k;
};
class Derived3 : public Derived1,
                 public Derived2 {
  virtual void do_something() {
    Derived1::do_something();
    Derived2:do_something();
    l++;
  }
  int l;
};
```

Abbildung 12

Wenn bei multipler Vererbung eine Klasse von einer Wurzelklasse über mehrere Ableitungspfade erreichbar ist, spricht man von wiederholter Mehrfachvererbung (*repeated inheritance*). Derartige Designs gelten oft als problematisch, wenn die Wurzelklasse über Attribute verfügt. Der Grund liegt üblicherweise in update-Methoden, die bei jeder neuen Unterklasse durch Aufruf der Oberklassen-Implementierungen und zusätzlicher, für die neue Unterklasse spezifische Anweisungen realisiert werden. Wie man in Abbildung 12 sehen kann, führt dieses Vorgehen bei der Implementierung der Methode *Derived3::do_something*() zu dem unerwünschten doppelten Aufruf von *Base::do_something*(). Dieser Effekt ließe sich nur umgehen, wenn die Implementierung in der abgeleiteten Klasse *Derived3* auf die interna der Oberklassen zugreift und die Implementierungen der zu redefinierenden Methoden selbst erneut vornimmt. Die Art der Codeduplizierung hat aber zur Folge, dass die Unterklasse *Derived3* sehr anfällig gegenüber Änderungen in einer der Oberklassen wird. Zur Vermeidung eines solchen Effekts haben Sprachen wie C++ und Java das Sichtbarkeitskonzept *private* eingeführt, welches abgeleiteten Klassen den Zugriff auf private Teile der Oberklasse unmöglich macht. Darüber hinaus erlaubt Java mit seinem *Interfaces* genannten Konzept wiederholte Mehrfachvererbung nur für Typen, die selbst über keinerlei Attribute oder Methodenimplementierungen verfügen. Interessanterweise bietet Eiffel [Meye98], eine Sprache die aus verschiedenen Gründen weit mehr auf Mehrfachvererbung baut als C++ und Java, konsequenterweise keine Möglichkeit, abgeleiteten Klassen den Zugriff auf die Implementierung der Oberklassen einzuschränken.

In Abschnitt 3 schlagen wir ein Sprachkonzept vor, welches die unerwünschten Mehrfachaufrufe von Basisklassen-Implementierungen vermeidet, ohne die Interna von Oberklassen offenlegen zu müssen.

2.3 Design Patterns

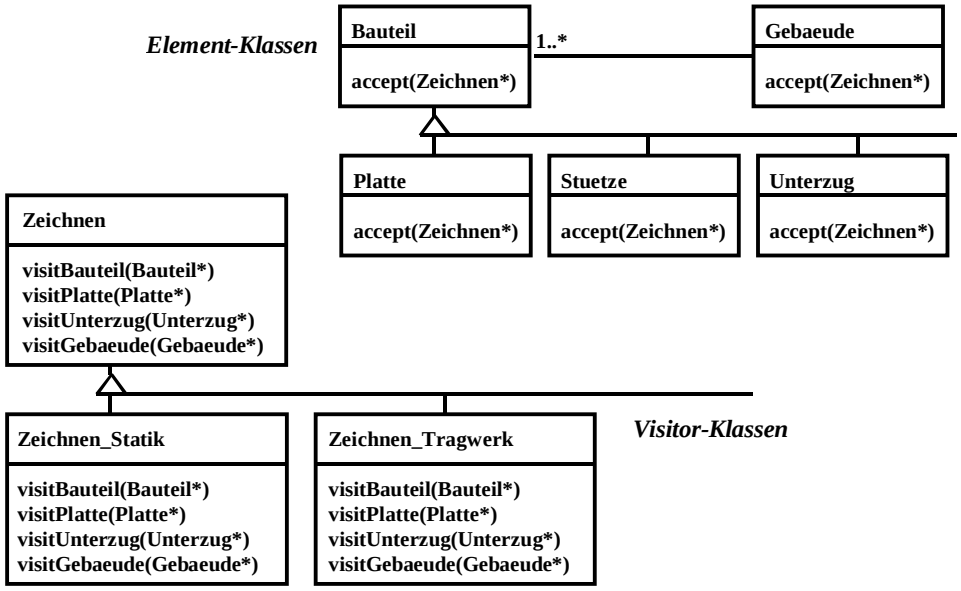

Abbildung 13: Visitor Pattern

Eine starke Trennung zwischen der Präsentation und dem eigentlichen Objektmodell für die Modellierung der Daten und ihrer Funktionalität wird von [KrPo88] unter dem Stichwort *Model/View/Controller* Architektur vorgeschlagen. Dieses Konzept erlaubt, weitere Formen der Präsentation (*Views* genannt, nicht zu verwechseln mit den Views bzw. virtuellen Relationen aus der Datenbankwelt) nachträglich hinzuzufügen. Die Eignung von MVC liegt jedoch hauptsächlich bei Benutzer-Schnittstellen, weniger bei Erweiterung oder gar Anpassung der Kernfunktionalität des *Models* selbst. Einen anderen Weg beschreibt das *Visitor Pattern* [GHJV94, PaJa98], das ursprünglich dazu konzipiert wurde, bestehende Klassen um ganz neue Operationen zu erweitern ohne die Klassen selbst ändern zu müssen. Ein häufig in diesem Zusammenhang genanntes Beispiel sind Knotenklassen in einem abstrakten Syntaxbaum, der innerhalb einer Compiler-Implementierung verwendet wird. Auf diesen Knoten, die z.B. Ausdrücke in einer Programmiersprache repräsentieren, gibt es eine Vielzahl von möglichen Funktionen, die potentiell während eines Compilerlaufs auszuführen sind: Syntaxprüfung, Semantikprüfung, Typecheck, Codegenerierung, Debugging-Informationsgenerierung, Optimierung etc. Das Visitor Pattern verteilt diese Operationen in unterschiedliche, „Visitoren“ genannte Klassen, die eine eigene Hierarchie bilden. Demgegenüber stehen die sogenannten „Elementklassen“, welche die eigentlichen Daten beinhalten. In Abbildung 13 sind die Visitoren durch die drei *Zeichnen*-Klassen, die Elemente durch die *Bauteil*-Klassen realisiert.

```
void Gebaeude::accept(const Zeichnen* z) const {
  bauteile->apply<b>(b->accept(z));
}
void Bauteil::accept(const Zeichnen* z) const {
  z->visitBauteil(this);
}
void Platte::accept(const Zeichnen* z) const {
  z->visitPlatte(this);
}
void Zeichnen_Tragwerk::visitPlatte(const Platte* b) const {
  // hier geschieht die eigentliche, tragwerkspezifische Zeichnung
}
Gebaeude* g = ...;
Zeichnen* z = new Zeichnen_Tragwerk;
g->accept(z);
```

Abbildung 14: `accept()` und `visit()` Implementierung beim Visitor Pattern

Der „Trick“ des Visitor Patterns liegt in einer Technik, die sich *double dispatch* nennt. Damit ist ein zweimaliges dynamisches Binden gemeint, welches beim ersten dispatch nach dem konkreten Elementtyp auflöst und im zweiten dispatch nach dem aktuellen Visitor. In Abbildung 14 können wir den Ablauf leicht verfolgen: *Gebaeude::accept* iteriert über alle Bauteil und ruft dort *accept* mit dem übergebenen Parameter *z* auf. Das dynamische Binden bei diesem accept-Aufruf sucht die konkrete Klasse des Bauteils – wir gehen in diesem Beispiel davon aus, dass sich hinter einem der Elemente des Attributs *bauteile* ein Objekt vom Typ *Platte* befindet. Dadurch ist der Typ des Bauteils aufgelöst. Im Rumpf der Methode *Platte::accept* wird nun mit dem übergebenen Visitor als Reciever-Objekt aufgerufen, dieses ist vom konkreten Typ *Zeichnen_Tragwerk*. Da die Visitoren eine *visit*-Funktion für jeden Typ aus der Elementklassenhierarchie besitzen, bleibt das Wissen über das konkrete Bauteil erhalten. Beim Eintritt in die Methode *Zeichnen_Tragwerk::visitPlatte* sind also alle Informationen vorhanden um die spezifische Aktion für diese Kombination Visitortyp/Elementtyp durchzuführen.

Die Anwendung des Visitor Pattern erlaubt uns zwar, nachträglich neue Funktionalität für eine bereits existierende Klassenhierarchie zu integrieren, sie bietet uns jedoch keine Möglichkeit, die Elementklassen auch physisch um neue Eigenschaften, d.h. zu speichernde Attribute, zu erweitern. Die Elementklassen in Abbildung 13 sind deshalb nicht korrekt dargestellt, da sie die teilproduktmodellspezifischen Daten weglassen. Um diese Daten zu modellieren, kann die Mehrfachvererbung aus Abbildung 9 verwendet werden. Dieses Design opfert jedoch die Typsicherheit des ursprünglichen Visitor Patterns, da z.B. der Rumpf der Methode *Zeichnen_Tragwerk::visitBauteil* mit einer expliziten Typumwandlung versehen werden müßte:

```
void Zeichnen_Tragwerk::visitBauteil(const Bauteil* b) const {
  static_cast<const BauteilTragwerk*>(b)->solid->zeichnen();
}
```

Ein weitere Nachteil des Visitor Patterns ist die strikte Trennung von Daten und Funktionen – eine Eigenschaft, die eigentlich völlig kontrovers zur Intention der Objektorientierung ist. Vielleicht noch nachteiliger ist die hohe Anzahl von Klassen, die durch den Einsatz der Visitoren entsteht, da Visitoren immer nur eine einzige Methode ersetzen können und keine Möglichkeit bieten, mehrere semantisch gruppierte Funktionen in einer Klasse zu bündeln. So können in unserem Beispiel keine Visitoren je Teilproduktmodell geschrieben werden, sondern es muss jeweils eine Visitorklasse pro Methode und pro Teilproduktmodell entwickelt werden. Da schon die Elementklassenhierarchie wegen der eventuell zu verwendenden Mehrfachvererbung recht komplex ist, wird die Gesamtzahl der entstehenden Klassen unangenehm groß und erschwert deswegen entscheidend die Übersichtlichkeit und Verständlichkeit des Designs.

2.4 Objektorientierte Sichtkonzepte

Einen anderen Weg versuchen Ansätze, die sich mit der Übertragung der aus relationalen Datenbank Management Systemen bekannten Sichten auf Objektorientierte Datenbanken beschäftigen. Diese Idee erscheint zunächst sehr naheliegend, weil in Analogie zu traditionellen Sichten neue, virtuelle Klassen durch Anfragen sehr einfach definiert werden können. Diese virtuellen Klassen beschreiben dann – ebenso wie die virtuellen Relationen – die auf anwendungsspezifische Bedürfnisse zugeschnittenen Objekte einer Datenbank.

Objektorientierte Sichtkonzepte lassen sich grob in zwei Richtungen und einige Mischformen dieser Richtungen unterscheiden, die in den folgenden Unterabschnitten diskutiert werden.

2.4.1 Objektgenerierende Sichten

Vorschläge für sogenannte objektgenerierende Sichten [AbBo91, Bert92, ByMc93, Rund92] gehen von der Idee aus, dass eine virtuelle Klasse mit neu erzeugten Objekten generiert wird, die Ergebnismenge der die Sicht definierenden Anfrage sind. Natürlich sind strukturverändernde Operationen wie die aus der relationalen Welt bekannte Projektion, aber auch Kreuzprodukt möglich. Gekoppelt mit Selektion ergeben sich sehr flexible und mächtige Möglichkeiten, Daten bzw. Objekte für Klienten auf unterschiedlichste Weise aufzubereiten bzw. zu präsentieren. Auch im Bereich der deduktiven objektorientierten Datenbank Management Systeme (DOOD)

sind derartige Ansätze verbreitet, wobei die Mächtigkeit zur Beschreibung virtueller Klassen bzw. deren Instanzen hier weit über lediglich restrukturierende Operationen hinausgeht [ACMT93, JaLa92, JGJS94, KLW95, Liu96]. Den hier zitierten Ansätzen ist jedoch gemein, dass sie von Paradigmen bzw. den Konzepten imperativer Sprachen sehr weit entfernt sind.

Problematisch sind diese Ansätze für objektgenerierende Sichten für die Anwendung im hier beschrieben Kontext in mancherlei Hinsicht. Für Anwendungen, die auf virtuelle Objekte auch schreibend zugreifen müssen entsteht das Problem, dass die Propagierung von Änderungen an Sicht-Objekten auf die physischen Originalobjekte im allgemeinen nicht automatisch erfolgen kann[3]. Aber auch bei nur lesenden Applikationen entstehen Schwierigkeiten, wenn nicht von einem ausschließlich mengenorientierten Zugriff auf Instanzen ausgegangen wird. In objektorientierten Programmiersprachen – und damit in allen Anwendungen auf objektorientierten Datenbanken, die mehr als eine triviale Komplexität aufweisen – ist jedoch die Dereferenzierung von Verweisen die übliche Methode, um Objektbeziehungen zu verfolgen. Anfragen, um in Beziehung stehende Objekte über Werte bzw. Schlüssel zu ermitteln, bilden schon wegen der damit verbundenen Performanzprobleme eine Ausnahme. Objekte, die als Ergebnis einer sichtdefinierenden Anfrage temporär erzeugt werden, können jedoch nicht das Ziel von Verweisen in anderen Objekten bilden, da diese Objekte nicht dauerhaft existieren. Um diese enorme Einschränkung zu umgehen, wurden diverse Verfahren vorgeschlagen [Bell00, BLT86, CeWi91, GMS93]. Verfahren, die dafür Sorge tragen, dass derartige Objekte aus virtuellen Klassen jeweils immer mit dem gleichen Object Identifier erzeugt werden, heben die Beschränkung, dass Objekte virtueller Klassen nicht Ziele von Referenzen sein können, auf.

[Bell00] schlägt z. B. vor, virtuelle Objekte nicht bei jedem Zugriff neu zu erzeugen, sondern diese physisch zu speichern („Materialisierung“) und ihnen dabei eindeutige OIDs zuzuweisen. Diese OIDs ermöglichen dem System eine 1:1 Abbildung der Basis-Objekte zu den von ihnen hergeleiteten virtuellen Objekten. Die Materialisierung der virtuellen Objekte wirkt sich neben der Unterstützung von Updates natürlich auch auf die Performanz aus, da die Häufigkeit der Erzeugung von Objekten und die Zuordnung von OIDs drastisch reduziert wird.

[3] Dieses Problem ist analog zu relationalen Sichten, bei denen Änderbarkeit auf virtuellen Relationen nur dann möglich ist, wenn ein Tupel einer Sicht die Primärschlüssel aller in ihm auftretenden Tupel aus den physischen Relationen enthält.

Im Rahmen der am Anfang dieses Kapitels aufgeworfenen Problemstellung können objektgenerierende Sichten als eine Art automatisierte Lösung des Ansatzes redundanter Objekte aus Abschnitt 2.1 gesehen werden. Damit bleibt auch das Problem der Redundanz (und des damit zumindest temporär notwendigen höheren Datenvolumens) bestehen. Um die Originalobjekte mit ihren generierten virtuellen Objekten konsistent zu halten sind komplizierte Verfahren zur Änderungspropagierung notwendig. Wie wir gesehen haben, sind solche Verfahren im Allgemeinen nicht vom Übersetzer herleitbar und müssen deshalb vom Entwickler spezifiziert werden. Noch ungünstiger ist jedoch die Tatsache, dass diese redundante Darstellung es nicht erlaubt, hinter einem einzigen Verweis auf ein Objekt je nach Anwendungskontext das gewünschte Verhalten zu binden. Denn für Anwendungen, die eine spezielle Sicht benötigen, müssen auch die Verweise von anderen Objekten auf diese generierten virtuellen Objekte zeigen. Bei komplexen Aggregationshierarchien explodieren damit automatisch die verschiedenen Darstellungen, obwohl sie unter Umständen bis auf die unterschiedliche Sicht in ihren Verweisen keinerlei Unterschiede aufweisen.

2.4.2 Objekterhaltende Sichten

Um viele der oben genannten Probleme zu umgehen, schlugen bereits 1991 eine Schweizer Gruppe [SLT91] ein objektorientiertes Sichtkonzept vor, in dem Objekte aus einer virtuellen Klasse nicht als Ergebnis einer Anfrage generiert werden, sondern eine physisch nur einmal vorhandene Instanz unter verschiedenen Typen präsentiert wird. Dieser Ansatz machte den Begriff der „*multiplen Instanziierung*“ populär, denn Instanzen einer (nicht-virtuellen) Klasse sind automatisch und ohne auszuführende Konvertierung auch Instanzen aller auf dieser Klasse definierten Sichten. Dabei ändert sich insbesondere der Object Identifier nicht. Dieses Vorgehen vermeidet ganz offensichtlich eine Vielzahl der Nachteile, die in Abschnitt 2.4.1 beschrieben wurden. Andererseits sind die Typ- bzw. strukturverändernden Operationen bei objekterhaltenden Sichten weniger mächtig als die für objektgenerierende: Kreuzproduktbildung und damit auch Joins sind auf diese Weise nicht zu realisieren, da die Kreuzproduktbildung selbstverständlich neue Tupel bzw. Objekte erzeugen muss. Dennoch folgten auf den Vorschlag von [SLT91] mit [BaKe93, CBR94, EdDo94, KRR95, Schi93] einige andere Forschungsgruppen diesem Weg, weil durch den objekterhaltenden Ansatz einerseits Änderungen auf Objekten virtueller Klassen sehr leicht möglich werden und andererseits Redundanz vermieden wird. Wenig untersucht wurde jedoch bei diesen Ansätzen das Problem des Methoden-Dispatch. Multiple Instanziierung hat automatisch zur Folge, dass

klassisches dynamisches Binden, wie es in allen traditionellen objektorientierten Sprachen benutzt wird, nicht mehr ausreichen kann. Diese Aspekte werden in Abschnitt 3.2.3 diskutiert.

Auch in diesem Bereich objektorientierter Sichtkonzepte stammen eine Reihe von Vorschlägen aus dem DOOD Umfeld. Insbesondere im Chimera Projekt [BeGu95, BGR99, CeMa94, GBCM97] wurde multiple Instanziierung als eine Möglichkeit der Generierung von Sichten vorgeschlagen. Zusätzlich zur gleichzeitigen, direkten Zugehörigkeit von Objekten zu mehreren Klassen beinhaltet dieser Vorschlag aber auch diverse Konzepte zur Migration von Objekten in andere Klassen. In unserem Beispiel würde Objektmigration es erlauben, ein *Bauteil* zu einem beliebigen Zeitpunkt in die Klasse *BauteilStatik* zu migrieren und damit auch implizit das Verhalten dieser Klasse für das fragliche Objekt zu aktivieren. Umgekehrt ist es auch möglich, ein Objekt zu generalisieren, d.h. die Instanz in eine Oberklasse zu migrieren. Diese Variante der Migration ist mit dem physischen Verlust der Attribute der spezielleren Klasse verbunden und kommt deshalb für unser Beispiel nicht in Frage. Außerdem sind mit derartigen Migrationen Typsicherheitsprobleme verbunden, da im Allgemeinen in anderen Objekten getypte Verweise auf das Objekt mit dem gerade verlorengegangenen Typ enthalten sein könnten. Diese Verweise werden mit der Migration ungültig oder fehlerhaft[4].

Migrationen bzw. Objektevolution, wie von Chimera in [BGR99] vorgeschlagen, zielen eigentlich auf zwei andere Anwendungsfälle: 1) die Unterstützung von Schemaevolution bzw. die Anpassung von Instanzen an durch Schemaevolution veränderte Klassen und 2) die Modellierung von – möglicherweise disjunkten –

[4] In Chimera wird dieses Problem umgangen, indem alle auf ein solches Objekt unter dem bisherigen Typ existierenden Verweis auf *nil* gesetzt werden. Diese Semantik kann immer noch zu überraschenden Effekten führen, wenn ein Verweis auf ein Objekt plötzlich indirekt „von außen“ entfernt wird. Außerdem stellt sich die Frage nach der effizienten Implementierbarkeit: Entweder sind alle Objektreferenzen über eine Indirektion realisiert, oder jegliche Form von Verweisen ist bidirektional zu speichern, oder die Suche nach auf ein Objekt verweisenden Referenzen kommt einem Scan eines Garbage Collectors gleich. Alle drei Ansätze sind für eine Vielzahl von Anwendungen durchaus geeignet, verbieten sich jedoch als allgemeine Lösung für eine imperative Programmiersprache.

Rollenverhalten, d.h. die dynamische Änderung des Typs eines Objektes im Hinblick auf seine Struktur (Menge der Attribute) und sein Verhalten (Methodenimplementierungen). In unserem Kontext wird jedoch die Darstellung verschiedener Aspekte eines Objektes, die statisch über seine gesamte Lebensdauer hinweg existieren, gefordert. Die ebenfalls von Chimera unterstützte multiple Instanziierung ist deshalb das passendere Konzept. Zur Umsetzung von Rollenkonzepten in objektorientierten Programmiersprachen vgl. auch [Knie96, Knie00].

2.5 Zusammenfassung

Die vorigen Abschnitte diskutierten ausführlich ein Problem, das auf den ersten Eindruck sehr einfach zu sein scheint: Die Modellierung anwendungsspezifischer Aspekte von Klassen bzw. ihrer persistenten Instanzen als Ableitungen eben dieser Klassen. Es zeigte sich jedoch schnell, dass naive Ansätze wie redundante Objekte (vgl. 2.1) oder multiple Vererbung (vgl. 2.2) nicht geeignet sind, um ein flexibles und für Erweiterungen offenes System zu entwerfen.

Vielversprechender scheint dagegen der Einsatz von Design Patterns zu sein (vgl. 2.3). Um jedoch die gewünschte Funktionalität mehr oder weniger flexibel zu realisieren, müssen mit dem Abstract Factory Pattern und dem Visitor Pattern[5] sehr viele zusätzliche Klassen eingeführt werden, die das zu realisierende System erheblich verkomplizieren. Ein weiterer Nachteil ist, dass alle diese Entscheidungen a priori, d.h. schon vor Eintreten einer ersten Anforderung nach Erweiterung, getroffen werden müssen um überhaupt später auf diese Anforderungen reagieren zu können. Vielleicht am gravierendsten ist aber die Tatsache, dass die Hinzufügung neuer Attribute – egal über welchen der hier diskutierten Ansätze – in allen Fällen eine physische Migration der persistenten Objekte zur Folge hat. Dies ist in den meisten ODBMS eine sehr teure Operation, die oft mit einer temporären Totalsperrung der betroffenen Datenbanken verbunden ist.

Nach Analyse all dieser scheinbar so großen Schwierigkeiten betrachten wir noch einmal die ursprüngliche Abbildung 1, welche der gewünschten Lösung eigentlich schon sehr nahe kommt: Es sind lediglich drei Klassen für die Beschreibung der Bauteile vorhanden, die relevanten Methoden sind sinnvollerweise in den

---

[5] Bei genauer Betrachtung stellen wir fest, dass zusätzlich das *Iterator Pattern* sowie *Factory Methods* [GHJV94] eingesetzt werden müssen, um eine flexible Iteration über die Menge der Bauteile in einem Gebäude zu erlauben - vgl. Abschnitt 3.

spezifischen Klassen redefiniert, wir haben weder Mehrfachvererbung, noch eine Trennung von Daten und Funktionen, noch müssen komplizierte Design Patterns für die Objektinstanziierung von vornherein integriert werden. Das einzige Problem ist, dass dieses Design in traditionellen objektorientierten Programmiersprachen zwingend ausschließt, dass eine Instanz der Klasse *Bauteil* auch gleichzeitig eine direkte Instanz der Klassen *BauteilTragwerk* und *BauteuilStatik* ist. Der folgende Abschnitt stellt ein Sprachkonzept vor, das diese Einschränkung aufhebt und damit einen äußerst flexiblen, einfach zu nutzenden Mechanismus für die Beschreibung anwendungsspezifischer Ausprägungen von Objekten zur Verfügung stellt.

3 Perspektiven

Diese Kapitel stellt unser eigenes Konzept für die Unterstützung anwendungsspezifischer Eigenschaften und Verhaltensweisen von Objekten vor. Die Grundidee dabei ist die *multiple Instanziierung*, d. h. Objekte sind direkte Instanz von mehr als einer Klasse. Multiple Instanziierung wurde in der Literatur schon mehrfach vorgeschlagen [BeGu95, Schi93, SLT91], die Schwachstellen dieser Ansätze wurden jedoch schon in Abschnitt 2.4 diskutiert. Wir haben unseren Ansatz *Perspektiven* genannt, weil die Ermöglichung verschiedener, zueinander jedoch konsistenter Blickwinkel auf Objekte bzw. ganze Klassen im Vordergrund stand. Im Unterschied zu den oben erwähnten Sichtkonzepten, die überwiegend aus der mengenorientierten Datenbankwelt stammen, handelt es sich bei unserem Ansatz um ein programmiersprachlich orientiertes Vorgehen. Deshalb betrachten wir nicht Objektmengen, sondern Klassen bzw. Typen, wobei wir zwischen den Begriffen in dieser Arbeit keinen Unterschied machen. Wir benutzen die Begriffe Klasse und Typ synonym als Beschreibung von Eigenschaften von Objekten, d.h. deren Schnittstellen, Attribute und Implementierung. Insbesondere verwenden wir den Begriff Klasse nicht als eine Menge von Instanzen. Diese klare Abgrenzung hat unter anderem auch den Hintergrund, dass Perspektiven im Gegensatz zu Sichten kein dynamisches Konzept sind – die Zugehörigkeit eines Objekts zu der einen oder anderen Perspektive wird folglich nicht über die Auswertung von Prädikaten zur Laufzeit entschieden, sondern durch Typ-Beschreibungen zur Übersetzungszeit.

Der Rest dieses Kapitels gliedert sich in eine Beschreibung der Grundlagen des Datenmodells in den Abschnitten 3.1.1 und 3.1.2, eine ausführliche Diskussion des dynamischen Bindens von multipel instanziierten Objekten im Abschnitt 3.2; eine Betrachtung von Mechanismen für konsistente Änderungen über Perspektiven hinweg findet sich in Abschnitt 3.3.

3.1 Datenmodell

Multiple Instanziierung, d.h. die Möglichkeit, Objekte als direkte Instanzen mehrerer Klassen zu erzeugen, hat einige interessante Auswirkungen in Bezug auf die Einordnung von Perspektiven im Vererbungsgraphen sowie der Ersetzbarkeit von getypten Variablen. Dieser Abschnitt gibt einige grundlegende Definitionen, die diese Effekte präzise beschreiben sollen.

3.1.1 Grundlagen

Definition 3.1: (*Klasse*)

Sei $SimpleTypes = \{integer, float, character, string\}$ die Menge der einfachen (Wert-) Typen und C die Menge der (benutzerdefinierten) *Klassen*. Eine Klasse $c \in C$ definiert eine Menge von *Attributen* $c.A = \{c.a_1, \ldots c.a_n\}$ und eine Menge von *Methoden* $c.F = \{c.f_1, \ldots, c.f_m\}$, mit $n, m \in N$. Jedes Attribut $c.a$ hat einen Typ aus $C \cup SimpleTypes$ der entweder einen Wert eines einfachen Typen oder eine Referenz auf einen Klasse beschreibt[6]. Analog hat jede Methode $c.f$ eine Signatur, die Argumenttypen und den (möglicherweise leeren) Rückgabetyp beschreibt. ❑

Definition 3.2: (*Oberklasse*, *Unterklasse*)

Sei C eine Menge von Klassen nach Definition 3.1. Eine Funktion $super{:}C \to 2^C$ definiert die Menge von *direkten Oberklassen* einer Klasse c wenn für alle $c \in C$

- $c \notin super(c)$ (1)
- $\forall c' \in super(c): c \notin super(c')$ (2)
- $\forall c', c'' \in super(c): c' \notin super(c'') \wedge c'' \notin super(c')$ (3)
- $\neg \exists c_1, \ldots, c_n \in C, n \geq 0:$ $c_1 \in super(c) \wedge c_2 \in super(c_1) \wedge \ldots \wedge c_n \in super(c_{n-1}) \wedge c \in super(c_n)$ (4)
- $\forall c' \in super(c): c.A \supseteq c'.A \wedge c.F \supseteq c'.F$ (5)

gilt. Die Funktion $sub{:}C \to 2^C$ definiert durch $sub(c) = \{c' \in C \mid c \in super(c')\}$ definiert die Menge der *direkten Unterklassen* einer Klasse c. ❑

Definition 3.2 definiert *super* als nicht reflexiv (1), antisymmetrisch (2), nicht transitiv (3) und azyklisch (4). Darüber hinaus erbt eine Klasse sowohl Attribute als auch Methoden von ihren Oberklassen (5). Zunächst gehen wir davon aus, dass geerbte Attribute und Methoden die gleiche Signatur besitzen wie ihre Oberklassen-Gegenstücke ("no-variance"). Abbildung 15 zeigt eine Menge von Klassen und ihren Ober-/Unterklassenbeziehungen in OMT-Notation [RBPE93].

Im Folgenden benutzen wir die Notation "–" für die reflexive und transitive Hülle, d.h. für eine Menge S und eine Funktion $f: S \to 2^S$ ist $\overline{f}: S \to 2^S$ definiert durch

$$\overline{f}(s) = \{s' \in S \mid \exists s_1, \ldots, s_n \in S, n > 0: s_1 \in f(s) \wedge s_2 \in f(s_1) \wedge \ldots s_n \in f(s_{n-1}) \wedge s' \in f(s_n)\} \cup \{s\}.$$

---

[6] Mengenwertige Attribute werden hier nicht betrachtet, können aber leicht in das Modell integriert werden.

Analog dazu ist für zwei Funktionen f_1 und f_2 mit gleichem Vor- und Nachbereich wie oben $\overline{f_1 \circ f_2} : S \times S \to 2^S$ definiert durch

$$\overline{f_1(s) \circ f_2(s')} = \left\{ s'' \in S \mid \exists s''' \in S : s''' \in f_1(s) \circ f_2(s') \wedge s'' \in \overline{f'(s''')} \right\} \cup \{s, s'\}$$

wobei $f'(s) = f_1(s) \circ f_2(s)$ und $\circ$ aus $\{\cup, \cap, \setminus\}$. Beispielsweise beschreibt $\overline{super(c)}$ die Menge der direkten und indirekten Oberklassen, $\overline{sub(c)}$ die Menge der direkten und indirekten Unterklassen und $\overline{super(c) \cup sub(c)}$ den Zusammenhangsgraph in der zu c gehörigen Klassenhierarchie.

Eine der wichtigsten Eigenschaften von streng getypten objektorientierten Datenmodellen ist die durch $\overline{super}$ definierte Ersetzbarkeit von Referenz-Variablen. Eine Variable vom Typ c kann genau dann auf ein Objekt vom Typ c' verweisen, wenn $c' \in \overline{super(c)}$. Als Konsequenz daraus ist der genaue Typ eines durch eine Variable referierten Objekts i.A. zur Compile-Zeit unbekannt.

Definition 3.3: (*statischer und dynamischer Typ*)

Seien C und $super$ gemäß den vorangegangenen Definitionen gegeben und $V = \{v_1, \ldots, v_n\}$ bezeichne die Menge der Referenz Variablen. Eine Funktion $stype : V \to C$ definiert den *statischen Typ* einer Variablen $v \in V$. Eine Funktion $dtype : V \to C$ definiert den dynamischen Typ einer Variablen $v \in V$ genau dann, wenn $\forall v \in V : stype(v) \in \overline{super(dtype(v))}$ gilt. ❑

Unsere Definition von $dtype$ berücksichtigt nicht den Datenbankzustand, da eine Variable über verschiedene Zustände hinweg auf unterschiedliche Objekte verweisen kann. Der Einfachheit halber haben wir jedoch an dieser Stelle Zeit aus unseren Definitionen herausgelassen, da wir Veränderungen in $dtype(v)$ nicht explizit betrachten. Die meisten streng getypten objektorientierten Programmiersprachen erzwingen die Forderung aus Definition 3.3 dadurch, dass Zuweisungen einer Referenz Variablen v an eine Referenz Variable v' genau dann akzeptiert werden, wenn $stype(v') \in \overline{super(stype(v))}$. In konventionellen objektorientierten Typsystemen sind Objekte direkte Instanzen genau der Klasse, deren Konstruktor sie instanziiert hat. $dtype(v)$ bezeichnet immer diesen Typ für das von v referenzierte Objekt.

3.1.2 Polymorphe Klassen

Um die Menge der Klassen, von denen ein Objekt direkte Instanz sein kann zu definieren, führen wir den Begriff der polymorphen (Klassen-) Ableitung ein. Informell bedeutet die polymorphe Ableitung einer Klasse, dass alle direkten Instanzen der Originalklasse auch direkte Instanzen der abgeleiteten Klasse und

umgekehrt sind. Polymorph abgeleitete Klassen nennen wir *Perspektiven* ihrer Basisklasse.

Definition 3.4: (*polymorphe Ober-/Unterklassen*)

Seien C und $super$ gemäß den vorangegangenen Definitionen gegeben. Eine Funktion $\sup er_p : C \to 2^C$ beschreibt die Menge der (direkten) *polymorphen Oberklassen* einer Klasse c genau dann, wenn für alle $c \in C$

- $super_p(c) \subseteq super(c)$ (1)
- $\forall c', c'' \in super_p(c) : \exists c''' \in C : c''' \in \overline{super_p(c') \cap super_p(c'')}$ (2)

gilt. Die Funktion $sub_p : C \to 2^C$ definiert durch $sub_p = \{c' \in C \mid c \in super_p(c')\}$ bezeichnet die Menge der (direkten) *polymorphen Oberklassen* einer Klasse c. ❑

Definition 3.4 fordert, dass $super_p$ konform zu $super$ ist (1). Offensichtlich sind nicht alle Untermengen von $super$ mögliche Kandidaten für $super_p$ da eine Klasse, die von zwei Klassen c und c' mit disjunkten Instanzmengen abgeleitet ist, nicht dieselbe Instanzmenge wie ihre Oberklassen haben kann. Deshalb fordert (2), dass c und c' einen gemeinsamen Vorfahren in Bezug auf $\overline{super_p}$ haben. Ein Beispiel für $super_p$ gibt Abbildung 16, wobei $super_p$ durch doppelte Linien und schattierte Dreiecke gezeichnet ist, d.h. $super_p(I) = \{F\}$, $super_p(J) = \{G\}$, $super_p(D) = \{B\}$ und $super_p(E) = \{B\}$, während $super_p(c) = \emptyset$ für alle $c \in \{A, B, C, F, G, H\}$. Klassen, die zu anderen Klassen durch $super_p$ oder sub_p verbunden sind, werden mit schattierten Kästen gezeichnet und heißen *polymorphe Klassen* – ihre direkten Instanzen heißen *polymorphe Objekte*. Polymorph abgeleitete Klassen bzw. Perspektiven haben alle Eigenschaften von traditionellen Unterklassen, mit Ausnahme der gemeinsamen Instanzmenge zu ihrer Oberklasse.

Definition 3.5: (*polymorpher Klassenkomplex*)

Seien C, $super_p$, und sub_p gemäß den vorangegangenen Definitionen gegeben. Die Funktion $pcc: C \to 2^C$ definiert durch $pcc(c) = \overline{sub_p(c) \cup super_p(c)}$ bezeichnet den *polymorphen Klassenkomplex* einer Klasse c. ❑

Jedes Objekt, das direkte Instanz einer Klasse c ist, ist auch direkte Instanz aller Elemente von $pcc(c)$. Man beachte, dass pcc idempotent ist, da für alle $c' \in pcc(c)$, $pcc(c') = pcc(c)$ gilt. Im Schema von Abbildung 15 ist pcc definiert durch $pcc(D) = \{B, D, E\}$, $pcc(F) = \{F, I\}$, $pcc(G) = \{G, J\}$, und pcc angewendet auf A, C und H ergibt die Menge, die lediglich das Argument von pcc enthält.

Eine wesentliche Folgerung aus der Definition von pcc ist die Möglichkeit der kovarianten Redefinition von Attributen und Methodenargumenten, sofern

Originaltyp der geerbten Definition und redefinierter Typ Elementen des selben pcc sind.

Definition 3.6: (*kovariante Redefinition*)

Seien C und sub_p sowie $c,c' \in C$ mit $c' \in super(c)$ gemäß den vorangegangenen Definitionen gegeben. Jedem Attribut $c.a \in c.A$ sei eine Signatur der Form $c.a :\rightarrow t$ und jeder Methode $c.f \in c.F$ eine Signatur der Form $c.f : t_1,\ldots,t_n \rightarrow t_{n+1}$ mit $t_1,\ldots,t_{n+1} \in C \cup SimpleTypes$ zugeordnet. $c.a$ heißt kovariant redefiniert bezüglich c' genau dann, wenn

- $\exists c'.a \text{ mit } c'.a \rightarrow t' : t,t' \in C \wedge t \in sub_p(t') \setminus t$

gilt. $c.f$ heißt kovariant redefiniert bezüglich c' genau dann, wenn

- $\exists c'.f \text{ mit } c'.f : t'_1,\ldots,t'_n \rightarrow t'_{n+1} : \forall t'_i, i \in (1,\ldots,n) :$
  $(t_i \in SimpleTypes \wedge t_i = t'_i) \vee (t_i, t'_i \in C \wedge t_i \in sub_p(t'_i) \setminus t_i)$

gilt. ❑

Kovariante Redefinition durch beliebige Unterklassen, wie sie z.B. von der Sprache Eiffel implementiert wird, ist i.A. nicht typsicher, da bei einer Zuweisung der Form $v \coloneqq v'$ mit $stype(v') \in super(stype(v))$ nicht zur Compile-Zeit geprüft werden kann, ob $dtype(v')$ den kovarianten Redefinitionen von $dtype(v)$ genügt. Die Beschränkung auf Redefinitionen innerhalb polymorpher Klassenkomplexe beseitigt dieses Problem jedoch, da das Vorhandensein der zusätzlichen Eigenschaften in der Redefinition innerhalb von $pcc(dtype(v))$ garantiert ist.

Abgesehen von polymorphen Klassenkomplexen sind wir auch an der Menge von Typen interessiert, die als Schnittstellen eines gegebenen Objekts genutzt werden können. Wie in Abschnitt 3.1.1 erwähnt, beschreibt $\overline{super}$ die implizite Konvertierung von Referenz-Variablen. Eine Konvertierung von Variablen des Typs F zum Typ I ist damit noch nicht erlaubt, obwohl alle direkten Instanzen von F auch direkte Instanzen von I sind. Tatsächlich gibt es in unserem Modell keine implizite Typ-Konvertierung zwischen Elementen eines polymorphen Klassenkomplexes, die nicht durch $\overline{super}$ verbunden sind. Das System bietet jedoch explizite Konvertierungsoperatoren an, die den Transfer von Objekten eines Modells in ein anderes unterstützen, d.h. Referenz-Variablen mit statischem Typ $c \in C$ können explizit zu allen Elementen genau der Menge von Typen konvertiert werden, die durch die folgende Definition beschrieben wird:

Definition 3.7: (*konvertierbare Typen*)

Seien C, V, $stype$, pcc und $super_p$ gemäß den vorangegangenen Definitionen gegeben. Die Funktion $ctypes{:}C \rightarrow 2^C$ definiert durch $ctypes(c) = \overline{pcc(c) \cup super(c)}$ bezeichnet die Menge von Typen, zu denen eine Variable v (explizit) *konvertierbar* ist. ❑

Die Idee dieser Definition ist es, der Klassenhierarchie rekursiv aufwärts bezüglich $super$ und abwärts bezüglich sub_p zu folgen. Als Beispiel ist die Berechnung von $ctypes(F)$ in Abbildung 16 bis Abbildung 20 illustriert.

Wir beginnen mit $pcc(F) = \{F, I\}$. Der nächste Schritt fügt alle direkten Oberklassen von F und I hinzu (Abbildung 18). Erneute Anwendung von pcc auf alle bisher berechneten Typen fügt D hinzu, da $D \in sub_p(B)$ (Abbildung 19). Im letzten Schritt berechnen wir A als Oberklasse von D (Abbildung 20). Mit anderen Worten kann eine Variable vom statischen Typ F – zusätzlich zur impliziten Konvertierung nach B – explizit zu A, D, E und I konvertiert werden. Diese Operation ist typsicher, da $ctypes$ leicht zur Compile-Zeit berechnet werden kann. Wir haben implizite Konvertierung für $ctypes(c) \setminus \overline{super(c)}$ nicht erlaubt um eine klare Trennung zwischen den verschiedenen Typen in $ctypes(c)$ zu erhalten, was uns hilfreich sowohl in Bezug auf Lesbarkeit des Codes als auch Auflösung von virtuellen Methodenaufrufen erscheint.

Die Definition der dynamischen Typen von Variablen aus Abschnitt 3.1.1 ist nun hinfällig, da eine Variable v i.A. mehr als einen dynamischen Typ hat und diese Typen nicht notwendigerweise Unterklassen von $stype(v)$ sein müssen.

Definition 3.8: (*dynamische Typen*)

Seien C, pcc, V und $stype$ gemäß den vorangegangenen Definitionen gegeben. Eine Funktion $dtypes{:}V \rightarrow 2^C$ bezeichnet die Menge der dynamischen Typen einer Referenz-Variablen v genau dann, wenn für alle $v \in V$

- $dtypes(v) \subseteq \overline{sub(stype(v)) \cup pcc(stype(v))}$ (1)
- $\exists c \in C : dtypes(v) = pcc(c)$ (2)

gilt. ❑

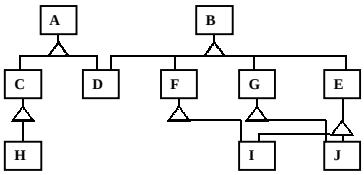

Abbildung 15

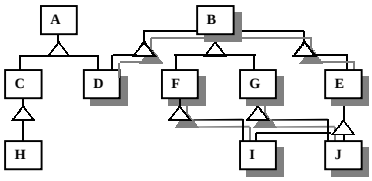

Abbildung 16

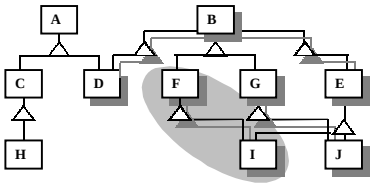

Abbildung 17

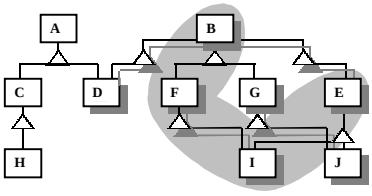

Abbildung 18

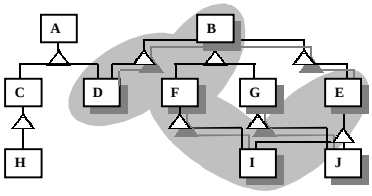

Abbildung 19

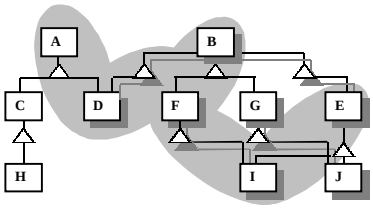

Abbildung 20

(1) ist ähnlich wie *ctypes*, folgt aber der Klassenhierarchie in entgegengesetzter Richtung. (2) beschränkt *dtypes* auf polymorphe Klassenkomplexe. Analog zur Definition von *dtype* in Abschnitt 3.1.1 bezeichnet $dtypes(v)$ immer den polymorphen Klassenkomplex der Klasse, deren Konstruktor das von v referenzierte Objekt instanziiert hat. Während in konventionellen objektorientierten Typsystemen jedes Objekt mit dynamischem Typ c auch indirekte Instanz aller Oberklassen von c ist, sind polymorphe Objekte, die durch den Konstruktor von c instanziiert wurden, zusätzlich auch indirekte Instanzen der Vereinigung von *ctypes* angewendet auf alle Elemente von $pcc(c)$. Da jedoch *pcc* idempotent ist, gilt immer $ctypes(c') = ctypes(c)$ für alle $c' \in pcc(c)$. Wir können also die Menge aller Klassen, von denen ein Objekt Instanz ist, wie folgt definieren:

Definition 3.9: (*Instanz-von*)

Seien C, V und *dtypes* gemäß den vorangegangenen Definitionen gegeben und sei $r(pcc(c))$ eine beliebiges Element des polymorphen Klassenkomplexes von c. Die Funktion $io: V \rightarrow 2^C$ definiert durch

$$io(v) = ctypes(r(dtypes(v)))$$

bezeichnet die Menge der Typen, von denen das von v referenzierte Objekt (direkte oder indirekte) *Instanz* ist. ❑

Für eine Variable v mit $stype(v) = D$ und $dtypes(v) = \{F, I\}$ stellt sich nun die Frage, welche Methoden Implementierung für einen Aufruf von *D.print* auszuführen ist, wobei wir annehmen, dass *print* von B geerbt und in D, E, F, G und I redefiniert ist. Konventionelle objektorientierte Typsysteme nutzen den (einzigen) dynamischen

Typ zur Auflösung des Methodenaufrufs – in unserem Fall sind jedoch weder F noch I Unterklassen von D. Demnach würde die Ausführung von *F.print* oder *I.print* unerwartete Effekte für einen Entwickler haben, der eine Variable vom statischen Typ D benutzt. Konsequenterweise muss die Methodenauflösung derart eingeschränkt werden, dass lediglich Implementierungen, die konform zum statischen Typ der Variablen sind, ausgewählt werden können.

Definition 3.10: (*implementierende Typen*)

Seien C, V, *sub*, sub_p, *stype*, *ctypes* und *io* gemäß den vorangegangenen Definitionen gegeben. Die Funktion $itypes: V \to 2^C$ definiert durch

$$itypes(v) = \left\{c \in io(v) \mid \neg\exists c' \in io(v): c' \neq c \wedge c' \in \overline{sub(c) \setminus sub_p(c)}\right\} \cap \overline{sub(stype(v))}$$

bezeichnet die Menge der *implementierenden Typen* einer Variablen v. ❑

Das erste Argument der Schnittmenge dieser Definition wählt alle Blattklassen von *ctypes* und deren polymorphe Klassenkomplexe aus. Die resultierende Menge wird mit allen Unterklassen von *stype*(v) geschnitten. Im oben erwähnten Beispiel ist $itypes(v) = \{D\}$ obwohl das von v referenzierte Objekt keine direkte Instanz von D ist.

3.2 Dynamisches Binden

Definition 3.10 beseitigt nicht alle Mehrdeutigkeiten für Methodenaufrufe, da z.B. $itypes(v) = \{D, F, I\}$ wenn $stype(v) = B$ und $dtypes(v) = \{F, I\}$. Dieser Abschnitt stellt unser Konzept zur Auswahl der korrekten Methodenimplementierung sowie des korrekten Typs kovariant redefinierter Attribute vor. Wesentliches Ziel bei der Entwicklung dieser Verfahren war es, vom CPL-Programmierer keinerlei Fallunterscheidungen bezüglich der dynamischen Typen von Variablen zu verlangen. Statt dessen sollten Konstrukte zur Verfügung gestellt werden, um das Verhalten von einzelnen Referenzen, Objekten, oder der gesamten Anwendung einfach und sicher zu manipulieren. Eine weitere Anforderung an den Methoden-Dispatcher ist die Vorhersagbarkeit des Verhaltens vom Entwickler, d.h. beim Auflösen von Mehrdeutigkeiten sollten nicht etwa „magische Konstanten“ wie vom Compiler vergebene Prioritäten innerhalb der Klassenhierarchie eine Rolle spielen.

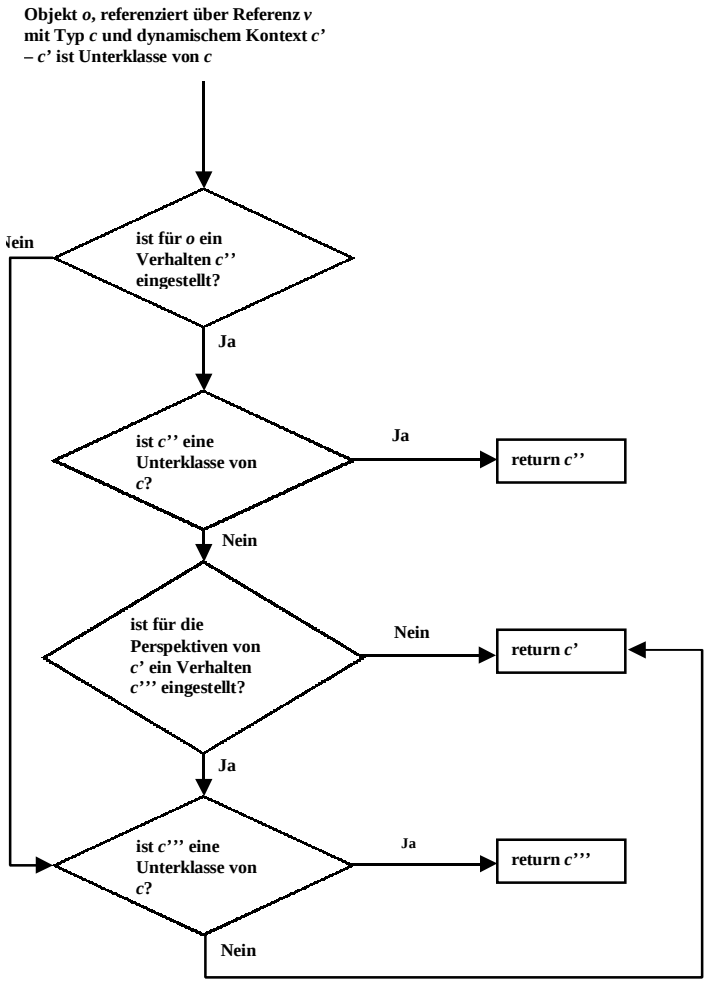

Abbildung 21: Algorithmus type-dispatch

3.2.1 Methoden

Die grundsätzliche Lösungsidee ist die Speicherung des *dynamischen Kontextes* in einer Referenz-Variablen. Mit dynamischem Kontext bezeichnen wir den aktuellen dynamischen Typ, der bei der letzten Zuweisung an die Variable eindeutig war. Wenn wir zusätzlich davon ausgehen, dass beim Erzeugen eines Objektes seiner zugehörigen Referenz als dynamischer Typ genau der Typ zugewiesen wird, dessen Konstruktor das Objekt instanziierte, dann sind eigentlich schon alle Mehrdeutigkeiten aufgelöst. Veränderungen im Verhalten können über Zuweisungen an Variablen anderen statischen Typs (explizit Konvertierung, siehe Abschnitt 3.1.2) erreicht werden. Dieses Verfahren reicht jedoch nicht aus, um deskriptiv das Verhalten mehrerer oder gar aller Objekte eines Typs in einer Anwendung zu verändern, da der Entwickler über alle Referenzen des fraglichen Typs iterieren müßte, um ihnen per Zuweisung einen neuen dynamischen Typ zu geben. Um dies zu vermeiden, bietet CPL – neben der Manipulation einzelner Referenzen – zwei weitere Operatoren zur Verhaltensmanipulation an: Änderung des Verhaltens einer spezifischen Instanz einer Klasse und Änderung des Verhaltens aller Instanzen

einer Klasse. Dies bedeutet jedoch zusätzlich Aufwand bei der Implementierung des dynamischen Bindens durch den CPL-Compiler, da Definition 3.10 verlangt, dass die tatsächlich ausgeführte Methode zu einer Unterklasse des statischen Typs der Referenz gehört.

Das in Abbildung 21 vorgestellte Verfahren ist eine vereinfachte Version des tatsächlich implementierten Algorithmus, da in CPL zusätzlich noch berücksichtigt werden muss, ob Objekt- oder Klassenverhalten zuletzt aktiviert worden sind. Die exakte Darstellung findet sich in Abbildung 22. Für den Programmierer bedeutet dies, dass beispielsweise die Aktivierung eines objektspezifischen Verhaltens eventuell vorher aktivierte Klassenverhalten für diese Instanz überschreibt. Ein Rücksetzen des Objektverhaltens (und damit die Reaktivierung des Klassen- oder Referenzverhaltens) ist ebenfalls durch einen speziellen Operator möglich.

```
type dispatch(pointer p, type stype) {
  if (p->time > 0) {
    if (p->type.is_subtype_of(stype)) {
      if (p->type.class_time > p->time) {
        if (p->type.class_type.is_subtype_of(stype)) {
          return select_type(p->type.class_type, p->original_type);
        }
      }
      else {
        return select_type(p->type, p->original_type);
      }
    }
  }
  if (p->type.class_type.time > 0) {
    if (p->type.class_type.is_subtype_of(stype)) {
      return select_type(p->type.class_type, p->original_type);
    }
  }
  return p.type;
}
```

Abbildung 22:Berechnung der aktuellen Perspektive für das dynamic binding

Wie man leicht sieht, kann *type-dispatch* in konstanter Zeit ausgeführt werden. Die beiden Überprüfungen der Typkonformität werden dabei durch je einen Zugriff auf eine Bitmatrix realisiert, welche die transitive Adjazenzmatrix der Vererbungshierarchie darstellt. Jede Klasse erhält vom CPL-Compiler einen eindeutigen 16-bit Identifikator, der sowohl als dynamischer Kontext in Referenzen als auch als Index in der Adjazenzmatrix dient. In den Objekten selbst dient dieser Identifikator als Äquivalent zum sog. VTable-Pointer in konventionellen

objektorientierten Sprachen. Die eigentliche Ausführung der Methode f könnte dann über einen Funktionszeiger, der sich in einem Array an der Stelle mit der Nummer der ausgewählten Klasse befindet, geschehen.

3.2.2 Attribute

Eine naive Implementierung des Bindens von redefinierten Attributen wäre die Erzeugung einer dynamisch gebundenen Zugriffsfunktion und deren Ausführung gesteuert durch Algorithmus *type-dispatch*. Der dadurch entstehende overhead bei Zugriffen auf redefinierte Attribute erschien uns jedoch zu groß[7]. Andererseits muss sich die Redefinition von Attributen auch dann auswirken, wenn diese in nicht dynamisch gebundenen Methoden benutzt werden. Betrachten wir als Beispiel eine Klasse c mit Attribut $c.a$ und $stype(c.a) = d$ sowie einer nicht virtuellen Methode $c.f$. Weiterhin sei $c' \in sub(c)$ mit $stype(c'.a) = d'$ und $d' \in pcc(d)$ gegeben

```
void C::f(void) const
{
   a->print();
   ...
}
...
C* c = ...
c->f();
```

.

Der Algorithmus zur Bindung von Attributen soll im Prinzip den gleichen Anforderungen wie *type-dispatch* genügen, jedoch keinen Funktionsaufruf auslösen. Mit anderen Worten soll im obigen Codefragment $d'.print$ ausgeführt werden, wenn der dynamische Kontext von *this* c' ist. Dazu wird zunächst *type-dispatch* ausgeführt, das als Ergebnis den dynamischen Kontext von *this* liefert. Vor dem Zugriff auf a wird durch einen lookup in einer Redefinitionsmatrix der zum dynamischen Typ von *this* passende dynamische Kontext des Attributs gesetzt. In dieser Matrix ist für jedes Paar (c, c') der Identifikator der Redefinition gespeichert, falls vorhanden. Dereferenzierung von a führt anschließend zum gewünschten Verhalten.

---

[7] Man beachte, dass bei virtuellen Methoden die Technik des Funktionszeigers oder einer Form von Fallunterscheidung für die möglichen implementierenden Typen quasi unvermeidbar ist, bei Attributzugriffen jedoch zu unerwarteten Effizienzeinbußen führen würde.

3.2.3 Vergleich mit anderen Arbeiten

Für das Method-Dispatch bei multipler Instanziierung können als Vergleich lediglich einige Arbeiten aus dem Bereich objekterhaltende Sichten für ODBMS herangezogen werden. Bei diesen aus dem Datenbank-Umfeld stammenden Vorschlägen werden oft programmiersprachliche Konzepte und Mechanismen wie Typsicherheit oder dynamisches Binden vernachlässigt. So untersucht keiner der vorgestellten Ansätze die Auswirkungen der Bildung von virtuellen Klassen auf die Redefinierbarkeit von Attributen/Methoden. Die Vorschläge zum Method-Dispatch unterscheiden sich grundsätzlich vom hier vorgestellten Konzept: Während [EdDo94] von einzelnen Anwendungen verlangen, niemals zwei Sichten einer Klasse, die in einer Ober-/Unterklasse-Beziehung stehen, zu nutzen und damit eine flexible Nutzung aller Sichten innerhalb einer komplexen Anwendung ausschließen, schlagen [KRR95] vor, im Falle mehrerer ausführbarer Methoden einen Laufzeitfehler zu erzeugen[8]. Beide Ansätze schränken die in der objektorientierten Programmierung so wichtige Möglichkeit der Methodenredefinition eklatant ein und wurden deshalb von uns als nicht brauchbar eingestuft.

Andere Gruppen schlagen Klassenprädikate für die Auflösung von Mehdeutigkeiten beim Methodenaufruf vor [AbBo91, SAD94, Schi93]. Der Programmierer schreibt für jede Klasse ein Prädikat, dessen Auswertung den Methoden-Dispatcher die Klassenzugehörigkeit eines gegebenen Objekts bestimmen lässt. Da diese Prädikate nicht notwendigerweise disjunkte Mengen erzeugen, wird der Dispatcher diejenigen Klassen auswählen, deren Prädikat beim Test als erstes den wert *true* zurückgegeben hat. Diese zunächst sehr flexibel erscheinende Vorgehensweise erweist sich aber als nicht typsicher: So würde beim Aufruf der Methode *zeichnen* eines Objekts in der Perspektive *GebaeudeTW* für die zu zeichnenden Bauteile trotzdem die Methode *BauteilStatik::zeichnen* ausgeführt, falls das Klassenprädikat von *BauteilStatik* gerade den Wert *true* für die fraglichen Bauteile zurückliefert. Dazu kommt, dass die Änderung des Verhaltens von Objekten immer mit einem Update

[8] Der Ansatz aus [KRR95] stammt aus der SmallTalk-orientierten, nicht statisch getypten Welt. Derartige Sprachen legen es dem Entwickler sehr nahe, zur Laufzeit Fehlermeldungen bzw. Ausnahmen zu erzeugen, wenn referenzierte Objekte die aufgerufene Methode nicht ausführen können bzw. nicht über diese Methode verfügen („message-not-understood“). In diesem Zusammenhang scheint die Idee, einen Laufzeitfehler bei Mehrdeutigkeiten im Method Dispatch zu erzeugen, verständlich. In unserer Arbeit gehen wir jedoch immer von der Voraussetzung einer strengen Typisierung aus und lehnen deshalb Laufzeitfehlermeldungen wegen Typfehlern ab.

verbunden ist, um das gewünschte Klassenprädikat zu aktivieren. Das wiederum führt einerseits zu evtl. großen Laufzeiteinbußen und verhindert andererseits die gleichzeitige Nutzung von Objekten in verschiedenen Perspektiven[9].

Der Vorschlag von [BeGu95] ist der einzig uns bekannte Ansatz, der für die Methodenauswahl Typkonformität zum statischen Typ der Referenzvariablen, über welche die Methoden aufgerufen werden, fordert. Die verbleibenden Mehrdeutigkeiten werden jedoch durch vom Compiler vergebene und damit vom Entwickler weder nachvollziehbare noch manipulierbare Prioritäten aufgelöst. Auf variante Attribute und deren Auswirkungen auf Perspektiven wird nicht weiter eingegangen.

3.3 Änderungspropagierung

Unter dem Begriff „Änderungspropagierung“ werden in der Datenbank-Literatur häufig Konzepte bzw. Mechanismen dargestellt, welche die Auswirkungen von Änderungen auf physisch gespeicherten Objekten beschreiben. Gemeint sind damit abgeleitete Änderungen in Sichtobjekten – also Tupeln in virtuellen Relationen oder Objekten in virtuellen Klassen. Im folgenden verwenden wir den Begriff jedoch in einer anderen Wiese: Änderungen an erweiterten Attributen einer Perspektiven werden auf Attribute einer anderen Perspektive des selben *pcc* propagiert, um semantische Abhängigkeiten zu modellieren. Anders als im Datenbank-Kontext handelt es sich bei dieser Diskussion also ausdrücklich nicht um die Rekonstruktion abgeleiteter Objekte nach erfolgten Updates der Basis-Objekte, sondern um die Manifestation von semantischen Beziehungen zwischen ansonsten voneinander unabhängigen Attributen.

Da Objekte direkte Instanzen mehrerer Klassen sind, ist die Konsistenz von Attributen, die in allen Elementen des polymorphen Klassenkomplexes sichtbar sind, von allein gegeben. Ein „Nachziehen“ von updates eines Attributs $c.a$ in der Klasse c' ist also nicht notwendig, wenn $c'.a$ ebenfalls Element von $c.A$ ist. Schwieriger wird die Situation jedoch, wenn Attribute, die nicht in allen Elementen des polymorphen Klassenkomplexes sichtbar sind, semantische Abhängigkeiten voneinander aufweisen. Als Beispiel betrachten wir einen polymorphen Klassenkomplex {*Bauteil*, *BauteilStatik*, *BauteilTragwerk*}, wobei *BauteilStatik* und *BauteilTragwerk* je über eine

[9] Ein update an einem persistenten Objekt mit dem Ziel, ein bestimmtes Klassenprädikat zu aktivieren, würde eine – im Allgemeinen unerwünschte – Verhaltensänderung des Objektes auch in anderen Anwendungen zur Folge haben.

eigene geometrische Repräsentation verfügen. Updates an der geometrischen Repräsentation in *BauteilTragwerk* sollten nun auch entsprechende updates in *BauteilStatik* zur Folge haben, soweit dies automatisch ausführbar ist. Die beiden Repräsentationen wissen jedoch nichts voneinander, da sie jeweils erst in den Erweiterungen *BauteilTragwerk* bzw. *BauteilStatik* definiert worden sind.

Die Semantik von updates auf Objekten, die direkte Instanz mehrerer Klassen sind, ist in der Literatur bislang noch wenig untersucht worden. Die einzigen Hinweise im Hinblick darauf, welche updates bei einem gegeben Aufruf der Form $c.f$ auszuführen sind, werden im Zusammenhang mit method dispatch Techniken gegeben. Der folgende Abschnitt wird die Probleme herausstellen, die in diesem Zusammenhang auftreten wenn nur die eine, vom method dispatcher ausgesuchte Methode ausgeführt wird.

3.3.1 Objekt-Integrität

3.3.1.1 Design by contract

Unser Begriff der Integrität von Objekten basiert auf dem *design by contract* (DBC) Paradigma [Meye98, GeDi94]. Der „Vertrag“ (engl. contract) zwischen Aufrufer (*client*) und Aufgerufenem (*supplier*) wird durch Methoden-Vorbedingungen (*preconditions*), Methoden-Nachbedingungen (*postconditions*) und Klassen-Invarianten (*invariants*) beschrieben. Letztere werden über dem Zustand des suppliers, einschließlich geschachtelter Unterobjekte, definiert. Postconditions werden über den Zustand des Empfängerobjektes bei Eintritt in den Methodenrumpf, den Zustand des Objekts bei Austritt aus dem Methodenrumpf sowie über die Out-Parameter der Methode definiert. Man beachte, dass auf diese Weise auch dynamische Integritätsbedingungen formuliert werden können. Preconditions können den Zustand des Objekts sowie die Werte aller übergebenen Parameter der Methode benutzen.

Die *require*-Klausel der Methode *Stack<T>::pop* aus Abbildung 23 verlangt, dass der stack nicht leer ist. Dies ist der Teil des Vertrages, der vom Client zu garantieren ist – vor Aufruf der Methode muss der Client sicher sein, dass alle preconditions eingehalten werden. Der für den supplier verbindliche Teil des Vetrages für diese Methode besteht aus der postcondition: die *ensure*-Klausel der Methode *Stack<T>::pop* besagt, dass die Größe des stack nach Ausführung des Rumpfs um eins kleiner sein muss als vorher. Mit dem Schlüsselwort *old* kann innerhalb von

postconditions immer auf den Zustand des Objekts vor Eintritt in den Rumpf zugegriffen werden.

```
template <class T> class Stack {
public:
   ...
   void pop();
   void push(T value);
   T    top() const;
   bool empty() const;
   int  size() const;
   ...
private:
   int      _size;
   Array<T> _array;
};
template <class T> void Stack<T>::pop() {
   require(!empty());
   ensure(size() == old.size() - 1);
   ...;
}
template <class T> void Stack<T>::invariant() const {
   CHECK(size() == 0 && empty() || size() > 0 && !empty());
   CHECK(size() < _array.size());
}
```

Abbildung 23: *design by contract* für die Klasse `Stack<T>`

Der vielleicht wichtigste Teil des Vertrages liegt jedoch in der Invariante: alle hier als *CHECK*-Klauseln formulierten Ausdrücke gehören zu den Verpflichtungen des suppliers; man kann sie auch als implizite postconditions für alle Methoden der Klasse ansehen. Der interessante Punkt bei Invarianten ist, dass sie die möglichen, konsistenten Zustände von Objekte beschreiben. Beispielsweise ist ein stack, dessen interner Array von Elementen kleiner ist, als die in einem weiteren Attribut gehaltene aktuelle Größe des stacks, in einem nicht definierten Zustand. Invarianten bieten deshalb die Möglichkeit, Integritätsbedingung für alle potentiellen Instanzen einer Klasse zentral und deskriptiv zu spezifizieren.

Alle drei Teile des Vertrages vererben sich bei der Ableitung neuer Unterklassen: Invarianten und postconditions können verschärft werden, preconditions abgeschwächt. Wenn eine neue Invarianten-Methode für die Unterklasse definiert wird, so enthält sie implizit alle Klauseln der Oberklasse zusätzlich zu den neu formulierten *CHECK*-Anweisungen. Auch für postconditions wird im Falle einer Redefinition der Methode in der Unterklasse nach deren Ausführung zunächst die in der Oberklasse formulierte postcondition geprüft, dann die neuen Klauseln der Redefinition. Preconditions dürfen nicht verschärft werden, weil dies dem „*Liskov*

Principle of Substitutality“ [LiWi93] widersprechen würde. Ein Client, der eine Variable vom Typ A hält und auf dieser eine Methode f aufruft, kennt im Allgemeinen nur deren precondition und nicht etwa die preconditions sämtlicher Redefinitionen von f in den Unterklassen von A. Insbesondere kann der Client nicht preconditions von Redefinitionen in Unterklassen kennen, die zum Zeitpunkt der Übersetzung des Client-Codes noch gar nicht definiert waren. Dementsprechend würde eine Verschärfung einer precondition unter Umständen zu einem Fehler in einem Teil des Client-Codes führen, der bisher einwandfrei funktionierte. Preconditions werden deshalb verodert, d.h. wenn in der Oberklasse die precondition von f fehlschlägt, so wird die *require*-Klausel der Redefinition zusätzlich geprüft. Wenn sie erfolgreich ist, wird mit der Ausführung fortgefahren, andernfalls gilt die gesamte precondition als gescheitert.

```
class Automobil {
   ...
   void limousine();
private:
   void fuegTuerEin(Tuer* t);
   List<Tuer*>* tueren;
};
void Automobil::limousine() {
   require(tueren->cardinality() == 2);
   ensure(tueren->cardinality() == 4);
   fuegeTuerEin(new Tuer);
   fuegeTuerEin(new Tuer);
}
void Automobil::invariant() const {
   CHECK(tueren != 0);
   CHECK(tueren->cardinality() == 2 || tueren->cardinality() == 4);
}
```

Abbildung 24

Das Laufzeitsystem kümmert sich um die tatsächliche Ausführung solcher Checks, falls dies vom Entwickler gewünscht wird. Invarianten werden dabei üblicherweise bei Austritt aus Methodenrümpfen, vor den postconditions geprüft. Wegen des *indirect-invariant-effect* [Meye98] kann eine Prüfung der Invarianten zusätzlich optional auch vor Eintritt in die Methode erfolgen. Invarianten werden jedoch grundsätzlich nicht geprüft, wenn eine Methode mit *this* als Empfänger aufgerufen wird. Dies ist notwendig, um Objekten einen temporär inkonsistenten Zustand zu erlauben. Zum Beispiel kann die Invariante einer Klasse Automobil verlangen, dass jedes Auto entweder zwei oder vier Türen besitzt (Abbildung 24). Beim Übergang von einem zweitürigen Fahrzeug zu einem viertürigen muss jedoch zwangsläufig für kurze Zeit ein inkonsistenter Zustand mit drei Türen angenommen werden. Der

Client kann auf diesen inkonsistenten Zustand jedoch niemals selbst zugreifen, weil er nur innerhalb einer von ihm aufgerufenen Methode auftritt und bei Austritt aus dieser Methode bereits wieder beseitigt ist.

Die einzelnen Bestandteile des Vertrages rufen sich niemals gegenseitig auf. Das bedeutet, dass beispielsweise bei einer Precondition-Prüfung in den von dort aus aufgerufenen Methoden selbst keine preconditions, postconditions oder Invarianten geprüft werden. Damit wird eine unter Umständen auftretende Endlosrekursion durch sich gegenseitig aufrufende Checks vermieden. Nähere Erläuterungen zum Ausführungsmodell des design by contract finden sich in [Meye98] und [Meye92].

3.3.1.2 Erweiterung von PBC für Perspektiven

Die Anpassung des DBC-Konzepts auf ein Typsystem mit polymorphen Klassen bezieht sich zunächst nur auf postconditions und Invarianten. Diese beiden Bestandteile des Vertrages sind schon deshalb für die Betrachtungen von updates die Wichtigsten, weil sie die Semantik von update-Operationen teilweise spezifizieren. Während in konventionellen Sprachen ein Objekt einer Klasse c alle Invarianten der Klassen $c \cup \overline{super(c)}$ zu erfüllen hat, muss sich dies bei Perspektiven auf die Menge $ctypes(c)$ erweitern. Dies ist eine sehr naheliegende Anwendung des Konzepts der Invarianten auf polymorphe Klassen, da das Objekt unter jeder Klasse $c' \in ctypes(c)$ benutzt werden kann und deshalb für alle diese Klassen einen konsistenten Zustand haben muss. In gleicher Weise verlangen wir für jeden Aufruf $c.f$ für eine Variable v mit $stype(v) = c$, dass die Konjunktion der postconditions aller Redefinitionen $c'.f$ mit $c' \in ctypes(c)$ gelten muss. Diese Form von Klassen-Invarianten und Methoden-postconditions erzwingt update-Konsistenz unabhängig davon, welche Methodenimplementierung ursprünglich vom Methoden-Dispatcher ausgesucht wurde. Das noch offene Problem ist jedoch, welche Unterstützung die Sprache für die Einhaltung derartiger Invarianten und postconditions bieten muss.

3.3.2 Update Operationen

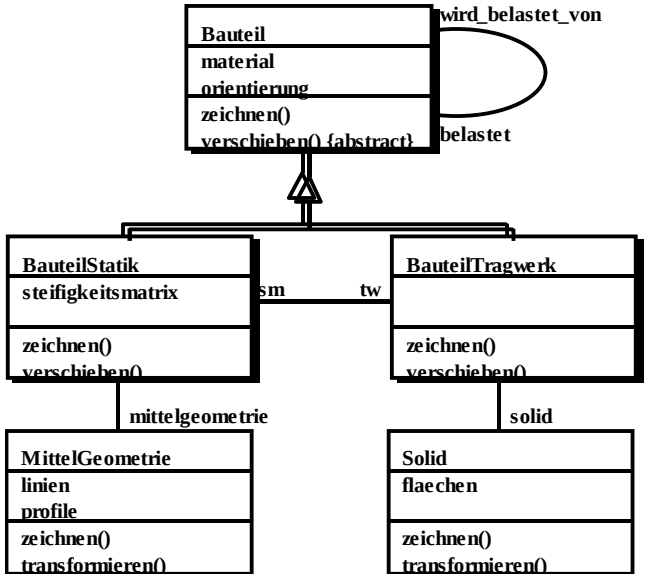

Abbildung 25

In der Abbildung 25 ist jedes Objekt, welches eine direkte Instanz von *Bauteil* ist, auch eine direkte Instanz von *BauteilStatik* und *BauteilTragwerk*. Die Frage ist, welche updates an einem Objekt tatsächlich ausgeführt werden müssen, wenn zum Beispiel die Methode *BauteilStatik::verschieben* aufgerufen wird. Die postconditions der verschieben-Methoden sind in Abbildung 26 als Kommentare spezifiziert, weil eine formale Beschreibung in diesem Fall nur sehr schwer möglich ist. Wenn wir diese beiden postconditions konjugieren und uns das konzeptionelle Objektlayout von Instanzen dieses *pcc* in Abbildung 27 betrachten, so stellen wir fest, dass eigentlich sowohl die Transformation auf dem Attribut *BauteilStatik::mittelgeometrie* als auch auf dem Attribut *BauteilTragwerk::solid* durchgeführt werden muss, um die postconditions zu erfüllen und das Objekt in einem konsistenten Zustand zu hinterlassen. Ein Aufruf der Methode *BauteilStatik::verschieben* alleine hätte das Scheitern der postcondition von *BauteilTragwerk::verschieben* zur Folge. In einem Design mit multipler Vererbung (siehe Abbildung 6 in Abschnitt 2.2) würde dieses Problem zwar nicht auftreten, aber zusätzlich zu denen in 2.2 diskutierten Nachteilen bedeutet dieser Ansatz auch, dass der Entwickler anwendungsübergreifend denken und programmieren muss – wir wollen jedoch einen Programmierer für die Tragwerkmodellierung nicht mit Methodenaufrufen, die ausschließlich andere Modelle betreffen, belasten.

```
void BauteilStatik::verschieben(double x, double y, double z) {
  // ensure: die Geometrie ‚mittelgeometrie' ist gemaess x, y, und z
  // transformiert worden
  Bauteil::verschieben(x, y, z);
  mittelgeometrie->transform(x, y, z);
  zeichnen(); // refresh des Objekt auf dem Bildschirm
}
void BauteilTragwerk::verschieben(double y, double y, double z) {
  // ensure: die Geometrie ‚solid' ist gemaess x, y, und z
  // transformiert worden
  Bauteil::verschieben(x, y, z);
  solid->transform(x, y, z);
  zeichnen(); // refresh des Objekt auf dem Bildschirm
}
```

Abbildung 26: Implementierungen von `verschieben`

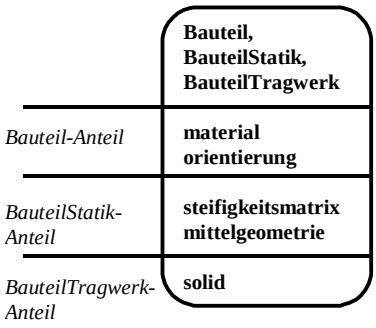

Abbildung 27: Attribute des *pcc*(*Bauteil*)

Intuitiv sollte die Semantik von update-Operationen $c.f$ in $pcc(c)$ auf irgendeine Weise äquivalent zum sukzessiven Aufruf der Implementierungen $c'.f$ für alle $c' \in pcc(c)$ in einer sinnvollen Reihenfolge sein. In unserem Beispiel würde das allerdings zu einer dreimaligen Ausführung von *Bauteil::verschieben* führen, was im Allgemeinen fatale Konsequenzen zur Folge hat. Ein etwas besserer Ansatz scheint es zu sein, jede Methode aus $pcc(c)$ jeweils auf dem selben Zustand des Basisklassen-Anteils *Bauteil* auszuführen. Dazu müssen die Methoden auf einer temporären Kopie des gemeinsamen Anteils des Objekts arbeiten. Nach der Ausführung aller Methodenrümpfe wird der resultierende Zustand des Objekts durch folgenden Algorithmus gebildet: kopiere alle Attribute des Objektes, die in allen temporären Kopien gleich sind, in die originale Instanz; für alle Attribute, deren Werte unterschiedlich in den verschiedenen Kopien sind ist zu prüfen, ob lediglich eine einzige Kopie einen anderen Wert hat als der gesamte Rest der Objektkopien. Fall nicht, so ist ein Fehler aufgetren. Dieses Vorgehen sollte garantieren, dass keine Konflikte bei den updates auf dem gemeinsame Anteil des Objekts auftraten.

Obwohl diese Technik auf den ersten Blick recht vielversprechend wirkt, sind mit ihr doch eine Reihe schwerer Nachteile verbunden. So sind temporäre Kopien und ihr

anschließendes Wieder-Zusammensetzen sehr zeitaufwendig. Wichtiger ist jedoch, dass es in objektorientierten Sprachen kein allgemeines *deep-copy*-Verfahren gibt. Aber selbst wenn wir benutzerdefinierte Kopieroperationen zulassen (wie z.B. Kopier-Konstruktoren oder Zuweisungs-Operatoren in C++) müssen wir eventuell enorme Datenmengen wiederholt rekursiv kopieren und vergleichen.

Abgesehen von diesen hauptsächlich durch Performanceüberlegungen motivierten Aspekten gibt es einen weiteren Punkt: wenn wir annehmen, dass die Implementierungen von *verschieben* – wie in Abbildung 26 beschrieben – nach der eigentlichen Operation *zeichnen* aufrufen, um das verschobene Objekt auf dem Bildschirm neu darzustellen, dann werden Benutzer des Tragwerkmodells irritiert sein, wenn plötzlich auch Zeichnungen aus dem Statikmodell auf ihrem Bildschirm auftauchen, weil alle Implementierungen von *verschieben* in *pcc*(*Bauteil*) ausgeführt werden. Dieser Effekt sollte durch Definition 3.10 eigentlich verhindert werden.

3.3.3 Methoden Implementierungs-Invarianten

Wegen der im letzten Abschnitt diskutierten Probleme scheint es mindestens schwierig zu sein, einen Algorithmus zu finden, der es dem Compiler ermöglicht, update-Operationen automatisch so zusammenzusetzen, dass durch ihre Ausführung alle postconditions aus $io(v)$ erfüllt werden. Dies führte uns zu einem anderen Ansatz: speziell wenn wir den letzten Punkt aus Abschnitt 3.3.2 betrachten fällt auf, dass ein Konzept zur Trennung „wichtiger“ updates von anderer methodenspezifischer Funktionalität notwendig ist. Diese ausgezeichneten updates müssen jeweils die modellspezifische postcondition erfüllen nachdem die Methode ausgeführt wurde. Die Idee unseres Ansatz ist es, basierend auf einer solchen Trennung alle „wichtigen“ updates von $ctypes(c)$ für jeden Aufruf von *verschieben* auszuführen; der verbleibende Rest des vom dispatcher ausgewählten Methodenrumpfes – in welchem üblicherweise keine updates im Hinblick auf den Datenbankzustand oder die Integrität des Objekts ausgeführt werden, sondern eher Bildschirm-Refresh-Operationen o.ä. spezifiziert sind – muss nur in diesem Teilmodell ausgeführt werden.

Es stellt sich nun die Frage, wie Methoden-Redefinitionen $c'.f$ von $c.f$ spezifiziert werden können, so dass sie das Objekt entsprechend der postcondition von $c'.f$ modifizieren können aber die unerwünschten Seiteneffekte nicht beinhalten. Wie oben erwähnt besteht die Idee in einer Trennung des Methodenrumpfes in zwei Bereiche. Ein Bereich implementiert die für $c'.f$ spezifischen Aktionen, welche die Integrität des Objekts für alle Elemente von $ctypes(c)$ nicht betreffen. Der zweite Teil

– den wir den *share-Block* nennen und der mit $c'.f_{share}$ notiert wird – implementiert die für $c'.f$ spezifischen updates. Die Menge der $c'.f$ share-Blöcke aller $c' \in ctypes(c)$ bildet die *Methoden-Implementierungs-Invariante*, oder kurz Implementierungs-Invariante, von f in $ctypes(c)$. Durch die Kennzeichnung der update Operationen, die für die postcondition der Methode relevant sind, ermöglicht der Benutzer es dem Compiler, die Implementierungs-Invariante, die für jeden Aufruf von $c'.f$ mit $c' \in ctypes(c)$ komplett ausgeführt zu werden hat, automatisch zusammenzusetzen. Man beachte, dass eine Implementierungs-Invariante im Gegensatz zu Klassen-Invarianten oder pre- und postconditions keine Spezifikation der vom Objekt zu erfüllenden Constraints ist, sondern eine Menge von Anweisungsfolgen bezeichnet die dazu dienen, das Objekt gemäß dieser Constraints zu modifizieren. Diese Menge von Anweisungsfolgen bleibt unabhängig davon, welche Methodenimplementierung vom Methoden-Dispatcher ausgewählt wurde, immer gleich und ist deshalb Invariante in Bezug auf das dynamische Binden. Das Konzept ist in Abbildung 28 anhand der *verschieben*-Methode in drei unterschiedlichen Perspektiven einer Bauteil-Klasse illustriert.

```
void Bauteil::verschieben(double x, double y, double z) {
  share {
    orientierung->transformiere(x, y, z);
  }
}

void BauteilTragwerk::verschiebe(double x, double y, double z) {
  share {
    solid->transformiere(x, y, z);
  }
  zeichnen(); // refresh auf dem Bildschirm
}

void BauteilStatik::verschiebe(double x, double y, double z) {
  share {
    mittelgeometrie->transformiere(x, y, z);
  }
  zeichnen(); // refresh auf dem Bildschirm
}
```

Abbildung 28: Implementierungs-Invariante der `verschiebe`-Methode

3.3.4 Ausführungsreihenfolge

Die Strategie für die Ausführungsreihenfolge von share-Blöcken spiegelt die übliche Vorgehensweise der „Methoden-Verfeinerung“ in konventionellen objektorientierten Systemen wider: zunächst wird die Implementierung der Oberklasse aufgerufen, darauffolgend zusätzliche, in der Redefinition der Methode spezifizierte

Anweisungsfolgen. Präziser ausgedrückt bedeutet dies, dass nach Eintritt in den Methodenrumpf $c.f$, der vom dispatcher für eine Variable v ausgewählt wurde, als erstes alle share-Blöcke aus $io(v)$ gemäß der durch *super* induzierten partiellen Ordnung ausgeführt werden. So wird für alle $c', c'' \in io(v)$ der share-Block von $c'.f$ vor dem share-Block von $c''.f$ ausgeführt, wenn $c' \in \overline{super(c'')}$. Im Beispiel von Abbildung 29 wird bei einer gegebenen Varibale v mit $stype(v) = BauteilStatik$ ein Aufruf von $v.verschieben(x, y, z)$ die Ausführung der Anweisungsfolgen $Bauteil.verschieben_{share}$, $BauteilTragwerk.verschieben_{share}$ und $BauteilStatik.verschieben_{share}$ auslösen, bevor schließlich der non-share-Bereich von $BauteilStatik.verschieben$ erreicht wird. In der Klassenhierarchie von Abbildung 20 würde bei einer Variablen v mit $stype(v) = D$ und $dtypes(v) = \{F, I\}$ der Aufruf einer Methode F dagegen die Folge $A.f_{share}$, $B.f_{share}$, $D.f_{share}$, $F.f_{share}$, $E.f_{share}$ und $I.f_{share}$ auslösen bevor der non-share Bereich von $D.f$ ausgeführt wird.

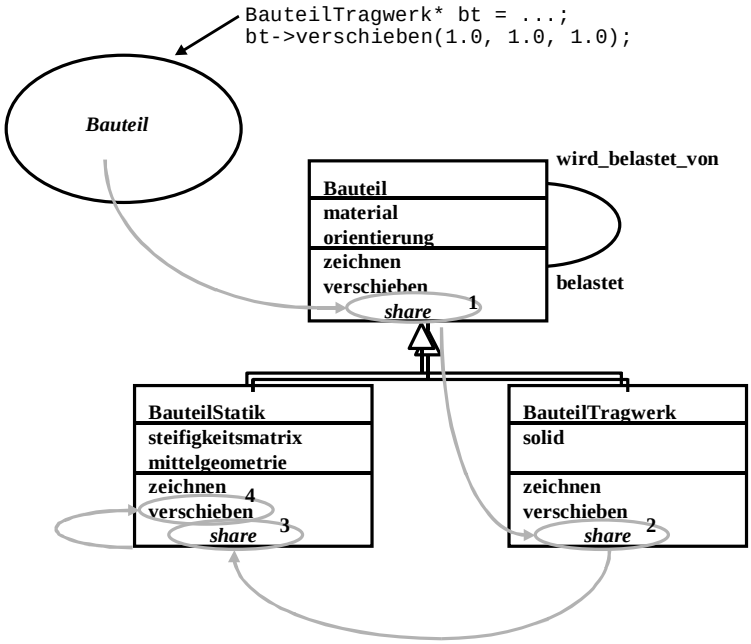

Abbildung 29: Ausführungsreihenfolge von share-Blöcken

Da Geschwisterklassen im Allgemein nicht voneinander wissen sollten ist die Reihenfolge der Ausführung von share-Blöcken zwischen Geschwistern undefiniert. Probleme entstehen in diesem Zusammenhang jedoch, wenn wenn zwei Klasen c_1 und c_2, die weder direkt noch indirekt voneinander erben, Attribute einer gemeinsamen Oberklasse c modifizieren und die Konsistenz der Objekte von der Reihenfolge dieser updates abhängt[10]. In diesem Fall existieren aber

[10] Das gleiche Problem entsteht, wenn sowohl $c_1.f_{share}$ als auch $c_2.f_{share}$ updates auf Objekten ausführen die als Referenzparameter für die Methode f übergeben werden. Als Programmierrichtlinie sollte

Abhängigkeiten zwischen c_1 und c_2 die besser explizit spezifiziert werden sollten, z.B. in dem das Schema um eine weitere Klasse c_3, die polymorph von c_1 und c_2 erbt, erweitert wird. Die share-Blöcke $c_1.f_{share}$ und $c_2.f_{share}$ werden in $c_3.f_{share}$ verlegt wo die Ausführungsreihenfolge explizit festgelegt werden kann.

Der Compiler realisiert diese Art der Methodenverfeinerung für alle Klassen $c' \in io(v)$ unabhängig davon, ob die Klassen polymorph verwandt sind oder ob es sich um konventionelle Vererbungsbeziehungen handelt[11]. Andererseits ist es häufig sinnvoll, in einer Methodenredefinition die Implementierung der Oberklasse direkt aufzurufen weil der Entwickller nicht nur an die Oberklasse betreffenden updates interessiert ist, sondern auch an den anwendungsspezifischen Teilen der Implementierungen. Konsequenterweise hat der Compiler zu garantieren, dass die Implementierungs-Invariante genau einmal ausgeführt wird nachdem die Methode von außen aufgerufen wurde. Damit müssen Aufrufe der share-Block-Folge unterdrückt werden, wenn eine Methodenimplementierung von einer Redefinition aufgerufen wird. Abbildung 30 verdeutlicht dies an Hand einer in zwei Redefinitionen auftretenden Methode f und ihren beiden share-Blöcke $A.f_{share}$ und $B.f_{share}$. $B.f$ ruft außerhalb von $B.f_{share}$ $A.f$ explizit auf und damit in diesem Bespiel $A.f \backslash A.f_{share}$, also den Aufruf der Funktion *doSomethingForA*() zu steuern. Dieser aus $B.f$ stattfindende Aufruf von $A.f$ darf jedoch nicht zu einer zweiten Initiierung der Ausführung von $A.f_{share}$ und $B.f_{share}$ führen. Die Ausführungsfolge für einen Aufruf der Form $v.f$ mit $dtypes(v) = B$ sollte vielmehr die Gestalt *doSomethingSharedForA*(), *doSomethingSharedForB*(), *doSomethingForA*(), *doSomethingForB*() haben.

---

gelten, dass updates auf Attributen der Oberklasse und auch auf Referenzparametern nur in der Oberklasse ausgeführt werden dürfen.

[11] Dabei handelt sich um einen weitere invarianten Aspekt von Methodenimplementierungs-Invarianten. Die Ausführung eines share-Blocks einer Methodenimplementierung in einer Basisklasse ist invariant zu allen möglichen Redefinitionen dieser Methode ihren Unterklassen.

```
class A {
public:
  virtual void f();
  ...
};

class B : public A {
public:
  virtual void f();
  ...
};
```

```
void A::f() {
  share {
    doSomethingSharedForA();
  }
  doSomethingForA();
}

void B::f() {
  share {
    doSomethingSharedForB();
  }
  A::f();
  // doSomethingForA()
  // soll aufgerufen werden!
  do_something_for_B();
}
```

Abbildung 30: Explizite Methodenverfeinerung und share-Blöcke

Der Compiler kann leicht genau eine Implementierungs-Invariante pro Methode und *pcc* erzeugen. Die Ausführung einer Implementierungs-Invariante wird abhängig von dem *pcc*, für den das Empfängerobjekt eines Methodenaufrus direkte Instanz ist, ausgelöst. Diese Technik kann sehr ähnlich zu konventionellen Techniken des dynmischen Bindens in traditionellen objektorientierten Programmiersprachen implementiert werden.

3.3.5 Updates auf untergeordneten Objekten

Eine wichtige Frage bei der Untersuchung der Ausdrucksmächtigkeit von Implementierungs-Invarianten ist ihre Äquivalenz zur multiplen Vererbung ohne polymorphe Klassen. Unglücklicherweise gibt es keine derartige Äquvalenz, da z.B. die Klasseninvariante der Klasse *Bauteil_SM_TW* aus Abbildung 6 eine Klausel beinhalten könnte, welche verlangt dass die geometrischen *solid* und *mittelgeometrie* Repräsentationen des Bauteils konsistent zueinander sind, wie immer dies auch sicherzustellen sein mag. Im Schema aus Abbildung 25 gibt es andererseits gar keine Möglichkeit, eine derartige Integritätsbedingung zu formulieren, da in keiner der beteiligten Klassen aus *pcc*(*Bauteil*) beide Repräsentationen bekannt sind. Man beachte, dass updates auf den geometrischen Repräsentationen konsistent zueinander sind, solange sie über Instanzen von *pcc*(*Bauteil*) ausgeführt werden; dafür wird durch die Implementierungs-Invarianten gesorgt. Ein Systemkomponente, die eine geometrische Modellierung implementiert, wird jedoch häufig eine Referenz auf ein Objekt vom Typ *MittelGeometrie* oder *Solid* erhalten und anschließend komplexe Operationen einschließlich updates auf diesen Objekte ausführen, wodurch das Interface aus den über *Bauteil* zugänglichen Methoden für Geometrien

umgangen wird und deshalb auch Methodenimplementierungs-Invarianten nicht zum Tragen kommen. Dies ist insbesondere dann kaum zu vermeiden, wenn Operationen aufgerufen werden, zu denen es in der korrespondierenden Repräsentation keine entsprechende Methode gibt. Während also allgemein geometriesche Funktionen wie Skallierung, Transfomation, Rotation etc. einfach und sehr effizient durch Implementierungs-Invarianten ind der Bauteil-Hierarchie abgebildet werden können, ist z.B. das Hinzufügen einer neuen Fläche in einer Boundary Repräsentation nicht ohne weiteres auch in einer Achs- und Flächenorientierten Mittelgeometrie darzustellen.

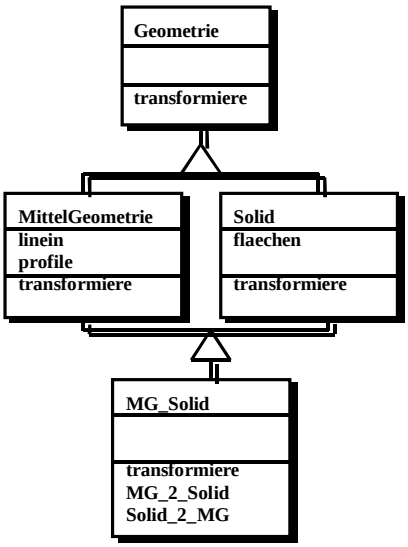

Abbildung 31

Eine Lösung dieses Problems kann in der polymorphen Ableitung der Klassen *MittelGeometrie* und *Solid* von einer gemeinsamen Oberklasse *Geometrie* bestehen (vgl. Abbildung 31). Wir definieren dann ein Referenzattribut vom Typ *Geometrie* in Bauteil und redefinieren dieses Attribut kovariant zum Typ *Solid* in der Klasse *BauteilTragwerk* und zum Typ *MittelGeometrie* in der Klasse *BauteilStatik*. Um diese Datenstrukturen konsistent zueinander zu halten brauchen wir jedoch in jedem Fall eine Systemkomponente, welche die erforderlichen Konversionen durchführen kann, d.h. eine Komponente, die beide geometrischen Datenstrukturen kennt. Dies kann durch eine Klasse *MG_Solid*, die von *Solid* und *MittelGeometrie* polymorph abgeleitet ist, modelliert werden. Diese Klasse hat damit sowohl eine *MittelGeometrie* als auch einen *Solid* geerbt. Update Methoden von *MittelGeometrie* und *Solid* sind in *MG_Solid* redefiniert, so dass im Anschluß an die ausgeführte Operation jeweils die andere Repräsentation durch Konversion aus der modifizierten Darstellung neu berechnet wird.

Eine optimierte Variante ist die Definition zweier Bool-wertiger Attribute in *MG_Solid*, inde enen bei updates vermerkt wird, ob dass die korrespondierende Darstellung nicht mehr up-to-date ist. Der Zugriff auf die Darstellung, deren Werte nicht mehr aktuell sind, löst dann die Konversion als lazy evaluation aus. Damit werden bei aufeinanderfolgenden updates einer Repräsentation ohne zwischendurch erfolgenden Zugriff auf die andere Darstellung Konversionen eingespart.

3.3.6 Vergleich mit anderen Sprachkonzepten

Während die hier vorgestellte Form der Änderungspropagierung in der Literatur so noch nicht vorgeschlagen wurde, finden sich jedoch eine Reihe von Konzepten, die mit unserem Ansatz zumindest in einem derartigen Zusammenhang stehen, dass sie hier diskutiert werden sollten. Abschnitt 3.3.6.1 stellt Mechanismen in den Programmiersprachen C++ [Stro94, ElSt90] und Beta [LMN93] vor, die zwar nicht multiple Vererbung unterstützen, sich aber mit dem Problem der Methodenverfeinerung in komplexen Vererbungshierarchien auseinandersetzen. Insbesondere interessieren uns dabei Konzepte, die eine automatische Aggregation von Methodenrümpfen aus den Implementierungen in der Klassenhierarchie gewährleisten. Abschnitt 3.3.6.2 diskutiert dagegen Ansätze aus der Datenbank-Literatur sowie Regelansätze.

3.3.6.1 Objektorientierte Programmiersprachen

Als eine der objektorientierten Sprachen, die Mehrfachvererbung und auch wiederholte Vererbung unterstützen, unterstützt C++ mit den *Konstruktoren* und *Destruktoren* genannten Initialisierungs- und Deinitialisierungsfunktionen einen Mechanismus an, der automatisch die tatsächliche Ausführungsfolge aus der Aggregation sämtlicher Implementierungen der Oberklassen eines Objekts zusammensetzt.

```
class A {
  char* a;
  virtual ~A() { delete a; }
  ...
};
class B : public virtual A {
  char* b;
  virtual ~B() { delete b; }
  ...
};
```

```
class C : public virtual A {
  char* c;
  virtual ~C() { delete c; }
  ...
};
class D : public B, public C {
  char* d;
  virtual ~D() { delete d; }
  ...
};
```

Abbildung 32: Destruktoren in C++

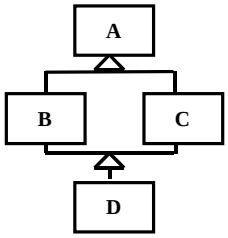

Abbildung 33: Klassenhierarchie zu Abbildung 32

Wenn wir die in C++ spezifizierte Klassenhierarchie aus Abbildung 32 und ihre Visualisierung in Abbildung 33 betrachten, so können wir für den Aufruf des mit dem Zeichen „~“ und darauffolgend den Klassennamen syntaktisch bezeichneten Destruktor für ein Objekt vom Typ *D* die Anweisungsfolge *delete d*, *delete b*, *delete c*, *delete a* feststellen. Das ist völlig analog zur Ausführungsreihenfolge von share-Blöcken, wie sie in Abschnitt 3.3.4 dargestellt wurde, mit Ausnahmne dessen, dass hier genau entgegen der durch *super* induzierten partiellen Ordnung vorgegangen wird. Tatsächlich können wir mit share-Blöcken genau den gleichen Effekt erzielen, wenn wir für dieses Beispiel jeweils den kompletten Methodenrumpf der Destruktoren in einen share-Block packen[12]. Konstruktoren in C++ funktionieren ganz ähnlich, wobei die Abarbeitungsreihenfolge hier allerdings wieder bei den Wurzelklassen beginnt und in die spezialisierteren Klassen fortschreitet. Weil Konstruktoren überladen und über Parameterlisten verfügen können, stellt C++ noch relativ komplizierte syntaktische Mittel zur Verfügung, um in Konstruktoren abgeleiteter Klassen ausgesuchte Konstruktoren der Oberklassen aufrufen zu können. Auf diese Details wollen wir jedoch an dieser Stelle nicht eingehen. Da Konstruktoren im Gegensatz zu Destruktoren keine dynamisch gebundenen Methoden sind und außerdem mit völlig verschiedenen Signaturen ausgestattet sein können, kann auch das Konzept der Implementierungs-Invarianten hier nicht den in C++ eingebauten Mechanismuns ersetzen. Es ist aber leicht möglich, in CPL innerhalb von Konstruktor-Rümpfen dynamisch gebundene Initialisierungsfunktionen mit einheitlicher Signatur aufzurufen und diese mit passenden share-Blöcken auszustatten.

Konstruktoren und Destruktoren in C++ können als Implementierungs-Invarainten bezeichnet werden, da sie genau wie share-Blöcke auf jeden Fall aufgerufen werden, wenn der Methodendispatcher eine Methodenmplementierung aussucht. Es sollte aber herausgestellt werden, dass Konstruktoren und Destruktoren in C++

---

[12] Konsequenterweise bietet CPL im Gegensatz zu C++ fast keine Sonderbehandlung von Destruktoren an – die Ausführungsreihenfolge der share-Blöcke geschieht jedoch wie in C++ von „unten nach oben“, weil bei der Deinitialisierung einen Abarbeitung von „oben nach unten“ im Allgemeinen mögliche Zugriffe auf bereits deinitialisierte Objektteile in den share-Blöcken der abgeleiteten Klassen zur Folge hätte. Der Benutzer con CPL wird üblicherweise durch Klammern des kompletten Rumpfs eines Destruktors einen Effekt äquivalent zu C++ erzielen wollen.

Spezialfälle von Methoden darstellen. Für andere Funktionen besteht keine Möglichkeit dieser automatisch Aggregation von Methodenrümpfen wie in share-Blöcken. Damit entsteht insbesondere auch das in Abschnitt 2.2.4 diskutierte Problem des mehrfachen Aufrufs der von Oberklassen-Implementierung der Methoden, welches in CPL ganz allgemein durch share-Blöcke gelöst wird. Zusätzlich bietet CPL mit share-Blöcken die Möglichkeit, nur bestimmte Teile eines Methodenrumpfes als unbedingt aufzurufen zu markieren.

Einen anderen Weg, den Aufruf von geerbten Methodenimplementierungen zu erzwingen und damit ebenfalls Implementierungs-Invarianten zu modellieren, geht die Sprache BETA [LMN93]. BETA unterstützt im Gegensatz zu C++ und CPL nur Einfachvererbung und kennt damit nicht das Problem aus Abschnitt 2.2.4. BETA hat ein sich von anderen objektorientierten Programmiersprachen stark unterscheidendes Modell des dynamischen Bindens. Grundsätzlich ruft der Methodendispatcher (sofern diese Bezeichnung hier sinnvoll scheinen mag) immer die Implementierung der Klasse auf, in der die Methode zum ersten Mal definiert wurde. Die Möglichkeit, eine Methode zu redefinieren bzw. zu spezialisieren besteht nur im Gebrauch des Schlüsselwortes „inner“. Der Entwickler muss damit in der Implementierung der Wurzelklasse bzgl. der zu implementierenden Methode bereits einen Einstiegspuinkt für mögliche Redefinitionen bzw. Verfeinerungen in abgeleiteten Klassen vorsehen. Gelangt der Kontrollfluss an die mit „inner()“ bezeichnete Stelle, so wird in die Implementierung der nächst abgeleiteten Klasse, die Oberklasse des betreffenden Empfängerobjekts ist, abgestiegen (Abbildung 34).

```
class A {
  void f() {
    doSomethingForA(); inner(); doAnotherThingForA();
  }
  ...
};
class B : public A {
  void f() {
    doSomethingForB(); inner(); doAnotherThingForB();
  }
  ...
};
```

Abbildung 34: Method refinement in à la BETA

Für den Aufruf von *f* auf einem Objekt vom Typ *B* käme damit in Abbildung 34 die Ausführungsfolge *doSomethingForA*(), *doSomethingForB*(), *doAnotherThingForB*() und *doAnotherThingForA*() zustande.

Der von BETA verfolgte Ansatz erzwingt Implementierungs-Invarianten schon durch die von C++ sehr unterschiedliche Semantik des dynamischen Bindens. Tatsächlich

hat der Entwickler in BETA keine Möglichkeit, bei der Redefinition einer Methode in einer abgeleiteten Klasse die Implementierung der Oberklasse zu umgehen. Dies schließt Ableitung und Redefinition zum Zwecke der effizienteren Implementierung bestimmter Methoden aus. Darüber hinaus kann der auf explizitem Aufruf der Redfinition durch *inner* beruhende Ansatz nicht mit Mehfachvererbung fertig werden, wofür er allerdings von den Autoren ganz bewußt nicht gedacht war.

3.3.6.2 Regelansätze und aktive Datenbanken

Erstaunlicherweise ist die Änderungspropagierung für Typsysteme mit multipler Instanziierung wenig untersucht. In der Literatur zu objekterhaltenden Sichten findet sich in keiner Arbeit auch nur ein Hinweis auf ein derartiges Problem. Lösungsansätze kann man demnach allenfalls in anderen Datenbank-Forschungsrichtungen suchen: Event-Condition-Action oder ECA Regeln [BBCR94, GeJa91] bieten die Möglichkeit, Anweisungsfolgen als Reaktionen auf Ereignisse zu spezifizieren. Nach Eintritt eines Ereignisses e wird eine optionale Bedingung b geprüft, und falls diese gilt wird eine Aktion a ausgeführt. Der Entwickler kann also die Verschiebung beispielsweise des *solid* Attributes in der *BauteilTragwerk* Perspektive als Reaktion auf das Ereignis „Aufruf der Methode *Bauteil::verschieben*()“ angeben. share-Blöcke können als eine Art Spezialfall von ECA-Regeln verstanden werden, wobei das Ereignis immer „Aufruf einer Methode $c'.f$ für eine Variable v mit $c' \in ctypes(dtypes(v))$ ist, die Bedingung b den Wert *true* hat und die Aktion a dem Aufruf des share-Blocks $c.f_{share}$ entspricht. Die explizite Integration der share-Blöcke hat sich jedoch aus einer Reihe von Gründen für vorteilhaft erwiesen:

- Die Redefinition einer Methode ist die „natürliche“ Vorgehensweise in der objektorientierten Programmierung. Bei der Implementierung durch ECA Regeln müßte dagegen schon bei der Spezifikation von *Bauteil::verschieben* die Erzeugung eines entsprechenden Ereignisses angegeben werden, auf das in den einzelnen Perspektiven reagiert werden kann.
- share-Blöcke berücksichtigen automatisch notwendige Reihenfolgebeschränkungen durch die Klassenhierarchie. Ein allgemeines ECA-Konzept kann dies nur durch vom Benutzer definierte Regelprioritäten simulieren.
- share-Blöcke sind sehr einfach und sehr effizient zu implementieren (siehe Abschnitt 4.2.4).

Andere Ansätze sind unter dem Schlagwort „constraint-repair Techniken“ bekannt [CFP94, FrPa93]. Für unser Beispiel aus Abbildung 1 würde das bedeuten, dass

dem System Integritätsbedingungen (constraints) bekannt gemacht werden, die die Erkennung einer Inkonsistenz nach der Verschiebung des Bauteils in nur einer Perspektive ermöglichen. Der „Repair“ Teil soll dann in der Lage sein, die Objekte wieder in einen konsistenten Zustand zu überführen. Die Nachteile eines solchen Ansatzes für das hier vorliegende Problem liegen auf der Hand:

1) Die Integritätsbedingungen müssen unter Kenntnis aller beteiligter Perspektiven spezifiziert werden,
2) die Repair Aktion muss algorithmisch beschreibbar sein – im Beispiel bedeutet dies, dass eine automatische Konvertierung von 3D-Geometrie in Systemmittelgeometrie und umgekehrt verfügbar sein muss,
3) die Erkennung einer Inkonsistenz und die anschließende Reparatur ist in den meisten Fällen um Größenordnung ineffizienter als die Vermeidung der Inkonsistenz durch share-Blöcke.

Während 1) und 2) in anderer Form auch in der Diskussion der Änderungspropagierung auf untergeordneten Objekten in Abschnitt 3.3.5 auftraten, ist 3) ein gravierendes Problem. So ist zum Beispiel die Verschiebung eines geometrischen Körpers im Allgemeinen sehr einfach und vor allem sehr effizient durch Matrixmultiplikationen zu implementieren. Das Ausführen der entsprechenden Operation für eine andere geometrische Repräsentation desselben Bauteils ist deshalb um Größenordnungen weniger aufwendig als eine Konversion der kompletten Darstellung.

3.3.7 Zusammenfassung: Änderungspropagierung

Implementierungs-Invarianten wurde entworfen und in [BCR96] erstmals vorgestellt, um die Spezifikation konsistenter updates für Objekte, die direkte Instanzen mehrerer Klassen sind, zu ermöglichen. Dieses Konzept ist – soweit wir wissen – das erste, welches dieses Problem zu lösen versucht. Die Mechanismen sind sehr einfach zu benutzen, da der Entwickler dem Compiler lediglich einen Hinweis geben muss, welche Anweisungen eines Methodenrumpfes relevant im Hinblick auf postconditions der Methode sind. Implementierungs-Invarianten erlauben es Benutzern, update Operationen zu formulieren ohne Objekteigenschaften und Constraints, die im fraglichen Anwedungsmodell nicht bekannt sind, zu kennen oder in Betracht ziehen zu müssen. Darüber hinaus adaptieren existierende Applikationen automatisch Erweiterungen der update Semantik durch neuen Anwendungen.

Ein weiterer Vorteil dieses Konzept ist die einfache Implementierbarkeit, solange ununterbrochen laufende Anwendungen nicht betrachtet werden. Für diese „24/7“ genannten Applikationen entsteht das Problem, dass durch die Integration einer neuen polymorph abgeleiteten Klasse c die update-Semantik für persistente Instanzen von $ctypes(c)$ automatisch um die Menge aller $c.f_{share}$ mit $c.f \in c.F$ zu erweitern ist, sobald eine die Klasse c enthaltende Applikation eine Datenbank öffnet. Derartige Anwendungen müssten allerdings neuen Code dynamisch nachladen, wie es z.B. von Laufzeitsystemen der Sprache Java unterstützt wird. Aber auch fast alle Unix Implementierungen bieten auf einer systemnäheren Ebene die Möglichkeit, zur Laufzeit einer Anwendung eine Bibliothek zu laden und deren Code auszuführen.

Implementierungs-Invarianten sind unabhängig von Integritätskonzepten, wie sie etwa durch verschiedene Datenbank Management Systeme oder auch Programmiersprachen angeboten werden. Tatsächlich sind die Integritätsbedingungen sehr vieler Softwaresysteme nur impizit im Sourcecode vorhanden, d.h. Entwickler haben eigene Begirffe von Konsistenz und schreiben den Code so, dass diese Konsistenz (hoffentlich) niemals verletzt werden kann. Implementierungs-Invarianten ermöglichen es dem Entwickler, Code zu schreiben, der durch die inkrementelle Eigenschaft der share-Blöcke nicht nur die eigenen, anwendungsspezifischen Begriffe von Konsistenz erfüllt, sondern auch die der übrigen Anwendungen, obwohl deren Modelle dem Entwickler nicht bekannt sein müssen.

3.4 Erweiterte Attribute

Für Anwendungen bzw. Gruppen von Anwendungen, die auf gemeinsam genutzte persistente Objekte zugreifen müssen ist es von Vorteil, wenn die Objekte bzw. Klassen, von denen bereits persistente Instanzen in Datenbanken existieren, sich für spezielle Bedürfnisse neuer Anwendungen um zusätzliche Attribute erweitern lassen. Wir haben in Abbildung 25 bereits gesehen, dass die geometrischen Repräsentationen von Bauteilen erst in den Perspektiven *BauteilStatik* und *BauteilTragwerk* spezifiziert wurden. Für solche Attribute stellt sich jedoch die Frage, welche Werte diese annehmen sollen, bevor eine Anwendung das erste mal explizit schreibend darauf zugreuft. Denn anders als Attribute in den nicht polymorph abgeleiteten Klassen werden diese „*erweiterten Attribute*“ nicht innerhalb von Konstruktor-Funktionen initialisiert, da sie zum Zeitpunkt der Übersetzung der Konstruktor-Funktionen noch gar nicht bekannt waren. Der Entwickler muss deshalb

einen Default-Wert für erweiterte Attribute vergeben der bei lesenden Zugriffen verwendet wird, bis der erste Schreibzugriff erfolgt ist.

3.5 Weitere Sprachkonzepte

Die meistverbreitete Programmiersprache für ODBMS ist derzeit C++. Leider bietet weder diese noch andere imperative objektorientierte Sprachen die Möglichkeit, mengenorientierte Operationen über Objektkollektionen deskriptiv zu formulieren. Dies ist im Datenbankkontext jedoch von besonderer Wichtigkeit, da nur bei deskriptiv formulierten Anfragen ein Anfrageoptimierer eingesetzt werden kann. Performanzsteigernde Maßnahmen wie das Hinzufügen von Indexen verbessern dann das Laufzeitverhalten der Anwendung ohne dass die Anfragen selbst verändert werden müssen.

Um diese programmiersprachliche Beschränkung zu umgehen, bieten einige ODBMS Anfragefunktionen an, die als Argument einen „query string“ erhalten. Dieser String wird zur Laufzeit analysiert, gegebenenfalls in einen Ausführungsplan umgesetzt und schließlich ausgeführt.

```
Set<Bauteil*>* collection = ...;
Set<Bauteil*>* query_result =
  collection->query(“material.name == \”Beton\””);
```

Abbildung 35: Anfragen mit Hilfe von zur Laufzeit geparsten query strings

Diese Technik birgt jedoch eine Reihe von Nachteilen in sich:

- syntaktische Fehler in Anfragen werden erst zur Laufzeit erkannt – der Vorteil von typsicheren, kompilierten Sprachen wird also an dieser Stelle aufgegeben;
- die Benutzung von Methoden, lokalen Variablen etc. in Anfragen ist in den meisten Systemen nur stark eingeschränkt möglich; in vielen Fällen sind in Anfragen lediglich Zugriffe auf Attribute und Konstanten erlaubt.

Den letzten Punkt illustriert Abbildung 36. Beim Versuch, eine Anfrage an eine Kollektion von Bauteilen zu formulieren, in der nach Bauteilen aus einem Material schwerer als ein gegebenes Referenzmaterial *m* selektiert wird, stößt die Methode des Analysierens von Anfragen zur Laufzeit an ihre Grenze. Das Laufzeitsystem eines ODBMS muss über Metainformationen verfügen, die es Attribute und eventuell Methoden von Klassen erkennen lassen. Ein Anfrage-Parser kann dementsprechend den Aufruf *material->schwerer_als()* als korrekt erkennen, weil der Elementtyp der Kollektion über ein Attribut namens *material* verfügt, das seinerseits vom Typ *Material* ist welcher über eine Methode *schwerer_als* besitzt.

```
Material* m = ...;
Set<Bauteil*>* collection = ...;
Set<Bauteil*>* query_result = collection(“material.schwerer_als(m)”);
```

Abbildung 36: Methodenaufrufe und lokale Variablen in query strings

Der tatsächliche Aufruf der Methode *schwerer_als* kann jedoch zur Laufzeit nicht mehr erreicht werden, weil die meisten objektorientierten Sprachen keine Möglichkeit anbieten, zur Laufzeit über Metainformationen Symbole – in diesem Fall Methodenrümpfe – zu binden[13].

CPL bearbeitet dagegen Anfragen, die aus beliebig komplexen boolschen Ausdrücken bestehen. Methodenaufrufe und Zugriffe auf alle sichtbaren lokalen Variablen werden ebenfalls unterstützt. Da CPL von C++ das Konzept der „const-correctness“ übernommen hat, kann der Compiler zusätzlich Vorkehrungen treffen, die mögliche Seiteneffekte in Anfragen weitgehend verhindern. In Anfragen sind deshalb nur Methodenaufrufe zulässig, bei denen alle Referenzparamter durch das Schlüsselwort `const` qualifiziert sind. Zusätzlich muss die aufgerufene Methode selbst als `const` spezifiziert sein. Neben Anfragen bietet CPL auch syntaktische Konstrukte für die Beschreibung von Indexen.

Zuverlässigkeit von Anwendungen und Datenkonsistenz ist für Datenbankanwendungen noch wichtiger als für konventionelle objektorientierte Softwaresysteme, da nicht nur ein Lauf einer Anwendung, sondern beliebig viele Läufe mehrerer Anwendungen über u.U. lange Zeiträume hinweg auf persistente Objekte zugreifen. Um insbesondere die Überprüfung der Datenkonsistenz für den Entwickler deskriptiv spezifizierbar zu machen, wurde das DBC Paradigma an Perspektiven angepasst und in das Datenmodell integriert (vgl. Abschnitt 3.3.1.2). Sehr mächtige Möglichkeiten ergeben sich durch die Kombination von Anfragen und DBC: Sei eine Klasse `Bauteil`, welche Elemente eines Tragsystems beschreibt, gegeben. Alle Instanzen von `Bauteil` sollen im `static`[14] Attribut `extent` abgelegt sein. Weiterhin habe die Klasse `Bauteil` ein Attribut `id` vom Typ `int`, das für jede

[13] Eine Ausnahme bietet Java: hier wird über das Reflection-API dem Entwickler explizit ermöglicht, Methoden als Laufzeit-Objekte zu ermitteln und diese auch mit Parameterlisten zu versorgen und auszuführen. Damit sind alle Voraussetzung für eine Implementierung des Query-Subsystems in Java gegeben. Der Nachteil der Erkennung von Syntax Fehlern erst zur Laufzeit bleibt aber bestehen.

[14] „static“ ist ein von C++ übernommenes Schlüsselwort und bezeichnet Attribute, deren Wert nur einmal pro Klasse existiert.

Instanz eindeutig sein soll. Diese Bedingungen an gültige Zustände von Instanzen der Klasse `Bauteil` lassen sich sehr leicht als Klasseninvariante formulieren:

```
void
Bauteil::invariant() const
{
 CHECK(extent->contains(this));
 CHECK(extent->select<b>(b->id == id)->cardinality() == 1);
}
```

Abbildung 37: Invariante der Klasse `Bauteil`[15]

Invarianten bestehen i. A. aus einer Reihe von CHECK-Anweisungen, die boolsche Ausdrücke als Argument erhalten. Schlägt eine dieser Anweisungen fehl, so wird das Programm mit einer entsprechenden Fehlermeldung abgebrochen. Der Compiler sorgt für den automatischen Aufruf von Invarianten bei Eintritt in und Austritt aus Methoden, die nicht aus anderen Methoden mit dem selben `this`-Argument aufgerufen werden. Ist der Entwickler vom fehlerlosen Funktionieren des Codes überzeugt, so kann er sämtliche runtime-checks durch einen Compiler-switch deaktivieren, um Laufzeit einzusparen.

[15] Die Syntax `select<k>` vereinbart eine Variable `b` vom gleichen Typ wie der Elementtyp der Kollektion, auf die `select` angewendet wird. Innerhalb des Prädikates in den darauffolgenden runden Klammern kann `b` benutzt werden, um auf das gerade zu testende Element der Kollektion zuzugreifen.

4 Implementierung

Dieses Kapitel diskutiert Implementierungstechniken für Perspektiven in einer streng getypten, objektorientierten Programmiersprache. Dabei stellt Abschnitt 4.2 zunächst eine Realisierung vor, die weitgehend ohne globales Wissen über eine zu kompilierende Applikation auskommt. Dadurch werden einzelne Quelldateien getrennt übersetzbar, was für die Programmentwicklung wegen der schnellen turn-around Zeiten ein sehr wichtiger Faktor im Hinblick auf Produktivität ist. Änderungen im Programmcode erfordern damit oft nur die Neuübersetzung einer einzigen Quellcode-Datei.

Abschnitt 4.3 dagegen beschäftigt sich mit Optimierungstechniken in der für die Objektorientierung typischen Aufwand des dynamischen Bindens zu verringern oder gar ganz zu vermeiden suchen. Offensichtlich wird mit Techniken, die dynamisches Binden in vielen Fällen vermeiden auch der für Perspektiven spezifische overhead für die Auswahl der aktuellen Perspektive eingespart. Für die Frage, ob Perspektiven jedoch mit sehr geringem Laufzeitaufwand gegenüber klassischen objektorientierten Sprachen implementiert werden können ist weit wichtiger, wie oft zwar dynamisches Binden, nicht jedoch die Auswahl der Perspektive notwendig ist. Deshalb beschäftigt sich ein Großteil des Abschnittes 4.3 mit dieser Frage, denn die Qualität einer Implementierung muss sich sicherlich im Vergleich zu traditionellen Sprachen und deren Laufzeitsystemen messen lassen.

4.1 Allgemeines

Das im Rahmen dieser Arbeit für das DFG-Schwerpunkt-Projekt „Objektorientierung in Planung und Konstruktion“ entwickelte System CEMENT (**C**ivil **E**ngineering **M**odeling Environm**ent**) mit seiner Programmiersprache CPL (**C**ement **P**rogramming **L**anguage) [BBCR94, BBCR95, CBBB98] besteht aus einem Compiler, einer Standardbibliothek mit Basisfunktionalität für Ein-/Ausgabe, numerische Aufgaben, Zeichnkettenverarbeitung etc. sowie einem interaktiven Schema- und Datenbrowser [BeFl96], der als Debugging Unterstützung einen interaktiven Zugriff auf persistente CPL-Objekte sowie Methodenaufrufe auf diesen Objekten ermöglicht. Der Rest dieser Arbeit beschäftigt sich jedoch mit dem Compiler, der aus in CPL geschriebenen Quelldateien C++ Code generiert, welcher sich für die Persistenz u.a. auf das objektorientierte Datenbanksystem ObjectStore [ODI99] abstützt.

Die in diesem Kapitel vorgestellten Codebeispiele orientieren sich sehr stark an der aktuellen Implementierung des Compilers, sind jedoch aus Gründen der Lesbarkeit

oft vereinfacht[16]. Tatsächlich nutzt der generierte Code von C++ weder dynamisches Binden noch Vererbung. Zugriffe auf Attribute und auch Methodenaufrufe werden grundsätzlich durch sogenannte *static* Funktionen implementiert[17]. Diese erhalten als erstes Argument immer ein Objekt vom Typ *Pointer,* der in der CPL- Implementierung als generische Objektreferenz auf Objekte beliebigen Typs verwendet wird. Die Klasse *Pointer* kapselt neben der Referenz auch den in Abschnitt 3.2.1 eingeführten "dynamischen Kontext" einer Referenz, d.h. den gespeicherten, referenzspezifischen dynamischen Typ. Als zweiten Parameter erhalten die meisten generierten Funktionen den statischen Typ *stype.* Der Wert dieses Parameters ist entweder der direkt vom Compiler zugewiesene statische Typ eine Ausdrucks, oder die durch den Algorithmus *type-dispatch* (vgl. Abbildung 21) berechnete Perspektive. So wird aus den Zeilen

```
auto Bauteil* b = b->zeichnen(};
```

folgender Code generiert:

```
Pointer b = ...;
Bauteil::zeichnen-dispatch(b, 0, true);
```

Dabei sei 0 die dem Typ Bauteil zugeordnete ID, der Parameter „true“ bewirkt den Aufruf der Implementierungs-Invariante, falls definiert (vgl. Abschnitt 4.1). Innerhalb eines Methodenrumpfes – d.h. nach dem dynamischen Binden – beschreibt der Typ- Parameter genau den aktuellen Typ bzw. die aktuelle Perspektive des Objekts. Attributzugriffe – in CPL nur in der Klasse, in der das Attribut definiert wird erlaubt – müssen die generische Objektreferenz vom Typ *Pointer* auf den korrekten Typ der Zielsprache C++ umwandeln, um das Attribut effizient lesen bzw. schreiben zu können. Abbildung 38 zeigt den generierten Code, der sich im Falle optimierender Übersetzung natürlich problemlos als inline-Funktionen oder äquivalent als

---

[16] Zum Beispiel abstrahieren wir hier vollständig von Fragen des *Name-Manglings,* der Sichtbarkeit bzw. Zugriffsberechtigung (`public`, `protected` und `private` in C++) und in vielen Fällen ignorieren wir das C++ Typsystem – der tatsächlich generierte Code muss an vielen Stellen explizite Typumwandlungen *(casts)* nutzen um dem backend C++-Compiler das CPL-Typsystem schmackhaft zu machen.

[17] `static` bedeutet in C++, Java und auch CPL, dass eine Methode kein „*this*“ als impliziten Parameter bekommt. Derartige Funktionen sind also nicht Objekten zugeordnet, können dementsprechend auch nicht direkt auf Attribute eines Objektes zugreifen. Natürlich sind `static` Funktionen, die ein Objekt explizit als Parameter erhalten, bis auf dynamisches Binden äquivalent zu non-`static` Methoden.

expandierter Code pro Attributzugriff realisieren lässt. In weiteren Codebeispielen in dieser Arbeit werden wir Attributzugriffe und Methodenaufrufe oft vereinfacht darstellen und z.B. die Typumwandlungsoperationen `static_cast` etc. aus Gründen der besseren Lesbarkeit weglassen.

```
Material* Bauteil::material(Pointer p)
{
  return static_cast<Bauteil*>(p.dereference())->material;
}
void Bauteil::material(Pointer p, pointer m)
{
  static_cast<Bauteil*>(p.dereference())->material = m;
}
```

Abbildung 38: Attributzugriffe

4.2 Nicht-optimierende Übersetzung

Dieser Abschnitt diskutiert im Wesentlichen die Aspekte der Implementierung, die nicht bereits durch die Abschnitte 3.2, 3.3 und 3.4 naheliegend sind. Dabei ist insbesondere zu untersuchen, welche für die Ausführung einer Applikation notwendigen Teile zur Linkzeit, also unter Zuhilfename globalen Wissens, generiert werden müssen.

4.2.1 Typinformation zur Laufzeit

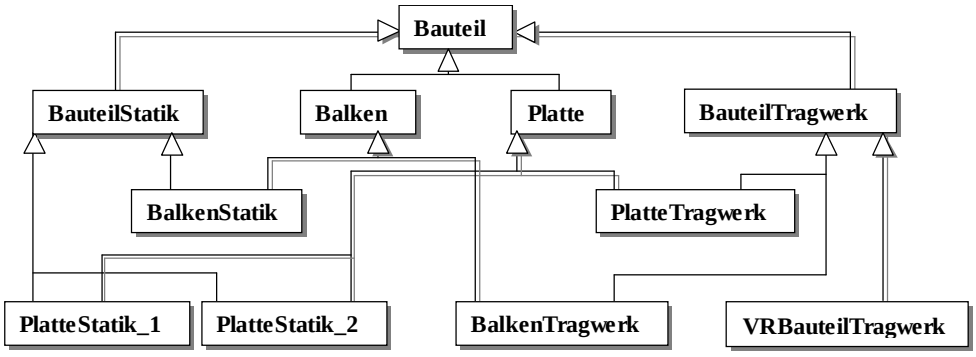

Abbildung 39

In Abbildung 22 sind für die Berechnung der Perspektive die Funktionen *is_subtype_of* und *select_type* notwendig. Um *dispatch,* wie gefordert, in konstanter Zeit ausführen zu können muss auch die Ausführbarkeit dieser beiden Funktionen in konstanter Zeit gewährleistet sein.

		Bauteil	**Balken**	**Platte**	**BauteilStatik**	**BalkenStatik**	**PlatteStatik**	**BauteilTragwerk**	**BalkenTragwerk**	**PlatteTragwerk**	**VRBauteilTragwerk**
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Bauteil	**0**	*s p 0*	*s 1*	*s 2*	*s p 3*	*s 4*	*s 5*	*s p 6*	*s 7*	*s 8*	*s p 9*
Balken	**1**		*s p 1*			*s p 4*			*s p 7*		
Platte	**2**			*s p 2*			*s p 5*			*s p 8*	
BauteilStatik	**3**	*p 3*	*4*	*5*	*s p 3*	*s 4*	*s 5*	*p 3*	*4*	*5*	*p 3*
BalkenStatik	**4**		*p 4*			*s p 4*			*p 4*		
PlatteStatik	**5**			*p 5*			*s p 5*			*p 5*	
BauteilTragwerk	**6**	*p 6*	*7*	*8*	*p 6*	*7*	*8*	*s p 6*	*s 7*	*s p 8*	*p 6*
BalkenTragwerk	**7**		*p 7*			*p 7*			*s p 7*		
PlatteTragwerk	**8**			*p 8*			*p 8*			*s p 8*	
VRBauteilTragwerk	**9**	*p 9*	*9*	*9*	*p 9*	*9*	*9*	*p 9*	*9*	*9*	*s p 9*

Tabelle 1: transitive und reflexive Hülle des Vererbungsgraphen (*s*), transitive, reflexive und symmetrische Hülle der polymorphen Ableitungen (*p*), Typauswahl bei *select_type* (*Ziffern*)

Typen werden in unserer Implementierung zur Laufzeit durch eindeutige Ganzzahl-Werte identifiziert. Eine sehr effiziente Implementierung von *is_subtype_of* besteht deshalb aus einem lookup in einer die reflexive und transitive Hülle des Vererbungsgraphen repräsentierenden Tabelle. Für jeden Eintrag wird lediglich ein bit benötigt, wie in Tabelle 1 durch den eingetragenen bzw. fehlenden Buchstaben *s* dargestellt. Es ist offensichtlich, dass eine derartige Matrix erst zur Linkzeit der Applikation, also einem Zeitpunkt, an dem alle beteiligten Klassen bekannt sind, generiert werden kann. Gleiches gilt für die *pcc*-Informationen, die in Tabelle 1 durch den Buchstaben *p* gekennzeichnet sind.

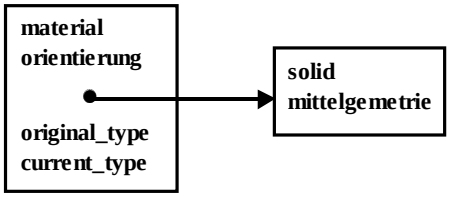

Abbildung 40: Typinfomation *original_type* und *current_type* im Objektlayout

Die Zahlen in der jeweils dritten Spalte von Tabelle 1 stellen den Typ dar, der von *select_type* berechnet wird. Wir können die Wirkung von *select_type* am besten durch die kleinen Code-Beispiele in Abbildung 41 illustrieren.

```
auto Bauteil* b = new PlatteStatik::create();                  // (1)
auto BauteilTragwerk* bt = view_cast<BauteilTragwerk*>(b);     // (2)
auto PlatteTragwerk* pt = view_cast<PlatteTragwerk*>(b);       // (3)
pt = dynamic_cast<PlatteTragwerk*>(b);                              //
(4)
auto VRBauteilTragwerk* vrbt =
     view_cast<VRBauteilTragwerk*>(pt);                             //
(5)
instance_behavior<VRBauteilTragwerk*>(b);                           //
(6)
auto BauteilStatik* bs = view_cast<BauteilStatik*>(vrbt);      // (7)
auto BalkenTragwerk* bt =
     dynamic_cast<BalkenTragwerk*>(vrbt>);                          //
(8)
```

Abbildung 41: *select_type*, *original_type*, *current_type*

(1) Die Erzeugung eines Objekts vom Typ *PlatteStatik* (*pcc* = {*Platte*, *PlatteStatik*, *PlatteTragwerk*}) setzt sowohl *original_type* als auch *current_type* in diesem Objekt jeweils auf *PlatteStatik*. Im pointer *b* steht *current_type* ebenfalls auf *PlatteStatik*, während der Compiler der Variablen selbst den *stype Bauteil* zuordnet (zur Laufzeit nicht gespeichert). Methodenaufrufe über die Variable *b* hätten zu diesem Zeitpunkt die Ausführungen der Implementierungen von *PlatteStatik* zur Folge.

(2) *original_type* und *current_type* des Objekts bleiben unverändert, der *current_type* der Variablen *bt* wird jedoch als Ergebnis von *select_type* auf *PlatteTragwerk* gesetzt. Methodenaufrufe über *bt* führen Implementierungen von *PlatteTragwerk* aus.

(3) Diese `view_cast`-Anweisung führt zu einem Typfehler, da $PlatteTragwerk \notin ctypes(Bauteil)$.

(4) Im Gegensatz zu `view_cast` ist `dynamic_cast` in diesem Fall zulässig. `dynamic_cast` gibt, wie in C++ ebenso, *nil* zurück falls der Typ des Objekts, das durch das Argument von `dynamic_cast` referenziert wird, nicht kompatibel zu dem in spitzen Klammern angegeben Typ ist. Mit anderen Worten: $dynamic\_cast<type>(object) \mathrel{!}= nil \Leftrightarrow type \in ctypes(original\_type(object))$. Methodenaufrufe über *pt* haben immer Ausführungen der Implementierungen von *PlatteTragwerk* zur Folge.

(5) Für die Variable *vrbt* sind nach Ausführung dieses korrekten Ausdrucks *current_type* und *stype VRBauteilTragwerk*. Methodenaufrufe resolvieren daher immer Implementierungen aus *VRBauteilTragwerk*. Unterschiede in der Bauteil-Hierarchie, d.h. verschiedene Semantiken für Platten und Balken werden damit „vergessen“. Ausnahmen sind Methoden, die in der Oberklasse *Bauteil* definiert und in den nicht polymorph abgeleiteten Klassen *Platte* und *Balken* redefiniert werden, nicht aber in den polymorph abgeleiteten Klassen *VRBauteilTragwerk* und *BauteilTragwerk* (vgl. Abschnitt 4.2.2).

(6) Mit `instance_behavior<type>(object)` wird dem mit `object` referenzierten Objekt das Verhalten des Typs `type` zugewiesen, d.h. *b* zeigt auf ein Objekt mit *current_type VRBauteilTragwerk*. Durch diese Anweisung werden auch zukünftige Methodenaufrufe über die Variable *b* mit *stype*(*b*) = *Bauteil* die Implementierungen von *VRBauteilTragwerk* dispatchen.

(7) Situationen dieser Art sind der Grund für die Speicherung des *original_type* in Objekten. Wäre nur die *current_type* Information im Objekt vorhanden, so könnte die Umwandlung des Typs nicht in *PlatteStatik* erfolgen, da das Wissen über diesen Typ mit der `instance_behavior` Operation aus (6) verlorengegangen ist. Der gespeicherte *original_type* ermöglicht jedoch problemlos die Ausfürhung von *select_type*, in diesem Fall *PlatteStatik* ergibt.

(8) diese Anweisung liefert *nil* als Ergebnis, weil $BalkenTragwerk \notin ctypes(PlatteStatik)$

4.2.2 Dispatch-Funktionen

Zur Realisierung des dynamischen Bindens in objektorientierten Sprachen gibt es eine Vielzahl von Ansätzen, von denen einige im Abschnitt 4.3 diskutiert werden. Unsere Implementierung verwendet auch im nicht-optimierenden Modus *dispatch*-Funktionen, die zur Linkzeit generiert werden. Abbildung 42 zeigt die dispatch-Funktion für die Klassenhierarchie aus Abbildung 39. Wir nehmen dabei an, dass zeichnen jeweils in *BauteilStatik*, *BauteilTragwerk* und *VRBauteilTragwerk* redefiniert ist, nicht jedoch in den übrigen Klassen. Man beachte die Eigenschaft von `switch`-statements in C/C++, dass jeweils alle aufeinanderfolgenden `case` Marken durchlaufen werden begonnen bei der Marke, deren Wert dem des `switch`-Ausdrucks entspricht und endend beim ersten `break`-statement. Gibt die Funktion *dispatch* (siehe Abbildung 22) also den Wert 0 für *Bauteil* zurück, so wird dennoch *Bauteil::zeichnen* ausgeführt. genauso, als wenn 1 (*Balken*) oder 2 (*Platte*) berechnet wurde.

```
void Bauteil::zeichnen_dispatch(pointer p, type stype)
{
  type t = dispatch(p, stype);
  switch (t) {
    case 0:
    case 1:
    case 2: Bauteil::zeichnen(p, t);              break;
    case 3:
    case 4:
    case 5: BauteilStatik::zeichnen(p, t);    break;
    case 6:
    case 7:
    case 8: BauteilTragwerk::zeichnen(p, t);  break;
    case 9: VRBauteilTragwerk::zeichnen(p, t); break;
  }
}
```

Abbildung 42: dispatch-Funktion für das dynamische Binden von *zeichnen* in der Klassenhierarchie aus Abbildung 39

In dieser Variante der Codegenerierung kann der Compiler für jeden Aufruf der Art

```
auto Bauteil* p = ...;
p->zeichnen();
```

immer

```
Bauteil* p = ...;
Bauteil::zeichnen_dispatch(p, 0, callShareBlocks);
```

erzeugen, wobei der zweite Parameter „0“ den vom Compiler berechneten Wert des statischen Typs *stype* eines Ausdrucks ist. *callShareBlocks* wird im folgenden Abschnitt erläutert.

Dies Art der Realisierung für dynamisches Binden unterscheidet sich sehr von verbreiteten Techniken, wie sie üblicherweise in C++ Implementierungen, aber auch in den meisten Eiffel-Compilern eingesetzt werden. Diese basieren auf Funktionstabellen, in denen die eigentlichen Implementierungen als Zeiger abgelegt sind. Funktionstabellen, die pro Klasse angelegt werden und durch einen in jeder Instanz der Klasse angelegten Zeiger auf diese Tabelle referenziert werden, können ohne globales Wissen zur Übersetzungszeit einzelner Dateien realisiert werden. Unsere dispatch-Funktionen dagegen brauchen Wissen über die Gesamtheit aller in der Applikation beteiligten Klassen. Wir haben diesen Ansatz jedoch gewählt, weil für die in Abschnitt 4.3 vorgestellte Optimierung besser geeignet ist und wir damit die Integration zweier grundsätzlich unterschiedlicher Implementierungstechniken vermeiden konnten.

4.2.3 Variante Attribute

Variante Attribute sind Attribute, deren Typ sich in einer abgeleiteten Klasse zu einem anderen, demselben *pcc* angehörenden Typ ändert. Ein Beispiel findet sich in Abschnitt 2: Ein Gebäude Klasse, welche das Zeichnen des Gebäudes durch Iteration über seine Bauteile und dort den jeweiligen Aufruf von zeichnen implementiert. Die Klasse und zwei in diesem Zusammenhang sinnvolle Perspektiven sind kurz in Abbildung 43 skizziert.

```
class Gebaeude {
public:
  constructor create();
  void zeichnen() const;
  ...
state:
  List<Bauteil>* bauteile;
};

class GebaeudeStatik perspective_of Gebaeude {
state:
  List<BauteilStatik>* bauteile;
};

class GebaeudeTragwerk perspective_of Gebaeude {
state:
  List<BauteilTragwerk>* bauteile;
};

void Gebaeude::zeichnen() const
{
  bauteile->apply<b>(b->zeichnen());
}
```

Abbildung 43: Perspektiven in der Aggregationshierarchie

In diesem Beispiel erscheint es nicht sinnvoll, von Entwicklern Redefinitionen der Methode *Gebaeude::zeichnen* zu verlangen, da ihr Rumpf unverändert aus der Iteration über die Bauteile besteht. Trotzdem ist intuitiv klar, dass die Bauteile eines Gebäudes unbedingt als Bauteile aus der Statik-Perspektive zu zeichnen sind, wenn das Gebäude unter der Statik Perspektive betrachtet bzw. genutzt wird.

```
void Gebaeude::zeichnen(Pointer p, Type stype, bool callShareSection)
{
  Cursor c(Gebaeude::bauteile(p, stype));
  for (c.tirst(); c.more(); c.next()) {
    Bauteil::zeichnen_dispatch(c.retrieve(), Bauteil_id, true);
  }
}

Pointer p = ...;
Gebaeude::zeichnen(p, GebaeudeStatik_id, true);
```

Abbildung 44: nicht korrekte Implementierung des Zugriffs auf redefinierte Attribute

Abbildung 44 zeigt eine Implementierung, welche diese besonderen Eigenschaften von redefinierten Attributen nicht berücksichtigt, sondern die bisher schon bekannte Codegenerierungstechnik nutzt. Wie leicht zu sehen ist, wird als statischer Typ beim Aufruf von *zeichnen* für die Bauteile *Bauteil_id,* die Typkennung der Klasse *Bauteil,* übergeben. Damit kann beim dynamischen Binden dieser Aufrufe die aktuelle Perspektive des Gebäudes nicht berücksichtigt werden. Gesucht ist also eine Implementierung, die den statischen Typ des redefinierbaren Attributes abhängig von dynamischen Typ von „*this*“ berechnet.

```
void Gebaeude::zeichnen(Pointer p, Type stype, bool callShareSection)
{
  Type t = type_dispatch(p, stype);
  Cursor c(Gebaeude::bauteile());
  for (c.first(); c.more(); c.next()) {
    Bauteil::
    zeichnen_dispatch(c.retrieve().view_cast(Gebaeude::
                                             type_of_bauteile(t)),
                      Gebaeude::type_of_bauteile(t), true);
  }
}
Type Gebaeude::type_of_bauteile(Type perspective)
{
  switch(perspective) {
  case Gebaeude id:         return Bauteil_id;
  case GebaeudeStatik id:   return BauteilStatik_id;
  case GebaeudeTragwerk_id: return BauteilTragwerk_id;
  }
}
```

Abbildung 45: Zugriff auf variante Attribute

Abbildung 45 zeigt die im CPL-Compiler implementierte Lösung: um ein korrektes Verhalten bei im Methodenrumpf auftretenden Attributzugriffen zu gewährleisten muss zunächst die aktuelle Perspektive *t* von „*this*“ berechnet werden. Diese wird dann als Argument einer Funktion benutzt, welche die Perspektive eines Attributs abhängig von *t* berechnet. Diese Funktion *Gebaeude::type_of_bauteile* wird zur

Linkzeit generiert, weil sie globales Wissen über die Perspektiven von *Gebaeude* bzw. den Redefinitionen der Attribute von *Gebaeude* nutzen muss.

4.2.4 Implementierungs-Invarianten

Auch bei der Realisierung der Implementierungs-Invarianten greift unsere Implementierung auf globales Wissen zurück. share-Blöcke werden als eigene Funktionen realisiert, die beim Eintritt in dynamisch gebundene Methoden verkettet aufgerufen werden. Diese Verkettung, in Abbildung 46 durch die Funktion *BauteilStatik::verschieben_shareTrigger* illustriert, kann in dieser Form erst zur Linkzeit erzeugt werden. Die übrigen in Abbildung 46 gezeigten Funktionen werden zur Übersetzungszeit der entsprechenden Quelldateien generiert.

```
void
Bauteil::verschieben_share(pointer p, type stype,
                           double x, double y, double z) {
  Orientierung::
    transformiere(Bauteil::orientierung(p, stype, x, y, z));
}
void
BauteilStatik::verschieben_share(pointer p, type stype,
                                 double x, double y, double z) {
  MittelGeometrie::
    transformiere(BauteilStatik::mittelgeometrie(p, stype, x, y, z));
}
void
BauteilTragwerk::verschieben_share(pointer p, type stype,
                                   double x, double y, double z) {
  Solid::transformiere(BauteilTragwerk::solid(p, stype, x, y, z));
}
...
void
BauteilStatik::verschieben_shareTrigger(pointer p, type stype,
                                        double x, double y, double z)
{
  Bauteil::verschieben_share(p, stype, x, y, z);
  BauteilStatik::verschieben_share(p, stype, x, y, z);
  BauteilTragwerk::verschieben_share(p, stype, x, y, z);
}
void
BauteilStatik::verschieben(pointer p, type stype, bool
callShareBlocks) {
  if (callShareBlocks) {
       BauteilStatik::verschieben_shareTrigger(p, stype, x, y, z);
  }
  // original non-share code der Methode
  BauteilStatik::zeichnen(p, stype, true);
}
```

Abbildung 46: Realisierung von share-Blöcken

In der Funktion *BauteilStatik::verschieben* aus Abbildung 46 sehen wir einen zusätzlichen Parameter *callShareBlocks*, dessen Bedeutung darin liegt, die in Abschnitt 3.3.4 erwähnte mögliche Mehrfachausführung der share-Blöcke durch explizite Aufrufe von Oberklassen-Implementierungen der Methode vermeiden. Abbildung 47 zeigt den CPL-Quellcode für ein solches Szenario, wobei wir davon ausgehen, dass der Entwickler in der Redefinition *BauteilStatik::verschieben* die Implementierung *Bauteil::verschieben* explizit aufruft.

```
void BauteilStatik::verschieben(double x, double y, double z)
{
  share {
    mittelgeomettrie->transformiere(x, y, z);
  }
  Bauteil::verschieben(x, y, z);
  zeichnen();
}
```

Abbildung 47: Aufruf der Oberklassen-Implementierung im Quellcode

Abbildung 48 zeigt, wie der Compiler sehr einfach diesen Fall behandeln kann. Bei einem qualifizierten Aufruf der geerbten Implementierung aus der Redefinition heraus wird der Parameter *callShareBlock* mit *false* belegt und damit die mehrfache Ausführung der share-Block Folgen vermieden.

```
void BauteilStatik::verschieben(pointer p, type stype,
                                bool callShareBlocks,
                                double x, double y, double z) {
  if (callShareBlocks) {
    BauteilStatik::verschieben_shareTrigger(p, stype, x, y, z);
  }
  Bauteil::verschieben(p, stype, false, x, y, z);
  Bauteil::zeichnen(p, stype, true);
}
```

Abbildung 48: generierter Code zu Abbildung 47

4.2.5 Erweiterte Attribute

Für die Realisierung erweiterter Attribute gibt es eine Vielzahl von Möglichkeiten, von denen an dieser Stelle zwei kurz diskutiert werden sollen. Voraussetzung für eine sinnvolle Implementierungstechnik ist die Eigenschaft, dass die Speicherung zusätzlicher, in polymorph abgeleiteten Klassen definierter Attribute das vom Compiler bestimmte physische Layout der polymorphen Basisklasse nicht verändert. Diese Forderung unterstützt einerseits die einfache Implementierbarkeit von getrennter Übersetzung einzelner Quelldateien, andererseits vermeidet sie Schema-Evolution bzw. eine physische Migration persistenter Objekte in dem Moment, wo eine neue Perspektive mit zusätzlichen Attributen definiert wird.

Getrennte Übersetzbarkeit würde durch das Fehlen dieser Eigenschaft erschwert, weil der Compiler zur Übersetzungszeit einer Klasse keine Annahmen über das physische Layout treffen könnte. Zugriffe auf Attribute etc. könnten deshalb entweder zur Linkzeit oder gar erst zur Laufzeit der Anwendung realisiert werden. Letzteres wäre aus Performance-Gründen nicht tragbar, denn alle anderen streng getypten Sprachen nutzen das Wissen über die Objektstruktur zur Übersetzungszeit

aus um optimale Zugriffszeiten auf Attribute zu ermöglichen[18]. CPL darf hier nicht weniger effizient sein.

Für eine objektorientierte Datenbank, auf der die Implementierung einer persistenten Programmiersprache wie CPL aufbaut, ist die physische Struktur von Objekten von großer Bedeutung. Manche Systeme, wie das kommerzielle System ObjectStore [ODI99] oder der an der Universität Texas in Austin, USA entwickelte Prototyp Texas Persistent Store [SKW92, WiKa92] übernehmen auch für die Darstellung von Objekten auf dem Hintergrundspeicher das Layout, welches der Compiler vergeben hat. Eine derartige Vorgehensweise hat den Vorteil, dass beim Einladen eines persistenten Objekts[19] in den Hauptspeicher keine zeitaufwendige Transformation zwischen Repräsentationen vorgenommen werden muss.

Wichtige Qualitätsmerkmale für mögliche Implementierungstechniken sind einerseits die Effizienz des Zugriffs vom eigentlichen Objekt zu seinen erweiterten Attributen, andererseits aber auch der Speicherbedarf. Letzteres interessiert besonders für die Fälle, in denen für viele Objekte erweiterte Attribute gar nicht angelegt werden müssen weil keine Anwendung ihre Werte jemals schreibt. In diesem Fall genügt für das Lesen der vom Entwickler angegebene Default-Wert. Lazy Evaluation – d.h. in diesem Fall Anlegen erweiterter Attribute erst beim ersten schreibenden Zugriff einer Anwendung – ist deshalb eine Technik, die für die Implementierung eingesetzt werden sollte.

---

[18] Eine Ausnahme bildet Java, Das Java-Kompilat – der sogenannte „Bytecode“ – beinhaltet keinerlei Annahmen über das Objektlayout. Diese Vorgehensweise erleichtert Implementierungen der Java Virtual Machine, die eine Abbildung des Bytecodes auf die konkrete Maschine vornimmt. Da im Bytecode keinerlei physischen Annahmen versteckt sind, hat der Implementierer der Virtual Machine die größten Freiheiten um der speziellen Hardware Architektur seiner Zielplattform gerecht zu werden. Mittlerweile sind jedoch von Native-Code Compilern für Java auf dem Markt, die selbstverständlich genau wie C++ Compiler den Zugriff auf Attribute durch Offset-Operationen realisieren.

[19] Das Einladen persistenter Objekte in den virtuellen Speicher eines Prozesses geschieht in den erwähnten Systemen jeweils seitenweise. Weil immer eine komplette Seite (4 bis 8 Kilobyte) eingelagert werden ist die Ersparnis wegen der fehlenden Transformation sehr deutlich.

```
void
BauteilStatik::f(pointer p, type stype, bool callShareBlocks)
{
  BauteilStatik::
    mittelgeometrie(p, new MittelGeometrie::create());
  MittelGeometrie::
    transformiere(BauteilStatik::mittelgeometrie(p, stype),
                  attribute_type(Mittelgeometrie, stype));
}
MittelGeometrie*
BauteilStatik::mittelgeometrie(pointer p, type stype)
{
  // lesender Zugriff auf ein erweitertes Attribut:
  ExtendedAttribute<Bauteil, MittelGeometrie>* a =
    mittelGeometrieCollection->lookup(p);
  if (a == nil) {
    return BauteilStatik::mittelgeometrie_default();
  }
  else {
    return a->mittelgeometrie;
  }
}
void
BauteilStatik::mittelgeometrie(pointer p, type stype,
                               MittelGeometrie* g)
{
  // schreibender Zugriff auf ein erweitertes Attribut:
  ExtendedAttribute<Bauteil, MittelGeometrie>* a =
    mittelGeometrieCollection->lookup(p);
  if (a == nil) {
    a = new ExtendedAttribute<Bauteil, MittelGeometrie>;
    mittelGeometroeCollection->insert(a, p);
  }
  a->mittelgeometrie = g;
}
```

Abbildung 49: Implementierung erweiterter Attribute durch globale Tabellen (Dictionaries)

4.2.5.1 Zugriff über Tabellen

Eine Realisierung erweiterter Attribute nutzt globale Kollektionen von Objekten, die jeweils ein Attribut und den Rückverweis auf das Objekt, deren Attribut sie speichern, beinhalten. Abbildung 50 skizziert diese Implementierung und zeigt die für dieses Beispiel notwendigen zwei globalen Dictionaries, die jeweils unter dem Objectidentifier *(OID)* des Originalobjektes als Schlüssel den Wert eines erweiterten Attributes vom Typ *Solid* oder *MittelGeometrie* ablegen. Ein schreibender und lesender Zugriff auf ein erweitertes Attribut – wie in der folgenden Methode

```
void BauteilStatik::f()
{
  mittelgeometrie = new MittelGeometrie::create();
  mittelgeometrie->transformiere(1, 1, 1);
}
```

beispielhaft dargestellt – wird durch den in Abbildung 49 gezeigten Code realisiert. Diese Technik hat eine Reihe von Vor- und Nachteilen, die im folgenden sehr kurz diskutiert werden .

+ ***Lazy Evaluation*** – die Originalobjekte haben keinerlei Platz für zukünftige erweiterte Attribute reserviert; der Speicheraufwand für alle Objekte, die ein bestimmtes Attribut niemals schreiben, ist deshalb gleich Null.

+ ***einfache Implementierbarkeit*** – die gesamte vom Compiler zu leistende Implementierung ist in Abbildung 49 dargestellt. Dictionaries sind Standard-Datenstrukturen, die in vielfältigen Varianten verfügbar sind. Da wir als Schlüssel einen OID verwenden, bietet sich für die Realisierung eine Hashtabelle an.

− ***hoher Speicherbedarf bei hoher Anzahl erweiterter Attribute*** – gemeint ist hier nicht die Anzahl unterschiedlicher Attributtypen bzw. Attributdefinitionen in Klassen, sondern das Verhältnis Anzahl Objekte/Anzahl dazugehöriger erweiterter Attributinstanzen einer Attributdefinition. Eine Hashtabelle ist eine dynamische Struktur und hat zwangsläufig einen internen Speicherbedarf für die Verwaltung der Hashbuckets etc.

− ***Performance*** – lookups in einer Hashtabelle haben zwar im Allgemeinen einen konstanten Laufzeitaufwand, im Datenbankkontext muss jedoch berücksichtigt werden, dass der Einstieg in die Tabelle und die Suche nach dem richtigen Hashbucket im Allgemeinen wenigstens zwei Seitenzugriffe bedeutet. Dazu kommen Concurrency Control Probleme, wenn unterschiedliche Anwendungen gleichzeitig erweiterte Attribute erzeugen bzw. löschen. Eine globale Tabelle wird dadurch schnell zum Engpass in Bezug auf Sperren, obwohl die eigentlichen Objekte problemlos gleichzeitig schreibbar sind.

Insbesondere der letzte Punkt ist von großer Bedeutung. Wenn die Kosten für Attributzugriffe für erweiterte Attribut wesentlich höher sind als die für Attribute in nicht polymorph abgeleiteten Klassen, dann werden Entwickler zwangsläufig aus Perfomance-Gründen häufig benötigte Attribute nicht als erweiterte Attribute ablegen und damit die Entkopplung unterschiedlicher Perspektiven torpedieren. Die Suche nach einem möglichst effizienten Ansatz führt deshalb zu der im folgenden Abschnitt 4.2.5.2 vorgestellten Lösung.

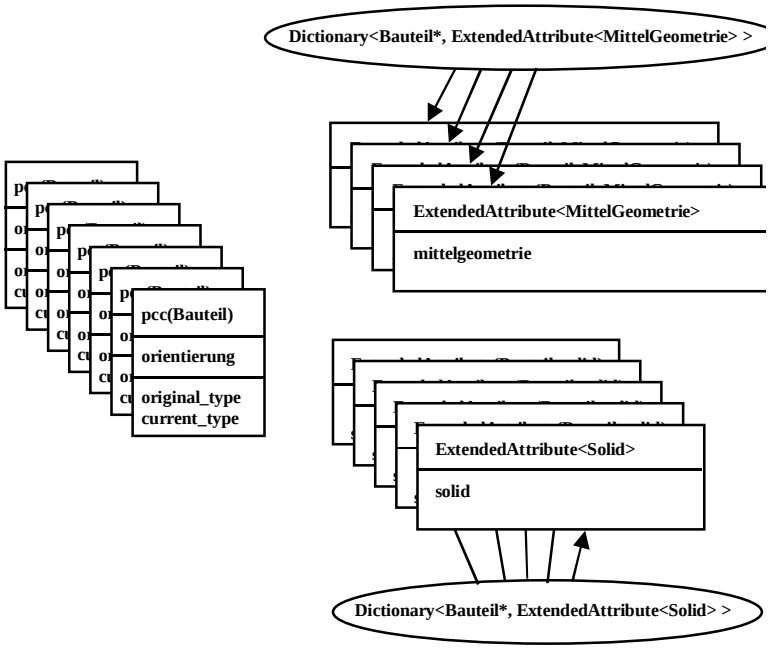

Abbildung 50: Realisierung erweiterter Attribute durch Tabellen

4.2.5.2 Zugriff über Referenzen im Objekt

Grundlegende Idee dieser Technik ist es, jede Art von lookup in Tabellen oder ähnlichem zu vermeiden und den Zugriff auf erweiterte Attribute durch maximal eine Dereferenzierung zu realisieren. Dazu wird in jedem Objekt ein Verweis auf die Basisklasse aller Behälter für erweiterte Attribute angelegt. Dieser Verweis hat zunächst den Wert *nil*, solange auf kein erweitertes Attribut des fraglichen Objekts schreibend zugegriffen wird. Beim ersten schreibenden Zugriff wird ein Objekt einer Klasse erzeugt, welche genau den zu schreibenden Wert enthält und zusätzlich über alle bis zu diesem Zeitpunkt bereits geschriebenen erweiterten Attributwerte des Objekts verfügt. Abbildung 51 zeigt diese Technik in Form des generierten Codes. Wesentliches Merkmal dieser Technik ist, dass beim Schreiben eines erweiterten Attributes, welches noch nicht angelegt war, eine neue Struktur erzeugt wird, in welche die bereits angelegten Attribute kopiert werden (vgl. Abbildung 52). Anschließend wird der alte Behälter für erweiterte Attribute gelöscht.

```
class EA Bauteil {
public:
  virtual bool             has_solid() const        { return false; }
  virtual Solid*           get_solid() const        { return nil;   }
  virtual void             set_solid(Solid* s)      {               }
  virtual bool             has_mg() const           { return false; }
  virtual MittelGeometrie* get_mg() const           { return nil;   }
  virtual void             set_mg(MittelGeometrie*) {               }
  virtual EA-Bauteil*      acquire_solid()          { return nil;   }
  virtual EA Bauteil*      acquire_mg()             { return nil;   }
} ;
class EA-BauteilTragwerk : public virtual EA-Bauteil {
public:
  bool        has_solid() const   { return true;  }
  Solid*      get_solid() const   { return solid; }
  void        set_solid(Solid* s) { solid = s;    }
  EA-Bauteil* acquire-mg() {
    EA-Bauteil* result = new EA-BauteilTragwerkStatik;
    result->_solid = _solid;
    return result;
  }
private:
  Solid* _solid;
};
class EA_BauteilTragwerkStatik : public EA_BauteilTragwerk,
                                 public EA_BauteilStatik {
public:
  bool has_solid() const
  { return EA_BauteilTragwerk::has_solid(); }
  Solid* get_solid() const
  { return EA_BauteilTragwerk::get_solid(); }
  void set_solid(Solid* s)
  { EA_BauteilTragwerk::set_solid(s); }
  bool has_mg() const
  { return EA_BauteilStatik::has_mg(); }
  MittelGeometrie* get_mg() const
  { return EA_BauteilStatik::get_mg(); }
  void set_mg(MittelGeometrie* mg)
  { EA_BauteilStatik::set_mg(mg); }
  EA_Bauteil* acquire_solid() { return nil; }
  EA Bauteil* acquire_mg()    { return nil; }
} ;
```

Abbildung 51: Implementierung erweiterter Attribute über Referenzen im Originalobjekt I

```
Solid* BauteilTragwerk::solid(Pointer p, Type stype)
{
  if (p->_extendedAttributes == nil) ||
      !p->_extendedAttributes->has_solid())
    return BauteilTragwerk::solid_default;
  else
    return p->_extendedAttributes->get_solid();
}
void BauteilTragwerk::solid(Pointer p, Type stype, Solid* s)
{
  if (p->_extendedAttributes == nil)
    p->_extendedAttributes = new EA_BauteilTragwerk;
  else if (!p->_extendedAttributes->has_solid()) {
    EA_Bauteil* ea = p->_extendedAttributes->acquire_solid();
    Delete p->_extendedAttributes ;
    p->_extendedAttributes = ea ;
  }
  p->_extendedAttributes->set_solid(s) ;
}
```

Abbildung 52: Implementierung erweiterter Attribute über Referenzen im Originalobjekt II

Abschließend eine kurze Darstellung der Vor- und Nachteile dieses Ansatzes, der für die CPL-Implementierung gewählt wurde:

+ ***Performance*** – wegen der Beschränkung auf genau eine Dereferenzierung bedeutet der Zugriff auf erweiterte Attribute höchstens einen Seitenzugriff.

+ ***einfache Implementierbarkeit*** – zwar erfordert die Generierung der Datenstrukturen aus Abbildung 51 eine Analyse der Menge aller erweiterter Attributdefinitionen für eine Klassenhierarchie und die Erzeugung von $n!$ Klassen bei n Attributdefinitionen, diese Generierung ist jedoch sehr einfacher Natur. Für jedes neu definierte Attribut a müssen die bereits generierten Klassen um die Methoden *has_a()*, *get_a()*, *set_a()* und *acquire_a()* erweitert werden. Da dies jedoch automatisch durch den Compiler geschieht ist es nicht als Nachteil zu bewerten.

+ ***geringer Speicherbedarf pro erweitertes Attribut*** – Im ungünstigsten Fall werden für ein einziges erweitertes Attribut zwei Worte benötigt; eines für den Verweis *_extendedAttribute* im Originalobjekt, une ein weiteres für die Typkennung (*virtual function table pointer* oder andere Form der Typkennung) in der Datenstruktur, welche das Attribut enthält. Für den Fall, dass mehrere erweiterte Attribute pro Objekt geschrieben werden, steigt dieser Aufwand nicht.

– ***höherer Speicherbedarf bei geringer Anzahl erweiterter Attribute*** – da von vornherein in jedem Objekt, ganz gleich ob es zu einer Klasse gehört, die niemals erweiterte Attribute besitzen wird oder nicht, ein Verweis auf solche

Attribute gehalten wird. Natürlich sind Optimierungen vorstellbar, in denen der Entwickler dem Compiler eine Liste von Klassen vorgibt, für die er keine erweiterten Attribute erlauben möchte. Tritt dieser Fall später dennoch ein, so muss Schemaevolution bzw. eine physische Migration der persistenten Instanzen einer solchen Klasse vorgenommen werden

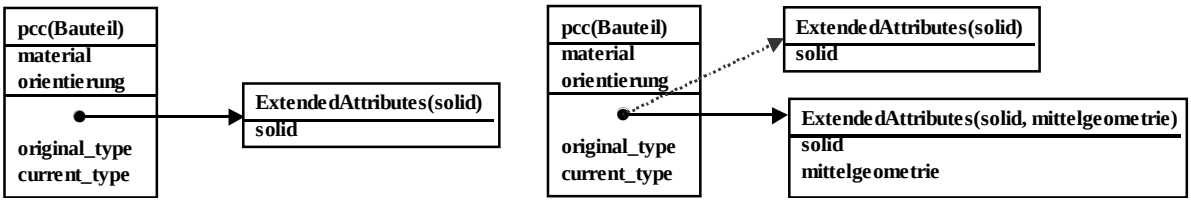

Abbildung 53: physisches Layout für erweiterte Attribute

4.3 Optimierung

Wie bereits erwähnt, basieren die im folgenden diskutierten Optimierungsverfahren auf globalem Wissen und unterscheiden sich damit maßgeblich von den Verfahren, wie sie z.B. in kommerziellen C++ Compilern realisiert sind. Globale Optimierung geht von der sogenannten closed-world-assumption aus: zur Linkzeit einer Anwendung muss nicht mehr mit weiteren (Unter-) Klassen und damit insbesondere nicht mit weiteren Redefinitionen vorhandener Methoden gerechnet werden. C++ oder Java-Compiler erzeugen im Gegensatz dazu zur Übersetzungszeit einzelner Quelldateien ein Kompilat, welches keinerlei Annahmen darüber vornimmt, ob weitere Unterklassen zur Anwendung gebunden werden. Dem Compiler stehen damit zur Generierung von Methodenaufrufen lediglich die Informationen aus der Klassendeklaration des statischen Typs des Empfängerobjekts zu Verfügung. Wird in dieser Klasse A die aufgerufene f Methode als *virtual* (C++) bzw. nicht als *final* (Java) deklariert, so erzeugt der Compiler einen Sprung in die virtuelle Funktionstabelle, auch wenn es später in der gesamten Anwendung keine einzige Unterklasse von A bzw. keine einzige Redefinition von f auftaucht.

Die wichtigste Aufgabe des Optimierers im CPL-Compiler ist es, Laufzeiteinbußen durch CPL-spezifische Konstrukte soweit wie möglich zu eliminieren. Dazu zählen im Wesentlichen der zusätzliche Aufwand beim dynamischen Binden sowie die beim Eintritt in nicht dynamisch gebundene Methoden auszuführende Berechnung der aktuellen Perspektive. Wie in Abschnitt 4.2.3 beschrieben ist letzteres für den Zugriff auf kovariant redefinierte Attribute notwendig. Ein weiterer Optimierungskandidat sind die in Abschnitt 3.3.3 eingeführten *share*-Blöcke: die Ausführung der automatisch zusammengesetzten *share*-Blöcke innerhalb einer Methode sollte keinerlei zusätzliche Kosten z.B. in Form von Methodenaufrufen verursachen.

Letzteres ist durch globale Analyse sehr einfach zu realisieren: Wenn alle `share`-Blöcke der Anwendung bekannt sind, können die zusammengesetzten Anweisungen aller zu berücksichtigenden Blöcke direkt in den Methodencode generiert werden.

4.3.1 Dead Code Elimination

Ziel der Dead Code Elimination ist es, möglichst wenig Maschinencode zu generieren, der zur Laufzeit der Applikation gar nicht aufgerufen werden kann. Wir betrachten dabei die Menge der Funktionen/Methoden, die vom Startpunkt einer Applikation aus erreichbar sind. Der Eintrittspunkt einer Applikation wird in CPL beim Link der Anwendung durch Angabe einer Klasse definiert, die über eine Methode mit der Signatur `static int main(const Array<String>*)` verfügt. Wir untersuchen nicht einzelne Anweisungen oder Anweisungsfolgen innerhalb von Methoden, die niemals aufgerufen können. Derartige Optimierungen überlassen wir dem Backend- Compiler, der den vom CPL-Compiler erzeugten C/C++-Code den üblichen Verfahren unterzieht.

Beim Übersetzen von Quelldateien erzeugt der CPL-Compiler einen Aufrufgraphen [GDDC97], der aus je einem Knoten pro Methode und von diesen Knoten ausgehenden Kanten für jeden Methodenaufruf innerhalb dieser Methode besteht. Eine Besonderheit sind dynamisch gebundene Methoden, da im Allgemeinen an der Stelle des Aufrufs (*call site*) nicht entschieden werden kann, welchen tatsächlichen Typ des Empfängerobjekt hat. Betrachten wir als Beispiel das folgende Codefragment:

```
int Mainclass::main()
{
  auto Bauteil* b = ...;
  b->verschieben(1.0, 1.0, 1.0);
  ...
}
```

Beim Übersetzen dieses Quelltextes ist dem Compiler nicht die Menge aller Redefinitionen von *Bauteil::verschieben* bekannt. Im Aufrufgraphen wird deshalb lediglich die Kante *Mainclass::main* → *Bauteil::verschieben* eingetragen. Zur Linkzeit der Anwendung kann der Compiler jedoch die gesamte Klassenhierarchie analysieren und die Kanten *Mainclass::main* → *BauteilStatik::verschieben,* und *Mainclass::main* → *BauteilTragwerk::verschieben* hinzufügen.

Eine CPL-spezifische Besonderheit in diesem Zusammenhang bilden Implementierungs-Invarianten. Um den Aufrufgraph korrekt aufbauen zu können müssen deshalb für jeden Knoten $c.f$ auch die share-Blöcke aus den Redefinitionen von f aus $ctypes(c)$ berücksichtigt werden.

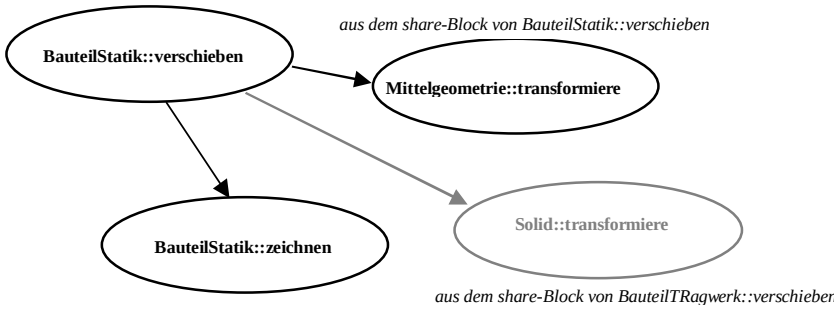

Abbildung 54: Aufrufgraph inklusive share-Blöcke

Abbildung 54 zeigt in den schwarz dargestellten Knoten und Kanten den Teil eines Aufrufgraphen, wie er in einer traditionellen objektorientierten Programmiersprache zu erzeugen wäre. Der grau dargestellte Knoten *Solid::transformiere* nebst hinführender Kante entsteht durch die Implementierungs-Invariante für *BauteilStatik::verschieben*. Man beachte, dass die durch Implementierungs-Invarianten induzierten zusätzlichen Kanten im Allgemeinen das Hinzubinden von „applikationsfremdem“ Code verlangen. In diesem Fall muss beispielsweise die Statik-Applikation auch Funktionalität von Teilen der Tragwerkmodeliierungs-Applikation, nämlich die Klasse *Solid* bzw. ihre *transformiere*-Methode, einbinden. Dies ist jedoch aus den in Abschnitt 3.3 diskutierten Gründen unbedingt notwendig um die Daten anwendungsübergreifend konsistent zu halten. Wichtig ist, dass derart entstehende Kanten sich nicht auf der Source-Code-Ebene auswirken, sondern lediglich Link-Abhängigkeiten produzieren.

4.3.2 Dead Type Elimination

Dead Type Elimination dient – anders als die im Abschnitt 4.3.1 kurz eingeführte Dead Code Elimination – nicht zur Verkleinerung des generierten Codes, sondern ist ein Hilfsmittel für die Ersetzung dynamisch gebundener Methodenaufrufe durch statisch gebundene. Wie wir in Abschnitt 4.3.3 sehen werden kann das Wissen, dass Instanzen bestimmter Klassen zur Laufzeit nicht existieren können, in einigen Fällen zur Vermeidung dynamischen Bindens führen.

In traditionellen objektorientierten Sprachen genügt für die Dead Type Elimination lediglich ein Durchsuchen des Aufrufgraphen nach Konstruktor-Aufrufen. Klassen, deren Konstruktoren nicht in dieser eingesammelten Menge enthalten sind, können eliminiert werden. Einen Spezialfall bildet in diesem Zusammenhang C++: Zur Erzeugung eines Objektes wird in C++ eine Kaskade von Konstruktoren durchlaufen, währenddessen sich der Typ des Objekts ändert. Betrachten wir die Klassen A und B aus Abbildung 55: während der Ausführung des Konstruktors der Klasse A wird der Aufruf der dynamisch gebundenen Methode f Typ A dispatched, im anschließenden

Rumpf des Konstruktors von *B* wird dagegen für *f* die Implementierung von *B* gewählt.

```
class A {
public:
  A{) { f{) ; }
  virtual void f() { … }
} ;
class B : public A {
public:
  B() { f() ; }
  virtual void f() { … }
} ;
```

Abbildung 55: Konstruktor-Kaskade in C++

Wir erhalten also für eine Variablendeklaration `A a;` die Ausführungsfolge *A::A, A::f*, *B::B*, *B::f*[20]. Für die Dead Type Elimination wirkt sich dieses Verhalten von C++ aus, weil eventuell neben den Klassen, deren explizite Konstruktor-Aufrufe im Aufrufgraphen gefunden wurden, auch noch deren Oberklassen in Betracht zu ziehen sind. CPL schließt sich jedoch der Vorgehensweise von Java und Eiffel an und betrachtet Objekte auch beim Durchlauf von Konstruktoren der Basisklassen von vornherein als Instanz des Typs, als dessen Instanz ein Objekt deklariert wurde.

Eine Besonderheit in Bezug Dead Type Elimination entsteht durch die Eigenschaft von CPL, persistente Objekte beschreiben zu können. Für eine Applikation ist es deshalb leicht denkbar, dass sie auf persistente Objekte aus einer Datenbank zugreift, deren Typ bzw. die Implementierungen der Methoden diesen Typs aber nicht kennt. Zum Beispiel kann eine Statik-Anwendung mit den Klassen *BauteilStatik, PlatteStatik* und *BalkenStatik* über eine persistente Kollektion vom Typ *List<Bauteil>* iterieren und dabei auf den neuen Typ *UnterzugStatik* treffen[21]. Um diesen Fall korrekt behandeln zu können muss der Entwickler die Möglichkeit geboten bekommen, zum

[20] Als weitere Besonderheit besitzt C++ sogenannte Destruktoren als Gegenstück zu Konstruktoren. Wegen des Fehlens von Garbage Collection werden diese implizit beim Freigeben des Speicherplatzes für ein Objekt aufgerufen (manueller Aufruf von `delete` oder Verlassen des Stackframes). Die Änderung des Typs eines Objekts geschieht in Destruktoren analog zu Konstruktoren, jedoch in umgekehrter Reihenfolge.

[21] In einer solchen Situation wird im Falle optimierter Codegenerierung der CPL-Compiler für das Öffnen von Datenbanken Schemavalidierungs-Code erzeugen, der derartige Fälle überprüft und gegebenenfalls das Programm mit einer Fehlermeldung abbricht.

Aufbau des Aufrufgraphen die Startfunktionen mehrerer Applikationen als Wurzeln angeben zu können.

Eine weit wichtigere Auswirkung des Typsystems von CPL auf die Dead Type Elimination ist jedoch die Tatsache, dass Perspektiven in Applikationen benutzt werden können, ohne dass jemals ein Konstruktor eines solchen Perspektive aufgerufen wird, wie man leicht an folgendem Beispiel sehen kann:

```
auto Bauteil* b = new Platte::create();
view_cast<VRBauteilTragwerk*>(b)->zeichnen();
```

Offensichtlich wäre es ein Fehler, wenn der Typ *VRBauteilTragwerk* der Dead Type Elimination zum Opfer fiele, weil kein Konstruktor-Aufruf dieser Klasse gefunden wird. Gleiches gilt natürlich auch für die übrigen Verhaltens-Manipulatoren aus CPL:

```
auto Bauteil* b = new Platte::create();
type_behavior<BauteilStatik>();
b->zeichnen();
instance_behavior<BauteilTragwerk*>(b);
b->zeichnen();
```

In diesem Fall werden die Klassen *BauteilStatik* und *BauteilTragwerk* ebenfalls nicht durch Konstruktor-Aufrufe als erreichbar gekennzeichnet, wohl aber durch Umschalten von Bauteil-Objekten auf diese Klassen als Perspektiven. Wie bei diesen Beispielen jedoch auffällt, wird mit dem Aktivieren einer Perspektive *BauteilStatik* potentiell auch das Verhalten von deren Unterklassen *PlatteStatik* und *BalkenStatik* eingebunden. Allgemeiner formuliert bedeutet dies, dass für jede via Konstruktor- Aufruf erreichbar markierte Klasse auch deren Perspektiven als erreichbar markiert werden müssen, sofern ein Typumwandlungsoperator diese Perspektive oder eine Oberklasse von ihr zum Ziel hat. Sei RC die Menge aller per Konstruktor-Aufruf und RT die Menge aller per Typumwandlungsoperator markierten Klassen. $P = \{c \mid \exists c' \in RC : c \in pcc(c')\}$ bezeichne die Menge aller Perspektiven für die Klassen aus RC. Dann ist $LP = \{c \in P \mid \exists c' \in RT : c \in \overline{sub(c')}\}$ die Menge aller lebendigen Perspektiven und $LC = RC \cup LP$ die gesuchte Menge aller lebendigen Klassen.

Natürlich gilt für die Aufrufe der Typumwandlungsoperatoren wie auch derer für Konstruktoren, dass sie selbst innerhalb erreichbarer Methodenrümpfe liegen müssen um zu einer Markierung ihres Typs zu führen.

4.3.3 Dynamisches Binden

Verfahren, die durch globale Optimierung dynamisches Binden in vielen Fällen vermeiden sind in der Literatur [AgHö95, AiHö95, BaSw96, CaGr94, GDC95, ZCC98] schon lange etabliert und auch bereits in kommerziellen Compilern für die objektorientierte Programmiersprache Eiffel implementiert. Weit verbreitet ist zum Beispiel das Verfahren, Methoden statisch zu binden wenn an der call site der Methode f für einen Ausdruck e die Bedingung $|itypes(e) \cap LC| = 1$ gilt. Ebenso überflüssig ist dynamisches Binden, wenn zwar $|itypes(e) \cap LC| > 1$ gilt, es jedoch keine Redefinition von f in $itypes(e) \cap LC \setminus \{stype(e)\}$ gibt. In unserem laufenden Beispiel wäre ein Aufruf der Art

```
auto PlatteStatik* p = ...;
p->zeichnen();
```

leicht statisch zu binden, weil es zu *PlatteStatik* keine Unterklasse gibt, in der die Methode *zeichnen* redefiniert ist. Ein Aufruf der Art

```
auto Platte* p = ...;
p->zeichnen();
```

hat dagegen die Menge {*Platte*, *PlatteStatik*, *PlatteTragwerk*} als mögliche Empfängertypen der Variable *b.* Hier wird es interessant, durch Dead Type Elimination die Menge dieser Klassen zu reduzieren. So ist es gut denkbar, dass eine Applikation, die den Bauingenieur bei der Tragwerkmodellierung unterstützt, lediglich diese Perspektive benutzt und damit die beiden Klassen *Platte* und *PlatteStatik* eliminiert[22].

Der CPL-Compiler erzeugt für das dynamische Binden, falls es vom Optimierer als notwendig analysiert wird, Fallunterscheidungen wie in Abbildung 42 illustriert. Diese dispatch-Funktionen werden spezifisch für einzelne call-sites bzw. generiert, so dass bei Ausdrücken mit speziellerem statischen Typ einige Verzweigungen in den Fallunterscheidungen eingespart werden können. Für den Aufruf

```
auto Balken* b = …;
b->zeichnen();
```

wird z.B. die disptach-Funktion

[22] An dieser Stelle sei erneut an persistente Objekte erinnert. Selbst wenn aus der betrachteten Applikation heraus die Klasse *Platte* nicht als erreichbar markiert werden kann, so werden wahrscheinlich andere Applikationen den Typ *PlatteTragwerk* oder *Platte* zur Objektinstanziierung nutzen und damit auch diese Klassen als Empfängertypen in Frage kommen.

```
void Balken::zeichnen_dispatch(Pointer p, Type stype) {
  Type t = type_dispatch(p, stype);
  switch (t) {
  case 1: Bauteil::zeichnen(p, t);         break;
  case 4: BauteilStatik::zeichnenn(p, t);  break;
  case 7: BauteilTragwerk::zeichnen(p, t); break;
  }
}
```

generiert. Für dispatch-Funktionen, deren Fallunterscheidung lediglich zwei mögliche Empfängertypen unterscheiden muss, erzeugt der CPL-Compiler inline-Code um den Aufwand eines Funktionsaufrufes für das dynamische Binden ganz zu vermeiden.

Die Fallunterscheidungen für mehr als zwei mögliche Empfängertypen werden jedoch durch binäre Suche über den linear geordneten Typidentifikatoren realisiert, so dass die Suchzeit für den richtigen Typ maximal logarithmisch zur Basis zwei im Verhältnis zur Menge der möglichen Empfängertypen an einer gegebenen call site ist. Die Realisierung mittels `switch` ist in unseren Beispielen lediglich für eine übersichtlichere Darstellung gewählt. Es mag überraschend erscheinen, eine solche Technik trotz ihrer nicht konstanten Laufzeit zu verwenden, obwohl doch *virtual function table* Implementierungen immer eine konstante Laufzeit für die Implementierungsauswahl garantieren. Verschiedene Gruppen [DrHö96, DMM96, ZCC98] haben jedoch festgestellt, dass heutige Hardware mit den durch die *virtual function table* Technik induzierten indirekten Sprüngen Schwierigkeiten bekommen, weil dadurch die Prozessor-Pipeline invalidiert wird. Eine ausführliche Diskussion dieses Sachverhalts findet sich in [DHV95] und [HöUn95]. Unser Verfahren hält sich recht genau an die dort vorgestellten Implementierungen, weshalb wir an dieser Stelle nicht weiter darauf eingehen wollen.

4.3.4 Vermeidung von *type_dispatch*

Während die Vermeidung dynamischen Bindens in CPL weitestgehend den Techniken aus traditionellen Compilern objektorientierter Sprachen ähnelt, muss ein leistungsfähiger Optimierer ganz besonders den für Perspektiven typischen Overhead reduzieren. Der deutlichste Unterschied zu klassischen objektorientierten Sprachen besteht im Aufruf von *type_dispatch,* der einerseits zum dynamischen Binden von Methoden, andererseits als Vorbedingung für den Zugriff auf variante Attribute ausgeführt wird (vgl. 4.2.3). Wie letzteres komplett eliminiert werden kann, diskutiert Abschnitt 4.3.5 ausführlich; der verbleibende Teil dieses Abschnitts konzentriert sich deshalb auf *type_dispatch* für dynamisches Binden von Methodenaufrufen.

Als erster Schritt kann die Implementierung des dynamischen Bindens dahingehend verfeinert werden, dass bei der Analyse der Menge von potentiellen Empfängertypen zwischen multipler Instanziierung und klassischen Unterklassen unterschieden wird. Sei e ein Ausdruck und f eine Methode – $e.f$ bezeichnet dann den uns interessierenden Methodenaufruf. Auf den Aufruf von *type_dispatch* kann verzichtet werden, wenn für alle $c \in itypes(e)$ gilt: $pcc(c) \cap itypes(e) \cap LC = \{c\}$. Gleiches gilt, wenn zwar für ein $c \in itypes(e)$ gilt: $|\, pcc(c) \cap itypes(e) \cap LC \,| > 1$, jedoch in $pcc(c) \cap itypes(e) \cap LC$ maximal eine Definition von f existiert. Dieser Fall trifft in der Praxis auf sehr viele Aufrufe zu; so ist zum Beispiel bei folgendem Codefragment

```
auto Bauteil* b = ...;
view_cast<BauteilStatik*>(b)->zeichnen();
```

Abbildung 56

der Ausdruck e durch `view_cast<BauteilStatik*>(b)`, f durch *zeichnen, stype*(e) = *BauteilStatik* und *itypes*(e) durch {*BauteilStatik, PlatteStatik, BalkenStatik*} gegeben. Offensichtlich gilt zwar für alle $c \in itypes(e) :|\, pcc(c) \,| > 1$, die Einschränkung der implementierenden Typen auf $\overline{sub(stype(e))}$ (vgl. Definition 3.10 in Abschnitt 3.1.2) schließt jedoch als mögliche Implementierungen für den Aufruf von f die Klassen *BauteilTragwerk, PlatteTragwerk, BalkenTragwerk, Bauteil, Platte* und *Balken* aus.

```
void BauteilStatik::zeichnen_dispatch(Pointer p, Type stype) {
  switch (p->original_type) {
  case 0: // Bauteil
  case 3: // BauteilStatik
  case 6: // BauteilTragwerk
          BauteilStatik::zeichnen(p, 3); break;
  case 1: // Balken
  case 4: // BalkenStatik
  case 7: // BalkenTragwerk
          BalkenStatik::zeichnen(p, 4);  break;
  case 2: // Platte
  case 5: // PlatteStatik
  case 8: // PlatteTragwerk
          PlatteStatik::zeichnen(p, 5);  break;
  }
}
```

Abbildung 57: dispatch-Funktion für *zeichnen* der call-site aus Abbildung 56

Abbildung 57 zeigt die für den Aufruf aus Abbildung 56 generierte dispatch-Funktion. Durch die Eliminierung des Aufrufs von *type_dispatch* reduziert sich der Rumpf dieser Funktion auf ein klassisches dynamisches Binden wie auch in objektorientierten Sprachen ohne Perspektiven.

4.3.5 Spezialisierung

Die in den vorangegangenen Abschnitten diskutierten Verfahren helfen zwar, einen nennenswerten Teil des Overheads zu eliminieren, der in einer nicht-optimierenden Realisierung durch Perspektiven induziert wird. In der Praxis werden Methodenrümpfe jedoch häufig bewusst so abstrakt formuliert, dass keine für den Optimierer verwendbaren Einschränkungen bzgl. der Empfängertypen eines Aufrufs gemacht werden können. Betrachten wir den Aufruf:

```
auto Gebaude* g = …;
view_cast<GebaeudeStatik*>(g)->zeichnen();
```

Die Implementierung der Methode *Gebaeude::zeichnen* aus Abbildung 4 abstrahiert völlig von speziellen Eigenschaften der in Gebäude-Objekten aggregierten Bauteile – sowohl von der Klassifizierung Platte/Balken als auch von den vorhandenen Perspektiven Statik/Tragwerkmodellierung. Unsere bisherigen Optimierungstechniken können in dieser Situation nichts ausrichten, da *Gebaeude::zeichnen* niemals redefiniert wird. Aber selbst in dem Fall, dass ein dynamisches Binden eingespart werden könnte – der erzielte Performanzgewinn stünde in keinem Verhältnis zur voraussichtlichen Laufzeit der Methode, die zum Großteil in der Schleife über alle Bauteile bzw. dem Zeichnen dieser Bauteile liegt. Im generierten Code für die Methode (Abbildung 45) finden wir in der Schleife jeweils den Aufruf der dispatch-Funktion für *Bauteil::zeichnen* sowie die Aufrufe von *Gebaeude::type_of_bauteile*, um den statischen Typ des kovariant redefinierten Attributes abhängig vom aktuellen Typ von *this* zu berechnen.

Eine sehr weitgehende Optimierung dynamisch gebundener Methodenaufrufe scheint immer einfach realisierbar, wenn der Typ von *this* eindeutig oder wenigstens sehr stark eingeschränkt ist. Hätte der Entwickler beispielsweise *Gebaeude::zeichnen* in den Perspektiven *GebaeudeStatik* und *GebaeudeTragwerk* wie folgt redefiniert

```
void GebaeudeStatik::zeichnen() const {
  // in der Klasse GebaeudeStatik hat _bauteile automatisch den Typ
  // List<BauteilStatik>*
  _bauteile->apply<e>(e->zeichnen());
}
void GebaeudeTragwerk::zeichnen() const {
  // in der Klasse GebaeudeTragwerk hat _bauteile automatisch den Typ
  // List<BauteilTragwerk>*
  _bauteile->apply<e>(e->zeichnen());
}
```

Abbildung 58: „Customized“ Redefinitionen von *Gebaeude::zeichnen*

so hätte der Compiler in den Rümpfen der beiden Methode Redefinitionen sämtliche Vorkehrungen für dynamisches Binden eliminieren können[23], wie die folgende Abbildung 59 zeigt.

```
void GebaeudeStatik::zeichnen(Pointer p, Type stype) {
  Cursor c(Gebaeude::_bauteile);
  for (c.first(); c.more(); c.next()) {
    // 3 ist die ID des Typs BauteilStatik
    BauteilStatik::zeichnen_dispatch(c.retrieve(), 3, true);
  }
}
void GebaeudeTragwerk::zeichnen(Pointer p, Type stype) {
  Cursor c(Gebaeude::_bauteile);
  for (c.first(); c.more(); c.next()) {
    // 6 ist die ID des Typs BauteilTragwerk
    BauteilTragwerk::zeichnen_dispatch(c.retrieve(), 6, true);
  }
}
```

Abbildung 59

Es scheint jedoch unsinnig, vom Entwickler derartige Redefinitionen zu verlangen, die keine neue Funktionalität realisieren und sich sogar textuell gar nicht von der Original-Definition der Methode unterscheiden. Statt dessen sollte der Compiler automatisch solch angepasste Methodenrümpfe erzeugen, wo immer diese Spezialisierung sinnvoll erscheint. Diese Technik wird in der Literatur *Customization* genannt [ChUn89, Lea90, DiHö97, ZCC98]. In traditionellen objektorientierten Programmiersprachen hat sie zum Ziel, das dynamische Binden aller Methodenaufrufe mit *this* als Empfänger zu unterbinden. Dem steht als Nachteil gegenüber, dass für jede abgeleitet Klasse, in der eine geerbte Funktion nicht redefiniert wird, vom Compiler eine solche Redefinition implizit zu generieren ist. Der dadurch zu erwartende Größenzuwachs des Kompilats wird üblicherweise durch Dead Code und Dead Type Elimination wenigstens teilweise kompensiert. Darüber hinaus schränkt diese Technik die Anzahl der Funktionsaufrufe, die durch die in Abschnitt 4.3.3 beschriebene Analyse statisch gebunden werden können, ein: selbst wenn keine vom Entwickler spezifizierte Redefinition einer Methode f in $itypes(e) \cap LC \setminus \{stype(e)\}$ existiert, wird der Compiler wegen der Spezialisierung

[23] Dabei sei wie in früheren Abschnitten auch auf die Annahme hingewiesen, dass die *zeichnen* Methode unterhalb der Perspektiv-Klassen *BauteilStatik* und *BauteilTragwerk* nicht redefiniert wird. Sollte dies aber der Fall sein, dann entfällt im Code der Abbildung 59 immer noch jeglicher Aufwand für die Behandlung von Perspektiven, insbesondere die Berechnung des Typs der varianten Attribute.

Methodenaufrufe im Allgemeinen nur dann statisch binden können, wenn $|itypes(e) \cap LC| = 1$ gilt. Trotzdem ist der Einsatz von Spezialisierung in konventionellen objektorientierten Programmiersprachen in den allermeisten Fällen vorteilhaft, da spätestens dann, wenn eine Methode in ihrem Rumpf eine Schleife mit Aufrufen an *this* enthält, der Performanz-Verlust durch das dynamische Binden der äußeren Funktion mehr als kompensiert wird.

In CPL ist der Gewinn durch Spezialisierung jedoch noch weit größer als in Sprachen ohne Perspektiven. Denn die genaue Kenntnis des Typs von *this* macht die Berechnung des Typs aller varianten Attribute überflüssig. Damit sind die Zugriffe auf derartige Attribute zur Compilezeit auflösbar und ermöglichen ihrerseits optimierte Aufrufe auf von Methoden mit den Attributen als Empfänger. Betrachten wir die Aufrufe von *BauteilStatik::zeichnen_dispatch* bzw. *BauteilTragwerk::zeichnen_dispatch* Abbildung 59: wie in Abschnitt 4.3.4 gezeigt, braucht an dieser Stelle kein *type_dispatch* mehr ausgeführt zu werden, da unterhalb von *BauteilTragwerk* und *BauteilStatik* jeweils für alle $c \in \overline{sub(BauteilTragwerk)}$ bzw. für alle $c \in \overline{sub(BauteilStatik)}$ gilt: $pcc(c) \cap itypes(c) = \{c\}$ und somit klassisches dynamisches Binden ausreicht um den Aufruf korrekt zu realisieren. Man beachte, dass bei Codegenerierung ohne Spezialisierung dieses nicht der Fall wäre, da lediglich der Typ *Bauteil* als Empfängertyp für die Methode *zeichnen* innerhalb der Schleife in *Gebaeude::zeichnen* zur Übersetzungszeit bekannt wäre.

Das dynamische Binden von lediglich durch den Compiler generierten Redefinition erfordert zwar im Allgemeinen den Aufruf von *type_dispatch*, dieser wäre jedoch auch im Falle der nicht dynamischen Bindung notwendig um den Zugriff auf variante Attribute korrekt realisieren zu können. Als zusätzlicher Aufwand gegenüber einer statischen Bindung derartiger Aufrufe schlägt also nur die Fallunterscheidung des Empfängertyps eines Methodenaufrufs zu Buche.

4.3.5.1 Vermeidung von überflüssigen Spezialisierungen

Für die wirkungsvolle Eliminierung eines Großteils des Overheads, der gegenüber konventionellen objektorientierten Programmiersprachen durch Perspektiven verursacht wird, ist der Einsatz von Spezialisierung sehr geeignet, wie der vorangegangene Abschnitt 4.3.5 gezeigt hat.

```
class A {
public:
  virtual int f() const { return 1;   }
  virtual int g() const { return f(); }
  virtual int h() const { return _i;  }
  virtual int j() const { return h(); }
  …
state:
  int _i;
};
class B extends A {
public:
  virtual int h() const { return 0; }
  …
};
```

Abbildung 60

Um die Nachteile, insbesondere die anwachsende Größe des generierten Codes, möglichst gering zu halten stellen wir ein Verfahren vor, welches die Generierung spezialisierter Methodenrümpfe weitestgehend vermeidet, wenn die spezialisierte Methode keinen Effizienzgewinn gegenüber der ursprünglichen Definition erbringt. Einfache Beispiele zeigt Abbildung 60: Eine Generierung spezialisierter Version der Methoden f und g für die Klasse B hätte keinerlei Effekt, weil die in A spezifizierten Methodenrümpfe weder direkt noch indirekt auf Methoden oder Attribute zugreifen, die vom Typ von *this* abhängen. Auch die Generierung einer spezialisierten Version von h für B ist überflüssig, weil h vom Entwickler selbst in B redefiniert wurde. Anders sieht es für die Methode j aus: ihr Rumpf besteht aus einem Aufruf der Funktion h, welche in der abgeleiteten Klasse B überschrieben wird. Eine Spezialisierung von j für *B* ist deshalb sinnvoll und ergibt:

```
int B::j(Pointer p, Type stype) const { return B::h(p, stype); }
```

Abbildung 61: Spezialisierung[24]

Allgemeiner formuliert werden lediglich Methoden spezialisiert, in deren Rumpf direkt oder indirekt auf in Unterklassen redefinierte Methoden oder Attribute mit *this* als Empfänger zugegriffen wird. Dieses Verfahren erlaubt eine wesentliche Einsparung an Spezialisierungen, da in objektorientierten Designs üblicherweise

[24] In diesem Falle würde der CPL-Optimierer einen Rumpf der Form

```
int B::j(Pointer p, Type stype) { return 0; }
```

generieren, weil Funktionen, deren Rumpf lediglich aus einem nicht-rekursiven Ausdruck oder einer Anweisung bestehen, immer als inline-Code generiert werden.

sehr selten redefinierte Funktionen wie z.B. sogenannte „*getter*“, d.h. Methoden, die lediglich den Wert eines Attributes zurückgeben, keinerlei automatische Redefinitionen erzeugen. Eine ausführlichere Diskussion von Spezialisierung in CPL und ihrer Implementierung findet sich in [Orth99].

4.3.6 Ergebnisse

Die wichtigste Aufgabenstellung für eine effiziente Implementierung von Perspektiven ist die Eliminierung des Overheads, der durch die in Abschnitt 4.2 vorgestellte Realisierung entsteht. Dies gilt insbesondere für Programmteile, die gar nicht die zusätzlichen Möglichkeiten polymorpher Ableitung bzw. multipler Instanziierung verwenden[25]. Wie wir in Abschnitt 4.3.4 gesehen haben, wird *type_dispatch* bei optimierter Codegenerierung nur dann ausgeführt, wenn es zu einem gegebenen Methodenaufruf Perspektiven gibt, die „lebendige“ Unterklassen des statischen Typs des Empfängerobjektes sind. Die Berechnung des Typs varianter Attribute entfällt ebenfalls bei Abwesenheit von Perspektiven, da die Redefinition des Typs c eines Attributes nur innerhalb von $pcc(c)$ erlaubt ist.

Doch auch im Falle von in der Applikation eingebundenen Perspektiven erweisen sich die in den Abschnitten 4.3.1 bis 4.3.5 vorgestellten Techniken als sehr leistungsfähig. Betrachten wir einen Ausschnitt aus einer realistischen Applikation, die in CPL während des DFG Schwerpunktprogramms „Objektiorientierte Modellierung in Planung und Konstruktion“ von der Forschungsgruppe des Instituts für Massivbau an der TH Darmstadt implementiert wurde [MPP98]. Abbildung 62 zeigt diese Modellierung. Ein Gebäude ist aus einer Menge von Bauteilen aggregiert – sowohl Gebäude als auch Bauteile können in den zu den Bauingenieur-Disziplinen korrespondierenden Perspektiven genutzt werden. Darüber hinaus gibt es eine Perspektive von Gebäuden im Tragwerkmodell, die eine Visualisierung mittels VRML unterstützt (*VRGebaeudeTragwerk*). Diese, nur für die Visualisierung definierte Perspektive zieht sich durch die komplette Aggragtionshierarchie durch. Dabei ist in der Klasse *VRGebaeudeTragwerk* das Attribut *bauteile* zum Typ *List<VRBauteilTragwerk>**, im Typ *VRBauteilTragwerk* das Attribut *geometrien* zum Typ *List<VRGeometrie3D>** und im Typ *VRGeometrie3D* das Attribut *flaechen* zum Typ *List<VRFlaeche>** redefiniert. Die Geometrie eines Bauteils ist entweder durch die Klasse *Geometrie2D* (System-Mittelgeometrie) für die Modellierung der Statik, oder durch die Klasse *Geometrie3D* (Boundary-Representation) für die Beschreibung des

[25] Wir folgen hier gerne Bjarne Stroustrups Devise „never pay for what you don’t need“ [Stro94]

Tragwerkmodells spezifiziert. Betrachten wir für unser Beispiel jetzt lediglich die Tragwerksmodellierung.

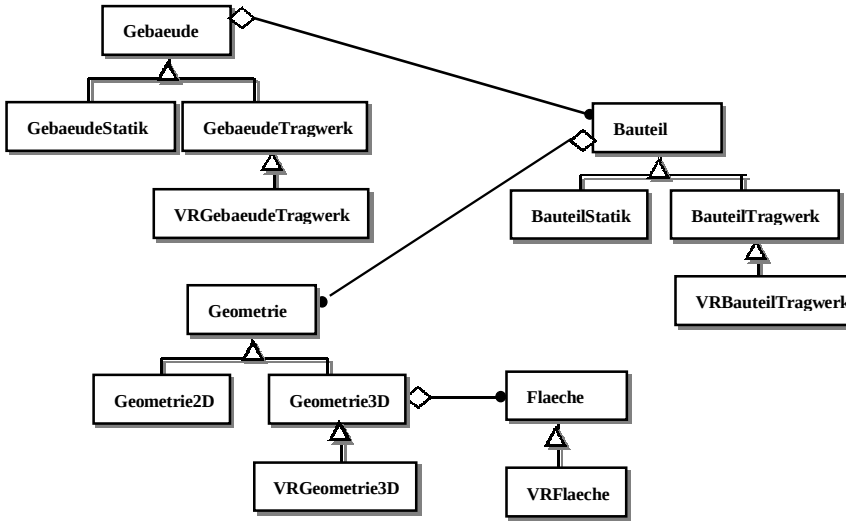

Abbildung 62: Perspektiven in der Aggregationhierarchie

Ein Aufruf der Methode *zeichnen* auf einem Objekt vom Typ *GebaeudeTragwerk* stößt deshalb drei geschachtelte Iterationen über Aggregationen an, in denen bei nicht-optimierter Codegenerierung jeweils wieder die passende Perspektive (wegen der varianten Attribute) ermittelt werden muss. Wenn wir davon ausgehen, dass ein Gebäude aus mehreren zehntausend Bauteilen, jedes Bauteil aus mehrer Geometrien und jede Geometrie aus mindestens sechs Flächen zusammengesetzt ist, so ergibt sich hier sehr schnell eine Größenordnung von einigen hunderttausend Aufrufen von *type_dispatch* zuzüglich der gleichen Anzahl von Berechnungen des Typs der varianten Attribute.

```
auto Gebaeude* g = …;
view_cast<VRGebaeudeTragwerk*>(g)->zeichnen();
```

Bei optimierter Codegenerierung mit Spezialisierung wird der CPL-Compiler für obigen Aufruf jedoch keine einzige Berechnung von *type_dispatch* erzeugen, ebenso wenig wie Ermittlungen des Typs der varianten Attribute. Abbildung 63 zeigt die generierten spezialisierten Methodenrümpfe, in denen die genaue Kenntnis des Typs von *this* alle CPL-spezifischen Berechnungen unnötig macht. Zusätzlich entfällt sogar auch das klassische dynamische Binden für dies Aufrufketten komplett.

Die Mächtigkeit der in diesem Kapitel vorgestellten Optimierungstechniken erlaubt es, auch den auf sehr abstraktem Niveau geschriebenen Code der *zeichnen*-Methoden von *Gebaeude* so stark an einzelne Aufrufsituationen anzupassen, dass der Compiler genügend einschränkende Annahmen über die Empfängertypen machen kann um *type_dispatch* und sehr häufig auch das dynamische Binden an sich

zu eliminieren. Damit sind alle Vorteile der zusätzlichen Flexibilität erhalten geblieben, ohne dass der Entwickler mit Laufzeiteinbußen bezahlen muss.

Ein Nachteil für die Performanz von CPL-Anwendungen ergibt sich jedoch durch die Tatsache, dass Referenzen in CPL eine interne Darstellung besitzen, die mehr als ein Maschinenwort für die Adresse und den Typidentifikator verlangt. Dies hat nicht nur den Effekt, dass Datenobjekte an sich größer werden und dadurch Anwendungen unter Umständen früher den physischen Hauptspeicher verbrauchen und zum paging führen. Weitaus negativer für die Performanz ist, dass Referenzen, die als Parameter übergeben werden, nicht in ein Register passen. Da auch der *this*-Pointer in CPL als eine solche Referenz dargestellt werden muss, tritt dieser Fall extrem häufig ein. Eine weitere Optimierungsvariante müsste also den Einsatz von Referenzen, die größer als ein Maschinenwort sind, wo immer möglich vermeiden. Dies ist sehr einfach immer dann zu realisieren, wenn durch die bisher geschilderten Verfahren der Aufruf von *type_dispatch* vermieden wird.

Weitere Ergebnisse und Messungen finden sich in [Orth99].

```
void VRBauteilTragwerk::zeichnen(Pointer p, Type stype, Bool css) {
  Cursor c(_geometrien);
  for (c.first(); c.more(); c.next()) {
    VRGeometrie3D::zeichnen(c.retrieve(), 10, true);
  }
}
void VRGeometrie3D::zeichnen(Pointer p, Type stype, Bool css) {
  Cursor c(_flaechen);
  for (c.first(); c.more(); c.next()) {
    VRFlaeche::zeichnen(c.retrieve(), 11, true);
  }
}
```

Abbildung 63: generierte Spezialisierungen der Funktionen *Bauteil::zeichnen* und *Geometrie::zeichnen*

5 Ausblick

Die in dieser Arbeit vorgestellten Konzepte wurde komplett in einem Compiler nebst Standard-Bibliothek und Debugging-Umgebung implementiert [BeFl96, CBBB98]. Als Anwendungen wurden zwei Bauingenieur-Systeme, die von den Forschungsgruppen der TH Darmstadt [MPP98] und der Ruhr Universität Bochum [HKB98] in CPL entwickelt wurden, realisiert. Beide Systeme haben je einen Umfang von ca. 20.000 Zeilen CPL Quellcode.

Inzwischen wurden Teile dieser Arbeit auch in kommerziellen Systemen genutzt bzw. reimplementiert. So wurden seit Mitte des Jahres 2000 in einem großen Telekommunikations-Projekt die Mechanismen aus Abschnitt 4.2.5.2 realisiert, um persistente Objekte auch nachträglich effizient mit weiteren Attributen versehen zu können.

Die Präsentation von Perspektiven aus dieser Arbeit soll jedoch nicht darüber hinwegtäuschen, dass einige für einen praktischen Einsatz wichtige Fragen bisher nicht vollständig beantwortet wurden. So erlaubt die in Abschnitt 4.3 Technik der globalen Optimierung zwar eine sehr mächtige und weitgehende Eliminierung des Overheads von Perspektiven und darüber hinaus in sehr vielen Fällen auch die Einsparung von traditionellem dynamischen Binden. Globale Optimierung macht aber die Nutzung von *shared libraries* unmöglich, da zu deren Übersetzungszeit die Anwendungskontexte der in den shared libraries enthalten Methoden und Klassen nicht bekannt sind. Andererseits sind aus sehr praktischen Erwägungen shared libraries im industriellen Einsatz nicht wegzudenken, da sie einerseits eine enorme Reduzierung des Speicheraufwands für den Code von Anwendungen darstellen, und andererseits auch den Zeitaufwand für das Erstellen von Applikationen („*build*“) erheblich reduzieren helfen[26].

Allerdings sind seit [HCU91, HöUn94] und seit kurzem auch durch [HotSpot] Techniken bekannt und implementiert, die Teile der hier beschriebenen Optimierungstechniken zur Laufzeit ausführen oder sogar noch weiterführen, weil zur Laufzeit gewonnene Daten über Empfängertypen an call sites für die Optimierung genutzt werden. Damit brauchen zur Linkzeit von Anwendungen nicht

[26] Das Argument des verbesserten Tunrarounds mag nebensächlich erscheinen, hat in der Industrie jedoch sehr stark spürbare Auswirkungen auf die Produktivität von Entwicklern. Wenn man bedenkt, dass in sehr großen Anwendungen bei shared libraries teilweise Linkzeiten um Stunden reduziert werden mag man ermessen, dass ein Verzicht auf diese Technologie schwer zu verkraften ist.

alle Kontexte bzw. call sites bekannt zu sein. Diese Technik erlaubt sogar das Laden von Klassen bzw. Code zur Laufzeit einer Anwendung, ohne auf das Potential der Optimierungstechniken zu verzichten. Eine genaue Untersuchung der in dieser Arbeit diskutierten Vorgehensweise zur Optimierung auf ihre Eignung für on-the-fly Optimierung steht jedoch noch aus.

In dieser Arbeit wurde als Anwendung von Perspektiven ein System benutzt, welches auf der Nutzung persistenter Objekt in verschiedenen Disziplinen des Bauingenieurwesens beruhte. Perspektiven sind aber auch in völlig anderen Zusammenhängen sehr vorteilhaft einsetzbar.

Moderne Software muss sich schnell und einfach aus vorgefertigten Komponenten zusammenstecken lassen. Schon jetzt geschieht beim täglichen Arbeiten mit Web-Browsern ein ständiges und weitgehend transparentes dynamisches Zusammenstecken einer Anwendung durch Laden von sogenannten „*Plug-Ins*“. Komplexere und mächtiger Komponentenmodelle sind z.B. „*Enterprise Java Beans*“, wie sie im Zusammenspiel mit Application-Servern genutzt werden, oder das schon klassisch zu nennende Komponentenmodell der Object Management Group (OMG), die *Common Object Request Broker Architecture* (CORBA). Die bisherige Praxis bzw. Erfahrungen mit derartigen Komponentenmodellen und Architekturkonzepten zeigt jedoch, dass die Hoffnung, eine maßgebliche Erhöhung der Wiederverwendbarkeit von Software durch Wiederverwendung einer bereits implementierten Komponente zu erreichen, getrogen hat. Denn selbst bei einer fast unübersehbaren Zahl an kommerziell oder frei erhältlichen Komponenten ist in den seltensten Fällen genau die Komponente mit der Schnittstelle und dem Verhalten verfügbar, welche die in einem neuen Anwendungskontext notwendige Funktionalität implementiert. Anpassungen an diesen bereits existierenden Komponenten, möglichst ohne Source-Code oder andere Interna kennen zu müssen, sind deshalb unumgänglich um die Produktivität durch Wiederverwendung von etablierten, getesteten Software-Komponenten zu erhöhen.

[Oder00] schlägt einen View-Mechanismus vor, um Klassen nachträglich mit zusätzlichen Methoden zu versehen und damit Komponenten, die derartige Klassen enthalten, an Änderungen adaptierbar zu machen, die zum Zeitpunkt ihres Designs nicht vorhersehbar waren. Die Verwendung von polymorphen Unterklassen und insbesondere die Möglichkeit der Definition von *share*-Blöcken erlaubt ebenso eine nachträgliche Anpassung von Funktionalität durch Hinzufügung neuer Klassen – ohne Manipulation der bestehenden Codes. Betrachten wir zum Beispiel eine Komponente oder Bibliothek, welche das Logging einzelner Benutzeraktionen in

Dateien implementiert. Um eine solche Komponente wiederzuverwenden, auch wenn zusätzlich Informationen im Logging berücksichtigt werden sollen, eignen sich Perspektiven wegen der share-Blöcke hervorragend. Ohne Veränderung des bestehenden Codes, ja sogar ohne dass der Quellcode überhaupt vorhanden sein muss, kann ein System sehr maßgeschneidert angepasst werden und global oder sehr selektiv mit neuer, angepasster Funktionalität angereichert werden.

Damit tragen Perspektiven ganz wesentlich zur Erfüllung der Forderung bei, bestehende Software und Software-Komponenten ausschließlich durch Hinzufügung neuer Klassen, nicht durch Manipulation bereits existierenden Codes, an neue Gegebenheiten anzupassen. Wie wir in Abschnitt 2.3 sehen konnten, ist dies ohne Erweiterung der Typsysteme klassischer objektorientierter Sprachen nur sehr eingeschränkt möglich. Und selbst dies eingeschränkte Art der Anpassung durch Erweiterung geht nur dann, wenn Architektur und Design der anzupassenden Komponente diese Anpassungen bereits in hohem Maße antizipiert haben.

6 Literatur

[AbBo91] ABITEBOUL, S.; BONNER, A.: *Objects and Views*, Proceedings SIGMOD, May 1991

[ACMT93] ATZENI, P.; CABBIBO, L.; MECCA, G.; TANCA, L.: *The logidata+ language and semantics*. In LOGIDATA+: Deductive Databases with Complex Objects, No. 701 in LNCS, Springer Verlag 1993

[AgHö95] AGESEN, O.; HÖLZLE. U.: *Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages*, OOPSLA'95

[AiHö95] AIGER, G.; HÖLZLE, U.: *Eliminating Virtual Function Calls in C++ Programs*, University of California, Santa Barbara, Technical Report TRCS 95-22

[BaKe93] BARCLAY, P. J.; KENNEDY, J. B.: *Viewing Objects*, Advances in Databases, Lecture Notes in Computer Science, Springer 1993

[BaSw96] BACON, D. F.; SWEENEY, P. F.: *Fast Static Analysis of C++ Virtual Function Calls*, OOPSLA '96, San Jhose, CA, USA 1996

[BBCR94] BERGMANN, A.; BODE, Th.; CREMERS, A. B.; REDDIG, W.: *Modeling Civil Engineering Constraints with Inter Object Relationships*, Proc. of the 1st ECPPM, Balkema Publishers, 1994

[BBCR95] BERGMANN, A.; BODE, Th.; CREMERS, A. B.; REDDIG, W.: *Integrating Civil Engineering Applications with Object-Oriented Database Systems*, Proc. of the 6th ICCCBE, Balkema Publishers, 1995

[BCR96] BALOVNEV, O. T.; CREMERS, A. B.; REDDIG, W.: *Updating Polymorphic Objects*, WOON'96 Int. Conf. on Object-Oriented Technology, Saint-Petersburg

[BeFl96] BERGER, T.; FLACHBART, D.: *CEBRA – ein generischer objektorientierter Datenbank-Browser für das CEMENT-System*, Diplomarbeit, Institut für Informatik III, Universität Bonn, 1996

[Bell00] BELLAHSENE, Z.: *Updates and object-generating views in ODBMS*, Data and Knowledge Engineering (34) 2000, pages 125 - 163

[Bert92] BERTINO, E.: *A View Mechanism for Object-Oriented Databases*, Advances in Database Technology EDBT ' 92

[BeGu95] BERTINEO, E.; GUERRINI, G.: *Objects with Multiple Most Specific Classes*, Proc. of ECOOP'95

[BGR99] BERTINO, E.; GUERRINI, G.; RUSCA, L.: *Object Evolution in Object Databases*, In B. Franhoefer and R. Pareschi, editors, Dynamic Worlds, pages 219-246. Kluwer Academic Publishers, 1999

[BLT86] BLAKELEY, L. A.; LARSON, P. A.; TOMPA, F. W.: *Efficiently Updating Materialized Views*, SIGMOD '86, Washington, USA, 1986

[ByMc93] BYEON, K. J.; MCLEOD, D.: *Towards the Unification of Views and Versions for Object Databases*, Lecture Notes in Computer Science 742, 1993

[CaGr94] CALDER, B.; GRUNEWALD, D.: *Reducing indirect function calls in C++ programs*, Principles of Pgogramming Languages (POPL), Oregon, USA, Jan. 1994

[CBBB98] CREMERS, A. B.; BERGMANN, A.; BERSE, H.; Bode, T.; BREUNIG, M.; COSTANZA, P.; FLACHBART, D.; KNIESEL, G.; REDDIG, W.: *Definition und Implementierung eines Bauwerkmodellkerns*, Objektorientierte Modellierung in Planung und Konstruktion, Deutsche Forschungsgemeinschaft (DFG), herausgegeben von D. Hartmann, Wiley-VCH, 1998

[CBR94] CREMERS, A. B.; BALOVNEV, O. T.; REDDIG, W.: *Views in Object-Oriented Databases*, ADBIS'94, Moskau, Russland, 1994

[CeMa94] CERI, S.; MANTHEY, R.: *Chimera: a model and language for active dood systems*, Proc. of the East/West Database Workshop, pages 3 – 16, 1994

[CeWi91] CERI, S.; WIDOM, J.: *Deriving Production Rulse for Incremental View Maintenance*, VLDB '91, Barcelona, Spain, 1991

[CFP94] CERI, S.; FRATERNALI, P.; PARABOSCHI, S.: *Automatic Generation of Production Rules for Integrity Maintenance*, Transactions On Database Systems 19(3): 367-422 1994

[ChUn89] CHAMBERS, C.; UNGAR, D.: *Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Language*, SIGPLAN '89, Oregon, USA, 1989

[DHV95] DRIESEN, K.; HÖLZLE, U.; VITEK, J.: *Message Dispatch on Pipelined Processors*, ECOOP '95, 1995

[DiHö97] DIECKMANN, S.; HÖLZLE, U.: *The Space Overhead of Customization*, University of California, Santa Barbara, TRCS 97-21, 1997

[DMM96] DIWAN, A.; MOOS, J. E. B.; MCKINLEY, K. S.: *Simple and Effective Analysis of Statically-Typed Object-Oriented Programs*, OOPSLA ’96, California, USA, 1996

[DrHö96] DRIESEN, K.; HÖLZLE, U.: *The Direct Cost of Virtual Function Calls in C++*, OOPSLA ’96, California, USA, 1996

[EdDo94] EDER, J.; DOBROVNIK, M.: *Adding View Support to ODMG 93*, ADBIS’94, Moscow

[ElSt90] ELLIS, M. A. ; STROUSTRUP, B. : *The Annotated C++ Reference Manual*, Addison Wesley, 1990

[FrPa93] FRATERNALI, P.; PARABOSCHI, S.: *A Review of Repairing Techniques for Integrity Maintenance*, Rules in Database Systems 1993: 333-346, Workshops in Computing, Springer

[GBCM97] GUERRINI, G.; BERTINO, E.; CATANIA, B.; GARCIA-MOLINA, J.: *A Formal Model of Views in Object-Oriented Database Systems*, Theory and Practice of Object Systems 3 (3), 1997

[GDC95] GROVE, D.; DEAN, J.; CHAMBERS, C.: *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis*, ECOOP ’95, August 1995

[GDDC97] GROVE, D.; DEFOUW, G.; DEAN, J.; CHAMBERS, C.: *Call Graph Construction in Object-Oriented Languages*, OOPSLA ’97, 1997

[GeJa91] GEHANI, N. H.; JAGADISH, H. V.: *Ode as an Active Database: Contraints and Triggers*, Proc. of the 17th Int. Conf. on Very Large Databases, 1991

[GHJV94] GAMMA, E.; HELM, R.; JOHNSON, R; VLISSIDES, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley, 1994

[GeDi94] GEPPERT, A.; DITTRICH, K. R.: *Specification and Implementation of Cosistency Constraints in Object-Oriented Database Systems: Applying Programming by Contract*, Proc. of BTW94, Dresden, Germany, 1994

[GMS93] GUPTA, A.; MUMICK, I. S.; SUBRAHMANIAN, V. S.: *Maintaining Views Incrementally*, SIGMOD ’93, 1993

[HCU91] HÖLZLE, U.; CHAMBERS, C.; UNGAR, D.: *Optimizing Dynamically-Typed Object-Oriented Languages witrh Polymorphic Inline Caching*, ECOOP ‘91

[HKB98] HARTMANN, D.; KOLENDER, U.; BRETSCHNEIDER, D.: *Entwicklung eines integrierten Programmsystems für den industriellen Stahlhochbau unter Verwendung objektorientierter Methoden*, In P. Wriggers and U. Meißner, editors, Finite Elemente in der Baupraxis, Ernst & Sohn, Berlin, 1998

[HöUn94] HÖLZLE, U.; UNGAR, D.: *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*, SIGPLAN ‘95 Notice 29(6), June 1994

[HöUn95] HÖLZLE, U.; UNGAR, D.: *Do Object-Oriented Languages Need Special Hardware Support?* ECOOP ‘95

[HotSpot] http://java.sun.com/products/hotspot/index.html

[JaLa92] JAMIL, H. M.; LAKHSHMANAN, L. V. S.: *Orlog: A logic for semantic object-orientied models*. In Proc. of the ACM Conference in Knowledge Management – CIKM 1992

[JGJS94] JARKE, M.; GALLERSDORFER, R.; JEUSFELD, M.; STAUFT, M.: *Conceptbase – a deductive object base for meta data management*. Journal of Intelligent Information Systems, 3:167 – 192, 1994

[KLW95] KIFER, M.; LAUSEN, G.; WU, J.: *Logical foundations of object-oriented and frame-based languages*. Journal of the ACM, May 1995

[Knie96] KNIESEL, G.: *Objects Don’t Migrate – Perspectives on Objects with Roles.* Technical Report IAI-TR-96-11, Institut für Informatik III, Universität Bonn. ISSN 0944-8535. April 1996

[Knie00] KNIESEL, G.: *Dynamic Object-Based Inheritance with Subtyping*. Dissertation, Institut für Informatik III, Universität Bonn, Juli 2000.

[KrPo88] KRASNER, G. E.; POPE, S. T.: *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*, Journal of Object-Oriented Programming, 1(3): 26-49, August/September 1988

[KRR95] KUNO, H. A.; RA, Y. G.; RUNDENSTEINER, E. A.: *The Object Slicing Technique: A flexible Object Representation and its Evaluation*, Technical report, University of Michigan, 1995

[Lea90] LEA, D.: *Customization in C++*, Proceedings of the 1990 Usenix C++ Cponference, California, USA, 1990

[Liu96] LIU, M.: Rol: *A deductive object base language*. Information Systems, 21(5):431 – 457, 1996

[LiWi93] LISKOV, B.; WING, J. M.: *A New Definition of the Subtype Relation*, ECOOP ’93, Germany 1993

[LMN93] LEHRMANN MADSEN, O.; MOLLER-PEDERSEN, B.; NYGAARD, K.: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley and ACM Press, 1993

[Meye92] MEYER, B.: *Eiffel: The Language*, Prentice Hall, 1992

[Meye98] MEYER, B.: *Object-Oriented Software Construction (2nd edition)*, Prentice Hall, 1998

[MPP98] MEISSNER, U.; PETERS, F.; PETERSEN, M.: *Objektorientierte Teilproduktmodelle für die Systemintegration von Planungs- und Konstruktionsvorgängen im Bauwesen*, Objektorientierte Modellierung in Planung und Konstruktion, Deutsche Forschungsgemeinschaft (DFG), herausgegeben von D. Hartmann, Wiley-VCH, 1998

[Oder00] ODERSKY, M.: *Objects + Views = Components?* Abstract State Machines 2000, Monte Verita, Switzerland, March 2000, to appear in Springer Lecture Notes.

[ODI99] Object Design, Incorporated: *ObjectStore User Guide*, Burlington, Massachusetts, 1999

[Orth99] ORTH, M.: *Technik und Implementation des optimierenden Compilers der CEMENT Programmiersprache CPL*, Diplomarbeit am Institut für Informatik III, Universität Bonn, 1999

[PaJa98] PALSBERG, J.; JAY, C. B.: *The Essence of the Visitor Pattern*, COMPSAC ,98, Austria, 1998

[PaSc94] PALSBERG, J.; SCHWARTZBACH, M. I.: *Object-Oriented Type Systems*, John Wiley & Sons, New York, London, Sydney, 1994

[RBPE93] RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F., LORENSEN, W.: *Object-Oriented Modeling and Design*, Prentice-Hall, 1993

[Rund92] RUNDENSTEINER, E. A.: *A Methodology for Supporting Multiple Views in Object-Oriented Databases*, Proc. of the 18th VLDB Conference, 1992

[Schi93] SCHIEFER, B.: *Objektorientierte Sichten für Objektorientierte Datenbanksysteme*, Dissertation am Forschungszentrum Informatik, Karlsruhe, Deutschland, 1993

[SLT91] SCHOLL, M. H.; LAASCH, C.; TRESCH, M.: *Updatable Views in Object-Oriented Databases*, Proceedings of the 2nd DOOD Conference, 1991

[SAD94] SOUZA DOS SANTOS, C.; ABITEBOUL, S.; DELOBEL, C.: *Virtual Schemas and Bases*, EDBT94

[SaPa97] SAMPAIO, P. R. F.; PATON, N. W.: *Deductive Object-Oriented Database Systems: A Survey*, In: Greppet A. Berndtsson M. Eds. Rules in Database Systems'97 Proceedings, LNCS 1312

[SKW92] SINGHAL, V.; KAKKAD, S. V.; WISLON, P.: *Texas: an Efficient, Portable Persistent Store*, Proc. Fifth Int' l. Workshop on Persistent Object Systems, Sept. 1992, San Miniato, Italy

[Stro94] STROUSTRUP, B.: *Design and Evolution of C++*, Addison Wesly, 1994

[WiKa92] WILSON, P.; KAKKAD, S. V.: *Pointer Swizzling at Page Fault Time: Efficiently and and Compatibly Supporting Huge Address Spaces on Standard Hardware*, Proc. Second Int' l. Worksbp on Object Orientation in Operating Systems, Sept. 1992, Dourdan, France

[ZCC98] ZENDRA, O.; COLNET, D.; COLLIN, S.: Efficient Dynamic Dispatch without Virtual Function Tables, OOPSLA'98