GO TO COM ISBN 3-8273-1678-2

Schnittstellenbasierte Programmierung

2

Kapitelüberblick

2.1 Einführung

Die immer wiederkehrenden Probleme einer Softwareentwicklung (Kosten- und Terminüberschreitungen; Ergebnisse, die zum Teil mit zu vielen Fehlern behaftet sind und im Extremfall auf Grund der enormen Fluktuation an Entwicklern nicht einmal mehr wartbar sind) führen im Laufe der Jahre immer wieder zu neuen Überlegungen und Konzepten, wie man den Prozess der Softwareentwicklung verbessern kann. Das Paradigma der *schnittstellenbasierten* Programmierung ist ein Beispiel dafür, wie über die Jahre hinweg eine neue Methodik entstanden ist, die sich heutzutage quasi als de facto Standard etabliert hat.

2.2 Ein neues Programmierparadigma: Die Schnittstelle

Da C++ schon einige Jahre industriellen Einsatzes hinter sich gebracht hat, haben die COM-Architekten neben der Würdigung der Verdienste, die sich C++ als objektorientierte Programmiersprache während dieser Zeit ohne jeden Zweifel erworben hat, auch einen Blick auf die kritisierten Eigenschaften geworfen.

Die Zusammenfassung von *Status* (Zustand von Daten, Gedächtnis), *Verhalten* (Methoden, die auf den Daten operieren) und *Identität* (eindeutige Identifizierung eines Objekts) unter dem gemeinsamen Dach eines Objekts war seit Ende der 80er Jahre anerkannt. Gleichermaßen war ein weiteres Paradigma, die Vererbung, der probate Mechanismus, um gemeinsame Softwareteile an zentraler Stelle zusammenzufassen. Obwohl nach diesem Verfahrensmuster unzählige Klassenbibliotheken von Drittherstellern entstanden sind, führte dieses auch als »*implementierungsvererbende Objektorientierung*« bezeichnete Paradigma der OOP nie zu uneingeschränkter Akzeptanz bei den Softwareentwicklern. Immer wieder wurde bemängelt, dass die Klassenbibliotheken trotz ihrer verfügbaren Funktionalität für weitere Spezialisierungen nie oder nur selten zu gebrauchen waren, da im Detail immer wieder diverse Fragen und Probleme auftauchten. In einem ersten Schritt umgingen viele Hersteller von Klassenbibliotheken dieses Problem dadurch, dass sie zu ihrem Produkt auch den Quelltext (unentgeltlich oder auch nicht) beigefügt haben.

Das hat zwar den einen oder anderen Entwickler zunächst erfreut, da er sich mal ansehen konnte, wie Quelltext quasi von Profis geschrieben wird (sofern denn Profis am Werk waren ...). Aus Sicht des Software-Entwicklungsprozesses war das aber ein ganz klarer Schritt rückwärts, da von diesem Zeitpunkt an für die eigene Software-Entwicklung das Studium fremder Quellen Voraussetzung war. Dazu hatte man noch ein weiteres Problem vor seiner Haustüre: Was tun, wenn man bei der eigenen Programmentwicklung auf intime Details der fremden Software gesetzt hat,

die in einer der nächsten Releases der Klassenbibliothek nicht mehr bzw. nur noch in geänderter Form vorhanden waren? All diese Probleme hatten letztlich die Konsequenz, dass sich trotz gebetsmühlenartig wiederholender Werbung für die Verwendung von Klassenbibliotheken (vor allem von Seiten des Software-Managements) diese sich nie in uneingeschränktem Maß beim Entwickler durchgesetzt haben.

Programmierer und Software-Architekten aus der Ära der implementierungsvererbenden Objektorientierung haben vor diesem Hintergrund nach neuen Konzepten gesucht und wurden bei einem Paradigma, das man als »*schnittstellenbasierte Objektorientierung*« bezeichnet hat, fündig. Bei dieser Art des objektorientierten Programmierentwurfs sind die Schnittstelle einer Methode und deren Implementierung strikt getrennt. Die Schnittstelle selbst (also ihre Definition) bringt nur die Eigenschaften zum Ausdruck, die irgendein Objekt, das diese Schnittstelle implementiert, in der konkreten Realisierung zu erbringen hat.

Jede Schnittstelle, die von einem Objekt zur Verfügung gestellt wird, übernimmt die Rolle eines Kontraktes zwischen Objekt und Client. Das Objekt verpflichtet sich, die Methoden gemäß den Definitionen der Schnittstelle exakt zu erfüllen. Der Client ist verpflichtet, die Methoden ausschließlich unter Beachtung dieser Regeln aufzurufen (und insbesondere nur solche Aktualparameter zu benutzen, die nach dem Kontrakt zulässig sind).

Zur Einhaltung dieses Kontraktes steht keinerlei Quelltext mehr zur Verfügung, der für weitere Interpretationen des Kontraktes inspiziert werden kann. Der Vertrag zwischen Objekt und Client ist auf einer Basis abzuwickeln, die sich ausschließlich auf der Ebene der Definition dieser Schnittstelle abspielt.

Schnittstellenbasierte Entwicklung ist eine der Grundlagen von COM, aber auch Java hat dieses Paradigma in seine Sprachdefinition mit aufgenommen (interessanterweise in Kombination mit dem Paradigma der »implementierungsvererbenden Objektorientierung«).

Für ein tiefer gehendes Verständnis der schnittstellenbasierten Programmierung in COM ist die Kenntnis des virtuellen Methodenaufrufs von C++ oder Java erforderlich. Sofern Sie sich damit auskennen, überspringen Sie doch einfach das nächste Kapitel.

2.3 Virtueller Methodenaufrufmechanismus in C++

Wie bei den Programmiersprachen ist es für COM genauso hilfreich, ein nicht zu oberflächliches Verständnis für die Abläufe in den tiefer liegenden Schichten zu besitzen. Viele Fragen wie etwa »Was um Himmels Willen macht denn mein C++-Compiler hier?« lassen sich eben nur dann

beantworten, wenn Sie neben den Konzepten eines Programmier-Paradigmas auch ein gewisses Know-how über die Implementierung besitzen. Darum werfen wir in mehreren Abschnitten dieses Buchs einen Blick hinter die Kulissen der schönen Fassade eines Quelltextes, um zu erkennen, wie gewisse COM-Aspekte in der Realität aussehen.

Bevor wir uns auf COM-relevante Implementierungsdetails stürzen, werfen wir einen Blick auf die Programmiersprachen C++ (und auch Java). Es wird Sie nicht weiter überraschen, wenn wir bereits an dieser Stelle verraten, dass COM-Objekte sich sehr elegant in C++ realisieren lassen. Zu diesem Zweck machen wir uns mit den Mechanismen, wie C++-Klassen bzw. deren Instanzen zur Laufzeit im Speicher abgelegt werden, etwas näher vertraut.

Bemerkung Für die Implementierung eines C++-Compilers gibt es natürlich eine Reihe bisweilen ganz unterschiedlicher Ansätze. Sinn und Zweck der folgenden Erörterungen ist es ausschließlich, einen unter mehreren gangbaren Wegen aufzuzeigen, eben um dem Leser eine Vorstellung von den Vorgängen zu vermitteln, die sich im Stockwerk »unterhalb des Quelltextes« abspielen. Da alle Beispiele dieses Buches mit dem Microsoft Visual C++-Compiler erstellt und getestet wurden, wurde darauf geachtet, dass die Implementierungshinweise für dieses Produkt zutreffen.

2.3.1 Speicherlayout einer Struktur in C

Fangen wir mit dem Vorläufer des C++-Konstrukts `class`, dem Sprachelement `struct` aus der Programmiersprache C, an und betrachten beispielsweise eine Struktur bestehend aus zwei Elementen, einem Zeichen `c` und einer ganzen Zahl `i`:

```
struct A {
  char c;
  int i;
};
```

Die einzelnen Elemente einer Instanz von `struct A` werden im Hauptspeicher Element für Element abgelegt (siehe Abbildung 2.1), in der Regel hält man sich an die Deklarationsreihenfolge innerhalb der Struktur. Elemente, die in Bezug auf die Anzahl der Bytes, die in ein CPU-Register passen, weniger Bytes in Anspruch nehmen, werden durch so genannte *padding bytes* (Füllbytes) ergänzt, so dass das nächste Element der Struktur wieder an einer für die CPU optimal adressierbaren Speicheradresse zum Liegen kommt.

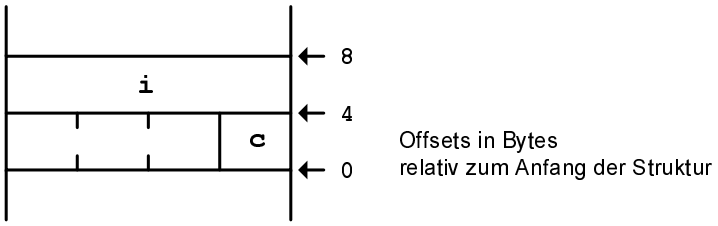

Abbildung 2.1: Die Datenablage einer C-Struktur im Hauptspeicher

2.3.2 Virtuelle Methoden in C++

Betrachten wir nun Klassen in C++: Hier nimmt der Begriff des virtuellen Methodenaufrufmechanismus eine zentrale Rolle ein. Dieses Merkmal der Programmiersprache C++, das in C nicht vorhanden ist, führt für viele Umsteiger von C nach C++ daher immer wieder zu Verständnisschwierigkeiten. Am geschicktesten beginnen wir die Erläuterung, indem wir einen Blick auf drei C++-Klassen werfen, die zunächst absichtlich ohne den Mechanismus des virtuellen Methodenaufrufs definiert werden:

Listing

```
01: class Figure
02: {
03: public:
04:     void paint ()
05:     {
06:         printf ("error: paint of class Figure called !\n");
07:     }
08: };
09:
10: class Rectangle : public Figure
11: {
12: public:
13:     void paint ()
14:     {
15:         printf ("painting a rectangle ...\n");
16:     }
17: };
18:
19: class Circle : public Figure
20: {
21: public:
22:     void paint ()
23:     {
```

```
24:         printf ("painting a circle ...\n");
25:     }
26: };
```

Listing 2.1: Ein Fallbeispiel bestehend aus einer Basisklasse `Figure` und zwei davon abgeleiteten Klassen `Rectangle` und `Circle`

`Circle` und `Rectangle` beschreiben zwei Klassen, die jeweils eine bestimmte geometrische Figur darstellen. Aus Gründen der Systematisierung definieren wir eine dritte Klasse `Figure`, die für alle geometrischen Figuren die Rolle der Basisklasse übernimmt.

Wesentlich an den beiden Klassen `Circle` und `Rectangle` ist, dass sie die Methode `paint`, die ja an sich schon in der Basisklasse `Figure` vorhanden ist, noch einmal definieren, um sie eben mit einer jeweils für die abgeleitete Klasse spezifischen Realisierung auszustatten. `Rectangle::paint` könnte ein Rechteck mit einer bestimmten Länge und Breite zeichnen, `Circle::paint` wiederum einen Kreis, der einen bestimmten Radius besitzt. Der Einfachheit halber habe ich in Wirklichkeit nur simple `printf`-Anweisungen programmiert, die die jeweilige Methode identifizieren.

Haben wir einen grafischen Container, der eine unbestimmte Anzahl von Kreisen und Rechtecken aufnehmen kann, dann ist es auf Grund der Ausprägung von `Rectangle` und `Circle` als Spezialisierung der Basisklasse `Figure` möglich, all diese konkreten Figuren in einem gemeinsamen Array abzuspeichern, dessen Datentyp durch `Figure` definiert ist:

```
Figure * someFigures [3];
```

Da jedes Rechteck und auch jeder Kreis einen Spezialfall einer Figur darstellt, also von `Figure` abgeleitet ist, kann zur Laufzeit ein Container folgende Belegung von `someFigures` bewirken:

```
someFigures [0] = new Rectangle ();
someFigures [1] = new Circle ();
someFigures [2] = new Rectangle ();
```

Die Wertzuweisung von abgeleiteten Klassen an ihre Basisklasse ist völlig legal, da jede Instanz von `Rectangle` oder `Circle` ja insbesondere auch eine Instanz von `Figure` ist (alle Daten der Basisklasse sind in der abgeleiteten Klasse ebenfalls vorhanden). Wendet man nun die Methode `paint` auf alle in dem Array befindlichen Figuren an, dann würde man sich natürlich wünschen, dass je in Abhängigkeit von der tatsächlich vorliegenden Figur entweder `Rectangle::paint` oder `Circle::paint` aufgerufen wird:

```
for (int i = 0; i < 3; i ++)
  someFigures [i] -> paint ();
```

In der `for`-Schleife wird bei jedem Aufruf von `paint` jedes Mal entweder ein `Circle` oder ein `Rectangle` angesprochen. Obwohl jede dieser Klassen eine überschriebene Variante von `paint` besitzt, erhalten wir die Ausgabe

```
error: paint of class Figure called !
error: paint of class Figure called !
error: paint of class Figure called !
```

Wir sind an die Grenzen der Leistungsfähigkeit von Methodenaufrufen in der von uns programmierten Art und Weise gestoßen. Der ablaufende Code hält sich exakt an die Vereinbarungen, die im Quelltext stehen, sprich die Definition von `someFigures` als Variable des Typs `Figure` sowie der Aufruf `someFigures [i] -> paint` veranlassen den Übersetzer, die zur Übersetzungszeit vorliegende Information für die Code-Generierung zu Grunde zu legen. Abgeleitete Methoden von `paint` haben so niemals eine Chance, zur Laufzeit ausgeführt zu werden (man spricht daher auch von der so genannten *statischen Bindung* oder auch von *early-binding*), der *compile-time*-Datentyp des Zeigers ist für den Aufruf der Methode zur Laufzeit entscheidend.

Wie erreichen wir nun, dass der Methodenaufruf

```
someFigures [i] -> paint ();
```

die Methode des Objekts aufruft, das in Wirklichkeit erzeugt wurde? Des Rätsel Lösung lautet: Man benutze das Schlüsselwort `virtual` bei der Definition der Methoden! Dieses zusätzliche Attribut hat zur Folge, dass zur Laufzeit die Methoden `paint` von `Rectangle` und `Circle` so aufgerufen werden, wie man es sich vorstellt. `virtual` bedeutet, dass beim Aufruf einer solchen Methode nicht die Definition der Methode zur Übersetzungszeit, sondern das tatsächlich zur Laufzeit vorliegende Objekt entscheidet, an welchem Objekt die Methode aufgerufen wird:

Listing

```
01: class Figure
02: {
03: public:
04:     virtual void paint () = 0;
05: };
06:
07: class Rectangle : public Figure
08: {
09: public:
10:     virtual void paint ()
11:     {
12:         printf ("painting a rectangle ...\n");
13:     }
```

```
14: };
15:
16: class Circle : public Figure
17: {
18: public:
19:     virtual void paint ()
20:     {
21:         printf ("painting a circle ...\n");
22:     }
23: };
```

Listing 2.2: Das Fallbeispiel aus Listing 2.1 wird mit Hilfe des Schlüsselworts `virtual` entscheidend verbessert

Mit diesen Methodendefinitionen erhalten wir als Ergebnis:

```
painting a rectangle ...
painting a circle ...
painting a rectangle ...
```

Der virtuelle Funktionsaufruf ist eines der herausragenden Merkmale von Programmiersprachen wie C++ oder Java. Virtuelle Methoden sind immer dann zu verwenden, wenn man zum Beispiel eine existierende Implementierung einer Basisklasse weiter verfeinern oder ggf. sogar ersetzen möchte, falls die vorhandene Implementierung für die konkrete Spezialisierung der Basisklasse nicht benutzbar ist. Hätte man keine virtuellen Methoden, dann müsste man einen Code-Abschnitt wie in Listing 2.2 etwa wie folgt programmieren:

Listing

```
01: #define RECTANGLE  1
02: #define CIRCLE     2
03:
04: for (i = 0; i < 3; i ++)
05: {
06:     switch (someFigures [i] -> kind)
07:     {
08:     case RECTANGLE:
09:         ((Rectangle *) someFigures [i]) -> paint ();
10:         break;
11:     case CIRCLE:
12:         ((Circle *) someFigures [i]) -> paint ();
13:         break;
14:     }
15: }
```

Listing 2.3: Ein Versuch, virtuelle Methoden in einer Programmiersprache nachzuahmen, die keinen virtuellen Methodenaufrufmechanismus besitzt (hier: C)

Das geht selbstverständlich und wird einem langjährigen C-Programmierer auch sehr vertraut vorkommen, hat aber doch eine ganze Reihe von gewichtigen Nachteilen:

- Leitet man eine weitere Klasse von `Figure` ab, so muss jedes Mal die `switch`-Anweisung ergänzt werden
- Die `for`-Schleife ist weit umfangreicher als die vergleichbare Variante mit virtuellen Methoden, damit unübersichtlicher und verursacht umfangreicheren Code
- Die Basisklasse muss eine zusätzliche Instanzvariable `kind` besitzen, die zur Laufzeit den tatsächlichen Typ der abgeleiteten Klasse beschreibt

Bemerkung

Nebenbei bemerkt: Haben Sie einen Blick auf die Klasse `Figure` in Listing 2.2 geworfen? In dieser Variante habe ich die Methode `paint` mit einem weiteren Sprachkonstrukt von C++ ausgestattet: Da `paint` an Instanzen der Klasse `Figure` mit Sicherheit nie aufgerufen wird, sollte man das in einem Programm auch zum Ausdruck bringen: Man definiert in solch einer Situation die Methode als *abstrakt* oder, wie man in C++ dazu sagt, als *pure virtual*: Das Suffix »`= 0;`« (Zeile 4 in Listing 2.2) soll in sehr maschinennaher Vorstellung zum Ausdruck bringen, dass die Methode in dieser Klasse nur durch einen `NULL`-Zeiger repräsentiert wird, es bleibt den abgeleiteten Klassen vorbehalten, zu einer konkreten Realisierung von `paint` beizutragen (und damit auch zu einer entsprechenden Adresse im Codesegment, die eben ungleich `NULL` ist). Von der Klasse `Figure` selbst lassen sich überhaupt keine Instanzen erzeugen!

Sollten Sie mehr mit Java als mit C++ vertraut sein, fragen Sie sich vielleicht nach der Lektüre der letzten Seiten: Schlüsselwort `virtual`, »*pure virtual*« definierte Methoden, wie groß sind denn die Unterschiede zwischen C++ und Java? Vorsicht – wenn Sie annehmen, dass die Unterschiede sehr groß sind, dann täuschen Sie sich: Die Unterschiede, denen Sie vermeintlich hier gegenüberstehen, sind nur rein syntaktischer Natur, sprich sie treten nur in der Art und Weise auf, wie C++- oder Java-Programme an der Sprachoberfläche zu programmieren sind. Wirft man einen Blick auf die konzeptionellen Eigenschaften der beiden Programmiersprachen, dann stellt man fest, dass die objekt-orientierten Paradigmen von Java und C++ an dieser Stelle identisch sind! Betrachten wir zu diesem Zweck unsere Rechtecke und Kreise in Java:

Listing

```
01: abstract class Figure
02: {
03:     public abstract void paint ();
04: }
05:
```

```
06: class Rectangle extends Figure
07: {
08:     public void paint ()
09:     {
10:         System.out.println ("painting a rectangle ...");
11:     }
12: }
13:
14: class Circle extends Figure
15: {
16:     public void paint ()
17:     {
18:         System.out.println ("painting a circle ...");
19:     }
20: }
21:
22: class Demo
23: {
24:
25:     public static void main (String argv []) {
26:         Figure someFigures [] = new Figure [3];
27:
28:         someFigures [0] = new Rectangle ();
29:         someFigures [1] = new Circle ();
30:         someFigures [2] = new Rectangle ();
31:
32:         for (int i = 0; i < someFigures.length; i ++)
33:             someFigures [i].paint ();
34:     }
35: }
```

Listing 2.4: Virtuelle Methoden im Vergleich von C++ und Java: Dieses Java-Programm besitzt dieselbe Ausgabe wie das C++-Programm in Listing 2.2

Die offensichtlichsten Unterschiede sind:

- Das Schlüsselwort `virtual` fehlt in Java:

  In Java sind per Voreinstellung *alle* Methoden virtuell. Im Gegenteil, möchte man eine Methode explizit als nicht-virtuell definieren, dann muss man diese mit dem Schlüsselwort `final` kennzeichnen. Für eine mit `final` definierte Methode bedeutet dies, dass eine abgeleitete Klasse diese Methode nicht überschreiben kann.

- Das Suffix »= 0;« gibt es in Java nicht:

  Man muss die Methode statt dessen als `abstract` definieren.

- Die Klasse `Figure` ist als abstract gekennzeichnet:

  Hier ist Java etwas strenger als C++, was die Definition von Klassen betrifft: Ist in einer Klasse mindestens eine Methode abstrakt, dann verlangt der Java-Übersetzer, dass man auch die ganze Klasse als `abstract` definiert. In C++ ist eine solche Klasse natürlich auch von ihrem Wesen her abstrakt, sprich sie ist nicht instantiierbar, nur gibt es hierfür in C++ kein syntaktisches Merkmal an der Klasse.

Gestärkt mit dem Wissen um den virtuellen Methodenaufrufmechanismus in C++ werfen wir nun einen Blick auf die Realisierung dieser Technik.

2.3.3 Speicherlayout einer C++-Klasse mit virtuellen Methoden

Wir starten mit einer Klasse, die virtuelle Funktionen enthält:

Listing

```
01: class A {
02: public:
03:     int a1;                // Datenelement a1
04:     int a2;                // Datenelement a2
05:     virtual int vf_A1 ();   // virtuelle Funktion vf_A1
06:     virtual int vf_A2 ();   // virtuelle Funktion vf_A2
07: };
```

Listing 2.5: Erstes Fallbeispiel einer Klasse A, die zwei virtuell deklarierte Methoden besitzt

Zu Klassen, die eine oder mehrere als `virtual` deklarierte Methoden enthalten, erzeugt der Compiler eine so genannte *virtuelle Funktionszeigertabelle* (engl. *virtual function table*), für die sich in der einschlägigen Literatur auch die Abkürzung *vtbl* eingebürgert hat. Dieses eindimensionale Feld enthält für jede virtuelle Methode einen Funktionszeiger, also einen Pointer, über den die Methode zur Laufzeit indirekt aufgerufen wird. Jede Instanz einer solchen Klasse wiederum besitzt in ihren Instanzdaten einen Zeiger auf diese Tabelle, den so genannten *virtual function table pointer*, kurz auch als *vptr* bezeichnet. Dieser Zeiger ist in den gängigen C++-Implementierungen an erster Stelle im Instanzdatenbereich angesiedelt, siehe Abbildung 2.2.

In Abbildung 2.2 haben wir den `this`-Zeiger verwendet. Dieser Pointer zeigt stets auf die Instanzdaten eines Objekts. Im Falle einer C++-Klasse mit virtuellen Funktionen zeigt der `this`-Zeiger natürlich ebenfalls auf die Instanzdaten, wir erkennen aber in Abbildung 2.2, dass sich in diesem Be-

reich an der ersten Stelle die Adresse der virtuellen Funktionszeigertabelle befindet, erst danach schließen sich die eigentlichen Datenelemente der Instanz an. In einem solchen Fall kann man daher auch sagen, dass »`this` den *vptr* adressiert«.

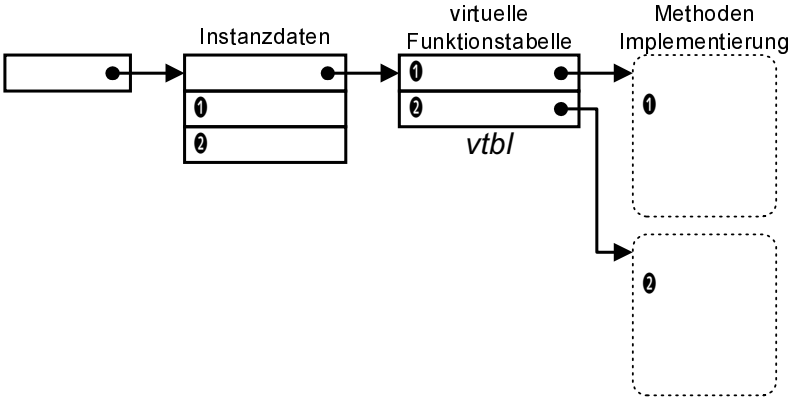

Abbildung 2.2: Ein Blick auf die Instanzdaten einer Klasse mit der so genannten »virtuellen Funktionszeigertabelle«

Welches Konzept verbirgt sich hinter dieser Realisierung? Nun, das Attribut `virtual` einer Methode bedeutet ja, dass diese Methode von einer abgeleiteten Klasse der Gestalt überschrieben werden kann, dass zur Laufzeit beim Aufruf einer Methode über einen Instanzzeiger nicht die im Quelltext vorliegende Typdefinition entscheidend ist, sondern der tatsächlich zur Laufzeit vorliegende Typ der Instanz. Virtuelle oder in diesem Abschnitt vielleicht besser auch als überschreibbar bezeichnete Methoden einer Klasse können deshalb durch den Compiler nicht mit einem direkten Funktionsaufruf übersetzt werden, da sonst die Möglichkeit des Überschreibens durch eine abgeleitete Klasse von vorneherein ausgeschlossen wird. Eine Flexibilität an dieser Stelle kann nur damit erreicht werden, dass der Übersetzer an der Stelle eines direkten Funktionsaufrufs einen indirekten absetzt. Zu diesem Zweck legt man pro C++-Klasse genau eine Tabelle bestehend aus Funktionszeigern an, eben die *vtbl*: In diese Tabelle werden zur Laufzeit die Adressen derjenigen Methoden eingetragen, die in der Klasse als `virtual` deklariert sind. Zu beachten ist: Die *vtbl* ist pro Klasse nur einmal vorhanden, also unabhängig davon, wie viele Instanzen von der Klasse existieren, siehe Abbildung 2.3.

Bemerkung Nicht-virtuelle Methoden einer Klasse werden direkt (und damit nicht mit dem Umweg über die virtuelle Funktionszeigertabelle) aufgerufen, sie sind damit auch kein Bestandteil dieser Tabelle.

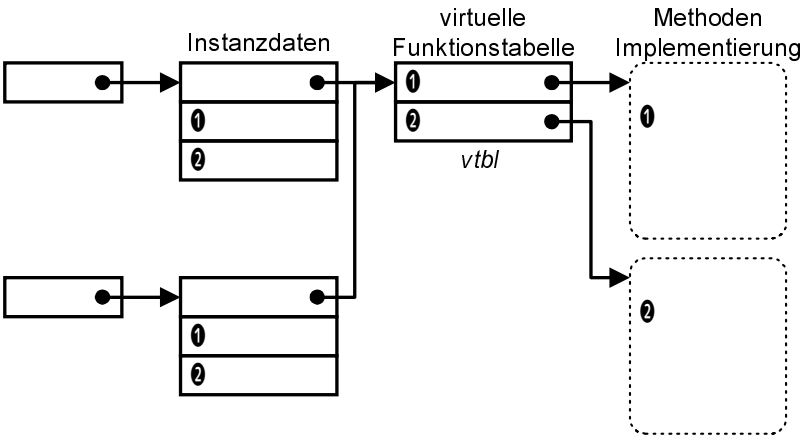

Abbildung 2.3: Die virtuelle Funktionszeigertabelle gibt es unabhängig von der Anzahl der Instanzen der Klasse nur einmal, hier aufgezeigt am Beispiel zweier Instanzdatenbereiche, die eine vtbl adressieren

2.3.4 Speicherlayout einer abgeleiteten C++-Klasse bei einfacher Vererbung

Die Erörterungen des letzten Abschnitts werden erst dann interessant, wenn wir die Vererbung von Klassen betrachten. Zunächst beschränken wir uns auf den Fall der einfachen Vererbung (sprich die abgeleitete Klasse besitzt nur eine Basisklasse).

Leitet sich eine Klasse durch einfache Vererbung von ihrer Vaterklasse ab, so werden alle Einträge der Vaterklasse zusammen mit den Einträgen der abgeleiteten Klasse in einer einzigen neuen *vtbl* zusammengefasst; insbesondere enthält jede Instanz der abgeleiteten weiterhin genau einen *vptr*.

Werfen wir einen Blick auf das Beispiel in Listing 2.6:

Listing

```
01: class B : public A {
02: public:
03:     int b1;                // neues Datenelement b1
04:     int b2;                // neues Datenelement b2
05:     virtual int vf_A1 ();  // virtuelle Funktion vf_A1,
06:                            // A::vf_A1 wird überschrieben !
07:     virtual int vf_B1 ();  // neue virtuelle Funktion vf_B1
08:     virtual int vf_B2 ();  // neue virtuelle Funktion vf_B2
09: };
```

Listing 2.6: Zweites Fallbeispiel einer Klasse B, die eine virtuelle Methode ihrer Basisklasse überschreibt

Zeichnung Abbildung 2.4 skizziert den Aufbau einer Instanz von B:

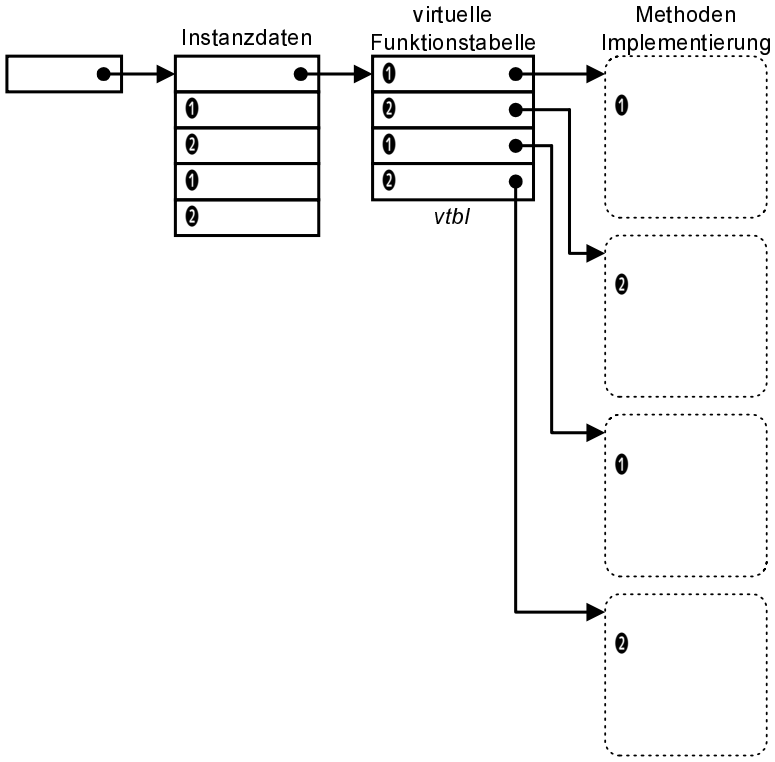

Abbildung 2.4: Die »virtuelle Funktionszeigertabelle« und der Instanzdatenbereich am Beispiel einer abgeleiteten Klasse

Folgende Eigenschaften sind zu beachten:

- Klasse `B` ist von Klasse `A` abgeleitet, besitzt wie `A` eine *vtbl* und enthält im Instanzdatenbereich einen *vptr*-Zeiger
- Für die Reihenfolge der Datenablage im Speicher gilt: Zuerst abgelegt sind die Daten der Basisklasse, danach die der abgeleiteten Klasse. Wir finden im Speicher also zuerst den *vptr* vor, gefolgt von `A::a1` und `A::a2` und dann `B::b1` und `B::b2`
- Für die *vtbl* gilt dieselbe Regel: Das Funktionszeigerfeld enthält zuerst alle Einträge für die virtuellen Methoden der Basisklasse, danach die der abgeleiteten Klasse

Betrachten wir jetzt die Methode `vf_A1`: `vf_A1` wird von `A` definiert und durch `B` überschrieben. Wie wirkt sich das auf die *vtbl* der Klasse `B` aus? Wir finden den Funktionszeiger `A::vf_A1` ersetzt durch `B::vf_A1` vor, sprich durch genau diese Tabellenänderung wird erreicht, dass zur Laufzeit die überschriebene Methode aufgerufen wird und nicht die der Basisklasse:

```
A * pA = new B;
pA -> vf_A1 ();     // B::vf_A1 wird aufgerufen!!!
```

Im Zusammenspiel mit Abbildung 2.4 wird nun verständlich, warum es zu einem Aufruf der Methode `B::vf_A1` kommt: `vf_A1` spezifiziert in einem Ausdruck wie `pA -> vf_A1` nur einen Offset in die *vtbl* (in diesem Beispiel Offset `0`) und keine Adresse im Codesegment; die Adresse der Methode selbst findet man unter dem entsprechenden Offset in der *vtbl*, diese wird dann mit einem indirekten Funktionsaufruf unter Bereitstellung der Aktualparameter auf dem Laufzeitstapel angesprungen.

2.3.5 Speicherlayout einer abgeleiteten C++-Klasse bei mehrfacher Vererbung

Wir müssen unsere bisherigen Betrachtungen noch mit einem Blick auf die mehrfache Vererbung fortsetzen, da diese in der COM-Entwicklung sehr häufig zum Einsatz kommt.

Betrachten wir das Ganze an einem Beispiel:

Listing

```
01: class A
02: {
03:     int a1;  // Datenelement a1
04:     int a2;  // Datenelement a2
05:
06:     virtual int vf_A1 ();  // virtuelle Funktion vf_A1
07:     virtual int vf_A2 ();  // virtuelle Funktion vf_A2
08: };
09:
10: class B
11: {
12:     int b1;  // Datenelement b1
13:     int b2;  // Datenelement b2
14:
15:     virtual int vf_B1 ();  // virtuelle Funktion vf_B1
16:     virtual int vf_B2 ();  // virtuelle Funktion vf_B2
17: };
18:
19: class C : public A, public B
20: {
```

```
21:     int c1;  // neues Datenelement c1
22:     int c2;  // neues Datenelement c2
23:
24:     virtual int vf_A1 ();  // virtuelle Funktion vf_A1,
25:                            // A::vf_A1 wird überschrieben!
26:
27:     virtual int vf_B1 ();  // virtuelle Funktion vf_B1,
28:                            // B::vf_B1 wird überschrieben!
29:
30:     virtual int vf_C1 ();  // neue virtuelle Funktion vf_C1
31: };
```

Listing 2.7: Fallbeispiel einer Klasse A, die mehrere Vaterklassen besitzt

Zeichnung Abbildung 2.5 skizziert den Aufbau einer Instanz von C:

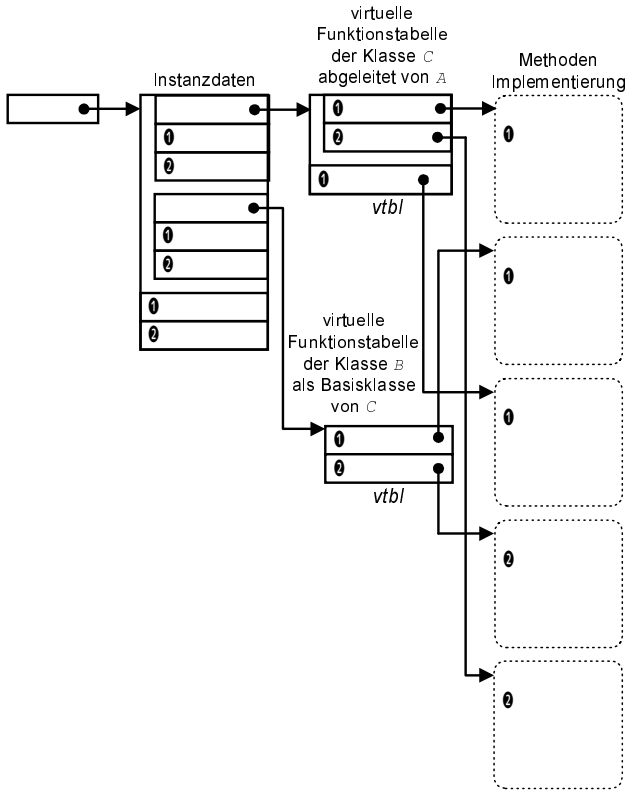

Abbildung 2.5: Ein Instanzdatenbereich mit zwei virtuellen Funktionszeigertabellen am Beispiel einer abgeleiteten Klasse mit mehrfacher Vererbung

Eine Klasse, die durch Mehrfachvererbung gebildet wird, besitzt einen *vptr* für jede ihrer Basisklassen, die virtuelle Funktionen besitzt. In Listing 2.7 erkennen wir ein Beispiel einer Klasse `C` mit den beiden Basisklassen `A` und `B`. Jede Basisklasse besitzt zwei Elemente und zwei virtuelle Funktionen. Die mit Hilfe der Mehrfachvererbung gebildete Klasse `C` besitzt ebenfalls zwei Elemente sowie drei virtuelle Funktionen, im Detail sind dies die Elemente `c1` und `c2` sowie die virtuellen Funktionen `C::vf_A1`, die `A::vf_A1` überschreibt, `C::vf_B1`, die `B::vf_B1` überschreibt, sowie eine dritte, ausschließlich in der Klasse `C` deklarierte Funktion `C::vf_C1`.

Abbildung 2.5 illustriert das Speicherlayout einer Instanz der Klasse `C`. Im Gegensatz zu Instanzen einer Klasse mit einfacher Vererbung enthält der Instanzdatenbereich einer mehrfach abgeleiteten Klasse pro Basisklasse (mit virtuellen Funktionen) einen *vptr*, der die entsprechende *vtbl* adressiert. Die *vtbls* der Basisklassen sind in ihrer *Struktur* identisch mit den *vtbls*, die diese Basisklasse als eigenständig betrachtete Klasse besitzen würde, abgesehen davon, dass Funktionszeiger von Funktionen, die in der abgeleiteten Klasse überschrieben sind, die entsprechenden Einträge in der *vtbl* ersetzen.

Keine Regel ohne Ausnahme: Die *vtbl* der Basisklasse, die in der Klassendeklaration von `C` in der Liste aller Basisklassen als erste aufgeführt ist (in unserem Beispiel Klasse `A`), wird um die Funktionszeiger der abgeleiteten Klasse ergänzt. Ihre Struktur entspricht damit ganz einfach formuliert einer *vtbl*, die den beiden Klassen entsprechen würde, wenn sie in einer einfachen Vererbungshierarchie in Beziehung stehen würden.

Damit sind wir am Ende unseres Ausblicks auf einfache und mehrfache Vererbung im Zusammenspiel mit virtuellen Funktionen angelangt. Mir ist bewusst, dass die Lektüre dieses Kapitels bei Ihnen vermutlich nicht gerade zu Begeisterungsstürmen geführt haben wird. Wenn wir uns in den nächsten Kapiteln intensiver mit COM-Schnittstellen beschäftigen werden und Sie diese im Umfeld von C++ auch bis in das letzte Detail verstehen wollen, dann stellen die Ausführungen dieses Kapitels jedoch mit Sicherheit eine große Hilfe dar.

2.4 Schnittstellenbasierte Programmierung in C++ und Java

Der Begriff der Schnittstelle besitzt in der modernen Software-Technologie eine zentrale Rolle. Wir wollen nun von allgemeinen Formulieren wie etwa »Die Niederschrift eines anerkannten Vertrags, über den ein Objekt und seine Clients miteinander kommunizieren« den Weg zu einer konkreteren Definition einschlagen:

Definition Unter einer *Schnittstelle* verstehen wir eine definierte Menge von Funktionen, die unter einem gemeinsamen Namen zusammengefasst sind.

Die Programmiersprache C++ besitzt das Sprachelement »Schnittstelle« nicht in expliziter Form wie beispielsweise die Sprache Java, aus diesem Grund haben die COM-Architekten darüber nachgedacht, wie man ein vergleichbares Konzept mit den vorhandenen C++-Sprachelementen modellieren könnte.

In C++ bestehen Klassen neben für jedermann verfügbaren Methoden noch aus weiteren Bestandteilen:

- private und geschützte Methoden (Schlüsselwort `private` und `protected`)
- spezielle Methoden für das Konstruieren und Freigeben einer Instanz (Konstruktoren und Destruktoren)
- instanzspezifische Daten pro Objekt

Zwar bietet C++ syntaktische Möglichkeiten, Teile einer Klasse – zum Beispiel eine Reihe von Methoden – als öffentlich zu deklarieren (Schlüsselwort `public`), nur lösen diese nicht das Problem, dass man die Implementierungsdetails der Klasse jederzeit ändern kann, ohne die Clients neu zu übersetzen, sofern man die öffentlich deklarierten Methoden konstant hält. Jede Änderung einer Klasse setzt bei C++ voraus, dass alle Softwareteile, die diese Klasse benutzen, neu zu übersetzen sind, da das binäre Layout einer Instanz im Speicher sich ändern kann. Glücklicherweise besitzt C++ einen Mechanismus, der dieses Problem umschifft: die *abstrakte* Basisklasse.

Definition Eine *abstrakte* Basisklasse ist eine Klasse, die eine oder mehrere Methoden besitzt, die virtuell definiert sind und keine Implementierung haben.

Und gleich noch eine Definition:

Definition Unter einer *reinen* abstrakten Basisklasse verstehen wir eine Klasse, die abstrakt ist und keine Datenelemente besitzt.

Betrachten wir eine Klasse, die für konkrete geometrische Figuren wie ein Rechteck oder einen Kreis zwei Methoden zur Berechnung der Fläche und des Umfangs festlegt:

Listing
```
01: struct GeometrischeFigur
02: {
03:     virtual double berechneFlaeche () = 0;
04:     virtual double berechneUmfang () = 0;
05: };
```

Listing 2.8: Ein Beispiel einer reinen abstrakten Basisklasse in C++

Auf den ersten Blick sehen abstrakte Basisklassen etwas merkwürdig aus: Sie sind C++-Klassen, lassen sich jedoch nicht instantiieren:

```
GeometrischeFigur * gf;
gf = new GeometrischeFigur;
```

Diese beiden Quellzeilen führen zu folgenden Fehlermeldungen des Übersetzers:

```
'GeometrischeFigur' :
  cannot instantiate abstract class due to following members:
'double GeometrischeFigur::berechneFlaeche(void)' :
  pure virtual function was not defined
'double GeometrischeFigur::berechneUmfang(void)' :
  pure virtual function was not defined
```

Die einzige Möglichkeit, abstrakte Klasse zu verwenden, bietet die Vererbung. Betrachten wir beispielsweise zwei reale geometrische Figuren wie ein Rechteck und einen Kreis, dann könnte eine Implementierung wie nachfolgend gezeigt aussehen:

Listing

```
01: class Rechteck : public GeometrischeFigur
02: {
03: private:
04:     int x, y;
05:     int breite, hoehe;
06:
07: public:
08:     virtual double berechneFlaeche ()
09:     {
10:         return breite * hoehe;
11:     }
12:
13:     virtual double berechneUmfang ()
14:     {
15:         return 2 * (breite + hoehe);
16:     }
17: };
18:
19: class Kreis : public GeometrischeFigur
20: {
21: private:
22:     int x, y;
23:     double radius;
24:
```

```
25: public:
26:     virtual double berechneFlaeche ()
27:     {
28:         return 3.14159265 * radius * radius;
29:     }
30:
31:     virtual double berechneUmfang ()
32:     {
33:         return 2 * 3.14159265 * radius;
34:     }
35: };
```

Listing 2.9: Die abstrakte Klasse `GeometrischeFigur` als Basisklasse der beiden konkreten Klassen `Rechteck` und `Kreis`

Sie werden sich jetzt sicherlich fragen: »Wozu habe ich mir denn die Arbeit gemacht und eine rein abstrakte Basisklasse definiert, die niemand benutzen kann?«

Der Punkt ist, dass rein abstrakte Basisklassen in C++ die Möglichkeit bieten

- eine Schnittstelle an einer Stelle zu definieren
- ihre Implementierung an einem anderen, besser geeigneten Ort vorzunehmen

Insbesondere lassen sich damit zu einem beliebigen Zeitpunkt Änderungen an der Implementierung vornehmen, ohne dass (bei konstant gehaltener Methodenschnittstelle) die Definition dieser Schnittstelle noch einmal zu übersetzen ist.

Bemerkung In Java ist das Schnittstellenkonzept expliziter Bestandteil der Sprache. Eine Reihe von zentralen Mechanismen wie das Ereignis-Handling (Schnittstellenfamilie `XXXListener`), die Hantierung von Threads (Schnittstelle `Runnable`), die Kennzeichnung von Methoden, die von einer anderen Virtual Machine aus aufgerufen werden können (Schnittstelle `Remote`), usw. basieren auf dem Java-Sprachelement `interface`, siehe Listing 2.10:

Listing

```
01: abstract interface GeometrischeFigur
02: {
03:     abstract public double berechneFlaeche ();
04:     abstract public double berechneUmfang ();
05: }
```

Listing 2.10: Ein Beispiel für eine Schnittstellendefinition in der Programmiersprache Java, siehe dazu auch das Beispiel in Listing 2.8

Vor diesem Hintergrund wenden wir uns nun der Definition einer Schnittstelle unter COM zu:

2.5 Schnittstellenbegriff in COM

Da nahezu alle gängigen, am Markt verfügbaren C++-Compiler Binärcode erzeugen, der virtuelle Methoden über die Indirektionsstufe einer *vtbl* aufruft, entschloss man sich bei Microsoft, die virtuelle Funktionszeigertabelle als Ausgangspunkt für die Definition von COM-Schnittstellen zu nehmen. Wir benötigen für eine vollständige Betrachtung des COM-Schnittstellenbegriffs zwei Teiletappen:

Definition (Teil 1)

Unter einer COM-Schnittstelle (auch als COM-Interface bezeichnet) versteht man die Zusammenfassung einer oder mehrerer (semantisch zusammengehöriger) Methoden zu einer logischen Gruppe, die in der Programmiersprache C++ mit Hilfe einer reinen abstrakten Basisklasse definiert werden.

Ein Zeiger auf eine COM-Schnittstelle (COM-Schnittstellenzeiger) ist ein Zeiger auf eine *vtbl*, die der korrespondierenden abstrakten Basisklasse in C++ zugeordnet ist.

Da COM-Server und -Clients nicht alle notwendigerweise mit der Programmiersprache C++ erstellt werden müssen, folgt als Konsequenz, dass jede COM-unterstützende Programmiersprache gezwungen ist, dieses durch eine *vtbl* geprägte Binärformat zu beherrschen:

Definition

Unter dem Begriff »binäres Standardformat« im Zusammenhang mit einer COM-Schnittstelle versteht man die Eigenschaft, dass ein COM-Client die Methoden eines COM-Objekts aufrufen kann unabhängig von der Programmiersprache, mit der eine konkrete Realisierung der Methoden dieser Klasse vorgenommen wird.

Egal, welche Programmiersprache verwendet wird, der Zugriff auf eine COM-Schnittstelle wie auch das Speicherlayout einer COM-Schnittstelle sind durch ein binäres Standardformat fest vorgegeben, siehe Abbildung 2.6.

Konsequenz: Wenn ein (in einer beliebigen Programmiersprache erstellter) Client die Parameter (Datentyp) und den Rückgabewert einer bestimmten Methode kennt und den Funktionszeiger aus der *vtbl* verwendet, kann er auf das COM-Objekt zugreifen, ohne (wie bei C++) die konkrete C++-Klasse zu kennen, mit deren Hilfe das Objekt programmiertechnisch erzeugt wurde.

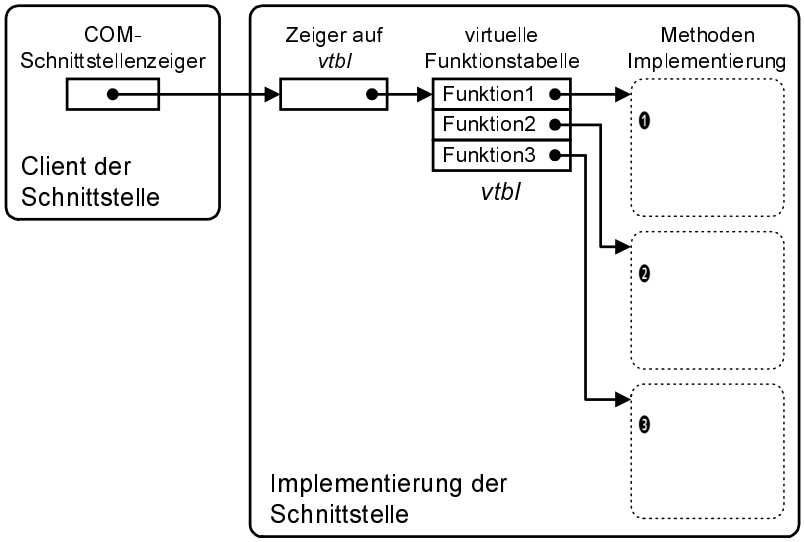

Abbildung 2.6: Der vtbl-Zeiger als programmiersprachenunabhängiges Merkmal dient als Definition eines Methodenaufrufs in COM

Um eine Methode einer COM-Schnittstelle aufrufen zu können, ist eine Zeigervariable zweimal zu dereferenzieren: Mit der ersten Dereferenzierung erhält man eine Zeigervariable, die auf ein Feld von Funktionszeigern zeigt (wir sprechen hier exakt von dem *vptr*-Zeiger, dem wir im letzten Abschnitt über virtuelle Methoden bereits begegnet sind). Die zweite Dereferenzierung führt uns schließlich zu der gesuchten Methode, die den Code zur Verfügung stellt und mit den erforderlichen Parametern aufzurufen ist. Der Vorgang des Aufrufs einer Methode einer COM-Schnittstelle ist also identisch mit dem Aufruf einer virtuellen Methode in C++.

An dieser Stelle sollte man sicherlich erwähnen, dass die Erstellung eines COM-Programms aus C++ heraus geboren wurde. Jede andere Programmiersprache, die COM unterstützen möchte, muss damit das *vtbl*-Layout von C++ unterstützen.

Fangen wir mit einem Beispiel an und betrachten eine COM-Schnittstelle, die nur aus einer einzigen Methode `SayHello` besteht, die als Parameter eine Zeichenkette entgegennimmt und deren korrekte Ausgabe (auf welchem Medium auch immer, hierzu wollen wir im Augenblick nicht näher eingehen) mit einem `HRESULT`-Wert quittiert. Mit C++ als Implementierungssprache sieht die Definition wie folgt aus:

Listing

```
01: struct ISayHello
02: {
03:     virtual HRESULT __stdcall SayHello (BSTR szMessage) = 0;
04: };
```

Listing 2.11: Erste Variante einer COM-Schnittstellendeklaration in C++

Die obige Darstellung einer COM-Schnittstelle ist zwar syntaktisch korrekt, vom visuellen Standpunkt aus betrachtet aber nicht sehr ansehnlich: Vor allem die Microsoft-spezifischen Schlüsselworte wie `__stdcall` sollten in einer Quelldatei nicht explizit in Erscheinung treten, zu diesem Zweck gibt es eine Reihe von `#define`-Anweisungen in der `#include`-Datei *basetyps.h*, die zu einer eleganteren Schreibweise führen:

Listing

```
01: DECLARE_INTERFACE (ISayHello)
02: {
03:     // *** ISayHello methods ***
04:     STDMETHOD(SayHello) (BSTR szMessage) PURE;
05: };
```

Listing 2.12: Zweite, besser lesbare Variante einer COM-Schnittstellendeklaration in C++

Dabei haben wir die folgenden Definitionen verwendet:

```
#define interface                  struct
#define DECLARE_INTERFACE(iface)   interface iface
#define STDMETHOD(method)          virtual HRESULT \
                                   STDMETHODCALLTYPE method
#define STDMETHOD_(type,method)    virtual type \
                                   STDMETHODCALLTYPE method
#define STDMETHODCALLTYPE          __stdcall
#define PURE                       = 0
```

Die Verwendung von `DECLARE_INTERFACE` wird von manchen Büchern der COM-Literatur vermieden, daher sieht man des Öfteren auch die folgende Variante:

Listing

```
01: interface ISayHello
02: {
03:     // *** ISayHello methods ***
04:     STDMETHOD(SayHello) (BSTR szMessage) PURE;
05: };
```

Listing 2.13: Alternative Variante zu Listing 2.12, um eine COM-Schnittstelle zu deklarieren

Einige Bemerkungen hierzu:

- COM-Schnittstellen werden bewusst mit dem C++-Schlüsselwort `struct` und nicht mit `class` definiert. In `struct`-Klassen sind von Haus aus alle Methoden als `public` definiert, man kann sich damit also Unmengen von `public`-Schlüsselwörtern ersparen, um die Schnittstellenmethoden öffentlich zugänglich zu machen.
- Da COM-Schnittstellen in verschiedenen Programmiersprachen realisierbar sind, muss für ihre Methoden festgelegt werden, wie ihr Aufruf für eine COM-konforme Zielplattform handzuhaben ist, also beispielsweise »in welcher Reihenfolge werden die Parameter auf den Stapel gebracht« (*argument-passing order*) oder »wer räumt die Parameter vom Stack wieder ab – die gerufene Methode oder der Rufer« (*stack-maintenance responsibility*). COM-konforme Methodenaufrufe sind bzgl. ihrer technischen Hantierung identisch mit der des Win32-APIs, und dieses wiederum wird in Visual C++ mit dem Schlüsselwort `__stdcall` eingestellt. Damit wird erreicht, dass die Parameter mit Blick auf den Quelltext von »rechts nach links« auf den Stapel geschoben werden und die aufgerufene Methode für das Aufräumen des Stapels verantwortlich ist.
- Rückgabewerte von COM-Schnittstellenmethoden *müssen* immer vom Typ `HRESULT` sein.
- Per Konvention beginnen Schnittstellennamen immer mit einem großen `I`.

Wir vervollständigen nun die Definition des COM-Schnittstellenbegriffs:

Definition (Teil 2)

Jede COM-Schnittstelle erbt die drei Methoden `QueryInterface`, `AddRef` und `Release` der vordefinierten COM-Standardschnittstelle `IUnknown`.

`IUnknown` ist die zentrale Schnittstelle, die im Mittelpunkt des gesamten Kernkonzeptes von COM steht. Aufgrund ihrer Bedeutung habe ich für ihre Erläuterung das folgende Kapitel gewählt, wir schließen dieses Kapitel noch kurz mit einigen allgemeinen Betrachtungen ab: Zunächst müssen wir die beiden Varianten der `ISayHello`-Schnittstelle um eine dritte, nun endgültige Fassung ergänzen:

Listing

```
01: interface ISayHello : public IUnknown
02: {
03:     // *** ISayHello methods ***
04:     STDMETHOD(SayHello) (BSTR szMessage) PURE;
05: };
```

Listing 2.14: Die `ISayHello`-COM-Schnittstelle in ihrem endgültigen Aussehen: Mit Ausnahme von `IUnknown` selbst muss jede COM-Schnittstelle die drei Methoden der `IUnknown`-Schnittstelle erben

Bei den Schnittstellen finden wir wie bei den COM-Klassen wieder das Problem der eindeutigen Namensgebung vor. Wie nicht anders zu erwarten, greifen wir auch an dieser Stelle auf den bereits bekannten Datentyp `GUID` zurück:

```
typedef GUID  IID;         // interface ID
typedef IID   * REFIID;    // IID reference
```

Sprich, das Verfahren der Benennung von Klassen mit Nummern findet auch bei COM-Schnittstellen Anwendung.

Beispiel

Für die Schnittstelle `ISayHello` haben wir die IID `{10000001-0000-0000-0000-000000000000}` gewählt.

Ebenfalls wie bei den CLSIDs müssen wir uns auch bei den IIDs auf die Suche nach einem lesbaren Stellvertreter begeben. Einen Namen zu finden ist ja nicht so schwer, es bietet sich beispielsweise in einer C++-Umgebung der Name der Struktur an, die die Schnittstelle definiert. Berücksichtigt man noch, dass per Konvention dieser Name stets mit einem großen `I` anfangen sollte, wie beispielsweise `ISayHello`, dann hat man sicherlich einen gute Festlegung getroffen. Interessant wird das Ganze aber erst, wenn diese Namen auch für jedermann sichtbar sind. Hierzu bietet die Windows-Registrierung einen Schlüssel `HKEY_CLASSES_ROOT\Interface` an, unter dem alle auf dem Rechner bekannten Schnittstellennamen (sortiert nach den Ziffern der IID) aufgelistet sind, siehe Abbildung 2.7:

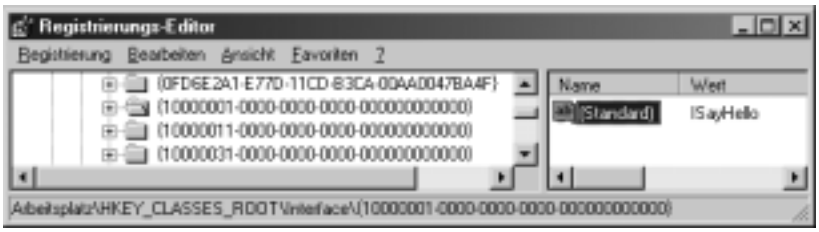

Abbildung 2.7: Unser Schnittstellenname `ISayHello` für jedermann sichtbar in der Windows- Registrierung

Nach diesem Ausflug in die Niederungen des Schnittstellenkonzepts im Umfeld von C++, COM und Java sind wir hinreichend gerüstet, um das nächste Etappenziel der COM-Klassen und COM-Objekte anzugehen.