(19) **Bundesrepublik Deutschland**
Deutsches Patent- und Markenamt

(10) **DE 695 33 530 T2** 2005.09.29

(12) # Übersetzung der europäischen Patentschrift

(97) **EP 0 664 510 B1**
(21) Deutsches Aktenzeichen: **695 33 530.8**
(96) Europäisches Aktenzeichen: **95 100 777.2**
(96) Europäischer Anmeldetag: **20.01.1995**
(97) Erstveröffentlichung durch das EPA: **26.07.1995**
(97) Veröffentlichungstag der Patenterteilung beim EPA: **22.09.2004**
(47) Veröffentlichungstag im Patentblatt: **29.09.2005**

(51) Int Cl.[7]: **G06F 9/44**

(30) Unionspriorität:
185465 **24.01.1994** **US**

(73) Patentinhaber:
Microsoft Corp., Redmond, Wash., US

(74) Vertreter:
Grünecker, Kinkeldey, Stockmair & Schwanhäusser, 80538 München

(84) Benannte Vertragsstaaten:
DE, FR, GB

(72) Erfinder:
Leach, Paul, Seattle, Washington 98102, US; Williams, Antony S., Redmond, Washington 98053, US; Jung, Edward, Seattle, Washington 98133, US; Hodges, Douglas C., Redmond, Washinton 98053, US; Koppolu, Srinivasa R., Redmond, Washington 98053, US; Mackichan, Barry B., Bainbridge Island, Washington 98110, US; Wittenberg, Craig, Mercer Island, Washington 98040, US

(54) Bezeichnung: **Verfahren und System zur dynamischen Aggregation von Objekten**

Beschreibung

[0001] Die vorliegende Erfindung betrifft allgemein ein Computerverfahren und -system zum Implementieren von Schnittstellen zu Objekten und insbesondere ein Verfahren und ein System zum Aggregieren von Objekten.

[0002] Mit der zunehmenden Komplexität von Softwaresystemen erhöhen sich die Kosten für die Softwareentwicklung. Um die Kosten der Softwareentwicklung zu minimieren, verwenden Entwickler häufig einen gemeinsamen Code. Dabei stehen bisher drei Typen der gemeinsamen Verwendung von Code zur Verfügung: (1) die gemeinsame Verwendung von Quellcode, (2) die gemeinsame Verwendung von kompiliertem Code und (3) die gemeinsame Verwendung mittels einer Vererbung.

[0003] Mit der zunehmenden Komplexität von Softwaresystemen sind außerdem die Softwarebenutzer mit einer zunehmenden Komplexität bei der Wartung und Erweiterung ihrer Computersysteme konfrontiert. Jedes Mal wenn ein Softwarehersteller neue oder verbesserte Funktionen entwickelt, muss ein an den Verbesserungen interessierter Benutzer diese Modifikationen in sein Computersystem integrieren. Die bestehenden Systeme erfordern, dass der Benutzer die Software aktualisiert, sodass der Benutzer wenigstens einen Teil des Systems neu installieren muss. Weiterhin erfordern die bestehenden Systeme, dass ein Softwareentwickler im voraus eine Möglichkeit für zukünftige Verbesserungen vorsieht und entsprechende Hooks in dem Originalsystem einfügt, damit zukünftige Verbesserungen geladen werden können.

[0004] Wenn ein Benutzer die Fähigkeiten einer Software in seinem Besitz verbessern möchte, indem er durch einen anderen Softwarehersteller entwickelte Fähigkeiten hinzufügt, ist der Benutzer auf die zuvor vom ursprünglichen Hersteller vorgesehenen Möglichkeiten beschränkt. Die bestehenden Systeme erfordern allgemein, dass die zwei entwickelten Softwareelemente zueinander passen und dass wenigstens eines der Softwareelemente die Fähigkeiten des anderen Softwareelements kennt. Bei den bestehenden Systemen müssen also spätere Modifikationen am Code sowie die gemeinsame Verwendung von Code bereits bei der Entwicklung der Software berücksichtigt werden.

[0005] Die gemeinsame Verwendung von Quellcode und von kompiliertem Code wird seit vielen Jahren genutzt. Unter der gemeinsamen Verwendung von Quellcode ist die Verwendung des gleichen Quellcodes durch verschiedene Computerprogramme oder durch verschiedene Versionen desselben Computerprogramms zu verstehen. Zum Beispiel umfasst ein Tabellenkalkulationsprogramm gewöhnlich einen Quellcode zum Steuern der Anzeige einer Tabelle. Wenn ein Textverarbeitungsprogramm das Einbetten einer Tabelle in einem Dokument gestattet, kann das Textverarbeitungsprogramm denselben (oder einen etwas modifizierten) Quellcode verwenden, um das eingebettete Tabellenobjekt anzuzeigen. Die gemeinsame Verwendung von Quellcode wird gewöhnlich durch denselben Entwickler genutzt, der seinen Quellcode nicht für andere Entwicklern freigibt. Aus Konkurrenzgründen geben Entwickler gewöhnlich ihren Quellcode nicht für andere Entwickler frei. Und auch wenn ein Entwickler seinen Quellcode freigibt, modifiziert der Empfänger des Quellcodes den Quellcode gewöhnlich, sodass zwei Versionen des Quellcodes aufrechterhalten werden.

[0006] Unter der gemeinsamen Verwendung von kompiliertem Code ist die Verwendung desselben kompilierten Codes durch verschiedene Computerprogramme zu verstehen. Der kompilierte Code wird gewöhnlich in einer statischen oder dynamischen Verknüpfungsbibliothek (SLL, DLL) gespeichert. Der in einer statischen Verknüpfungsbibliothek gespeicherte kompilierte Code wird gemeinsam verwendet, wenn ein Computerprogramm vor der Ausführung damit verknüpft wird. Der in einer dynamischen Verknüpfungsbibliothek gespeicherte kompilierte Code wird gemeinsam verwendet, wenn ein Computerprogramm während der Ausführung damit verknüpft wird. Der Entwickler eines Rechtschreibprüfprogramms kann zum Beispiel den kompilierten Code freigeben, indem er das Programm kompiliert und den kompilierten Code in einer statischen Verknüpfungsbibliothek speichert. Die statische Verknüpfungsbibliothek kann dann an die Entwickler von Textverarbeitungsprogrammen verteilt werden, die den kompilierten Rechtschreibprüfcode in ihr Textverarbeitungsprogramm verknüpfen können. Der Entwickler des Rechtschreibprüfprogramms muss den kompilierten Code gewöhnlich modifizieren, um speziellen Anforderungen von bestimmten Entwicklern zu erfüllen. Diese Modifikationen erhöhen gewöhnlich die Komplexität (und die Größe) des kompilierten Codes und können in Konflikt zu Anforderungen von anderen Empfängern geraten. Alternativ hierzu kann der Entwickler mehrere Versionen der statischen Verknüpfungsbibliothek bereitstellen. Die Pflege von mehreren Versionen kann jedoch kostenaufwändig sein.

[0007] Objekt-orientierte Programmiertechniken verwenden ein als Vererbung bezeichnetes Konzept, um die gemeinsame Verwendung von Code zu gestatten. Es wird im Folgenden eine Übersicht über diese wohlbe-

kannten Programmiertechniken gegeben, weil die vorliegende Erfindung weiter unten mit Bezug auf eine Objekt-orientierte Programmierung beschrieben wird. Die beiden Hauptmerkmale von Objekt-orientierten Programmiersprachen sind die Unterstützung einer Datenverkapselung und einer Datentyp-Vererbung. Die Datenverkapselung ist eine Verbindung von Funktionen mit Daten. Unter einer Vererbung ist die Fähigkeit zu verstehen, einen Datentyp anhand von anderen Datentypen zu definieren.

[0008] In der C++-Sprache werden Objekt-orientierte Techniken mittels der Verwendung von Klassen unterstützt. Eine Klasse ist ein benutzerdefinierter Typ. Eine Klassendefinition beschreibt die Datenglieder und die Funktionsglieder der Klasse. Zum Beispiel definiert die folgende Definition Datenglieder und ein Funktionsglied einer als CIRCLE (Kreis) bezeichneten Klasse.

```
class CIRCLE
{ public:
        int x, y;
        int radius;
        void draw();
};
```

[0009] Die Variablen x und y geben die zentrale Position eines Kreises an, und ein variabler Radius spezifiziert den Radius des Kreises. Diese Variablen sind die Datenglieder der Klasse CIRCLE. Die Funktion draw (Zeichnen) ist eine benutzerdefinierte Funktion, die den Kreis mit dem spezifizierten Radius an der spezifizierten Position zeichnet. Die Funktion draw ist ein Funktionsglied der Klasse CIRCLE. Die Datenglieder und die Funktionsglieder einer Klasse sind dadurch miteinander verbunden, dass die Funktion auf einer Instanz der Klasse operiert. Eine Instanz einer Klasse wird auch als Objekt der Klasse bezeichnet.

[0010] In der Syntax von C++ definiert die folgende Anweisung die Objekte a und b als zur Typklasse CIRCLE gehörig.

```
CIRCLE a, b;
```

[0011] Diese Definition veranlasst die Zuordnung von Speicher für die Objekte a und b. Die folgenden Anweisungen weisen Daten zu den Datengliedern der Objekte a und b zu.

```
a.x = 2;
a.y = 2;
a.radius = 1;
b.x = 4;
b.y = 5;
b.radius = 2;
```

[0012] Die folgenden Anweisungen werden verwendet, um die durch die Objekte a und b definierten Kreise zu zeichnen.

```
a.draw();
b.draw();
```

[0013] Eine abgeleitete Klasse ist eine Klasse, die die Eigenschaften – Datenglieder und Funktionsglieder – ihrer Basisklassen erben. Zum Beispiel erbt die folgende abgeleitete Klasse CIRCLE_FILL (Kreis füllen) die Eigenschaften der Basisklasse CIRCLE.

```
class CIRCLE_FILL : CIRCLE
{ public;
        int pattern;
        void fill();
};
```

[0014] Diese Definition spezifiziert, dass die Klasse CIRCLE_FILL alle die Daten- und Funktionsglieder in der Klasse CIRCLE zusätzlich zu den Daten- und Funktionsgliedern erhält, die in der Definition der Klasse

CIRCLE_FILL eingeführt werden – d. h. das Datenglied pattern (Muster) und das Funktionsglied fill (Füllen). In diesem Beispiel weist die Klasse CIRCLE_FILL die Datenglieder x, y, radius und pattern sowie die Funktionsglieder draw und fill auf. Die Klasse CIRCLE_FILL „erbt" sozusagen die Eigenschaften der Klasse CIRCLE. Eine Klasse, die die Eigenschaften einer anderen Klasse erbt, ist eine abgeleitete Klasse (z. B. CIRCLE_FILL). Eine Klasse, die keine Eigenschaften von einer anderen Klasse erbt, ist eine primäre (Root)-Klasse (z. B. CIRCLE). Eine Klasse, deren Eigenschaften durch eine andere Klasse geerbt werden, ist eine Basisklasse (z. B. ist CIRCLE eine Basisklasse für CIRCLE_FILL). Eine abgeleitete Klasse kann die Eigenschaften von mehreren Klassen erben, d. h. eine abgeleitete Klasse kann mehrere Basisklassen aufweisen. Dies wird als mehrfache Vererbung bezeichnet.

[0015] Eine abgeleitete Klasse kann spezifizieren, dass eine Basisklasse virtuell geerbt werden soll. Die virtuelle Vererbung einer Basisklasse bedeutet, dass nur das Erbe der virtuellen Basisklasse in der abgeleiteten Klasse existiert. Zum Beispiel ist Folgendes ein Beispiel für eine abgeleitete Klasse mit zwei nicht-virtuellen Basisklassen.
class CIRCLE_1: CIRCLE {...};
class CIRCLE_2: CIRCLE {...};
class PATTERN: CIRCLE_1, CIRCLE_2 {...};

[0016] In dieser Definition erbt die Klasse PATTERN die Klasse CIRCLE zwei Mal nicht-virtuell durch die Klassen CIRCLE_1 und CIRCLE_2. Es gibt also zwei Instanzen der Klasse CIRCLE in der Klasse PATTERN.

[0017] Folgendes ist ein Beispiel für eine abgeleitete Klasse mit zwei virtuellen Basisklassen.
class CIRCLE_1: virtual CIRCLE {...};
class CIRCLE_2: virtual CIRCLE {...};
class PATTERN: CIRCLE_1, CIRCLE_2 {...};

[0018] Die abgeleitete Klasse PATTERN erbt die Klase CIRCLE zwei Mal virtuell über die Klassen CIRCLE_1 und CIRCLE_2. Weil die Klasse CIRCLE zwei Mal virtuell geerbt wird, gibt es nur ein Objekt der Klasse CIRCLE in der abgeleiteten Klasse PATTERN. Dem Fachmann sollte deutlich sein, dass die virtuelle Vererbung sehr nützlich sein kann, wenn die Klassenableitung komplexer ist.

[0019] Eine Klasse kann auch spezifizieren, ob ihre Funktionsglieder virtuell sind. Die Definition, dass ein Funktionsglied virtuell ist, bedeutet, dass die Funktion durch eine Funktion desselben Namens und Typs in der abgeleiteten Klasse außer Kraft gesetzt werden kann. In dem folgenden Beispiel ist die Funktion draw in den Klassen CIRCLE und CIRCLE_FILL als virtuell definiert.

```
class CIRCLE
{ public:
        int x, y;
        int radius;
        virtual void draw();
};


class CIRCLE_FILL : CIRCLE
{ pulic:
        int pattern;
        virtual void draw();
};
```

[0020] Wenn eine virtuelle Funktion definiert wird, ohne eine Implementierung vorzusehen, wird dies als eine rein virtuelle Funktion bezeichnet. Eine rein virtuelle Funktion ist eine virtuelle Funktion, die mit dem reinen Spezifikationselement „= 0" definiert wird. Wenn eine Klasse eine rein virtuelle Funktion spezifiziert, muss eine abgeleitete Klasse eine Implementierung für dieses Funktionselement spezifizieren, bevor dieses Funktionsglied aufgerufen werden kann.

[0021] Um auf Objekte zuzugreifen, sieht die C++-Sprache einen Zeiger-Datentyp vor. Ein Zeiger speichert Werte, die Adressen von Objekten im Speicher sind. Über einen Zeiger kann auf ein Objekt Bezug genommen werden. Die folgende Anweisung definiert eine Variable c_ptr als Zeiger auf ein Objekt der Typenklasse CIRCLE und setzt die Variable c_ptr zum Speichern der Adresse des Objekts c.

```
CIRCLE *c_ptr;
c_ptr = &c;
```

[0022] Mit diesem Beispiel fortfahrend definiert die folgende Anweisung das Objekt als zu der Typklasse CIRCLE gehörig und das Objekt b als zu der Typklasse CIRCLE_FILL gehörig.

```
CIRCLE a;
CIRCLE_FILL b;
```

[0023] Die folgende Anweisung bezieht sich auf die Funktion draw, wie in der Klasse CIRCLE definiert.

```
a.draw();
```

[0024] Die folgende Anweisung dagegen bezieht sich auf die Funktion draw, die in der Klasse CIRCLE_FILL definiert ist.

```
b.draw();
```

[0025] Weiterhin definieren die folgenden Anweisungen das Objekt b als ein Objekt der Typenklasse CIRCLE und rufen die Funktion draw auf, die in der Klasse CIRCLE_FILL definiert ist.

```
CIRCLE *c_ptr;
c_ptr = &b;
c_ptr->draw(); // CIRCLE_FILL::draw()
```

[0026] Die aufgerufene virtuelle Funktion ist also die Funktion CIRCLE_FILL::draw.

[0027] Fig. 1 ist ein Blockdiagramm, das typische Datenstrukturen zeigt, die zur Wiedergabe eines Objekts verwendet werden. Ein Objekt setzt sich aus Instanzdaten (Datengliedern) und Gliedfunktionen zusammen, die das Verhalten des Objekts implementieren. Die zur Wiedergabe eines Objekts verwendeten Datenstrukturen umfassen eine Instanzdatenstruktur **101**, eine Virtuellfunktionstabelle **102** und die Funktionsglieder **103**, **104**, **105**. Die Instanzdatenstruktur **101** enthält einen Zeiger auf die Virtuellfunktionstabelle **102** und enthält Datenglieder. Die Virtuellfunktionstabelle **102** enthält einen Eintrag für jedes virtuelle Funktionsglied, das für das Objekt definiert ist. Jeder Eintrag enthält eine Bezugnahme auf den Code, der das entsprechende Funktionsglied implementiert. Das Layout dieses Probeobjekts entspricht dem Modell, das in der US-Patentanmeldung mit der Seriennummer 07/682,537 mit dem Titel „A Method for Implementing Virtual Functions and Virtual Bases in a Compiler for an Object Oriented Programming Language" definiert ist, die hier unter Bezugnahme eingeschlossen ist. Im Folgenden wird ein Objekt als eine Instanz einer Klasse gemäß der Definition der C++-Programmiersprache beschrieben. Dem Fachmann sollte deutlich sein, dass Objekte auch unter Verwendung von anderen Programmiersprachen definiert werden können.

[0028] Die Vererbung einer Klasse ist ein Typ von gemeinsamer Codeverwendung. Ein Entwickler einer Klasse kann die Implementierung der Klasse für andere Entwickler freigeben. Diese anderen Entwickler können dann Klassen erstellen, die von der freigegebenen Klasse abgeleitet sind. Auf diese Weise werden die Funktionsglieder der freigegebenen Klasse gemeinsam genutzt. Wenn jedoch eine Klasse geerbt wird und eine virtuelle Funktion außer Kraft gesetzt wird, dann kann das Testen der außer Kraft setzenden virtuellen Funktion komplex sein. Die außer Kraft setzende virtuelle Funktion kann den Zustand des Objekts auf eine Weise modifizieren, die nicht außer Kraft gesetzte Funktionen beeinflusst. Jede geerbte Funktion muss daher unabhängig in Verbindung mit dem Testen der außer Kraft setzenden virtuellen Funktion getestet werden. Um die Komplexitäten des Testens zu reduzieren, können die Entwickler einer Klassenimplementierung den Quellcode mit der Implementierung freigeben. Leider ist die Freigabe von Quellcode wie bereits weiter oben genannt mit häufig nicht erwünscht.

[0029] Ein Vorteil bei der Verwendung von Objekt-orientierten Techniken besteht darin, dass diese Techniken verwendet werden können, um die gemeinsame Verwendung von Objekten zu vereinfachen. Insbesondere vereinfachen Objekt-orientierte Techniken die Erstellung von zusammengesetzten Dokumenten. Ein zusammengesetztes Dokument ist ein Dokument, das durch verschiedene Computerprogramme erzeugte Objekte enthält. (Gewöhnlich werden nur die Datenglieder des Objekts und der Klassentyp in einem zusammengesetzten Dokument gespeichert.) Zum Beispiel ist ein Textverarbeitungsdokument, das ein durch ein Tabellenkalkulationsprogramm erzeugtes Tabellenobjekt enthält, ein zusammengesetztes Dokument. Ein Textverarbeitungs-

programm erlaubt es einem Benutzer, ein Tabellenobjekt (z. B. eine Zelle) in einem Textverarbeitungsdokument einzubetten. Um diese Einbettung zu gestatten, wird das Textverarbeitungsprogramm unter Verwendung der Klassendefinition des einzubettenden Objekts kompiliert, um auf die Funktionsglieder des eingebetteten Objekts zuzugreifen. Das Textverarbeitungsprogramm muss also unter Verwendung der Klassendefinition jeder Klasse von Objekten kompiliert werden, die in einem Textverarbeitungsdokument eingebettet werden können. Um ein Objekt einer neuen Klasse in ein Textverarbeitungsdokument einzubetten, muss das Textverarbeitungsprogramm mit der neuen Klassendefinition neu kompiliert werden. Es können also nur Objekte von durch den Entwickler des Textverarbeitungsprogramms ausgewählten Klassen eingebettet werden. Außerdem können neue Klassen nur mit einer neuen Version des Textverarbeitungsprogramms unterstützt werden.

[0030] Damit Objekte einer beliebigen Klasse in zusammengesetzte Dokumente eingebettet werden können, sind Schnittstellen definiert, über die auf ein Objekt zugegriffen werden kann, ohne dass das Textverarbeitungsprogramm während des Kompilierens auf die Klassendefinitionen zuzugreifen braucht. Eine abstrakte Klasse ist eine Klasse, in der wenigstens ein virtuelles Funktionsglied ohne Implementierung (ein reines virtuelles Funktionsglied) enthalten ist. Eine Schnittstelle ist eine abstrakte Klasse ohne Datenglieder, deren virtuelle Funktionen alle rein sind. Eine Schnittstelle sieht also ein Protokoll für die Kommunikation zwischen zwei Programmen vor. Schnittstellen werden gewöhnlich für die Ableitung verwendet: ein Programm implementiert Klassen, die Implementierungen für die Schnittstellen vorsehen können, von denen die Klassen abgeleitet werden. Danach werden Objekte als Instanzen dieser abgeleiteten Klassen erstellt.

[0031] Die folgende Klassendefinition ist eine beispielhafte Definition einer Schnittstelle. In diesem Beispiel erlaubt ein Textverarbeitungsprogramm der einfacheren Darstellung halber nicht die Einbettung von beliebigen Klassen von Objekten in seinen Dokumenten, sondern nur die Einbettung von Tabellenobjekten. Ein Tabellenobjekt, das diese Schnittstelle vorsieht, kann unabhängig von seiner Implementierung eingebettet werden. Weiterhin kann ein beliebiges Tabellenobjekt eingebettet werden, unabhängig davon, ob es vor oder nach der Kompilierung des Textverarbeitungsprogramms implementiert wird.

```
class ISpreadSheet
{       virtual void File() = 0;
        virtual void Edit() = 0;
        virtual void Formula() = 0;
        virtual void Format() = 0;
        virtual void GetCell (string RC, cell *pCell) = 0;
        virtual void Data() = 0;
}
```

[0032] Der Entwickler eines Tabellenkalkulationsprogramms muss eine Implementierung der Schnittstelle vorsehen, damit Tabellenobjekte in ein Textverarbeitungsdokument eingebettet werden können.

[0033] Wenn das Textverarbeitungssystem ein Tabellenobjekt einbettet, muss das Programm auf den Code zugreifen, der die Schnittstelle für das Tabellenobjekt implementiert. Um auf den Klassencode zuzugreifen, ist jede Implementierung mit einer eindeutigen Klassenkennzeichnung versehen. Zum Beispiel kann ein Code, der ein durch die Microsoft Corporation entwickeltes Tabellenobjekt implementiert, mit der Klassenkennzeichnung „MSSpreadsheet" versehen sein, während ein Code, der ein durch eine andere Firma entwickeltes Tabellenobjekt implementiert, mit der Klassenkennzeichnung „LTSSpreadsheet" versehen sein kann. Eine Registrierung wird in jedem Computersystem aufrechterhalten, in der die Klassenkennzeichnungen auf den Code abgebildet sind, der die Klasse implementiert. Wenn ein Tabellenkalkulationsprogramm auf einem Computersystem installiert wird, wird gewöhnlich diese Registrierung aktualisiert, um die Verfügbarkeit dieser Klasse von Tabellenkalkulationsobjekten wiederzugeben. Solange ein Entwickler von Tabellenkalkulationen jedes durch die Schnittstelle definierte Funktionsglied implementiert und die Registrierung aufrechterhalten wird, kann das Textverarbeitungsprogramm Instanzen der Tabellenobjekte des Entwicklers in ein Textverarbeitungsprogramm einbetten. Das Textverarbeitungsprogramm greift auf die Funktionsglieder der eingebetteten Tabellenobjekte unabhängig davon zu, von wem oder wie diese implementiert wurden.

[0034] Es kann der Fall sein, dass Tabellenkalkulations-Entwickler nur bestimmte Funktionsglieder implementieren möchten. Zum Beispiel möchte ein Tabellenkalkulations-Entwickler keine Datenbank-Unterstützung im-

plementieren, während er alle anderen Funktionsglieder unterstützen möchte. Damit ein Tabellenkalkulations-Entwickler nur einige der Funktionsglieder unterstützen kann, während weiterhin Objekte eingebettet können werden sollen, werden mehrere Schnittstellen für Tabellenobjekte definiert. Zum Beispiel können die Schnittstellen IDatabase und IBasic wie folgt für ein Tabellenobjekt definiert werden.

```
class IBasic
{       virtual void File() = 0;
        virtual void Edit() = 0;
        virtual void Formula() = 0;
        virtual void Format() = 0;
        virtual void GetCell (string RC, cell *pCell) = 0;
}

class IDatabase
{       virtual void Data() = 0;
}
```

[0035] Jeder Tabellenkalkulations-Entwickler implementiert dann die Schnittstelle IBasid und optional auch noch die Schnittstelle IDatabase.

[0036] In der Laufzeit muss das Textverarbeitungsprogramm bestimmen, ob ein einzubettendes Tabellenobjekt die Schnittstelle IDatabase unterstützt. Um diese Bestimmung vorzunehmen, wird eine andere Schnittstelle (die von jedem Tabellenobjekt implementiert wird) mit einem Funktionselement definiert, das angibt, welche Schnittstellen für das Objekt implementiert werden. Diese Schnittstelle hat den Namen IUnknown (und wird als die unbekannte Schnittstelle oder die Objektverwaltungsschnittstelle bezeichnet) und ist wie folgt definiert.

```
class IUnknown
{virtual HRESULT QueryInterface (REFIID iid, void **ppv) = 0;
virtual ULONG AddRef() = 0;
virtual ULONG Release () = 0;
}
```

[0037] Die Schnittstelle IUnknown definiert das Funktionsglied (die Methode) QueryInterface. Die Methode QueryInterface erhält eine Schnittstellenkennzeichnung (z. B. „IDatabase") in dem Parameter iid (des Typs REFIID) und gibt einen Zeiger auf die Implementierung der identifizierten Schnittstelle für das Objekt aus, für die das Verfahren in dem Parameter ppv aufgerufen wird. Wenn das Objekt die Schnittstelle nicht unterstützt, dann gibt das Verfahren eines Fehlermeldung aus. Der Typ HRESULT gibt einen vordefinierten Status aus, der Typ REFIID gibt einen Bezug auf eine Schnittstellenkennzeichnung an, und der Typ ULONG gibt eine lange Ganzzahl ohne Vorzeichen an.

Code-Tabelle 1

```
HRESULT XX::QueryInterface(REFIID iid; void **ppv)
{       ret = TRUE;
        switch (iid) {
                case IID_Basic:
                        *ppv = pIBasic;
                        break;
                case IID_IDatabase:
                        *ppv = pIDatabase;
                        break;
                case IID_IUnknown:
                        *ppv = this;
                        break;
                default:
                        ret = FALSE;
                }
                if (ret = TRUE) {AddRef();};
                return ret;
}
```

[0038] Die Code-Tabelle 1 enthält einen C++-Pseudocode für eine typische Implementierung der Methode QueryInterface für die Klasse XX, die die Klasse IUnknown erbt. Wenn das Tabellenobjekt die Schnittstelle IDatabase unterstützt, dann enthält die Methode QueryInterface das entsprechende case-Etikett in der switch-Anweisung. Die Variablen pIBasic und pIDatabase verweisen jeweils auf einen Zeiger auf die virtuellen Funktionstabellen der Schnittstellen IBasic und IDatabase. Die Methode QueryInterface ruft die Methode AddRef (weiter unten beschrieben) auf, um eine Bezugszählung für das Objekt der Klasse XX zu inkrementieren, wenn ein Zeiger auf eine Schnittstelle ausgegeben wird.

Code-Tabelle 2

```
void XX::AddRef(){refcount++;}
void XX::Release(){if (--refcount==) delete this;}
```

[0039] Die Schnittstelle IUnknown definiert auch die Methoden AddRef und Release, die zur Implementierung der Bezugszählung verwendet werden. Jedes Mal wenn ein neuer Bezug auf eine Schnittstelle erstellt wird, wird die Methode AddRef aufgerufen, um eine Bezugszählung zu dem Objekt zu inkrementieren. Wenn ein Bezug nicht mehr benötigt wird, wird die Methode Release aufgerufen, um die Bezugszählung zu dem Objekt zu dekrementieren. Wenn der Bezug auf null geht, wird das Objekt freigegeben. Die Code-Tabelle 2 enthält einen C++-Pseudocode für eine typische Implementierung der Methoden AddRef und Release für die Klasse XX, die die Klasse IUnknown erbt.

[0040] Die Schnittstelle IDatabase und die Schnittstelle IBasic erben die Schnittstelle IUnknown. Die folgenden Definitionen zeigen die Verwendung der Schnittstelle IUnknown.

```
class IDatabase : public IUnknown
{ public:
        virtual void Data() = 0;
}

class IBasic : public IUnknown
{ public:
        virtual void File() = 0;
        virtual void Edit() = 0;
        virtual void Formula() = 0;
        virtual void Format() = 0;
        virtual void GetCell (string RC, cell *pCell) = 0;
```

[0041] Fig. 2 ist ein Blockdiagramm, das eine Probedatenstruktur eines Tabellenobjekts mit verschachtelten Klassen zeigt. Das Tabellenobjekt umfasst eine Objektdatenstruktur **201**, eine IBasic-Schnittstellen-Datenstruktur **203**, eine IDatabase-Schnittstellen-Datenstruktur **204**, die Virtuellfunktionstabellen **202**, **205**, **206** und die Methoden **207** bis **221**. Die Objektdatenstruktur **201** enthält einen Zeiger auf die Virtuellfunktionstabelle **202** und Zeiger auf die Schnittstellen IBasic und IDatabase. Jeder Eintrag in der Virtuellfunktionstabelle **202** enthält einen Zeiger auf eine Methode der Schnittstelle IUnknown. Die IBasic-Schnittstellen-Datenstruktur **203** enthält einen Zeiger auf die Virtuellfunktionstabelle **205**. Jeder Eintrag in der Virtuellfunktionstabelle **205** enthält einen Zeiger auf eine Methode der Schnittstelle IBasic. Die IDatabase-Schnittstellen-Datenstruktur **204** enthält einen Zeiger auf die Virtuellfunktionstabelle **206**. Jeder Eintrag in der Virtuellfunktionstabelle **207** enthält einen Zeiger auf eine Methode der Schnittstelle IDatabase. Weil die Schnittstellen IBasic und IDatabase die Schnittstelle IUnknown erben, enthält jede virtuelle Funktionstabelle **205** und **206** einen Zeiger auf die Methoden QueryInterface, AddRef und Release. Im Folgenden wird eine Objektdatenstruktur durch die Form **222** wiedergegeben, die mit den Etiketten der Schnittstellen versehen ist, über die auf das Objekt zugegriffen werden kann.

[0042] Der folgende Pseudocode zeigt, wie ein Textverarbeitungsprogramm bestimmt, ob ein Tabellenobjekt die Schnittstelle IDatabase unterstützt.

```
if (pSpreadsheet->QueryInterface(„IDatabase", &pIDatabase))
// IDatabase supported
else
// IDatabase not supported
```

[0043] Der Zeiger pSpreadsheet ist ein Zeiger auf eine Instanz der Tabellenklasse von Fig. 2. (pSpreadsheet zeigt auf die Datenstruktur **201**.) Wenn das Objekt die Schnittstelle IDatabase unterstützt, setzt die durch die Methode **207** definierte Methode QueryInterface den Zeiger pIDatabase, sodass er auf die IDatabase-Datenstruktur **204** zeigt, und gibt diesen als Wert aus.

[0044] Normalerweise kann ein Objekt durch eine variable Definition oder durch den „neuen" Operator instanziert werden (eine Instanz des Objekts wird im Speicher erstellt). Beide Techniken der Instanzierung benötigen jedoch die Klassendefinition bei der Kompilierung. Es besteht ein Bedarf für eine neue Technik, die es einem Textverarbeitungsprogramm ermöglicht, ein Tabellenobjekt in Laufzeit zu instanzieren. Eine Technik sieht eine globale Funktion CreateInstanceXX vor, die im Folgenden definiert ist.

```
static void CreateInstanceXX (REFIID iid, void **ppv) = 0;
```

[0045] Die Methode CreateInstance XX instanziert ein Objekt der Klasse XX und gibt einen Zeiger ppv auf die Schnittstelle des durch den Parameter iid angegebenen Objekts aus.

[0046] EP-A-0 546 682 beschreibt eine Methode und ein System zur Unterstützung einer dynamischen Bindung zwischen einer abgeleiteten Klasse und ihrer Elterklasse. Wenn die abgeleitete Klassenimplementierung Informationen zu der Größe der Elterklasse-Statusdatenstruktur benötigt, werden die entsprechenden Informationen von dem Elterklassen-Objekt abgerufen. Es wird eine Methodenprozedurentabelle mit den Adressen von verschiedenen Methoden für das bestimmte Objekt verwendet. Die durch Objekte geerbten Methoden wei-

sen ihre Methodenprozeduradressen an demselben Offset im Speicher auf, mit dem sie in der Methodenprozedurentabelle erscheinen, was in Etikett 1240 der Elterklasse angegeben wird, von der sie geerbt werden.

[0047] Es ist eine Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Aggregieren von Objekten anzugeben.

[0048] Diese Aufgabe wird durch den Gegenstand der unabhängigen Ansprüche 1 und 14 gelöst.

[0049] Bevorzugte Ausführungsformen werden durch den Gegenstand der abhängigen Ansprüche definiert.

[0050] Es ist eine andere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum dynamischen Modifizieren des Objektverhaltens anzugeben.

[0051] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum dynamischen Aggregieren von Objekten anzugeben.

[0052] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum statischen Aggregieren von Objekten anzugeben.

[0053] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Einschließen eines Objekts in einem anderen Objekt anzugeben, wobei eine Schnittstelle des eingeschlossenen Objekts für einen Client des eingeschlossenen Objekts exponiert wird.

[0054] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Einschließen eines Objekts in einem anderen Objekt anzugeben, nachdem das einschließende Objekt instanziert wurde.

[0055] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum dynamischen Kombinieren von Objekten verschiedenen Typs zu einem einzelnen Objekt anzugeben.

[0056] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Implementieren eines Objekts anzugeben, das entweder in einem anderen Objekt eingeschlossen werden kann oder nicht in einem anderen Objekt eingeschlossen werden kann, ohne die Implementierung des Objekts zu modifizieren.

[0057] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Implementieren eines aggregierten Objekts anzugeben, sodass ein Client nicht erkennt, dass es sich um ein aggregiertes Objekt handelt.

[0058] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Einschließen von Objekten anzugeben, wobei ein eingeschlossenes Objekt selbst ein einschließendes Objekt sein kann, bis hin zu einer beliebigen Einschließungsebene.

[0059] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Verbessern des Verhaltens eines Basisobjekts anzugeben, indem eine Schnittstelle hinzugefügt wird, die das Standardverhalten des Basisobjekts außer Kraft setzt.

[0060] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Bereitstellen einer Standardfunktionalität für Objekte anzugeben, indem diese in einem einschließenden Objekt eingeschlossen werden, wobei das eingeschlossene oder das einschließendes Objekt die Standardfunktionalität implementiert.

[0061] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System zum Implementieren eines steuernden Verhaltens über die übliche Funktionalität in eingeschlossenen Objekten anzugeben.

[0062] Es ist eine weitere Aufgabe der vorliegenden Erfindung, ein Verfahren und ein System anzugeben, die bestimmen, welche Schnittstelle für einen Client vorgesehen werden soll, wenn der Client eine Schnittstelle anfordert, die durch mehr als ein eingeschlossenes Objekt implementiert wird.

[0063] Diese und andere im Verlauf der folgenden Beschreibung deutlich werdende Aufgaben werden durch

das Verfahren und das System zum Aggregieren von Objekten in einem Computersystem gelöst. In einer bevorzugten Ausführungsform aggregiert das System ein eingeschlossenes Objekt in einem einschließenden Objekt. Das eingeschlossene Objekt weist eine Objektverwaltungsschnittstelle und eine oder mehrere externe Schnittstellen auf, während das einschließende Objekt eine steuernde Objektverwaltungsschnittstelle aufweist. Jede durch das aggregierte Objekt für einen Clienten exponierte Schnittstelle weist ein Abfragefunktionsglied auf, um eine Kennzeichnung einer Schnittstelle zu empfangen und einen Bezug auf die identifizierte Schnittstelle auszugeben. Das Abfragefunktionsglied der steuernden Objektverwaltungsschnittstelle des einschließenden Objekts empfängt eine Kennzeichnung einer durch das einschließende Objekt exponierten Schnittstelle und gibt einen Bezug auf die exponierte Schnittstelle aus. Ein bevorzugtes Verfahren erstellt eine Instanz des eingeschlossenen Objekts. Wenn das Abfragefunktionsglied einer exponierten Schnittstelle des eingeschlossenen Objekts eine Kennzeichnung einer Schnittstelle empfängt, ruft es das Abfragefunktionsglied der steuernden Objektverwaltungsschnittstelle des einschließenden Objekts auf, gibt diesem die empfangene Kennzeichnung und gibt den durch das aufgerufene Abfragefunktionselement der steuernden Objektverwaltungsschnittstelle des einschließenden Objekts ausgegebenen Bezug als einen Bezug an die identifizierte Schnittstelle aus.

[0064] In einer bevorzugten Ausführungsform der statischen Aggregation wird ein Abfragefunktionsglied eines eingeschlossenen Objekts mit Wissen der externen Schnittstellen des eingeschlossenen Objekts implementiert und hat kein Wissen zu den Schnittstellen (neben der steuernden Objektverwaltungsschnittstelle) des einschließenden Objekts oder anderer eingeschlossener Objekte. Das Abfragefunktionsglied einer steuernden Objektverwaltungsschnittstelle des einschließenden Objekts wird mit Wissen der exponierten Schnittstellen der eingeschlossenen Objekte implementiert.

[0065] In einer bevorzugten Ausführungsform der dynamischen Aggregation kann ein Objekt dynamisch modifiziert werden, indem Schnittstellen-Instanzen, die durch Objekte implementiert werden, während der Ausführung eines Client-Programms aggregiert werden können. Schnittstellen werden durch das dynamische Einschließen der Objekte aggregiert, die diese zu einem Mehrtypen-Objekt implementieren. Jede hinzuzufügende Schnittstelle wird durch ein Objekt implementiert, das aggregierbar ist. Ein Mehrtypen-Objekt wird erstellt, um als einschließendes Objekt zu dienen. Das Mehrtypen-Objekt weist eine Funktionsglied zum Hinzufügen einer Schnittstelle auf, das verwendet werden kann, um Schnittstellen zu aggregieren, indem diese zu dem einschließenden Mehrtypen-Objekt hinzugefügt werden.

[0066] Das Mehrtypen-Objekt weist auch ein Funktionsglied zum Hinzufügen eines Objekts auf, um alle Schnittstellen eines Objekts zu aggregieren. Das Mehrtypen-Objekt weist auch ein Abfragefunktionsglied zum Abrufen von Bezügen zu den hinzugefügten Schnittstellen auf eine Anfrage von einem Client hin auf. Dieses Abfragefunktionsglied ist Teil der steuernden Objektverwaltungsschnittstelle des einschließenden Mehrtypen-Objekts. Weiterhin wird eine Instanz eines Objekts erstellt, die die zu aggregierende Schnittstelle implementiert. Während der Erstellung wird ein Zeiger auf das einschließende Mehrtypen-Objekt zu dem einzuschließenden Objekt gegeben, damit das eingeschlossene Objekt mit dem einschließenden Mehrtypen-Objekt kommunizieren kann. Das erstellte Objekt für die Implementierung der zu aggregierenden Schnittstelle weist ein Abfragefunktionsglied auf, das das Abrufen eines Bezugs auf die zu aggregierende Schnittstelle unterstützt. Ein bevorzugtes Verfahren ruft das Funktionsglied zum Hinzufügen einer Schnittstelle oder das Funktionsglied zum Hinzufügen eines Objekts des einschließenden Mehrtypen-Objekt auf und gibt diesem einen Bezug auf das erstellte Objekt, das die zu aggregierende Schnittstelle implementiert. Später wird das Abfragefunktionsglied des einschließenden Mehrtypen-Objekts aufgerufen, um einen Bezug auf die aggregierte Schnittstelle abzurufen.

[0067] Fig. 1 ist ein Blockdiagramm, das typische Datenstrukturen zeigt, die zur Wiedergabe eines Objekts verwendet werden.

[0068] Fig. 2 ist ein Blockdiagramm, das eine Probedatenstruktur eines Tabellenobjekts unter Verwendung von verschachtelten Klassen zeigt.

[0069] Fig. 3 ist ein Blockdiagramm, das ein aggregiertes Objekt zeigt.

[0070] Fig. 4 ist ein Blockdiagramm des Datenstrukturlayouts einer Instanz eines Objekts der Klasse S1.

[0071] Fig. 5 ist ein Blockdiagramm des Datenstrukturlayouts eines Objekts der Klasse S3.

[0072] Fig. 6A und Fig. 6B sind Blockdiagramme, die das Zusammenwirken zwischen einem einschließen-

den Objekt und einem eingeschlossenen Objekt zeigen.

[0073] Fig. 7A, Fig. 7B und Fig. 7C sind Blockdiagramme der Sequenz zum Hinzufügen von zwei Objekten zu einem Mehrtypen-Objekt.

[0074] Fig. 8 ist ein Blockdiagramm des Datenstrukturlayouts einer Instanz eines Objekts der Klasse S1.

[0075] Fig. 9 ist ein Flussdiagramm der Methode AddObjekt der Schnittstelle IMultitype, die durch ein Mehrtypen-Objekt implementiert wird.

[0076] Fig. 10 ist ein Flussdiagramm der Methode QueryInterface der steuernden Schnittstelle IUnknown für ein Mehrtypen-Objekt.

[0077] Fig. 11 ist ein Blockdiagramm, das das Datenstrukturlayout eines Mehrtypen-Objekts in Entsprechung zu Fig. 7C zeigt, nachdem die Schnittstellen IBasic, IPrint und IDatabase dynamisch unter Verwendung der Methode AddObject aggregiert wurden.

[0078] Fig. 12 ist eine bildliche Wiedergabe eines Tabellenobjekts und eines Datenbankabfrage-Objekts, die zusammen aggregiert werden können, um ein verbundenes Datenbankabfrage-Objekt zu erstellen.

[0079] Fig. 13 ist ein Blockdiagramm eines aggregierten verbundenen Datenbankabfrage-Objekts.

[0080] Die vorliegende Erfindung gibt ein Verfahren in einem Computersystem zum Aggregieren von Objekten an. Objekte können entweder statisch oder dynamisch aggregiert werden. Bei einer statischen Aggregation weist ein einschließendes Objekt gewöhnlich beim Kompilieren ein Wissen zu den exponierten Schnittstellen der eingeschlossenen Objekt auf. Die Objektverwaltungsschnittstelle des einschließenden Objekts wird deshalb dafür angepasst, Schnittstellenzeiger auf exponierte Schnittstellen von eingeschlossenen Objekten unter Verwendung dieses Wissens auszugeben. Instanzen dieser statisch aggregierten Objekte werden dynamisch erstellt (in Laufzeit).

[0081] Unter Verwendung einer dynamischen Aggregation wird ein einschließendes Objekt instanziert und kann verwendet werden, um Objekte oder Schnittstellen in Laufzeit zu aggregieren. Das einschließende Objekt weist kein a-priori-Wissen zu den eingeschlossenen Objekten oder Schnittstellen auf, sodass kein zur Kompilationszeit bereitgestelltes Wissen durch das einschließende Objekt verwendet wird. Entsprechend verfügen die eingeschlossenen Objekte und Schnittstellen über kein Wissen zu der Implementierung oder dem Vorhandensein von Schnittstellen des einschließenden Objekts, mit Ausnahme der steuernden Objektverwaltungsschnittstelle, die zum Aggregieren von Objekten und Schnittstellen verwendet wird. Außerdem ist ein Regelmechanismus vorgesehen, um den Zugriff auf aggregierte Objekte und Schnittstellen zu steuern.

[0082] Jeder dieser Typen von Aggregation wird in den folgenden Abschnitten erläutert. In einer bevorzugten Ausführungsform sind die Verfahren und Systeme der vorliegenden Erfindung auf einem Computersystem implementiert, das eine zentrale Verarbeitungseinheit, einen Speicher und Ein-/Ausgabegeräte umfasst.

Statische Aggregation

[0083] In einer bevorzugten Ausführungsform der statischen Aggregation sieht ein aggregiertes Objekt eine Vielzahl von Schnittstellen zu seinen Clients vor. Das Computerprogramm, das ein Objekt instanziert, wird als Client bezeichnet. Ein aggregiertes Objekt umfasst ein oder mehrere eingeschlossene Objekte und eine Implementierung der Schnittstelle IUnknown, die als die steuernde Schnittstelle IUnknown des aggregierten Objekts bezeichnet wird. Ein aggregiertes Objekt exponiert für seine Clients seine eigenen Schnittstellen und die Schnittstellen der eingeschlossenen Objekte. Die Methode QueryInterface der steuernden Schnittstelle IUnknown gibt einen Zeiger auf jede durch das aggregierte Objekt exponierte Schnittstelle aus. Das aggregierte Objekt instanziert jedes eingeschlossene Objekt. Diese Instanzierung kann während der Erstellung des aggregierten Objekts durchgeführt werden oder kann aufgeschoben werden, bis eine Schnittstelle des eingeschlossenen Objekts angefordert wird. Jedes eingeschlossene Objekt enthält einen Zeiger auf die steuernde Schnittstelle IUnknown. Die Methode QueryInterface einer exponierten Schnittstelle eines eingeschlossenen Objekts ist vorzugsweise implementiert, um die Methode QueryInterface einer Schnittstelle IUnknown aufzurufen. Wenn das eingeschlossene Objekt implementiert ist, weiß der Entwickler gewöhnlich nicht, welche Schnittstellen das einschließende Objekt exponieren kann. Folglich ruft die Methode QueryInterface eines eingeschlossenen Objekts die Methode QueryInterface der steuernden Schnittstelle IUnknown auf, um einen Zei-

ger auf die angeforderte Schnittstelle abzurufen. Die Methode QueryInterface der steuernden Schnittstelle IUnknown wird gewöhnlich mit einem Wissen zu allen exponierten Schnittstellen implementiert. Wenn ein Objekt nicht eingeschlossen ist, ist die steuernde Schnittstelle IUnknown die Schnittstelle IUnknown des Objekts. Wenn umgekehrt ein Objekt eingeschlossen ist, ist die steuernde Schnittstelle IUnknown die Schnittstelle IUnknown des einschließenden Objekts.

[0084] In einer bevorzugten Ausführungsform hält ein aggregiertes Objekt eine Bezugszählung aufrecht. Wenn das aggregierte Objekt instanziert wird, wird seine Bezugszählung auf eins gesetzt. Die Methode QueryInterface der steuernden Schnittstelle IUnknown inkrementiert die Bezugszählung, wenn ein Bezug auf den Client ausgegeben wird. Die Methode AddRef einer exponierten Schnittstelle eines eingeschlossenen Objekts ruft die Methode AddRef der steuernden Schnittstelle IUnknown auf, um die Bezugszählung des aggregierten Objekts zu inkrementieren. Entsprechend ruft die Methode Release einer exponierten Schnittstelle eines eingeschlossenen Objekts die Methode Release der steuernden Schnittstelle IUnknown auf, um die Bezugszählung des aggregierten Objekts zu dekrementieren und das aggregierte Objekt zu löschen, wenn die Bezugszählung gleich null ist. Wenn ein eingeschlossenes Objekt instanziert wird, wird die Bezugszählung des eingeschlossenen Objekts auf eins gesetzt. Wenn das aggregierte Objekt gelöscht wird, wird die Methode Release der Schnittstelle IUnknown jedes eingeschlossenen Objekts aufgerufen, um das eingeschlossene Objekt zu löschen.

[0085] Fig. 3 ist ein Blockdiagramm, das ein aggregiertes Objekt zeigt. Das aggregierte S3-Objekt exponiert die Schnittstellen A, B, C, F und die steuernde Schnittstelle IUnknown. Das aggregierte (einschließende) S3-Objekt umfasst das eingeschlossene S1-Objekt **303**, das eingeschlossene S2-Objekt **302** und die Implementierung I3 **304**. Das eingeschlossene S1-Objekt implementiert die externen Schnittstellen C und D, und das eingeschlossene S2-Objekt implementiert die externen Schnittstellen E und F. (Eine externe Schnittstelle ist eine Schnittstelle eines Objekts, das durch ein einschließendes Objekt exponiert werden kann. Eine interne Schnittstelle ist eine Schnittstelle eines Objekts, das nicht durch ein einschließendes Objekt exponiert werden kann.) Die Implementierung I3 implementiert die externen Schnittstellen A, B und die steuernde Schnittstelle IUnknown. Ein Client des aggregierten S3-Objekts muss nicht wissen, dass das Objekt ein Aggregat ist. Das aggregierte S3-Objekt instanziert die Objekte S1 und S2 entweder während der Erstellung des aggregierten S3-Objekts oder zu einem späteren Zeitpunkt. Die Implementierung I3 enthält Zeiger auf die Schnittstellen IUnknown der Objekte S1 und S2. Die Objekte S1 und S2 werden initialisiert, um einen Zeiger zu der steuernden Schnittstelle IUnknown zu enthalten.

[0086] Die Methode QueryInterface einer exponierten Schnittstelle kann einen Zeiger zu jeder exponierten Schnittstelle ausgeben und inkrementiert die Bezugszählung des aggregierten Objekts, wenn ein Zeiger ausgegeben wird. Die Methode QueryInterface der steuernden Schnittstelle IUnknown hat direkten Zugriff auf die Zeiger der Schnittstellen A, B und der steuernden Schnittstelle IUnknown – diese Implementierung I3 implementiert und ruft die Methode QueryInterface der Schnittstelle IUnknown der eingeschlossenen Objekte auf, um Zeiger zu den exponierten Schnittstellen – C und F – der eingeschlossenen Objekte S1 und S2 abzurufen. Wenn ein Zeiger auf eine exponierte Schnittstelle ausgegeben wird, inkrementiert die Methode QueryInterface der steuernden Schnittstelle IUnknown die Bezugszählung des aggregierten S3-Objekts, indem sie die Methode AddRef der steuernden Schnittstelle IUnknown aufruft. Die Methode QueryInterface jeder exponierten Schnittstelle (neben der steuernden Schnittstelle IUnknown) ruft vorzugsweise die Methode QueryInterface der steuernden Schnittstelle IUnknown auf.

Code-Tabelle 3

```
void CreateInstanceS1 (IUnknown *punkOuter, REFIID iid, void **ppv)
{       IUnknown *punk;
        S1::CreateInstance (punkOuter, &punk);
```

```
        punk->QueryInterface (iid, ppv);
        punk->Release ();
}

class IC: public IUnknown
{ // methods of IC}

class ID: public IUnknown
{ // methods of ID}

class S1: public IUnknown
{
public:
        static void CreateInstance(IUnknown *punkOuter, IUnknown **punk)
        {       S1 *pS1 = new S1(punkOuter);
                pS1->QueryInterface(IID_IUnknown, ppunk);
        }

private:
        void S1(IUnknown *punkOuter) : m_C(this), m_D(this)
        {       if (punkOuter == NULL)
                        m_punkOuter = this
                else
                        m_punkOuter = punkOuter;
                m_refcount = 0;
        }

        class C: public IC
        {
        public:
                void C(S1 *pS1) {m_pS1 = pS1;}

                virtual boolean QueryInterface (REFIID iid, void **ppv)
                { return m_pS1->m_punkOuter->QueryInterface(iid, ppv);}

                virtual void AddRef()
                { m_pS1->m_punkOuter->AddRef();}
```

```
        virtual void Release()
        { m_pS1->m_punkOuter->Release();}

        // other methods of IC

    private:
        S1      *m_pS1;
    }
    friend C;
    C       m_C;

    class D: public ID
    {
    public:
        void D(S1 *pS1) {m_pS1 = pS1;}

        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_pS1->m_punkOuter->QueryInterface(iid, ppv);}

        virtual void AddRef()
        { m_pS1->m_punkOuter->AddRef();}

        virtual void Release()
        { m_pS1->m_punkOuter->Release();}

        // other methods of ID

    private:
        S1      *m_pS1
    }
    friend D;
    D       m_D;

public:
    virtual boolean QueryInterface (REFIID iid, void **ppv)
    {       ret = TRUE;
```

```
            switch (iid)
            { case IID_C:
                    *ppv = &m:_C;
                    m_punkOuter->AddRef();
                    break;
            case IID_D:
                    *ppv = &m_D;
                    m_punkOuter->AddRef();
                    break;
            case IID_IUnknown:
                    *ppv = this;
                    AddRef();
                    break;
            default:
                    ret = FALSE;
            }
            return ret;
        }

        virtual void AddRef(){ m_refcount++;}

        virtual void Release() {if (--m_refcount == 0) delete this;}

private:
        IUnknown    *m_punkOuter;
        int         m_refcount;
}
```

[0087] Die Code-Tabelle 3 enthält einen C++-Pseudocode für eine bevorzugte Klassendefinition des S1-Objekts, die in einem Aggregat (einem aggregierbaren Objekt) zusammen mit einer globalen Funktion eingeschlossen werden kann, die eine Instanz des Objekts erstellt. Die Klassen IUnknown, IC und ID sind Schnittstellen, die die Methoden jeder Schnittstelle definieren. Die Klasse S1 implementiert die Schnittstelle IUnknown, die Schnittstelle IC und die Schnittstelle ID. Die Klasse S1 implementiert die Schnittstellen IC und ID als externe Schnittstellen. Fig. 4 ist ein Blockdiagramm des Datenstrukturlayouts einer Instanz eines Objekts der Klasse S1. Die Instanzstruktur **401** enthält die Datenglieder der Klasse S1 (m_C, m_D, m_punkOuter, m_refcount) und einen Zeiger auf den Virtuellfunktionstabellen-Zeiger (S1::vfptr). Die Datenglieder m_C und m_D sind jeweils Instanzen eines Objekts der Klassen C und D. Die Klassen C und D sind Freunde der Klasse S1, sodass die Objekte C und D auf die privaten Glieder der Klasse S1 zugreifen können. Der Virtuellfunktionstabellen-Zeiger S1::vfptr zeigt auf die Virtuellfunktionstabelle **402**, der Virtuellfunktionstabellen-Zeiger in dem Datenglied m_C S1::C::vfptr zeigt auf die Virtuellfunktionstabelle **403** und der Virtuellfunktionstabellen-Zeiger in dem Datenglied m_D S1::D::vfptr zeigt auf die Virtuellfunktionstabelle **403A**. Die Virtuellfunktionstabelle **402** enthält Zeiger auf die virtuellen Funktionen, die für die Schnittstelle IUnknown definiert sind; die Virtuellfunktionstabelle **403** enthält Zeiger auf die virtuellen Funktionen, die für die Schnittstelle C definiert sind; und die Virtuellfunktionstabelle **403A** enthält Zeiger auf die virtuellen Funktionen, die für die Schnittstelle D definiert sind. Die Ellipsen in den Virtuellfunktionstabellen **403** und **403A** geben jeweils Zeiger auf zusätzliche

Funktionsglieder der Klassen C und D an. Die Funktionen **404** bis **408** sind Funktionsglieder der Klasse S1. Die Funktion **407** ist der Konstruktor für die Klasse S1. Die Funktion **408** ist die Funktion CreateInstance für die Klasse S1. Die Funktionen **409** bis **412** sind die Funktionsglieder der Klasse C. Die Funktion **412** ist der Konstruktor für die Klasse C. Die Funktionen **413** bis **416** sind die Funktionsglieder der Klasse D. Die Funktion **416** ist der Konstruktor für die Klasse D.

[0088] Wie in der Code-Tabelle 3 gezeigt, gibt die Methode S1::QueryInterface einen Zeiger auf die Schnittstelle C, die Schnittstelle D oder die Schnittstelle IUnknown aus. Wenn ein Zeiger auf die Schnittstelle C oder die Schnittstelle D ausgegeben wird, ruft die Methode S1:QueryInterface die Methode S1::AddRef auf, um die Bezugszählung für das S1-Objekt zu inkrementieren. Die Methode S1::AddRef inkrementiert die Bezugszählung, und die Methode S1::Release dekrementiert die Bezugszählung und löscht das S1-Objekt, wenn die Bezugszählung gleich null ist. Wenn ein Zeiger auf die Schnittstelle C oder die Schnittstelle D ausgegeben wird, ruft die Methode S1:QueryInterface die Methode AddRef der steuernden Schnittstelle IUnknown auf, wobei es sich um die Methode S1:AddRef handelt, wenn das S1-Objekt nicht aggregiert ist.

[0089] Die globale Funktion CreateInstanceS1 erstellt eine Instanz eines Objekts der Klasse S1. Ein Client ruft diese Funktion auf, um ein Objekt der Klasse S1 zu instanzieren. Auf diese Weise kann ein Client ein Objekt der Klasse S1 instanzieren, ohne auf die Klassendefinition der Klasse S1 beim Kompilieren oder in Laufzeit zugreifen zu müssen. Die Funktion CreateInstanceS1 erhält einen Zeiger auf die steuernde Schnittstelle IUnknown (punkOuter), wenn das instanzierte S1-Objekt in einem aggregierten Objekt eingeschlossen ist, sowie eine Kennzeichnung (iid) einer auszugebenden Schnittstelle. Die Funktion CreateInstanceS1 gibt einen Zeiger (ppv) auf die identifizierte Schnittstelle aus. Die Funktion CreateInstanceS1 ruft die Methode S1::CreateInstance auf und gibt dieser den Parameter punkOuter. Die Methode S1::CreateInstance instanziert ein S1-Objekt und gibt einen Zeiger (punk) auf die Schnittstelle IUnknown des S1-Objekts aus. Die Funktion CreateInstanceS1 ruft die Methode QueryInterface des S1-Objekts auf, um einen Zeiger auf die identifizierte Schnittstelle abzurufen. Die Funktion CreateInstanceS1 ruft dann die Methode Release des S1-Objekts auf, weil der temporäre Zeiger punk nicht mehr benötigt wird.

[0090] Die Methode S1::CreateInstance instanziert ein S1-Objekt und gibt einen Zeiger (ppunk) auf die Schnittstelle IUnknown des S1-Objekts aus. Die Methode S1::CreateInstance erhält einen Zeiger (punkOuter) auf die steuernde Schnittstelle IUnknown. Die Methode S1::CreateInstance verwendet den Operator new, um das S1-Objekt zu instanzieren. Während der Instanzierung wird der Konstruktor S1::S1 aufgerufen und erhält den Wert des Parameters punkOuter. Nachdem das S1-Objekt erstellt wurde, ruft die Methode S1::CreateInstance die Methode S1::QueryInterface auf, um einen Zeiger auf die Schnittstelle IUnknown des S1-Objekts abzurufen.

[0091] Der Konstruktor S1::S1 initialisiert die Datenglieder m_C, m_D, m_punkOuter und m_refcount. Der Konstruktor S1::S1 erhält den Parameter punkOuter. Während der Instanzierung der Datenglieder m_C und m_D werden die Konstruktoren C::C und D::D aufgerufen und erhalten diesen Zeiger auf das S1-Objekt. Wenn der Wert des Parameters punkOuter gleich NULL ist, setzt der Konstruktor S1:S1 das Datenglied m_punkOuter auf den Wert für diesen Zeiger (der auf das neu instanzierte S1-Objekt zeigt). Wenn der Wert des Parameters punkOuter nicht-NULL ist, setzt der Konstruktor S1::S1 das Datenglied m_punkOuter auf den Wert des Parameters punkOuter. Das Datenglied m_punkOuter zeigt auf den Wert der steuernden Schnittstelle IUnknown des Aggregats, wenn das S1-Objekt eingeschlossen ist, und zeigt auf die steuernde Schnittstelle IUnknown des S1-Objekts, wenn das S1-Objekt nicht eingeschlossen ist. Der Konstruktor S1::S1 initialisiert auch das Datenglied m_refcount auf null.

[0092] Der Konstruktor C::C erhält einen Zeiger auf das S1-Objekt. Der Konstruktor S1::S1 speichert den erhaltenen Zeiger in dem Datenglied C::m_pS1. Das Datenglied C::m_pS1 wird durch die Methoden der Klasse C verwendet, um auf das Datenglied S1::m_punkOuter zuzugreifen.

[0093] Die Methoden C::QueryInterface, C::AddRef und C::Release rufen die entsprechenden Methoden der Schnittstelle IUnknown auf, auf die durch das Datenglied S1::m_punkOuter gezeigt wird, das, wenn das S1-Objekt eingeschlossen ist, auf die steuernde Schnittstelle IUnknown des Aggregats zeigt.

[0094] Der Konstruktor und die anderen Methoden der Klasse D sind denjenigen der Klasse C analog.

[0095] Fig. 4 zeigt eine Instanz eines S1-Objekts, das nicht Teil eines Aggregats ist. Die Datenglieder S1::C::m_pS1, S1::D::m_pS1 und S1::m_punkOuter werden initialisiert, um auf das S1-Objekt selbst zu zeigen. Die Methoden QueryInterface, AddRef und Release der Datenglieder m_C und m_D rufen die Methoden

IUnknown der Schnittstelle des S1-Objekts auf.

[0096] Das S2-Objekt, das die Schnittstellen E und F implementiert, ist dem S1-Objekt wie oben beschrieben analog.

Code-Tabelle 4

```
void CreateInstanceS3 (IUnknown *punkOuter, REFIID iid, void **ppv)
{       IUnknown *punk;
        S3::CreateInstance (punkOuter, &punk);
        punk->QueryInterface (iid, ppv);
        punk->Release ();
}

class IA: public IUnknown
{ // methods of class IA}

class IB: public IUnknown
{ // methods of class IB}

class S3: public IUnknown
{
public:
        static void CreateInstance(IUnknown *punkOuter, IUnknown **punk)
        {       S3 *pS3 = new S3(punkOuter);
                CreateInstanceS1(pS3->m_punkOuter, IID_IUnknown, pS3->m_punkS1);
                CreateInstanceS2(pS3->m_punkOuter, IID_IUnknown, pS3->m_punkS2);
```

```
        pS3->QueryInterface(IID_IUnknown, ppunk);}

private:
    void S3(IUnknown *punkOuter) : m_A(this), m_B(this)
    {   if (punkOuter == NULL)
            m_punkOuter = this
        else
            m_punkOuter = punkOuter;
        m_refcount = 0;}

    void ~S3()   {m_punkS1->Release();
            m_punkS2->Release();}

    class A: public IA
    {
    public:
        void A(S3 *pS3) {m_pS3 = pS3}

        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_pS3->m_punkOuter->QueryInterface(iid, ppv);}

        virtual void AddRef()
        { m_pS3->m_punkOuter->AddRef();}

        virtual void Release()
        { m_pS3->m_punkOuter->Release();}

        \\ other methods of IA

    private:
        S3   *m_pS3;
    };
    friend A;
    A    m_A;

    class B: public IB
    {
```

```
    public:
        void B(S3 *pS3) {m_pS3 = pS3}

        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_pS13->m_punkOuter->QueryInterface(iid, ppv);}

        virtual void AddRef()
        { m_pS3->m_punkOuter->AddRef();}

        virtual void Release()
        { m_pS3->m_punkOuter->Release();}

        \\ other methods of IB

    private:
        S3      *m_pS3;
    };
    friend B;
    B       m_B;

public:
    virtual boolean QueryInterface (REFIID iid, void **ppv)
    {       ret = TRUE;
        switch (iid)
        { case IID_C:
            ret = m_punkS1->QueryInterface(iid, ppv);
            break;
        case IID_F:
            ret = m_punkS2->QueryInterface(iid, ppv);
            break;
        case IID_A:
            *ppv = &m_A;
            m_punkOuter->AddRef();
            break;
        case IID_B:
            *ppv = &m_B;
            m_punkOuter->AddRef();
```

```
                break;
        case IID_IUnknown:
                *ppv = this;
                AddRef();
                break;
        default:
                ret = FALSE;
        }
        return ret;
    }

    virtual void AddRef(){ m_refcount++;}

    virtual void Release() {if (--m_refcount == 0) delete this;}

private:
    IUnknown    *m_punkOuter;
    int         m_refcount;
    IUnknown    *m_punkS1;
    IUnknown    *m_punkS2;
};
```

[0097] Die Code-Tabelle 4 enthält einen C++-Pseudocode für eine bevorzugte Klassendefinition eines aggregierten Objekts. Die Klasse S3 exponiert die Schnittstellen IUnknown, A, B, C und F. Um die Schnittstelle C vorzusehen, schließt die Klasse S3 ein S1-Objekt ein und exponiert die Schnittstelle C. Um die Schnittstelle F vorzusehen, schließt die Schnittstelle S3 ein S2-Objekt ein und exponiert die Schnittstelle F. Das S3-Objekt exponiert die Schnittstellen C und F, indem es Zeiger auf die Schnittstellen C und F über die Methode QueryInterface der steuernden Schnittstelle IUnknown ausgibt. Die Schnittstelle D des S1-Objekts und die Schnittstelle E des S2-Objekts sind externe Schnittstellen, aber das S2-Objekt exponiert diese Schnittstellen nicht.

[0098] Die Methoden S3::QueryInterface, S3::AddRef und S3::Release bilden die steuernde Schnittstelle IUnknown für das Aggregat. Die Methode S3::QueryInterface gibt einen Zeiger auf die steuernden Schnittstellen A, B, C oder F aus. Wenn ein Zeiger auf die steuernde Schnittstelle IUnknown ausgegeben wird, ruft die Methode S3::QueryInterface die Methode S3::AddRef auf, um die Bezugszählung für das S3-Objekt zu inkrementieren. Die Methode S3::AddRef inkrementiert die Bezugszählung, und die Methode S3::Release dekrementiert die Bezugszählung und löscht das S3-Objekt, wenn die Bezugszählung gleich null ist. Wenn ein Zeiger auf die Schnittstellen A, B, C und F ausgegeben wird, ruft die Methode S3::QueryInterface die Methode AddRef der steuernden Schnittstelle auf, die, wenn das S3-Objekt nicht aggregiert ist, die Methode S3::AddRef ist.

[0099] Die globale Funktion CreateInstanceS3 erstellt eine Instanz eines Objekts der Klasse S3. Ein Client ruft diese Funktion auf, um ein Objekt der Klasse S3 zu instanzieren. Auf diese Weise kann ein Client ein Objekt der Klasse S3 instanzieren, ohne auf die Klassendefinition der Klasse S3 beim Kompilieren oder in Laufzeit zugreifen zu müssen. Die Funktion CreateInstanceS3 erhält einen Zeiger auf die steuernde Schnittstelle IUnknown (punkOuter), wenn das instanzierte S3-Objekt in einem aggregierten Objekt eingeschlossen wird, sowie eine Kennzeichnung (iid) einer Schnittstelle, die durch die auszugebende Klasse S3 exponiert wird. Die Funktion CreateInstanceS3 gibt einen Zeiger (ppv) auf die identifizierte Schnittstelle aus. Die Funktion CreateInstanceS3 ruft die Methode S3::CreateInstance auf und gibt dieser den Parameter punkOuter. Die Methode S3::CreateInstance instanziert ein S3-Objekt und gibt einen Zeiger (ppunk) auf die Schnittstelle

IUnknown des S3-Objekts aus. Die Funktion CreateInstanceS3 ruft dann die Methode S3::QueryInterface auf, um einen Zeiger auf die identifizierte Schnittstelle abzurufen. Die Funktion CreateInstanceS3 ruft dann die Methode S3::Release auf, weil der temporäre Zeiger punkt nicht mehr benötigt wird.

[0100] Die Methode S3::CreateInstance instanziert ein S3-Objekt und gibt einen Zeiger (ppunk) auf die Schnittstelle IUnknown des S3-Objekts aus. Die Methode S3::CreateInstance erhält einen Zeiger (punkOuter) auf die steuernde Schnittstelle IUnknown. Die Methode S3::CreateInstance verwendet den Operator new, um das S3-Objekt zu instanzieren. Während der Instanzierung wird der Konstruktor S3::S3 aufgerufen und erhält den Wert des Parameters punkOuter. Nachdem das S3-Objekt erstellt wurde, ruft die Methode S3::CreateInstance die Funktion CreateInstanceS1 auf, um das eingeschlossene S1-Objekt zu erstellen. Die Methode S3::CreateInstance gibt den Parameter pS3->m_punkOuter und die Schnittstellenkennzeichnung für die Schnittstelle IUnknown aus und erhält einen Zeiger auf die Schnittstelle IUnknown des S1-Objekts. Die Methode S3::CreateInstance speichert den ausgegebenen Zeiger in dem Datenglied S3::m_punkS1. Die Methode S3::CreateInstance ruft dann die Funktion CreateInstanceS2 auf, um ein S2-Objekt analog zu der Erstellung des S1-Objekts zu erstellen. Die Methode S3::CreateInstance ruft die Methode S3::QueryInterface auf, um einen Zeiger auf die Schnittstelle IUnknown abzurufen.

[0101] Die Methode S3::AddRef inkrementiert die Bezugszählung des S3-Objekts. Die Methode S3::Release dekrementiert die Bezugszählung. Wenn die Bezugszählung gleich null ist, löscht die Methode S3::Release das S3-Objekt.

[0102] Der Konstruktor S3::S3 initialisiert die Datenglieder m_A, m_B, m_punkOuter und m_refcount. Der Konstruktor S3::S3 erhält den Parameter punkOuter. Während der Instanzierung der Datenglieder m_A und m_B werden die Konstruktoren A::A und B::B aufgerufen und erhalten diesen Zeiger auf das S3-Objekt. Wenn der Wert des Parameters punkOuter gleich NULL ist, setzt der Konstruktor S3:S3 das Datenglied m_punkOuter auf den Wert für diesen Zeiger (der auf das neu instanzierte S3-Objekt zeigt). Wenn der Wert des Parameters punkOuter nicht-NULL ist, setzt der Konstruktor S3::S3 das Datenglied m_punkOuter auf den Wert des Parameters punkOuter. Das Datenglied m_punkOuter zeigt auf den Wert der steuernden Schnittstelle IUnknown des Aggregats, wenn das S3-Objekt eingeschlossen ist, und zeigt auf die steuernde Schnittstelle IUnknown des S3-Objekts, wenn das S3-Objekt nicht eingeschlossen ist. Der Konstruktor S3::S3 initialisiert auch das Datenglied m_refcount auf null.

[0103] Der Destruktor S3::~S3 ruft die Methode S1::Release auf, um die Bezugszählung des eingeschlossenen S1-Objekts zu dekrementieren. Weil die Bezugszählung während der Instanzierung des S1-Objekts auf eins gesetzt wurde, löscht die Methode S1::Release das S1-Objekt. Der Destruktor S3::~S3 dekrementiert die Bezugszählung des S2-Objekts auf analoge Weise.

[0104] Die Methoden der Schnittstellen A und B weisen ein analoges Verhalten auf wie die Methoden der Schnittstelle C. Die Schnittstellen A und C können also exponiert werden, wenn ein S3-Objekt eingeschlossen wird.

[0105] Fig. 5 ist ein Blockdiagramm, das das Datenstrukturlayout eines S3-Objekts zeigt. Das Datenstrukturlayout umfasst Instanzdaten **501**, Virtuellfunktionstabellen **502**, **503** und **504**, Methoden **505** bis **517** und Instanzen eines S1-Objekts **401**–**416** und eines S2-Objekts **519**. Die Instanzdaten **501** enthalten einen Zeiger auf die Virtuellfunktionstabelle für die steuernde Schnittstelle IUnknown, Datenglieder m_A und m_B, die Instanzen der Klasse A und B sind, das Datenglied m_punkOuter, das auf die Schnittstelle IUnknown des S3-Objekts zeigt, das Datenglied m_refcount, das die Bezugszählung für das S3-Objekt enthält, das Datenglied m_punkS1, das auf die Schnittstelle IUnknown des eingeschlossenen S2-Objekts zeigt, und ein Datenglied m_punkS2, da auf die Schnittstelle IUnknown des eingeschlossenen S2-Objekts **519** zeigt. Wenn das eingeschlossene S1-Objekt instanziert wird, wird sein Datenglied S2::m_punkOuter initialisiert, um auf die Schnittstelle IUnknown des S3-Objekts zu zeigen. Wenn entsprechend das eingeschlossene S2-Objekt instanziert wird, wird sein Datenglied S2::m_punkOuter initialisiert, um auf die Schnittstelle IUnknown des S3-Objekts zu zeigen.

[0106] Fig. 6A und Fig. 6B sind Blockdiagramme, die das Zusammenwirken zwischen einem einschließenden Objekt und einem eingeschlossenen Objekt zeigen. Fig. 6A ist ein Blockdiagramm, das ein Objekt der Klasse S1 zeigt, das nicht in einem anderen Objekt eingeschlossen ist. Das Objekt **601** der Klasse S1 enthält ein Datenglied m_punkOuter, das auf die Schnittselle IUnknown zeigt, sowie die Methoden **603**, **604**, **605** und **606**. Die Methode IUnknown::QueryInterface **603** gibt einen Zeiger auf die angeforderte Schnittstelle aus und inkrementiert die Bezugzählung. Die Methoden C::QueryInterface **605** und C::AddRef **606** rufen die entspre-

chenden Methoden der Schnittstelle IUnknown auf. Die Implementierung der Methoden der Klasse D (nicht gezeigt) sind analog zu denjenigen der Klasse C. Fig. 6B ist ein Blockdiagramm, das ein Objekt der Klasse C3 darstellt, das Objekte der Klasse S1 und S2 einschließt. Das S2-Objekt, das analog zu dem S1-Objekt ist, ist nicht gezeigt. Das Datenglied m_punkOuter **602** des Objekts **601** des Klasse S1 zeigt auf die Schnittstelle IUnknown des S3-Objekts **610**. Die Methode IUnknown::QueryInterface **613** gibt einen Zeiger auf jedes der exponierten Objekte aus und ruft die Methode IUnknown::QueryInterface **603** auf, auf die durch das Datenglied m_punkS1 **619** gezeigt wird, um einen Zeiger auf die C-Schnittstelle abzurufen. Das Datenglied m_punkOuter **612** zeigt auf die Schnittstelle IUnknown des S3-Objekts **610**. Die Methoden QueryInterface **615** und **617** der Objekte der Klasse A und B rufen die Methoden auf, auf die durch das Datenglied m_punkOuter **612** gezeigt wird.

[0107] In der oben beschriebenen Ausführungsform der vorliegenden Erfindung ruft die Methode QueryInterface der steuernden Schnittstelle IUnknown eines Aggregats die Methode QueryInterface der Schnittstelle IUnknown von eingeschlossenen Objekten auf, um Zeiger auf die exponierten Schnittstellen abzurufen. In einer alternativen Ausführungsform der vorliegenden Erfindung kann ein einschließendes Objekt Zeiger auf Schnittstellen von eingeschlossenen Objekten speichern, die das einschließende Objekt exponiert. Wenn also die Methode QueryInterface der steuernden Schnittstelle IUnknown aufgerufen wird, kann die Methode die gespeicherten Zeiger abrufen und ausgeben, nachdem die Methode AddRef der steuernden Schnittstelle IUnknown aufgerufen wurde, anstatt die Methode QueryInterface der Schnittstelle IUnknown des eingeschlossenen Objekts aufzurufen. Um diese alternative Ausführungsform zu implementieren, definiert ein einschließendes Objekt ein Datenglied für jeden gespeicherten Zeiger. Wenn das eingeschlossene Objekt instanziert wird (gewöhnlich während der Erstellung des einschließendes Objekts), wird die Methode QueryInterface der Schnittstelle IUnknown des eingeschlossenen Objekts aufgerufen, um einen Zeiger auf die exponierte Schnittstelle abzurufen. Vorzugsweise ist der abgerufene Zeiger kein gezählter Bezug, sodass das einschließende Objekt effektiv nur einen Zeiger (z. B. S3::m_punkS1) auf ein eingeschlossenes Objekt aufrechterhält. Das eingeschlossene Objekt kann dann durch einen einzigen Aufruf der Methode Release gelöscht werden. Nachdem also der Zeiger gespeichert wurde, wird die Methode Release der exponierten Schnittstelle aufgerufen, um die Bezugszählung für den gespeicherten Zeiger zu entfernen.

[0108] In der oben beschriebenen Ausführungsform der vorliegenden Erfindung enthält die Implementierung des Verfahrens QueryInterface der steuernden Schnittstelle IUnknown eine Schaltanweisung, die spezifiziert, welche Schnittstellen exponiert werden. Zum Beispiel enthält die Schaltanweisung der Methode S3::QueryInterface ein Case-Etikett für jede exponierte Schnittstelle A, B, C, F und die steuernde Schnittstelle IUnknown. Die exponierten Schnittstellen werden also während der Implementierung des einschließenden Objekts statisch definiert. In einer alternativen Ausführungsform kann die Methode QueryInterface der steuernden Schnittstelle IUnknown ohne spezifisches Wissen zu den externen Schnittstellen der eingeschlossenen Objekte implementiert werden. Wenn die Methode QueryInterface aufgefordert wird, einen Zeiger auf eine Schnittstelle auszugeben, die sie nicht implementiert, kann die Methode die Methode QueryInterface der Schnittstellen IUnknown der eingeschlossenen Objekte aufrufen, um einen Zeiger auf die identifizierte Schnittstelle abzurufen, wenn diese durch ein eingeschlossenes Objekt identifiziert wird. Die Code-Tabelle 5 enthält einen C++-Pseudocode für eine bevorzugte Implementierung der Methode QueryInterface der steuernden Schnittstelle IUnknown eines S3-Objekts, das diese alternative Ausführungsform implementiert. Zusätzlich zu der Ausgabe eines Zeigers auf jede externe Schnittstelle der eingeschlossenen Objekte kann die Methode QueryInterface der steuernden Schnittstelle IUnknown implementiert werden, um bestimmte externe Schnittstellen nicht zu exponieren, während alle anderen externen Schnittstellen exponiert werden.

Code-Tabelle 5

```
virtual boolean QueryInterface (REFIID iid, void **ppv)
{ ret = TRUE;
switch (iid)
{ case IID_A:
        *ppv = &m_A;
```

```
        m_punkOuter->AddRef();
        break;
case IID_B:
        *ppv = &m_B;
        m_punkOuter->AddRef();
        break;
case IID_IUnknown:
        *ppv = this;
        AddRef();
        break;
default:
        if (m_punkS1->QueryInterface (iid,ppv)) { return ret;};
        if (m_punkS2->QueryInterface (iid, ppv)) { return ret;};
}
return ret;
}
```

[0109] In den oben beschriebenen Ausführungsformen wurde keine Fehlerprüfung angewendet. Es ist jedoch vorzuziehen, dass verschiedene Typen von Fehlerprüfung vorgenommen werden, um sicherzustellen, dass ein Aggregat korrekt erstellt ist. Wenn zum Beispiel ein einschließendes Objekt versucht, ein nicht aggregierbares Objekt einzuschließen, dann sollte die Instanzierung des einschließendes Objekts fehlschlagen (z. B. gibt die Funktion CreateInstanceS1 ein Flag aus, das einen Fehler angibt).

[0110] In den oben beschriebenen Ausführungsformen kann ein aggregiertes Objekts selbst ein eingeschlossenes Objekt in einem einschließenden Objekt sein. Dieses verschachtelte Einschließen kann bis zu einer beliebigen Tiefe erfolgen. Alternativ hierzu kann ein aggregiertes Objekt implementiert werden, um nicht aggregierbar zu sein. Die Funktion CreateInstanceXX für die Klasse XX kann ein Flag ausgeben, das einen Fehler angibt, wenn der Parameter punkOuter nicht gleich null ist, d. h. wenn eine Aggregation gewünscht wird.

[0111] In der oben beschriebenen Ausführungsform wird ein Objekt für jede externe Schnittstelle eines aggregierbaren Objekts als ein Datenglied des aggregierbaren Objekts instanziert. In einer alternativen Ausführungsform werden die externen Schnittstellen durch das aggregierbare Objekt geerbt und nicht als Datenglieder des aggregierbaren Objekts implementiert. Die Code-Tabelle 6 enthält einen C++-Pseudocode für eine bevorzugte Klassendefinition S2 einer aggregierbaren Klasse mit den externen Schnittstellen C und D. Die Klasse S1 erbt die abstrakten Klassen IC und ID. Die Implementierungen der Schnittstellen IC und ID müssen keinen Zeiger auf die abgeleitete Klasse S1 speichern, um auf das Datenglied m_punkOuter zuzugreifen, sondern es ist eine spezielle nicht geerbte Implementierung der Schnittstelle IUnknown (IUnknownS1) erforderlich. Umgekehrt speichern die Implementierungen der Schnittstellen IC und ID wie in der Code-Tabelle 3 gezeigt den Zeiger auf die abgeleitete Klasse S1 in dem Datenglied m_pS1. Dem Fachmann sollte deutlich sein, dass auch andere Implementierungen unter Verwendung einer Vererbung von Schnittstellen möglich sind.

Code-Tabelle 6

```
class S1 : public IC, public ID
{ public:
        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_punkOuter->QueryInterface(iid, ppv);}

        virtual boolean AddRef()
        { m_punkOuter->AddRef();}

        virtual void Release()
        { m_punkOuter->AddRef();}

        // implementation of IC and ID

private:
        class IUnknownS1 : public IUnknown
        {
        public:
                IUnknownS1 (S1 *pS1)
                { m_pS1 = pS1;
                m_refcount = 0;}

                virtual boolean QueryInterface (REFIID iid, void **ppv)
                {ret = TRUE;
                switch (iid)
                {case IID_IUnknown:
                        *ppv = this;
                        AddRef();
```

```
                break;
        case IID_C:
                ppv = (IC)m_pS1;
                m_pS1->m_punkOuter->AddRef();
                break;
        case IID_D:
                *ppv = (ID *)m_pS1;
                m_pS1->m_punkOuter->AddRef();
                break;
        default:
                ret = FALSE;
        };
        }

        virtual void AddRef() {m_refcount++;}

        virtual void Release() {if (--m_refcount == 0) delete m_pS1;}

    private:
        int     m_refcount;
        S1      m_pS1;
    }

    friend IUnknownS1;
    IUnknownS1          m_IUnknownS1;

public:
    static void CreateInstance (IUnknown *punkOuter, IUnknown **ppunk)
    {   S1 *pS1 = new S1(punkOuter);
        pS1->QueryInterface(IID_Unknown, ppunk);
    }

private:
    void S1 (IUnknown *punkOuter) : m_IUnknownS1(this)
    {   if (punkOuter == NULL)
                m_punkOuter = &m_IUnknownS1;
        else
```

```
            m_punkOuter = punkOuter;
        }

        IUnknown    m_punkOuter;
}
```

Dynamische Aggregation

[0112] In einer bevorzugten Ausführungsform der dynamischen Aggregation werden Schnittstelleninstanzen kombiniert, indem diese zu einem beliebigen Zeitpunkt nach der Erstellung eines einschließendes Objekts zu dem einschließenden Objekt hinzugefügt werden. Auf diese Weise kann eine neue oder veränderte Schnittstelle mit einem bestehenden (Basis)-Objekt kombiniert werden, um das Verhalten des Basisobjekts zu verändern, nachdem der Code für das Basisobjekt kompiliert oder verknüpft wurde. Das heißt, obwohl das Verhalten des Basisobjekts (wie durch die Methoden des Basisobjekts implementiert) nach außen als verändert erscheint, wurden die das Verhalten des Basisobjekts implementierenden Methoden tatsächlich nicht geändert. Das Basisobjekt wird in dem einschließenden Objekt eingeschlossen, und die neuen oder veränderten Schnittstellen werden danach hinzugefügt.

[0113] Wenn eine externe Anforderung für den Zugriff auf eine bestimmte Schnittstelle gemacht wird, ist das einschließende Objekt dafür verantwortlich, zu bestimmen, welche Schnittstelle ausgegeben werden soll und wie die angeforderte Schnittstelle aufgerufen werden soll, wenn mehr als eine entsprechende Schnittstelle existieren. Wenn zum Beispiel drei IPrint-Schnittstellen in dem aggregierten Objekt existieren, bestimmt das einschließende Objekt, welche IPrint-Schnittstelle ausgegeben werden soll oder ob seine eigene IPrint-Schnittstelle ausgegeben werden soll, die weiß, wie eine Kombination aus den Methoden der anderen IPrint-Schnittstellen aufgerufen wird. Das einschließende Objekt kann diese Bestimmung entweder auf der Basis eines fixen oder eines spezifizierbaren Satzes von Kombinationsregeln vornehmen.

[0114] Diese Kombinationsregeln können verwendet werden, um das Standardverhalten eines eingeschlossenen Basisobjekts außer Kraft zu setzen, indem ein Zugriff auf eine neue Implementierung einer zuvor definierten Schnittstelle des eingeschlossenen Basisobjekts vorgesehen wird. Diese Regeln können also verwendet werden, um das Verhalten eines eingeschlossenen Basisobjekts zu verbessern, indem Fähigkeiten hinzugefügt werden, die zu Beginn nicht als Teil des eingeschlossenen Basisobjekts definiert waren. Sowohl das Außerkraftsetzen wie das Verbessern werden ermöglicht, indem eine neue oder veränderte Schnittstelle zu dem Basisobjekt hinzugefügt wird. Zusätzlich zu diesen Fähigkeiten kann ein einschließendes Standardobjekt ein Standardverhalten für die eingeschlossenen Objekte implementieren (Schnittstellen, die aufzurufende Methoden implementieren, wenn diese nicht durch die eingeschlossenen Objekte oder hinzugefügten Schnittstellen vorgesehen werden). Ein einschließendes Standardobjekt kann auch das steuernde (außer Kraft setzende) Verhalten für eine Methode implementieren, die gewöhnlich für alle eingeschlossenen Objekte vorhanden ist (etwa für das Drucken).

[0115] In einer bevorzugten Ausführungsform kann ein Objekt dynamisch modifiziert werden, indem Schnittstelleninstanzen (durch Objekte implementiert) gestattet werden, die während der Ausführung eines Client-Programms zu aggregieren sind. Das Computerprogramm, das ein Objekt instanziert, ist ein Client-Programm. Die Aggregation ist der Prozess zum Kombinieren der Fähigkeiten von mehreren distinkten Objekten durch das Einschließen ihrer entsprechenden Schnittstellen innerhalb eines einschließenden Objekts. Das einschließende Objekt ist dann dafür verantwortlich, den Zugriff auf alle Schnittstellen zu unterstützen, die es durch die Implementierung einer steuernden Schnittstelle IUnknown durch das einschließende Objekt exponieren möchte.

[0116] Eine statische Aggregation erfordert, dass das einschließende Objekt im voraus über Wissen zu den Schnittstellen (Objekten) verfügt, die es aggregieren möchte. Bei Verwendung einer statischen Aggregation entscheidet ein Programmierer im voraus, welche der Aggregatobjekt-Schnittstellen des einschließenden Objekts exponiert werden sollen, und implementiert dann die Methode QueryInterface der steuernden Schnittstelle IUnknown des einschließenden Objekts, um auf eine Aufforderung hin Zeiger auf diese exponierten Schnittstellen auszugeben. Die Methode QueryInterface der steuernden Schnittstelle IUnknown bewerkstelligt diese Aufgabe, indem sie Bezüge zu den entsprechenden Schnittstellen IUnknown der einzelnen eingeschlossenen Objekte aufrechterhält. (Diese Bezüge werden erstellt, wenn das einschließende Objekt eingeschlossene Objekte instanziert.) Wenn eine Anforderung für einen Bezug auf eine exponierte Schnittstelle von einem der ein-

geschlossenen Objekte empfangen wird, ruft die Methode QueryInterface der steuernden Schnittstelle IUnknown die entsprechende Schnittstelle IUnknown des eingeschlossenen Objekts auf, um auf die Anforderung zu antworten. Weil eingeschlossene Objekte über kein Wissen dazu verfügen, welche Schnittstellen das einschließende Objekt exponiert, werden alle durch ein eingeschlossenes Objekt empfangenen externen Anfragen zu dem einschließenden Objekt gegeben, um den Zugriff auf die Schnittstellen zu ermöglichen, die in den anderen aggregierten Objekten definiert sind.

[0117] Die vorliegende Erfindung unterstützt auch die dynamische Aggregation von Schnittstellen. In einer bevorzugten Ausführungsform sieht ein einschließendes Objekt eine Methode zum Registrieren von instanzierten Schnittstellen und zum späteren Abrufen von Bezügen auf diese ab. Wenn weiterhin eine Schnittstelle von dem aggregierten Objekt angefordert wird, sieht die vorliegende Erfindung eine Methode zum Modifizieren der Bestimmung durch, welche Schnittstelle(n) abzurufen sind und wie diese in Kombination abzurufen sind, wenn mehr als eine Instanz derselben Schnittstelle in dem aggregierten Objekt vorhanden ist.

[0118] In einer bevorzugten Ausführungsform wird eine dynamische Aggregation unter Verwendung eines Mehrtypen-Objekts implementiert. Ein Mehrtypen-Objekt ist ein Objekt, das Objekte verschiedenen Typs aggregieren kann. Nur Schnittstellen, die derart codiert wurden, dass sie aggregiert werden können, können in einem Mehrtypen-Objekt eingeschlossen werden. (Das heißt zum Beispiel, dass derartige Schnittstellen Schnittstellen- und Bezugszähl-Anforderungen an ein einschließendes Objekt weiterleiten können.) Ein Mehrtypen-Objekt sieht eine IMultitype-Schnittstelle zum Anfordern der Aggregation von bestimmten Schnittstellen oder Objekten und zum Hinzufügen von Regeln vor, um zu bestimmen, wie eine angeforderte Schnittstelle aufgerufen werden kann. Die Code-Tabelle 7 enthält einen Pseudocode für eine bevorzugte Definition der IMultitype-Schnittstelle.

Code-Tabelle 7

```
class IMultitype: public IUnknown {
        virtual HRESULT AddObject (ULONG list, BOOLEAN headoflist,
IUnknown *punkobj) = 0;
        virtual HRESULT AddInterface (REFIID iid, ULONG list, BOOLEAN headoflist,
                void **´ppv = 0;
        virtual HRESULT AddRule (REFIID iid, IRULE *prule) = 0;
        virtual Enum (ULONG i; REFIID iid, ULONG list, BOOLEAN headoflist;
                void **ppv) = 0;
}
```

[0119] Fig. 7A, Fig. 7B und Fig. 7C sind Blockdiagramme, die eine Sequenz zum Addieren von zwei Objekten zu einem Mehrtypen-Objekt zeigen. Fig. 7A ist ein Blockdiagramm einer Instanz eines Mehrtypen-Objekts. Das Objekt MTO **7A01** implementiert eine exponierte Schnittstelle, die IMultitype-Schnittstelle MT und eine steuernde Schnittstelle IUnknown. Wenn eine externe Schnittstelle zu dem Mehrtypen-Objekt hinzugefügt wird, wird das Mehrtypen-Objekt ein aggregiertes Objekt. Die Mehrtypen-Objekt-Implementierung enthält drei Listen **7A02**, **7A09** und **7A10** der Schnittstellen, die es zu der Aggregation hinzugefügt hat. Das Mehrtypen-Objekt verwendet diese Listen, um die verschiedenen Schnittstellen seiner eingeschlossenen Aggregat-Objekte über die steuernde Schnittstelle IUnknown des Mehrtypen-Objekts aufzurufen. Das Mehrtypen-Objekt enthält auch eine Liste mit Regeln **7A11** zum Zugreifen auf und Kombinieren von Schnittstellen aus den Schnittstellenlisten **7A02**, **7A09** und **7A10**.

[0120] Die Interaktion dieser unterschiedlichen Listen verleiht dem Mehrtypen-Objekt leistungsstarke Fähigkeiten. Die Liste der Regeln **7A11**, die fix sein kann oder unter Verwendung der Methode AddRule spezifiziert werden kann, spezifiziert die Interaktion und die Verwendung der unterschiedlichen Schnittstellenlisten für eine bestimmte Schnittstelle. Es können also Regeln zum Auswählen von anderen Regeln sowie Regeln zum Auswählen und Kombinieren von bestimmten Schnittstellen vorhanden sein. Drei unterschiedliche Schnittstellenlisten **7A02**, **7A09** und **7A10** sind vorgesehen, um Außerkraftsetzungs-, Verbesserungs-, Standard- und Steuerfähigkeiten zu unterstützen. Wenn eine Schnittstelle zu dem Mehrtypen-Objekt hinzugefügt wird, spezifiziert das Client-Programm, das das Aggregat erstellt, die Liste, die beim Hinzufügen der Schnittstelle verwendet werden soll. Die Liste **7A10** enthält eine normale Liste (normal), die Liste **7A09** enthält eine Standardliste (de-

fault) und die Liste **7A10** enthält eine Außerkraftsetzungsliste (override). Im Grunde ist die Außerkraftsetzungsliste dafür vorgesehen, Außerkraftsetzungs- und Steuerfähigkeiten zu implementieren, indem auf Schnittstellen gezeigt wird, auf die vor den Schnittstellen auf der normalen Liste zugegriffen werden muss. Die Standardliste ist dafür vorgesehen, auf die Schnittstellen zu zeigen, auf die nur dann zugegriffen wird, wenn die Außerkraftsetzungsliste und die Normalliste eine angeforderte Schnittstelle nicht enthalten. Die Interaktion dieser Listen wird in Verbindung mit der Beschreibung der IRules-Schnittstelle näher erläutert.

[0121] Fig. 7B ist ein Blockdiagramm, das das Mehrtypen-Objekt MTO nach dem Aggregieren der IBasic-Schnittstelle unter Verwendung des AddObject-Verfahrens darstellt. Die Methode AddObject fügt alle Schnittstellen eines spezifizierten Objekts zu einem Mehrtypen-Objekt hinzu. Das Aggregat-Objekt MTO **7B01** umfasst die Mehrtypen-Schnittstelle, die mit Bezug auf Fig. 7A erläutert wurde, sowie ein eingeschlossenes Tabellenobjekt S1 **7B04**. Das eingeschlossene Objekt S1 implementiert eine Instanz der externen Schnittstelle IBasic (B), eine Instanz der externen Schnittstelle IPrint (P) und eine Instanz von IUnknown. (Eine externe Schnittstelle ist eine Schnittstelle eines Objekts, das durch ein einschließendes Objekt exponiert wird. Eine interne Schnittstelle ist eine Schnittstelle eines Objekts, das nicht durch ein einschließendes Objekt exponiert wird.) Wenn das eingeschlossene Objekt S1 zu der normalen Liste des Mehrtypen-Objekts MTO hinzugefügt wird, enthält die normale Liste der aggregierten Schnittstellen **7B02** ein einziges Element **7B03**, das die IUnknown-Schnittstelle des eingeschlossenen Objekts S1 identifiziert. Die S1 IUnknown-Schnittstelle gibt auf eine Aufforderung hin Zeiger auf die externen Schnittstellen B und P aus. Weil S1 nicht aggregierbar ist, erhält es bei der Instanzierung einen Zeiger **7B05** auf das einschließendes Objekt MTO, das dann verwendet werden kann, um auf die anderen Schnittstellen zuzugreifen, die als Teil des Objekts MTO aggregiert sind.

[0122] Fig. 7C ist ein Blockdiagramm, das das Mehrtypen-Objekt MTO nach dem Hinzufügen der IDatabase-Schnittstelle unter Verwendung der Methode AddObject zeigt. Zu diesem Zeitpunkt umfasst das Aggregat-Objekt MTO **7C01** die IMultitype-Schnittstelle, die mit Bezug auf Fig. 7A erläutert wurde, ein eingeschlossenes Tabellenobjekt S1, das mit Bezug auf Fig. 7B erläutert wurde, und ein eingeschlossenes Datenbankobjekt S2 **7C07**, das Datenbankfähigkeiten implementiert. Das eingeschlossene Objekt S2 implementiert eine Instanz der externen Schnittstelle IDatabase (D) und eine Instanz von IUnknown. Wenn das eingeschlossene Objekt S2 unter Verwendung der Methode AddObject der IMultitype-Schnittstelle zu dem Mehrtypen-Objekt MTO hinzugefügt wird, enthält die normale Liste der aggregierten Schnittstellen **7C02** zwei Elemente **7C03** und **7C06**. Das Element **7C06** identifiziert die IUnknown-Schnittstelle des eingeschlossenen Objekts S2. Ähnlich wie S1 kann die S2 IUnknown-Schnittstelle einen Zeiger auf die externe Schnittstelle D ausgeben und enthält einen Zeiger **7C08** auf das einschließende Objekt MTO für einen Zugriff auf die anderen MTO-Schnittstellen.

[0123] Dem Fachmann sollte deutlich sein, dass viele alternative Ausführungsformen der Datenstrukturen verwendet werden können, um die hinzugefügten Schnittstellen und Objekte zu verfolgen. Zum Beispiel kann die Anzahl und die Art der verwendeten Listen geändert werden. Insbesondere kann nur eine Liste vorgesehen werden oder können die Außerkraftsetzungsliste oder die Standardliste optional vorgesehen werden. Weiterhin könnte die Anforderung aufgestellt werden, dass jedes Listenelement präzise nur auf die zu aggregierende Schnittstelle und nicht auf die IUnknown-Schnittstelle des Objekts zeigen darf, wenn ein gesamtes Objekt aggregiert wird (nur ein Mehrtypen-Objekt des AddInterface-Stils unterstützt wird). Alternativ hierzu können die Anforderung aufgestellt werden, dass jedes Listenelement auf die IUnknown-Schnittstelle des Objekts unabhängig davon zeigt, welche Schnittstelle zu der Aggregation hinzugefügt wird (nur ein Mehrtypen-Objekt des AddObjekt-Stils unterstützt wird). Außerdem könnten andere Listenimplementierungen einschließlich von verschiedenen sortierten Listen oder Hash-Tabellen von Schnittstellenkennzeichnungen verwendet werden.

Code-Tabelle 8

```
void CreateInstanceS1 (IUnknown *punkOuter, REFIID iid, void **ppv)
{       IUnknown *punk;
        S1::CreateInstance (punkOuter, &punk);
        punk->QueryInterface (iid, ppv);
        punk->Release ();
}

class IBasic: public IUnknown
{       virtual void File() = 0;
        virtual void Edit() = 0;
        virtual void Formula() = 0;
        virtual void Format() = 0;
        virtual void GetCell;
}

class IPrint: public IUnknown
{       virtual void Print (void **ppobj) = 0;
}

class S1: public IUnknown
{
public:
        static void CreateInstance(IUnknown *punkOuter, IUnknown **punk)
        {       S1 *pS1 = new S1(punkOuter);
                pS1->QueryInterface(IID_IUnknown, ppunk);
        }

private:
        void S1(IUnknown *punkOuter) : m_B(this), m_P(this)
        {       if (punkOuter == NULL)
                        m_punkOuter = this
                else
                        m_punkOuter = punkOuter;
                m_refcount = 0;
        }
```

```
class B: public IBasic
{
public:
        void B(S1 *pS1) {m_pS1 = pS1;}

        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_pS1->m_punkOuter->QueryInterface(iid, ppv);}

        virtual void AddRef()
        { m_pS1->m_punkOuter->AddRef();}

        virtual void Release()
        { m_pS1->m_punkOuter->Release();}

        // other methods of IBasic including File, Edit, Formula, Format, GetCell

private:
        S1      *m_pS1;
};
friend B;
B       m_B;

class P: public IPrint
{
public:
        void P(S1 *pS1) {m_pS1 = pS1;}

        virtual boolean QueryInterface (REFIID iid, void **ppv)
        { return m_pS1->m_punkOuter->QueryInterface(iid, ppv);}

        virtual void AddRef()
        { m_pS1->m_punkOuter->AddRef();}

        virtual void Release()
        { m_pS1->m_punkOuter->Release();}
```

```
        // other methods of IPrint including Print

    private:
        S1      *m_pS1;
    }
    friend P;
    P       m_P;

public:
    vitual boolean QueryInterface (REFIID iid, void **ppv)
    {       ret = TRUE;
        switch (iid) {
        case IID_B:
            *ppv = &m_B;
            m_punkOuter->AddRef();
            break;
        case IID_P:
            *ppv = &m_P;
            m_punkOuter->AddRef();
            break;
        case IID_IUnknown:
            *ppv = this;
            AddRef();
            break;
        default:
            ret = FALSE;
        }
        return ret;
    }

    virtual void AddRef(){ m_refcount++;}

    virtual void Release() {if (--m_refcount == 0= delete this;}

private:
    IUnknown    *m_punkOuter;
    int         m_refcount;
}
```

[0124] Die Code-Tabelle 8 enthält einen C++-Pseudocode für eine bevorzugte Klassendefinition des Objekts S1 in Fig. 7A–Fig. 7C, der in einem Aggregat (einem aggregierbaren Objekt) zusammen mit einer globalen Funktion eingeschlossen werden kann, die eine Instanz des S1-Objekts erstellt. Die Klasen IUnknown, IBasic und IPrint sind Schnittstellen, die die Methoden jeder Schnittstelle definieren, die S1 enthält. Die Klasse S1 implementiert die IUnknown-Schnittstelle, die IBasic-Schnittstelle und die IPrint-Schnittstelle. Die Schnittstellen IBasic und IPrint sind als externe Schnittstellen implementiert.

[0125] Fig. 8 ist ein Blockdiagramm des Datenstrukturlayouts einer Instanz eines Objekts der Klasse S1. Die Instanzstruktur **801** enthält die Datenglieder der Klasse S1 (m_B, m_P, m_punkOuter) und den Virtuellfunktionstabellen-Zeiger (S1::vfptr). Die Datenglieder m_B und m_P sind jeweils Instanzen von Objekten der Klassen B und P (die Klassenimplementierungen der Schnittstellen IBasic und IPrint sind). Die Datenglieder m_B und m_P sind Freunde der Klasse S1, sodass die Objekte m_B und m_P auf die privaten Glieder der Klasse S1 wie etwa m_punkOuter zugreifen können. Der Virtuellfunktionstabellen-Zeiger S1::vfptr zeigt auf die Virtuellfunktionstabelle **802**, der Virtuellfunktionstabellen-Zeiger S1::B::vfptr in dem Datenglied m_B zeigt auf die Virtuellfunktionstabelle **803** und der Virtuellfunktionstabellen-Zeiger S1::P::vfptr in dem Datenglied m_P zeigt auf die Virtuellfunktionstabelle **804**. Die Virtuellfunktionstabelle **802** enthält Zeiger auf die virtuellen Funktionen (Methoden), die für die steuernde IUnknown-Schnittstelle definiert sind, die Virtuellfunktionstabelle **803** enthält Zeiger auf die virtuellen Funktionen, die für die IBasic-Schnittstelle definiert sind, und die Virtuellfunktionstabelle **804** enthält Zeiger auf die virtuellen Funktionen, die für die IPrint-Schnittstelle definiert sind. Die Methoden **805** bis **809** sind die Funktionsglieder der Klasse S1. Die Methode **808** ist der Konstruktor für die Klasse S1. Die Methode **809** ist die Methode CreateInstance für die Klasse S1. Die Methoden **810** bis **818** sind die Funktionsglieder der Klasse B. Die Methode **813** ist der Konstruktor für die Klasse B. Die Methoden **819** bis **823** sind die Funktionsglieder der Klasse P. Die Methode **823** ist der Konstruktor für die Klasse P. Weil Fig. 8 eine Instanz eines S1-Objekts zeigt, das nicht Teil eines Aggregats ist, werden die Datenglieder S1::B::m_pS1, S1::P::m_pS1 und S1::m_punkOuter (Zeiger auf das einschließende Objekt) initialisiert, um auf das S1-Objekt selbst zu zeigen.

[0126] Das Objekt S1 entspricht gemäß der Definition durch die Code-Tabelle 8 den Anforderungen für ein aggregierbares Objekt, wie mit Bezug auf die statische Aggregation erläutert. Für die Zwecke der dynamischen Aggregation zeigt die Code-Tabelle 8, wie S1 automatisch mit seinem einschließenden Objekt kommunizieren kann, wenn es aggregiert ist. Die Code-Tabelle 8 zeigt außerdem, was die Funktion S1::QueryInterface ausgibt. Insbesondere wird bei der Erstellung eines S1-Objekts ein Zeiger auf die steuernde IUnknown-Schnittstelle eines einschließenden Mehrtypen-Objekts zu der Methode CreateInstance gegeben. Dieser Zeiger wird dann durch die QueryInterface-Methoden der externen Schnittstellen (IBasic und IPrint) von S1 verwendet, um Schnittstellenanfragen an das einschließende Mehrtypen-Objekt zu leiten. Wenn eine S1-Schnittstelle von dem einschließenden Mehrtypen-Objekt angefordert wird, ruft die Methode QueryInterface der steuernden IUnknown-Schnittstelle des einschließenden Mehrtypen-Objekts die Methode S1::QueryInterface auf, die einen Zeiger auf die entsprechende Instanz der Schnittstelle IBasic, die Schnittstelle IPrint oder die Schnittstelle IUnknown ausgibt, und inkrementiert die Bezugszählung des S1-Objekts entsprechend. (Der durch das einschließende Mehrtypen-Objekt zum Aufrufen von S1::QueryInterface verwendete Mechanismus wird weiter unten ausführlicher erläutert.) Dem Fachmann sollte deutlich sein, dass es viele Alternativen für das Geben eines Zeigers auf die steuernde IUnknown-Schnittstelle eines einschließenden Mehrtypen-Objekts an ein aggregierbares Objekt gibt. Anstatt zum Beispiel den Zeiger während der Erstellung zu geben, kann eine Methode eigens für das Bereitstellen dieses Zeigers definiert werden. Unter Verwendung dieser Ausführungsform kann ein Objekt nach dem Aggregieren später wieder deaggregiert werden, oder es kann ein Objekt später in ein anderes einschließendes Objekt aggregiert werden.

[0127] Um zu verstehen, wie die Code-Tabelle 8 mit einem Mehrtypen-Objekt wie in Fig. 7A–Fig. 7C gezeigt interagiert, ist es hilfreich, die Aufrufsequenz bei Client-Anfragen zu betrachten. Die Code-Tabelle 9 zeigt die Pseudocode-Sequenz von Aufrufen in Entsprechung zu Fig. 7C, wenn eine Client-Anwendung die IBasic-Schnittstelle anfordert, wenn der Client über einen Zeiger auf das Mehrtypen-Objekt MTO verfügt.

Code-Tabelle 9

```
MTO::QueryInterface (IID_IBasic, ppv)
which finds an aggregated object that supports the IBasic interface
S1::IUnknown:: QueryInterface (IID_IBasic, ppv)
which return pointer to the B interface
```

[0128] In dem ersten Aufruf (MTO::QueryInterface) bestimmt MTO aus seinen Listen aggregierter Schnittstel-

len, welche QueryInterface-Methode von welchem Objekt aufzurufen ist, und ruft diese dann in dem zweiten Aufruf (S1::IUnknown::QueryInterface) auf.

[0129] Die Code-Tabelle 10 zeigt, wie die Pseudocode-Sequenz von Aufrufen variiert, wenn die Client-Anwendung einen Zeiger auf eine der Schnittstellen des eingeschlossenen Objekts (wie etwa auf die IPrint-Schnittstelle von S1) anstatt einen Zeiger auf das einschließende Mehrtypen-Objekt aufweist.

Code-Tabelle 10

```
P::QueryInterface (IID_IBasic, ppv)
which forwards the call to the enclosing object
MTO:: IUnknown:: QueryInterface (IID_IBasic, ppv)
// m_punkOuter points to MTO:: IUnknown
which finds an aggregated object that supports the IBasic interface
S1::IUnknown::QueryInterface (IID_IBasic, ppv)
which returns pointer to the B interface
```

[0130] Die Code-Tabelle 10 zeigt, wie eine Aggregation automatisch Anfragen an das einschließende Objekt weiterleitet, um auf andere Schnittstellen in dem Aggregat zuzugreifen. In diesem Fall leitet die Funktion des eingeschlossenen Objekts die Anfrage an die QueryInterface-Methode des einschließenden Objekts (MTO) weiter. Dann funktioniert die Methode MTO::QueryInterface wie in der Code-Tabelle 9 angegeben.

[0131] Das S2-Objekt, das die IDatabase-Schnittstelle implementiert, ist analog zu dem oben beschriebenen S1-Objekt beschaffen.

Code-Tabelle 11

```
void CreateInstanceMTO (IUnknown *punkOuter, REFIID iid, void **ppv)
{       IUnknown *punk;
        MTO::CreateInstance (punkOuter, &punk);
        punk->Query Interface (iid, ppv);
        punk->Release ();
}
```

```
class IMultitype: public IUnknown
{       virtual HRESULT AddObject (ULONG list, BOOLEAN headoflist,
                                   IUnknown *punkobj) = 0;
        virtual HRESULT AddInterface (REFIID iid, ULONG list, BOOLEAN headoflist,
                                   void *pv) = 0;
        virtual HRESULT AddRule (REFIID iid, IRule *prule) = 0;
        virtual HRESULT Enum (ULONG i, REFIID iid, ULONG list, BOOLEAN headoflist,
                                   void **ppv) = 0;
}

class MTO: public IUnknown
{
public:
        static void CreateInstance(IUnknown *punkOuter, IUnknown **ppunk)
        {       MTO *pMTO = new MTO(punkOuter);
                pMTO->QueryInterface(IID_IUnknown, ppunk);}

private:
        void MTO(IUnknown *punkOuter); m_MT(this)
                {       if (punkOuter == NULL)
                                m_punkOuter = this;
                        else
                                m_punkOuter = punkOuter;}

        class MT:public IMultitype
        {
        public:
                void MTO(MTO *pMTO) {m_pMTO = pMTO}

                virtual boolean QueryInterface (REFIID iid, void **ppv)
                { return m_pMTO->m_punkOuter->QueryInterface(iid,ppv);}

                vitual void AddRef()
                { m_pMTO->m_punkOuter->AddRef();}

                virtual void Release()
                { m_pMTO->m_punkOuter->Release();}
```

```
virtual boolean AddObject (ULONG list, BOOLEAN headoflist,
                                IUnknown *punkobj)

{       item *pitem
        piten = new (item);
        pitem->iid = IID_Unknown;
        pitem->pobj = punkobj;
        pitem->pnext = null;
        pitem->pprev = null;
        switch (list) {
                case NORMAL_LIST:
                  // ... if headoflist == true, insert as first item in normal list,
                  // otherwise insert as last item;
                case DEFAULT_LIST:
                  // ... if headoflist == true, insert as first item in default list,
                  // otherwise insert as last item;
                case OVERRIDE_LIST:
                  // ... if headoflist == true, insert as first item in override list,
                  // otherwise insert as last item;
                default:
                  // ... insert at head of normal list;
        }
}

virtual boolean AddInterface (REFIID iid, ULONG list, BOOLEAN
                                headoflist, void *pv)

{       ...
        pitem->iid = iid;
        pitem->pobj = pv;
        // same code as for AddObject method except that list item points to
        // the particular interface and not to the IUnknown interface
        ...
}

\\ other methods of IMultitype ...
```

```
        private:
                MTO *m_pMTO;
        };
        friend MT;
        MT      m_MT;

public:
        virtual boolean QueryInterface(REFIID idd, void **ppv)
        {       boolean done = TRUE;
                item *pitem;

                switch (iid) {
                        case IID_IMultiType:
                                *ppv = &m_MT;
                                m_punkOuter->AddRef();
                                break;
                        case IID_IUnknown:
                                *ppv = this;
                                AddRef();
                                break;
                        default:
                                // search through the override list for the first matching interface
                        done = FALSE;
                        pitem = m_poverride_itemhead;
                        while ((done == FALSE) && (pitem->pnext ! = null)) {
                                switch (pitem->iid) {
                                  case IID_IUnknown:
                                    if (pitem->pobj->QueryInterface(iid, ppv) == TRUE)
                                        done = True;
                                    else pitem = pitem->pnext;
                                    break;
                                  default:
                                    if (pitem->iid == iid) {
                                        ppv = pitem->pobj;
                                        done = TRUE;
                                    }
```

```
                    else pitem = pitem->pnext;
                }}
            // search through the normal list for the first matching interface
            // if not yet found
            if (done == FALSE) {
                pitem = m_pnormail_itemhead;
                while ((done == FALSE) && (pitem->pnext != null)) {
                    ... // same code as for override list
                }
            // search through the default list for the first matching interface
            // if not yet found
            if (done == FALSE) {
                pitem = m_pdefault_itemhead;
                while ((done == FALSE) && (pitem->pnext != null)) {
                    ... // same code as for override list
                }
            break;
        }
        return done;
        }

        virtual void AddRef() { m_refcount++;}

        virtual void Release() {if (--m_refcount == 0) delete this;}

private:
        IUnknown    *m_punkOuter;
        int         m_refcount, m_occurrence;

        struct item {
            REFIID iid;
            void   *pobj;
            item   *pnext;
            item   *pprev };

        item   *m_pnormal_itemhead = null, *m_pnormal_itemtail = null,
            *m_pdefault_itemhead = null, *m_pdefault_itemtail = null,
*m_poverride_itemhead = null, *m_poverride_itemtail = null,
```

[0132] Die Code-Tabelle 11 ist ein C++-Pseudocode für eine bevorzugte Klassendefinition eines Mehrtypen-Objekts, das zum dynamischen Aggregieren von Schnittstellen verwendet werden kann. Die Klasse MTO implementiert eine Instanz der Mehrtypen-Schnittstelle und der steuernden IUnknown-Schnittstelle für das Mehrtypen-Objekt.

[0133] Die globale Funktion CreateInstanceMTO erstellt eine Instanz eines Objekts der Klasse MTO. Ein Client ruft diese Funktion auf, um ein Objekt der Klasse MTO zu instanzieren. Unter Verwendung dieser Funktion kann ein Client ein Objekt der Klasse MTO instanzieren, ohne beim Kompilieren oder in Laufzeit auf die MTO-Klassendefinition zugreifen zu müssen. Die Funktion CreateInstanceMTO erhält einen Zeiger auf die steuernde IUnknown-Schnittstelle (punkOuter), wenn das instanzierte MTO-Objekt in einem anderen Objekt aggregiert wird. Die Funktion ruft die Methode MTO::CreateInstance auf und gibt den Parameter punkOuter aus.

[0134] Der Konstruktor MTO::MTO instanziert die Datenglieder m_MT und m_punkOuter. Während der Instanzierung des Datenglieds m_MT der Klasse MT wird der Konstruktor MT::MT aufgerufen und erhält diesen Zeiger für das MTO-Objekt. (In C++ zeigt dieser Zeiger auf die Objektinstanz selbst.) Der Konstruktor MT::MT setzt dann eine lokale Variable, die zurück auf die MTO-Klasse zeigt. Der Konstruktor MTO::MTO erhält den Parameter punkOuter. Wenn der Wert von punkOuter gleich null ist, setzt der Konstruktor MTO::MTO das Datenglied m_punkOuter, um auf das neu instanzierte MTO-Objekt zu zeigen. Wenn dagegen punkOuter nicht gleich null ist, weil das Objekt zum Beispiel als Teil einer größeren Aggregation aggregiert ist, setzt der Konstruktor MTO::MTO das Datenglied m_punkOuter auf den Wert des Parameters punkOuter. Das heißt, das Datenglied m_punkOuter zeigt auf den Wert der steuernden IUnknown-Schnittstelle des Aggregats, wenn das MTO-Objekt eingeschlossen ist, und zeigt auf die IUnknown-Schnittstelle des MTO-Objekts, wenn das MTO-Objekt nicht eingeschlossen ist.

[0135] Die durch das MTO-Objekt implementierte IMultitype-Schnittstelle enthält die vier Methoden AddObject, AddInterface, AddRule und Enum. Die Methode AddObject ist dafür verantwortlich, ein Objekt hinzuzufügen, das in dem Mehrtypen-Objekt einzuschließen ist (alle Schnittstellen des Objekts werden verfügbar gemacht). Dies wird mit Bezug auf **Fig. 9** erläutert. Die Methode AddInterface ist dafür verantwortlich, eine einzelne Schnittstelleninstanz zu dem Mehrtypen-Objekt hinzuzufügen. Die Methode AddInterface wird in Verbindung mit der Methode AddObject erläutert. Die Methode AddRule ermöglicht es einer Client-Anwendung, eine Kombinationsregel zu spezifizieren, die verwendet wird, um zu bestimmen, welche Kombination von Objekten abgefragt werden sollen oder welche Schnittstellen auszugeben sind, wenn eine Client-Anwendung eine besondere Schnittstellenkennzeichnung anfordert. Dies wird in Verbindung mit den Regelobjekten näher erläutert. Die Methode Enum wird durch die Regelobjekte verwendet, um eine Aufzähung der verschiedenen durch das Mehrtypen-Objekt aufrechterhaltenen Schnittstellenlisten vorzusehen. Diese Methode wird in Verbindung mit den Regelobjekten näher erläutert.

[0136] **Fig. 9** ist ein Flussdiagramm der Methode AddObj der durch ein Mehrtypen-Objekt implementierten IMultitype-Schnittstelle. **Fig. 9** entspricht dem in der Code-Tabelle 11 gezeigten Code für AddObject. Die Methode AddObj wird durch eine Client-Anwendung verwendet, um dynamisch einen Mehrtypen-Objekt-Zugriff zu allen Schnittstellen eines Objekts hinzuzufügen. Diese Methode erhält eine Listenkennzeichnung, die angibt, zu welcher Liste das Objekt hinzugefügt werden soll, eine Angabe dazu, ob das Objekt am Anfang oder am Ende der spezifizierten Liste hinzugefügt werden soll, und einen Zeiger auf die IUnknown-Schnittstelle des zu aggregierenden Objekts. Diese Methode implementiert zusammen mit der Methode AddInterface Strukturen, um Informationen zu den durch ein Mehrtypen-Objekt eingeschlossenen Objekten und Schnittstellen aufrechtzuerhalten.

[0137] Eine typische Implementierung verwendet drei Listenstrukturen, die sich aus Elementen zusammensetzen, die jeweils auf eine Schnittstelle eines eingeschlossenen Objekts zeigen. Wenn die Methode AddObject aufgerufen wird, um das gesamte Objekt einzuschließen, wird ein neues Element zu der spezifizierten Listenstruktur hinzugefügt; das neue Element zeigt auf die IUnknown-Schnittstelle des eingeschlossenen Objekts. Diese IUnknown-Schnittstelle kann dann verwendet werden, um auf die Komponentenschnittstellen des eingeschlossenen Objekts zuzugreifen. Wenn dagegen die Methode AddInterface aufgerufen wird, um eine einzelne Schnittstelle eines Objekts einzuschließen, wird ein neues Listenelement zu der spezifizierten Listenstruktur hinzugefügt; das neue Element zeigt auf die einzelne Schnittstelle, damit direkt auf diese zugegriffen werden kann. In einer typischen Implementierung wird jedes Listenelement durch eine Schnittstellenkennzeichnung indiziert, zeigt auf eine Schnittstelle eines eingeschlossenen Objekts und zeigt auf das nächste Element in der Liste. Weil Clients entweder am Anfang oder am Ende einer Liste hinzugefügt werden können, kann eine doppelt verknüpfte Liste verwendet werden, um die Effizienz zu erhöhen.

[0138] Beim Aufrufen der Methode spezifiziert ein Client, zu welcher Liste die Anwendung die spezifizierte Schnittstelle bzw. das spezifizierte Objekt hinzufügen möchte. Eine Normalliste („normal") wird verwendet, wenn die Client-Anwendung einfach Schnittstellen oder Objekte zu der Aggregation hinzufügen möchte. Eine Außerkraftsetzungsliste („override") oder eine Standardliste („default") werden verwendet, wenn die Client-Anwendung Schnittstellen hinzufügen möchte, deren Methoden zu einem anderen Zeitpunkt als diejenigen in der normalen Liste aufgerufen werden. In einer typischen Implementierung gibt die Methode QueryInterface der steuernden IUnknown-Schnittstelle nach einer Anforderung für eine bestimmte Schnittstelle die angeforderte Schnittstelle aus, wobei sie zuerst die Außerkraftsetzungsliste durchsucht, dann die Normalliste durchsucht und dann schließlich die Standardliste durchsucht. Dem Fachmann sollte deutlich sein, dass viele andere Implementierungen und Suchstrategien möglich sind, wobei die Anzahl der Listenstrukturen, die Suchorder oder die Bestimmung von Entsprechungen zu einer angeforderten Schnittstelle verändert werden können. In einer bevorzugten Ausführungsform kann die Client-Anwendung die Bestimmungsregeln ändern.

[0139] Die Schritte von Fig. 9 zeigen, wie ein Element zu der spezifizierten Liste hinzugefügt wird. In Schritt **901** weist die Methode ein neues Listenelement hinzu, und in Schritt **902** initialisiert die Methode das Element, sodass es auf die IUnknown-Schnittstelle des Objekts mit den Schnittstellen zeigt, die die Client-Anwendung aggregieren möchte, und die Schnittstellenkennzeichnung des Elements (zur Angabe der IUnknown-Schnittstelle) enthält. In Schritt **903** bestimmt die Methode, ob die Normaliste spezifiziert wurde. Wenn dies der Fall ist, fährt die Methode mit Schritt **904** ort, ansonsten fährt sie mit Schritt **907** fort. In Schritt **904** bestimmt die Methode, ob die Client-Anwendung ein Element am Anfang der Normalliste einfügen möchte. Wenn dies der Fall ist, fährt die Methode in Schritt **905** fort, ansonsten fährt sie in Schritt **906** fort. In Schritt **905** fügt die Methode das initialisierte Element am Anfang der Normalliste ein und fährt fort. In Schritt **906** fügt die Methode das initialisierte Element am Ende der Normalliste ein und wird beendet. Die Schritte **907** bis **914** operieren analog auf den Außerkraftsetzungs- und Standardlisten.

[0140] Die Methode AddInterface der IMultitype-Schnittstelle arbeitet ähnlich wie die Methode AddObject. Der Hauptunterschied besteht darin, dass anstatt eines hinzugefügten Listenelements, das auf die spezifizierte IUnknown-Schnittstelle des einzuschließenden Objekts zeigt, das hinzugefügte Listenelement auf eine spezifizierte Schnittstelle zeigt und die erhaltene Schnittstellenkennzeichnung angibt. Auf diese Weise kann eine einzelne Schnittstelle eines Objekts aggregiert werden, ohne dass andere Schnittstellen exponiert werden.

[0141] Wie in der Code-Tabelle 11 angegeben, leiten die Methoden QueryInterface, AddRef und Release der IMultitype-Schnittstelle (der geerbten IUnknown-Schnittstelle) Anforderungen an die IUnknown-Schnittstelle des Elterobjekts weiter, das diese IMultitype-Schnittstelle (MTO) implementiert.

[0142] Die durch das MTO-Objekt implementierte steuernde IUnknown-Schnittstelle enthält die Methoden QueryInterface, AddRef und Release. Die Methoden AddRef und Release implementieren die Bezugszählung des Mehrtypen-Objekts. Wenn die Bezugszählung gleich null ist, wird das MTO-Objekt gelöscht.

[0143] Fig. 10 ist ein Flussdiagramm der Methode QueryInterface der steuernden IUnknown-Schnittstelle für ein Mehrtypen-Objekt. Fig. 10 entspricht dem in der Code-Tabelle 11 gezeigten Code für QueryInterface. Die Methode QueryInterface lokalisiert eine angeforderte Schnittstelle unter Verwendung des Wissens zu seiner eigenen Implementierung und von Informationen aus den aggregierten Schnittstellenlisten. Die Methode nimmt einen Eingabepeparameter, der die angeforderte Schnittstellenkennzeichnung ist, und gibt einen Zeiger auf die angeforderte Schnittstelle aus. In den Schritten **1001**–**1004** bestimmt die Methode, ob die angeforderte Schnittstelle die durch das Mehrtyp-Objekt selbst implementierte ist. Wenn dies nicht der Fall ist, sucht die Methode in den Schritten **1005**–**1014** alle eingeschlossenen Objekte oder Schnittstellen, bis es die angeforderte Schnittstelle findet.

[0144] In Schritt **1001** bestimmt die Methode, ob die angeforderte Schnittstellenkennzeichnung äquivalent zu IID_IMultitype ist. Wenn dies der Fall ist, fährt sie mit Schritt **1002** fort, ansonsten fährt sie mit Schritt **1003** fort. In Schritt **1002** setzt die Methode den Ausgabeparameter, sodass er auf die Instanz von IMultitype zeigt, die durch das Objekt MTO implementiert wird, und wird beendet. In Schritt **1003** bestimmt die Methode, ob die angeforderte Schnittstellenkennzeichnung äquivalent zu IID_IUnknown ist. Wenn dies der Fall ist, fährt sie mit Schritt **1004** fort, ansonsten fährt sie mit Schritt **1005** fort. In Schritt **1004** setzt die Methode den Ausgabeparameter auf diesen Zeiger, der die Instanz der durch das Mehrtypen-Objekt implementierten Schnittstelle IUnknown ist, und wird beendet.

[0145] In den Schritten **1005** bis **1014** durchläuft die Methode die drei Listen und sucht nach dem ersten Listenelement, das auf eine Schnittstelle zeigt, die der angeforderten Schnittstelle entspricht. Wenn diese Schnitt-

stelle gefunden wird, wird sie in dem Parameter ppv ausgegeben, und die Methode kehrt zu einem erfolgreichen Status zurück. Dem Fachmann sollte deutlich sein, dass diese Implementierung nur ein Beispiel für viele Typen von Suchen ist, die verwendet werden können. In Schritt **1005** wird ein temporärer Listenindikator auf die nächste Liste aus dem Satz der durch ein Mehrtypen-Objekt implementierten Listen gesetzt. In einer bevorzugten Ausführungsform enthält dieser Satz von Listen eine Außerkraftsetzungs-, eine Normal- und eine Standardliste. In Schritt **1006** setzt die Methode eine temporäre Variable pitem, sodass diese auf den Anfang der aktuellen Liste zeigt. In Schritt **1007** bestimmt die Methode, ob sie alle Elemente der aktuellen Liste durchlaufen und immer noch keine entsprechende Schnittstelle gefunden hat. Wenn die Methode das Ende der aktuellen Liste erreicht hat, fährt die Methode mit Schritt **1008** fort, ansonsten fährt sie mit Schritt **1009** fort. Wenn die Methode in Schritt **1008** bestimmt, dass weitere Listen vorhanden sind, die durchsucht werden können, kehrt die Methode zu Schritt **1005** zurück, um mit dem Durchsuchen einer neuen Liste zu beginnen. Ansonsten tritt die Methode in einen nicht-erfolgreichen Status ein, weil keine entsprechende Schnittstelle gefunden wurde. In Schritt **1009** bestimmt die Methode, ob das aktuelle Listenelement, auf das durch die temporäre Variable pitem gezeigt wurde, auf eine IUnknown-Schnittstelle zeigt. Wenn dies der Fall ist, fährt die Methode mit Schritt **1010** fort, ansonsten fährt sie mit Schritt **1013** fort. (Wenn das aktuelle Listenelement auf eine IUnknown-Schnittstelle zeigt, muss das Objekt für diese Schnittstelle nicht weiter nach einer entsprechenden Schnittstelle abgefragt werden.) In Schritt **1010** ruft die Methode die Methode QueryInterface der IUnknown-Schnittstelle auf, auf die durch das aktuelle Listenelement gezeigt wird. In Schritt **1011** bestimmt die Methode, ob die angeforderte Schnittstellenkennzeichnung gefunden wurde. Wenn nicht, fährt die Methode mit Schritt **1012** fort, ansonsten kehrt sie zurück. Wenn die angeforderte Schnittstellenkennzeichnung gefunden wurde, hat der Aufruf von QueryInterface bereits den Ausgabeparameter ppv so gesetzt, dass er auf die angeforderte Schnittstelle zeigt. In diesem Fall gibt die Methode einen erfolgreichen Status aus. Ansonsten inkrementiert die Methode in Schritt **1012** den Zeiger für das aktuelle Listenelement (pitem), sodass es auf das nächste Element in der Liste zeigt, und fährt in Schritt **1007** mit der Suche in der aktuellen Listenstruktur fort. In Schritt **1013** vergleicht das Verfahren die angeforderte Schnittstellenkennzeichnung mit dem Schnittstellenkennzeichnungsfeld des aktuellen Listenelements. Wenn diese gleich sind, fährt die Methode mit Schritt **1014** fort, während sie ansonsten in Schritt **1012** die Suche fortsetzt. In Schritt **1014** setzt die Methode den Ausgabeparameter ppv, sodass er auf die angeforderte Schnittstelle zeigt (auf die durch das aktuelle Listenelement gezeigt wird), und tritt in einen erfolgreichen Status ein.

[0146] Fig. 11 ist ein Blockdiagramm, das das Datenstrukturlayout eines Mehrtypen-Objekts in Entsprechung zu Fig. 7C zeigt, nachdem die Schnittstellen IBasic, IPrint und IDatabase dynamisch unter Verwendung der Methode AddObject aggregiert wurden. Das Datenstrukturlayout umfasst Instanzdaten **1101**, Virtuellfunktionstabellen **1104** und **1105**, Methoden **1107** bis **1115** und Instanzen von aktuell eingeschlossenen (aggregierten) Objekten. Die Instanzdaten **1101** enthalten einen Zeiger auf die Virtuellfunktionstabelle für die steuernde IUnknown-Schnittstelle, das Datenglied m_MT, das eine Instanz der Klasse MT ist (eine Implementierung der IMultitype-Schnittstelle), das Datenglied m_punkOuter, das auf die IUnknown-Schnittstelle des MTO-Objekts zeigt, und das Datenglied m_pnormal_itemhead, das auf den Anfang der normalen Liste der aktuell hinzugefügten Schnittstellen zeigt. Wie dargestellt, enthält die Liste aktuell zwei Elemente. Das Listenelement **1102** zeigt auf eine Instanz eines S1-Objekts **801**–**823** (wie in Fig. 8 definiert), und das Listenelement **1103** zeigt auf eine Instanz eines S2-Objekts **1106**. Obwohl nicht gezeigt, kann das Mehrtypen-Objekt auch Zeiger auf eine default-Liste und eine override-Liste enthalten, die leer sind.

[0147] Wenn das eingeschlossene S1-Objekt instanziert wird, wird sein Datenglied S1::m_punkOuter initialisiert, um auf die IUnknown-Schnittstelle des MTO-Objekts zu zeigen. Wenn entsprechend das eingeschlossene S2-Objekt instanziert wird, wird sein Datenglied S2::m_punkOuter instanziert, um auf die IUnknown-Schnittstelle des MTO-Objekts zu zeigen. Diese Aufgabe wird wie zuvor erläutert bewerkstelligt, indem ein Zeiger auf das MTO in der Funktion CreateInstanceXX erhalten wird (wobei XX für S1 oder S2 steht).

Code-Tabelle 12

```
MTO *pMTO;
IMultType *my_pMT;
IUnknown *pSpreadSheet, *pDataBase;
CreateInstanceMTO (NULL, IID_IUnknown, pMTO);
CreateInstanceS1 (pMTO, IID_IUnknown, pSpreadSheet);
CreateInstanceS2 (pMTO, IID_IUnknown, pDataBase);
pMTO->QueryInterface (IID_Multitype, my_pMT);
my_pMT->AddObject (IID_ISpreadSheet, NORMAL_LIST, true, pSpreadSheet)
my_pMT->AddObject (IID_IDataBase, NORMAL_LIST, true, pDataBase);
```

```
...
// Some later time, some other client can invoke the database capability of a spreadsheet object pSpreadSheet
-> QueryInterface(IID_IDataBase, ppv);
ppv->Data();
```

[0148] Die Code-Tabelle 12 enthält einen Pseudocode für die bevorzugte Liste von Schritten zum dynamischen Aggregieren eines Satzes von Schnittstellen. Dieses Beispiel entspricht dem Hinzufügen der Schnittstellen für IBasic, IPrint und IDatabase wie in Fig. 7C gezeigt unter Verwendung der Methode AddObject, um alle Schnittstellen eines Objekts hinzuzufügen. Zuerst wird ein neues Mehrtypen-Objekt unter Verwendung der Funktion CreateInstanceMTO hinzugefügt. Unter Verwendung dieser Funktion wird die IUnknown-Schnittstelle für das neu instanzierte Mehrtypen-Objekt abgerufen. Diese Schnittstelle wird zu den S1- und S2-Objekten gegeben, wenn diese instanziert werden, damit sie auf das einschließende Mehrtypen-Objekt zugreifen können. Zu einem bestimmten Zeitpunkt wird eine Instanz des S1-Objekts erstellt und erhält den Zeiger auf das Mehrtypen-Objekt. Weiterhin wird eine Instanz des S2-Datenbankobjekts erstellt und erhält den Zeiger auf das Mehrtypen-Objekt. Nach der Erstellung dieser Instanzen von S1 und S2 wird die IMultitype-Schnittstelle von dem Mehrtypen-Objekt abgerufen. Diese Schnittstelle wird verwendet, um die Tabellen- und Datenbankschnittstellen dynamisch zu dem Mehrtypen-Objekt hinzuzufügen. Dann wird die Methode AddObject der IMultitype-Schnittstelle des Mehrtypen-Objekts aufgeruen, um die S1- und S2-Objekte in das Mehrtypen-Objekt zu aggregieren. Sobald diese Objekte in das Mehrtypen-Objekt eingeschlossen wurden, kann ein Client eine Schnittstelle des Mehrtypen-Objekts oder eines seiner eingeschlossenen Objekte verwenden, um auf beliebige andere Schnittstellen in dem aggregierten Objekt zuzugreifen. Die Code-Tabelle 12 zeigt ein Beispiel für das Abrufen der Datenbankschnittstelle unter Verwendung des grundlegenden Tabellenobjekts (S1) sowie für das folgende Aufrufen der Datenmethode dieser IDatabase-Schnittstelle.

Code-Tabelle 13

```
class IMultitype: public IUnknown
{       virtual HRESULT AddObject (ULONG list, BOOLEAN headoflist,
                                   IUnknown *punkobj) = 0;
        virtual HRESULT AddInterface (REFIID iid, ULONG list, BOOLEAN headoflist,
                                   void *pv) = 0;
        virtual HRESULT AddRule (REFIID iid, IRule *prule) = 0;
        virtual HRESULT Enum (ULONG i, REFIID iid, ULONG list, BOOLEAN headoflist,
                                   void **ppv) = 0;
}

class IRule: public IUnknown
{       virtual HRESULT Init (IMultitype *pMTO) = 0;
}

class MTO: public IUnknown
{
public:
```

```
        static void CreateInstance(IUnknown *punkOuter, IUnknown **ppunk)
        {       MTO *pMTO = new MTO(punkOuter);
                pMTO->QueryInterface(IID_IUnknown, ppunk);}

private:
        void MTO(IUnknown *punkOuter) : m_MT(this)
                {       if (punkOuter == NULL)
                                m_punkOuter = this;
                        else
                                m_punkOuter = punkOuter;
                        pQI = new(MyQI);      // make a QueryInterface default rule
                        pQI->QueryInterface(IID_IRule, prule);
                        m_MT->AddRule(IID_IUnknown, prule);
                }

        class MT: public IMultitype
        {
        public:
                void MT(MTO *pMTO) {m_pMTO = pMTO}

                virtual HRESULT QueryInterface (REFIID iid, void **ppv)
                { return m_pMTO->m_punkOuter->QueryInterface(iid, ppv);}

                virtual void AddRef()
                { m_pMTO->m_punkOuter->Release();}

                virtual HRESULT AddObject (ULONG list, BOOLEAN headoflist,
                                                        IUnknown *punkobj)
                {       item *pitem;
                        pitem = new (item);
                        pitem->iid = IID_IUnknown;
                        pitem->pobj = punkobj;
                        pitem->pnext = null
                        pitem->pprev = null;
                        switch (list) {
                          case NORMAL_LIST:
                                // ... if headoflist == true, insert as first ietem in normal list,
```

```
                    // otherwise insert as last item;
                case DEFAULT_LIST:
                    // ... if headoflist == true, insert as first ietem in default list,
                    // otherwise insert as last item;
                case OVERRIDE_LIST:
                    // ... if headoflist == true, insert as first ietem in override list,
                    // otherwise insert as last item;
                default:
                    // ... insert at head of normal ist;
                }
        }

        virtual HRESULT AddInterface (REFIID iid, ULONG list, BOOLEAN headoflist,
                                                    void *pv)
        {       ...
                pitem->iid = iid;
                pitem->pobj = pv;
                // same code as for AddObject method except that list item points to
                // the particular interface and not to the IUnknown interface
                ...
        }

        virtual HRESULT AddRule (REFIID iid, URule *prule)
        // this method adds a rule object to the list of rules in the multitype object
        {       ruleitem *pruleitem;

        pruleitem = new(ruleitem);
        pruleitem->iid = iid;
        pruleitem->prule = prule;
        pruleitem->pnext = null;
        pruleitem->pprev = null;
        // insert in the rule list – one way to do this is to insert the rule as last item
        // so QueryInterface is always first
        ...
        prule->Init(m_pMTO);          // tell the rule about the multitype object
}
virtual HRESULT Enum (ULONG i, REFIID iid, ULONG list, BOOLEAN headoflist,
```

```
                    void **ppv)
// this method returns the i'th occurrence of the element corresponding to the
// specified iid in the specified list beginning with the head or tail of the list
{       int     counter = 0;
        item    *pitem = null;
        ruleitem        *pruleitem = null;
        boolean         done = FALSE;

        switch (list) {
         case NORMAL_LIST:
                if (headoflist) {
                 pitem = m_pMTO->m_pnormal_itemhead;
                 while ((!done) && (pitem ! = null)) {
                        // for each item in the list, compare either the pobj field if the
                        // item points to an interface that has been added or query
                        // interface the object to see if the interface exists if the entire
                        // object has been added
                        switch (pitem->iid) [
                         case IID_IUnknown:
                                if ((pitem->pobj->QueryInterface(iid, ppv) == TRUE)

                                        (counter == i)) done = TRUE;
                                else pitem=pitem->pnext;
                                break;
                         default
                                if ((pitem->iid == iid) && (counter == i)) {
                                        done = TRUE;
                                        ppv = pitem->pobj;
                                } else pitem = pitem->pnext;
                                break;
                                }};
                        else    {       // ... start from tail an work backwards using
                                        // pitem = pitem->pprev;
                        };
                        break;
                case DEFAULT_LIST:
                        // works the same as the normal list except uses the default list
```

```
                    break;
                case OVERRIDE_LIST:
                    // workst the same as the normal list except uses the override
                    list
                    break;
                case RULE_LIST:
                    if (headoflist) {
                        pruleitem = m_pMTO->m_prule_itemhead;
                        done = FALSE;
                        while ((!done) && (pruleitem ! = null)) {
                            if pruleitem->iid == iid {
                                ppv = pruleitem->prule;
                                done = TRUE;
                            } else pruleitem = pruleitem->pnext; }
                    } else {        //... start from tail and work backwards using
                            // pruleitem = pruleitem->pprev;
                    };
                    break;
                default:
                    // use the same steps as for the normal list starting from the
                    // head
                }
            }

        private:
            MTO   *m_pMTO;
        };      // end of class definition for MT object
        friend MT;
        MT      m_MT;

public:
        virtual HRESULT QueryInterface(REFIID iid, void **ppv)
        {       IRule *prule;
                ruleitem *pruleitem;
                boolean done = TRUE, foundrule = FALSE;
                switch (iid) {
                  case IID_IMultiType:
```

```
        *ppv = %m_MT;
        m_punkOuter->AddRef();
        break;
case IID_IUnknown:
        *ppv = this;
        AddRef();
        break;
default:
        done = FALSE;
        // search through the rule list for the first matching IUnknown interface
        // and invoke it as the selection rule to access the combining rule for
        // the requested interface
        pruleitem = m_prule_itemhead;
        while ((foundrule == FALSE) && (pruleitem ! = null)) {
                if pruleitem-> == IID_IUnknown {
                        prule = pruleitem->prule;
                        foundrule = TRUE;
                } else pruleitem = pruleitem->pnext;
        }
        if (foundrule) {        // get and call its QueryInterface method
                prule-> QueryInterface(IID_IUnknown, pselect);
                done = pselect-> QueryInterface(iid, ppv);
        }
        else {  // find the combining rule on the rule list and return it if it exists
                // this code ist the default selection rule if one ist not provided
                pruleitem = m_prule_itemhead;
                foundrule = FALSE;
                while ((foundrule == FALSE) && (pruleitem != null)) {
                        if pruleitem->iid == iid {
                                prule = pruleitem->prule;
                                foundrule = TRUE;
                        } else pruleitem = pruleitem->pnext;
                }
                if (foundrule) {        // find the requested interface and return it
                        prule->QueryInterface(iid, ppv);
                else // no combining rule exists so just return 1st interface found
                        done = FALSE;
```

```
                        // search through all lists for the matching interface
                        if ((m_MT->Enum(1, iid, OVERRIDE_LIST, true, ppv))
                                == FALSE) {done = TRUE };
                        else if ((m_MT->Enum(1, iid, NORMAL_LIST, true, ppv))
                                = = TRUE) {done = TRUE};
                        else if ((m_MT->Enum(1, iid, DEFAULT_LIST, true,
                                ppv)) == TRUE) {done = TRUE};
                }
                break;
        }
        return done;
}

virtual void AddRef() { m_refcount++;}

virtual void Release() {if (--m_refcount == 0= delete this;}

private:
        IUnknown        *m_punkOuter;
        int             m_refcount;

        struct item {
                REFIID iid;
                void    *pobj;
                item    *pnext;
                item    *pprev };

        item    *m_pnormal_itemhead = null, *m_pnormal_itemtail = null,
                *m_pdefault_itemhead = null, *m_pdefault_itemtail = null,
                *m_poverride_itemhead = null, *m_poverride_itemtail = null,

        struct ruleitem {
                REFIID iid;
                IRule   *prule
                ruleitem *pnext
                ruleitem *pprev };
```

```
        ruleitem *m_prule_itemhead = null, *mprule_itemtail = null;

};      // end of class definition for MTO object

class MyQI: public IUnknown
{
private:
        void MyQI: m_R(this) {any other initialization code }

        class R: public IRule
        {
        public:
                void R(MyQI *pMyQI) {m_pMyQI = pMyQI) }

                // .. IUnknown methods are also implemented here which call the controlling
                // IUnknown methods for the enclosing object (public methods shown below)

                virtual HRESULT Init (IMultitype*pMTO) {m_pMyQI->m_pMTO = pMTO }
        private:
                MyQI   *m_pMyQI;
        };      // end of class R definition

        friend R
        R       m_R;

        int             m_refcount;
        IMultitype      *m_pMTO;

public:
        virtual HRESULT QueryInterface (REFIID iid, void **ppv)
        {       IRule *prule;
                boolean done = TRUE, foundrule = FALSE;

                switch (iid) {
                        case IID_IRule:
                                *ppv = & m_R;
                                AddRef();
```

```
                    break;
                case IID_IUnknown:
                    *ppv = this;
                    AddRef();
                    break;
                default: // this is the same as code for our default rule for IUnknown
                    // that knows how to search the lists of an MTO object

                foundrule = m_pMTO->Enum(1, iid, RULE_LIST, true, prule);
                    if (foundrule)   // return the combining rule for the requensted IID
                    {prule->QueryInterface(iid, ppv);}
                else {  // no combining rule exists so just return 1st interface found
                    done = FALSE;
                    // search through all lists for the first matching interface
                    if ((m_pMTO->Enum(1, iid, OVERRIDE_LIST, true, ppv))
                        == TRUE) { done = TRUE };
                    else if ((m_pMTO->Enum(1, iid, NORMAL_LIST, true,
                        ppv)) == TRUE) { done = TRUE };
                    else if ((m_pMTO->Enum(1, iid, DEFAULT_LIST, true,
                        ppv)) == TRUE) { done = TRUE };
                };
                break;
            };
            return done;
        }

        virtual void AddRef() { m_refcount++;}

        virtual void Release() { if (--m_refcount == 0) delete this;}
};
```

[0149] Die Code-Tabelle 13 enthält einen C++-Pseudocode für eine bevorzugte Klassendefinition für ein Mehrtypen-Objekt, das verbessert wurde, um eine Liste von Regelobjekten zu unterstützen. Es sei darauf nochmals hingewiesen, dass diese Regelobjekte entweder Kombinationsregeln zum Kombinieren von angeforderten Schnittstellen eines Mehrtypen-Objekts oder eine Auswahlregel zum Auswählen eines Regelobjekts (einschließlich einer Kombinationsregel) aus der Liste der Regelobjekte enthalten können. Jedes Regelobjekt implementiert eine Instanz der IRules-Schnittstelle, um das Regelobjekt in ein Mehrtypen-Objekt einzubinden. Jedes Regelobjekt implementiert weiterhin eine Instanz der Schnittstelle, für die das Regelobjekt eine Kombinationsregel vorsieht. Um zum Beispiel alle eingeschlossenen aggregierten Objekte zu drucken, kann ein Regelobjekt vorgesehen werden, dass eine Kombinationsregel für die IPrint-Schnittstelle implementiert. Ein derartiges Regelobjekt enthält eine Implementierung der IRules-Schnittstelle und eine Implementierung der IPrint-Schnittstelle. Dieses spezifische Beispiel wird weiter unten mit Bezug auf die Code-Tabelle 14 erläutert.

[0150] Die Code-Tabelle 13 zeigt auch einen C++-Pseudocode für eine bevorzugte Klassendefinition für ein

Regelobjekt, das eine Auswahlregel zum Auswählen von Regelobjekten aus einer Liste von Regelobjekten enthält. Der einfacheren Darstellung halber, ist die durch dieses Regelobjekt vorgesehene Regel dem Standardcode ähnlich, der durch ein Mehrtypen-Objekt verwendet wird, um auf Regelobjekt aus der Liste von Regelobjekten (wie durch MTO::QueryInterface definiert) zuzugreifen. Dem Fachmann sollte jedoch deutlich sein, dass dieses auswählende Regelobjekt auch eine beliebige andere Regel für das Zugreifen auf Regelobjekte implementieren kann.

[0151] Wie in der Code-Tabelle 13 gezeigt, implementiert ein Mehrtypen-Objekt die zwei Schnittstellen IMultitype und IUnknown. Diese Schnittstellen sind im wesentlichen die gleichen wie die in der Code-Tabelle 11 beschriebenen, mit Ausnahme der im Folgenden beschriebenen Unterschiede. Der wesentliche Unterschied zwischen der Code-Tabelle 11 und der Code-Tabelle 13 besteht darin, dass die Methode QueryInterface der steuernden IUnknown-Schnittstelle (MTO::QueryInterface) nicht mehr alle drei Listen von eingeschlossenen Schnittstellen nach einer entsprechenden Schnittstelle in Übereinstimmung mit einem fixen Satz von Regeln durchsucht. (Die fixen Regeln von **Fig. 11** sehen vor, dass die erste gefundene Schnittstelle aus den Außerkraftsetzungs-, Normal- und Standardlisten in dieser Reihenfolge ausgegeben wird.) Wenn MTO::QueryInterface statt dessen bestimmt, dass eine Schnittstelle angefordert wird, die nicht durch das MTO-Objekt selbst implementiert wird, bestimmt sie, ob eine solche auf der Liste der Regelobjekt des MTO-Objekts vorhanden ist. Das gefundene Regelobjekt ist dann dafür verantwortlich, entweder eine Implementierung der angeforderten Schnittstelle vorzusehen oder eine Schnittstelle aus den drei Listen von aggregierten Schnittstellen des MTO-Objekts auszugeben. Eine Client-Anwendung kann also das Verhalten der Methode QueryInterface der steuernden IUnknown-Schnittstelle des aggregierten Objekts ändern, indem sie ihre eigenen Regelobjekte zum Abrufen oder Kombinieren von bestimmten Schnittstellen vorsieht. Diese Regelobjekte können unabhängig von der Implementierung des Mehrtypen-Objekts implementiert werden, weil sie eine durch das Mehrtypen-Objekt bereitgestellte öffentliche Aufzählungsmethode für den Zugriff auf die eingeschlossene Schnittstelle und die Regellisten verwenden können. Außerdem kann eine Client-Anwendung die Auswahlregel ändern, die durch MTO::QueryInterface verwendet wird, um das Regelobjekt in Entsprechung zu der angeforderten Schnittstelle zu finden, indem sie ein Regelobjekt für die IUnknown-Schnittstelle vorsieht.

[0152] Zusätzlich zu den bereits mit Bezug auf die Code-Tabelle 11 beschriebenen Methoden AddObject und AddInterface sieht die Klasse MTO zwei Methoden zum Handhaben von Regelobjekten vor: AddRule und Enum. Die Methode AddRule wird ähnlich wie AddInterface ausgeführt. Die Methode AddRule erstellt ein neues Listenelement für die Liste der Regelobjekte und initialisiert dieses Element, um auf die Schnittstellenkennzeichnung und die IRule-Schnittstelle des Regelobjekts zu zeigen, die als Eingabeparameter erhalten wurden. Die Methode setzt dann das neue Listenelement in die Liste der Regelobjekte ein und ruft die Methode Init der IRule-Schnittstelle auf, auf die durch das neue Listenelement gezeigt wird, um dem Regelobjekt den Zugriff zurück auf das MTO-Objekt zu ermöglichen. Ein derartiger Zugriff wird durch das Regelobjekt verwendet, um die Methode Enum für den Zugriff auf andere Liste von Schnittstellen aufzurufen.

[0153] Wie gezeigt, fügt die Methode AddRule ein neues Regelobjekt am Ende der Liste von Regelobjekten hinzu. Außerdem nehmen die QueryInterface-Methoden unter Verwendung der Liste von Regelobjekten an, dass nur ein Regelobjekt pro Schnittstellenkennzeichnung vorhanden ist (oder nur die erste gefundene gültig ist). Deshalb wird vorzugsweise eine Fehlerprüfung oder Reihenfolgenkontrolle durchgeführt. Dem Fachmann sollte deutlich sein, dass wie bei den anderen eingeschlossenen Schnittstellenlisten ein neues Regelobjekt entweder am Anfang oder am Ende der Liste eingefügt werden kann. Außerdem kann die Methode AddRule geändert werden, um einen Parameter auszugeben, der ähnlich wie bei AddObject und AddInterface angibt, wo das neue Regelobjekt hinzugefügt werden soll. Alternativ hierzu können die Regellistenelemente identisch wie die anderen Listenelemente implementiert werden, wobei die Methode AddInterface verwendet werden kann, um Regelobjekte in die Liste der Regelobjekte einzufügen. Dem Fachmann sollte deutlich sein, dass viele andere Techniken möglich sind und dass in einem abstrakten Sinne auch Regelobjekte dynamisch aggregierte Objekte sind.

[0154] Die Klasse MTO sieht die Methode Enum zum Suchen und Abrufen einer spezifizierten Schnittstelle vor. Regelobjekte zum Kombinieren von Regeln können diese Methode verwenden, um Schnittstellen von den drei aggregierten Schnittstellenlisten abzurufen. Außerdem können Regelobjekte zum Auswählen von Kombinationsregeln diese Methode verwenden, um Regelobjekt-IRule-Schnittstellen aus den Listen von Regelobjekten abzurufen. Wie gezeigt, gibt die Methode Enum das i-te Auftreten einer Schnittstelle aus, die einer angeforderten Schnittstellenkennzeichnung aus einer spezifizierten Liste entspricht. Die Methode beginnt am Anfang oder Ende der Liste, je nachdem wie dies in dem Aufruf spezifiziert ist. In einer alternativen Ausführungsform wird der Enumerator durch vier separate Methoden implementiert: eine pro aufgezählte Liste.

[0155] Für die Normal-, Standard- und Außerkraftsetzungslisten untersucht die Methode Enum jedes Listenelement aus der spezifizierten Liste, bis sie die angeforderte Schnittstellenkennzeichnung findet oder die Elemente in der Liste durchlaufen hat. Wenn während der Untersuchung ein Listenelement auf ein ganzes Objekt zeigt, dann ruft die Methode Enum die Methode QueryInterface des Objekts auf, auf das durch das Listenelement gezeigt wird, um potentiell die angeforderte Schnittstelle abzurufen und auszugeben. Wenn das Listenelement dagegen auf eine andere Schnittstelle als IUnknown zeigt, wird die Schnittstellenkennzeichnung, auf die das Listenelement zeigt, direkt mit der angeforderten Schnittstellenkennzeichnung verglichen. Wenn diese einander entsprechen, wird die Schnittstelle, auf die das Listenelement zeigt, ausgegeben. Wenn ein Listenelement keine entsprechende Schnittstelle vorsieht, fährt die Methode mit dem Durchsuchen des Rests der spezifizierten Liste fort.

[0156] Die Methode Enum sieht ein etwas anderes Verhalten für das Suchen und Abrufen aus der Liste von Regelobjekten vor. Anstatt die angeforderte Schnittstellenkennzeichnung auszugeben, verwendet die Methode die Schnittstellenkennzeichnung, um ein entsprechendes Element auf der Liste der Regelobjekte zu finden, und gibt dann einen Zeiger auf die IRule-Schnittstelle des Regelobjekts in Entsprechung zu der angeforderten Schnittstellenkennzeichnung aus.

[0157] In der Code-Tabelle 13 wurde die Methode QueryInterface des einschließenden MTO-Objekts modifiziert, um die Liste von Regelobjekten zu unterstützen. Die Methode bestimmt zuerst, ob die in einem Eingabeparameter angeforderte Schnittstelle durch das MTO-Objekt selbst implementiert wird. Wenn dies der Fall ist, gibt die Methode diese Schnittstelle aus. Ansonsten ruft die Methode eine Auswählregel ab und auf, wenn eine solche auf der Liste der Regelobjekte vorhanden ist. Wenn schließlich keine Auswählregel vorhanden ist, sieht die Methode QueryInterface einen Standardauswahlcode vor.

[0158] Ein Regelobjekt sieht eine Auswahlregel vor, wenn es zu der Liste der Regelobjekte unter der IUnknown-Schnittstellenkennzeichnung hinzugefügt wird. Vorzugsweise wird diese Schnittstellenkennzeichnung für diesen Zweck reserviert. In einer Ausführungsform wird das erste derartige gefundene Regelobjekt als Auswahlregel abgerufen. Wenn eine Kombinationsregel vorhanden ist, ist die abgerufene Auswahlregel für das Suchen und Abrufen einer Kombinationsregel verantwortlich. Und wenn keine Kombinationsregel vorhanden ist, ist die abgerufene Auswahlregel für das Suchen und Abrufen einer Schnittstelle aus den drei Listen der aggregierten Schnittstellen verantwortlich. Die abgerufene Schnittstellenkennzeichnung wird aufgerufen, indem das Regelobjekt (unter Verwendung der IRule-Schnittstelle) abgefragt wird, das der IUnknown-Schnittstellenkennzeichnung für seine IUnknown-Schnittstelle entspricht, und indem dann die Methode QueryInterface dieser abgerufenen IUnknown-Schnittstelle aufgerufen wird. Es ist zu beachten, dass die abgerufene IUnknown-Schnittstelle die steuernde IUnknown-Schnittstelle des Regelobjekts ist.

[0159] Wenn keine Auswahlregel existiert, sieht die Methode MTO::QueryInterface einen Standardauswahlcode vor. Dieser Standardauswahlcode gibt eine Kombinationsregel für die angeforderte Schnittstelle aus, wenn eine solche vorhanden ist, oder gibt die erste entsprechende Schnittstelle aus, die in den normal-, override- oder default-Listen in dieser Reihenfolge gefunden wird. (Der Standardauswahlcode verhält sich, wenn keine Kombinationsregel existiert, ähnlich wie die Methode MTO::QueryInterface.) Das die Kombinationsregel implementierende Regelobjekt ist dann dafür verantwortlich, entweder eine Implementierung der angeforderten Schnittstelle vorzusehen oder eine Schnittstelle aus den drei Listen von aggregierten Schnittstellen des MTO-Objekts auszugeben.

[0160] Als ein Beispiel zeigt die Code-Tabelle 13 eine Implementierung für ein Regelobjekt, das eine Auswahlregel zum Auswählen von Regelobjekten aus der Liste von Regelobjekten vorsieht. Die Klasse MyQI ist ein Regelobjekt, das eine IRule-Schnittstelle und eine IUnknown-Schnittstelle implementiert. Der einfacheren Darstellung halber ist die Klasse MyQI als ein nicht aggregierbares Objekt gezeigt. Die IRule-Schnittstelle sieht eine Methode Init vor, um einen Zeiger auf das einschließende Mehrtypen-Objekt (MTO) zu erhalten, das dieses Regelobjekt enthält. Dieser MTO-Zeiger kann später für den Zugriff auf die Methode Enum der IMultitype-Schnittstelle verwendet werden, um auf die aggregierten Schnittstellen und Objekte zuzugreifen. Die Methode Init wird durch AddRule aufgerufen, wenn ein Regelobjekt zu der Liste der Regelobjekte hinzugefügt wird. Die IUnknown-Schnittstelle sieht eine Implementierung von QueryInterface vor, die weiß, wie ein Kombinations-Regelobjekt aus der Liste der Regelobjekte auszuwählen ist. Wie gezeigt, sieht MyQI::QueryInterface eine identische Funktionalität wie MTO::QueryInterface vor. Die Implementierung unterscheidet sich jedoch dadurch, dass MyQI::QueryInterface die Methode Enum verwendet, um eine Kombinationsregel zu finden, und nicht, um die Liste selbst zu suchen. Dem Fachmann sollte deutlich sein, dass MTO::QueryInterface auf identische Weise implementiert werden könnte.

Code-Tabelle 14

```
class IPrint: public IUnknown
{       virtual boolean Print (void **ppobj) = 0;
}

class myPrintRule: public IUnknown
...
private:
...
        void myPrintRule: m_R(this), m_P(this) {/*any other initialization code */}

        class R: public IRule
        {
        public:
                void R(myPrintRule *prnyPrintRule) {m_pMyRule = pmyPrintRule) }

                // ... IUnknown methods are also implemented here which call the controlling
                // IUnknown methods for the enclosing object (public methods shown below)

                virtual HRESULT Init (IMultitype *pMTO) {m_pMyRule→m_pMTO = pMTO)
        private:
                myPrintRule   *m_pMyRule;
        };      //end of class R definition

        friend R;
        R       m_R;

        class P: public IPrint
        {
        public:
```

```
            void P(myPrintRule *pmyPrintRule) {m_pMyRule = pmyPrintRule) }

            // ... IUnknown methods are also implemented here which call the controlling
            // IUnknown methods for the enclosing object (public methods shown below)

            virtual void Print (void * pobj)
            // call each aggregated object print routine on each list in order of overriding
            // and then normal list and only call print routine from default list if no print
            // routine exists on normal list
            {       IID_IMultitype *p_iMT;
                    IID_IPrint      *p_iprint;
                    int     i,j;
                    p_iMT = m_pMyRule->m_pMTO;
                    for(i = 1; p_iMT->Enum(i, IID_IPrint, OVERRIDE_LIST, true, &p_iprint);
                            i++) p_iprint->Print(pobj);
                    for(j = 1; p_iMT->Enum(j, IID_IPrint, OVERRIDE_LIST, true, &p_iprint);
                            j++) p_iprint->Print(pobj);
                    if ((i==1) && (j==1) && (p_iMT->Enum(1,IID_IPrint, DEFAULT_LIST,
                            true, &p_print))) p_print->Print (pobj);
                    };
            }
        private:
            myPrintRule   *m_pMyRule;
        };
        friend P;
        P       m_P;

        int             m_refcount;
        IMultitype      *m_pMTO;

public:
...
        virtual HRESULT QueryInterface(REFIID iid, void **ppv)
        {       ret = TRUE;
                switch (iid)    {
                        ...
                        case IID_IPrint:
```

```
                    *ppv = &m_P;
                    AddRef();
                    break;
                case IID_IRule:
                    *ppv = &m_R;
                    AddRef();
                    break;
                case IID_IUnknown:
                    *ppv = this;
                    AddRef();
                    break;
                ...
            }
            retunr ret;
        }

        virtual void AddRef() { m_refcount++;}

        virtual void Release() {if (--m_refcount == 0) delete this;}

};
```

[0161] Die Code-Tabelle 14 zeigt einen C++-Pseudocode für ein Beispiel der Verwendung eines Mehrtypen-Objekts und eines Regelobjekts, um ein außer Kraft setzendes Verhalten für einen Satz von aggregierten Objekten oder Schnittstellen vorzusehen. Wenn gewöhnlich ein Satz von Objekten aus verschiedenen Typen aggregiert wird, sieht jedes Objekt seine eigene Druckmethode vor, die weiß, wie das Objekt zu drucken ist. Die Code-Tabelle 14 zeigt einen C++-Pseudocode für eine bevorzugte Klassendefinition eines Regelobjekts, das eine Kombinationsregel für die IPrint-Schnittstelle vorsieht. Die vorgesehene IPrint-Schnittstelle enthält eine Methode Print, die die Druckmethode jedes eingeschlossenen Objekts oder jeder eingeschlossenen Schnittstelle in einem einschließenden Mehrtypen-Objekt aufruft.

[0162] Die Klasse myPrintRule sieht eine Implementierung für wenigstens zwei Schnittstellen vor, wie dies alle Regelobjekte tun: in diesem Fall IRule und IPrint. Die durch die Klasse myPrintRule vorgesehene IRule-Schnittstelle ist funktionell mit der Klasse MyQI identisch, die bereits mit Bezug auf die Code-Tabelle 13 erläutert wurde. Die IPrint-Schnittstelle sieht einfach eine Kombinationsregel vor, die die drei Listen von aggregierten Objekten in einem Mehrtypen-Objekt durchläuft, nach anderen IPrint-Schnittstellen sucht und, wenn gefunden, deren Print-Routinen aufruft. Wie mit Bezug auf die Code-Tabelle 13 erläutert, gibt nach dem Hinzufügen dieses Regelobjekts zu der Liste der Regelobjekte in einem Mehrtypen-Objekt die Methode QueryInterface der steuernden IUnknown-Schnittstelle des Mehrtypen-Objekts vorzugsweise einen Zeiger auf diese Kombinationsregel-IPrint-Schnittstelle aus, wenn eine Client-Anwendung QueryInterface aufruft und IID_IPrint auf dem aggregierten Objekt anfordert. Wenn also die Client-Anwendung die Methode Print aufruft, wird statt dessen die durch dieses Regelobjekt implementierte Methode Print aufgerufen, um sicherzustellen, dass das aggregierte Objekt seine Komponenten korrekt druckt.

Code-Tabelle 15

```
pPrintRule = new(myPrintRule); // make a IPrint interface combining rule
pPrintRule->QueryInterface(IID_IRule, prule);
p_MTO->QueryInterface(IID_IMultitype, p_MT);
```

p_MT->AddRule(IID_IPrint, prule;) // add it to the Multitype object

[0163] Die Code-Tabelle 15 zeigt eine C++-Pseudocodesequenz von Aufrufen, die das in der Code-Tabelle 14 definierte Regelobjekt installieren. Zuerst wird eine neue Instanz des Regelobjekts der Klasse myPrintRule erstellt. Dann wird die IRule-Schnittstelle des neuen Regelobjekts für die spätere Verwendung abgerufen. Nachdem dann ein Zeiger auf ein Mehrtypen-Objekt (p_MTO) erhalten wurde, wird die IMultitype-Schnittstelle des Mehrtypen-Objekts abgerufen. Schließlich wird die Methode AddRule dieser abgerufenen IMultitype-Schnittstelle aufgerufen und erhält die Schnittstellenkennzeichnung für das neue Regelobjekt (IID_IPrint) sowie einen Zeiger auf die IRule-Schnittstelle des neuen Regelobjekts.

[0164] In einer bevorzugten Ausführungsform wird ein aggregiertes Objekt dauerhaft gespeichert, wobei derselbe für das Drucken erläuterte Mechanismus verwendet wird. Insbesondere sieht ein Mehrtypen-Objekt ein Kombinations-Regelobjekt für die Schnittstelle IPersistFile oder für eine andere Schnittstelle vor, die für das Speichern und Abrufen von Objekten verantwortlich ist. (Die Schnittstelle IPersistFile sieht Methoden zum Speichern und Laden von Objekten vor.) Auf diese Weise kann das Mehrtypen-Objekt die Speicherfunktion der eingeschlossenen Objekte und Schnittstellen außer Kraft setzen, um sicherzustellen, dass das gesamte aggregierte Objekt gespeichert wird. Die Kombinationsregel speichert vorzugsweise zuerst alle Datenstrukturen, die das Mehrtypen-Objekt benötigt, und ruft dann die Methode Save auf jeder der eingeschlossenen IPersistFile-Schnittstellen oder -Objekte auf. Um also an dem Speichern des aggregierten Objekts teilzunehmen, fügt ein Client-Programm die IPersistFile-Schnittstelle eines einzuschließenden Objekts hinzu oder schließt das gesamte Objekt ein. Es ist keine separate Speicherstelle in einer separaten Datei erforderlich: das Mehrtypen-Objekt kann seine eigenen Daten in ähnlicher Weise speichern wie Objektdaten in einer einzelnen Datei gespeichert werden.

[0165] In einer alternativen Ausführungsform, die nur eine Liste von aggregierten Schnittstellen oder Objekten und Regeln (anstatt von drei Listen plus der Liste der Regelobjekte) implementiert, benötigen die Mehrtyp-Objekten keine eigene Speicherung. Es ist kein Speicherplatz neben dem durch die aggregierten Objekte verwendeten erforderlich, weil das Mehrtypen-Objekt die einzelne Liste neu erstellen kann, indem es Ordnungs- und Indizierungsinformation der aggregierten Objekte in der hierarchischen peristenten Speicherstruktur untersucht. Ein Mehrtypen-Objekt, das mehrere Schnittstellenlisten implementieren kann, muss gewöhnlich Informationen speichern, um zu unterscheiden, auf welche Schnittstelleninstanzen durch welche Liste gezeigt wird.

[0166] Die vorliegende Ausführungsform der Erfindung nimmt an, dass wenn die steuernde IUnknown::QueryInterface-Methode die QueryInterface-Methoden der eingeschlossenen Objekte aufruft, ein derartiger Aufruf synchron durchgeführt wird. Das heißt, die QueryInterface-Methode des eingeschlossenen Objekts macht eine Ausgabe, bevor die steuernde IUnknown::QueryInterface-Methode die QueryInterface-Methode des nächsten eingeschlossenen Objekts aufruft. In einer alternativen Ausführungsform ruft die steuernde IUnknown::QueryInterface-Methode die QueryInterface-Methoden des eingeschlossenen Objekts asynchron auf und wartet nicht auf deren Ausgabe. Statt dessen werden Standard-Zwischenprozess-Kommunikationsmechanismen verwendet, um die steuernde Routine darüber zu benachrichtigen, dass die Methode des eingeschlossenen Objekts abgeschlossen ist. Eine derartige Ausführungsform ist in einer Netzwerk- oder Multiprozessorumgebung nützlich.

[0167] Eine typische Anwendung der vorliegenden Erfindung für die dynamische Aggregation besteht darin, dass ein Benutzer Instanzen von Objekten zu einem leistungsstärkeren Objekt unter Verwendung eines Programms kombiniert, das mit Wissen zu der dynamischen Aggregation entwickelt wurde. Zum Beispiel soll in dem mit Bezug auf den Stand der Technik erläuterten Beispiel der Tabellen-Schnittstelle angenommen werden, dass ein Softwarehersteller ein Tabellenkalkulationsprodukt vertreibt, mit dem ein Benutzer aggregierbare Tabellenobjekte erstellen kann, die nur die IBasic-Schnittstelle für die grundlegende Handhabung einer Tabelle unterstützen. (Das heißt, unter Verwendung dieses Produkts kann ein Benutzer Instanzen der IBasic-Schnittstelle erstellen, die Tabellenobjekte sind.) Es soll weiterhin angenommen werden, dass ein anderer Softwarehersteller ein Datenbankprodukt vertreibt, mit dem ein Benutzer aggregierbare Datenbankabfrageobjekte erstellen kann. Diese aggregierbaren Datenbankabfrageobjekte unterstützen die IDatabaseQuery-Schnittstelle, die zum Beispiel eine Methode DoQuery zum Ausführen einer Abfrage und eine Methode DefineQuery zum Spezifizieren einer Abfrage durch einen Benutzer umfasst. Wenn diese Datenbankabfrageobjekte mit anderen Objekten verbunden werden, die als Eingabedaten dienen können, können sie aufgerufen werden, um die verbundenen Eingabedaten abzufragen. Es soll weiterhin angenommen werden, dass in dem Computersystem ein Programm wie etwa ein Dateiverwaltungsprogramm vorgesehen ist, das Objekte verwaltet und weiß, wie Mehrtypen-Objekte erstellt werden und andere Objekte und Schnittstellen aggregiert werden. Dazu weiß das Dateiverwaltungsprogramm, wie ein Mehrtypen-Objekt hergestellt wird, das eine Implementierung der IMulti-

type-Schnittstelle vorsieht.

[0168] Fig. 12 ist eine bildliche Darstellung eines Tabellenkalkulationsobjekts und eines Datenbankabfrageobjekts, die aggregiert werden können, um ein verbundenes Datenbankabfrageobjekt zu erstellen. Um ein leistungsstärkeres Objekt zu erstellen, das eine spezifizierte Abfrage auf einer spezifizierten Datenbank durchführen kann (das verbundene Datenbankabfrageobjekt **1205**), erstellt der Benutzer zuerst unter Verwendung des Tabellenkalkulationsprodukts eine Instanz eines Tabellenobjekts **1201**, die der Benutzer als Eingabedaten verwenden möchte. Dann erstellt der Benutzer eine Instanz eines Datenbankabfrageobjekts **1202** unter Verwendung des Datenbankprodukts. Der Benutzer ruft dann die Methode DefineQuery der IDatabaseQuery-Schnittstelle auf, indem er eine Definitionstaste **1204** drückt und Text für die Datenbankabfrage unter Verwendung einer Datenbanksprache wie z. B. der SQL-Sprache eingibt. Dann ruft der Benutzer das Dateiverwaltungsprogramm **1208** auf. Unter Verwendung des Dateiverwaltungsprogramms wählt der Benutzer das Tabellenobjekt **1201** und das Datenbankabfrageobjekt **1202** und fordert an, dass das Dateiverwaltungsprogramm **1208** diese aggregiert. Der Benutzer kann diese Anforderung zum Beispiel tätigen, indem er die Anzeigewiedergabe des Tabellenobjekts **1201** auf den Dateneingabeport **1203** der Anzeigewiedergabe des Datenbankabfrageobjekts **1202** zieht und dort ablegt. In Reaktion auf die Anforderung des Benutzers instanziert das Dateiverwaltungsprogramm **1208** ein neues verbundenes Datenbankabfrageobjekt **1205**, indem es ein einschließendes Mehrtypen-Objekt instanziert und die Methode IMultitype::AddObject für das Tabellenobjekt **1201** und das Datenbankabfrageobjekt **1202** aufruft. Das Dateiverwaltungsprogramm **1208** gibt an das Tabellenobjekt **1201** und das Datenbankabfrageobjekt **1202** einen Zeiger auf die steuernde IUnknown-Schnittstelle des neu instanzierten Mehrtypen-Objekts. Der Benutzer kann dann das verbundene Datenbankabfrageobjekt aufrufen, indem er die „Go"-Taste **1206** wählt, um eine spezifizierte Abfrage auf den verbundenen Tabellendaten durchzuführen.

[0169] Dem Fachmann sollte deutlich sein, dass ein Objekt, sobald es aggregiert ist, den Regeln des einschließenden Objekts unterworfen ist. Das Verhalten des Tabellenobjekts **1201** und des Datenbankabfrageobjekts kann sich also nach dem Einschließen ändern.

[0170] Fig. 13 ist ein Blockdiagramm eines aggregierten verbundenen Datenbankabfrageobjekts. Das aggregierte Objekt **1304** entspricht dem verbundenen Datenbankabfrageobjekt, das mit Bezug auf Fig. 12 beschrieben wurde. Das aggregierte Objekt **1304** umfasst ein Mehrtypen-Objekt **1301**, ein Tabellenobjekt **1302** und ein Datenbankabfrageobjekt **1303**. Das Mehrtypen-Objekt **1301** wird wie oben beschrieben in Reaktion auf die Benutzeranforderung an das aggregierte Tabellenobjekt **1302** und das Datenbankanfrageobjekt **1303** erstellt. Während des Aggregationsprozesses erstellt das Mehrtypen-Objekt Zeiger **1305** und **1306**, die auf die aggregieren Objekte zeigen, und gibt den zu aggregierenden Objekten einen Zeiger auf die steuernde IUnknown-Schnittstelle. Wenn der Benutzer später die „Go"-Taste (1200 in Fig. 12) drückt, wird die steuernde IUnknown-Schnittstelle des Mehrtypen-Objekts **1307** aufgerufen, um die Methode DoQuery der IDatabaseQuery-Schnittstelle des Datenbankabfrageobjekts **1303** unter Verwendung der oben beschriebenen Ausführungsformen zu lokalisieren und aufzurufen. Die Methode DoQuery kann dann eine (bekannte) gewünschte Methode des Tabellenobjekts **1302** abfragen und aufrufen, indem sie unter Verwendung der Methode QueryInterface nach einer (bekannten) gewünschten Schnittstellenkennzeichnung des Tabellenobjekts **1302** sucht. (Zur Erinnerung sei hier darauf hingewiesen, dass der Aufruf der Methode QueryInterface an die Methode QueryInterface der steuernden IUnknown-Schnittstelle **1307** des Mehrtypen-Objekts **1301** weitergeleitet wird, die dann die gewünschte Schnittstelle lokalisiert, wenn sie existiert.) Das Datenbankabfrageobjekt **1303** kann auch die Verfahren des Tabellenobjekts **1302** aufrufen, ohne auf die Definitionen beim Kompilieren der Schnittstellen des Tabellenobjekts **1302** zugreifen zu müssen, solange es die Namen (und Parameter) der Methode und der Schnittstellenkennzeichnung kennt.

[0171] Die vorliegende Erfindung wurde mit Bezug auf eine bevorzugte Ausführungsform beschrieben, wobei die Erfindung jedoch nicht auf diese Ausführungsform beschränkt ist. Modifikationen der Erfindung können durch den Fachmann vorgenommen werden. Der Erfindungsumfang wird durch die beigefügten Ansprüche definiert.

Patentansprüche

1. Verfahren des Betreibens eines Computersystems zum Hinzufügen einer Schnittstelle zu einem Objekt, wobei die Schnittstelle durch ein Objekt (**302**, **303**; **601**) implementiert wird, das in der Lage ist, in ein einschließendes Objekt (**301**; **610**) eingeschlossen zu werden, wobei das Verfahren die folgenden Schritte umfasst:
Erzeugen einer Instanz des einschließenden Objekts (**301**; **610**), wobei das einschließende Objekt (**301**; **610**) ein Schnittstelle-Hinzufügen-Funktionselement zum Hinzufügen einer Schnittstelle zu dem einschließenden Objekt (**301**; **610**) und ein Abfragefunktionselement zum Wiedergewinnen eines Verweises auf eine hinzuge-

fügte Schnittstelle aufweist;
Erzeugen einer Instanz eines in dem einschließenden Objekt (**301**, **610**) einzuschließenden Objekts (**302**, **303**; **601**), wodurch dem einzuschließenden Objekt (**302**, **303**; **601**) ein Verweis auf das einschließende Objekt (**301**; **610**) übergeben wird, wobei das einzuschließende Objekt (**302**, **303**; **601**) eine Schnittstelle aufweist und ein Abfragefunktionselement zum Wiedergewinnen eines Verweises auf die Schnittstelle aufweist, und
Aufrufen des Schnittstelle-Hinzufügen-Funktionselements des einschließenden Objekts (**301**; **610**), wodurch dem Schnittstelle-Hinzufügen-Funktionselement des einschließenden Objekts (**301**; **610**) ein Verweis auf das einzuschließende Objekt (**302**, **303**; **601**) übergeben wird, um dadurch das Objekt einzuschließen und die Schnittstelle des eingeschlossenen Objekts (**302**, **303**) dem einschließenden Objekt (**301**; **610**) hinzuzufügen, wodurch, wenn das Abfragefunktionselement des einschließenden Objekts (**301**; **610**) aufgerufen wird, das Abfragefunktionselement des einschließenden Objekts (**301**; **610**) einen Verweis auf die hinzugefügte Schnittstelle des eingeschlossenen Objekts (**302**, **303**; **601**) zurückgibt.

2. Verfahren nach Anspruch 1, wobei das Aufrufen des Abfragefunktionselements des einschließenden Objekts (**301**; **610**) asynchron durchgeführt wird.

3. Verfahren nach Anspruch 1, wobei das Aufrufen des Abfragefunktionselements des einschließenden Objekts (**301**; **610**) synchron durchgeführt wird.

4. Verfahren nach Anspruch 1, wobei das einschließende Objekt (**301**; **610**) eine Standardschnittstelle bereitstellt.

5. Verfahren nach Anspruch 1, wobei das einschließende Objekt (**301**; **610**) eine Schnittstelle bereitstellt, die sich über die hinzugefügte Schnittstelle des eingeschlossenen Objekts (**302**, **303**; **601**) hinwegsetzt.

6. Verfahren nach Anspruch 1, wobei das einschließende Objekt (**301**; **610**) keine Kenntnis von der durch das eingeschlossene Objekt (**302**, **303**; **601**) preisgegebenen Schnittstelle hat.

7. Verfahren nach Anspruch 1, das weiter die folgenden Schritte umfasst:
Erzeugen einer Instanz eines in dem einschließenden Objekt (**301**; **610**) einzuschließenden zweiten Objekts, wodurch dem zweiten Objekt ein Verweis auf das einschließende Objekt (**301**; **610**) übergeben wird, wobei das zweite Objekt eine preisgegebene Schnittstelle aufweist, und
Aufrufen des Schnittstelle-Hinzufügen-Funktionselements des einschließenden Objekts (**301**; **610**), wobei die preisgegebene Schnittstelle des zweiten Objekts dem einschließenden Objekt (**301**; **610**) hinzugefügt wird, um dadurch das zweite Objekt einzuschließen, wodurch, wenn das Abfragefunktionselement des eingeschlossenen ersten Objekts aufgerufen wird, das Abfragefunktionselement des eingeschlossenen ersten Objekts einen Verweis auf die hinzugefügte, preisgegebene Schnittstelle des eingeschlossenen zweiten Objekts zurückgibt.

8. Verfahren nach Anspruch 7, wobei sich die preisgegebene Schnittstelle des eingeschlossenen zweiten Objekts über eine Schnittstelle des eingeschlossenen ersten Objekts hinwegsetzt.

9. Verfahren nach Anspruch 7, wobei die preisgegebene Schnittstelle des eingeschlossenen zweiten Objekts eine Schnittstelle des eingeschlossenen ersten Objekts aufruft.

10. Verfahren nach Anspruch 7, wobei die preisgegebene Schnittstelle des eingeschlossenen zweiten Objekts eine Standardschnittstelle ist.

11. Verfahren nach Anspruch 7, wobei das eingeschlossene erste Objekt einen Zeiger auf das einschließende Objekt (**301**) enthält, das zweite Objekt ein Abfragefunktionselement zum Wiedergewinnen eines Verweises auf die preisgegebene Schnittstelle des zweiten Objekts aufweist, und weiter die folgenden Schritte umfasst:
erstens, Aufrufen des Abfragefunktionselements des eingeschlossenen ersten Objekts, um eine preisgegebene Schnittstelle des zweiten Objekts anzufordern;
zweitens, Wiedergewinnen eines Verweises auf das Abfragefunktionselement des einschließenden Objekts durch den in dem eingeschlossenen ersten Objekt enthaltenen Zeiger auf das einschließende Objekt (**301**; **610**);
drittens, Aufrufen des Abfragefunktionselements des einschließenden Objekts (**301**; **610**), Übergeben der angeforderten Schnittstelle;
viertens, von dem Abfragefunktionselement des einschließenden Objekts (**301**) Aufrufen des Abfragefunktionselements des eingeschlossenen zweiten Objekts, Übergeben der angeforderten Schnittstelle, und

fünftens, von dem Abfragefunktionselement des eingeschlossenen zweiten Objekts, Zurückgeben eines Verweises auf die angeforderte Schnittstelle.

12. Verfahren nach Anspruch 7, wobei das einschließende Objekt (**301**; **610**) ein Regel-Hinzufügen-Funktionselement zum Hinzufügen von Regeln aufweist, um zu bestimmen, für welche hinzugefügte Schnittstelle ein Verweis wiederzugewinnen ist, und weiter umfassend den Schritt des Aufrufens des Regel-Hinzufügen-Funktionselements des einschließenden Objekts (**301**; **610**), um eine Regel hinzuzufügen, um zu bestimmen, für welche hinzugefügte Schnittstelle ein Verweis wiederzugewinnen ist, wodurch, wenn das Abfragefunktionselement des einschließenden Objekts (**301**; **610**) aufgerufen wird, das Abfragefunktionselement nach der hinzugefügten Regel bestimmt, für welche hinzugefügte Schnittstelle ein Verweis zurückzugeben ist.

13. Verfahren nach Anspruch 1, wobei das Computersystem eine Vorrichtung zur bleibenden Speicherung aufweist, weiter umfassend die folgenden Schritte:
nach Einschließen eines Objekts (**302**, **303**; **610**) in das einschließende Objekt (**301**; **610**), um dadurch ein Gesamtobjekt zu erzeugen, Speichern des Gesamtobjekts auf einer bleibenden Speichervorrichtung, und
anschließend Laden des gespeicherten Gesamtobjekts (**301**; **610**) aus der bleibenden Speichervorrichtung.

14. Computersystem zum Hinzufügen einer Schnittstelle zu einem Objekt, wobei das System umfasst:
ein einschließendes Objekt (**301**; **610**) mit einem Schnittstelle-Hinzufügen-Funktionselement zum Hinzufügen einer Schnittstelle zu dem einschließenden Objekt (**301**; **610**) und einem Abfragefunktionselement zum Wiedergewinnen eines Verweises auf eine hinzugefügte Schnittstelle, und
ein in dem einschließenden Objekt (**301**; **610**) einzuschließendes Objekt (**302**, **303**; **601**), wodurch ein Verweis auf das einschließende Objekt (**301**; **610**) an das einzuschließende Objekt übergeben wird, wobei das einzuschließende Objekt (**302**, **303**; **601**) eine Schnittstelle und ein Abfragefunktionselement zum Wiedergewinnen eines Verweises auf die Schnittstelle aufweist, wobei das Objekt eingeschlossen (**302**, **303**; **601**) wird und die Schnittstelle hinzugefügt wird, indem das Schnittstelle-Hinzufügen-Funktionselement des einschließenden Objekts (**301**; **610**) aufgerufen wird, wodurch dem Schnittstelle-Hinzufügen-Funktionselement des einschließenden Objekts (**301**; **610**) ein Verweis auf das einzuschließende Objekt (**302**, **303**; **601**) übergeben wird, wobei, nachdem das Objekt eingeschlossen ist, das Schnittstelle-Abfragen-Funktionselement einen Verweis auf die hinzugefügte Schnittstelle des eingeschlossenen Objekts (**302**, **303**; **601**) zurückgibt.

Es folgen 16 Blatt Zeichnungen

Anhängende Zeichnungen

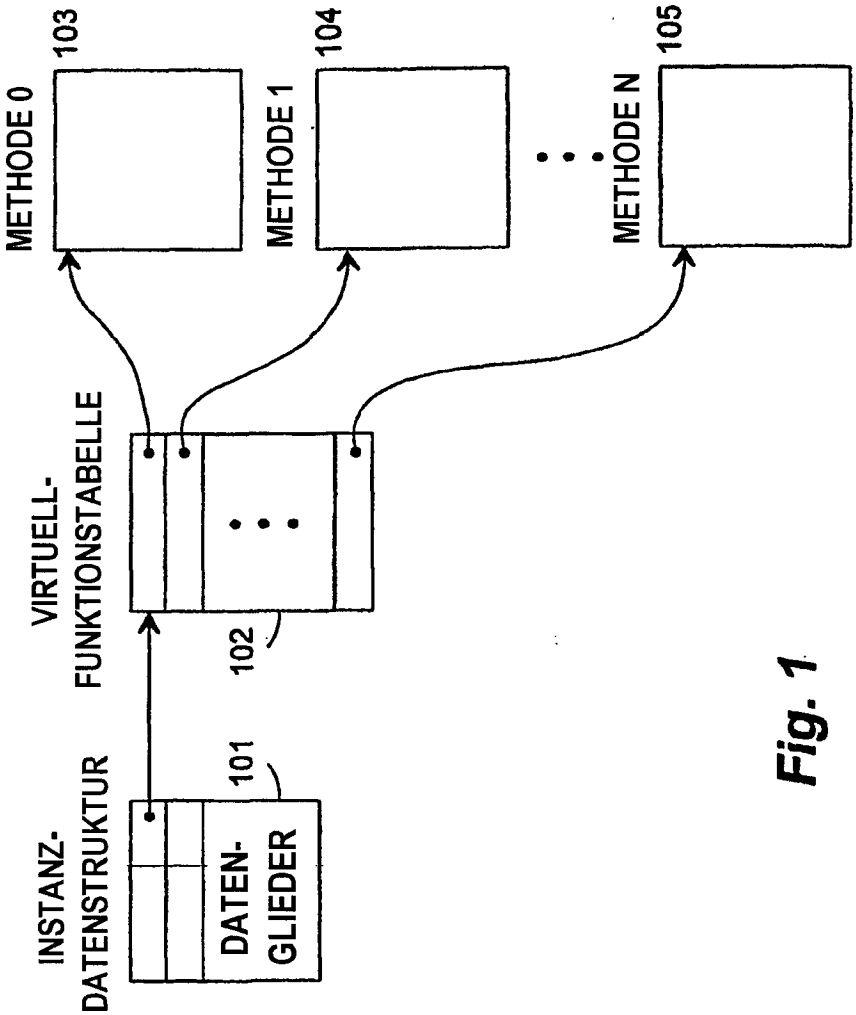

Fig. 1

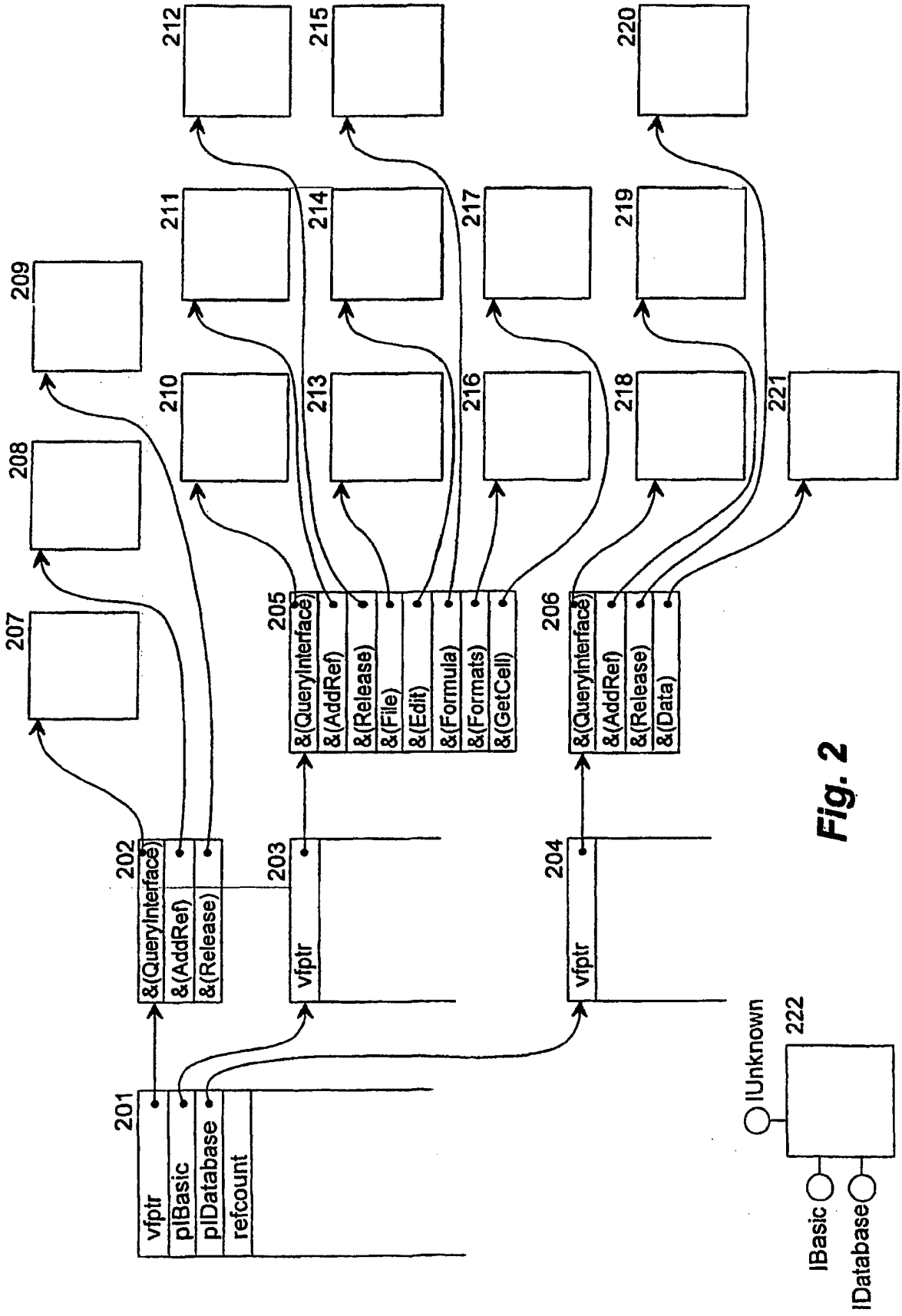
201
vfptr
plBasic
plDatabase
refcount
202
&(QueryInterface)
&(AddRef)
&(Release)
203
vfptr
204
vfptr
205
&(QueryInterface)
&(AddRef)
&(Release)
&(File)
&(Edit)
&(Formula)
&(Formats)
&(GetCell)
206
&(QueryInterface)
&(AddRef)
&(Release)
&(Data)
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
IUnknown
IBasic
IDatabase

Fig. 2

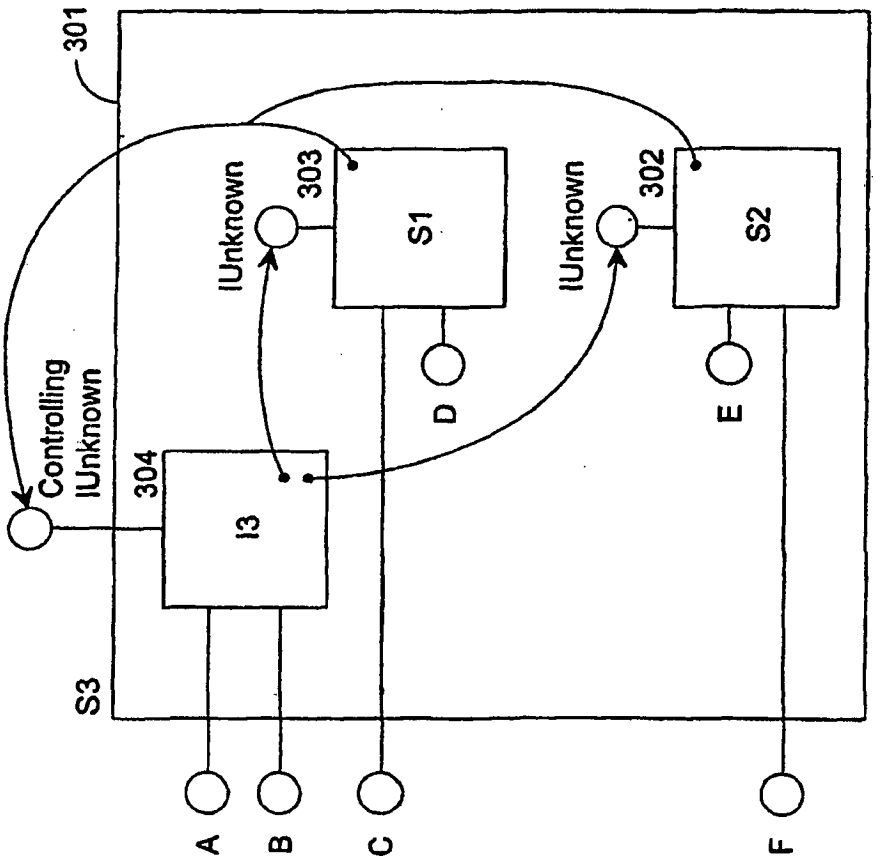
301
303
IUnknown
S1
302
IUnknown
S2
Controlling
IUnknown
304
I3
D
E
S3
A
B
C
F

Fig. 3

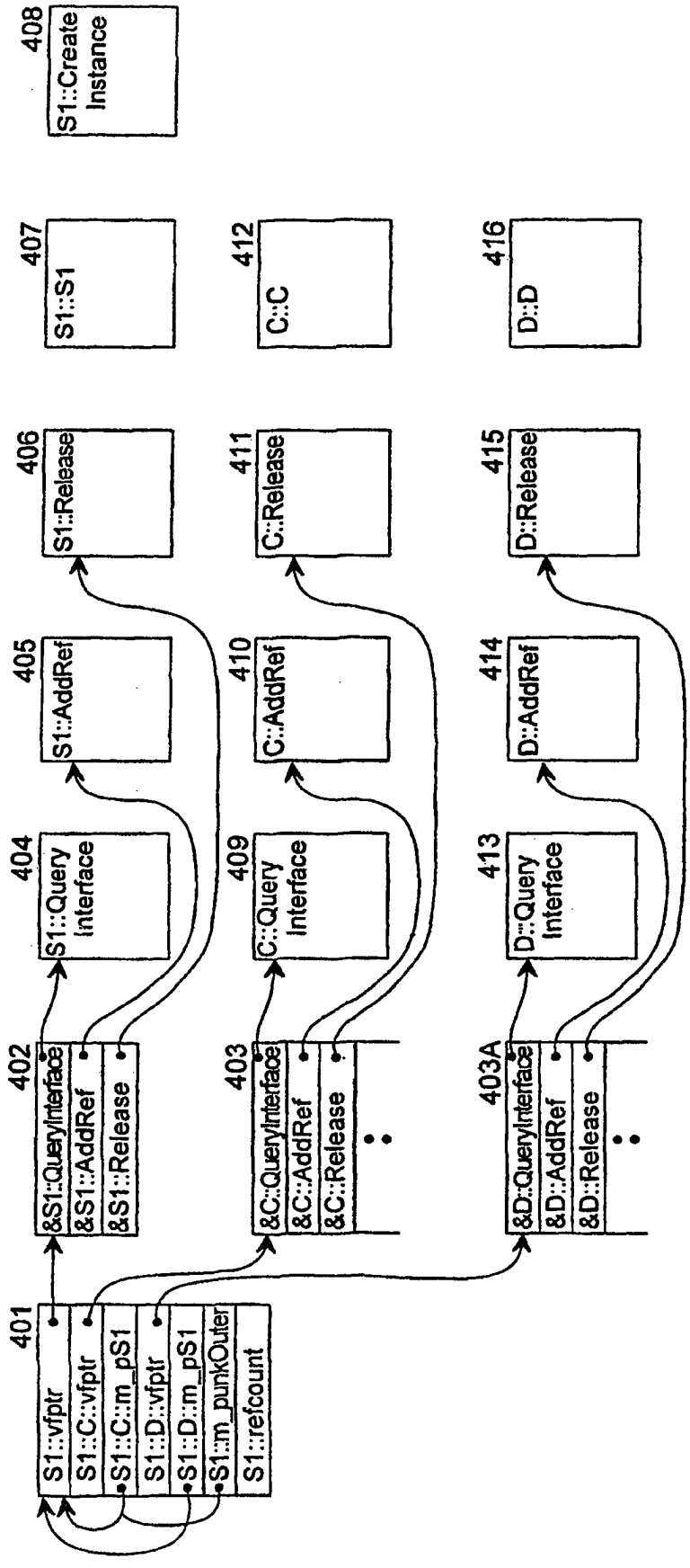
408
S1::Create Instance
407
S1::S1
412
C::C
416
D::D
406
S1::Release
411
C::Release
415
D::Release
405
S1::AddRef
410
C::AddRef
414
D::AddRef
404
S1::Query interface
409
C::Query interface
413
D::Query Interface
402
&S1::QueryInterface
&S1::AddRef
&S1::Release
403
&C::QueryInterface
&C::AddRef
&C::Release
403A
&D::QueryInterface
&D::AddRef
&D::Release
401
S1::vfptr
S1::C::vfptr
S1::C::m_pS1
S1::D::vfptr
S1::D::m_pS1
S1::m_punkOuter
S1::refcount

Fig. 4

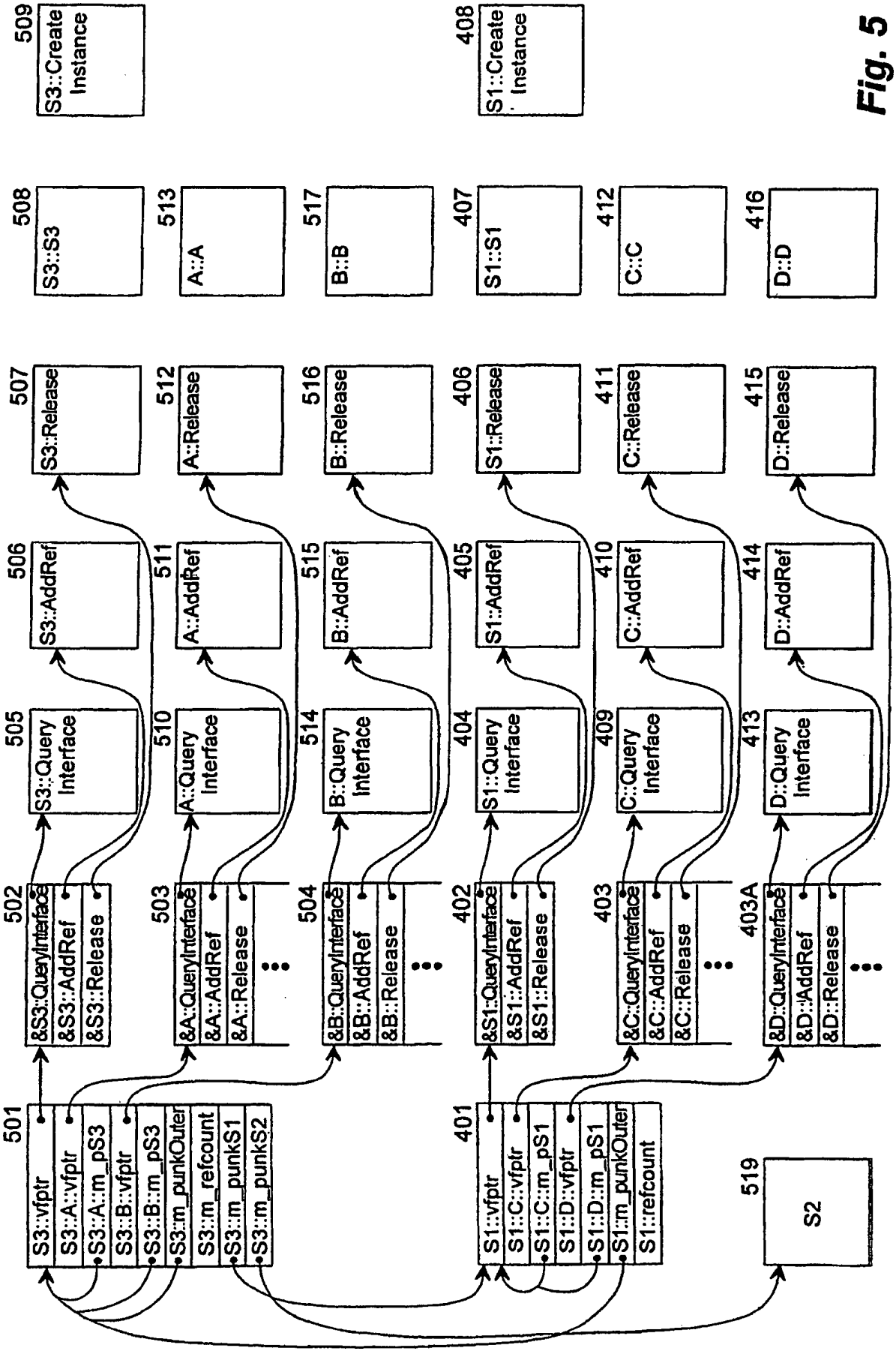
509
S3::Create Instance
408
S1::Create Instance
508
S3::S3
513
A::A
517
B::B
407
S1::S1
412
C::C
416
D::D
507
S3::Release
512
A::Release
516
B::Release
406
S1::Release
411
C::Release
415
D::Release
506
S3::AddRef
511
A::AddRef
515
B::AddRef
405
S1::AddRef
410
C::AddRef
414
D::AddRef
505
S3::Query Interface
510
A::Query Interface
514
B::Query Interface
404
S1::Query Interface
409
C::Query Interface
413
D::Query Interface
502
&S3::QueryInterface
&S3::AddRef
&S3::Release
503
&A::QueryInterface
&A::AddRef
&A::Release
504
&B::QueryInterface
&B::AddRef
&B::Release
402
&S1::QueryInterface
&S1::AddRef
&S1::Release
403
&C::QueryInterface
&C::AddRef
&C::Release
403A
&D::QueryInterface
&D::AddRef
&D::Release
501
S3::vfptr
S3::A::vfptr
S3::A::m_pS3
S3::B::vfptr
S3::B::m_pS3
S3::m_punkOuter
S3::m_refcount
S3::m_punkS1
S3::m_punkS2
401
S1::vfptr
S1::C::vfptr
S1::C::m_pS1
S1::D::vfptr
S1::D::m_pS1
S1::m_punkOuter
S1::refcount
519
S2

Fig. 5

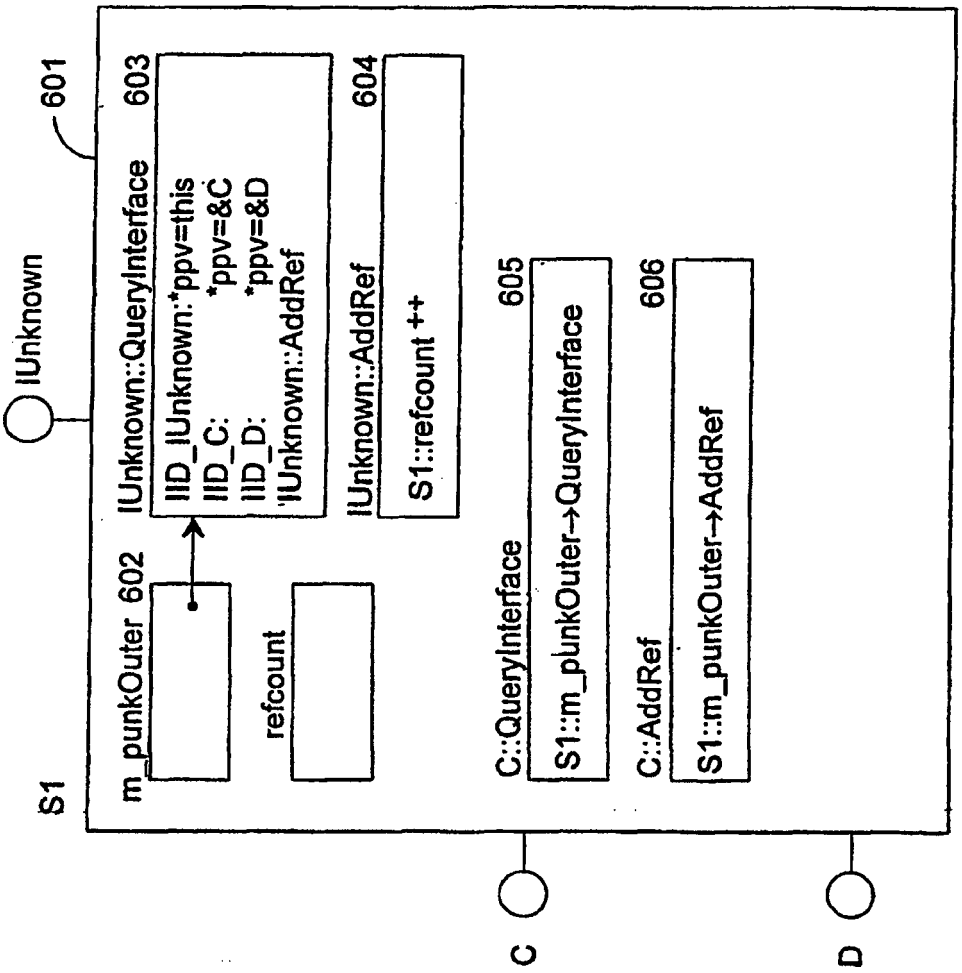
S1
601
IUnknown
m_punkOuter 602
IUnknown::QueryInterface 603
IID_IUnknown:*ppv=this
IID_C: *ppv=&C
IID_D: *ppv=&D
'IUnknown::AddRef
refcount
IUnknown::AddRef 604
S1::refcount ++
C::QueryInterface 605
S1::m_punkOuter→QueryInterface
C::AddRef 606
S1::m_punkOuter→AddRef
C
D

Fig. 6A

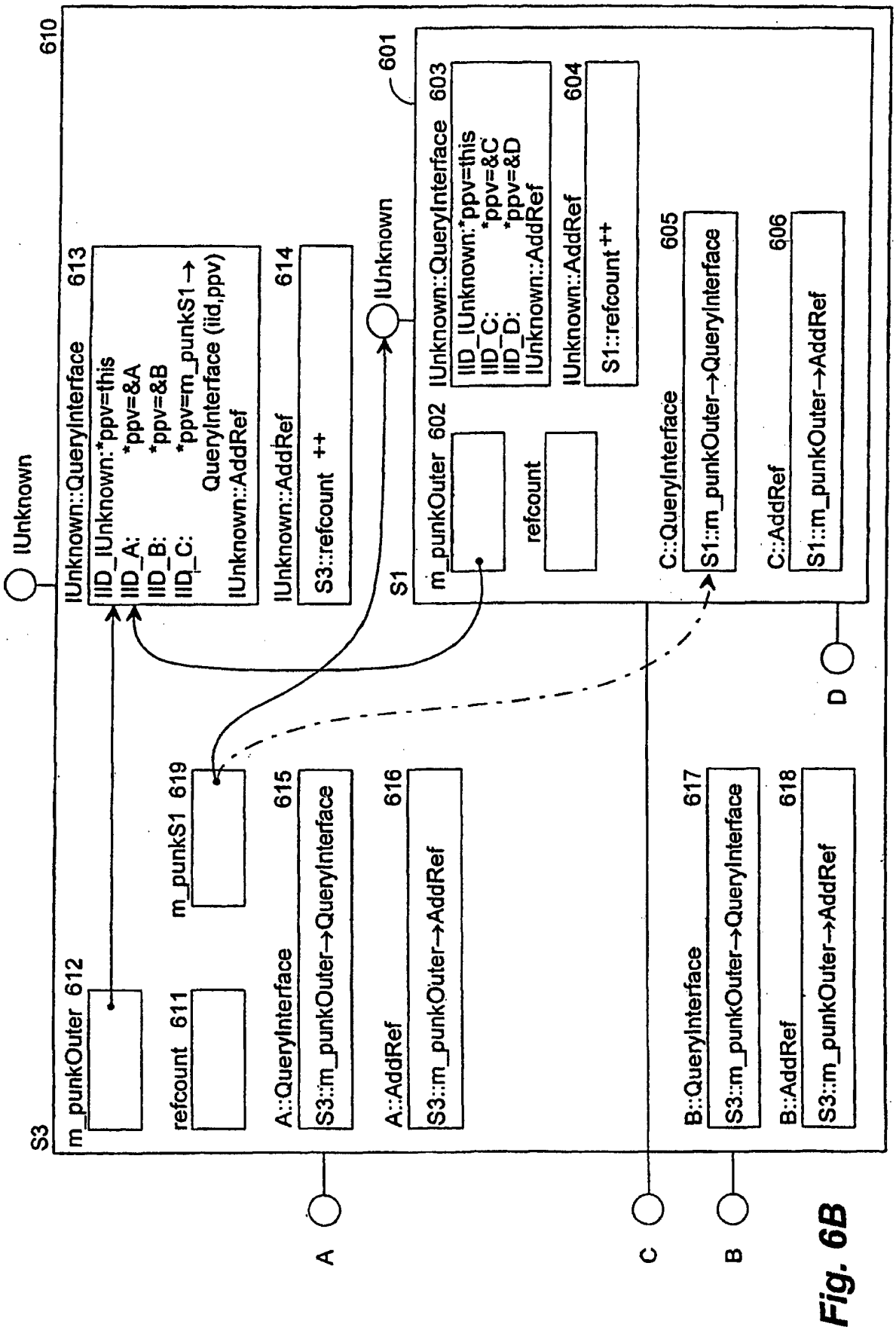
S3
610
IUnknown
m_punkOuter 612
refcount 611
m_punkS1 619
IUnknown::QueryInterface 613
IID_IUnknown:*ppv=this
IID_A: *ppv=&A
IID_B: *ppv=&B
IID_C: *ppv=m_punkS1→QueryInterface (iid,ppv)
IUnknown::AddRef
IUnknown::AddRef 614
S3::refcount ++
A::QueryInterface 615
S3::m_punkOuter→QueryInterface
A::AddRef 616
S3::m_punkOuter→AddRef
IUnknown
601
S1
m_punkOuter 602
refcount
IUnknown::QueryInterface 603
IID_IUnknown:*ppv=this
IID_C: *ppv=&C
IID_D: *ppv=&D
IUnknown::AddRef
IUnknown::AddRef 604
S1::refcount ++
C::QueryInterface 605
S1::m_punkOuter→QueryInterface
C::AddRef 606
S1::m_punkOuter→AddRef
B::QueryInterface 617
S3::m_punkOuter→QueryInterface
B::AddRef 618
S3::m_punkOuter→AddRef
A
C
B
D

Fig. 6B

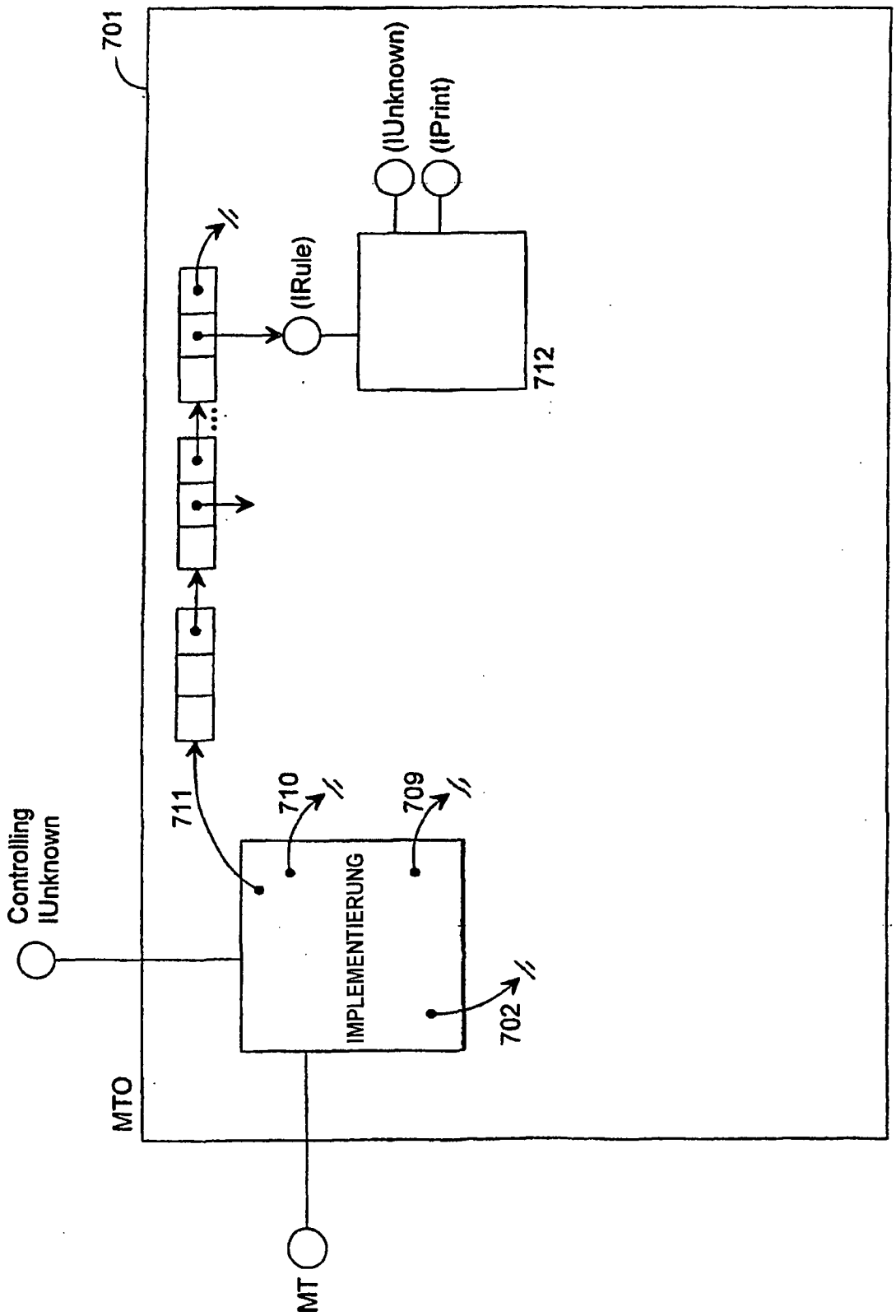
701
(IUnknown)
(IPrint)
(IRule)
712
711
710
709
IMPLEMENTIERUNG
702
Controlling
IUnknown
MTO
MT

Fig. 7A

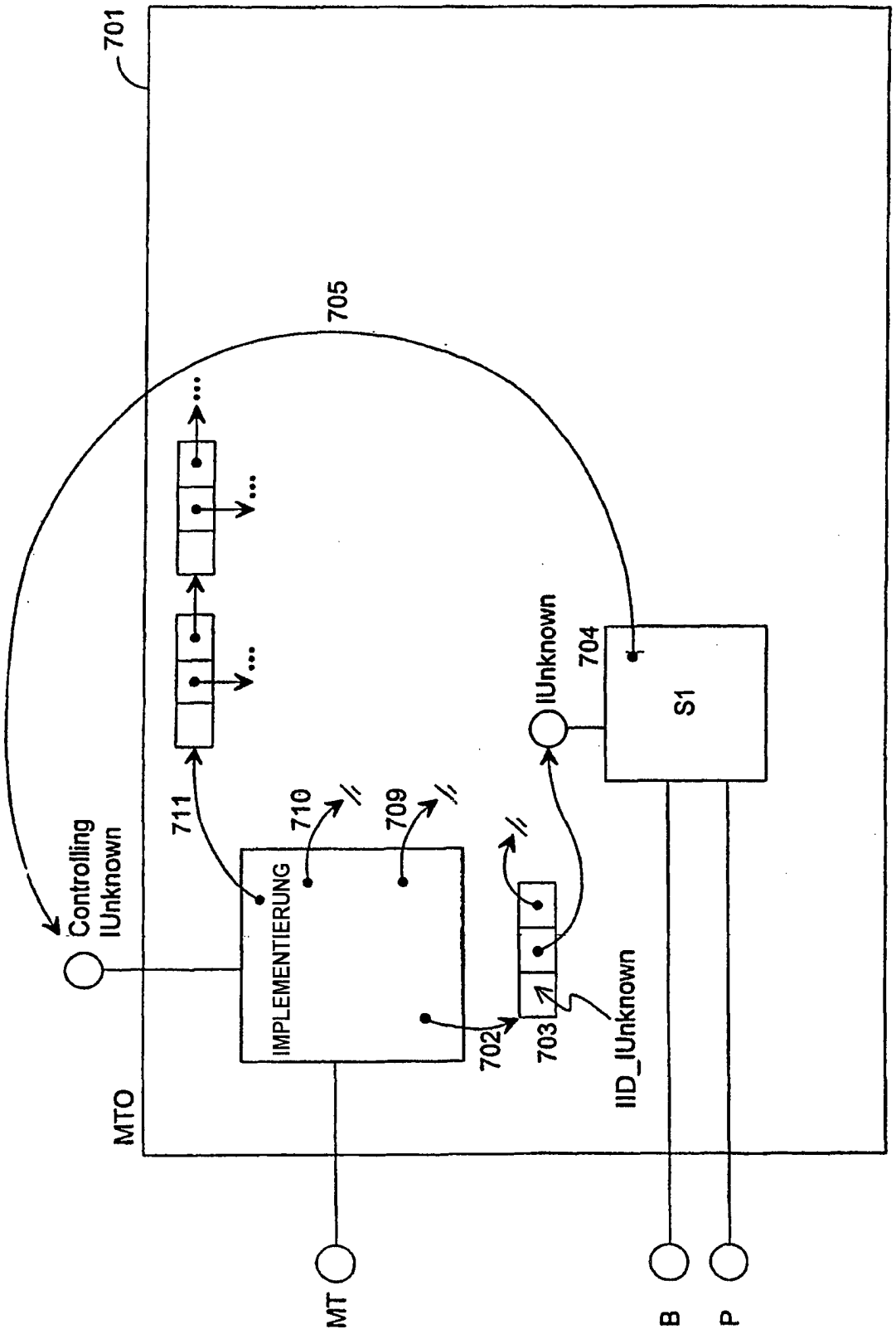
701
705
711
710
709
Controlling
IUnknown
IMPLEMENTIERUNG
IUnknown
704
S1
MTO
702
703
IID_IUnknown
MT
B
P

Fig. 7B

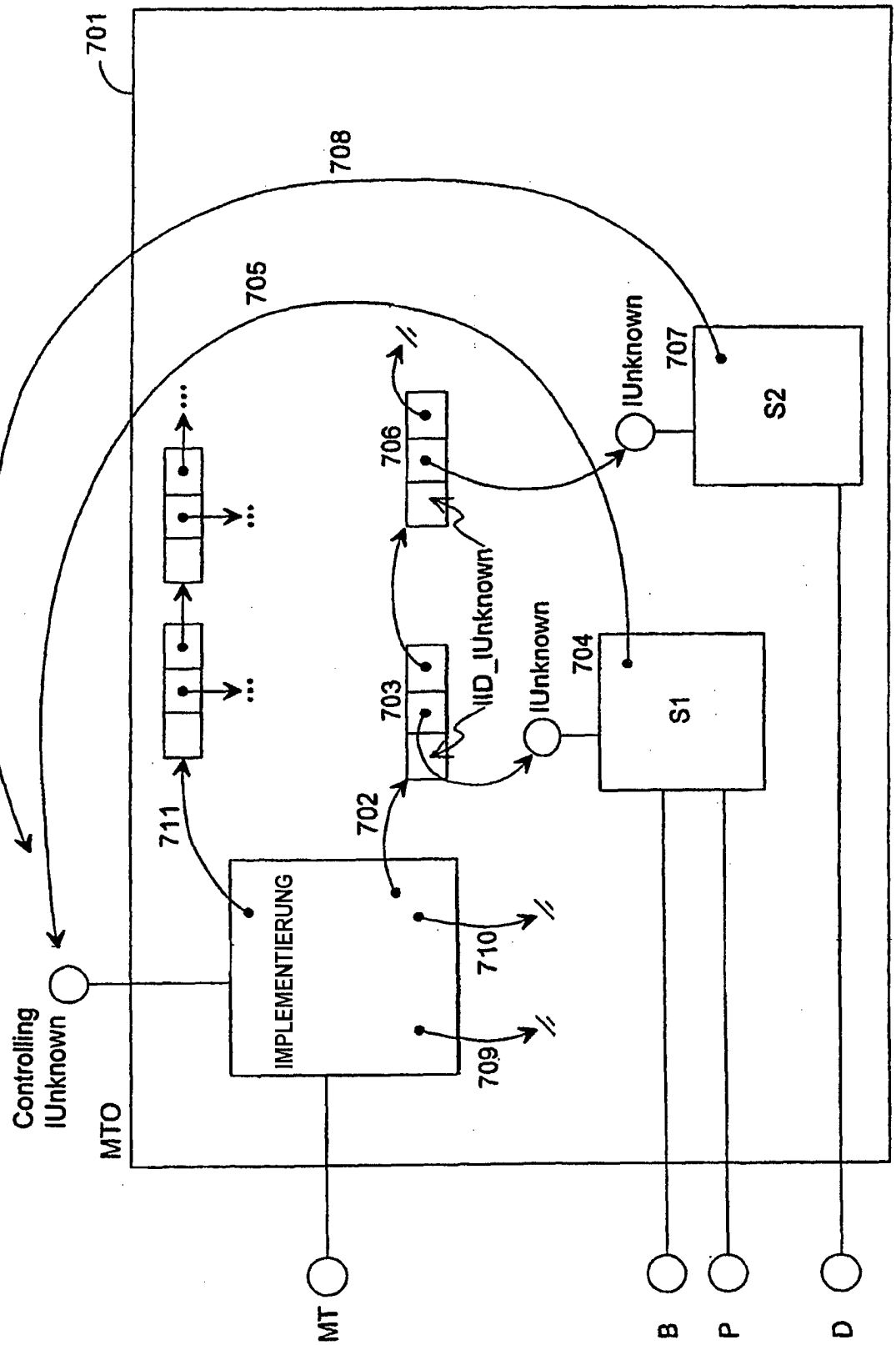
701
708
705
IUnknown
707
S2
706
IID_IUnknown
IUnknown
704
S1
703
702
711
IMPLEMENTIERUNG
710
709
Controlling IUnknown
MTO
MT
B
P
D

Fig. 7C

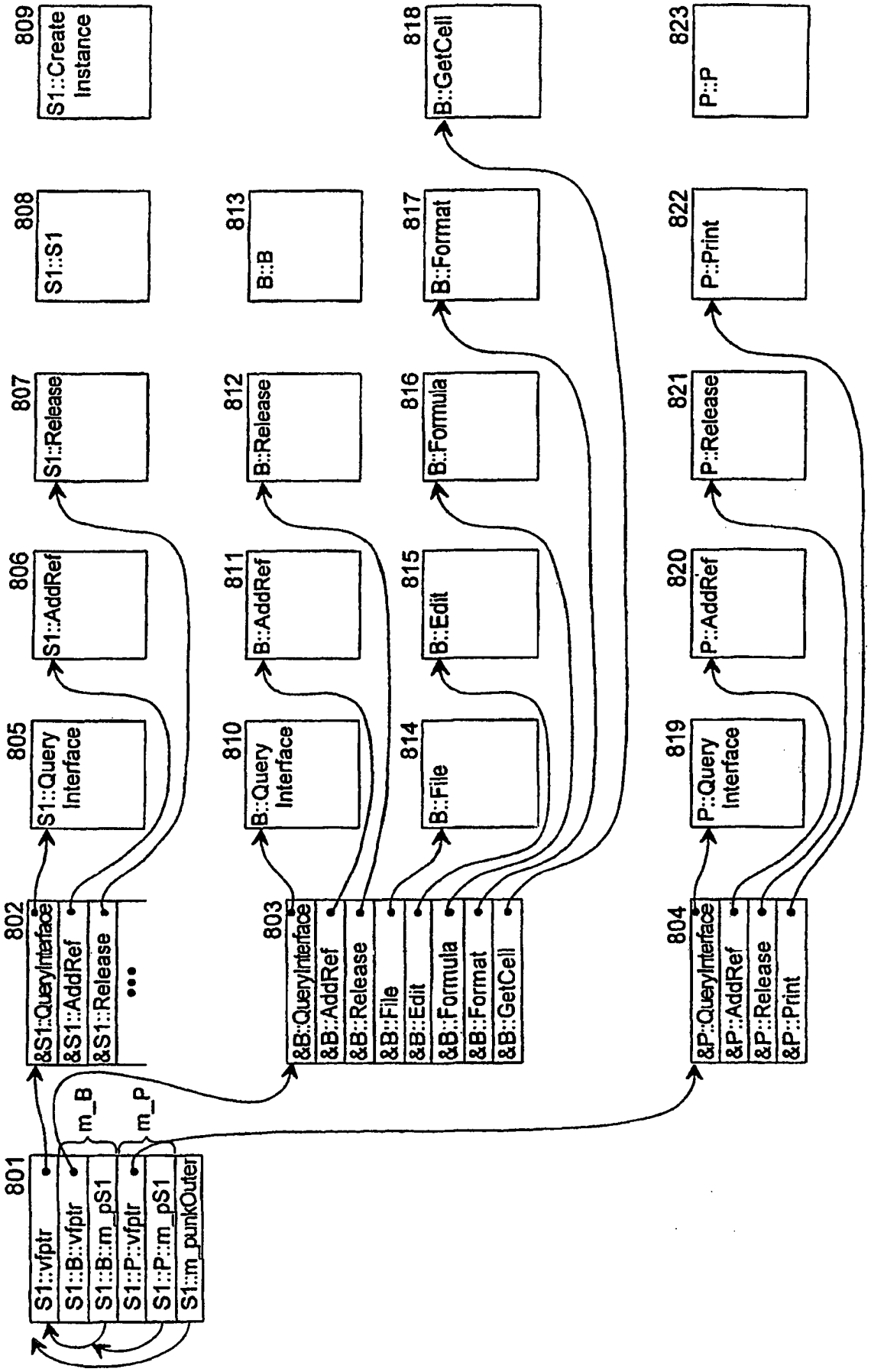
801
S1::vfptr
S1::B::vfptr
S1::B::m_pS1
S1::P::vfptr
S1::P::m_pS1
S1::m_punkOuter
m_B
m_P
802
&S1::QueryInterface
&S1::AddRef
&S1::Release
...
803
&B::QueryInterface
&B::AddRef
&B::Release
&B::File
&B::Edit
&B::Formula
&B::Format
&B::GetCell
804
&P::QueryInterface
&P::AddRef
&P::Release
&P::Print
805
S1::Query Interface
806
S1::AddRef
807
S1::Release
808
S1::S1
809
S1::Create Instance
810
B::Query Interface
811
B::AddRef
812
B::Release
813
B::B
814
B::File
815
B::Edit
816
B::Formula
817
B::Format
818
B::GetCell
819
P::Query Interface
820
P::AddRef
821
P::Release
822
P::Print
823
P::P

Fig. 8

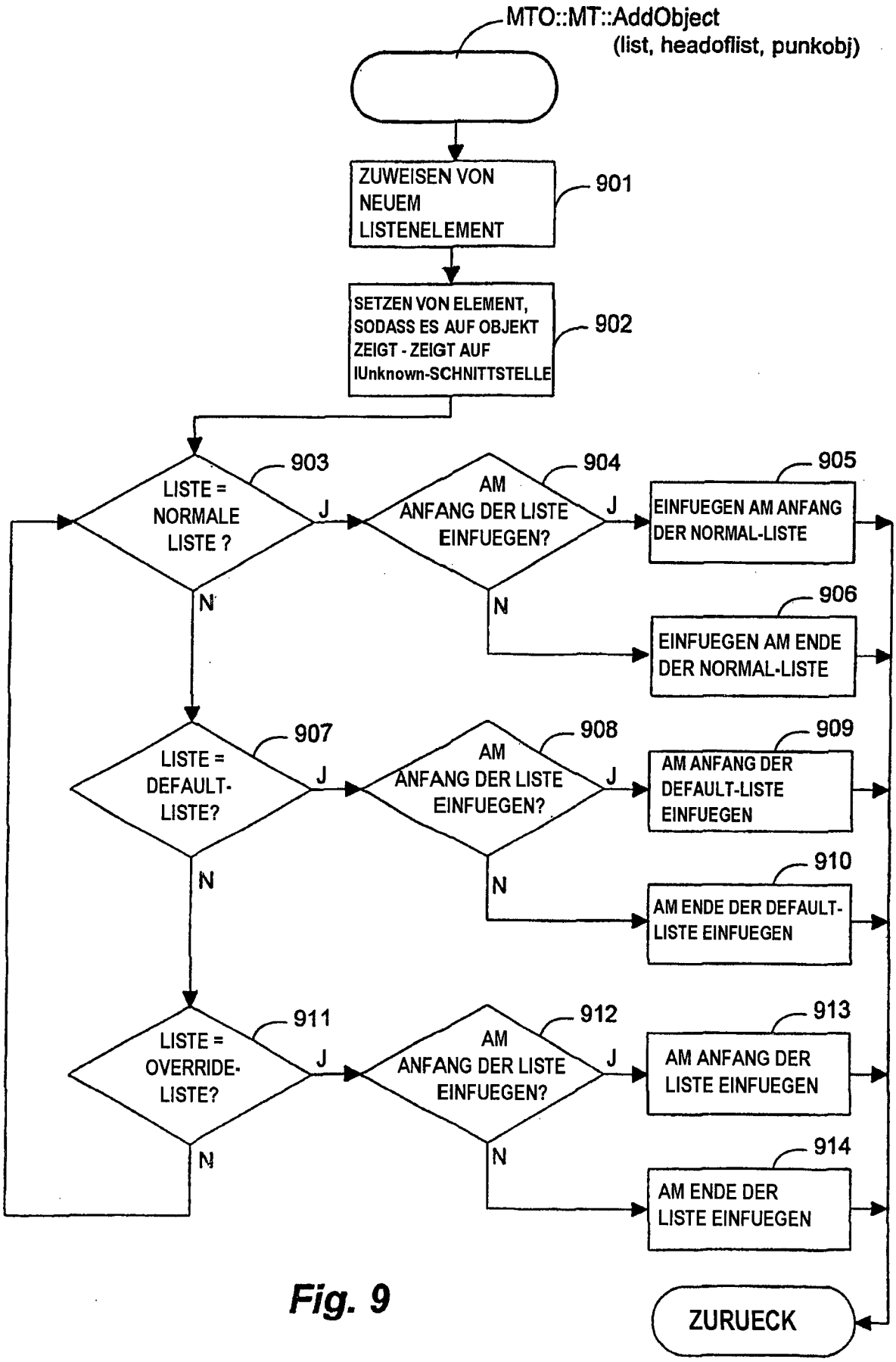
MTO::MT::AddObject
(list, headoflist, punkobj)
ZUWEISEN VON NEUEM LISTENELEMENT
901
SETZEN VON ELEMENT, SODASS ES AUF OBJEKT ZEIGT - ZEIGT AUF IUnknown-SCHNITTSTELLE
902
LISTE = NORMALE LISTE ?
903
J
N
AM ANFANG DER LISTE EINFUEGEN?
904
J
N
EINFUEGEN AM ANFANG DER NORMAL-LISTE
905
EINFUEGEN AM ENDE DER NORMAL-LISTE
906
LISTE = DEFAULT-LISTE?
907
J
N
AM ANFANG DER LISTE EINFUEGEN?
908
J
N
AM ANFANG DER DEFAULT-LISTE EINFUEGEN
909
AM ENDE DER DEFAULT-LISTE EINFUEGEN
910
LISTE = OVERRIDE-LISTE?
911
J
N
AM ANFANG DER LISTE EINFUEGEN?
912
J
N
AM ANFANG DER LISTE EINFUEGEN
913
AM ENDE DER LISTE EINFUEGEN
914
ZURUECK

Fig. 9

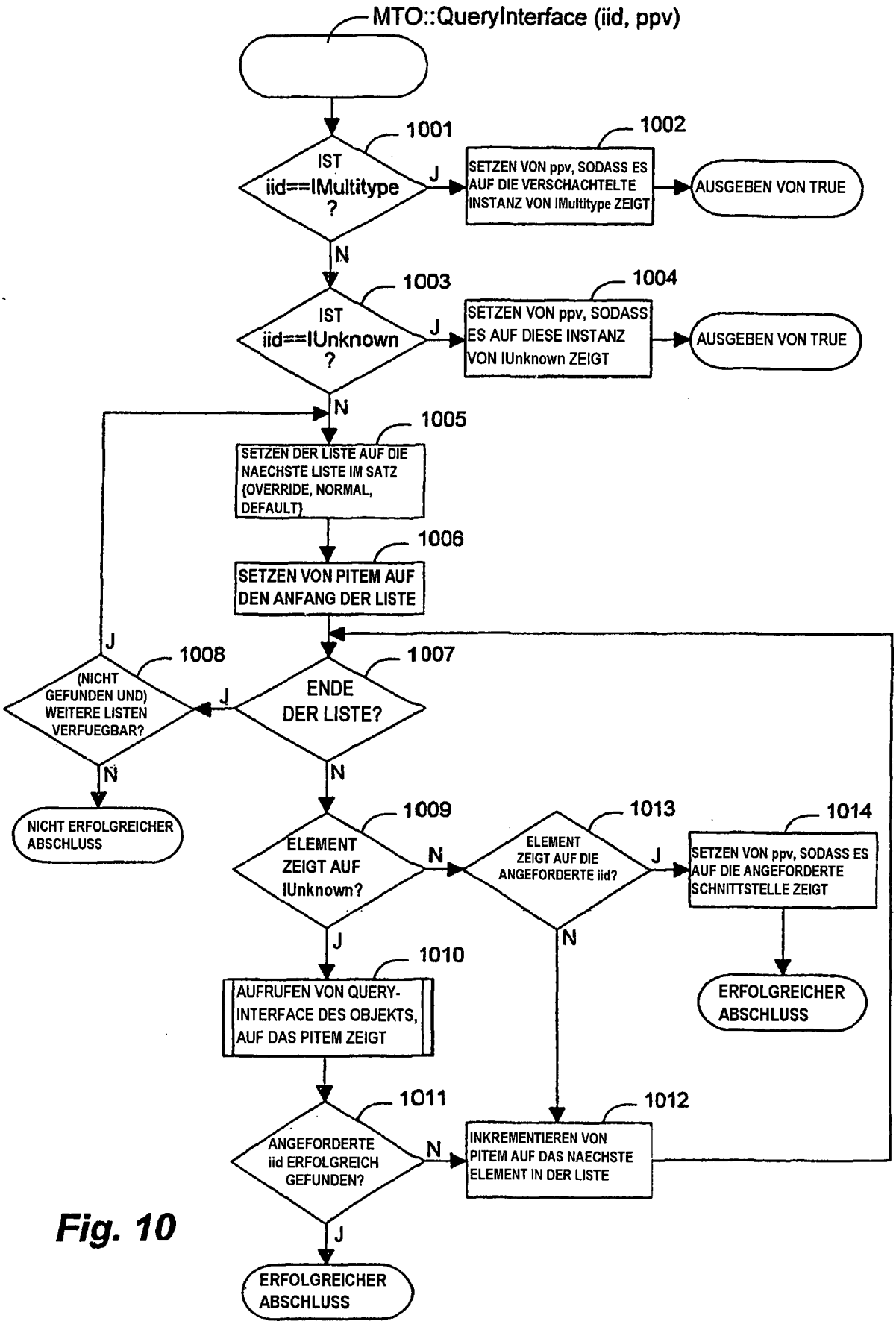
MTO::QueryInterface (iid, ppv)
1001
IST iid==IMultitype ?
J
1002
SETZEN VON ppv, SODASS ES AUF DIE VERSCHACHTELTE INSTANZ VON IMultitype ZEIGT
AUSGEBEN VON TRUE
N
1003
IST iid==IUnknown ?
J
1004
SETZEN VON ppv, SODASS ES AUF DIESE INSTANZ VON IUnknown ZEIGT
AUSGEBEN VON TRUE
N
1005
SETZEN DER LISTE AUF DIE NAECHSTE LISTE IM SATZ {OVERRIDE, NORMAL, DEFAULT}
1006
SETZEN VON PITEM AUF DEN ANFANG DER LISTE
J
1008
(NICHT GEFUNDEN UND) WEITERE LISTEN VERFUEGBAR?
J
1007
ENDE DER LISTE?
N
NICHT ERFOLGREICHER ABSCHLUSS
1009
ELEMENT ZEIGT AUF IUnknown?
N
1013
ELEMENT ZEIGT AUF DIE ANGEFORDERTE iid?
J
1014
SETZEN VON ppv, SODASS ES AUF DIE ANGEFORDERTE SCHNITTSTELLE ZEIGT
ERFOLGREICHER ABSCHLUSS
J
1010
AUFRUFEN VON QUERY-INTERFACE DES OBJEKTS, AUF DAS PITEM ZEIGT
N
1011
ANGEFORDERTE iid ERFOLGREICH GEFUNDEN?
N
1012
INKREMENTIEREN VON PITEM AUF DAS NAECHSTE ELEMENT IN DER LISTE
J
ERFOLGREICHER ABSCHLUSS

Fig. 10

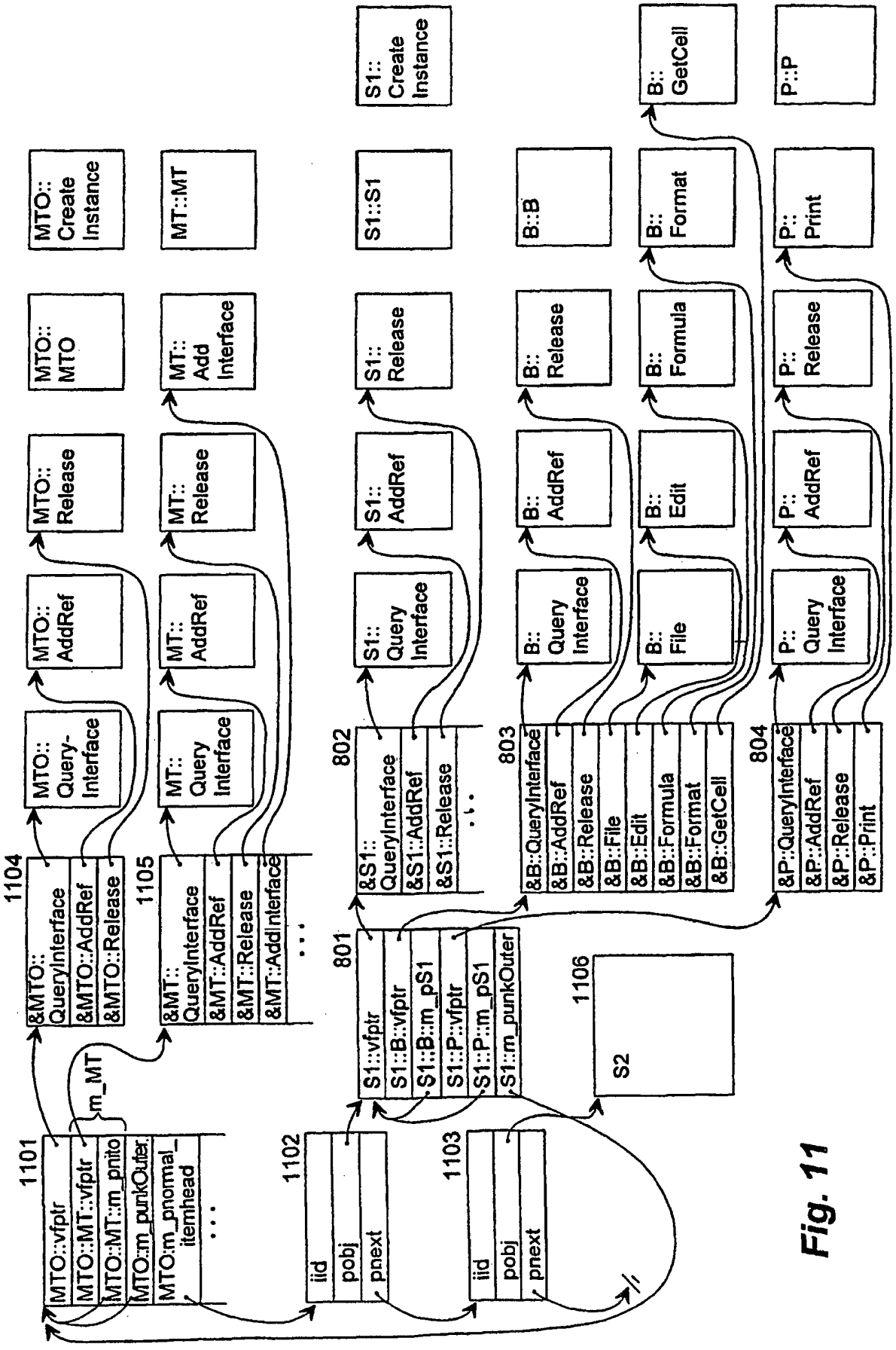
1101
MTO::vfptr
MTO::MT::vfptr
MTO::MT::m_pnito
MTO::m_punkOuter
MTO::m_pnormal_itemhead
m_MT
1102
iid
pobj
pnext
1103
iid
pobj
pnext
1104
&MTO::QueryInterface
&MTO::AddRef
&MTO::Release
1105
&MT::QueryInterface
&MT::AddRef
&MT::Release
&MT::AddInterface
801
S1::vfptr
S1::B::vfptr
S1::B::m_pS1
S1::P::vfptr
S1::P::m_pS1
S1::m_punkOuter
802
&S1::QueryInterface
&S1::AddRef
&S1::Release
803
&B::QueryInterface
&B::AddRef
&B::Release
&B::File
&B::Edit
&B::Formula
&B::Format
&B::GetCell
804
&P::QueryInterface
&P::AddRef
&P::Release
&P::Print
1106
S2
MTO::Create Instance
MTO::MTO
MTO::Release
MTO::AddRef
MTO::Query-Interface
MT::MT
MT::Add Interface
MT::Release
MT::AddRef
MT::Query Interface
S1::S1
S1::Release
S1::AddRef
S1::Query Interface
S1::Create Instance
B::B
B::Release
B::AddRef
B::Query Interface
B::Format
B::Formula
B::Edit
B::File
B::GetCell
P::P
P::Print
P::Release
P::AddRef
P::Query Interface

Fig. 11

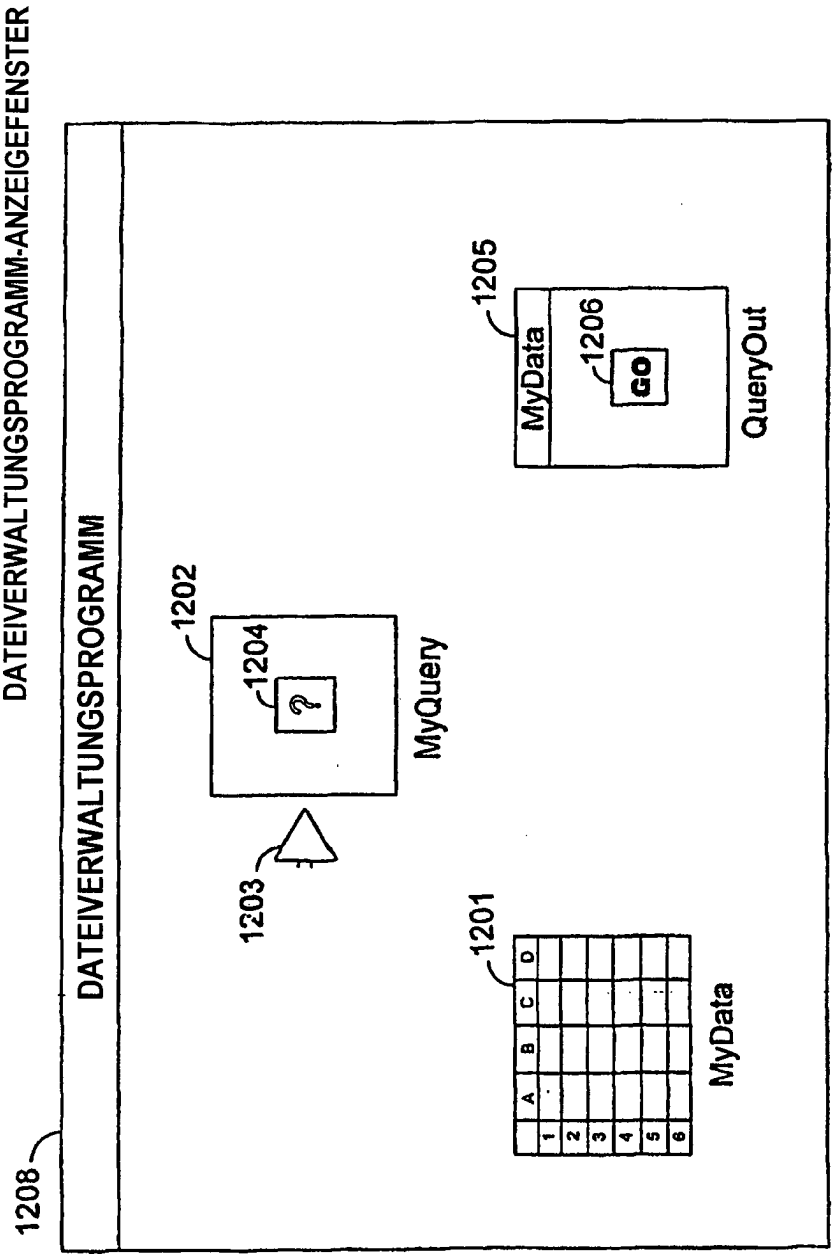
DATEIVERWALTUNGSPROGRAMM-ANZEIGEFENSTER
DATEIVERWALTUNGSPROGRAMM
1208
1201
A
B
C
D
1
2
3
4
5
6
MyData
1202
1204
?
MyQuery
1203
1205
MyData
1206
GO
QueryOut

Fig. 12

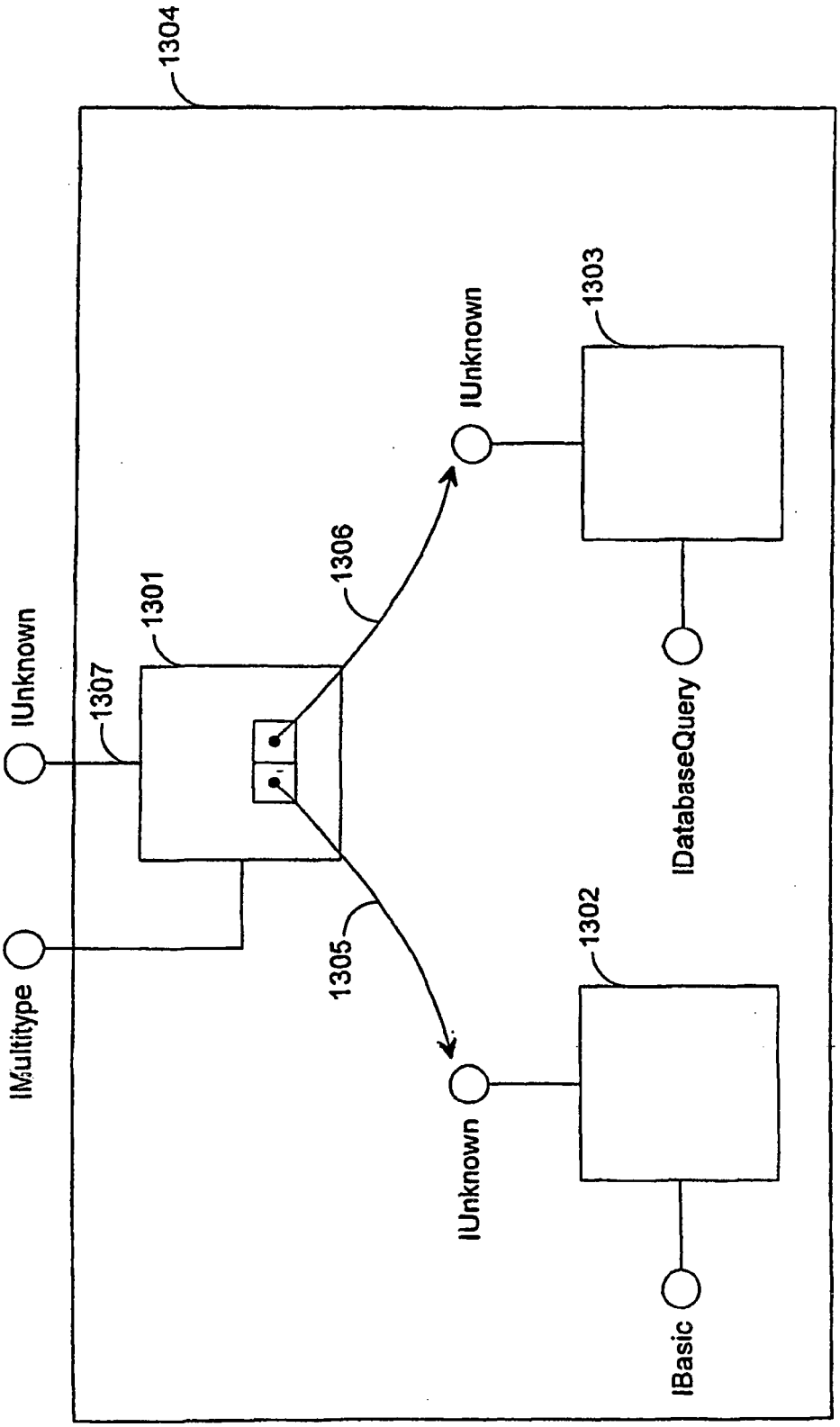
1304
IUnknown
1303
IUnknown
1306
1301
1307
IUnknown
IDatabaseQuery
IMultitype
1305
1302
IUnknown
IBasic

Fig. 13