Dynamische Methodenaufrufe in Corba 2.0

Tim Paehler

24. Juni 1997

Inhaltsverzeichnis

**1 Einführung:
Der Standard-Methodenaufruf in Corba 2**

2 Mechanismen für den dynamischen Methodenaufruf 3

2.1 Der Naming Service (NS) 5
2.2 Das Interface Repository (IR) 6
2.3 Das Dynamic Invocation Interface (DII) 10

1 Einführung: Der Standard-Methodenaufruf in Corba

Als Referenzentwurf eines verteilten Systems legt Corba das Verfahren eines Methodenaufrufs und die Schnittstellen der dabei teilnehmenden Objekte (Client, Server sowie der ORB als Middleware) fest.

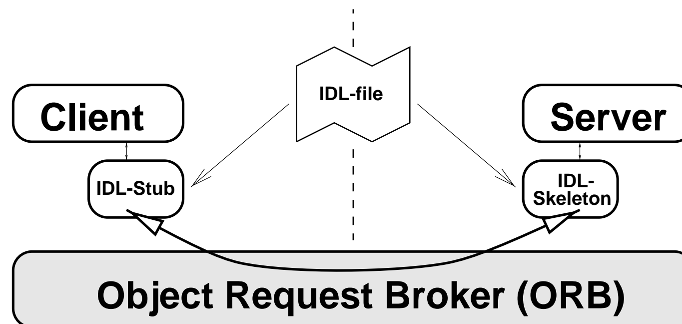

Abb.1 Standard-Methodenaufruf in Corba

Client und Server enthalten nach der Corba-Spezifikation je ein durch den IDL-Compiler generiertes Stück Programmcode als Aufsatz: Der IDL-Stub für den Client und der IDL-Skeleton für den Server.
Die Aufgabe des Stubs und des Skeletons besteht nun darin, die im abstrakten IDL-Code beschriebene Schnittstelle in die jeweils verwendete Programmiersprache umzusetzen.
In einer C++-Implementation würde dies zum Beispiel bedeuten, daß eine Datei `hello_world.idl` durch den IDL-Compiler in `hello_worldC.cc` (den Client-Stub) und `hello_worldS.cc` (den Server-Skeleton) sowie eine gemeinsam verwendete Headerdatei `hello_world.hh` erzeugt wird (Abb.2). Diese Dateien werden dann auf Client- und Serverseite in den Programmcode eingebunden.

In der laufenden verteilten Anwendung ist es nun Aufgabe des Stubs und des Skeletons, über Kommunikation mit dem ORB den entfernten in einen lokalen Methodenaufruf umzuwandeln. Das bedeutet konkret, daß der Stub dem Client alle Methoden der Serverimplementation anbietet, weshalb man ihn auch als Proxy bezeichnet.

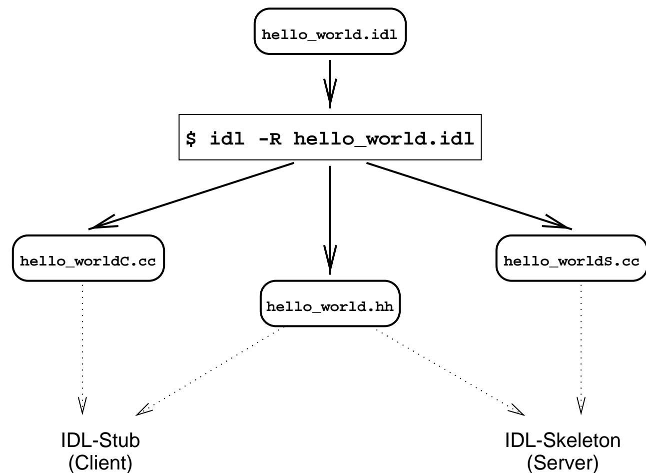

Abb.2 Stub- und Skeletongenerierung in C++

Durch die Definition der Schnittstelle in IDL und die Bereitstellung der aus ihr generierten Programmcodestücke wird damit erreicht, daß sich die Programmierung von Client und Server völlig unabhängig voneinander gestalten läßt. Man könnte also nach der Festlegung der Schnittstelle die Implementation der beiden Seiten verschiedenen Entwicklerteams zuweisen.

Von der Clientseite her ergibt sich damit allerdings folgender Nachteil: Der Client muß von Beginn an "wissen", welche Methoden die Serverimplementation bereitstellt, die Aufrufe können also nur statisch ausgeführt werden. Aus diesem Grund wurden der Corba-Spezifikation Mechanismen zur Unterstützung dynamischer Aufrufe beigefügt. Diese sind im wesentlichen das Interface Repository (IR) und das Dynamic Invocation Interface (DII).

2 Mechanismen für den dynamischen Methodenaufruf

Die Notwendigkeit, einen Methodenaufruf dynamisch zu gestalten, ist in vielen Szenarien denkbar. Konkret könnte man sich etwa ein Programm vorstellen, mit dem sich die einzelnen Methoden einer Serverimplementation interaktiv anzeigen lassen, und dann diese durch Aufruf antesten. Ein solches Programm würde ein leistungsfähiges Debug-Werkzeug darstellen. Ein erster Schritt zu einem solchen Tool soll im praktischen Teil meiner Arbeit durch das Itool aufgezeigt werden.

An diesem Beispiel wird jedoch schon klar, wie ein solches Programm arbeiten würde und welche Mechanismen die Corba-Implementation bereitstellen muß:

- Es muß zunächst eine Institution geben, die über die vorhandenen Server informiert und auf Anfrage eine Objektinstanz zurückgibt.
- Hat man nun eine Referenz auf einen Server, so benötigt man in der Regel Informationen über die von diesem Objekt bereitgestellten Methoden. Da dabei auch die notwendigen Typ- und Modulinformationen benötigt werden, ist es sinnvoll, die Daten aller bereitgestellten Schnittstellen in einer zentralen "Auskunftsstelle" zu sammeln. Diese müßte damit alles an Informationen anbieten, was sich im statischen Fall in den einzelnen IDL-Dateien der Objekte befindet.
- Es muß nun schließlich möglich sein, aus den Informationen der Auskunftsstelle einen gültigen Methodenaufruf zu konstruieren und die dazugehörigen Rückgabeparameter auszulesen.

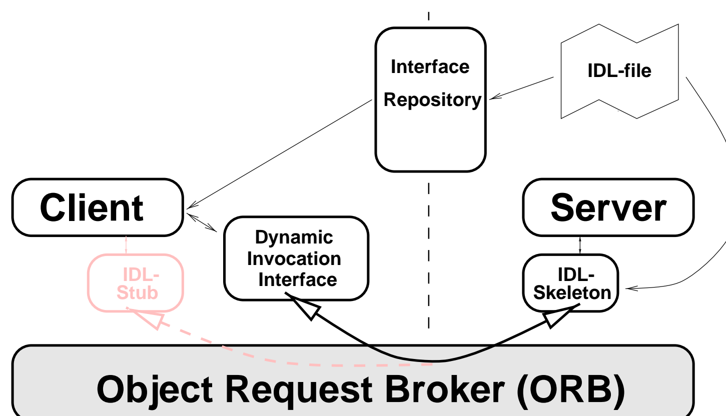

Abb.3 Dynamischer Methodenaufruf in Corba: An die Stelle des Stubs treten das IR und das DII

Tatsächlich werden die beiden Punkte von verschiedenen Einrichtungen in der Corba-Installation übernommen: Der Naming Service ordnet dem Servernamen ein konkretes Objekt zu, das Interface Repository fungiert als Schnittstellendatenbank und das Dynamic Invocation Interface sorgt schließlich für die Abarbeitung dynamischer Funktionsaufrufe bei einem vorgegebenen Objekt. Wir wollen diese drei Einrichtungen nun genauer betrachten.

2.1 Der Naming Service (NS)

Der Naming Service soll hier nur kurz angesprochen werden, eine ausführlichere Beschreibung findet sich in [ON96]. Bei ihm handelt es sich um einen extern eingebundenen Dienst, der sogenannte *COSS-Naming*, dessen Schnittstelle von der OMG durch die Definition in IDL-Code festgesetzt wurde, dessen Implementierung jedoch wiederum den ORB-Herstellern überlassen wurde.

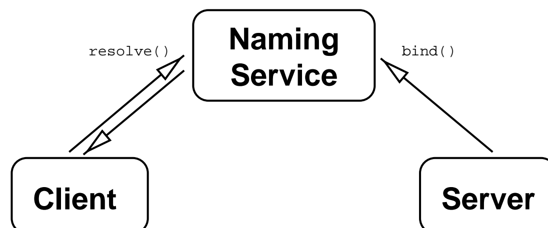

Abb.6 Die Funktion des Namensdienstes: Der Server meldet sich mittels bind() beim Naming Service an, der Client erhält via resolve() eine Referenz auf den Server

In Analogie zu in Rechnernetzen verwendeten Namensdiensten wie etwa DNS oder NIS nimmt der Naming Service eine eindeutige Abbildung von Namen auf Objekte vor. Am hilfreichsten ist es wahrscheinlich, sich den NS als eine Art hierarchisches Meta-Dateisystem vorzustellen, in dem die Objekte die Rolle der Dateien in einem normalen Dateisystem übernehmen. Die Rolle der Verzeichnisse besetzen dabei sogenannte `NamingContexts`, deren Einordnung in den Namensraum durch die Methode

```
NamingContext::bind_context(in Name n, in NamingContext nc);
```

bewerkstelligt wird. Die Objektnamen selbst werden durch die Methode

```
NamingContext::bind(in Name n, in Object o);
```

erzeugt. Dabei stellt `Name` eine Sequenz aus Namenskomponenten dar, die die hierarchische Benennung des Objektes realisiert (z.B. würde der Objekt-Name "/objects/finance/bankingobject" in einer Variable vom IDL-Datentyp `Name`, also in einer `Sequence<NameComponent,3>` gespeichert, wobei die Zeichenketten der drei Komponenten die Werte "objects", "finance" und "bankingobject" enthielten). Die zentrale Methode bei der Objektsuche in einem Namenskontext ist die Funktion

```
Object NamingContext::resolve(in Name n);
```

die auf eine Namensanfrage das zugehörige Objekt zurückgibt oder eine Exception vom Typ `NotFound`, `CannotProceed` oder `InvalidName` zurückgibt.

2.2 Das Interface Repository (IR)

Das Interface Repository ist ein Bestandteil des ORB und dient als Archiv für IDL-Definitionen, ähnlich einer Datenbank. Sowohl das IR selbst, als auch die in ihm enthaltenen Einträge werden dabei durch Corba-Objekte repräsentiert, so daß die Abfrage des IR im Rahmen der üblichen verteilten Methodenaufrufe geschieht.
Da das IR auf der Client-Seite den IDL-Stub hinsichtlich der vollständigen Beschreibung der IDL-Definition ersetzen soll, muß es folglich alle Informationen einer IDL-Datei bereitstellen. Damit bietet es sich an, den Aufbau des IRs in Analogie zu einer solchen Datei zu wählen.

Betrachten wir also den Aufbau einer IDL-Datei an einem Beispiel:

```
//IDL
module DataObjectOperations {
    typedef long objectindex;
    exception noPermissions{};

    interface DataObject1Operations : BaseOperations {
        <Typ-Definitionen>
        <Konstanten-Definitionen>
        <Exception-Definitionen>
        <Attribut-Definitionen>
        [ <Resultattyp> ] <Operationsname> ( <Argumente> )
        raises( noPermissions );
        // ... weitere Operations-Definitionen
    };
    // ... weitere Interface-Definitionen
    // ... weitere Modul-Definitionen
};
```

Es fällt dabei nun auf, daß wie in vielen anderen Programmiersprachen IDL-Definitionen in anderen Definitionen enthalten sein können. Setzt man

diese 'Enthalten-sein-in'-Relation in ein Objektmodell um, so ergibt sich für die Anordnung der Einträge im IR ein hierarchischer Aufbau, in dem Objekte die Bezeichnungen `Container` und `Contained` besitzen können (s. Abb.4), für einige Objekte im IR gilt sogar beides: Eine Moduldefinition `ModuleDef` ist einerseits ein `Container`, da es z.B. Interface- und Type-Definitionen sowie weitere Modulbeschreibungen beinhalten kann, andererseits ist es selbst im `Repository`-Objekt oder einem anderen Modul enthalten.

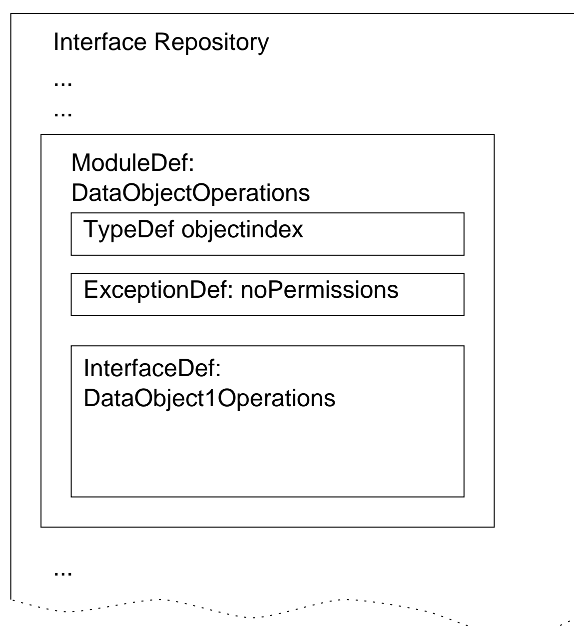

Abb.4 Einträge im Interface Repository: Die Definitionen werden als Container betrachtet

Da das Interface Repository nun transparent in die Objektumgebung von Corba 2.0 eingebettet werden soll, müssen folglich das `Repository`-Objekt selbst und die Eigenschaften der in ihm enthaltenen Einträge in IDL spezifiziert werden. Dies geschieht in der vom Corba 2.0-Standard hergeleiteten Datei `CORBA2.0/ir.idl` [Re96]. Die Tatsache, daß allein die IDL-Definitionen des IR von der OMG standardisiert sind, erlaubt es dabei wiederum verschiedenen Softwareherstellern, mit eigenen Implementationen des IRs in Konkurrenz zueinander zu treten, ohne dem Standard dabei einen Abbruch zu tun. In der Praxis sieht dies jedoch leider noch anders aus: Das IR des in den Beispiel-Programmen benutzten Orbix-ORB verwendet eigene IDL-Definitionen, die in einigen Punkten von der OMG-Spezifikation abweichen.

Wir wollen nun etwas konkreter auf die IDL-Struktur des Interface Repository und seiner Objekte eingehen. Die oben angesprochene Enthalten-sein-Relation wird durch Vererbung der Klassen `Container` und `Contained` ausgezeichnet, die ihrerseits wieder das Basisinterface `IRObject` erben (s. Abb.5).

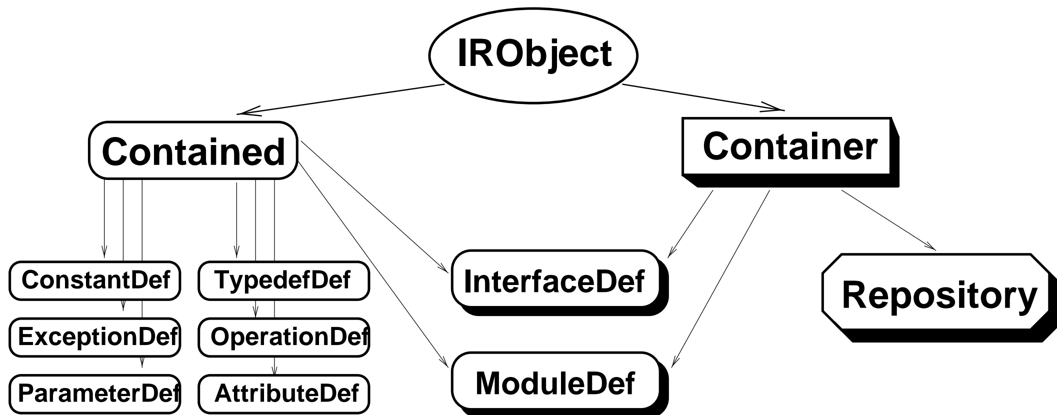

Abb.5 Die Vererbungshierarchie der IR-Objekte

Dabei stellt der `Container` z. B. Methoden zum Auffinden von enthaltenen Objekten sowie Listen über seinen Inhalt zur Verfügung, während die `Contained`-Schnittstelle eine einheitliche Beschreibung des gesuchten IR-Objektes bereitstellt. Die von diesen Basisschnittstellen abgeleiteten Definitionen sorgen nun für die spezifischen Beschreibungen der IR-Objekte. Betrachten wir z.B. die Corba-Spezifikation von `InterfaceDef` in `ir.idl`:

```
//IDL
interface InterfaceDef : Container, Contained, IDLType {
   attribute InterfaceDefSeq base_interfaces;

   boolean is_a(in RepositoryId interface_id);

   struct FullInterfaceDescription  {
Identifier name;
RepositoryId id;
RepositoryId defined_in;
VersionSpec version;
OpDescriptionSeq     operations;
AttrDescription attributes;
RepositoryIdSeq base_interfaces;
```

```
TypeCode type;
};

   FullInterfaceDescription describe_interface();
   ...
}
```

Trägt man durch Idl-Kompilieren unter Verwendung des Compilerflags `-R` ein Interface in das IR ein, so wird ein Objekt vom Typ `InterfaceDef` angelegt und in das `Repository`-Objekt gelegt. Nun lassen sich die Informationen dieses Eintrags auf zwei verschiedene Arten abfragen: Einerseits kann man durch Kommunikation mit dem `Repository`-Objekt über die Container-Methode `lookup_name()` eine Referenz auf den Eintrag erlangen, zum anderen kann man, wenn man bereits von der Serverseite ein Implementationsobjekt zu diesem Interface erhalten hat, in diesem Objekt die Methode `_get_interface()` aufrufen, welche ebenfalls den Eintrag liefert. Im ersten Fall könnte dies z.B. in Java für das interface `testif` konkret folgendermaßen aussehen[Io96]:

```
//Java
  import IE.Iona.Orbix2.*;
  import IE.Iona.Orbix2.CORBA.*;
...
  try {
    String id;
    // Repository-Referenz besorgen:

    _RepositoryRef rep = Repository._bind(":IR","deeppurple.uni-muenster.de");

    // rep in _ContainerRef umwandeln, um auf dessen Methoden zugreifen zu
    // koennen:

    _ContainerRef container = Container._narrow(rep);

    // Interface-Definition fuer "testif" suchen und einer Variable vom Typ
    // ContainedRef zuweisen:

    _ContainedRef contd = container.lookup_name("testif",10,"all",true).buffer[0];

    // contd in _InterfaceDefRef umwandeln:
```

```
_InterfaceDefRef if = interfaceDef._narrow(contained);

// Genaue Beschreibung des Interface extrahieren:

FullInterfaceDescription fulld = if.describe_interface();
...

} catch (SystemException e){}
```

Ein ähnliches Verfahren wird im Beispielprogramm `Itool.java` zum Durchsuchen des Interface Repository benutzt.
Damit ist klar, wie wir an die Informationen über unbekannte Interfaces gelangen können. Im folgenden wollen wir uns mit den Mechanismen des dynamischen Funktionsaufrufs beschäftigen.

2.3 Das Dynamic Invocation Interface (DII)

Wie man aus dem Rückgabetyp der `resolve()`-Methode des Naming Service (s. 2.1) bereits erkennen kann, hat man es bei dynamisch ermittelten Objekten mit Instanzen der Basisklasse `CORBA::Object` zu tun. Das bedeutet, daß die Methoden für den dynamischen Funktionsaufruf in dieser oder zumindest durch diese Klasse bereitgestellt werden müssen. Wir wollen nun in einem Beispiel den dynamischen Aufruf konkret darstellen. Betrachten wir dabei zunächst den einfachsten Fall des Methodenaufrufs ohne Parameter: In einer bereits erhaltenen Objektreferenz `objRef` soll die Methode `test_method();` aufgerufen werden.

```
//Java
...
_ObjectRef objRef = ...

Request request = objRef._request("test_method");

request.invoke();
...
```

Wie man an diesem Beispiel sieht, muß für den dynamischen Aufruf lediglich ein Pseudo-Objekt der Klasse `Request` erzeugt werden, in dem schließlich die `invoke()`-Methode aufgerufen wird. Wie verfährt man nun mit der Übergabe von Parametern an den Funktionsaufruf? Es erscheint sinnvoll, etwas wie

eine Parameterliste zu erstellen, in der die einzelnen Argumente zusammen mit ihrer Typinformation und ihrem Aufrufmodus (`in`, `out`, `inout`) enthalten sind. Genau dies wird durch die Klassen `NVList` und `NamedValue` bereitgestellt. Wir wollen zunächst am Beispiel einer Java-Implementation den dynamischen Funktionsaufruf von

```
string test_method2(in int iarg, in string sarg);
```

mit den Werten `iarg = 42` und `sarg = "bla"` betrachten:

```
//Java
...
_ObjectRef oRef = ...

Request request = oRef._request("test_method");

//NVList fuer Request erzeugen
NVList arglist = request.arguments();

//Argumente in NVList einfuegen
Any arg1 = new Any(),
arg1.insertLong(42);
arglist.add_value("iarg", arg1, _CORBA.ARG_IN);

Any arg2 = new Any();
arg2.insertString("bla");
arglist.add_value("sarg", arg2, _CORBA.ARG_IN);

request.invoke();

request.extractOutParams();
String res = request.result();
...
```

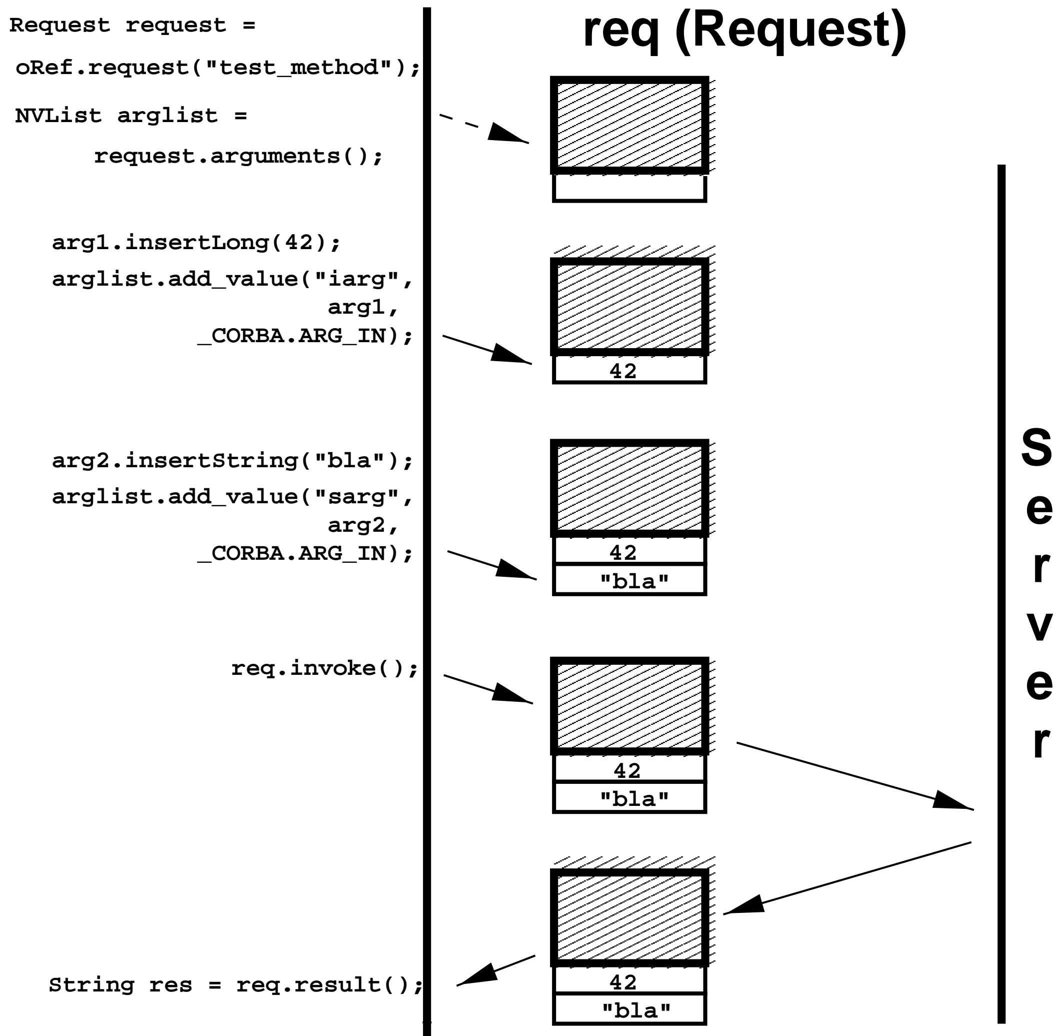

Abb.6 Ablauf eines dynamischen Methodenaufrufs: Zunächst wird ein Request-Objekt erschaffen, dann der Parameterliste die jeweiligen Werte und Aufrufmodi hinzugefügt.

Wie man erkennen kann, werden die Argumente direkt in die Liste vom Typ `NVList` eingefügt, wo sie in einem `NamedValue` gespeichert werden (dies ist auch der Rückgabetyp der `add_value()`-Operation). Der `NamedValue` selbst tritt lediglich dann explizit in Erscheinung, wenn Parameter bearbeitet oder extrahiert werden sollen, Dies ist zum Beispiel bei `out`- und `inout`-Werten nach dem Funktionsaufruf der Fall. Wie beim Interface Repository

und Naming Service wird auch die Implementation des DII in Corba 2.0 dem ORB-Hersteller überlassen, die Standard-Konformität jedoch durch die Spezifikation der Objekte in pseudo-IDL sichergestellt. Diese wollen wir als Dokumentation der Funktionalität der Klassen `NVList` und `NamedValue` betrachten:

```
//IDL
module CORBA {
  ...
  pseudo interface NamedValue {
        readonly attribute Identifier  name;
        readonly attribute any  value;
        readonly attribute Flags flags ;
  };

  pseudo interface NVList {
        readonly attribute unsigned long count;

        NamedValue add_item(
                      in Identifier item_name,
                      in Flags flags );

        NamedValue add_value(
                      in Identifier item_name,
                      in any  val,
                      in Flags flags );

        NamedValue item(in unsigned long index) raises (Bounds);

        NamedValue remove(in unsigned long index) raises (Bounds);
  };
  ...
};
```

Es gilt also Folgendes: Einer `NVList` kann durch den Aufruf der Methode `add_item()` oder `add_value()` ein Parameter hinzugefügt werden, der mittels der Methode `item()` abgefragt werden kann. Der so erhaltenen `NamedValue` läßt sich dann durch Abfrage der Attribute `name`, `value` und `flags` abfragen.

Verändern läßt sich ein bereits eingefügter `NamedValue` nicht, er muß mittels `NVList::remove()` aus der Liste entfernt und durch einen neuen ersetzt werden.

Wir wollen nun noch einen Blick auf die verschiedenen Aufrufoptionen eines dynamisch erzeugten `Requests` werfen. Dazu betrachten wir zunächst wieder die im Standard festgelegte IDL-Definition:

```
// IDL
  pseudo interface Request {
        readonly attribute Object target;
        readonly attribute Identifier operation;
        readonly attribute NVList arguments;
        readonly attribute NamedValue result;
        readonly attribute Environment env;

        attribute Context ctx;

        Status invoke();
        Status send_oneway();
        Status send_deferred();
        Status get_response();
        boolean poll_response();
      };
```

Wie man sieht, befinden sich unter den Methoden neben dem normalen `invoke()`-Aufruf weitere Operationen, die einen asynchronen Aufruf unterstützen. Bei dieser Aufrufsart wird der `Request` zunächst via `send_deferred()` auf den Server übertragen. Später können dann durch Aufruf der Methode `poll_response()` und `get_response()` der Rückgabewert und die `inout`/`out`-Parameter ausgelesen werden, um diese weiter zu verarbeiten. Auf diese Weise ist es z.B. möglich, die Wartezeit bis zur Ausgabe des Ergebnisses des Methodenaufrufs vom Server durch weitere Aktionen auf Seiten des Clients auszufüllen oder mehrere konkurrierende Methodenaufrufe zu starten und das schnellste Ergebnis auszuwerten. Enthält der Funktionsaufruf keinen Rückgabewert oder `inout`/`out`-Parameter, so bietet sich die Verwendung von `send_oneway()` an. Bei dieser Aufrufsart wartet der Client nicht auf das Ende der Operation, wodurch ihm jedoch auch der Erfolg des Aufrufs im unklaren bleibt.

Literatur

[Re96] Jens Peter Redlich. Corba 2.0 - Praktische Einführung für C++ und Java. Addison-Wesley 1996

[Or97] Robert Orfali, Dan Harkey Client/Server Programming with Java and CORBA. John Wiley and Sons 1997

[Io96] OrbixWeb Programming Guide, OrbixWeb Reference Guide. IONA Technologies Ltd. 1996

[ON96] OrbixNames Guide. IONA Technologies Ltd. 1996