12. Konfiguration CORBA-basierter Applikationen

CORBA *(Common Object Request Broker Architecture)* ist eine objektorientierte Middleware für verteilte heterogene Systeme, die von der Object Management Group entworfen und spezifiziert wurde [OMG02a]. Es ermöglicht Klienten entfernte Dienstanbieter wie lokale Objekte zu benutzen, wobei Klient und Dienstanbieter in unterschiedlichen Programmiersprachen implementiert sein können. Die Schnittstellen von CORBA-Objekten werden mit der Schnittstellenbeschreibungssprache CORBA-IDL spezifiziert, die im folgenden abgekürzt nur IDL genannt wird.

IDL bietet zusammengesetzte Typen, eingeschränkte Typpolymorphie und die Möglichkeit abstrakte Datentypen, die von dem Anwender erstellt wurden, als Wert zu übergeben. Desweiteren bietet CORBA vielfältige Möglichkeiten die konkrete Implementierung eines CORBA-Objektes mit der fernaufrufbaren Schnittstelle zu verbinden und ermöglicht so z.B. persistente Objekte oder eine automatische Lastverteilung. Die OMG hat zusätzlich etliche Standarddienste zur Verwaltung, Lokalisation etc. von verteilten Objekten spezifiziert, welche die Erstellung verteilter Applikationen vereinfachen.

Die erste CORBA-Version 1.1 wurde 1991 veröffentlicht und seitdem kontinuierlich verbessert und erweitert. Die Implementierungen sind dementsprechend ausgereift und CORBA hat sich zu einem vielfältig eingesetzten Industriestandard entwickelt [COR02a]. Aufgrund des enormen Funktionsumfangs ist die Einarbeitung in das System aufwendig. Zusätzlich ist die Abbildung der IDL auf die eingesetzte Programmiersprache aufgrund der Unterstützung heterogener Programmiersprachen nicht immer nahtlos, so dass sich der Anwender mit technischen Details zu beschäftigen hat.

Daher ist eine Unterstützung des Entwicklers wünschenswert, die es ihm ermöglicht, sich auf die Implementierung der CORBA-Objekte und auf die Erstellung der verteilten Applikation durch Konfiguration der CORBA-Objekte zu konzentrieren. Es sollten ihm alle Möglichkeiten von CORBA zur Verfügung stehen, ohne dass er sich mit den technischen Details beschäftigen muss, sofern dieses möglich ist. In diesem Kapitel wird ein Erweiterungsmodul vorgestellt, welches versucht, diesem Ziel möglichst nahe zu kommen.

Um eine möglichst große Anzahl von Applikationsklassen zu unterstützen, setzt dieses Modul auf zwei abstrakten Erweiterungsmodulen auf:

- Verteilte Informations- und Kontrollsysteme (VIKS, Kapitel 11)
- Arbeitsablaufbasierte Applikationen (Kapitel 6)

Der Einsatz des VIKS-Moduls deckt die typischen meist großen und langlaufenden CORBA Applikationen ab. Bei diesen werden CORBA-Objekte an unterschiedlichen Orten gestartet und miteinander zu einer Applikation verbunden, die vielfältige Benutzerschnittstellen haben kann. Ein Beispiel sind Informationssysteme, in denen mehrere Datenbanken und grafische Terminals interagieren. Derartige Applikationen besitzen i.a. zusätzlich einen dynamischen nichtdeterministischen Charakter in dem Sinne, dass spontan neue Dienstanbieter oder Klienten in das System integriert werden.

Eine andere Applikationsklasse, in der CORBA eingesetzt wird, umfasst Adapterapplikationen. Hier interagiert eine Applikation aktiv mit CORBA-Objekten und vermittelt so zwischen diesen. Ein Beispiel hierfür ist eine Protokollierungsapplikation, die zyklisch den Zustand verteilter Objekte abfragt und diesen unter Einsatz eines speziellen Protokollierungsdienstes sichert, welcher wiederum von einem speziellen CORBA-Objekt erbracht wird.

Für dieses CORBA-Erweiterungsmodul müssen nun die abstrakten Architekturelemente der beiden obigen Erweiterungsmodule konkretisiert werden. Hierfür werden zunächst die technischen Eigenschaften von CORBA eingeführt, die von diesem Modul unterstützt werden. Daraufhin wird in Kapitel 12.2 dann das Architekturvokabular mit der entsprechenden Umsetzung und Laufzeitunterstützung präsentiert. In Kapitel 13 werden dann einige exemplarische CORBA-Anwendungen vorgestellt, die dieses Erweiterungsmodul einsetzen.

12.1. CORBA Grundlagen

In diesem Kapitel werden die technischen Grundlagen von CORBA vorgestellt, die einem Applikationsentwickler bekannt sein sollten, wenn er komplexere CORBA-Applikationen entwickeln möchte. Zusätzlich wird die Terminologie des CORBA-Systems eingeführt, welche dann in dem Architekturvokabular des Erweiterungsmoduls benutzt wird.

Wie einführend schon erwähnt, ermöglicht es CORBA, entfernte Dienstanbieter über einen lokalen Stellvertreter wie entfernte Objekte zu benutzen, die eine in IDL spezifizierte Schnittstelle besitzen und über eine fast eindeutige Referenz[1], der *Inter-Object-Reference (IOR)*, ansprechbar sind. Ein CORBA-Objekt ist konzeptionell von dessen Implementierung zu trennen. Die Zuordnung von Referenz zu Implementierung geschieht dynamisch und kann sich während der Lebenszeit eines CORBA-Objekts auch ändern.

Zur Verdeutlichung stellt Abbildung 12.1 schematisch einen möglichen Ablauf eines entfernten Methodenaufrufs dar. Bevor dieser Aufruf stattfinden kann, muss der Klient sich eine Referenz auf das aufzurufende CORBA-Objekt besorgen. Diese enthält u.a. den Typ der IDL-Schnittstelle des Objekts und Informationen, wie dessen Implementierung zu lokalisieren ist. Aus diesen In-

[1] Aufgrund von Freiheiten in der Implementierung von CORBA-Systemen wird in der CORBA-Spezifikation nur gefordert, dass man bei zwei beliebigen Referenzen stets entscheiden kann, ob sie unterschiedlichen Objekten zugeordnet sind. Die Entscheidbarkeit, dass zwei Referenzen auf das gleiche Objekt verweisen, wird dagegen nicht garantiert.

formationen wird im Adressraum des Klienten von dessen CORBA-System ein Stellvertreter *(Stub)* erzeugt.

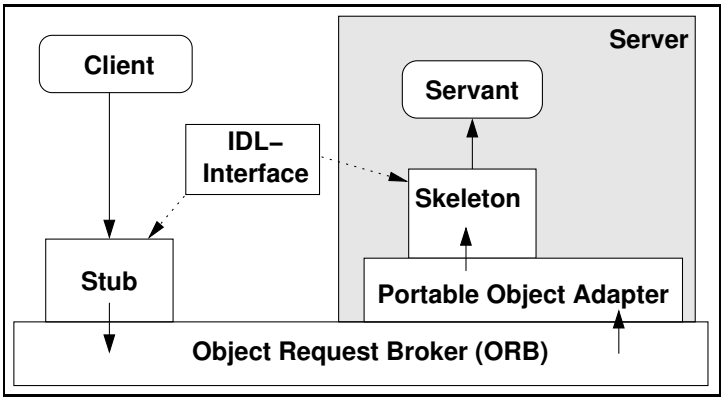

Abbildung 12.1.: Schematischer Überblick eines entfernten Methodenaufrufs in CORBA

Über diesen Stellvertreter lässt sich das entfernte Objekt fast wie ein lokales Objekt benutzen. Hierfür delegiert der Stellvertreter einen Methodenaufruf an die Implementierung des CORBA-Objekts unter Einsatz des *Object-Request-Brokers (ORB)*. Dieser verbindet konzeptionell alle Klienten mit ihren CORBA-Objekten. Konkret lokalisiert die ORB-Komponente in dem CORBA-Laufzeitsystem des Klienten den Trägerprozess der Implementierung des CORBA-Objekts, der in der CORBA-Spezifikation als *Server* bezeichnet wird.

Ein Server kann unterschiedliche Implementierungen für verschiedene CORBA-Objekte enthalten. Die Zuordnung eines Methodenaufrufs, der vom ORB dem Server übergeben wurde, an die richtige Implementierung wird von dem *Portable Object Adapter (POA)* erbracht. Da dem POA der Methodenaufruf in komprimierter binärer Form übergeben wird, ist dieser vor der Übergabe an die Implementierung zu entpacken. Nun kann die eigentliche Implementierung, der sogenannte *Servant*, mit den Parametern aufgerufen werden und die Ergebnisse an den Klienten zurückgeschickt werden.

Diese letzten Schritte werden von einem Treiber *(Skeleton)* ausgeführt, der aus der IDL-Schnittstelle automatisch erzeugt wurde. Wie die Anbindung des Servant an das Skeleton realisiert ist, hängt von der Programmiersprache ab, in welcher der Servant geschrieben wurde. Bei objektorientierten Sprachen, wie Java oder C++, werden i.a. zwei Ansätze unterstützt. Zum einen kann das Skeleton als abstrakte Oberklasse zur Verfügung stehen. Der Servant muss dann nur von dieser erben. Alternativ kann das Skeleton ein konkretes Objekt sein, dem der Servant bei der Konstruktion übergeben wird, so dass er Aufrufe an diesen delegieren kann.

Die hier skizzierte Zuordnung zwischen Referenz und Servant ist allerdings für etliche Anwendungen unzureichend. Wenn man beispielsweise sehr viele CORBA-Objekte von einem Server tragen lassen möchte, dann kann es effizienter sein, die Servants nur dann im Arbeitsspeicher zu halten, wenn diese auch benötigt werden. Bei langlaufenden Applikationen sind zusätzlich persistente Objekte erforderlich. Eine Bedingung für Persistenz ist, dass Server

terminiert und neu hochgefahren werden können, ohne dass entfernt gespeicherte CORBA-Referenzen ungültig werden.

Eine andere Anforderung, die aus dem Bereich der Datenbank-basierten Applikationen stammt, besteht darin, dass verschiedene CORBA-Objekte einem Servant zugeordnet werden. Dies ermöglicht es bei vertretbarem Aufwand, dass jedem Eintrag in einer Datenbank ein CORBA-Objekt zugeordnet wird, welches Zugriff auf die Daten ermöglicht. Diese möglicherweise sehr große Menge von CORBA-Objekten sind dann nur einem Servant zugeordnet, welche den Zugriff auf die Datenbank durchführt. Da nun unabhängig von der Größe der Datenbank stets nur ein programmiersprachliches Objekt vorhanden sein muss, ist dieser Ansatz skalierbar.

Um diese Anforderungen angemessen erfüllen zu können, wurde der POA mit vielfältigen Parametrisierungsmöglichkeiten ausgestattet. Diese werden im folgenden kurz angerissen. Für eine umfassende Beschreibung der Möglichkeiten sei auf [BVD01, HV99] verwiesen.

Ein Server kann verschiedene POAs mit jeweils unterschiedlichen Konfigurationen enthalten. Diese sind hierarchisch als Baum angeordnet, wobei die Wurzel mit dem Namen `RootPOA` in jedem Server immer vorhanden ist. Jeder POA hat einen Namen, der im Kontext seines übergeordneten POAs eindeutig ist. Abbildung 12.2 zeigt ein Beispiel für eine POA Konfiguration.

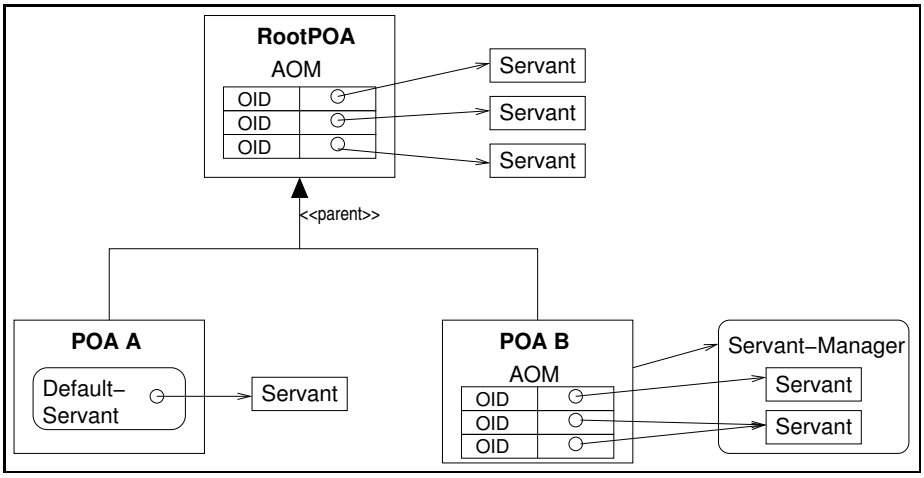

Abbildung 12.2.: Beispiel einer Konfiguration multipler POAs

Um einen Servant zu lokalisieren, erhält der POA vom ORB einen Objektbezeichner, der in der CORBA-Referenz enthalten ist. Der Inhalt und Aufbau dieses Objektbezeichner wird frei von dem POA bestimmt, der die Objektreferenz erzeugt hat. Um nun die Zuordnung zwischen Objektbezeichner und Servants zu verwalten, gibt es für einen POA zwei Ansätze:

- Alle eingehenden Methodenaufrufe werden an einen Standardservant *(Default-Servant)* übergeben, der diese dann bearbeitet. Dieser wird von dem Applikationsentwickler bereitgestellt und verwaltet damit selbstständig alle Servants für diesen POA.

- Alle aktiven Servants werden in einer *Active-Object-Map (AOM)* gespeichert. Wenn der POA diese AOM selbst verwaltet, wird die statische Zuordnung zwischen Referenz und Servant realisiert, die in dem obigen Beispiel eines entfernten Methodenaufrufs skizziert wurde. Zusätzlich ist es aber auch möglich, einen von dem Applikationsentwickler bereitgestellten Servant-Verwalter (*Servant-Manager*) zu registrieren. Dieser stellt dann eine weitere Indirektionsstufe für den Zugriff auf die AOM dar, und ermöglicht es z.B. Servants zu erzeugen und in der AOM zu registrieren, wenn sie noch nicht existent sind.

Es ist zudem möglich, beide Ansätze zu kombinieren, indem auf den Standardservant zurückgegriffen wird, wenn ein Servant nicht in der AOM zu finden ist.

Zusätzlich zu diesen Möglichkeiten des applikationsseitigen Zugriffs auf CORBA-Elemente, gibt es einige Standarddienste, die Applikationsentwickler i.a. benötigen. Zu den beiden wichtigsten zählen der Namens- und der Vermittlungsdienst. Der Namensdienst verwaltet die Zuordnung zwischen hierarchisch zusammengesetzten Namen und CORBA-Referenzen. Der Vermittlungsdienst *(Trading-Service)* dagegen ermöglicht einem CORBA-Objekt, sich mit seinen Eigenschaften und Fähigkeiten zu registrieren. Interessierte Klienten können sich dann unter Einsatz einer Abfragesprache Referenzen passender Objekte besorgen.

Zusätzlich zu den hier vorgestellten Möglichkeiten bietet CORBA noch weitere Fähigkeiten, die von dem im folgenden Kapitel vorgestellten Erweiterungsmodul nicht unterstützt werden. Hier ist einerseits die große Menge weiterer CORBA-Dienste zu nennen. Da diese allerdings stets wiederum als CORBA-Objekte mit entsprechenden Schnittstellen modelliert sind, kann man diese auch als solche in seine Applikationsarchitektur integrieren. Allerdings ist auch eine spezifische Integration vorstellbar. So könnte man z.B. den Ereignisdienst *(Eventservice)* mit einem neuen Konnektortyp für einen Ereigniskanal modellieren.

Außerdem bietet CORBA auch dynamische Elemente an, die es erlauben, auf CORBA-Objekte zuzugreifen, deren Schnittstelle zur Übersetzungszeit noch nicht bekannt waren. Da diese allerdings selten Einsatz finden, wurde auf eine Unterstützung verzichtet. Abschließend wurde auch auf alle proprietäre Erweiterungen verzichtet, die nicht von der OMG spezifiziert wurden, sondern von dem konkret eingesetzten CORBA-System abhängen. Dazu gehört auch das Implementation-Repository, welches für persistente Server benötigt wird.

12.2. Architekturvokabular und Umsetzung

Obwohl CORBA prinzipiell programmiersprachenunabhängig ist, beschränkt sich das hier vorgestellte Erweiterungsmodul auf die Anbindung an Java in einem Rechnernetz bestehend aus Unix-ähnlichen Systemen. Insbesondere wird eine SSH-basierte Infrastruktur vorausgesetzt, die eine entfernte Prozesserzeugung ermöglicht. Die Unterstützung weiterer Programmiersprachen und

anderer Systeme bereitet keine konzeptionellen Probleme. Es wurde aber im Rahmen dieser Arbeit darauf verzichtet.

Für eine detaillierte Beschreibung der Anbindung von CORBA an Java wird auf [BVD01] verwiesen. Um das folgende zu verstehen, sollte die obige Einführung in CORBA ausreichen. Wie oben erwähnt, setzt das CORBA-Erweiterungsmodul sowohl auf das VIKS-Modul als auch auf das Modul für arbeitsablaufbasierte Applikationen auf. Das CORBA-Architekturvokabular lässt sich daher auch anhand der Oberklassen der Elementtypen in zwei Teile aufteilen. Diese werden in den folgenden beiden Kapiteln mit ihrer Umsetzung beschrieben. Bei der Beschreibung wird davon ausgegangen, dass die beiden Erweiterungsmodule, auf die aufgesetzt wird, bekannt sind.

12.2.1. Elemente für CORBA-basierte Informations- und Kontrollsysteme

Das VIKS-Erweiterungsmodul ermöglicht die Entwicklung verteilter Applikationen, die aus autonomen Dienstanbietern und Klienten bestehen. Eine derartige Applikation wird aufgesetzt, indem zuerst die einzelnen Applikationskomponenten an den richtigen Orten erzeugt werden und dann konfiguriert werden. Die Konfiguration beinhaltet dabei vor allem, dass die einzelnen Applikationskomponenten miteinander verknüpft werden.

Das VIKS-Modul stellt aber nur abstrakte Architekturelemente zur Verfügung, die es erlauben, Abhängigkeiten bzgl. der Platzierung und der Konstruktion und Konfiguration zu spezifizieren. Es beinhaltet auch eine Laufzeitunterstützung, welche eine Platzierungsanalyse vornehmen kann und auch einen Rahmen für Konstruktion und Konfiguration enthält. Dieses CORBA-Modul erweitert diese abstrakten Elemente nun um konkrete Architekturelemente für CORBA-basierte Applikationen. Außerdem bietet es Erweiterungen für das VIKS-Laufzeitsystem, um CORBA-Objekte entfernt zu konstruieren und zu konfigurieren.

Mit diesen neuen Elementen ist es nun möglich, das hypothetische Beispiel einer Überwachungsapplikation aus Abbildung 11.1 (s. Seite 142) konkret unter Einsatz von CORBA zu implementieren. Alle Komponenten, die Dienste anbieten, sind CORBA-Objekte. Wenn eine Komponente Dienste einer anderen benötigt, dann wird sie mit dem Anschluss des Dienstanbieters verbunden. In dem Beispiel benötigt die grafische Schnittstelle `GUI` Zugriff auf die Kontrollkomponente. Hierfür ist ihr Anschluss `ctrl` mit dem Anschluss `ControlI` verbunden, welcher die Schnittstelle der Kontrollkomponente repräsentiert.

Die Kolokalität der Komponenten `Camera` und `ImageEval` wird durch ein Attribut in dem verbindenden Konnektor ausgedrückt. Für die Erzeugung der Laufzeitexemplare werden bei diesem Beispiel Java-Objekte erzeugt. Die dafür benötigten Java-Klassen werden über Attribute der Komponenten spezifiziert. Im Falle von CORBA-Objekten werden die Java-Objekte nach der Erzeugung mit dem CORBA-System verbunden.

Die Konfiguration der gesamten Applikation geschieht, indem auf den Java-Objekten bestimmte Methoden aufgerufen werden. So besitzt z.B. das Java-

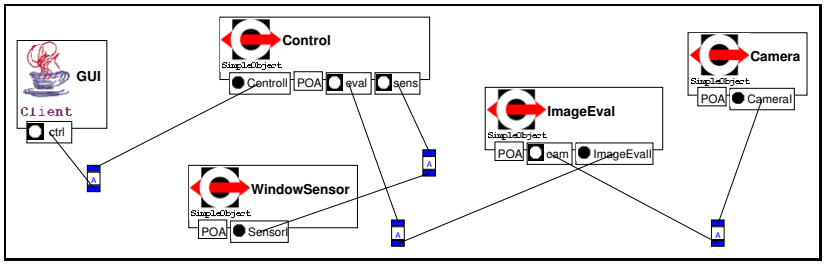

Abbildung 12.3.: Überwachungsapplikation mit CORBA

Objekt für die Komponente `Control` eine Methode `setSensor()`. Diese wird bei der Konfiguration mit der CORBA-Referenz des Laufzeitexemplars der `WindowSensor`-Komponente aufgerufen. Die Methode für die Konfiguration wird mit einem Attribut des Anschlusses `sens` bestimmt.

Dieses Beispiel soll einen ersten Eindruck des Architekturvokabulars vermitteln. Die Abbildungen 12.4 und 12.5 stellen die Architekturelemente getrennt nach Komponenten und Anschlüssen dar. Die Elementtypen des VIKS-Moduls, auf die aufgesetzt wird, sind über den Paketnamen kenntlich gemacht. Es sind zusätzlich zu den Elementtypen auch ihre Visualisierungen in dem grafischen Editor in den Diagrammen enthalten.

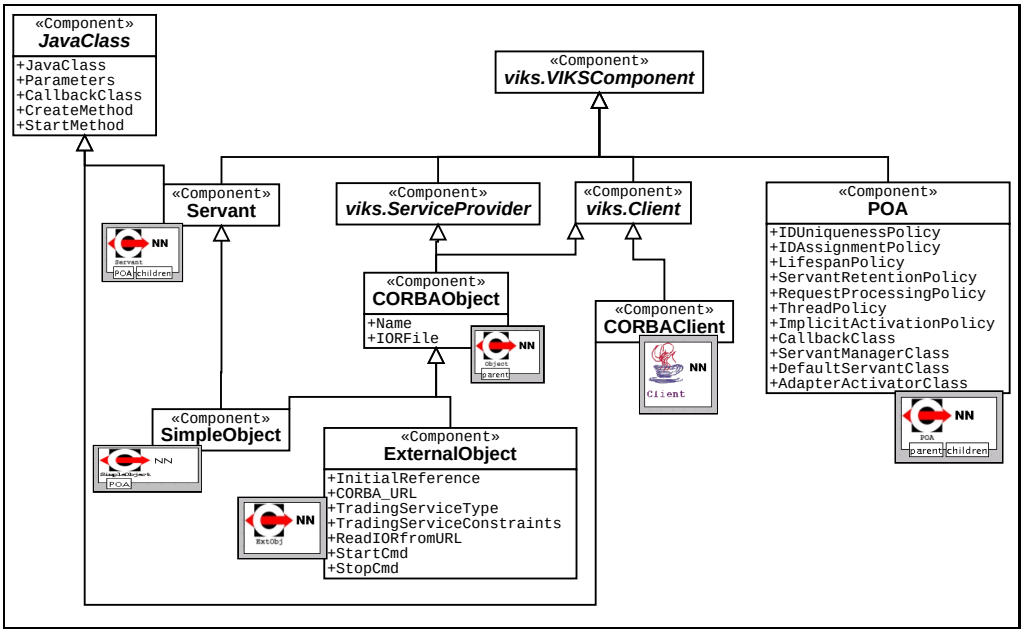

Abbildung 12.4.: Komponentenklassen des CORBA-Architekturvokabulars VIKS

Zuordnung von Implementierung zu Schnittstelle

In CORBA wird zwischen der fernaufrufbaren Schnittstelle eines CORBA-Objekt und der Implementierung dieser Schnittstelle, dem Servant, unterschieden. Häufig wird die Schnittstelle selbst als das CORBA-Objekt bezeich-

net. In dem Vokabular wird diese Terminologie aufgegriffen.

Fernaufrufbare Schnittstellen sind Komponenten des Typs `CORBAObject`, während Implementierungen durch Komponenten des Typs `Servant` dargestellt werden. Um zwei Komponenten einander zuzuordnen, wird der `parent` Anschluss des CORBA-Objekts mit dem `children` Anschluss des Servants über einen VIKS-Konnektor vom Typ `Association` miteinander verbunden.

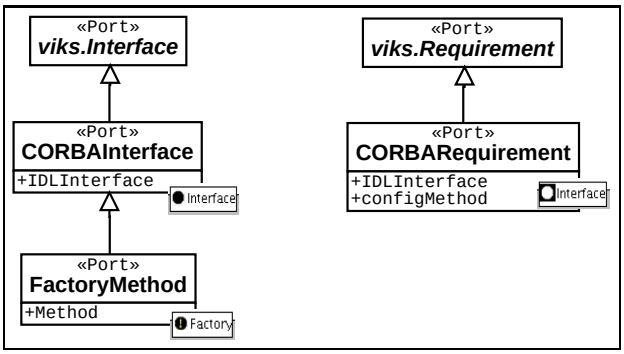

Abbildung 12.5.: Anschlussklassen des CORBA-Architekturvokabulars VIKS

Klienten werden mit ihrem `CORBARequirement` Anschluss an ein CORBA-Objekt über dessen `CORBAInterface` Anschluss verbunden. Dafür wird ein `Association` Konnektor eingesetzt. Die beiden Anschlusstypen besitzen das Attribut `IDLInterface`, welches auf den Namen der benutzten IDL-Schnittstelle gesetzt wird. Eine `CORBAObject`-Komponente kann auch auf andere `CORBAObject`-Komponenten als Klient zugreifen.

Ein Servant kann mit mehreren Schnittstellen verbunden sein. Häufig wird aber einem Servant genau einer Schnittstelle zugeordnet sein. Für diesen Fall gibt es den Komponententyp `SimpleObject`, der Schnittstelle und CORBA-Objekt in sich vereint.

Der Javacode des Servants wird über das Attribut `JavaClass` der Oberklasse `JavaClass` gesetzt. Wenn eine auf einem Servant basierende CORBA-Komponente *ccp* der Applikation zur Laufzeit konstruiert wird, wird zuerst ein Objekt *del* der spezifizierten Javaklasse *jcl* erzeugt. Diese muss alle Methoden der IDL-Schnittstelle von *ccp* implementieren. Konkret bedeutet dieses, dass für eine IDL-Schnittstelle `Foo` *jcl* die Java-Schnittstelle `FooOperations` implementieren muss. Sie muss keine weitere CORBA-Funktionalität unterstützen.

Diese Funktionalität wird von einer Klasse `FooPOATie` bereit gestellt, die automatisch aus der IDL-Schnittstelle generiert wird. Ein Objekt *serv* dieser Klasse wird nun erzeugt und so konfiguriert, dass alle Methodenaufrufe an *del* delegiert werden. Abschließend wird *serv* über den spezifizierten POA an das CORBA-System angeschlossen und steht ab dann als Dienstanbieter zur Verfügung. Die Zuordnung von POAs zu CORBA-Objekten wird später beschrieben. Zusammengefasst wird also in etwa der folgende Javacode bei der Konstruktion einer CORBA-Komponente ausgeführt:

```
FooOperations del = new JCl();
```

```
serv = new FooPOATie(del);
org.omg.CORBA.Object cob =
  poa.servant_to_reference(serv);
```

Der konkret benutzte Code ist allerdings komplexer, da die einzelnen Klassen über den Reflektionsmechanismus von Java geladen werden. Hinzukommend stehen noch zusätzliche Möglichkeiten zur Parametrisierung dieses Konstruktionsvorgangs zur Verfügung.

So können mit dem `Parameters`-Attribut Parameter spezifiziert werden, die dem Konstruktor der spezifizierten Java-Klasse bei der Konstruktion übergeben werden. Neben Literalen können auch einige Variablen benutzt werden, die von dem Laufzeitsystem ersetzt werden. So wird z.B. die Zeichenkette `POA` durch eine Referenz auf den benutzten Portable-Object-Adapter ersetzt und ermöglicht so einen Zugriff auf das CORBA-System. Weitere Details sind unten in dem Abschnitt über Auflösung von Abhängigkeiten zwischen Klienten und Dienstanbietern zu finden. Zusätzlich lässt sich eine Methode `StartMethod` bestimmen, die auf dem erzeugten Implementierungsobjekt nach der Konfigurationsphase aufgerufen wird.

Rückrufmethoden

Mit dem `CallbackClass`-Attribut läßt sich eine Java-Klasse spezifizieren, die Methoden zur Verfügung stellt, die während der einzelnen Phasen wie Konstruktion und Konfiguration ausgeführt werden. Abbildung 12.6 zeigt die entsprechende Java-Schnittstelle. Die einzelnen Methoden erhalten die Parameter, die in der entsprechenden Phase bzgl. der zu bearbeitenden Komponente schon bekannt sind. Eine Rückrufmethode, die vor der Konfigurationsphase aufgerufen wird, ist nicht nötig, da der entsprechende Code auch gleich nach der Konstruktionsphase ausgeführt werden kann.

```
public interface UserCallback
{
    void preConstruction(ORB orb, String name);
    void postConstruction(POA poa, Object obj);

    void postConfiguration(org.omg.CORBA.Object obj);

    void preShutdown();
    void postShutdown();
}
```

Abbildung 12.6.: Schnittstelle der Javaklasse für Rückrufmethoden des Applikationsentwicklers

Möchte man keine Javaklasse spezifizieren, sondern das Fabrikentwurfsmuster einsetzen, kann man mit `CreateMethod` eine Methode der Rückrufklasse bestimmen. Diese wird dann mit den spezifizierten Parametern benutzt, um ein Implementierungsobjekt zu erzeugen, zu dem die Methodenaufrufe delegiert werden.

Veröffentlichung von CORBA-Objekten

Wenn ein CORBA-Objekt erzeugt wurde, gibt es zwei Möglichkeiten, es öffentlich bekannt zu machen. Einerseits kann es sich bei dem Namensdienst eintragen. Hierfür dient das Attribut `Name` in dem Komponententyp `CORBAObject`. Alternativ kann auch die Referenz auf das CORBA-Objekt als IOR in eine Datei geschrieben werden. Der Dateiname wird über das Attribut `IORFile` gesetzt. Beide Ansätze lassen sich auch kombinieren.

Zuordnung von CORBA-Objekten zu POAs

CORBA-Objektkomponenten werden mit ihrem `POA` Anschluss über einen Konnektor des Typs `Association` mit dem entsprechenden POA verknüpft. Wenn einer CORBA-Objektkomponente kein POA explizit zugeordnet wurde, wird der Standard-POA benutzt. POA-Hierarchien werden aufgebaut, indem POAs miteinander assoziiert werden. Der übergeordnete POA wird dabei mit dessen `children` Anschluss über einen Assoziationskonnektor mit dem `parent` Anschluss des untergeordneten POAs verbunden. Abbildung 12.7 zeigt exemplarisch diese Verknüpfungen.

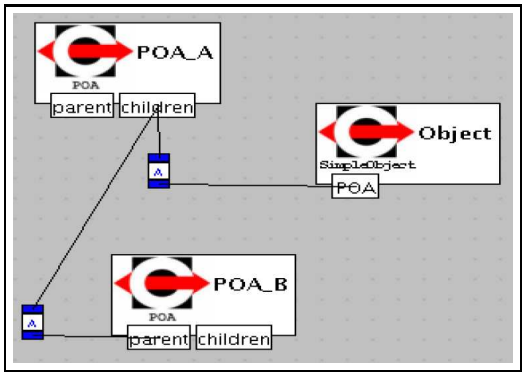

Abbildung 12.7.: Beispiel für die Komposition von POAs

Einbinden externer CORBA-Objekte

CORBA-Applikationen sind häufig nicht in sich abgeschlossen sondern Teil eines größeren Systems. Sie müssen also in der Lage sein, auch CORBA-Objekte zu benutzen, die extern verwaltet werden. Dafür müssen sie diese lokalisieren und dann mit ihnen kommunizieren können. Um für die Kommunikation entsprechende Stellvertreterobjekte zu erzeugen, muss die IDL-Schnittstelle des externen CORBA-Objekts bekannt sein. Diese ist als Attribut in dem `CORBAInterface` Anschluss gegeben.

Es gibt mehrere Möglichkeiten ein externes Objekt zu lokalisieren. Welche benutzt werden, hängt von den Attributen ab, die in der `ExternalObject` Komponente gesetzt sind. Wenn mehrere gesetzt sind, dann gibt es eine feste Reihenfolge, wie versucht wird, ein Objekt aufzufinden. Die folgende Aufzählung beschreibt die Möglichkeiten in dieser Reihenfolge:

1. Mit dem Attribut `InitialReference` kann man eine initiale Referenz für den ORB bestimmen. Diese initialen Referenzen werden durch die

allgemeine Konfiguration des CORBA Systems bestimmt. Sie bestehen aus Zuordnungen von Namen zu CORBA-Objekten, die von jedem lokalen ORB-Vertreter aufgelöst werden können.

2. Eine Referenz auf ein CORBA-Objekt lässt sich mit einem URL-Schema darstellen. So ist die URL
   `corbaloc::host.fu-berlin.de:4711/POA_A/ObjectId`
   eine Referenz auf ein CORBA-Objekt, das von einem Server-Prozess auf dem Rechner `host.fu-berlin.de` getragen wird. Der Server-Prozess lauscht auf dem Port `4711` auf eingehende Methodenaufrufe. Die Implementierung des CORBA-Objekts, der Servant, wird von dem POA mit dem Namen `POA_A` unter dem Bezeichner `ObjectId` verwaltet.

3. Wie in der Einführung schon erwähnt, gehört neben dem Namensdienst auch der Vermittlungsdienst zu den Standarddiensten von CORBA. Über die Attribute `TradingServiceType` und `TradingServiceConstraints` lässt sich eine Anfrage an den Vermittlungsdienst spezifizieren, um das externe Objekt zu lokalisieren.

4. Ist das Attribut `Name` gesetzt, das von dem Komponententyp `CORBAObject` stammt, so wird der Namensdienst zum Auffinden des Objekts eingesetzt.

5. Eine Referenz auf ein CORBA-Objekt, eine IOR, lässt sich in eine Zeichenkette umwandeln, die dann weitergegeben oder in eine Datei gespeichert werden kann. Mit dem Attribut `ReadIORfromURL` lässt sich eine solche Datei spezifizieren. Wenn alle obigen Möglichkeiten zu keiner Lösung geführt haben, wird versucht eine IOR aus der spezifizierten Datei zu lesen.

6. Als letzte Möglichkeit wird ein externes Programm gestartet. Dieses wird über SSH evtl. entfernt ausgeführt und kann dann das externe Objekt aufsetzen. Dafür muss in dem Attribut `startCmd` der Name des Programms gesetzt sein. Nach der Ausführung werden wiederum die obigen Lokalisationsmöglichkeiten in der gleichen Reihenfolge eingesetzt, um das externe Objekt zu lokalisieren. Gelingt dieses nicht, bricht der Gesamtvorgang mit einem Fehler ab.

Auflösung von Abhängigkeiten zwischen Klienten und Dienstanbietern

Es gibt zwei Ansätze wie Abhängigkeiten zwischen Dienstanbietern und Klienten aufgelöst werden können. Erstens kann der Dienstanbieter dem Klienten bei dessen Konstruktion übergeben werden. Hierfür wird das Attribut `Parameters` aus dem Komponententyp `JavaClass` benutzt, das bei der Erzeugung eines Servants berücksichtigt wird. Ein Parameter `CP_PORT(id)` an einem Klienten *c* wird bei der Erzeugung im Konstruktor durch eine Referenz auf den Dienstanbieter, ein CORBA-Objekt, ersetzt, der mit dem Anschluss `id` der Komponente *c* verbunden ist.

Der zweite Ansatz besteht darin, auf einem Klienten nach dessen Konstruktion eine Methode aufzurufen, welche ihm den Dienstanbieter bekannt macht.

Angenommen der Dienstanbieter wäre ein CORBA-Objekt, das Druckerdienste anbietet. Ein Klient, der auf einen Drucker angewiesen ist, hätte dann eine Methode:

```
void setPrinter(Printer prt);
```

Nachdem sowohl der Klient als auch das Druckerobjekt konstruiert sind, wird diese Methode auf dem Klienten aufgerufen. Da der Klient ein Javaobjekt ist, welches von dem Laufzeitsystem erzeugt wurde, stellt dieser Aufruf kein Problem dar.

12.2.2. Elemente für arbeitsablaufbasierte CORBA-Applikationen

Das in Kapitel 6 vorgestellte Erweiterungsmodul stellt konkrete und abstrakte Architekturelemente zur Verfügung, um Taskgraph-artige Applikationen zu erstellen, die aus einer evtl. nebenläufigen Abarbeitung von Operationen bestehen. In dem Kontext CORBA-basierter Applikationen könnte eine derartige Applikation verschiedene CORBA-Objekte steuern.

Abbildung 12.8 zeigt die zwei Architekturelemente für arbeitsablaufbasierte CORBA-Applikationen. Die Komponente `CORBAOperation` stellt den Aufruf einer Methode eines CORBA-Objektes dar. Ein Methodenaufruf liefert evtl. Ergebniswerte zurück. Diese sind i.a. zu klein, als dass sie in einem entfernten Datenobjekt gespeichert werden sollten. Allerdings sollten diese Rückgabewerte nicht verloren gehen, sondern der nächsten Operation zur Verfügung stehen. `CORBAOperationResult` ist ein entsprechend erweiterter Konnektor.

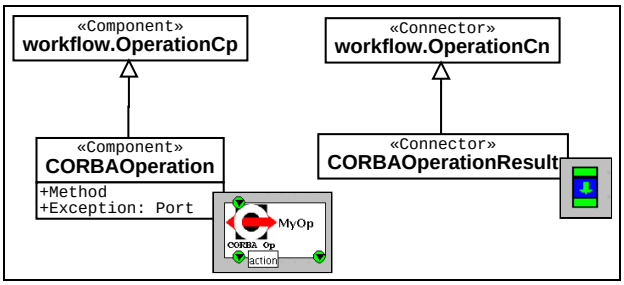

Abbildung 12.8.: Komponentenklassen des CORBA-Architekturvokabulars für arbeitsablauf-basierte Applikationen

Mit dem Attribut `Method` des Komponententyps `CORBAoperation` wird die Methode spezifiziert, die bei der Aktivierung ausgeführt wird. Die Komponente, die das CORBA-Objekt spezifiziert, auf dem die Methode aufgerufen wird, ist mit dem `Object` Anschluss zu verbinden. Kommt es bei der Ausführung der Methode zu einer Ausnahme wird der Kontrollfluss nicht über den Standardanschluss `done` geleitet sondern über den Anschluss `Exception`.

Der Konnektortyp `CORBAOperationResult` besitzt alle Eigenschaften, die der Typ `OperationCn` besitzt. Er kann also mehrere Operationen über eine gegebene Strategie miteinander koordinieren. Für eine genauere Beschrei-

bung siehe Seite 72. Zusätzlich fragt er, wenn ein Aktivierungssignal von einer `CORBAOperation` Komponente kam, von dieser den Ergebniswert ab und gibt diesen an die zu aktivierende Komponente weiter, sofern diese wiederum vom Typ `CORBAOperation` ist.

Damit eine CORBA-Operationskomponente *c* auf einen Ergebniswert einer vorhergegangenen Operation zugreifen kann, muss das Attribut `Method` geeignet gesetzt sein. Ist diese z.B. auf `meth(INPUT0)` gesetzt, wird der erste Parameter der von *c* aufzurufenden Methode `meth` mit dem vorherigen Ergebniswert besetzt. Da CORBA-Methoden Ergebnisse auch in Parametern ablegen können, müssen diese ebenfalls berücksichtigt werden. Ist z.B. der dritte Parameter der vorher aufgerufenen Methode ein solcher Rückgabewert, so kann man diesen mit `meth(INPUT1)` benutzen. Durch diese Erweiterung ist der Kontrollfluss-basierte Ansatz für arbeitsablaufbasierte Applikationen um Datenfluss erweitert worden. Die Datentypen, die zwischen den Operationen fließen können, sind durch die Schnittstellenbeschreibungssprache von CORBA genau spezifiziert.

12.3. Verwandte Arbeiten

Im Gegensatz zu dem hier vorgestellen deklarativen Ansatz ist es auch möglich, CORBA-basierte Applikationen aufzusetzen, indem die einzelnen zentralisierten Komponenten auf den gewünschten Rechnern zuerst gestartet und dann mit einer imperativen oder funktionalen Programmiersprache explizit konfiguriert werden. Als Programmiersprache bietet es sich an, eine verbreitete Skriptsprache, wie z.B. Python oder Perl, zu benutzen, die mittels zusätzlicher Bibliotheken um die Fähigkeit erweitert wird, CORBA-Objekte anzusprechen [fno02, per02]. Für die Programmiersprache Lua existiert die Erweiterung *LuaSpace*, die neben dem Zugriff auf CORBA-Objekte zusätzlich auch eine dynamische Rekonfiguration CORBA-basierter Applikationen unterstützt [BR00].

Der Vorteil dieses Ansatzes, eine übliche Programmiersprache für die Konfiguration verteilter Komponenten zu benutzen, besteht darin, dass der Anwendungsentwickler eine starke Kontrolle über die Applikation ausüben kann und ihm damit Möglichkeiten zur Verfügung stehen, die mit dem in diesem Kapitel vorgestellten Erweiterungsmodul nicht oder nur mit einem hohen Aufwand realisiert werden können. Insbesondere bei Applikationen mit einer dynamischen Architektur wird der Entwickler durch die Beschränkung des Erweiterungsmoduls bzgl. der Spezifikation dynamischer Architekturen auf eine $1 - *$ Beziehung zwischen Dienstanbieter und Klienten spürbar eingeschränkt.

Andererseits erfordert der explizite Konfigurationsansatz im Vergleich zu dem impliziten deklarativen Ansatz einen erhöhten Implementierungs- und Wartungsaufwand für Applikationen, die sich mit den hier zur Verfügung gestellten Architekturelementen adäquat ausdrücken lassen. In den Fallstudien im Kapitel 13 wird demonstriert werden, dass dies für viele typische Anwendungen aus dem Bereich der verteilten Informations- und Kontrollsysteme

gilt. Ein weiterer Vorteil besteht darin, dass aufgrund der relativ geringen Anzahl von Architekturelementen, deren Semantik formal spezifiziert ist (s. Kapitel 11), sich die Korrektheit einer Applikation einfacher überprüfen lässt, als bei dem Einsatz einer allgemeinen Programmiersprache.

In [MTK97] wird ein System für die Erstellung CORBA-basierter Applikationen vorgestellt, das auf der Architekturbeschreibungssprache Darwin basiert. Aus der Applikationsarchitektur werden automatisch IDL-Schnittstellen erzeugt. Die Anbindung von Code an die Schnittstellen und das Installieren und Starten zentralisierter Komponenten ist von dem Entwickler händisch durchzuführen.

Das System ZCL basiert ebenfalls auf einer Architekturbeschreibungssprache [PB02]. Jede Komponente einer Applikationsarchitektur repräsentiert ein CORBA Objekt. Der Applikationsentwickler erstellt für jede Komponente ein LuaSpace-Programm, welches die Aktivierung und Deaktivierung des zugehörigen CORBA-Objekts implementiert. Dieses Programm kann eine zentrale Konfigurations-Tabelle abfragen, um das zugehörige CORBA-Objekt korrekt zu konfigurieren.

In ZCL ist im Gegensatz zu dem vorgestellten Erweiterungsmodul keine statische Abhängigkeitsanalyse notwendig, da Abhängigkeiten zur Laufzeit dynamisch aufgelöst werden. Hierfür werden generische Konnektoren eingesetzt, die alle Aufrufe zwischen Dienstanbietern und Klienten vermitteln. Ist der von dem Klienten kontaktierte Dienstanbieter nicht aktiv, wird er unter Einsatz des dem Anbieter zugehörigen LuaSpace-Programms aktiviert. Der Einsatz dieses vermittelnden Konnektors ermöglicht zudem eine dynamische Rekonfiguration einer CORBA-basierten Applikation. Allerdings entsteht durch den Konnektoransatz, der sich bei jedem entfernten Aufruf auswirkt, ein signifikanter zusätzlicher Rechen- und Kommunikationsaufwand zur Laufzeit.

Ebenso wie bei dem hier vorgestellten Erweiterungsmodul besteht das Entwurfsziel des Systems Olan darin, unterschiedliche zentralisierte Softwaremodule zu einer verteilten Applikation zu verschmelzen, die als Applikationsarchitektur gegeben ist [BBB+98, IBRZ00]. Allerdings beschränkt sich Olan nicht auf eine spezifische Middleware, wie CORBA, sondern benutzt einen Schichtenansatz, um unterschiedliche Middlewaresysteme zu unterstützen. Jede Interaktion zwischen zwei Softwaremodulen wird über Olan-Stellvertreter (Stubs) durchgeführt, welche die jeweils benutzte Middleware ansprechen. Hierdurch ist es sehr einfach, unterschiedliche Middlewaresysteme miteinander zu kombinieren. Allerdings entsteht wie auch bei ZCL ein zusätzlicher Aufwand zur Laufzeit. Zusätzlich verbietet dieser Schichtenansatz, der auf der obersten Schicht nur eine feste Menge von Konnektoren dem Applikationsentwickler anbietet, die Nutzung spezifischer Funktionalität der eingesetzten Middleware-Systeme.