1 CORBA

Das CORBA-Komponentenmodell ist eine neue Technologie für komponentenorientierte, CORBA-basierte verteilte Anwendungen. CORBA als Grundlage des CORBA-Komponentenmodells stellt hierbei die Grundkonzepte für Verteilung und Kommunikation bereit. Das folgende Kapitel ist CORBA selbst gewidmet. Im ersten Teil erfolgt eine zeitliche Einordnung dieser Technologie. Daran anschließend wird ein Überblick über CORBA und seine Bestandteile und Kernkonzepte gegeben. Danach werden die für das CORBA-Komponentenmodell wesentlichen Bestandteile einzeln vorgestellt.

1.1 Überblick CORBA

Rechnernetzwerke realisieren effiziente und adaptive Berechnungswerkzeuge.

Die Computertechnik hat in den vergangenen Jahrzehnten eine rasante Entwicklung genommen. Wurden anfangs Rechner ausschließlich einzeln und unabhängig von anderen Rechnern für die Lösung einer Aufgabe herangezogen, dominieren heute vernetzte Rechnerkonfigurationen, bei denen mehrere Rechner gemeinsam an der Lösung einer Aufgabe beteiligt sind. Dabei lassen sich im Wesentlichen zwei Arten von vernetzten Rechnerverbünden beobachten, homogene Verbünde und heterogene Verbünde. Bei den homogenen Verbünden arbeiten weitestgehend baugleiche Rechner zusammen, die über die gleiche Netzwerktechnologie verbunden sind. Solche Rechnerverbünde sind häufig unter der alleinigen Kontrolle einer einzelnen Institution oder Person. Das zur Kommunikation benutzte Protokoll kann hierbei relativ einfach gestaltet werden, da aufgrund der einheitlichen Architektur alle Rechner ein gemeinsames Verständnis über die Art und Struktur von Datentypen haben.

An heterogenen Verbünden sind verschiedene Rechner mit unterschiedlichen Architekturen, Betriebssystemen und Netzwerktechnologien beteiligt. Entsprechend aufwendig muss ein Protokoll zur Kommunikation zwischen den beteiligten Rechnern gestaltet werden, da es nicht nur von den unterschiedlichen verwendeten Netz-

werktechnologien und Architekturen abstrahieren muss, sondern auch die in den verschiedenen Architekturen verwendeten Datentypen und Datenstrukturen geeignet aufeinander abbilden muss.

Historisch gesehen abstrahieren Programmiersprachen immer stärker von der konkreten Technologie.

In den Anfangszeiten der Computertechnik wurden Programme ausschließlich manuell in binärem Code geschrieben. Neben der Tatsache, dass dies ein extrem zeitaufwendiger und fehlerträchtiger Prozess war, setzte es außerdem voraus, dass der Entwickler die Zielmaschine bis in das letzte Detail kennt. Später aufkommende Abstraktionen in Form von Assemblersprachen hatten anfänglich mit geringer Akzeptanz zu kämpfen, da man von dieser Abstraktion eine ineffektive Nutzung der technischen Ressourcen eines Rechners erwartete. In kurzer Zeit setzten sich trotzdem die Assemblersprachen durch, da der Programmierer sich hier wesentlich besser auf das eigentliche Problem konzentrieren konnte und weniger die aktuellen technischen Gegebenheiten in Betrachtung ziehen musste. Später wurden die Assemblersprachen durch Hochsprachen wie Simula, Pascal, Fortran, Cobol, C++ und Java abgelöst, von denen die meisten auch heute noch unverzichtbar für die Softwareentwicklung sind.

Objektorientierte Programmiersprachen ermöglichen eine klare Trennung von Daten und Zugriffschnittstelle.

Ein Meilenstein in der Softwareentwicklung wurde durch die objektorientierte Programmierung erreicht. Das wesentliche Konzept dieser Programmiertechnik ist das *Objekt*, ein programmiersprachliches Konstrukt, das seinen Zustand, der in Form von programmiersprachlichen Daten vorliegt, kapselt und Zugriff auf diesen Zustand über wohldefinierte Interfaces gestattet. Objekte stellen über ihr Interface Dienste bereit, die von anderen genutzt werden können. Hauptinteraktionspunkte an solchen Interfaces sind Methoden, die Daten entgegennehmen und berechnete Daten zurückliefern. Aufrufe solcher Methoden führen zu Zustandsänderungen des Objekts. Somit lässt sich eine klare Rollenverteilung erkennen: der Aufrufer einer Methode an einem Objekt ist in der Rolle eines Klienten des Objekts, während das Objekt, das den Methodenaufruf verarbeitet in der Rolle eines Servers ist. Diese Rollenverteilung findet sich auch in modernen Netzwerken und Rechnerverbünden wieder. Ein Rechner (Klient) nutzt Dienste, die durch einen anderen Rechner (Server) angeboten werden. Es gibt eine Vielzahl von Gründen dafür, dass diese Dienste durch einen anderen Rechner angeboten werden:

- Ein Dienst stellt hohe Anforderungen an die Rechenkapazität.
- Ein Dienst soll vielen Nutzern offen stehen.
- Die Wartung und Aktualisierung der durch den Dienst genutzten Daten soll zentral vorgenommen werden.

- Bei der Bereitstellung des Dienstes werden sensible oder geschützte Daten benutzt, die in dieser Form dem Nutzer nicht zugängig gemacht werden sollen.

CORBA kombiniert das Klienten-Server-Modell mit den Konzepten der objektorientierten Programmierung.

Die Aufteilung in Klient und Server als Beispiel eines einfachen Rechnerverbundes setzt jedoch die Vernetzung dieser Rechner voraus. Dazu muss ein geeigneter Mechanismus gefunden werden, mit dem die Dienste des Servers dem Klienten zur Verfügung gestellt werden können. Die Daten, die zur Steuerung des Dienstes dienen, müssen über das Netzwerk ausgetauscht werden. In den Anfängen wurden die dafür erforderlichen Protokolle von Hand implementiert. Das war zum einen ein fehlerträchtiger Prozess, zum anderen waren diese Protokolle meist sehr stark abhängig von der unterstützten Anwendung und damit wenig adaptierbar an neue Anforderungen.

Später entlasteten Technologien wie ONC/RPC (*Open Network Computing/Remote Procedure Call*) und DCE/RPC (*Distributed Computing Environment/Remote Procedure Call*) den Programmierer beim Entwurf und der Implementierung verteilter Programme. Mit solchen RPC-Technologien lassen sich Anwendungen implementieren, bei denen Prozeduraufrufe transparent über ein Netzwerk an anderen Anwendungsteilen gerufen werden können. Diese Technologien fanden vor allem in prozeduralen Sprachen Anwendung.

Mit der zunehmenden Beliebtheit von objektorientierten Programmiersprachen wie C++ und Java erwuchs schnell der Wunsch, die von den prozeduralen Sprachen bekannten Prozeduraufrufe über Netzwerkgrenzen hinweg auch auf die Methodenrufe an Objekten auszudehnen. Hier setzt die *Common Object Request Broker Architecture (CORBA)* an (s. Born et al. 2004). CORBA ist eine Technologie zur Realisierung verteilter, objektorientierter Anwendungen. Zentrales Konzept von CORBA ist das *CORBA-Objekt*, das seine Dienste über ein wohldefiniertes Interface bereitstellt. Klienten nutzen die Dienste eines CORBA-Objekts in dem sie Operationsaufrufe an das CORBA-Objekt senden. Diese Operationsaufrufe, die den herkömmlichen Methodenaufrufen entsprechen, werden transparent in entsprechende Netzwerknachrichten umgewandelt und vom gerufenen CORBA-Objekt ausgewertet. Das Ergebnis des Operationsaufrufs wird wiederum transparent als Netzwerknachricht an den Klienten übermittelt.

CORBA ermöglicht plattformunabhängige Softwareentwicklung.

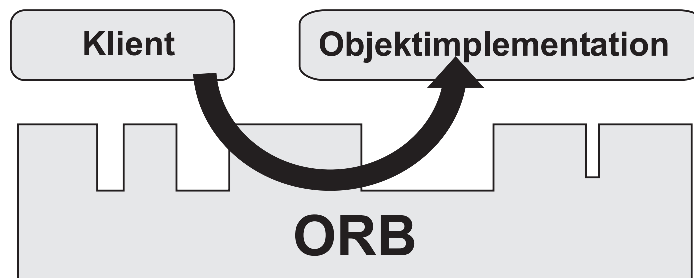
Zu den herausragenden Eigenschaften von CORBA gehört die Plattformunabhängigkeit. Anwendungen können auf einer Vielzahl von Plattformen entwickelt und eingesetzt werden, von der Mehrprozessor-Maschine bis hin zum *Personal Data Assistant* (PDA). CORBA ermöglicht eine Softwareentwicklung unabhängig von der im Rechner eingesetzten CPU und des eingesetzten Betriebssystems.

Zusätzlich dazu abstrahiert CORBA auch von der verwendeten Programmiersprache, so dass es möglich wird, verteilte Anwendungen aus Anwendungsteilen zusammenzusetzen, die in unterschiedlichen Programmiersprachen entwickelt wurden. Dies ermöglicht, Teile der Applikation in der am besten dafür geeigneten Programmiersprache zu schreiben. So kann beispielsweise ein CORBA-Objekt im Server in C++ geschrieben werden, um eine optimale Performanz des Dienstes zu erreichen, während die grafische Oberfläche des Klienten in Java geschrieben ist und von den Stärken dieser Sprache bei der Implementierung von grafischen Nutzerinterfaces profitiert.

Die zentrale Vermittlungsstelle bei der Kommunikation von Methodenaufrufen zwischen Klient und CORBA-Objekt ist der *Object Request Broker* (ORB). Der ORB ist für die Codierung und Decodierung von Netzwerknachrichten zur Übermittlung von Operationsaufrufen und Operationsantworten verantwortlich und stellt grundlegende unterstützende Laufzeitdienste für CORBA-Objekte bereit.

CORBA-Objekte werden von so genannten *Servants* implementiert. Ein *Servant* bezeichnet das konkrete programmiersprachliche Konstrukt, das die Funktionalität des Interfaces eines CORBA-Objekts oder mehrerer CORBA-Objekte implementiert. In C++ werden *Servants* durch C++-Klassen implementiert. Der Begriff *Server* bezeichnet in CORBA den Kontext in dem ein *Servant* läuft. Das ist in den meisten Fällen ein Betriebssystemprozess.

Abb. 1.1: Der Object Request Broker



CORBA-Objekte werden durch Objektreferenzen adressiert. Eine Objektreferenz verdeckt den augenblicklichen Aufenthaltsort eines CORBA-Objekts. Durch dieses Konzept gelingt es, die konkrete Verteilung von CORBA-Objekten in verteilten Applikationen für einen Klienten transparent zu machen.

Das Interface eines CORBA-Objekts wird durch eine spezielle Beschreibungssprache spezifiziert, die *Interface Definition Language* (IDL). Durch IDL werden Datentypen, Interfaces und zuge-

hörige Operationen und Parameter beschrieben. CORBA-Objekte können grundsätzlich nur ein CORBA-Interface implementieren.

Daneben definiert der CORBA-Standard eine Reihe von so genannten *Common Object Services*, Diensten, die häufig benötigte Aufgaben in verteilten Anwendungsumgebungen realisieren. Diese Dienste sind selbst wieder als CORBA-Interfaces spezifiziert.

Die wichtigsten Bestandteile des CORBA-Standards werden in den folgenden Kapiteln vorgestellt. Dabei kann dieser kurze Einführung in CORBA ein sorgfältiges Studium des Standards nicht ersetzen. Vielmehr sollen diejenigen Konzepte von CORBA noch mal in Erinnerung gerufen werden, die hauptsächlich beim CORBA-Komponentenmodell Anwendung finden.

1.2 Object Request Broker

Der ORB ist die zentrale Schaltzentrale bei der CORBA-Kommunikation.

Der *Object Request Broker* ist der zentrale Bestandteil der CORBA-Architektur. Er wird sowohl auf Klientenseite als auch auf Serverseite benutzt. Seine Hauptaufgabe ist die Übertragung von Operationsaufrufen (*Requests*) vom Klienten zum Server and Operationsantworten (*Reply*) vom Server zum Klienten. Dazu muss er im Vorfeld die Implementierung des CORBA-Objekts auffinden und auf den Empfang eines Operationsaufrufs vorbereiten. Nach der Abarbeitung der Operation im gerufenen CORBA-Objekt wird das Resultat des Operationsaufrufs mit seiner Hilfe an den Klienten zurück übermittelt.

Der ORB ermöglicht weiterhin den Zugriff auf eine Reihe grundlegender Hilfsfunktionen in CORBA. So bietet er Funktionen an, um Objektreferenzen in eine Zeichenkette umzuwandeln, um sie an geeigneter Stelle für einen späteren Gebrauch ablegen zu können. Umgekehrt kann eine in einer Zeichenkette abgelegte Objektreferenz mit Hilfe des ORB wieder in eine interne Objektreferenz umgewandelt werden.

Der ORB ist nur konzeptionell ein monolithisches Gebilde. In den meisten ORB-Produkten und damit in den meisten verteilten CORBA-Anwendungen wird die Funktionalität des ORB in einer Bibliothek bereitgestellt. Bei der Kommunikation zwischen Klient und Server kann damit ein Teil des ORB in der Bibliothek des Klienten implementiert sein, während ein anderer Teil in der Bibliothek des Servers implementiert ist. Auch vollkommen andere Konstellationen der ORB-Partitionierung sind denkbar.

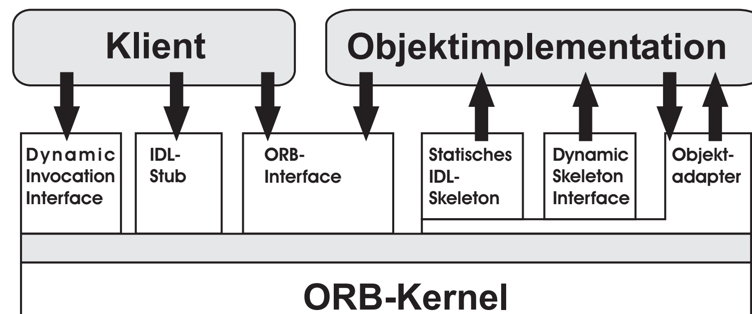
Zugriff auf die Hilfsfunktionen des ORB erfolgt für Klienten und CORBA-Objekte gleichermaßen über das ORB-Interface. Dieses In-

terface ist standardisiert und ermöglicht damit identischen Zugriff auf die ORB-Funktionen, unabhängig von der jeweils genutzten ORB-Implementierung.

Stubs und Skeletons sind typspezifische ORB-Interfaces, DII und DSI sind generische ORB-Interfaces.

IDL-Stubs sind vom Interface des Zielobjekts abhängige klientenseitige ORB-Interfaces zum Initiieren eines Operationsaufrufs. Wie auch in den streng typisierten Programmiersprachen wie beispielsweise C++ und Java muss man den Typ eines Objekts kennen, um eine Operation an diesem aufzurufen. In CORBA gibt es über diese Möglichkeit hinaus auch einen Aufrufmechanismus, der an die schwach typisierten Sprachen wie Python und Tcl angelehnt ist. Das *Dynamic Invocation Interface* (DII) ist ein vom Interface des Zielobjekts unabhängiges klientenseitiges ORB-Interface, das ebenfalls für einen Operationsaufruf verwendet werden kann. Hier müssen nur der Name der Operation und die Parameter bekannt sein, um eine Operation an einem beliebigen CORBA-Objekt aufzurufen. Beide ORB-Interfaces haben ihre Berechtigung. Während die typspezifischen *Stub*-Interfaces auf natürliche Weise die Signatur des entsprechenden CORBA-Interface nachbilden, ist für die Nutzung des *Dynamic Invocation Interface* häufig ein mehrstufiger Prozess zum Aufbau eines Operationsaufrufs erforderlich. Der Vorteil des *Dynamic Invocation Interface* ist es, dass kein typspezifischer Programmcode benötigt wird.

Abb. 1.2: Interfaces am ORB



IDL-Skeletons sind in Analogie zu *IDL-Stubs* typspezifische serverseitige ORB-Interfaces zur Implementierung eines CORBA-Objekts. Das *Dynamic Skeleton Interface* (DSI) ist ein vom implementierten CORBA-Interface unabhängiges serverseitiges ORB-Interface zur Implementierung eines CORBA-Objekts. Es ermöglicht es, nun auch bei den Objektimplementierungen eine schwache Typisierung zu verfolgen. Somit ist mit dem *Dynamic Skeleton Interface* eine einfache Realisierung von Delegierungen für beliebige Typen von

CORBA-Objekten möglich. Hier steht wieder dem Nachteil einer komplizierteren Implementierung eines solchen CORBA-Objekts der Vorteil gegenüber, nicht auf zusätzliche typspezifische programmiersprachliche Konstrukte angewiesen zu sein.

Für beide Implementierungsstrategien gibt es relevante Anwendungsfälle. In den meisten Fällen werden CORBA-Objekte unter Nutzung eines typspezifischen *IDL-Skeleton* implementiert, während man für die Implementierung generischer Brücken zwischen CORBA und einer weiteren Technologie meist auf das *Dynamic Skeleton Interface* zurückgreifen muss, da es für eine solche Brücke nicht möglich ist, sämtliche zu bedienenden CORBA-Interfaces im Vorfeld zu kennen.

Objektadapter bieten serverseitige Interfaces für jeweils eine bestimmte Klasse von Objektimplementierungen an.

Ein Objektadapter ist der wichtigste Zugriffspunkt für Objektimplementierungen auf serverseitige Dienste des ORB. Zu den Aufgaben eines Objektadapters zählen:

- Generierung und Interpretation von Objektreferenzen,
- Vermitteln der Operationsaufrufe an die verwalteten Objekte,
- Sicherheit von Interaktionen,
- Objekt- und Implementierungsaktivierung,
- Abbildung von Objektreferenzen auf Implementierungen und
- Registrierung von Implementierungen.

Der Grund für die Einführung eines Objektadapters in Ergänzung zum ORB-Interface auf Server-Seite liegt in der großen Vielfältigkeit von Objektimplementierungen hinsichtlich Objektgranularitäten, Lebenszeiten, Implementierungsarten und weiteren Eigenschaften begründet. Diese Vielfältigkeit macht es für den ORB schwierig, ein einziges für alle Objekte gleichermaßen effizientes und bequem zu nutzendes Interface anzubieten. Mit speziellen Objektadaptern ist es möglich, für eine Reihe von Objektimplementierungen mit ähnlichen Anforderungen ein optimales Interface zum ORB anzubieten. War in den Anfangstagen von CORBA der *Basic Object Adapter* (BOA) die erste Wahl für nahezu alle CORBA-Implementierungen, so existiert heute mit dem *Portable Object Adapter* (POA) ein im Gegensatz zum BOA zwischen ORB-Implementierungen verschiedener Hersteller portabler und leistungsfähiger Objektadapter, der den BOA aus allen Bereichen verdrängt hat.

1.3 Interface Definition Language

Die Interface Definition Language ermöglicht die programmiersprachenunabhängige Beschreibung von Objektinterfaces.

Eines der grundlegenden Prinzipien bei der Entwicklung von CORBA-Anwendungen ist die Unabhängigkeit von der benutzten Implementierungssprache. So gibt es keinen technologischen Grund für den Programmierer, die Benutzung (Klient) und die Realisierung eines Dienstes (Server) in der gleichen Sprache zu vollziehen. Vielmehr kann er für den jeweiligen Kontext eine passende Sprache auswählen. Häufig ist es beispielsweise der Fall, dass im Klienten eine Sprache wie Java benutzt wird, die eine einfache Integration in graphische Oberflächen ermöglicht. Für die Realisierung von Diensten wird dagegen in vielen Fällen auf Compilersprachen wie C++ zurückgegriffen. Die Unabhängigkeit der Benutzung und Realisierung eines Dienstes muss jedoch auf einer gemeinsamen Basis erfolgen. In einer einzelnen Programmiersprache würde man dazu beispielsweise Deklarationen von Datentypen und programmiersprachlichen Objekten benutzen. Mit CORBA soll aber gerade die Unabhängigkeit von den Programmiersprachen erreicht werden. Somit kann beispielsweise die Schnittstelle eines CORBA-Objekts nicht durch die Bekanntgabe gemeinsam zu nutzender C++-Datentypen ausgedrückt werden, sondern es muss eine Abstraktion von konkreten programmiersprachlichen Konzepten erreicht werden.

CORBA-Interfaces werden durch die Interface Definition Language beschrieben.

Die in CORBA gewählte Lösung ist die *Interface Definition Language* (IDL). IDL ist keine Programmiersprache, sondern eine Sprache zur Beschreibung von Objektschnittstellen, Operationen und Datentypen. Zentrales Konzept in IDL ist das *CORBA-Interface*, das die Schnittstelle eines CORBA-Objekts definiert. Ein CORBA-Interface stellt einen Vertrag zwischen CORBA-Objekt und Klient über die Nutzung dar. In einem CORBA-Interface werden Attribute und Operationen gruppiert. Attribute sind konfigurierbare Werte an einem CORBA-Objekt. Die Operationen, die in einem Interface definiert sind, können von einem Klienten gerufen werden und müssen von dem CORBA-Objekt realisiert werden. Es existieren vier verschiedene Arten von Operationsparametern:

- *in*-Parameter sind Eingabeparameter,
- *inout*-Parameter sind sowohl Eingabeparameter als auch Ausgabeparameter,
- *out*-Parameter sind Ausgabeparameter,
- *return*-Parameter spezifizieren den Rückgabewert (genau einmal vorhanden pro Operation).

Als Parameterdatentypen unterstützt CORBA primitive Datentypen wie *Short*, *Float* und *String*, konstruierte Datentypen wie Strukturen und Aufzählungen und Vorlagedatentypen (*template types*) wie Sequenzen.

Während die ersten Versionen der *Interface Definition Language* nur den Operationsruf per Referenz unterstützten (*call by reference*), wurde in spätere Versionen der Sprache auch der Operationsruf per Wert (*call by value*) integriert. Eine Sonderposition nehmen hierbei die so genannten *Valuetypes* ein. *Valuetypes* vereinen Eigenschaften von Interfaces und Strukturen. Sie unterstützen Einfachvererbung von anderen *Valuetypes*, können einen komplexen Zustand besitzen (d.h. beliebige Graphen mit Rekursionen und Zyklen) und können ein Interface anbieten. Zur Kommunikation von Fehlern bei der Ausführung einer Operation definiert IDL spezielle strukturierte Typen, die Ausnahmen (*Exceptions*) genannt werden.

Lokale Interfaces definieren programmiersprachliche Schnittstellen.

Zusätzlich zu normalen CORBA-Interfaces können mit IDL auch prozess-lokale Interfaces beschrieben werden. Im Unterschied zu einem echten CORBA-Interface, können für diese Interface-Art keine Objektreferenzen generiert und exportiert werden. Lokale Interfaces bieten damit einen eleganten Weg, um programmiersprachliche Schnittstellen in CORBA-Anwendungen zu beschreiben.

1.3.1 Sprachabbildungen

Sprachabbildungen definieren die Abbildung von CORBA-Konstrukten auf programmiersprachliche Konstrukte.

Mit der *Interface Definition Language* ist es möglich, Interfaces von CORBA-Objekten zu beschreiben. Durch die gemeinsame Festlegung auf ein Interface gelingt es dem Klienten und einem CORBA-Objekt ein gemeinsames Verständnis über die Schnittstelle zu einem Dienst zu haben. Mit der Definition des Interfaces ist es jedoch für den Anbieter eines Dienstes nicht getan, sondern er muss auch die Implementierung des Interface bereitstellen. IDL selbst bietet keine Konstrukte zur Beschreibung einer solchen Implementierung eines CORBA-Interface an. Vielmehr definiert der CORBA-Standard für die verschiedenen Konstrukte von IDL entsprechende Sprachabbildungen für die verschiedenen gebräuchlichen Programmiersprachen. Eine solche Sprachabbildung bezieht sich immer auf eine konkrete Programmiersprache und beschreibt, wie CORBA-Interfaces in dieser Programmiersprache zu implementieren sind. Dies ist notwendig, da ORB-Hersteller und Hersteller von CORBA-Objekten in den meisten Fällen verschieden voneinander sind und somit klar festgelegt werden muss, wie der ORB Operationsaufrufe an ein CORBA-Objekt eines Fremdherstellers ausliefern muss.

Sprachabbildungen legen neben der Art und Weise der Implementierung eines CORBA-Interface auch fest, auf welche programmiersprachlichen Datentypen die CORBA-Datentypen abzubilden sind. Beispielsweise ist für den CORBA-Datentyp *Short* festgelegt, dass er mindestens einen Wertebereich von 16 Bit besitzt. Für Programmiersprachen, in denen nur Datentypen mit einem Wertebereich von 32 Bit existieren, ist es also erlaubt, den CORBA-Datentyp *Short* auch auf einen solchen programmiersprachlichen Datentyp abzubilden. Insbesondere kann es bei diesen Sprachabbildungen dazu kommen, dass zwei in CORBA verschiedene Datentypen auf denselben programmiersprachlichen Datentyp abgebildet werden. Es existieren inzwischen Sprachabbildungen für fast jede gegenwärtig verwendete Programmiersprache, sei es Compilersprache oder Interpretersprache.

Bei der Definition der Sprachabbildungen in die unterschiedlichen Programmiersprachen wurde versucht, die Abbildung möglichst natürlich in die jeweilige Sprache einzubetten. So werden die IDL-Schnittstellen normalerweise immer auf das Schnittstellenkonzept der Sprache abgebildet. Für C++ bedeutet das, dass das Interface-Konzept von CORBA auf ein entsprechendes Klassen-Interface einer C++-Klasse abgebildet wird. Bei Sprachen, die keine objektorientierten Konzepte unterstützen, wird die Abbildung jedoch häufig schwierig und schlecht handhabbar, da diese Konzepte dann nachgebildet werden müssen.

Die Transformation einer IDL-Beschreibung eines CORBA-Interface in die entsprechenden Konstrukte einer bestimmten Programmiersprache erfolgt in der Regel automatisch durch ein spezielles Werkzeug, das vom ORB-Hersteller bereitgestellt wird. Dieses Werkzeug wird IDL-Compiler genannt. Ein solcher Compiler generiert alle zur Implementierung eines CORBA-Objekts notwendigen programmiersprachlichen Konstrukte, beispielsweise der IDL-*Skeletons*, inklusive aller Hilfskonstrukte, die nicht direkt an der Implementierung eines CORBA-Objekts beteiligt sind. Darüber hinaus erzeugt er auch die typspezifischen IDL-*Stubs* für die Verwendung des Interface durch einen Klienten.