Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Fachgebiet Telematik
Prof. Dr. D. Reschke

Hauptseminararbeit

.NET als alternative Middleware-Technologie

Referent: Prof. Dr. Dietrich Reschke
Betreuer: Dipl.-Inf. Thorsten Strufe
Bearbeiter: Jens Zimmermann
Abgabetermin: 05. Juli 2002

Inhaltsverzeichnis

1. Einführung und Überblick 1

2. Grundlagen und Bestandteile der .NET-Plattform 1

2.1. Historische Entwicklung 1

2.2. Bestandteile der .NET-Plattform 2

3. Das .NET-Framework 3

3.1. Die Common Language Runtime 3

3.1.1. Einführung 3
3.1.2. Beschreibung der Laufzeitumgebung 3
3.1.3. Das gemeinsame Typsystem 4
3.1.4. Assemblies 6

3.2. Die gemeinsame Klassenbibliothek 6

4. Web Services 7

5. .NET-Remoting 8

5.1. Einführung 8

5.2. Technische Grundlagen 8

5.2.1. Arten von Remoting-Objekten 8
5.2.2. Remote- und Proxyobjekte 9
5.2.3. Channel 9
5.2.4. Objekthosting 10
5.2.5. Remotingszenarien 11

5.3. Beispiel 13

5.3.1. Einführung 13
5.3.2. Entwicklung der .NET -Anwendung 13
5.3.3. Umwandlung in eine Remote-Anwendung 14

6. .NET im Vergleich mit anderen Technologien 15

6.1. Einführung 15

6.2. Vergleich mit der Java 2 Enterprise Edition 15

6.3. Middleware -Ansätze 15

7. Fazit 17

Abbildungsverzeichnis

Abbildung 1: Entwicklung zu .NET 2
Abbildung 2: Methodenaufruf in .NET 4
Abbildung 3: Sprachintegration 4
Abbildung 4: Komponentenkommunikation mit COM und .NET 5
Abbildung 5: Datentypen im .NET 6
Abbildung 6: Klassenbibliothek der CLR 7
Abbildung 7: Zugriff auf Remoteobjekte über HTTP (vgl. [Sirinivasan 2000]) 12
Abbildung 8: Zugriff auf Remoteobjekte über TCP (vgl. [Srinivasan 2000]) 12
Abbildung 9: Kommunikation mit DCOM-Objekten 13
Abbildung 10: Entwicklung der CORBA-Anwendung, vgl. [Gruhn u.a. 2000] 16

Tabellenverzeichnis

Tabelle 1: Besonderheiten der Internet-Generationen 2
Tabelle 2 :.NET-Remotingszenarios (vgl. [Srinivasan 2000]) 12
Tabelle 3 : Gemeinsame Merkmale von .NET und J2EE (vgl. [Vawter u.a. 2001]) 15

1. Einführung und Überblick

Mittlerweile ist Microsoft .NET überall im Gespräch und es gibt die ersten Produkte auf .NET-Basis. Neben der Entwicklungs- und Laufzeitumgebung von Microsoft haben schon andere Firmen Produkte auf Basis von .NET präsentiert. In der vorliegenden Arbeit wird die .NET-Technologie kurz vorgestellt und anhand eines kleinen Beispiels das .NET-Remoting und die Entwicklung mit dem Microsoft .NET Framework SDK (Software Development Kit) erläutert.

Zur Entwicklung von .NET-Anwendungen stellt Microsoft zwei Pakete zur Verfügung. Zum einen kann man die Programme mit dem kostenlosen .NET Framework SDK erstellen, welches die vollständige Funktionalität hat und auf der Homepage von Microsoft heruntergeladen werden kann. Zum anderen kann man Visual Studio .NET, welche der Nachfolger des Visual Studio 6 ist, zur Entwicklung einsetzen. Allerdings muß diese Entwicklungsumgebung käuflich erworben werden. Man hat aber den Vorteil einer von den Vorgängerversionen her bekannten, leistungsstarken und nutzerfreundlichen Entwicklungsumgebung. Auch enthält sie, im Gegensatz zum Framework SDK, das vollständige Hilfesystem MSDN (Microsoft Developer Network).

Microsoft .NET ist aber mehr als nur ein Nachfolger der Microsoft-Entwicklungstools. Es handelt sich hierbei um einen vollständig neuen Ansatz, die sogenannte „.NET-Strategie“ von Microsoft. Es sollen PCs mit den modernen Gadgets, wie Mobiltelefon und PDA, verheiratet werden. Die tragende Rolle als gemeinsames Kommunikationsmittel spielt hierbei das Internet. Microsoft .NET selbst „…besteht aus einem umfangreichen Satz von Werkzeugen zur Softwareentwicklung fürs Internet, für Windows-PCs und für alle erdenklichen anderen Gerätschaften“ [Siering 2002]. Alle von Microsoft verfügbaren Server werden in naher Zukunft als .NET-Versionen veröffentlicht und mit den „.NET myServices“ unternimmt Microsoft einen erneuten Anlauf, verschiedenste Dienste auf Basis von .NET zu verkaufen. Ein Beispiel hierfür ist Microsoft Passport.

2. Grundlagen und Bestandteile der .NET-Plattform

2.1. Historische Entwicklung

Den Anfang der Entwicklung von Komponenten zur Kommunikation von Rechnern untereinander begann im Jahr 1992 mit der Einbindung von Client/Server-Techniken in das Betriebssystem von Microsoft. Parallel dazu erfolgte 1996 die Vorstellung des Internet Explorers (IE) und des Internet Information Servers (IIS), welche eine neue Generation von Software darstellten. Microsoft selbst bezeichnet sie als 1. Generation von Internet-Software. Die 2. Generation stellt Windows DNA (Distributed Internet Applications Architecture) dar. Hierbei handelt es sich um eine Architektur zur Entwicklung von Verteilten Anwendungen und Produkten, auf denen diese Architektur aufsetzt (Bsp. Windows 2000 mit COM+ als Application Server und Visual Studio als Enwicklungsumgebung). Einer der wichtigsten Bestandteile sind COM (Component Object Model) bzw. DCOM (Distributed COM) und dessen Nachfolger COM+. Näheres zu Windows DNA ist unter [Willers 2000] nachzulesen. Die beiden Stränge wurden 2000 durch die Vorstellung von .NET vereinigt, was die nachfolgende Abbildung nochmals grafisch darstellt. Der Tabelle 1 sind die Besonderheiten der drei Internet-Generationen zu entnehmen.

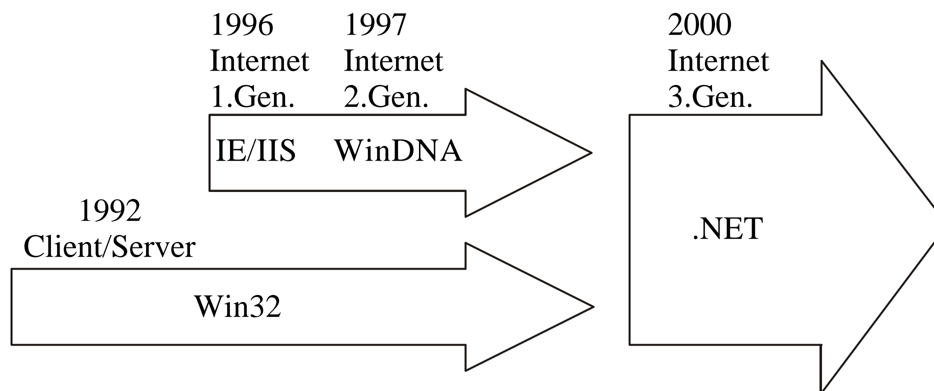

Abbildung 1: Entwicklung zu .NET

1. Generation 1994-1996	2. Generation 1996-2000	3. Generation 2000+
Statische Seiten	Dynamische Seiten	Nutzung von Diensten
E-Mail, Basisinformationen	Personalisierung, E-Commerce	Web Services
IE/IIS	Windows DNA	Microsoft .NET

Tabelle 1: Besonderheiten der Internet-Generationen

2.2. Bestandteile der .NET-Plattform

Die .NET-Plattform besteht aus vier wesentlichen Bestandteilen, zu denen

- Framework und Tools,
- Building Block Services,
- Enterprise Server und
- Mobile Devices gehören. [Willers 2000]

Zum Framework und zu den Tools gehören die entsprechende Entwicklungsumgebung und eine moderne Klassenbibliothek. Im nachfolgenden wird auf diesen Punkt genauer eingegangen. Unter Building Block Services versteht man „vorgefertigte Web Services, die man direkt als Komponente in seine Programme einbinden kann" [Willers 2000] und natürlich auch selbst entwickelte Web Services. Die Infrastruktur für den Betrieb von Web Services wird durch die Enterprise Server zur Verfügung gestellt. Hierbei handelt es sich um die aktuell verfügbaren Microsoft-Server mit der .NET-Erweiterung. Der letzte Bestandteil der .NET-Plattform sind die Mobile Devices, zu denen Geräte gehören, auf denen als Betriebssystem Windows CE läuft. Auf ihnen sollen zukünftig auch .NET-Anwendungen laufen.

Das .NET-Framework ist die neue Entwicklungsplattform für .NET-Anwendungen. Microsoft beschreibt sie als: „platform for building, deploying and running Web Services and applications". Die Bestandteile des .NET-Frameworks sind

- die Common Language Runtime (CLR),
- die gemeinsamen Klassenbibliothek und
- die Active Server Pages (ASP.NET).

Ferner gehören zum Framework die zwei Entwicklungsumgebungen Visual Studio .NET und das .NET Framework SDK sowie die Web Services und das .NET Remoting.

Im weiteren wird speziell auf die CLR, die gemeinsame Klassenbibliothek und das .NET Remoting eingegangen. Web Services werden aufgrund der Komplexität nur kurz vorgestellt.

3. Das .NET-Framework

3.1. Die Common Language Runtime

3.1.1. Einführung

Ein großer Nachteil der bisherigen Modelle zur Kommunikation und Integration von Softwarekomponenten (COM, DCOM und COM+) war die Notwendigkeit der Integration eines Layers, der das Typsystem von COM implementiert. Darüber hinaus mußte die Sprache an COM angepaßt werden. Hierfür war die Konvertierung von Sprachtypen in COM-Typen und umgekehrt notwendig. Des weiteren mußte der Entwickler zusätzlichen Code programmieren, welcher den Aufrufkonventionen von COM genügt. Die dadurch notwendige zusätzliche Programmierung macht den Code komplex und fehleranfälliger. Die CLR setzt hier an und bietet ein einheitliches Integrationsmodell. (vgl. [Willers 2001])

3.1.2. Beschreibung der Laufzeitumgebung

Ähnlich wie bei Java erzeugt der Compiler keinen nativen Code mehr, sondern plattformunabhängigen Zwischencode. Das heißt, für die Ausführung der entwickelten Programme ist eine Laufzeitumgebung notwendig, die CLR. Als einziger Compiler kann der Visual C++-Compiler weiterhin nativen Code erzeugen, damit Kernel und Treiber entwickelt werden können, für die eine Laufzeitumgebung ungeeignet ist, da maximale Performance zwingend erforderlich und die Plattformabhängigkeit gegeben ist. Die automatische Speicherverwaltung wird, wie bei Java, durch eine Garbage Collection, für welche die CLR verantwortlich ist, vorgenommen.

Der Compiler erzeugt eine plattform- und prozessorunabhängige Zwischensprache, die bei der Ausführung durch einen Just-in-time-Compiler in nativen Code (Maschinencode) umgewandelt und ausgeführt wird. Diese Zwischensprache wird als Microsoft Intermediate Language (MSIL) bezeichnet. Jeder .NET-Compiler erzeugt diesen Code, unabhängig von der zugrundeliegenden Programmiersprache. Dieser Code wird dann unter Aufsicht der CLR ausgeführt und als Managed Code bezeichnet. Bestimmte Aktionen, wie das Anlegen eines Objekts oder der Aufruf einer Methode, erfolgen hier nicht direkt, sondern werden an die CLR delegiert, die wiederum zusätzliche Dienste ausführen kann. Hierzu zählt z.B. die Versionsüberprüfung. Die nachfolgende Abbildung zeigt das Vorgehen beim Aufruf der Methode einer Klasse aus einer anderen Klasse heraus.

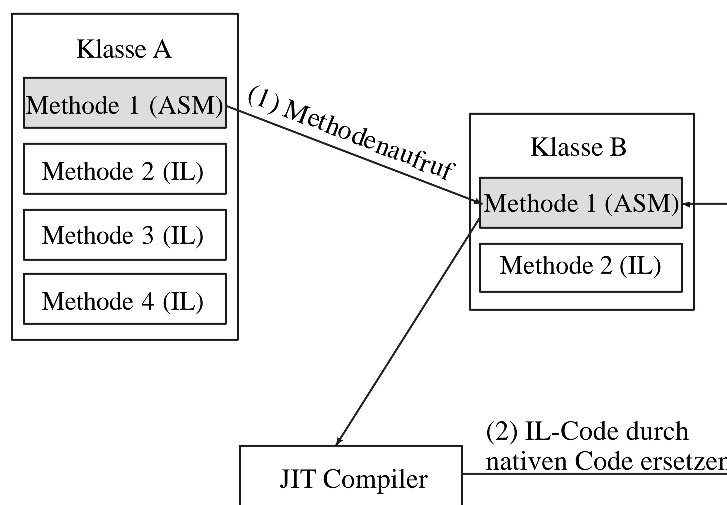

Abbildung 2: Methodenaufruf in .NET

Die Umwandlung von MSIL-Code in Maschinencode gewährleistet, daß so immer die schnellstmögliche Ausführungsgeschwindigkeit gegeben ist.
Microsoft liefert sowohl mit dem Framework SDK, wie auch mit Visual Studio .NET Compiler für Visual Basic, C++ und der neuen Programmiersprache C# aus. Fremdanbieter haben auch schon Compiler für andere Programmiersprachen, wie Cobol, Pascal, Eiffel und Small-Talk angekündig, da die MSIL komplett offengelegt und standardisiert ist. Ein funktionstüchtiger Pascal-Compiler im Beta-Statium ist bereits im Internet verfügbar. Die nachfolgende Abbildung stellt die Sprachintegration auf Codeebene noch einmal kurz dar.

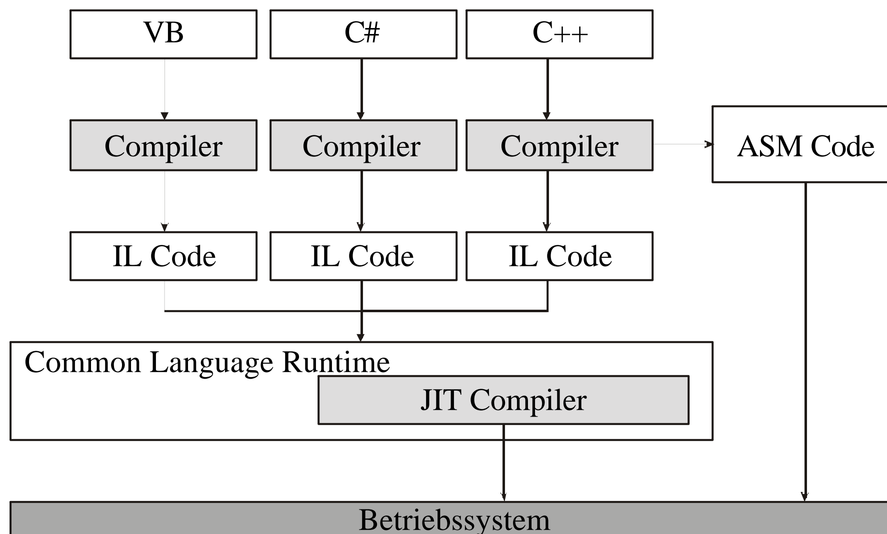

Abbildung 3: Sprachintegration

Ein großer Vorteil der Integration auf Codeebene ist, daß jeder Programmierer mit der Programmiersprache entwickeln kann, die ihm am besten liegt. Einzige Voraussetzung dafür ist, daß der Compiler MSIL-Code erzeugen kann. Es ist so auch möglich, eine Klasse in einer Programmiersprache zu entwickeln und sie dann in einer anderen Programmiersprache als Basisklasse einzusetzen, d.h. von dieser Klasse andere abzuleiten.
Microsoft verfolgt allerdings mit der Plattformunabhängigkeit nicht das Ziel, daß die entwikkelten Programmen unter verschiedenen Betriebssystemen (Unix, Linux, Windows oder Be-OS) laufen, sondern auf unterschiedlichen Geräten mit einem Betriebssystem von Microsoft (Windows, Windows CE etc.). Da die CLR vollständig offengelegt ist, besteht aber die Möglichkeit, daß andere Firmen eine Laufzeitumgebung für ihr Betriebssystem entwickeln. Erste Versuche in diese Richtung gibt es auch schon, wozu z.B. das Projekt Mono zählt (http://www.go-mono.org).

3.1.3. Das gemeinsame Typsystem

Jede .NET-Sprache nutzt dasselbe Typsystem, welches durch die CLR zur Verfügung gestellt wird und sich Common Type System (CTS) nennt. Damit wandert das Typsystem vom Compiler in die Laufzeitumgebung und ist nicht mehr Bestandteil der jeweiligen Programmiersprache. Es gibt somit nicht mehr verschiedene Repräsentationen von ein und demselben Datentyp. So ist die Größe eines „normalen" Integers in C++ genauso wie in C# und unabhängig vom zugrundeliegenden Betriebssystem. Auch das Problem mit der Konvertierung von Zeichenketten wurde so beseitigt.

Durch das gemeinsame Typsystem entfällt auch die Konvertierung und Anpassung an COM-Aufrufkonventionen. Komponenten unterschiedlicher Programmiersprachen können so problemlos miteinander kommunizieren. Den Unterschied zwischen der Kommunikation von COM-Komponenten auf der einen und .NET-Komponenten auf der anderen Seite wird durch die nachfolgende Abbildung dargestellt.

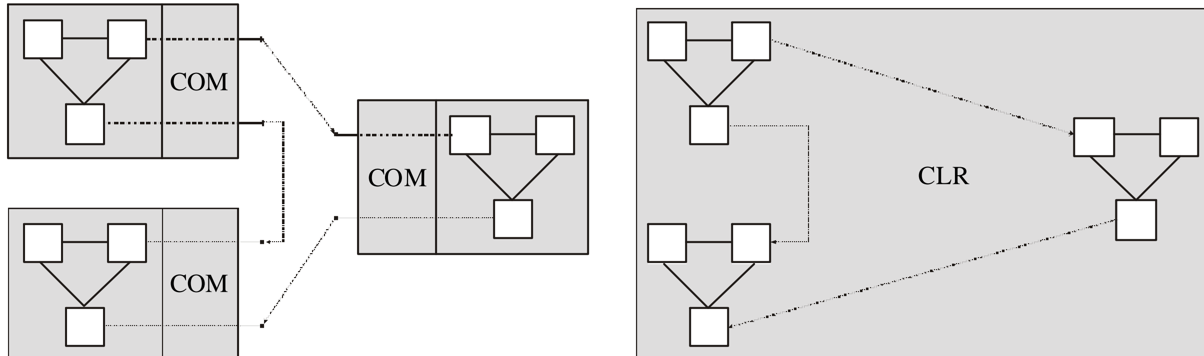

Abbildung 4: Komponentenkommunikation mit COM und .NET

Durch das gemeinsame Typsystem entfällt somit der fehleranfällige Konvertierungscode. Alte COM-Komponenten können durch .NET trotzdem noch angesprochen werden und .NET-Komponenten können über COM und DCOM angesprochen werden. Notwendig hierfür ist allerdings ein Mapping.

Sämtliche Datentypen des CTS, welche als Managed Types bezeichnet werden, werden alle vom Typ System.Object abgeleitet, so daß es sich bei allen Typen um Objekte handelt. Allerdings wird zwischen zwei Arten von Objekten unterschieden. Zum einen gibt es Value Types, die

- auf dem Stack angelegt werden,
- Daten enthalten,
- nicht den Wert Null annehmen können und
- im wesentlichen primitive Datentypen (int, float, char etc.), Aufzählungen und Strukturen repräsentieren.

Zum anderen gibt es Reference Types, die

- auf dem Heap angelegt werden,
- Referenzen auf Objekte enthalten,
- den Wert Null annehmen können und
- im wesentlichen Zeichenketten, Klassen und Felder repräsentieren. (vgl. [Willers 2001])

Wenn ein Value Type eine Objektmethode aufruft bzw. eine Methode des Value Types aufgerufen wird, wird durch die CLR automatisch ein temporäres Objekt auf dem Heap angelegt und der Wert des Value Types kopiert. Danach erfolgt der Methodenaufruf. Nachdem die Methode abgearbeitet ist, wird das temporäre Objekt wieder vom Heap entfernt. Diese Techniken werden als Boxing (Konvertierung eines Reference Types in einen Value Type) und Unboxing (Konvertierung eines Value Types in einen Reference Type) bezeichnet. Nachfolgene Abbildung zeigt einige Vertreter der beiden Objektarten.

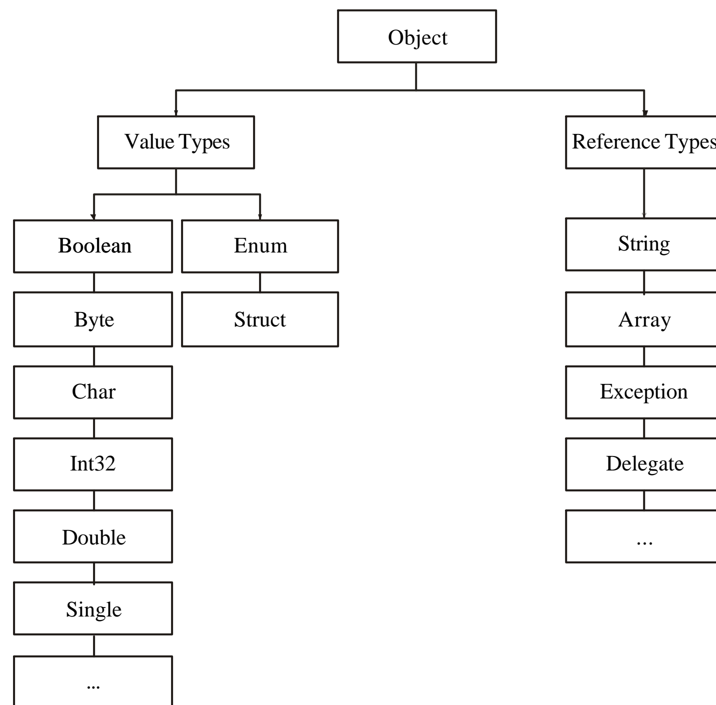

Abbildung 5: Datentypen im .NET

Beim Übersetzen durch den Compiler werden Metadaten in die Komponente geschrieben, welche die Beschreibung sämtlicher Typen der Komponente enthalten. „Dazu zählen in der Regel Schnittstellen, Klassen und deren Member-Variablen“ [Willers 2001]. Diese Metadaten können während der Laufzeit ausgelesen werden, was als Reflection bezeichnet wird.

3.1.4. Assemblies

Die Versionierung ist eines der größten Probleme in der COM-Welt. Es besteht nicht die Möglichkeit, mehrere Versionen einer Komponente zu installieren. Nutzen mehrere Programme die gleiche Komponente und wird diese Komponente aktualisiert oder verändert, kann es vorkommen, daß es dabei zu Problemen kommt. Dieses Phänomen wird als „DLL-Hölle“ bezeichnet. Mit .NET hat Microsoft ein Verfahren eingeführt, welches dieses Problem lösen soll. Alle Komponenten, die ein Programm benötigt, werden als Assembly bezeichnet. Jedes Assembly verfügt dabei über Metadaten, welche die Abhängigkeit der Komponente beschreiben und als Manifest bezeichnet wird. Die einfachste Form eines Assembly ist ein Private Assembly, bei dem alle benötigten Komponenten in das Verzeichnis oder ein Unterverzeichnis der Anwendung kopiert werden. So kann jede Anwendung mit einer eigenen Version der Komponente arbeiten, man muß allerdings mit möglichen Redundanzen von Komponenten rechnen. Über eine Konfigurationsdatei (.CFG) im XML-Format gibt man hier an, wo sich die entsprechenden Komponenten befinden. Diese Konfigurationsdatei muß denselben Namen tragen, wie die Anwendung, und sich im gleichen Verzeichnis befinden.
Wenn mehrere Anwendungen eine Komponente benutzen sollen, müssen diese in einem Shared Assembly zusammengefaßt werden. Im Gegensatz zu COM sind die Komponenteninformationen nicht mehr systemweit gültig und in der Registry vermerkt, sondern es befinden sich die entsprechenden Informationen anwendungsspezifisch in Konfigurationsdateien.

3.2. Die gemeinsame Klassenbibliothek

Alle .NET-Compiler benutzen die umfangreiche Klassenbibliothek der CLR. So kann von allen Sprachen aus auf dieselben Klassen zurückgegriffen werden. Diese Bibliothek bietet

umfangreiche Basisklasse von der Bearbeitung von Zeichenkette über die Arbeit mit Threads, bis hin zur Fehlerbehandlung durch Exceptions. Zugriffe auf das Betriebssystem und das „Win32-API" erfolgen nicht mehr direkt, sondern werden über Klassen abstrahiert (vgl. [Willers 2001]). Die heute gebräuchlichen Bibliotheken, wie das Windows-API, die Microsoft Foundation Classes (MFC) oder die ODBC, werden alle durch die Klassenbibliothek ersetzt. Die nachfolgende Abbildung stellt die Klassenbibliothek der CLR grafisch dar.

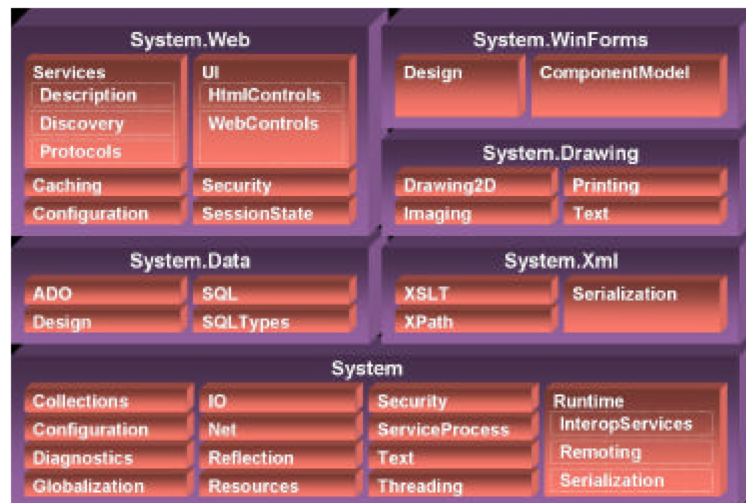

Abbildung 6: Klassenbibliothek der CLR

Wie der Abbildung zu entnehmen ist, gibt es nur eine einzige große Klassenbibliothek, die Klassenbibliothek System. Sie teilt sich nochmals in fünf große Unterbereiche auf. Zu ihnen gehören Bibliotheken zur Erstellung von grafischen Windowsanwendungen (System.WinForms), Web Services (System.Web) und Datenbankarbeit (System.Data). Andere wichtige Bibliotheken findet man direkt unter System, zu ihnen gehören Bibliotheken, die zur Erstellung von verteilten Anwendungen (Net), zur Ein- und Ausgabe von Daten (IO) und zur Erstellung von Anwendungen mit nebenläufigem Verhalten (Threading) benötigt werden.
Im Gegensatz zur CLR sind nur Teile dieser Klassenbibliothek offen gelegt.

4. Web Services

Ein Web Service ist eine Anwendung, die ihre Methoden über eine Schnittstelle nach außen, z.B. über das Inter- oder das Intranet, zur Verfügung stellt, die Implementierung dieser Methoden für den Client aber vollständig transparent ist. Vereinfacht gesagt sind Web Sercvices Software-Komponenten, die über das Internet dynamisch miteinander verknüpft werden (vgl. [Schmidt 2002]). Die Informationen, die ein Webserver hat, können so von anderen Programmen genutzt werden. Der Web Service wird von einem Client über das Internet mit einem Unified Resource Locator (URL) angesprochen und der Aufruf der Methoden dieses Web Service erfolgt mittels des Simple Object Access Protocol (SOAP). Das hat den Vorteil gegenüber den objektspezifischen Protokollen, wie DCOM oder IIOP, daß beim Client und beim Server keine homogene Infrastruktur benötigt wird. Der Client schickt eine per XML verpackte Nachricht mittels des Hypertext Transfer Protocol (HTTP) zum Server und dieser antwortet ebenfalls mit einer XML-Nachricht. SOAP „definiert, wie die XML-Nachrichten aufgebaut sein müssen und wie die Aufruffolge auszusehen hat" [Willers 2000]. Dadurch ist die Plattformunabhängigkeit gewährleistet. Näheres zu SOAP ist unter [Brien 2002] nachzulesen.
Da die Kommunikation über HTTP erfolgt, können auch über eine Firewall hinweg die Dienste eines Web Service in Anspruch genommen werden, da nur der Port für das HTTP-

Protokoll (in der Regel Port 80) freigegeben sein muß. Web Services können auch miteinander zu einer verteilten Web-Applikation kombiniert werden. Hier werden die verschiedenen Web Services geeignet kombiniert und die Ausgabedaten eines Web Service können als Eingabedaten eines anderen Web Service dienen. Als Beispiel soll hier die Planung einer Reise dienen. Über einen Web Service kann man die Flugpläne verschiedener Fluggesellschaften abfragen und den Flug mit Hilfe eines anderen Web Service buchen. Zum Schluß erfolgt die Buchung eines Hotels mit ebenfalls einem anderen Web Service. (vgl. [Willers 2000])

5. .NET-Remoting

5.1. Einführung

„Microsoft .NET Remoting ist ein Framework, das es Objekten ermöglicht, anwendungsdomänenübergreifend miteinander zu kommunizieren." [Obermeyer u.a. 2000] Dazu enthält das Framework eine Reihe von Diensten, wie z.B. Aktivierung und Objektlebensdauer oder Kommunikationschannel, die für die Datenübertragen zwischen Client und Server verantwortlich sind. Vor der Übertragung werden diese Nachrichten codiert und nach dem Empfang wieder decodiert. Hierbei ist eine Binärcodierung bei leistungskritischen Daten oder eine XML-Codierung zur Kommunikation mit anderen Remoting-Frameworks möglich. Wie bei den Web Services wird für den Transport der XML-codierten Daten das SOAP-Protokoll verwendet.

5.2. Technische Grundlagen

5.2.1. Arten von Remoting-Objekten

Es gibt drei Arten von Remoting-Objekten, die sich in zwei Oberklassen zusammenfassen lassen:

- Clientaktivierte Objekte (CAOs) und
- Serveraktivierte Objekte.

Der Unterschied zwischen den beiden liegt in der Verwaltung der Lebensdauer des entsprechenden Remoting-Objektes. Bei Clientaktivierten Objekten erfolgt die Verwaltung der Lebensdauer durch einen Lebensdauer-Manager, der sicherstellt, daß das Objekt in die Garbage Collection aufgenommen wird, sobald es nicht mehr benötigt wird. Der Fall liegt zum Beispiel dann vor, wenn keine gültige Referenz auf das Remoteobjekt mehr vorliegt. Bei Serveraktivierten Objekten kann man zwischen zwei Modellen wählen:

- Single Call-Objekte und
- Singleton-Objekte.

Single Call-Objekte werden für den Eingang von einzelnen Anforderungen angelegt. Hierbei können keine Daten zwischen den einzelnen Methodenaufrufen gespeichert werden, da die Objekte bei jedem Aufruf neu instanziiert werden. Die Attribute des entsprechenden Objektes haben somit bei jedem Methodenaufruf die selben Werte (Initialwerte). Singleton-Objekte hingegen werden nur einmalig angelegt und speichern den Status zwischen den einzelnen Aufrufen. So können unterschiedliche Clients mit den selben Daten arbeiten, da das Objekt nur einmal angelegt und bei jeder Objekt-Anforderung eines Clients eine Referenz auf dieses Objekt zurückgeliefert wird.

Clientaktivierte Objekte werden auf Anforderung eines Clients aktiviert, wobei das Verfahren mehr oder weniger der herkömmlichen COM-Aktivierung entspricht. Sobald der Client mit Hilfe des new-Operators ein Remoteobjekt anfordert, wird serverseitig ein Instanz der geforderten Klasse angelegt und eine Objektreferenz (ObjRef) zurückgeliefert. Mit dieser Objektreferenz wird auf der Clientseite ein Proxy erstellt, auf dem die Methodenaufrufe des Clients

ausgeführt werden. Zwischen den einzelnen Methodenaufrufen wird der Status des Objekts für den jeweiligen Client gespeichert, so daß die Attributwerte erhalten bleiben. Verschiedene Clients greifen allerdings nicht, wie bei den Singleton-Objekten, auf das selbe Objekt zu, da mit jedem clientseitigen Aufruf des new-Operators ein neues Objekt serverseitig angelegt und durch den Proxy verwaltet wird.

5.2.2. Remote- und Proxyobjekte

Um ein Objekt in ein Remoteobjekt umzuwandeln, muß es von der Klasse MarshalByRef-Object abgeleitet werden. Nach der Aktivierung dieses Objektes durch den Client erhält der Client entsprechend einen Proxy zum Remoteobjekt, welcher die Operationen zum Remoteobjekt umleitet. Wird das Objekt von der Klasse MarshalByValue abgeleitet, wird eine vollständige Kopie des Objektes angelegt, wenn es von einer Anwendung in eine andere übergeht. Es gibt drei verschiedene Arten, um Objekte zwischen Anwendungen zu übergeben:

- als Parameter in Methodenaufrufen,
- als Rückgabewert von Methodenaufrufen und
- als Werte, die sich aus dem Eigenschafts- oder Feldzugriff einer .NET-Komponente ergeben. [Srinivasan 2000]

Befinden sich nach der Übergabe einer Referenz auf ein Remoteobjekt zwischen zwei Anwendungen das Objekt und der Proxy in derselben Anwendungsdomäne, so wird der Methodenaufruf nicht in eine Nachricht konvertiert und zur Remote-Anwendung geschickt, sondern durch den Proxy direkt ausgeführt.

Die Erstellung eines Proxyobjektes erfolgt, wenn ein Client ein Remoteobjekt aktiviert. Es dient als Vermittler zwischen dem Client und den Remoteobjekten und stellt die Weiterleitung der Aufrufe an die entsprechende Remoteobjektinstanz sicher.

5.2.3. Channel

Channel dienen dem Nachrichtentransport an und von Remoteobjekten. Erfolgt ein Aufruf einer Methode des Remoteobjekts, werden die Parameter und andere Details, die sich auf den Methodenaufruf beziehen, über diesen Channel zum Remoteobjekt transportiert. Die Ergebnisse des Methodenufrufs werden auf dieselbe Art und Weise an den Aufrufer zurückgeschickt.

Remoteobjekte nutzen Channel gemeinsam und können einen Channel nicht besitzen. Die Serveranwendung, welche für das Hosten der Remoteobjekte verantwortlich ist, muß die benötigten Channel und die dazugehörigen Objekte registrieren. Sobald ein Channel registriert ist, beginnt er automatisch den angegebenen Anschluß auf Clientanfragen zu überwachen. So ist es auch möglich, ein SingleCall-Objekt zu erreichen, welches erst beim Methodenaufruf erzeugt wird. Nach der Registrierung eines Remoteobjektes wird eine Referenz auf das entsprechende Objekt erstellt und in einer Tabelle gespeichert. Sobald eine Anfrage eines Clients auf einem Channel eintrifft, wird diese durch das Remoting-Framework untersucht, um das Zielobjekt zu ermitteln. Dazu wird die Tabelle nach dem jeweiligen Objektverweis hin durchsucht und der Aufruf an das Objekt weitergeleitet. Exisitiert keine Referenz auf das gewünschte Objekt, wird es durch das Remoting-Framework aktiviert und danach der Aufruf an das Objekt weitergeleitet.

Bei der Kommunikation von Objekten über ein Netzwerk spielt bei einigen Anwendungen die Datensicherheit eine große Rolle. Hierzu müssen vom Entwickler entsprechende Implementierungen, wie Autorisierung oder Verschlüsselung hinzugefügt werden können. Hierfür lassen sich die Channels entsprechend anpassen. Bevor die Nachricht an das Remoteobjekt durch die Serveranwendung weitergeleitet wird, passiert es verschiedene Instanzen (SecuritySink, TransportSink und FormatterSink). Hier können die entsprechenden Sicherheitsvorkehrungen getroffen werden. Nach Bearbeitung der Anfrage durch das Remoteobjekt werden

diese Instanzen wieder durchlaufen, diesmal in umgekehrter Reihenfolge, und danach an das aufrufende Objekt zurückgeschickt.
Es gibt zwei verschiedene Arten von Channels, über welche die Nachrichten transportiert werden können:

- HTTP-Channel und
- TCP-Channel.

„Der HTTP-Channel transportiert Nachrichten an und von Remoteobjekten über das SOAP-Protokoll." [Obermeyer u.a. 2000] Diese Nachrichten werden vor dem Transport durch den SOAP-Formatierer nach XML konvertiert, serialisiert und mit dem erforderlichen SOAP-Header versehen. Sollen die Daten in einem Binärdatenstrom transportiert werden, muß der Binärformatierer mit angegeben werden. Anschließend wird der Datenstrom über das HTTP-Protokoll zum Ziel-URI (Unified Ressource Identifier) transportiert. Wenn die Daten über einen TCP-Channel übertragen werden sollen, wird das TCP-Protokoll zum Transport der Daten zum Ziel-URI verwendet. Es wird ein Binärformatierer verwendet, um die Nachrichten in einen Binärstrom zu serialisieren und zu transportieren.

5.2.4. Objekthosting

Alle Remoteobjekte müssen, bevor ein Client darauf zugreifen kann, im Remoting-Framework registriert sein. Diese Registrierung wird von sogenannten Hostinganwendungen durchgeführt. Folgende Aktionen führen Hostinganwendungen aus:

- Start der Hostinganwendung,
- Registrierung von einem oder mehreren Channels zum Nachrichtentransport,
- Registrierung von einem oder mehreren Remotingobjekten und
- Warten auf Beendigung.

Nachdem die Hostinganwendung beendet ist, sind die registrierten Channel und Remoteobjekte nicht mehr verfügbar, da sie an den Prozeß gebunden sind, durch den sie gestartet wurden. Objekte und Channel werden automatisch aus den Remotingdiensten entfernt, in denen sie registriert wurden. Folgende Informationen sind bei der Registrierung eines Remoteobjekts im Framework erforderlich:

- Name des Assembly, in dem die Klasse enthalten ist,
- der Typ des Remoteobjekts,
- der Objekt-URI, welchen die Clients zum Auffinden des Objekts verwenden und
- der Objektmodus für die Aktivierung (SingleCall oder Singleton).

Ein Remoteobjekt kann durch den Aufruf von:

- RegisterWellKnownType und Übergabe der oben genannten Informationen als Parameter oder
- ConfigureRemoting und Ablage der Informationen in einer Konfigurationsdatei

registriert werden. Beim Aufruf von ConfigureRemoting wird der Name und der Pfad der Konfigurationsdatei als Parameter übergeben. Beide Funktionen unterscheiden sich nur in der Art und Weise der Informationsbereitstellung und sind sonst äquivalent. Ein Aufruf von RegisterWellKnownType hat folgenden Aufbau:

RegisterWellKnownType (*Assembly*, *Klassenname*, *Objekt-URI*, *Objektmodus*)

Mit der Registrierung eines Objekts wird dieses nicht instanziiert, sondern erst, wenn ein Client versucht, eine Methode aufzurufen oder das Objekt clientseitig aktiviert wird.
Jeder Client, der die Objekt-URI kennt, kann nun einen Proxy durch die Registrierung eines Channels abrufen und das Objekt aktivieren. Hierzu gibt es drei Möglichkeiten:

- new,
- GetObject und
- CreateInstance.

Die Registrierung eines TCP-Channels erfolgt mit folgender Anweisung:

ChannelServices.RegisterChannel (new TCPChannel);

Ein Remoteobjekt kann folgendermaßen aktiviert werden:

Klasse Objektname = (*Klasse*)Activator.GetObject (typeof (*Klasse*), „tcp://*Rechner*: *TCP-Port*/*ObjektName*)

Dabei gibt Klasse die entsprechende Klasse des Remoteobjekts an und Objektname ist der Name, unter dem das Objekt auf dem Server erreichbar ist. Beim Übersetzen des Clients benötigt der Compiler Typinformationen zum Remoteobjekt. Diese können ihm folgendermaßen zur Verfügung gestellt werden:

- Bereitstellen eines Verweises auf das Assembly in der die entsprechende Klasse gespeichert ist,
- Erzeugung einer Schnittstellenklasse für das Remoteobjekt und Verwendung dieser Schnittstelle als Verweis beim Übersetzen oder
- Verwendung des SOAPSUDS-Tools zum Extrahieren der erforderlichen Metadaten aus dem Endpunkt.

Bei der Verwendung des SOAPSUDS-Tools wird eine Verbindung zum Endpunkt hergestellt, die Metadaten extrahiert und ein Assembly oder eine Quellcodedatei erzeugt, die zum Übersetzen des Clients verwendet wird.

Die eben genannten Aufrufe stellen noch keine Verbindung zum Server-Objekt her, da das Framework genügend Informationen enthält, um den Proxy zu erstellen. Die Netzwerkverbindung wird erst hergestellt, wenn der Client eine Methode des Remotingobjekts aufruft. Aus diesem Grund kann auch erst hier das Objekt serverseitig instanziiert werden.

Bei .NET muß der Host eines .NET-Remotingobjekts nicht zwingend ein eigenständiges Programm sein. .NET-Remotingobjekte können durch:

- eine verwaltete ausführbare Datei,
- den Internet Information Service (IIS) oder durch
- den .NET-Komponentendienst

gehostet werden. Die einfachste Möglichkeit bietet hierbei das Hosting über eine .NET-EXE-Datei. Werden Remotingobjekte im IIS gehostet, empfangen sie Nachrichten über den HTTP-Kanal. Durch das Hosting im .NET-Komponentendienst können die Vorteile der verschiedenen COM+-Dienste genutzt werden. In dieser Arbeit wird nur auf das Hosting durch eine verwaltete ausführbare Datei eingegangen. Für nähere Informationen und Beispielen zu den anderen beiden Varianten wird auf die Arbeit von [Srinivasan 2000] verwiesen.

5.2.5. Remotingszenarien

Es gibt unterschiedliche Szenarien, für die das .NET-Remoting eingesetzt werden kann. Es können nicht nur .NET-Komponenten untereinander, sondern auch .NET-Komponenten mit COM-Komponenten oder mit Webdiensten kommunizieren. Folgende Tabelle gibt eine kurze Übersicht über die Kombinationsmöglichkeiten. Dies sind aber nicht alle Möglichkeiten, da das Remoting-Framework erweiterbar ist und man eigene Kommunikationskanäle und Serialisierungsformatierer schreiben kann, um mit beliebigen anderen Komponenten zu kommunizieren.

Client	Server	Nutzlast	Protokoll
.NET-Komponente	.NET-Komponente	SOAP/XML	HTTP
.NET-Komponente	.NET-Komponente	Binär	TCP
Verwaltet/nicht verwaltet	.NET-Webdienst	SOAP/XML	HTTP
.NET-Komponente	Nicht verwaltete herkömmliche COM-Komponente	NDR (Network Data Representation, Netzwerkdatendarstellung)	DCOM
Nicht verwaltete herkömmliche COM-Komponente	.NET-Komponente	NDR	DCOM

Tabelle 2 :.NET-Remotingszenarios (vgl. [Srinivasan 2000])

Es besteht die Möglichkeit, .NET-Remotingobjekte als Web Services offenzulegen. Dazu müssen sie nur im IIS gehostet werden. Der IIS übernimmt hierbei die entscheidende Rolle bei der Kommunikation zwischen Client und Web Service. Somit ist auch ein plattformübegreifender Zugriff auf .NET-Remotingobjekte möglich, der auch nicht durch eine Firewall blockiert werden kann, da der Nachrichtenaustausch über HTTP erfolgt. Einzige Voraussetzung ist die Möglichkeit der Verarbeitung einer WSDL-Datei (Web Service Description Language) auf der Seite des Clients und die dadurch ermöglichten SOAP-Aufrufe. Kommunizieren zwei .NET-Objekte über den HTTP-Kanal miteinander, so wird diese Kommunikation auch nicht durch eine Firewall blockiert. Folgende Abbildung verdeutlicht die beiden Szenarien.

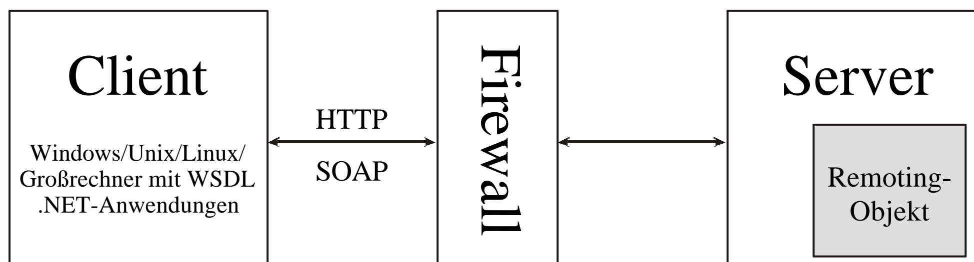

Abbildung 7: Zugriff auf Remoteobjekte über HTTP (vgl. [Sirinivasan 2000])

Kommunizieren zwei .NET-Objekte über einen TCP-Kanal miteinander, so kann in der Regel keine Firewall zwischen den beiden Rechner liegen, auf denen sich die Objekte befinden. Die Ursache hierfür liegt in der Kommunikation über Sockets. Der Vorteil ist allerdings die Vermeidung von Overhead, da die Objekte offengelegt sind.

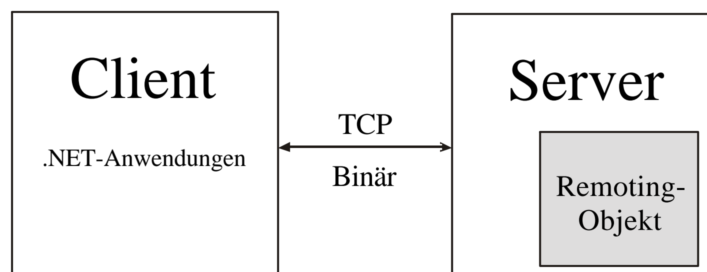

Abbildung 8: Zugriff auf Remoteobjekte über TCP (vgl. [Srinivasan 2000])

Wenn eine .NET-Anwendung mit einer COM-Komponente oder ein COM-Client mit einem .NET-Remotingobjekt kommunizieren soll, müssen die Objekte entsprechend offengelegt

werden. Dies ist notwendig, da COM-Komponenten oder COM-Clients nicht durch die CLR verwaltet werden. Soll ein COM-Objekt durch eine .NET-Anwendung instanziiert werden, wird das COM-Objekt über einen Runtime Callable Wrapper (RCW) offen gelegt. Ein .NET-Remoting-Serverobjekt hingegen wird durch einen COM Callable Wrapper (CCW) für die COM-Anwendung offengelegt. Beide dienen als Proxy für das jeweilige Server-Objekt und kommunizieren über DCOM mit den Clients.Die Wrapper sehen für die .NET-Objekte wie jede andere verwaltete Klasse aus, dienen aber nur als Vermittler zwischen verwaltetem und nicht verwaltetem Code. Folgende Abbildung soll das Wrapper-Konzept verdeutlichen.

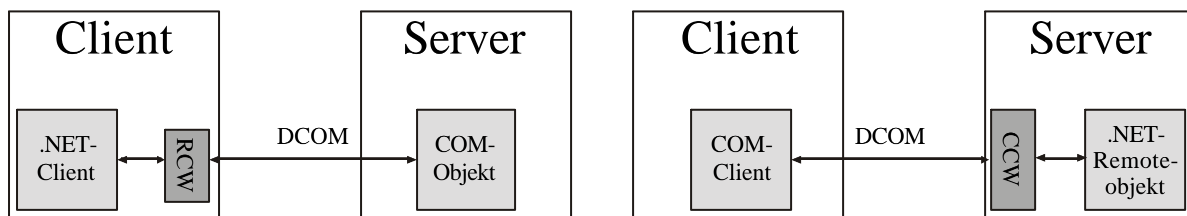

Abbildung 9: Kommunikation mit DCOM-Objekten

5.3. Beispiel

5.3.1. Einführung

Mit einer Beispiel-Implementierung soll der Überblick über .NET abgeschlossen werden. An diesem Beispiel wird kurz das Vorgehen bei der Implementierung einer .NET-Anwendung mit dem .NET Framework SDK erläutert und gezeigt, wie diese Anwendung in eine Remote-Anwendung umgewandelt wird. Dem Anhang A sind die entsprechenden Quelltexte und die Make-Files, die zur Erleichterung der Übersetzung der Anwendung erzeugt wurden, zu entnehmen. Als Programmiersprachen wurden sowohl C#, als auch C++ verwendet, um die Sprachunabhängigkeit von .NET zu verdeutlichen.

5.3.2. Entwicklung der .NET-Anwendung

Auf die Implementierung einer Hello-World-Anwendung wurde verzichtet, da man die grundlegenden Besonderheiten von .NET und des .NET Remoting-Frameworks besser an einer etwas umfangreicheren Anwendung mit Methodenaufrufen und der Nutzung einer Klassenvariablen erläutern kann. So läßt sich besser darstellten, daß man mit dem Remoteobjekt genauso arbeiten kann, wie mit einem lokalen Objekt. Statt dessen wurde die Verwaltung eines Kontos implementiert. Dazu wurde die Klasse „Bank“ entwickelt, welche zur Ablage des aktuellen Kontostandes, der nicht negativ sein darf, eine Klassenvariable enthält. Zur Veränderung und zum Auslesen dieser Klassenvariablen dienen die Methoden „einzahlen“, „abheben“ und „get_kontostand“. Hierbei manipulieren die ersten beiden Methoden die Klassenvariable und die letztere dient als Zugriffsmethode.
Als Programmiersprache wurde C# gewählt, da mit ihr Remote-Anwendungen leichter zu erstellen sind. Grundsätzlich kann man aber auch C++ oder Visual Basic.NET verwenden. Auf die Erläuterung des Quelltextes soll hier verzichtet werden. Eine Einführung in C# bekommt man u.a. unter [Eller 2001] und unter [Breymann u.a. 2002].
Nachdem die Klasse zur Verwaltung eines Kontos in einem beliebigen Texteditor geschrieben wurde, muß sie mit dem mitgelieferten C#-Compiler (csc.exe) übersetzt werden. Da es sich hierbei nicht um eine ausführbare Datei (EXE) handelt, die erzeugt werden soll, sondern um eine Bibliothek (DLL), muß dies dem Compiler über den entsprechenden Parameter mitgeteilt werden. Wenn man dies nicht angibt, bekommt man eine Fehlermeldung, da keine Methode

„Main" existiert, die als Einstiegspunkt für ein ausführbares Programm dient. Zur Übersetzung gibt man folgendes Kommando ein: „csc /target:library Bank.cs". Nach erfolgreicher Übersetzung steht die DLL-Datei zur weiteren Verwendung zur Verfügung.
Um eine Instanz von der Klasse zu erzeugen, wurde eine ausführbare Anwendung entwickelt, die dies realisiert. Mit dieser Anwendung können die Methoden des Objekts beliebig aufgerufen werden. Sie kann einfach mit dem Kommando „csc /R:Bank.dll Client.cs" übersetzt werden, wobei über den Parameter „/R" die zu verwendende Bibliothek angegeben wird. Die Anwendung läßt sich auf jedem Rechner ausführen, auf dem .NET, d.h. die Laufzeitumgebung, installiert ist.

5.3.3. Umwandlung in eine Remote-Anwendung

Um aus der lokalen Anwendung eine Remote-Anwendung zu machen, d.h. das Bank-Objekt für entfernte Aufrufe verfügbar zu machen, muß als erstes eine Hostinganwendung entwickelt werden. Dazu wurde auf eine ausführbare Datei zurückgegriffen, mit der das Objekt dem Remoting-Framework bekannt gemacht wird. Diese Anwendung muß gestartet werden, bevor das Objekt instanziiert werden. Damit innerhalb der Server-Anwendung die Remote-Klassen des .NET-Frameworks verfügbar sind, muß die Bibliothek Remoting eingebunden werden. Für die Channels werden die Klassen der Bibliothek Channels und für den Binärdatenstrom die Bibliothek Channels.TCP benötigt. Danach wird ein TCP-Channel-Objekt erzeugt und beim Remoting-Framework registriert. Als Port für die Kommunikation wird Port 8085 reserviert. Im nächsten Schritt wird das Objekt bekannt gemacht und ihm der Objekt-URI „Bank" zugewiesen. Als Objektmodus für die Aktivierung des Objekts wird Singleton gewählt. Dadurch wird das Objekt einmalig, beim ersten Methodenaufruf durch einen Client, angelegt. Jede Anwendung, die dieses Objekt referenziert, bzw. eine Instanz dieser Klasse anfordert, bekommt eine Referenz auf dieses Objekt. Bei der Auwertung der Klassenvariablen wird dies sichtbar. Sobald die Hostinganwendung beendet wird, ist das Objekt nicht mehr verfügbar und wird zerstört.
Damit die Bank-Klasse als Remote-Klasse verfügbar ist, muß sie von der Klasse MarshalByRefObject abgeleitet werden. Zusätzlich müssen die oben genannten Bibliotheken für das Remoting eingebunden werden. Eine Schnittstellendefinition, wie bei anderen Middleware-Ansätzen, entfällt. Nach der Übersetzung beider Dateien und dem Start des Servers ist die Klasse verfügbar und kann instanziiert werden. Um dies zu realisieren, wurde die oben beschriebe ausführbare Anwendung so umgeschrieben, daß ein Zugriff auf entfernte Objekte ermöglicht wird. Dazu müssen wieder die Remote-Bibliotheken eingebunden werden und ein Channel-Objekt angelegt und registriert werden. Danach kann das Remoteobjekt über die Angabe des Rechners, auf dem das Objekt liegt, der Objekt-URI („Bank") und dem entsprechenden Port (8085) referenziert werden. Zugriffe auf Methoden oder Attribute des Objekts erfolgen genauso, wie Zugriffe auf Methoden oder Attribute eines lokal verfügbaren Objekts.
Um zu zeigen, daß die Wahl der Programmiersprache nicht entscheidend ist, wurde die Server-Klasse zum einen in C# und zum anderen in C++ geschrieben. Beide Varianten haben identische Funktionalität. Zum Schluß müssen alle Bestandteile übersetzt werden, wobei hier für Details auf das Make-File verwiesen wird (Build.bat). Zu beachten ist bei der Übersetzung der C++-Anwendung, daß durch den Compiler verwalteter Code erzeugt wird, was durch den Parameter /clr dem Compiler mitgeteilt wird, da man sonst nicht das .NET Remoting-Framework nutzen kann. Jetzt kann der Server gestartet und mit dem Client auf das Remoteobjekt zugegriffen werden. Um zu zeigen, daß nur eine Instanz der Klasse verfügbar ist, können mehrere Clients gestartet und sich über die Zugriffsfunktion jeweils der Inhalt der Klassenvariable angezeigt werden.

6. .NET im Vergleich mit anderen Technologien

6.1. Einführung

Zum Abschluß der Arbeit soll .NET kurz mit anderen Technologien verglichen werden. Dabei soll ein Vergleich mit einem anderen ganzheitlichen Ansatz, so wie .NET, gemacht werden und ein Vergleich des .NET-Remoting-Frameworks mit einer anderen Middleware-Technologie. Neben Microsoft hat Sun Microsystems mit Sun ONE (Open Net Environment) ein System am Markt, dessen Ziele sich zum großen Teil mit denen des .NET Framework decken. Nähere Informationen dazu sind im Internet erhältlich (http://www.sun.com/software/sunone). IBM hat mit IBM WebSphere die erste Software-Development-Plattform, die Web Services komplett implementiert, auf den Markt gebracht. Auch hier sind nähere Informationen im Internet verfügbar (http://www-3.ibm.com/software/ad/adstudio/). Bei beiden Ansätzen ist die Java 2 Enterprise Edition der Hauptbestandteil und soll aus diesem Grund mit .NET im nachfolgenden Kapitel verglichen werden.

6.2. Vergleich mit der Java 2 Enterprise Edition

Als erstes soll .NET mit der Java 2 Enterprise Edition (J2EE) verglichen werden. Auch J2EE ist ein ganzheitlicher Ansatz, mit dem u.a. lokale Anwendungen, Web Services, Remote-Anwendungen und dynamische Webseiten erstellt werden können. Bei der J2EE ist man auf die Programmiersprache Java angewiesen, wobei man bei .NET zwischen verschiedenen Sprachen wählen und sich die heraussuchen kann, die einem am besten liegt. Der Vorteil für die J2EE hingegen ist, daß sie für unterschiedliche Plattformen und Betriebssysteme verfügbar ist. .NET ist zwar auch plattformunabhängig, läuft zur Zeit aber nur unter Microsoft-Betriebssystemen. Ein weiterer wichtiger Unterschied liegt darin, daß es sich bei .NET um ein Produkt handelt, von dem einige Implementierungen standardisiert wurden (z.B. die CLR), und bei J2EE um einen ganzheitlichen Standard. Die nachfolgende Tabelle gibt einen kurzen Überblick über die Merkmale der beiden Technologien.

Merkmal	J2EE	.NET
Technologietyp	Standard	Produkt
Middleware-Anbieter	über 30	Microsoft
Interpreter	JRE	CLR
Dynamische Webseiten	JSP	ASP.NET
Middle-Tier Komponenten	EJB	.NET Mangaged Components
Datenbankzugriff	JDBC, SQL/J	ADO.NET
SOAP, WSDL, UDDI	ja	ja
Uneingeschränkte Middleware	ja	ja

Tabelle 3 : Gemeinsame Merkmale von .NET und J2EE (vgl. [Vawter u.a. 2001])

Zusätzlich zu den oben genannten gemeinsamen Merkmalen haben beide Ansätze unterschiedliche Merkmale auf die hier nicht näher eingegangen werden soll. Einen ausführlichen Vergleich von .NET und J2EE findet man unter [Vawter u.a. 2001].

6.3. Middleware-Ansätze

Der Vergleich von .NET mit anderen Middleware-Technologien soll hier auf eine Beispiel-Implementierung einer Remote-Anwendung in .NET und CORBA (Common Object Request Broker Architecture) beschränkt werden. Neben CORBA gibt es allerdings auch weitere Middleware-Technologien, wie z.B. Java RMI (Remote Methode Invocation). Alle Ansätze haben dabei ihre Vor- und Nachteile. Bei Java RMI ist man z.B. auf die Programmiersprache

Java angewiesen und das .NET-Remoting ist zur Zeit noch auf Windows-Plattformen beschränkt, bietet aber mehrere Programmiersprachen für die Implementierung. CORBA ist hingegen vollständig plattform- und betriebssystemunabhängig, dafür aber umständlich zu implementieren und dadurch fehleranfällig.
Um die Unterschiede bei der Implementierung zwischen CORBA und .NET aufzuzeigen, wurde eine lokale Anwendung in Java entwickelt, welche die gleiche Funktionalität wie die entwickelte C#-Anwendung des vorangegangenen Kapitels aufweist. Diese Anwendung wurde danach mittels CORBA in eine Remote-Anwendung umgewandelt. Als Programmiersprache wurde Java in der Version 1.4 und als CORBA-Implementierung die Java-IDL gewählt. Die Quelltexte und das Make-File zur CORBA-Anwendung sind Anhang A zu entnehmen.
Bei der Entwicklung einer CORBA-Anwendung muß, nachdem die Struktur und der Aufbau der Remote-Klasse bekannt sind, die Schnittstelle geschaffen werden. Dazu dient die programmiersprachenunabhängige Interface Definition Language (IDL), in der die angebotenen Dienste des CORBA-Objekts beschrieben werden, ihre Implementierung aber verborgen bleibt. Danach werden mit Hilfe des IDL-Compilers die CORBA-spezifischen Klassen erzeugt, zu denen der Client-Stub und der Server-Skeleton zählen. Nachdem diese Klassen erzeugt wurden, können die Implementierungen des Servers, des Clients und des CORBA-Objekts abgeschlossen werden. Um aus der lokalen Klasse (im Bsp. Bank.java) eine Remote-Klasse (im Bsp. BankImpl.java) zu machen, müssen die eben erzeugten Klassen importiert und die Klasse von der Implementation Base (_BankImplBase[1]) abgeleitet werden. Um das Objekt zu hosten ist, wie u.a. beim .NET-Remoting, eine Server-Anwendung (im Bsp. BankServer.java) notwendig. Auf die Erläuterung des Servers soll an dieser Stelle verzichtet werden. Es wird auf entsprechende Literatur zu CORBA verwiesen, eine Kurzüberblick bekommt man unter [Gruhn u.a. 2000]. Zum Schluß wird ein Client (im Bsp. BankClient.java) entwikkelt, der eine Referenz auf das CORBA-Objekt hat und die Dienste dieses Objekts nutzt. Beim Client müssen wieder die durch die IDL erzeugten Klassen und weitere CORBA-spezifische Klassen, z.B. für der Namensdienst (CosNaming), importiert werden. Danach wird der ORB erstellt und initialisiert und das entsprechende Objekt (im Bsp. Bank) referenziert. Jetzt können die Methoden wie bei einem lokalen Objekt aufgerufen werden, wobei der Stub die Parameter des Methodenaufrufs einpackt (Marshalling) und der Skeleton diese wieder entpackt (Unmarshalling) und den Methodenaufruf durchführt. Die Rückgabewerte der Methode nehmen dann den umgekehrten Weg. Die Vorgehensweise bei der Erzeugung einer CORBA-Anwendung verdeutlicht die nachfolgende Abbildung.

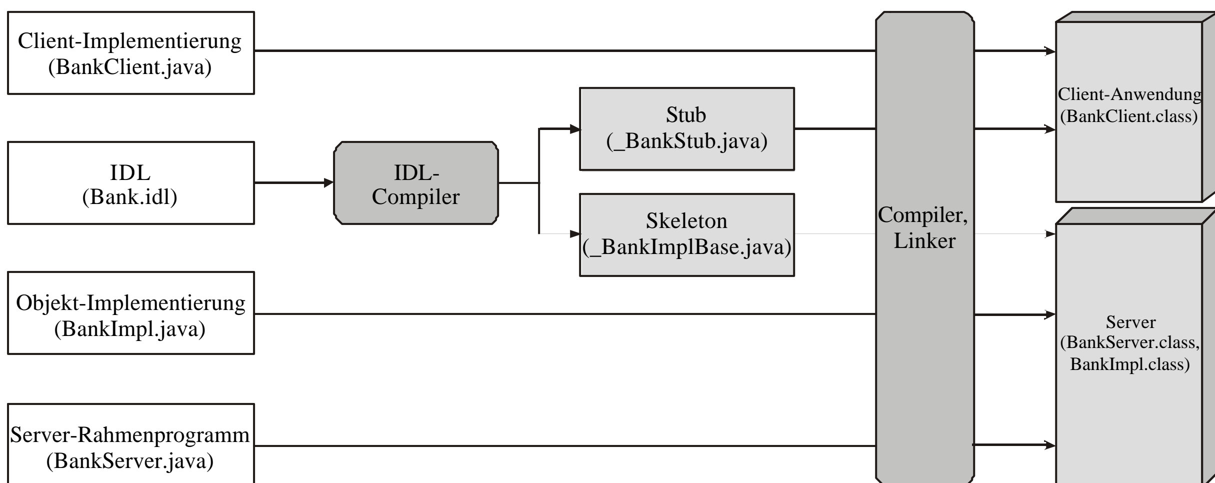

Abbildung 10: Entwicklung der CORBA-Anwendung, vgl. [Gruhn u.a. 2000]

[1] Anm.: Die Implementation Base wurde durch die Compilierung der IDL-Schnittstelle erzeugt.

7. Fazit

Mit .NET hat Microsoft eine komplett neue Technologie in der Microsoft-Welt entwickelt. Dabei handelt es sich aber nicht um eine einzelne isolierte Technologie, sondern um eine Palette unterschiedlicher Technologien, deren Fundament primär auf Web-Techniken und XML beruht (vgl. [Stal 2000]). Microsoft bietet damit ein umfassendes Paket, angefangen vom Betriebssystem, über Office-Anwendungen und Back-Office-Lösungen, bis hin zum Werkzeug für die Entwicklung von lokalen und webbasierten Anwendungen. Zusätzlich bieten Microsoft entgeltliche Dienste im Rahmen von Microsoft Passport.

Als Entwickler von Anwendungen für Microsoft-Betriebssysteme wird man nicht mehr an .NET vorbeikommen. Im Bereich von Internet-Anwendungen wird sich ein Zweikampf von J2EE und .NET entwickeln. Näheres dazu findet man unter [Schmidt 2002]. Ist man auf die Kommunikation von Anwendungen über das Netz angewiesen, ist entscheidend, auf welchen Plattformen diese Anwendungen laufen sollen. Wenn dabei nur Betriebssysteme von Microsoft eingesetzt werden, fällt die Wahl eindeutig auf .NET. Kommen dabei auch noch andere Plattformen in Betracht, muß man den Entwicklungsaufwand für eine Lösung mit .NET und den einer anderen Middleware gegeneinander abwiegen. Da .NET auch mit Komponenten kommunizieren kann, die mit COM, COM+ oder DCOM entwickelt wurden, müssen bestehende Anwendungen nicht neu geschrieben, sondern können angepaßt und erweitert werden.

Literaturverzeichnis

[Born 2001]
Born, A.: Dot-Net: Bauplatz für Webservices – Alles wird .Net, Network World 21/01 (http://www.networkworld.de/artikel/index.cfm?pageid=172&nwwpage=158, Abruf: 21.06.2002)

[Breymann u.a. 2002]
Breymann, U., Loviscach, J.: Die neue C-Klasse – C# im Vergleich mit C++ und Java, c't 04/02, S. 98-105

[Brien 2002]
Brien, C.: SOAP – Vergleich mit herkömmlichen VA-Technologien, Hauptseminararbeit, Technische Universität Ilmenau 2002

[Eller 2001]
Eller, F.: C# lernen, 1. Auflage, Addison Wesley Verlag München 2001

[Groß 2002]
Groß, R.: Datenaustausch in verteilten Umgebungen – Daily SOAP, Windows 2000 Magazin 03/02 (http://www.d-s-t-g.com/news.asp?id=048, Abruf: 21.06.2002)

[Gruhn u.a. 2000]
Gruhn, V., Thiel, A.: Komponentenmodelle, 1. Auflage, Addison Wesley Verlag München 2000

[Klöpper 2001]
Klöpper, M.: Aktuelle Initiativen großer Hersteller: Microsoft .NET, Seminararbeit, Universität Münster 2001

[Loviscach u.a. 2002]
Loviscach, J., Schulz, H., Violka, K.: Sunspiration - .NET und SunONE im Plattformvergleich, c't 04/02, S. 92-97

[Obermeyer u.a. 2000]
Obermeyer, P., Hawkins, J.: Microsoft .NET Remoting: A Technical Overview (http://msdn.microsoft.com/library/en-us/dndotnet/html/hawkremoting.asp, Abruf: 21.06.2002)

[Obermeyer u.a. 2001]
Obermeyer, P., Hawkins, J.: Format for .NET Remoting Configuration Files, MSDN 2001/2002 (http://msdn.microsoft.com/library/en-us/dndotnet/html/remotingconfig.asp, Abruf: 21.06.2002)

[Schlede 2002]
Schlede, F.-M.: Interview zu .NET und Microsoft Server – Basis der Visionen, Windows 2000 Magazin 03/02

[Schmidt 2002]
Schmidt, H.: Web-Services: Sun und Microsoft konkurrieren um den neuen Megatrend im Internet, Frankfurter Allgemeine Zeitung 38/2002, S. 23

[Siering 2002]
Siering, P.: Das Microsoft-Internet - .NET und was dranhängt, c't 04/02, S. 86-91

[Srinivasan 2000]
Srinivasan, P.: Einführung in das Microsoft .NET-Remotingframework, MSDN
(http://www.microsoft.com/germany/ms/msdnbiblio/artikel/remoting.htm, Abruf: 21.06.2002)
[Stal 2000]
Stal, M.: „Microsoft .NET" – Evolution oder Revolution?, Objektspektrum 06/2000, S. 14-18

[Vawter u.a. 2001]
Vawter, C., Roman, E.: J2EE vs. Microsoft.NET – A comparison of building XML-based web services, Sun 2001
(http://www.theserverside.com/resources/articles/J2EE-vs-DOTNET/article.html,
Abruf: 21.06.2002)

[Willers 2000]
Willers, M.: Microsoft.NET – Alles wird gut!?, MSDN

[Willers 2001]
Willers, M.: Die „.NET Common Language Runtime": Überblick und technischer Einstieg, Objektspektrum .../2001, S. 91-96

Anhang A
Quelltexte und Make-Files

1. Lokale Anwendung in C#

Bank.cs:

```
using System;

public class Bank
{
	double kontostand;

	public Bank()
	{
		kontostand = 0.0;
	}

	public bool einzahlen (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				Console.WriteLine ("Betrag negativ!");
			else

				Console.WriteLine ("Betrag Null!");
			return false;
		}

		Console.WriteLine ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand += betrag));

		return true;
	}

	public bool abheben (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				Console.WriteLine ("Betrag negativ!");
			else
				Console.WriteLine ("Betrag Null");
			return false;
		}

		if ((kontostand - betrag) < 0)
		{
			Console.WriteLine ("Konto nicht gedeckt! Kontostand: " + kontostand);
			return false;
		}

		Console.WriteLine ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand -= betrag));

		return true;
	}

	public double get_kontostand()
	{
		return kontostand;
	}
}
```

Client.cs:

```
using System;

public class Client
{
	public static void Main (string[] args)
	{
		Bank bank = new Bank();
		char eingabe = 'q';
		Console.WriteLine ("\n(E)inzahlen\n(A)bheben\n(K)ontostand\n(Q)uit");

		do
		{
			Console.Write ("\nGewuenschte Aktion: ");
			eingabe = (Console.ReadLine ())[0];

			switch (eingabe)
			{
				case 'e':
				case 'E':
				{
					Console.Write ("Einzuzahlender Betrag: ");
					bank.einzahlen (Convert.ToDouble(Console.ReadLine()));
					break;
				}
				case 'a':
				case 'A':
				{
					Console.Write ("Abzuhebender Betrag: ");
					bank.abheben (Convert.ToDouble(Console.ReadLine()));
					break;
				}
				case 'k':
				case 'K':
				{
					Console.WriteLine ("Aktueller Kontostand: " +
					bank.get_kontostand());

					break;
				}
			}
		}
		while (eingabe != 'q' && eingabe != 'Q');

	}

}
```

2. Remote-Anwendung mit .NET

Bank.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace BankBeispiel
{
	// Klasse als Remote-Klasse von MarshalByRefObject ableiten
	public class BankServer : MarshalByRefObject
	{
		double kontostand;

		public BankServer()
		{
			kontostand = 0.0;
		}

		public bool einzahlen (double betrag)
		{

			if (betrag <= 0)
			{
				if (betrag < 0)
					Console.WriteLine ("Betrag negativ!");
				else

					Console.WriteLine ("Betrag Null!");
				return false;
			}

			Console.WriteLine ("Kontostand alt: " + kontostand + " Kontostand neu: " +
			(kontostand += betrag));

			return true;
		}

		public bool abheben (double betrag)
		{
			if (betrag <= 0)
			{
				if (betrag < 0)
					Console.WriteLine ("Betrag negativ!");
				else
					Console.WriteLine ("Betrag Null");
				return false;
			}

			if ((kontostand - betrag) < 0)
			{
				Console.WriteLine ("Konto nicht gedeckt! Kontostand: " + kontostand);
				return false;
			}

			Console.WriteLine ("Kontostand alt: " + kontostand + " Kontostand neu: " +
			(kontostand -= betrag));

			return true;
```

```
		}

		public double get_kontostand()
		{
			return kontostand;
		}
	}

}
```

Client.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace BankBeispiel
{
  public class Client
  {
    public static int Main(string [] args)
    {
      if (args.Length == 1)
      {
                    // Channel-Objekt anlegen
                    TcpChannel chan = new TcpChannel();
                    // Channel registrieren
                    ChannelServices.RegisterChannel(chan);
                    // Remoteobjekt referenzieren
                    BankServer bank = (BankSever)Activator.GetObject(typeof(
                    BankBeispiel.BankServer), "tcp://" + args[0] + ":8085/Bank");

                  char eingabe = 'q';
                  Console.WriteLine ("\n(E)inzahlen\n(A)bheben\n(K)ontostand\n(Q)uit");

                  do
                  {
                          Console.Write ("\nGewuenschte Aktion: ");
                          eingabe = (Console.ReadLine ())[0];

                          switch (eingabe)
                          {
                                  case 'e':
                                  case 'E':
                                  {
                                          Console.Write ("Einzuzahlender Betrag: ");
                                          bank.einzahlen (Convert.ToDouble(Console.ReadLine()));
                                          break;
                                  }
                                  case 'a':
                                  case 'A':
                                  {
                                          Console.Write ("Abzuhebender Betrag: ");
                                          bank.abheben (Convert.ToDouble(Console.ReadLine()));
                                          break;
                                  }
                                  case 'k':
                                  case 'K':
                                  {
                                          Console.WriteLine ("Aktueller Kontostand: " +
                                          bank.get_kontostand());
                                          break;
                                  }
                          }
                  }
                  while (eingabe != 'q' && eingabe != 'Q');


      }
      else
```

```
        Console.WriteLine ("Aufruf in der Form: client <HOST>");
      return 0;
    }
  }
}
```

Server.cs:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

namespace BankBeispiel {
  public class Server {

    public static int Main(string [] args) {

      // Channel-Objekt erstellen
      TcpChannel chan = new TcpChannel(8085);
      // Channel registrieren
      ChannelServices.RegisterChannel(chan);
      // Objekt registrieren
      RemotingConfiguraton.RegisterWellKnownServiceType(Type.GetType(
      "BankBeispiel.BankServer,bank"), "Bank", WellKnownObjectMode.Singleton);

      System.Console.WriteLine("<ENTER> zum Beenden...");
      System.Console.ReadLine();
      return 0;
    }
  }
}
```

Server_cpp.cpp:

```
#using <mscorlib.dll>
#using <System.dll>
#using <System.Runtime.Remoting.dll>

using namespace System;
using namespace System::Runtime::Remoting;
using namespace System::Runtime::Remoting::Channels;
using namespace System::Runtime::Remoting::Channels::Tcp;

void main()
{
    TcpChannel* chan = new TcpChannel(8085);
    ChannelServices::RegisterChannel(chan);
    RemotingConfiguration::RegisterWellKnownServiceType(Type::GetType(
    "BankBeispiel.BankServer,bank"), "Bank", WellKnownObjectMode::Singleton);

    Console::WriteLine("<ENTER> zum Beenden...");
    Console::ReadLine();
}
```

Build.bat:

```
@echo off
md bin
csc /target:library /out:bin\bank.dll bank.cs
csc /R:bin\bank.dll /out:bin\client.exe client.cs
csc /out:bin\server.exe server.cs
cl /clr /o bin\server_cpp.exe server_cpp.cpp
del server_cpp.obj
```

Delete.bat:

```
@echo off
del bin\client.exe
del bin\server.exe
del bin\server_cpp.exe
del bin\bank.dll
rd bin
```

3. Lokale Anwendung in Java:

Bank.java:

```
public class Bank
{
	double kontostand;

	Bank (double betrag)
	{
		kontostand = betrag;
	}

	boolean einzahlen (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				System.out.println ("Betrag negativ!");
			else
				System.out.println ("Betrag Null!");
			return false;
		}

		System.out.println ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand += betrag));

		return true;
	}

	boolean abheben (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				System.out.println ("Betrag negativ!");
			else
				System.out.println ("Betrag Null");
			return false;
		}

		if ((kontostand - betrag) < 0)
		{
			System.out.println ("Konto nicht gedeckt! Konotostand: " + kontostand);
			return false;
		}

		System.out.print ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand -= betrag));

		return true;
	}

	double get_kontostand()
	{
		return kontostand;
	}
}
```

Client.java:

```
import java.io.*;

public class Client
{
	public static void main(String[] args)
	{
		Bank bank = null;
		char auswahl = '0';
		double betrag;
		BufferedReader input;

		if (args.length != 1)
		{
			System.out.println ("Kein Argument uebergeben!");
			System.exit (1);
		}

		try
		{
			bank = new Bank (Double.parseDouble(args[0]));
		}
		catch (NumberFormatException e)
		{
			System.err.println ("Argument ist keine Zahl!");
			System.exit (1);
		}

		System.out.println ("\n(E)inzahlen\n(A)bheben\n(K)ontostand\n(Q)uit");
		input = new BufferedReader (new InputStreamReader (System.in));

		do
		{
			try
			{
				System.out.print ("\nGewuenschte Aktion: ");
				auswahl = (input.readLine()).charAt (0);

				switch (auswahl)
				{
					case 'e': ;
					case 'E':
					{
						System.out.print ("Einzuzahlender Betrag: ");
						try
						{
							bank.einzahlen (Double.parseDouble
							(input.readLine()));
						}
						catch (NumberFormatException e)
						{
							System.out.println ("Nur Zahlen erlaubt!");
						}
						break;
					}
					case 'a': ;
					case 'A':
					{
						System.out.print ("Auszuzahlender Betrag: ");
						try
```

```
                                    {
                                        bank.abheben (Double.parseDouble
                                        (input.readLine()));
                                    }
                                    catch (NumberFormatException e)
                                    {
                                        System.out.println ("Nur Zahlen erlaubt!");
                                    }
                                    break;
                                }
                                case 'k': ;
                                case 'K':
                                {
                                    System.out.println ("Kontostand: " +
                                    bank.get_kontostand ());
                                }
                            }
                        }
                        catch (IOException e)
                        {
                            System.err.println ("I/O-Fehler");
                        }
                    }
                    while (auswahl != 'Q' && auswahl != 'q');
            }
    }
```

4. Remote-Anwendung mit CORBA:

BankImpl.java:

```
import BankApp.*;

public class BankImpl extends _BankImplBase
{
	double kontostand;

	BankImpl ()
	{
		kontostand = 0.0;
	}

	public boolean einzahlen (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				System.out.println ("Betrag negativ!");
			else
				System.out.println ("Betrag Null!");
			return false;
		}

		System.out.println ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand += betrag));

		return true;
	}

	public boolean abheben (double betrag)
	{
		if (betrag <= 0)
		{
			if (betrag < 0)
				System.out.println ("Betrag negativ!");
			else
				System.out.println ("Betrag Null");
			return false;
		}

		if ((kontostand - betrag) < 0)
		{
			System.out.println ("Konto nicht gedeckt! Konotostand: " + kontostand);
			return false;
		}

		System.out.print ("Kontostand alt: " + kontostand + " Kontostand neu: " +
		(kontostand -= betrag));

		return true;
	}

	public double get_kontostand()
	{
		return kontostand;
	}
}
```

BankImpl.java:

```
// Stubs
import BankApp.*;
// Namensdienst
import org.omg.CosNaming.*;
// Exceptions des Namensdienst
import org.omg.CosNaming.NamingContextPackage.*;
// CORBA-Klassen
import org.omg.CORBA.*;

public class BankServer
{
	public static void main(String args[])
	{

		try
		{

			// ORB Erstellen und Initialisieren
			ORB orb = ORB.init(args, null);

			// Objekt erstellen und beim ORB registrieren
			BankImpl bankRef = new BankImpl();
			orb.connect(bankRef);

			// Root Namenskontext erstellen
			org.omg.CORBA.Object objRef =
			orb.resolve_initial_references("NameService");

			NamingContext ncRef = NamingContextHelper.narrow(objRef);

			// Objektreferenz benennen
			NameComponent nc = new NameComponent("Bank", "");
			NameComponent path[] = {nc};
			ncRef.rebind(path, bankRef);

			System.out.println ("Bank-Server gestartet");

			// Auf Anfragen vom Client warten
			java.lang.Object sync = new java.lang.Object();
			synchronized (sync)
			{
				sync.wait();
			}

		}
		catch(Exception e)
		{
			System.err.println("ERROR: " + e);
			e.printStackTrace(System.out);
		}
	}
}
```

BankClient.java:

```
// Stubs
import BankApp.*;
// Namensdienst
import org.omg.CosNaming.*;
// CORBA-Klassen
import org.omg.CORBA.*;
import java.io.*;

public class BankClient
{
	public static void main(String[] args)
	{
		Bank bank = null;
		try
		{
			// ORB erstellen und initialisieren
			ORB orb = ORB.init(args, null);

			// Root Namenskontext erstellen
			org.omg.CORBA.Object objRef =
			orb.resolve_initial_references("NameService");
			NamingContext ncRef = NamingContextHelper.narrow(objRef);

			// Objekt über Namensdienst referenzieren
			NameComponent nc = new NameComponent("Bank", "");
			NameComponent path[] = {nc};
			bank = BankHelper.narrow(ncRef.resolve(path));
		}
		catch (Exception e)
		{
			System.err.println ("CORBA-Fehler!");
		}

		char auswahl = '0';
		double betrag;
		BufferedReader input;

		System.out.println ("\n(E)inzahlen\n(A)bheben\n(K)ontostand\n(Q)uit");
		input = new BufferedReader (new InputStreamReader (System.in));

		do
		{
			try
			{
				System.out.print ("\nGewuenschte Aktion: ");
				auswahl = (input.readLine()).charAt (0);

				switch (auswahl)
				{
					case 'e': ;
					case 'E':
					{
						System.out.print ("Einzuzahlender Betrag: ");
						try
						{
							bank.einzahlen (Double.parseDouble(
							input.readLine()));
						}
						catch (NumberFormatException e)
```

```
                        {
                            System.out.println ("Nur Zahlen erlaubt!");
                        }
                        break;
                    }
                    case 'a': ;
                    case 'A':
                    {
                        System.out.print ("Auszuzahlender Betrag: ");
                        try
                        {
                            bank.abheben (Double.parseDouble (
                            input.readLine()));
                        }
                        catch (NumberFormatException e)
                        {
                            System.out.println ("Nur Zahlen erlaubt!");
                        }
                        break;
                    }
                    case 'k': ;
                    case 'K':
                    {
                        System.out.println ("Kontostand: " +
                        bank.get_kontostand ());
                    }
                }
            }
            catch (IOException e)
            {
                System.err.println ("I/O-Fehler");
            }
        }
        while (auswahl != 'Q' && auswahl != 'q');
    }
}
```

Bank.idl:

```
/* IDL-definiertes Interface der Bank-Anwendung */

module BankApp {
  interface Bank {

    boolean einzahlen (in double betrag);
    boolean abheben (in double betrag);
    double get_kontostand();
  };
};
```

Build.bat:

```
@echo off
md bin
idlj -oldImplBase -f all Bank.idl
javac -d bin BankClient.java
javac -d bin BankServer.java
```

Delete.bat:

```
@echo off
del BankApp\*.*/q
del bin\*.*/q
del bin\BankApp\*.*/q
rd BankApp
rd bin\BankApp
rd bin
```