1 C# und das .NET-Framework

C# (sprich: *see sharp*) ist eine von Microsoft entwickelte Programmiersprache für die .NET-Plattform ([HTWG10]). Obwohl man .NET-Programme in ganz verschiedenen Sprachen schreiben kann (unter anderem in C++, Visual Basic, Java, Cobol oder Eiffel), hat Microsoft mit C# eine neue »Haussprache« geschaffen, um damit die Mächtigkeit von .NET voll auszureizen. C# ist eine objektorientierte Sprache, die sich äußerlich stark an Java anlehnt, aber in ihrer Mächtigkeit deutlich darüber hinausgeht. Sie besitzt all jene Eigenschaften, die man benötigt, um Programme nach dem neuesten Stand der Softwaretechnik zu entwickeln.

C# ist keine revolutionäre Sprache. Sie ist vielmehr eine Kombination aus Java, C++ und Visual Basic, wobei man versucht hat, von jeder Sprache die bewährten Eigenschaften zu übernehmen und die komplexen Eigenschaften zu vermeiden. C# wurde von einem kleinen Team unter der Leitung von *Anders Hejlsberg* entworfen. Hejlsberg ist ein erfahrener Sprachdesigner. Er war bei Borland Chefentwickler von Delphi und ist dafür bekannt, seine Sprachen auf die Bedürfnisse von Praktikern zuzuschneiden.

In diesem Kapitel geben wir einen Überblick über die wichtigsten Eigenschaften von C#. Aufgrund der Ähnlichkeiten zu Java stellen wir dabei die Merkmale von C# denen von Java gegenüber, wobei wir davon ausgehen, dass der Leser bereits programmieren kann und eine Sprache wie Java oder C++ beherrscht. Da man als C#-Entwickler nicht umhinkommt, auch die Grundkonzepte von .NET zu kennen, gehen wir am Ende dieses Kapitels auch kurz auf .NET ein.

1.1 Ähnlichkeiten zwischen C# und Java

Auf den ersten Blick sehen C#-Programme wie Java-Programme aus. Jeder Java-Programmierer sollte daher in der Lage sein, C#-Programme zu lesen. Neben der fast identischen Syntax wurden folgende Konzepte aus Java übernommen:

- *Objektorientierung*
  C# ist wie Java eine objektorientierte Sprache mit einfacher Vererbung. Klassen können nur von einer einzigen Klasse erben, aber mehrere Schnittstellen (Interfaces) implementieren.

- *Typsicherheit*
  C# ist eine typsichere Sprache. Viele Programmierfehler, die durch inkompatible Datentypen in Anweisungen und Ausdrücken entstehen, werden bereits vom Compiler abgefangen. Zeigerarithmetik oder ungeprüfte Typumwandlungen wie in C++ sind in Anwendungsprogrammen verboten. Zur Laufzeit wird sichergestellt, dass Array-Indizes im erlaubten Bereich liegen, dass Objekte nicht durch uninitialisierte Zeiger referenziert werden und dass Typumwandlungen zu einem definierten Ergebnis führen.
- *Garbage Collection*
  Dynamisch erzeugte Objekte werden vom Programmierer nie selbst freigegeben, sondern von einem Garbage Collector automatisch eingesammelt, sobald sie nicht mehr referenziert werden. Das beseitigt viele unangenehme Fehler, die z.B. in C++-Programmen auftreten können.
- *Namensräume*
  Was in Java Pakete sind, nennt man in C# Namensräume. Ein Namensraum ist eine Sammlung von Deklarationen und ermöglicht es, gleiche Namen in unterschiedlichem Kontext zu verwenden.
- *Threads*
  C# unterstützt leichtgewichtige parallele Prozesse in Form von Threads. Es gibt wie in Java Mechanismen zur Synchronisation und Kommunikation zwischen Prozessen.
- *Generizität*
  Sowohl Java als auch C# kennen generische Typen und Methoden. Damit kann man Bausteine herstellen, die mit anderen Typen parametrisierbar sind (z.B. Listen mit beliebigem Elementtyp).
- *Reflection*
  Wie in Java kann man auch in C# zur Laufzeit auf Typinformationen eines Programms zugreifen, Klassen dynamisch zu einem Programm hinzuladen, ja sogar Objektprogramme zur Laufzeit zusammenstellen.
- *Attribute*
  Der Programmierer kann beliebige Informationen an Klassen, Methoden oder Felder hängen und sie zur Laufzeit mittels Reflection abfragen. In Java heißt dieser Mechanismus *Annotationen*.
- *Bibliotheken*
  Viele Typen der C#-Bibliothek sind denen der Java-Bibliothek nachempfunden. So gibt es vertraute Typen wie Object, String, ICollection oder Stream, meist sogar mit den gleichen Methoden wie in Java.

Auch aus C++ wurden einige Dinge übernommen, zum Beispiel das Überladen von Operatoren, die Zeigerarithmetik in systemnahen Klassen (die als *unsafe* gekennzeichnet sein müssen) sowie einige syntaktische Details z.B. im Zusammenhang mit Vererbung. Aus Visual Basic stammt beispielsweise die foreach-Schleife.

1.2 Unterschiede zwischen C# und Java

Neben diesen Ähnlichkeiten weist C# aber wie alle .NET-Sprachen auch einige Merkmale auf, die in Java fehlen:

- *Referenzparameter*
  Parameter können nicht nur durch *call by value* übergeben werden, wie das in Java üblich ist, sondern auch durch *call by reference*. Dadurch sind nicht nur Eingangs-, sondern auch Ausgangs- und Übergangsparameter realisierbar.
- *Objekte am Keller*
  Während in Java alle Objekte am Heap liegen, kann man in C# Objekte auch am Methodenaufrufkeller anlegen. Diese Objekte sind leichtgewichtig und belasten den Garbage Collector nicht.
- *Blockmatrizen*
  Für numerische Anwendungen ist das Java-Speichermodell mehrdimensionaler Arrays zu ineffizient. C# lässt dem Programmierer die Wahl, mehrdimensionale Arrays entweder wie in Java anzulegen oder als kompakte Blockmatrizen, wie das in C, Fortran oder Pascal üblich ist.
- *Einheitliches Typsystem*
  Im Gegensatz zu Java sind in C# alle Datentypen (auch `int` oder `char`) vom Typ `object` abgeleitet und erben die dort deklarierten Methoden.
- *goto-Anweisung*
  Die viel geschmähte goto-Anweisung wurde in C# wieder eingeführt, allerdings mit Einschränkungen, so dass man mit ihr kaum Missbrauch treiben kann.
- *Versionierung*
  Bibliotheken werden bei der Übersetzung mit einer Versionsnummer versehen. So kann eine Bibliothek gleichzeitig in verschiedenen Versionen vorhanden sein. Jede Applikation verwendet immer diejenige Version der Bibliothek, mit der sie übersetzt und getestet wurde.

Schließlich hat C# noch eine ganze Reihe von Eigenschaften, die zwar die Mächtigkeit der Sprache nicht erhöhen, aber bequem zu benutzen sind. Sie fallen unter die Kategorie »*syntactic sugar*«, d.h., man kann mit ihnen Dinge tun, die man auch in anderen Sprachen realisieren könnte, nur dass es in C# eben einfacher und eleganter geht. Dazu gehören:

- *Properties* und *Events*
  Diese Eigenschaften dienen der Komponententechnologie. Properties sind spezielle Felder eines Objekts. Greift man auf sie zu, werden automatisch get- und set-Methoden aufgerufen. Mit Events kann man Ereignisse definieren, die von Komponenten ausgelöst und von anderen behandelt werden.

- *Indexer*
  Ein Index-Operator wie bei Array-Zugriffen kann durch get- und set-Methoden selbst definiert werden.
- *Delegates*
  Delegates sind im Wesentlichen das, was man in Pascal *Prozedurvariablen* und in C *Function Pointers* nennt. Sie sind allerdings etwas mächtiger. Zum Beispiel kann man mehrere Prozeduren in einer einzigen Delegate-Variablen speichern.
- *foreach-Schleife*
  Damit kann man bequem über Arrays, Listen oder Mengen iterieren.
- *Iteratoren*
  Man kann spezielle Iterator-Methoden schreiben, die eine Folge von Werten liefern, welche dann mit foreach durchlaufen werden kann.
- *Lambda-Ausdrücke*
  Lambda-Ausdrücke sind parametrisierte Codestücke, die man an Variablen zuweisen und später aufrufen kann. Sie sind eine Kurzform für namenlose Methoden.
- *Query-Ausdrücke*
  Sie erlauben SQL-ähnliche Abfragen auf Hauptspeicherdaten wie Arrays oder Listen.

1.3 Das .NET-Framework

Wer in C# programmiert, kommt früher oder später nicht umhin, sich auch in die Grundlagen des .NET-Frameworks einzuarbeiten, für das C# entwickelt wurde. Das .NET-Framework ist eine Schicht, die auf Windows (und später vielleicht auch einmal auf anderen Betriebssystemen) aufsetzt (siehe Abb. 1–1) und vor allem zwei Dinge hinzufügt:

- Eine **Laufzeitumgebung** (die *Common Language Runtime*), die automatische Speicherbereinigung (*garbage collection*), Sicherheitsmechanismen, Versionierung und vor allem Interoperabilität zwischen verschiedenen Programmiersprachen bietet.
- Eine **objektorientierte Klassenbibliothek** mit umfangreichen Funktionen für grafische Benutzeroberflächen (*Windows Forms*), Web-Oberflächen (*ASP.NET*), Datenbankanschluss (*ADO.NET*), Web-Services, Collection-Klassen, Threads, Reflection und vieles mehr. Sie ersetzt in vielen Fällen das bisherige Windows-API und geht weit über dieses hinaus.

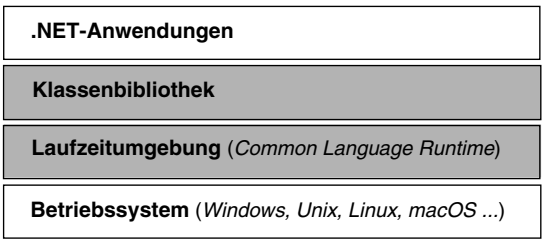

Abb. 1–1 *Grobarchitektur des .NET-Frameworks*

Obwohl .NET von Microsoft entwickelt wurde, basiert es auf offenen Standards. Der ECMA-Standard 335 definiert zum Beispiel die Common Language Runtime und Teile der Klassenbibliothek, der ECMA-Standard 334 beschreibt die Sprache C#, und auch in Web-Services werden allgemeine Standards wie SOAP, WSDL oder UDDI verwendet. Im Rahmen eines Open-Source-Projekts ([Mono]) wurde das .NET-Framework auf Linux portiert, und Microsoft selbst stellt sogar große Teile des Quellcodes der CLR unter dem Namen SSCLI (*Shared Source Common Language Infrastructure*) zur Verfügung ([SNS03]).

Dieser Abschnitt gibt einen Überblick über die wichtigsten Teile des .NET-Frameworks. Eine ausführlichere Beschreibung findet man zum Beispiel in [BBMW03], [NEGWS12]. oder in [SDKDoc]. Teile der Klassenbibliothek werden in Kapitel 19 beschrieben.

Common Language Runtime

Die *Common Language Runtime* (CLR) ist die Laufzeitumgebung, unter der .NET-Programme ausgeführt werden und die unter anderem Garbage Collection, Sicherheit und Interoperabilität unterstützt.

Ähnlich wie die Java-Umgebung basiert die CLR auf einer *virtuellen Maschine* mit einem eigenen Befehlssatz (CIL – *Common Intermediate Language*), in den die Programme aller .NET-Sprachen übersetzt werden. Unmittelbar vor der Ausführung (*just in time*) werden CIL-Programme dann in den Code der Zielmaschine (z.B. in Intel-Code) umgewandelt (siehe Abb. 1–2). Der CIL-Code garantiert die Interoperabilität zwischen den verschiedenen Sprachen und die Portabilität des Codes, die JIT-Compilation (*just in time compilation*) stellt sicher, dass die Programme trotzdem effizient ausgeführt werden.

Damit verschiedene Sprachen zusammenarbeiten können, genügt es aber nicht, sie in CIL-Code zu übersetzen. Es muss auch gewährleistet sein, dass sie die gleiche Art von Datentypen benutzen. Die CLR definiert daher auch ein gemeinsames Typsystem – das *Common Type System* (CTS), das festlegt, wie Klassen, Interfaces und andere Typen auszusehen haben. Das CTS erlaubt nicht nur, dass eine Klasse, die zum Beispiel in C# implementiert wurde, von einem Visual-Basic-Programm benutzt werden kann; es ist sogar möglich, diese C#-Klasse in Visual Basic

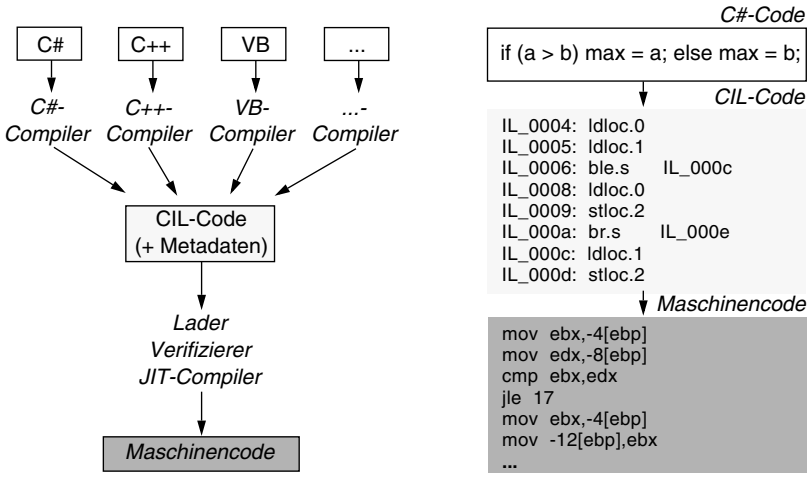

Abb. 1–2 *Quellcode, CIL-Code und Maschinencode*

durch eine Unterklasse zu erweitern oder eine Ausnahme (*exception*), die in C# ausgelöst wurde, von einem Programm in einer anderen Sprache behandeln zu lassen.

Die CLR stellt Mechanismen zur Verfügung, die .NET-Programme sicherer und robuster machen. Dazu gehört zum Beispiel der *Garbage Collector*, der dafür zuständig ist, den Speicherplatz von Objekten freizugeben, sobald diese nicht mehr benutzt werden. In älteren Sprachen wie C oder C++ ist der Programmierer für die Freigabe von Objekten selbst verantwortlich. Dabei kann es vorkommen, dass er ein Objekt freigibt, das noch von anderen Objekten benutzt wird. Diese Objekte greifen dann »ins Leere« und zerstören fremde Speicherbereiche. Umgekehrt kann es vorkommen, dass ein Programmierer vergisst, Objekte freizugeben, obwohl sie nicht mehr referenziert werden. Diese bleiben dann als Speicherleichen (*memory leaks*) zurück und verschwenden Platz. Solche Fehler sind schwer zu finden, können aber dank Garbage Collector unter .NET nicht vorkommen.

Wenn ein CIL-Programm geladen und in Maschinencode übersetzt wird, prüft die CLR mittels eines *Verifizierers*, dass die Typregeln des CTS nicht verletzt werden. Es ist zum Beispiel verboten, eine Zahl als Adresse zu interpretieren und damit auf fremde Speicherbereiche zuzugreifen.

Assemblies

.NET unterstützt komponentenorientierte Softwareentwicklung. Die Komponenten heißen *Assemblies* und sind die kleinsten Programmbausteine, die separat ausgeliefert werden können. Ein Assembly ist eine Sammlung von Klassen und anderen Ressourcen (z.B. Bildern) und wird entweder als ausführbare EXE-Datei oder als Bibliotheksbaustein in Form einer DLL-Datei (*dynamic link library*) gespeichert (siehe Abb. 1–3). In manchen Fällen kann ein Assembly auch aus mehreren Dateien bestehen.

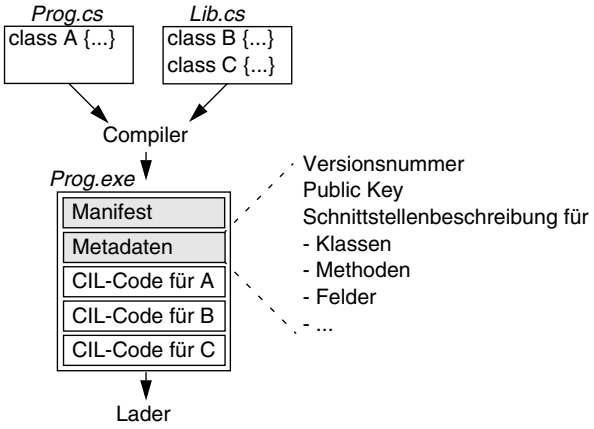

Abb. 1–3 *Vom Compiler erzeugtes Assembly Prog.exe*

Jedes Assembly enthält neben Code auch *Metadaten*, also die Schnittstellenbeschreibung seiner Klassen, Felder, Methoden und sonstigen Programmelemente. Zusätzlich enthält es ein *Manifest*, das man sich als Inhaltsverzeichnis vorstellen kann. Assemblies sind also selbstbeschreibend und können mittels *Reflection* vom Lader, Compiler und anderen Werkzeugen analysiert und benutzt werden.

Assemblies dienen auch der Versionierung, d.h., sie haben eine mehrstufige Versionsnummer, die für alle in ihnen enthaltenen Klassen gilt. Wenn eine Klasse übersetzt wird, werden in ihrem Objektcode die Versionsnummern der aus anderen Assemblies benutzten Klassen vermerkt. Der Lader lädt dann jene Assemblies, die der erwarteten Versionsnummer entsprechen. Unter .NET können also mehrere gleichnamige DLLs mit unterschiedlichen Versionsnummern nebeneinander existieren (*side by side execution*), ohne sich in die Quere zu kommen. Das bedeutet das Ende der »*DLL Hell*« unter Windows, bei der durch die Installation neuer Software alte DLLs durch gleichnamige neue überschrieben werden konnten und dadurch existierende Software plötzlich nicht mehr funktionierte.

Assemblies müssen auch nicht mehr in die Windows-Registry eingetragen werden. Man kopiert sie einfach ins Applikationsverzeichnis oder in den so genannten *Global Assembly Cache* und kann sie ebenso einfach wieder entfernen.

Assemblies sind gewissermaßen die Nachfolger von COM-Komponenten. Anders als unter COM (*Component Object Model*) braucht man Assemblies aber nicht mehr durch eine IDL (*Interface Definition Language*) zu beschreiben, da sie ja die vollständigen Metadaten enthalten, die der Compiler aus ihrem Quellcode gewonnen hat. Das Common Type System stellt sicher, dass Software, die in unterschiedlichen Sprachen geschrieben wurde, die gleiche Art von Metadaten benutzt und somit binärkompatibel ist. Investitionen in die COM-Technologie sind aber nicht verloren. Es ist möglich, COM-Komponenten von .NET-Klassen aus zu verwenden und umgekehrt (siehe Kapitel 24).

ADO.NET

ADO.NET umfasst alle Klassen der .NET-Bibliothek, die für den Zugriff auf Datenbanken und andere Datenquellen (z.B. XML-Dateien) zuständig sind. Es gab bereits eine Vorgängertechnologie namens ADO (*ActiveX Data Objects*), die jedoch mit ADO.NET nur den Namen gemeinsam hat. ADO.NET ist objektorientiert und somit strukturierter und einfacher zu benutzen.

ADO.NET unterstützt das relationale Datenmodell mit Transaktionen und Sperrmechanismen. Dabei ist es unabhängig von verschiedenen Anbietern und Datenbankarchitekturen. Implementierungen konkreter Datenbankanbindungen an MS SQL Server, OLE DB (*Object Linking and Embedding Database*) und ODBC (*Open Database Connectivity*) werden durch gemeinsame Interfaces abstrahiert.

Der Zugriff auf Datenquellen kann verbindungsorientiert oder verbindungslos erfolgen. Im ersten Fall wird eine ständige Verbindung zur Datenquelle aufrechterhalten, im zweiten Fall wird ein Schnappschuss eines Teils der Datenbank in ein DataSet-Objekt geholt und dann lokal weiterverarbeitet. In beiden Fällen greift man auf die Daten in der Regel mittels SQL (*Structured Query Language*) zu.

ASP.NET

ASP.NET ist jener Teil der .NET-Technologie, der die Programmierung dynamischer Webseiten abdeckt. Mit der Vorgängertechnologie ASP (*Active Server Pages*) hat auch ASP.NET nur den Namen gemeinsam. Das Programmiermodell hat sich grundlegend geändert.

Mit ASP.NET werden Webseiten am Server dynamisch aus aktuellen Daten zusammengestellt und in Form von reinem HTML an Klienten geschickt, wo sie von jedem Web-Browser angezeigt werden können. Im Gegensatz zu ASP wird in ASP.NET ein objektorientiertes Programmiermodell verwendet. Sowohl die Webseite als auch die in ihr vorkommenden GUI-Elemente sind Objekte, die man über einen Namen ansprechen und auf deren Felder und Methoden man in Programmen zugreifen kann. All das geschieht in einer compilierten Sprache wie C# oder Visual Basic .NET und nicht wie in ASP in einer interpretierten Sprache wie JavaScript oder VBScript. Daher hat man auch Zugriff auf die gesamte Klassenbibliothek von .NET.

Die Verarbeitung von Benutzereingaben folgt einem ereignisgesteuerten Modell. Wenn der Benutzer ein Textfeld ausfüllt, einen Button anklickt oder einen Eintrag aus einer Liste wählt, wird ein Ereignis ausgelöst, das dann durch serverseitigen Code behandelt werden kann. Obwohl der Server – wie am Internet üblich – zustandslos ist, wird der Zustand einer Webseite zwischen den einzelnen Benutzeraktionen aufbewahrt, und zwar in der Seite selbst. Das stellt eine wesentliche Erleichterung gegenüber älteren Programmiermodellen dar, bei denen der Programmierer für die Zustandsverwaltung selbst verantwortlich war.

ASP.NET bietet eine reichhaltige Bibliothek von GUI-Elementen, die weit über das hinausgeht, was unter HTML verfügbar ist, obwohl alle GUI-Elemente letzt-

endlich auf HTML abgebildet werden. Der Programmierer hat sogar die Möglichkeit, eigene GUI-Elemente zu implementieren und somit die Benutzeroberfläche von Webseiten seinen speziellen Bedürfnissen anzupassen. Besonders einfach ist die Darstellung von Datenbankabfrageergebnissen in Form von Listen und Tabellen, was von ASP.NET weitgehend automatisiert wird. Eine weitere Neuheit von ASP.NET sind Validatoren, mit denen Benutzereingaben auf ihre Gültigkeit überprüft werden können.

Mit der Entwicklungsumgebung Visual Studio .NET kann man Webseiten interaktiv erstellen, wie man das bei Benutzeroberflächen von Desktop-Anwendungen gewohnt ist. GUI-Elemente können mit der Maus in einem Fenster positioniert werden. Über Menüs und Property-Fenster kann man Attribute setzen und Methoden spezifizieren, die als Reaktion auf Benutzereingaben aufgerufen werden sollen. All das verwischt die Unterschiede zwischen der Programmierung lokaler Desktop-Anwendungen und Internet-Anwendungen und erleichtert zum Beispiel das Erstellen von Web-Shops und tagesaktuellen Informationsseiten (z.B. Börseninformationen). ASP.NET wird in Abschnitt 27.3 näher erklärt.

Web-Services

Web-Services werden von Microsoft als einer der Kernpunkte der .NET-Technologie bezeichnet, obwohl es sie auch außerhalb von .NET gibt. Es handelt sich um Prozedurfernaufrufe (*remote procedure calls*), die als Protokolle meist HTTP und SOAP (eine Anwendung von XML) benutzen.

Das Internet hat sich als äußerst leistungsfähig und geeignet erwiesen, um auf weltweit verstreute Informationen und Dienste zuzugreifen. Bisher erfolgte dieser Zugriff jedoch meist über Web-Browser. Web-Services sollen nun eine neue Art des Zusammenspiels zwischen verteilten Applikationen ermöglichen, bei denen die Kommunikation ohne Web-Browser abläuft. Normale Desktop-Anwendungen können sich Informationen wie aktuelle Wechselkurse oder Buchungsdaten über ein oder mehrere Web-Services holen, die als Prozeduren auf anderen Rechnern laufen und über das Internet angesprochen werden.

Die Aufrufe und Parameter werden dabei in der Regel mittels SOAP [SOAP] codiert, eines auf XML basierenden Standards, der von den meisten großen Firmen unterstützt wird. Der Programmierer merkt jedoch von all dem nichts. Er ruft einen Web-Service wie eine normale Methode auf, und .NET sorgt dafür, dass der Aufruf nach SOAP umgewandelt, über das Internet verschickt und auf dem Zielrechner wieder decodiert wird. Am Zielrechner wird die gewünschte Methode aufgerufen, die ihre Ergebnisse wieder transparent über SOAP an den Rufer zurückschickt. Der Rufer und die gerufene Methode können dabei in ganz verschiedenen Sprachen geschrieben sein und auf unterschiedlichen Betriebssystemen laufen.

Damit .NET die SOAP-Codierung und Decodierung korrekt durchführen kann, werden Web-Services samt ihren Parametern mittels WSDL (*Web Services Description Language* [WSDL]) beschrieben. Auch das erledigt .NET automatisch. Web-Services werden in Abschnitt 27.2 dieses Buchs näher erklärt.

1.4 Übungsaufgaben[1]

1. **Eignung von C# für große Softwareprojekte**
   Inwiefern helfen die Eigenschaften von C# bei der Entwicklung großer Softwareprojekte?
2. **Merkmale von .NET**
   Was sind die Hauptmerkmale des .NET-Frameworks? Welche dieser Merkmale ähneln der Java-Umgebung und welche sind neu?
3. **Sicherheit**
   Begründen Sie, warum C# eine sichere Sprache ist. Welche Arten von Programmierfehlern oder gefährlichen Situationen werden vom C#-Compiler oder der CLR abgefangen?
4. **Interoperabilität**
   Warum können unter .NET Programme, die in unterschiedlichen Sprachen geschrieben wurden, nahtlos zusammenarbeiten?
5. **Assemblies**
   Warum sind .NET-Assemblies einfacher zu installieren und zu deinstallieren als COM-Objekte?
6. **Web-Recherche**
   Besuchen Sie die Webseiten [MS], [MSDN] und [CodeGal], um sich einen Überblick über das .NET-Framework und C# zu verschaffen.
7. **Mono**
   Besuchen Sie die Webseite [Mono], um mehr über die Portierung von .NET auf Linux zu erfahren.

1. Musterlösungen zu den Übungsaufgaben in diesem Buch findet man unter [JKU].