Kapitel 1

.NET Framework – Grundlagen und Einführung

C# (sprich *see scharp*), Visual Basic .NET und das .NET Framework sind die Themen, die derzeit im Programmierbereich hoch gehandelt werden. Neugier und auch Unsicherheiten machen sich breit, da vielen Entwicklern die Einordnung der zukünftigen Programmentwicklung mit dem .NET Framework und den darauf basierenden Programmiersprachen sehr schwer fällt. Obgleich das Thema Internet und die Bereitstellung von Web-Diensten als Hauptthemen forciert werden, können Sie mit den Programmiersprachen des .NET Framework auch weiterhin Windows-Anwendungen und eigenständige Komponenten und Steuerelemente entwickeln. Das .NET Framework ist dabei einfach betrachtet eine riesige Fundgrube an Steuerelementen und Objekten, die auch den Zugriff auf Funktionen erlauben, die früher ausschließlich unter Verwendung von API-Aufrufen nutzbar waren. Was sich hinter dem .NET Framework verbirgt und wie Sie das Framework in Programmierprojekten einsetzen, soll im Folgenden genauer beleuchtet werden. Dabei wird auch versucht zu erklären, welche Vorteile die neue Programmierumgebung hat und welche Probleme damit gelöst werden sollen.

Der ein oder andere von Ihnen wird vielleicht schon auf mehrjährige Erfahrung im Bereich der Softwareentwicklung zurückblicken können. Sei es als Programmierer oder Teamleiter, Trainer oder Produkt-/Projektmanager, Sie wissen um die Probleme, Ängste und Sorgen(!), wenn ein neues Produkt – in unserem Fall ein Programm bzw. eine Anwendung – geplant, entwickelt (und getestet), verkauft und gepflegt werden muss.

Auch wenn Sie noch über keine große Erfahrung auf diesem Gebiet verfügen oder sogar erst heute den Entschluss gefasst haben, im Bereich der Software tätig werden zu wollen, sind Ihnen genau die gleichen Schwierigkeiten bekannt – nur aus einem anderen Blickwinkel.

1.1 Probleme der Softwareentwicklung

Natürlich können an dieser Stelle nicht alle Probleme der aktuellen Softwareentwicklung aufgelistet werden. Sie werden jedoch die wichtigsten vorfinden und wieder erkennen. Eingeschlossen sind Probleme, die nicht nur während der Implementierungsphase eines Softwareprojekts, sondern auch sehr viel früher in der Planungsphase oder später während der Testphase oder in Wartung und Pflege auftauchen. In jeder Projektphase werden nicht nur Zeit, Ressourcen und Kosten verschlungen, sondern auch einige Nerven aller Beteiligten.

Sei es die Wahl der Zielplattform (Betriebssystem und Programmiersprache), die Art und Form der Auslieferung der Software an den Kunden oder (den Entwicklern und Administratoren unter Ihnen sehr bekannt) die DLL-Hölle und Versions-Konflikte, all diese Themen werden hier kurz beleuchtet.

1.1.1 Die Wahl der Zielplattform

Ein erfahrener Programmierer oder gar ein ganzes Team von Programmierern, Teamleitern und Managern quält sich zu einem bestimmten Zeitpunkt eines laufenden Projekts mit der Frage, welche Programmiersprache auf welchem Betriebssystem verwendet werden soll. Es werden die Vorzüge und Nachteile, Leistungsmerkmale und Schwachpunkte diskutiert, und jeder ist der Meinung, er kenne alle Einzelheiten der Sprachen und der Betriebssysteme ganz genau.

Weniger erfahrene Entscheidungsträger (sei es im beruflichen oder privaten Leben) müssen die Eigenschaften, die Gegenstand der eben erwähnten Diskussionen sind, zunächst einmal kennen lernen, bevor sie anschließend herausfinden können, welche Plattform am besten geeignet ist, das geplante Projekt zu implementieren.

Ob es nun sehr große Programme für Firmen oder kleine Tools für den Heimanwender sind; die Fragen nach dem Betriebssystem beinhalten Namen und Produkte wie Microsoft Windows 98, Windows Millennium Edition, Windows NT oder Windows 2000, Apple Macintosh oder UNIX (in irgendeiner Ausprägung, z.B. Linux). Selbst wenn Ihnen das Betriebssystem ganz egal sein sollte, da Sie eine Anwendung für einen Web Browser planen, müssen Sie Einschränkungen machen und/oder Sie werden durch zahlreiche, von Ihnen nicht beeinflussbare Vorgaben eingeschränkt. Denn z.B. nicht jeder Browser unterstützt alle Versionen von HTML, und manche Funktionen führen in unterschiedlichen Browsern zu ganz verschiedenen Ergebnissen.

Die Wahl des Betriebssystems, auf dem Ihr neues (Software-)Produkt laufen soll, ist nicht ganz unabhängig von der Programmiersprache, die bei der Entwicklung zum Einsatz kommen soll. So können Sie zwar (nach mehr oder weniger langen und mehr oder weniger zahlreichen Projekt-Meetings) entscheiden, welche Sprachen für welches Betriebssystem vorhanden und geeignet sind. Aber dann müssen Sie wählen, was davon den Anforderungen genügt und ob die Programmierer über die nötige Erfahrung verfügen.[1]

Plattformunabhängigkeit ist gefordert, so dass die Frage nach dem Betriebssystem erst gar nicht mehr auftaucht und die Programmiersprache frei gewählt werden kann. Letzteres vielleicht sogar nach den Vorlieben der Programmierer selbst. Noch besser wäre es, wenn ein Programm aus verschiedenen Modulen bestehen könnte, die jedes für sich in einer anderen Sprache geschrieben worden sind. Dadurch stünde eine Art Baukasten zur Verfügung, deren Bausteine über Team-, Abteilungs- und sogar Firmen- und Produktgrenzen hinweg getauscht und/oder verkauft werden könnten, ohne Einschränkungen aufgrund der verwendeten Programmiersprache oder des Betriebssystems.

---

1 ... und, ob ihr erfahrener Linux-Programmierer geneigt ist oder überzeugt werden kann, seine C/C++ Kenntnisse auf Windows zu übertragen – und umgekehrt.

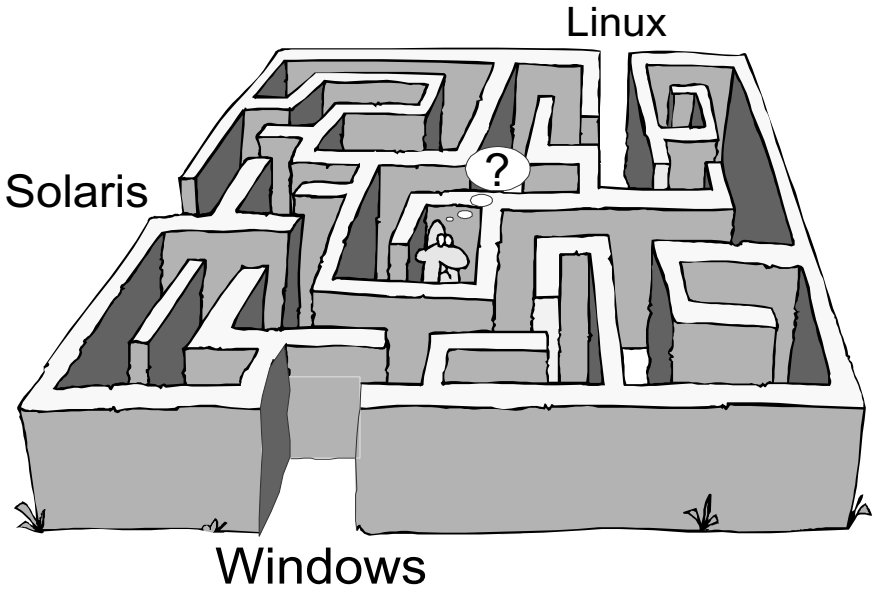

Abbildung 1.1: Die Wahl der Zielplattform wird nicht nur von technischen, sondern auch von politischen Faktoren beeinflusst.

1.1.2 Die Form der Auslieferung

Ein weiteres Problem, das ebenfalls in den Entscheidungsprozess über die Zielplattform mit einbezogen werden sollte, ist die Frage nach der Art der Auslieferung des Produkts an die Kunden. Kaum jemand kommt heutzutage daran vorbei, seine Software auf CD-ROM (evtl. Disketten oder DVD) zu vertreiben. Doch physikalische Medien sind teuer: der Träger selbst, die Verpackung, das Versenden, etc.; dies alles erhöht die Kosten sowohl für den Hersteller als auch für den Käufer.

Die Auslieferung auf rein elektronischem Weg bietet viele Vorteile, zumindest was die Kosten betrifft. Aber dieser Art des Vertriebs von Softwareprodukten über das Internet wird kein großes Vertrauen entgegengebracht – zu Recht. Als Anbieter unterliegen Sie der Gefahr, dass der Vertrieb sich verselbständigt und Ihre Produkte plötzlich der ganzen Welt zur Verfügung stehen, Sie jedoch keine müde Mark damit einfahren können. Als Kunde müssen Sie großes Vertrauen in das Stück Software haben, das Sie sich da gerade aus dem Internet heruntergeladen haben. Denn ein Schutz gegen böse Machenschaften wie Trojanische Pferde, Viren oder gefälschte Signaturen ist nahezu unmöglich. Außerdem ist ein Softwarekauf übers Internet mit sehr vielen »Unannehmlichkeiten« verbunden. Denn ein Anbieter möchte selbstverständlich auch auf der eher sicheren Seite des Lebens stehen und verlangt von einem Kunden korrekte Daten über die Person und seine Bank-

verbindung. Und wie immer wieder aus unterschiedlichen Pressemedien zu erfahren ist, birgt das Versenden von sensiblen Daten übers Internet zahlreiche Gefahren in sich.

Auch die Pflege und Wartung der Software, während sie beim Kunden im Einsatz ist, beeinflusst die Entscheidung über die Zielplattform. Spätestens dann, wenn die Auslieferung eines oder mehrerer Teilmodule nicht mehr ausreicht, um neuen Anforderungen der Kunden zu entsprechen und eine Neuimplementierung des Produkts diskutiert wird, müssen Sie sich den Kopf über alle oben erwähnten Themen erneut zerbrechen.

1.1.3 DLL-Hölle und Versions-Konflikte

Hinzu kommt das Thema *DLL-Hölle*. Im WinNT-Verzeichnis (und Unterverzeichnissen) eines Computers mit Windows 2000 Professional, auf dem drei oder vier Anwendungen wie z.B. Microsoft SQL Server, Microsoft Office und Corel Draw installiert sind, befinden sich weit über 2000 verschiedene DLLs. Insgesamt können es durchaus 4000–5000 DLLs in allen Verzeichnissen auf einem wie beschrieben eingerichteten Rechner sein. Dies gilt natürlich nicht nur für Windows NT; auch bei den anderen Windows-Betriebssystemen, wie z.B. Windows ME, Windows XP Home oder Professional häufen sich im Laufe der Zeit und mit der Installation jeder weiteren Anwendung diese Dateien (DLLs) und füllen die Verzeichnisse auf der Festplatte.

Wenn Sie nun ein Programm geschrieben haben, beinhaltet dies in der Regel nicht so viele Komponenten. Aber manchmal muss die ein oder andere davon ausgetauscht werden, und die neue Version muss immer noch kompatibel sein. Kompatibel bedeutet, dass andere Komponenten weiterhin auf die (ausgetauschte) neue Komponente verweisen und deren Funktionalitäten nutzen können, während gleichzeitig die Stabilität und/oder korrekte Funktionsweise der kompletten Anwendung immer noch gewährleistet sind.

Wenn eine Anwendung nun Gebrauch von Komponenten macht, die von anderen Anwendungen oder dem Betriebssystem zur Verfügung gestellt werden, muss der Code immer auf dem aktuellen Stand gehalten werden, um Inkompatibilitäten zu vermeiden oder ihnen vorzubeugen. Denn wenn sich die Implementierung einer Komponente, die von einer Anwendung verwendet wird, in zu großem Maße ändert, kann es passieren, dass die Anwendung nicht mehr stabil oder überhaupt nicht mehr läuft.

Es mag wohl keine Möglichkeit eines Schutzes gegen diese Art der Inkompatibilität geben. Aber wäre die Möglichkeit nicht von Vorteil, einfach angeben zu können, mit welcher Version einer DLL (oder anderen Komponenten) eine bestimmte Anwendung ausgeführt werden kann? Dies meint nicht eine Notiz auf der Verpackung, etwa in der Art: »*Diese Anwendung läuft mit der Komponente iofsh344.dll in der Version 2.17a.*«

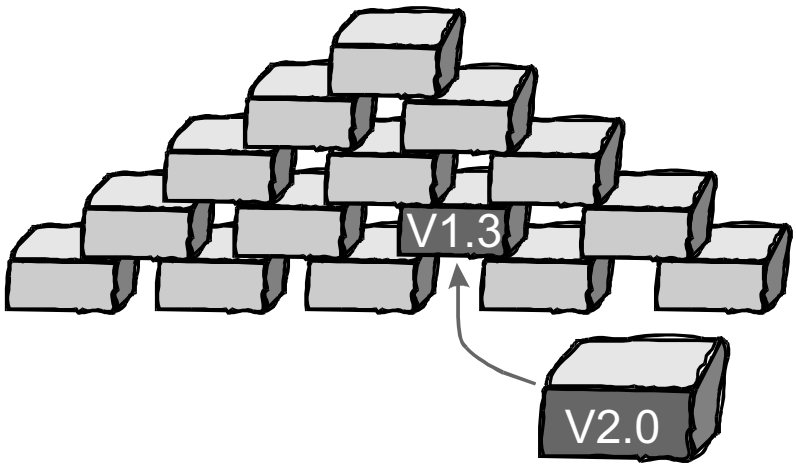

Abbildung 1.2: Bei einer Versionsänderung einer Komponente kann es zu Inkompatibilitäten kommen, die nicht nur die Stabilität einer einzelnen Anwendung, sondern sogar die eines kompletten Systems beeinträchtigen können.

Vielmehr sollte es möglich sein, mehrere unterschiedliche Versionen einer Komponente auf einem Computer zu installieren und zu registrieren(!) und verschiedene Anwendungen so zu implementieren oder zu konfigurieren, dass sie mit der entsprechenden Version der Komponente ausgeführt werden kann.

1.2 .NET – Programmierung mit Zukunft

Next Generation Web Services (*NGWS*) war der Name, unter dem Microsoft erstmals im Jahr 2000 Ideen und softwaretechnische Lösungen für eine neue Entwicklungs-Plattform und Architektur der Öffentlichkeit präsentierte. NGWS enthielt von Beginn an eine Laufzeitumgebung für Programme, die jedoch kein eigenständiges Betriebssystem darstellt. Dieses Laufzeitsystem lädt und verwaltet Programme, führt sie aus und kapselt dabei alle unterliegenden Ressourcen wie z.B. das Dateisystem, Netzwerk- und andere Protokolle oder Speicher(management).

Das Laufzeitsystem, zugehörige Bibliotheken und zusätzliche Dienste bilden ein so genanntes *Framework*. Basierend auf offenen Standards wie HTTP, XML und SOAP werden vorhandene Infrastrukturen genutzt, und das Internet wird zum zentralen Mittelpunkt der Architektur.

1.2.1 .NET – Was ist das?

NGWS ist in die deutsche Sprache nicht leicht zu übersetzen und entspricht in etwa »*Nächste Generation von Internetdiensten*«. Aufgrund der umfangreichen und mit fortschreitender Entwicklungszeit gewachsenen Funktionalität ist der Name *Next*

Generation Web Services irreführend. Das Framework bietet mehr als lediglich eine Basis für Internetdienste. Der heutige Name ist *Microsoft .NET* – oder kurz .NET – und es ist mittlerweile mehr eine Strategie als lediglich eine Plattform. .NET besteht aus der Kombination von genannten Internetdiensten, den .NET Enterprise Servern und dem .NET Framework.

Das Laufzeitsystem

Programme, die für .NET entwickelt worden sind, benötigen die .NET Laufzeitumgebung, um gestartet und ausgeführt werden zu können. Der Code, in dem diese .NET Anwendungen geschrieben worden sind, wird in der deutschen Dokumentation und vorhandenen Literatur als »verwalteter« Code bezeichnet. Da der Code jedoch nicht nur verwaltet wird, sondern noch viel mehr Dinge mit ihm und um ihn herum passieren (worauf im Folgenden noch eingegangen werden soll), wird hier die Bezeichnung *managed* für .NET Code bevorzugt und *unmanaged* für (traditionellen) Code, der ohne das .NET Laufzeitsystem ausgeführt werden kann.

Das bedeutet, dass die Laufzeitumgebung von .NET auf einem Computer installiert sein muss, um entsprechende Programme darauf laufen lassen zu können. Ein .NET Programm kann also nicht auf einem Computer ohne .NET laufen! Zusätzlich zum Programm und eventuell angebundenen oder abhängigen Komponenten wird bei einem Programmstart die .NET Laufzeitumgebung geladen und ausgeführt, sofern dies nicht bereits vorher geschehen ist. Dies führt zu einem hohen Anspruch an die zur Verfügung stehenden Ressourcen, wie z.B. die Größe des Arbeitsspeichers. Aber gemessen an den resultierenden Vorteilen ist das ein verhältnismäßig geringer Preis, der zu bezahlen ist.

Denn ein .NET Programm ist nicht von einem bestimmten Betriebssystem abhängig – vielmehr hat eine Applikation, die in der .NET Laufzeitumgebung ausgeführt wird, keinerlei Kenntnis über das unterliegende Betriebssystem. Mit anderen Worten: Einem Programm, das für .NET geschrieben und kompiliert worden ist und das in der .NET Laufzeitumgebung ausgeführt wird, ist es egal, auf welchem Betriebssystem .NET gerade läuft.[2]

Neben der beschriebenen Plattformunabhängigkeit bietet .NET auch zahlreiche Möglichkeiten der Interaktion von Komponenten. Der Quellcode eines .NET Programms kann zunächst in einer beliebigen Programmiersprache implementiert sein. Prinzipiell entwickelt ein Programmierer nach wie vor in einer Sprache seiner Wahl – sei es C++, Visual Basic oder Cobol – und zwar mit einem Werkzeug seiner Wahl – Kommandozeile oder IDE. Dieser von Menschen lesbare Code muss, bevor er von einem Computer ausgeführt werden kann, in eine entsprechende »Maschinensprache« transformiert werden. Ein entsprechender Compiler muss also in der Lage sein, den Quellcode in den entsprechenden .NET Code konvertieren zu können.

---

2 Dies gilt natürlich nur dann, wenn keine Dienste des Betriebssystems genutzt bzw. aufgerufen werden.

Kompilieren in .NET bedeutet, dass diese Transformation in zwei wesentlichen Schritten durchgeführt wird. Zunächst wird der Quellcode eines Programms von einem Compiler in eine Zwischensprache übersetzt. Diese Zwischensprache heißt *Intermediate Language* (IL). Dieser Vorgang findet zum Zeitpunkt der Programmentwicklung statt. IL ist eine Assembler-ähnliche Sprache. Die wichtigste Eigenschaft jedoch ist, dass die Intermediate Language nicht an eine bestimmte Hardwareplattform gebunden ist.

In .NET werden Anwendungen, Programme und Komponenten in der Regel im IL-Zwischenformat ausgeliefert und auf einem Computer physikalisch gespeichert. Dabei können einzelne Komponenten durch Verweise an Programme gebunden und auf diese Art wieder verwendet werden. Und erst zur Ausführungszeit wird IL dann von der Laufzeitumgebung in einem weiteren Schritt in den ausführbaren Code konvertiert.

Dienste fürs Internet

Web Services (bzw. so genannte Webdienste) sind solche Komponenten, die Dienste und Funktionalitäten zur Verfügung stellen. Angefangen bei einfachen Berechnungen über Kommunikationsdienste bis hin zu aufwändigen Datenbankoperationen können Web Services ein breites Spektrum verschiedenster Aufgaben übernehmen.

Doch sind Web Services nicht dazu gedacht, nur auf einem lokalen Computer, sondern speziell in einem Netzwerk zur Verfügung gestellt zu werden. Dort können die von ihnen angebotenen Dienste über lokale Rechnergrenzen hinweg in Anspruch genommen werden. Das Wichtige ist hierbei, dass Web Services grundsätzlich für das Internet konzipiert wurden. Die gebotenen Funktionalitäten und Dienste können von unterschiedlichsten Clients genutzt werden.

Ein Web Browser oder die Verwendung von mobilen Geräten sind ohne starke Interaktion und Kommunikation mit einem sog. User nur von geringem (wenn überhaupt von irgendeinem) Nutzen. Wohingegen Web Applikationen nicht interaktiv (ohne Eingriff oder Interaktion mit Menschen) miteinander arbeiten und kommunizieren können. Grundverschiedene Ansätze und Lösungen; doch können alle – vom Web Browser über mobile Geräte bis hin zu Web Applikationen bzw. Programmen – einen Web Service aufrufen und dessen Dienste nutzen.

Ein Web Service wird genau einmal implementiert und auf einem Server für Zugriffe von außen bereitgestellt. Von einem Web Browser aus kann unter Verwendung der entsprechenden URL z. B. eine Beschreibung des Web Service angefordert oder die exponierten Methoden aufgerufen werden. Es ist darüber hinaus jedoch möglich, innerhalb eines Programms übers Internet auf Web Services bzw. dessen Methoden zuzugreifen. Damit sind sie auch so etwas ähnliches wie Prozeduren bzw. Methoden. Web Services machen aufgrund ihrer beschriebenen Eigenschaften das interaktive Web zu einem programmierbaren Web.

.NET Enterprise Server

Aufbauend auf dem .NET Framework und den Web Services sind es die *.NET Enterprise Server*, die durch zahlreiche Funktionalitäten zusätzlichen Nutzen in Ihre Applikationen bringen können. Sie bieten Möglichkeiten zum Datenbankzugriff und Nachrichtenaustausch auf Basis von XML, Austausch von Daten in verschiedenen Standardformaten und vieles mehr.

Zu den Enterprise Servern zählen (ohne Anspruch auf Vollständigkeit): Microsoft Exchange Server 2000, Microsoft SQL Server 2000, Microsoft BizTalk Server 2000, Microsoft Commerce Server 2000, Microsoft SharePoint Portal Server 2001, Microsoft Host Integration Server 2000, Microsoft Internet Security and Acceleration Server 2000, Microsoft Application Center 2000 und Microsoft Mobile Information 2001 Server.

Zu den wichtigsten und bekanntesten dieser Produkte sei an dieser Stelle kurz die besondere Bedeutung für .NET erläutert:

Microsoft BizTalk Server 2000 beherrscht zahlreiche Standards des Informationsaustausches und bietet die Basis dafür, verschiedene Plattformen, Anwendungen und Dienste in automatisierten Arbeitsabläufen zu verbinden. Dabei werden Protokolle und Formate wie HTTP(S), SMTP, XML und SOAP unterstützt. BizTalk ist aufgrund seiner Funktionalitäten der Integrationsserver der .NET Plattform.

Zu den Produkten Microsoft SQL Server 2000 und Microsoft Exchange Server 2000 muss nicht viel erläutert werden. Nur soviel: Beide bieten nun umfangreiche Unterstützung von XML und die Möglichkeit des Zugriffs übers Internet mittels HTTP.

Der Microsoft Host Integration Server 2000 ist die Anwendung der Wahl, wenn es darum geht, Internet/Intranet, Client/Server und Host Systeme zu integrieren.

Microsoft Internet Security and Acceleration (ISA) Server 2000 basiert auf dem Sicherheitsmechanismus von Windows 2000 und beinhaltet u. a. eine Firewall. Der ISA Server ermöglicht dadurch einen sicheren und administrierbaren Internetzugriff.

Mit Hilfe der .NET Enterprise Server können .NET Applikationen um zahlreiche Funktionen erweitert werden. Sie können unter Verwendung verschiedener Standards (EDIFACT, XML, Emails/SMTP etc.) Informationen und Daten austauschen, Datenbanken integrieren, Web Storage Systeme und Intranet Portale aufbauen oder Dokumenten Management über Netzwerke (inkl. Internet) betreiben.

Dieses Zusammenspiel der verschiedensten Anwendungen und Dienste (Server) auf Basis ungleicher Datenformate wird *Orchestrierung* genannt. Dies führt zu einem flexiblen System, das es erlaubt, existierende Enterprise-Infrastrukturen mit .NET basierten Lösungen in Einklang zu bringen.

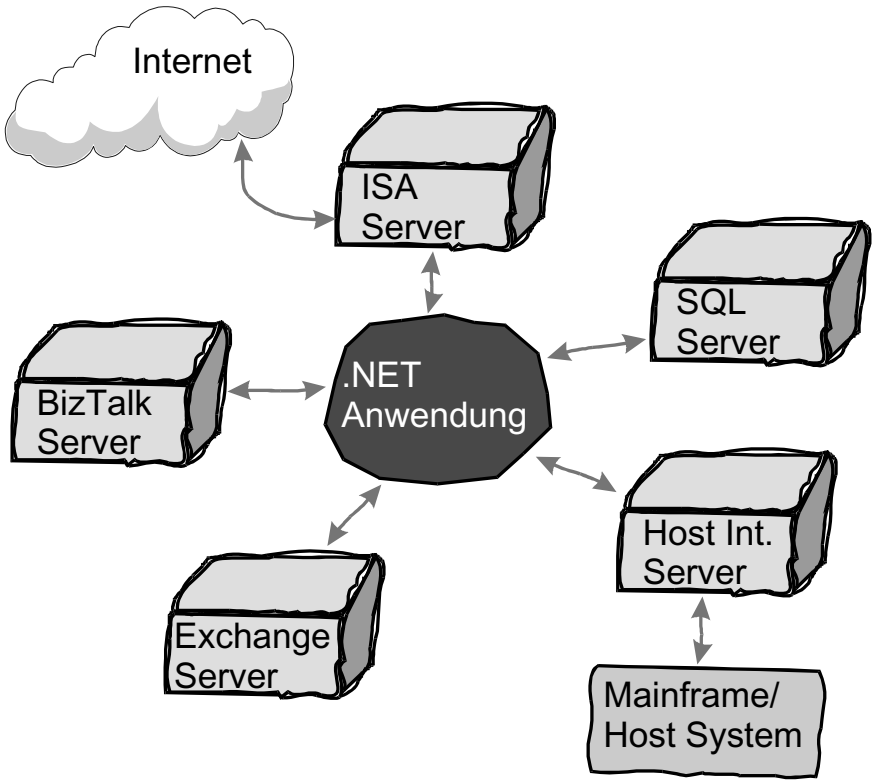

Abbildung 1.3: Rund um .NET gibt es eine Reihe von .NET Enterprise Server, welche die Entwicklung vereinfachen und die Funktionalität einer .NET-Applikation erweitern.

Programmiersprachen und Visual Studio

Um den Neuerungen und den damit verbundenen gehobenen Ansprüchen gerecht zu werden, wurden auch die Programmiersprachen Visual Basic und Visual C++ entsprechend angepasst und weiterentwickelt. Außer Änderungen in der Namensgebung, bei der die Versionsnummer weggefallen ist, erscheint vor allem Visual Basic .NET in einem völlig anderen Licht, während Visual C++ .NET nahezu unverändert ist.

Visual Basic ist für .NET zu großen Teilen von Grund auf neu konzipiert und entwickelt worden und unterstützt nun auch eine echte objektorientierte Entwicklung durch Konzepte wie z. B. *Vererbung* oder *Polymorphie*. Visual C++ .NET wurde um einige Funktionalitäten (Compileroptionen, etc.) erweitert. Die *Microsoft Foundation Classes* (*MFC*) und die *Active Template Library* (*ATL*) haben ebenfalls an Umfang gewonnen. Jedoch ist Visual C++ .NET grundsätzlich nicht dazu gedacht, Applikationen für .NET zu erstellen. Deshalb sind die wichtigsten Neuerungen bei C++ die *Managed Extensions*, mit deren Hilfe recht schnell bestehende

C++ Programme und Komponenten nach .NET portiert werden können. Der Visual C++ .NET Compiler kann also sowohl managed als auch unmanaged Code erzeugen.

Mit C# wurde der C-Sprachenfamilie eine neue komponenten-orientierte Sprache hinzugefügt, in der auch der größte Teil von .NET selbst geschrieben wurde. C# verbindet die Einfachheit von Visual Basic mit den Stärken von C/C++. C++-Entwickler werden zum Beispiel Header-Dateien vermissen, die in C# nicht existieren, wohingegen Visual Basic-Programmierer verstärkt mit Objekten jonglieren dürfen oder auch müssen.

Ein Entwickler benötigt eine entsprechende Entwicklungsumgebung, um Programme für .NET zu erstellen. Zusammen mit .NET stellt Microsoft eine neue Version des Visual Studio, Visual Studio .NET, vor und gibt einem Programmierer damit das passende Werkzeug an die Hand.

1.2.2 Was sind die Absichten und Ziele?

Grundidee von .NET ist es, eine Entwicklungsplattform fürs Internet zur Verfügung zu stellen, die gleichzeitig eine von Hardware und Betriebssystem unabhängige Laufzeitumgebung repräsentiert. Um zu verstehen, welche Vorteile .NET bringt und welche Absichten und Ziele dahinter stecken, werden nun die einzelnen Elemente von .NET etwas genauer betrachtet. Zuerst wird das .NET Framework und anschließend die *Web Services* (Web Dienste) untersucht.

Das .NET Framework

Das .NET Framework besteht aus verschiedenen Bausteinen, die jeweils unterschiedliche Aufgaben erfüllen und die ähnlich wie Legosteine – wenn Sie einem zweifachen Vater diese Analogie verzeihen – aufeinander aufsetzen und teilweise ineinander verzahnen.

Die unterste Ebene des Frameworks bildet die *Common Language Runtime* (*CLR*). Ihre Hauptaufgabe besteht darin .NET Applikationen zu verwalten und deren Ablauf zu überwachen, was übrigens auch der Grund ist, warum der Code, in dem .NET Programme geschrieben sind, *managed code* heißt. Die CLR ist also verantwortlich für das Laden und Ausführen von Applikationen und Komponenten. Weitere Aufgaben sind u. a. die Unterstützung des Sicherheitsmechanismus bzgl. Code und Ressourcen, das Überprüfen der Herkunft und Identität von Komponenten und die Kapselung unterliegender Schichten, wie z. B. die Dienste des Betriebssystems.

Die Common Language Runtime

Auch die CLR selbst ist modular aufgebaut und enthält verschiedene Elemente, die für die unterschiedlichen Aufgaben verantwortlich sind. Auf einige dieser Elemente wird an dieser Stelle etwas näher eingegangen.

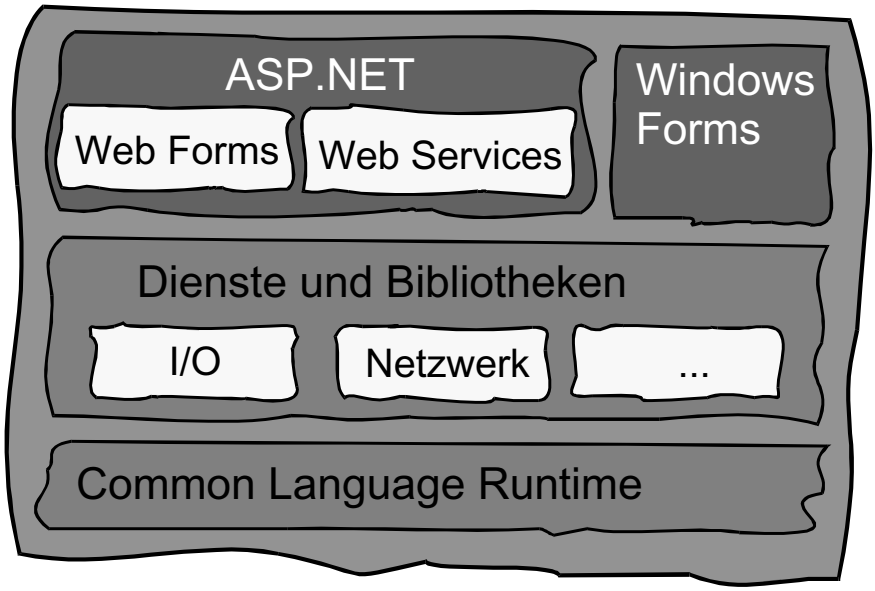

Abbildung 1.4: Das .NET Framework ist aus verschiedenen Schichten aufgebaut, die aufeinander aufsetzen und ein sehr strukturiertes und konsequentes Programmiermodell bieten.

Soll ein .NET Programm ausgeführt werden, sorgt darin enthaltener Code dafür, dass die .NET Laufzeitumgebung gestartet wird. Der *class loader* der CLR lädt die in IL vorliegenden Programme und Komponenten und bereitet diese zur Ausführung vor. Er löst alle Verweise auf und lädt die verwendeten Typen und zusätzliche Komponenten. Gleichzeitig unterstützt der Klassenlader (*class loader*) den Sicherheitsmechanismus, indem er Informationen, die von geladenen Programmen und Komponenten bereitgestellt werden, an die CLR weiterreicht. Dazu gehören z.B. die Herkunft einer Komponente und die Signatur des Herstellers.

Die so präparierte Applikation wird anschließend an einen Compiler weitergegeben, der dann IL in ausführbaren Code übersetzt. Dieser sogenannte *Just-In-Time* (*JIT*) Compiler wird ebenfalls von der CLR zur Verfügung gestellt. Die Trennung in zwei unabhängige Kompilierungsvorgänge hat große Vorteile. IL ist zunächst einmal hardware- und plattformunabhängig. Bei aktuellen Neuentwicklungen (ohne .NET) muss eine Optimierung bzgl. der Zielplattform und Hardware bereits zur Entwicklungszeit durchgeführt werden. Es müssen also zu einem recht frühen Zeitpunkt einige Annahmen und Verallgemeinerungen bzgl. Hard- und Software gemacht werden (siehe *1.1.1 Die Wahl der Zielplattform*). Mit .NET ist es nun jedoch möglich, dass erst zur Laufzeit (»just in time«) eine Optimierung erfolgt, die speziell auf das einzelne System zugeschnitten werden kann, auf dem die Laufzeitumgebung und damit das entsprechende Programm gerade ausgeführt wird.

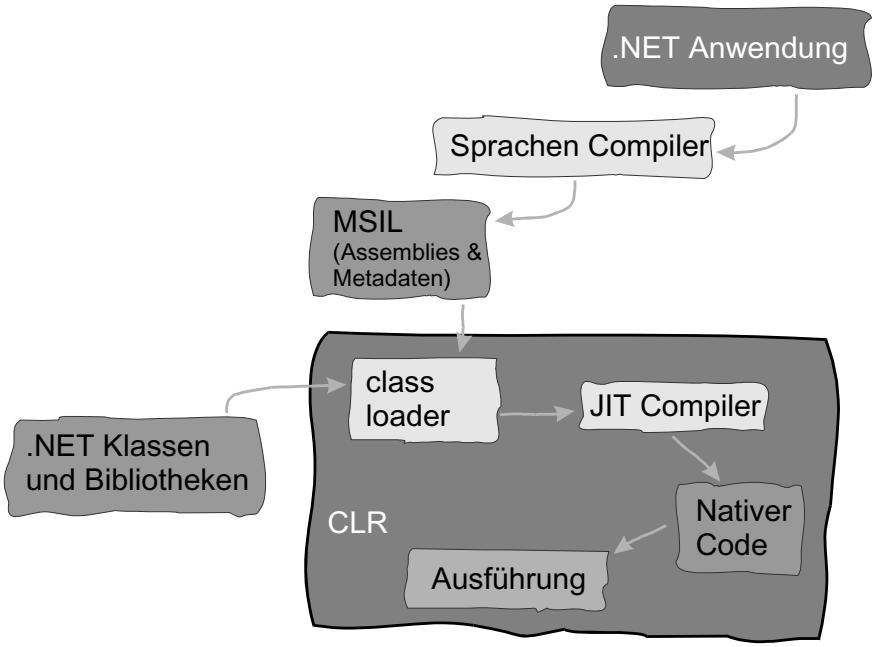

Abbildung 1.5: Die Ausführung eines Programms in .NET erfolgt unter der Kontrolle der Common Language Runtime (CLR). Die Kompilierung erfolgt in zwei wesentlichen Schritten, wovon der erste zum Entwicklungszeitpunkt und der zweite von einem JIT-Compiler erst zur Laufzeit durchgeführt wird.

Anschließend wird das Programm unter weiterer Kontrolle der CLR ausgeführt. Weitere Kontrolle bedeutet u. a.

- Ein Programm kann Bezüge zu weiteren Komponenten und entsprechende Verweise enthalten. Wie oben beschrieben werden diese Verweise zum Ladezeitpunkt aufgelöst, jedoch können auch solche enthalten sein, die nur in Abhängigkeit vom Programmablauf und daraus resultierenden Anforderungen aufgelöst werden können. In diesem Fall müssen die benötigten Komponenten nachgeladen werden.
- In Programmen kommt es nicht selten vor, dass benötigter Speicherplatz angefordert wird, der im weiteren Ablauf entsprechende Verwendung findet. Sobald er nicht mehr benötigt wird, sollte der Platz wieder freigegeben werden, damit er anderen, konkurrierenden Programmen zur Verfügung steht. Dieses Speichermanagement ist eine anspruchsvolle und teure Aufgabe; anspruchsvoll für den Entwickler und teuer, weil äußerst zeitaufwändig. In .NET wird die Speichernutzung von der CLR beobachtet und protokolliert, und das Speichermanagement erfolgt automatisiert – ein programmatischer Eingriff ist nicht mehr notwendig.

- Die CLR kann mehrere voneinander unabhängige Applikationen verwalten. Um eine echte Trennung zu ermöglichen, muss z. B. sichergestellt sein, dass Speicherbereiche eines bestimmten Programms nicht von anderen erreicht oder modifiziert werden können. Eine .NET Applikation muss vollständig isoliert sein.

Erfüllt Code bestimmte Regeln und Vorgaben, wird er als *typensicher* (*typesafe*) bezeichnet. Die Vorgaben für typensicheren Code sind folgende. Er darf:

a. nur auf Speicher zugreifen, der ihm explizit zugewiesen wurde, und

b. auf Typen nur durch deren veröffentlichte Schnittstellen zugreifen.

Typensicherer Code wird zunächst vom Sprachencompiler generiert und später noch einmal vom JIT Compiler überprüft und verifiziert. Was aber sind nun Typen? Typen repräsentieren Werte und stellen gleichzeitig eine Art Vertrag dar, der von diesen Werten erfüllt werden muss. .NET unterscheidet dabei zwischen *Werte-* (*value types*) und *Referenztypen* (*reference types*) bzw. Objekten. Wertetypen sind Typen, von denen immer eine Kopie übergeben wird, wenn sie als Parameter einer Methode verwendet werden. Im Gegensatz dazu werden bei Verwendung von Objekten Referenzen übergeben. Dies sind Verweise auf dasselbe Objekt, was bedeutet, es wird dabei keine Kopie des Objektes erzeugt.

Das *Common Type System* (*CTS*) – ein weiterer Baustein der CLR – enthält eine Beschreibung all jener Typen, die in .NET verfügbar sind. Diese Typen haben auch Methoden, die auf die Typen selbst bzw. auf deren Datenelemente angewandt werden können. Unterstützt ein Compiler einer Programmiersprache .NET, muss er nicht zwangsläufig auch alle .NET Typen unterstützen. Die *Common Language Specification* (*CLS*) bildet daher eine Untermenge des CTS und enthält eine größte gemeinsame Untermenge der Typen, die eine .NET Sprache grundsätzlich unterstützen sollte.

Dadurch ist gewährleistet, dass alle in .NET Programmen verwendeten Typen an einer zentralen Stelle definiert sind. Das bedeutet, dass ein *Integer* in C++ genauso aussieht, wie ein *Integer* in Visual Basic; nämlich ein 4 Byte bzw. 32 Bit langer Integer-Wert mit einem Wertebereich von -2.147.483.648 bis +2.147.483.647. Da aber die Typen in den verschiedenen .NET Sprachen in der CLS bzw. CTS definiert sind, ist es mit .NET darüber hinaus möglich, von einer in Visual Basic geschriebenen Basisklasse abzuleiten und eine Klasse in C++ zu schreiben, die von dieser erbt. Common Type System und die Common Language Specification gemeinsam sind es, die im Wesentlichen die Basis für diese Art der Interaktion bilden.

Dienste und Bibliotheken

Oberhalb der CLR befindet sich das *Services Framework*. Diese Schicht stellt Bibliotheken und Dienste zur Verfügung, deren Klassen von jeder beliebigen .NET Sprache aus aufrufbar sind. Dies beinhaltet Klassen für Datenbankzugriffe, Ein/Ausgabeklassen, Klassen für Netzwerkzugriffe, Debugging-Klassen für Programmierwerkzeuge, usw.

Innerhalb dieser Klassenbibliotheken werden alle Klassen in *Namespaces* strukturiert und verwaltet. Ein Namespace beinhaltet Klassen, die einen bestimmten Bereich oder ein Konzept von .NET implementieren. So enthält beispielsweise der Namespace *System.Web* alle untergeordneten *Subnamespaces*, wie *System.Web.UI*, *System.Web.Security* oder *System.Web.Configuration*. Darin enthalten sind alle zugehörigen Klassen, die entsprechende Methoden und Eigenschaften repräsentieren. Um beim genannten Beispiel *System.Web* zu bleiben, sind Methoden bzgl. der Benutzerschnittstelle (*user interface – UI*) einer Web Applikation in *System.Web.UI* oder Methoden der Konfiguration (*Configuration*) in *System.Web.Configuration* enthalten.

Alles in .NET ist ein Objekt! Alle Objekte in .NET erben direkt oder indirekt von *System.Object* – die Mutter aller Objekte. Zum Beispiel lässt sich die Ahnenlinie des Elements *Button* einer Webseite etwa so darstellen:

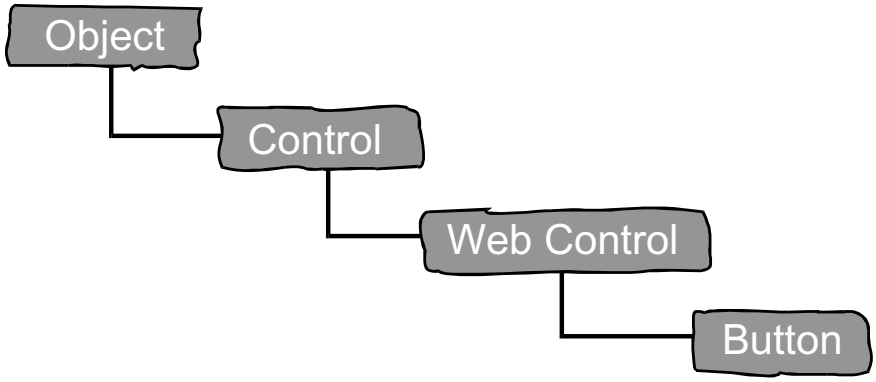

Abbildung 1.6: Die Ahnenlinie der Klasse Button – alles erbt direkt oder indirekt von System.Object.

Das Services Framework stellt nicht nur Klassen zur Verfügung, die in einer Standard Klassenbibliothek erwartet werden können. Es sind auch Klassen enthalten, die Zugriff auf Dienste des Betriebssystems ermöglichen. Dadurch muss sich ein Entwickler nicht um die Details kümmern und – dies ist der wesentliche Vorteil – er kann seine Applikationen plattformunabhängig erstellen. Die Plattformunabhängigkeit bezieht sich selbstverständlich auf eine Hardware und Betriebssystemunabhängigkeit, denn die Implementierungen müssen doch auf .NET abgestimmt sein.

ASP .NET, Web Services und Windows Forms

Die oberste Schicht des Frameworks bilden die *Active Server Pages* (*ASP .NET*) und – parallel dazu – *Windows Forms*. ASP .NET bietet die Möglichkeit, Benutzeroberflächen und sowohl sichtbare als auch nicht-sichtbare (Steuer)Elemente von Internetseiten mit Software-Komponenten zu kombinieren.

Durch die Web Forms ist es nun möglich, Layout und Logik von ASP .NET Seiten vollständig voneinander zu trennen. Zur Implementierung der Logik – gemeint ist hier die unterliegende Funktionalität einer Page – können nun jedoch Compilersprachen (inkl. Skriptsprachen) verwendet werden. Dieser Code wird nicht mehr wie beim Vorgänger ASP mit jedem Zugriff interpretiert, sondern kompiliert. Dies geschieht nur ein einziges Mal, und zwar genau dann, wenn der erste Zugriff auf eine bestimmte ASP .NET Seite erfolgt; sie wird kompiliert und dann sofort ausgeführt. Bei jedem folgenden Zugriff wird dann die bereits kompilierte, in IL vorliegende Datei geladen und ausgeführt. Kapitel 4 beschäftigt sich ausführlich mit ASP .NET.

Windows Forms unterstützt die mehr traditionelle Art, Windows Programme inklusive Benutzeroberfläche zu schreiben. Damit kann einfach und schnell auf die von den unterschiedlichen Windows-Versionen (alle, auf denen .NET-Programme ausgeführt werden können) zur Verfügung gestellten Dienste zugegriffen werden. Auch den Windows Forms ist ein eigener Abschnitt in diesem Buch gewidmet.

Heutzutage müssen Applikationen in kurzer Zeit entwickelt werden können, aber eine unkomplizierte Nutzung gestatten und – im Falle von Komponenten – einfach wiederzuverwenden sein. Gleichzeitig sollten sie möglichst sprachen- und plattform-unabhängig sein. Basierend auf den beschriebenen Konzepten werden diese Anforderungen von .NET Applikationen und Komponenten erfüllt. Ein weiteres zentrales Element sind die Web Services, die außerdem eine Kommunikation und Interaktion übers Internet ermöglichen.

Web Services sind *Black Boxes*, die ihre Funktionalität unter Verwendung offener Standards wie z.B. HTTP und XML zur Verfügung stellen. Dadurch können Web Applikationen miteinander verbunden werden. Der wichtigste Aspekt ist, dass Web Services ohne das Wissen wieder verwendet werden können, in welcher Sprache sie implementiert sind.

Die unterschiedlichsten Clients können unter Verwendung der entsprechenden URL auf Web Services zugreifen: Web Browser, mobile Geräte und auch andere Web Services können die Dienste in Anspruch nehmen. Verteilte Web-Applikationen können also einfach und schnell erstellt werden.

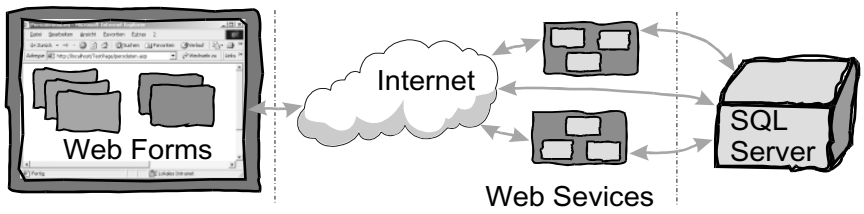

Abbildung 1.7: Ein Beispiel einer 3-schichtigen Web Applikation mit .NET: Die Web Forms bilden die Benutzerschnittstelle (user interface, UI) und in der mittleren Schicht (middle tier) können Web Services zum Einsatz kommen.

Bei einer 3-schichtigen Architektur (*three-tier-model*) einer Anwendung fürs Internet repräsentieren beispielsweise Windows Forms bzw. Web Forms die Benutzerschnittstelle, die mittlere Ebene besteht aus dem Internet und evtl. Web Services, während die dritte Ebene einen Server wie z. B. einen der .NET Enterprise Server enthalten könnte.

1.2.3 Ausrichtung der Programmierung

Microsoft .NET besteht aus einer Kombination von .NET Framework, den .NET Enterprise Servern und verschiedenen .NET Diensten, wie z. B. Web Services oder ASP .NET. Natürlich werden alle drei Gruppen wesentlichen Einfluss auf die Programmierung haben. Dabei verspricht Microsoft .NET, Anwendungen, Dienste und Geräte einfacher und sicher zu machen und deren Bedienung, Funktionalitäten und Erscheinungsbild zu personalisieren.

Neue Technologien wie beispielsweise die Web Forms von ASP .NET oder Windows Forms bringen neue Möglichkeiten, Flexibilität und Sicherheit auf die Seite des Clients. Nicht nur .NET Anwendungen, die eine Interaktivität mit dem Menschen erfordern, sondern auch (und vor allem) Web Services können radikale Änderungen in Business-Applikationen fürs Internet hervorrufen.

Die .NET Enterprise Server sind oben bereits behandelt worden. An dieser Stelle sei lediglich noch einmal hervorgehoben, dass diese Gruppe von Servern sehr umfangreiche und wertvolle Unterstützung für (XML-basierte) Web Applikationen bieten kann. Im Bereich der Verteilung und Verbreitung von Web Services, der Verwaltung und der Orchestrierung haben die .NET Enterprise Server einen hohen potentiellen Nutzen.

Der Dreh- und Angelpunkt des ganzen Systems sind die Web Services. Sie stellen wiederverwendbare Komponenten dar, die so konzipiert sind, dass sie XML- und/oder SOAP-basierten Zugriff von anderen Web Services und Web Anwendungen übers Internet ermöglichen. Damit wird das rein interaktive Web, wie Sie es heute kennen, zum programmierbaren Web.

Da .NET auf XML und weiteren offenen Standards basiert, wird die Integration unterschiedlichster Ressourcen wie Web Services oder Web Storage Systeme in verteilten Anwendungen übers Internet möglich.

Dank der universellen »Basis-Sprache« XML können die unterschiedlichsten Clients mit .NET arbeiten und Web Services aufrufen und deren Funktionalität konsumieren. Ob PC, Laptop, Workstation, Telefon, Handheld PC oder eine Spielekonsole, solange die jeweils installierten Applikationen XML »sprechen und verstehen« können, steht einer Interaktion untereinander oder mit einem Server nichts im Wege.

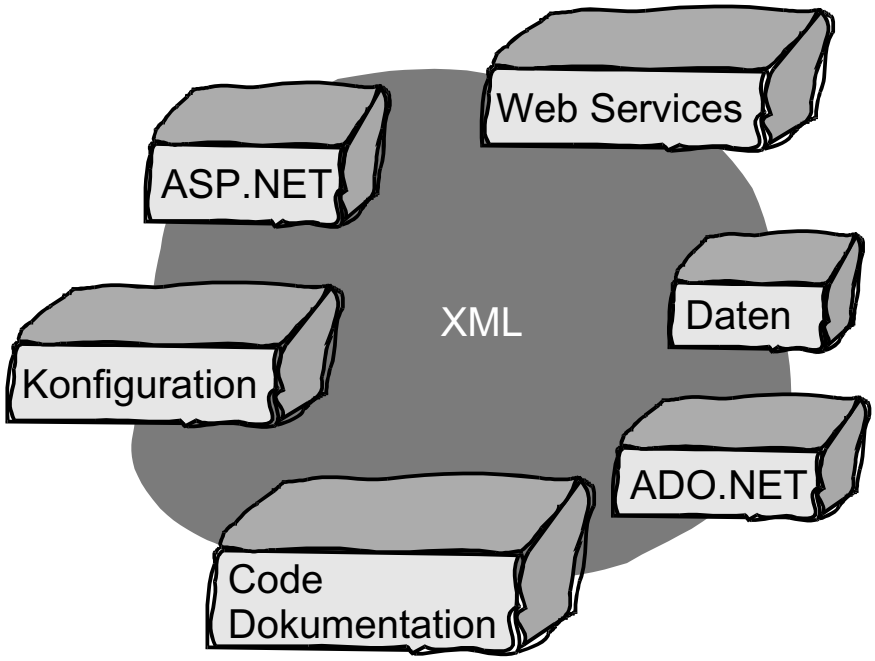

Abbildung 1.8: XML ist überall in .NET zu finden. Auch einige der .NET Enterprise Server (hier nicht dargestellt) verwenden XML z.B. zur internen Kommunikation.

XML

Die *eXtensible Markup Language* (*XML*) findet in vielen Bereichen von .NET Verwendung. Dort wird sie z.B. als Nachrichten-Format in der Kommunikation oder als Format für Konfigurationsdateien eingesetzt. Durch XML werden der Datenaustausch und die Integration von Software sehr vereinfacht. Sie ist ein offener und anerkannter Standard – standardisiert vom *World Wide Web Konsortium* (*W3C*) in 1998. XML ist eine Anwendung des ISO-Standards *SGML* (*Structured General Markup Language*) und bietet – anders als z.B. HTML – die Möglichkeit zur Modifikation und Erweiterung. Das bedeutet, dass neue, eigene Tags (auch XML-Befehle oder -Elemente genannt) hinzugefügt werden können.

XML macht den Austausch von Daten auch übers Internet sehr einfach, und .NET bietet die Lösungen und das Werkzeug mit diesen (XML-) Daten zu arbeiten. Anders als die meisten anderen Formate, die zum Austausch von Daten verwendet werden (SWIFT, X12, EDIFACT, etc.), ist XML sehr einfach und selbstbeschreibend. *XML-Dokumente* enthalten Daten und weitere Informationen, die eine detaillierte, hierarchische Beschreibung dieser Daten repräsentieren. Als Meta-Auszeichnungssprache mit den genannten Eigenschaften liegt ein großer Vorteil darin, dass XML auf unterschiedlichen Betriebssystemen in verschiedenen Anwendungen verwendet werden kann.

XML dient zur Trennung von Daten auf der einen und der Darstellung dieser Daten auf der anderen Seite. Die Verwendung von XML beschränkt sich jedoch selten auf die Arbeit mit der in der XML 1.0 Spezifikation des W3C festgelegten Version (vg. http://www.w3c.org). In der Regel kommen noch *Dokumenttyp-Definitionen* (*DTD*), *XML Schema*, *XML Stylesheet Language Transformations* (*XSLT*) und evtl. andere Standards hinzu.

DTD

Eine DTD enthält eine strukturelle Beschreibung des Aufbaus und der logischen Elemente einer Klasse von XML-Dokumenten. Durch sie wird festgelegt, welche Elemente in einer gültigen Instanz in welcher Reihenfolge und Häufigkeit auftreten dürfen. So lässt sich beispielsweise bestimmen, dass ein Element weitere Elemente enthalten darf, dass Elemente mindestens einmal (1) vorkommen dürfen oder welches Format die Daten innerhalb der Elemente haben sollen.

In diesem Sinne ist eine DTD eine Art Schablone, die dazu dient, die korrekte Form und Gültigkeit eines Dokuments zu verifizieren. Ein XML-Dokument, dessen Inhalt sich an die Regeln der W3C-Spezifikation hält, wird als *wohlgeformt* bezeichnet. Soll es darüber hinaus ein *gültiges* Dokument sein, muss es eine Dokumenttyp-Deklaration enthalten (und den Vorgaben der angegebenen DTD folgen). Die Dokumenttyp-Deklaration ist die Deklaration der DTD und gibt den Ort an, wo sich die DTD befindet. Ein wohlgeformtes XML-Dokument mit einer durch eine DTD vorgegebenen Struktur wird auch als *gültige Instanz* bezeichnet.

Um Ihnen einen besseren Eindruck von XML und der Verwendung von DTD zu verschaffen, sehen Sie hier zunächst ein gültiges XML-Dokument:

Listing 1.1: Ein Beispiel einer XML-Datei mail.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message SYSTEM "mail.dtd">
<mail>

  <target>Andreas Maslo</target>
  <source>Joerg Freiberger</source>
  <subject>Developers Guide</subject>

  <message>
    Hallo Andreas,
    viele Gruesse vom Niederrhein
    mfg,
    Joerg.
  </message>
</mail>
```

QUELLTEXT ZU XML
Die in Listing 1.1 gezeigte XML-Datei *mail.xml* ist auf der Buch-CD im Verzeichnis *\Programme\Kap1* enthalten.

Die erste Zeile enthält die optionale XML-Deklaration. Sie muss, wenn sie angegeben wird, am Beginn des Dokumentes stehen. Hier werden u. a. die verwendete XML-Version und der Zeichensatz spezifiziert. Um das Dokument zu einem gültigen Dokument zu machen, folgt in der zweiten Zeile die Angabe der Dokumenttyp-Deklaration.

Das Element der obersten Ebene dieses XML-Dokuments ist `<mail>`. Darin eingebettet lassen sich die Elemente `<target>`, `<source>`, `<subject>` und `<message>` erkennen.

Im Beispiel wird in der Dokumenttyp-Deklaration auf die DTD *mail.dtd* verwiesen, die im Listing 1.2 wiedergegeben ist.

Listing 1.2: Die zum Beispiel in gehörende DTD-Datei mail.dtd

```
<!-- mail DTD Version 1 -->
<!ELEMENT mail (target, source, subject, message) >
<!ELEMENT target (#PCDATA)>
<!ELEMENT source (#PCDATA)>
<!ELEMENT subject (#PCDATA)>
<!ELEMENT message (#PCDATA)>
```

QUELLTEXT ZU DTD
Die in Listing 1.2 gezeigte DTD-Datei *mail.dtd* ist auf der Buch-CD im Verzeichnis *\Programme\Kap1* enthalten.

In der DTD finden sich die Elemente aus dem XML-Dokument wieder. Es wird in Zeile 2 eine *mail*, die aus einem Empfänger (*target*), einem Sender (*source*), einem *subject* und der eigentlichen *message* besteht, definiert. Diese Elemente selbst enthalten reinen Text (*Parsed Character Data – PCDATA*), was in den Zeilen 3 bis 6 festgelegt wird.

Wenn Sie beide Dateien in einem Verzeichnis ablegen und das XML-Dokument mit einem Web Browser aufrufen, sollten Sie etwa das folgende Bild sehen:

XML Schema und Namespaces

XML Schema stellt eine Alternative zu DTD dar. Beides sind Sprachen zur Definition von XML Dokumenten. DTD hat jedoch den Nachteil, dass sie nicht in XML geschrieben ist. Dadurch kann sie nicht – im Gegensatz zu XML Schema – erweitert werden. Außerdem ist ein XML Schema einfacher zu lesen, zu schreiben und zu verstehen.

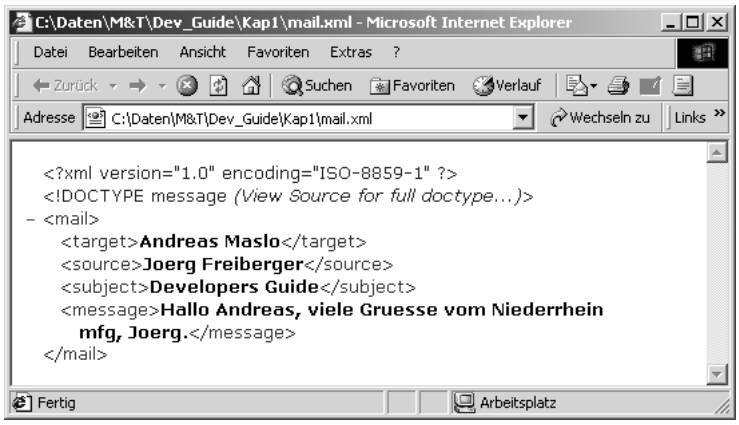

Abbildung 1.9: Die Beispiel-Datei mail.xml im Internet Explorer

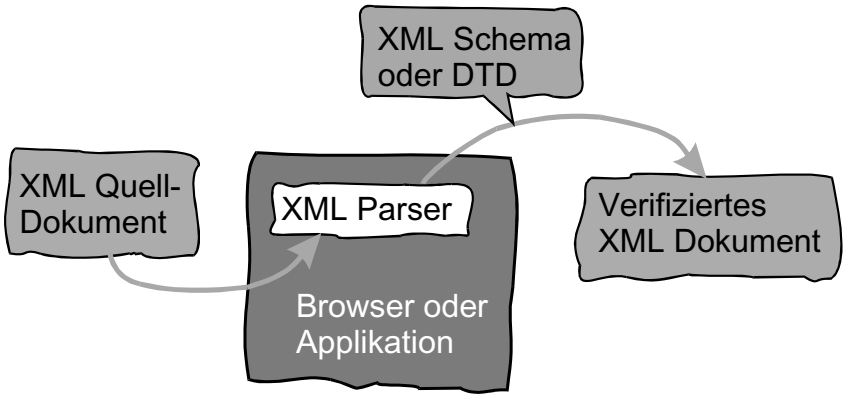

Abbildung 1.10: Die Verifikation eines XML Dokuments erfolgt unter Zuhilfenahme von XML Schema bzw. DTD.

Auch ein XML-Schema dient zur Verifikation der Gültigkeit eines XML-Dokuments. Ein XML-Parser, der Bestandteil eines Web Browsers oder einer anderen Applikation sein kann, übernimmt die Überprüfung anhand einer vorhandenen DTD-Datei bzw. eines XML Schemas. Das obige Beispiel einer DTD würde als XML-Schema in etwa wie folgt aussehen:

Listing 1.3: XML-Schema Datei mail.xsd zum Beispiel aus Listing 1.1

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  <xsd:complexType name="mail">
    <element name="target" type="string"/>
    <element name="source" type="string"/>
    <element name="subject" type="string"/>
```

```
    <element name="message" type="string"/>
  </xsd:complexType>
</xsd:schema>
```

QUELLTEXT ZU XML-SCHEMA
Die in Listing 1.3 gezeigte XML-Schema-Datei *mail.xsd* ist auf der Buch-CD im Verzeichnis *\Programme\Kap1* enthalten.

In der ersten Zeile wird im Wurzelelement *schema* unter Verwendung des Attributs *xmlns* ein Namespace angegeben – hier der Namespace des W3C für XML-Schema. Dieser Namespace legt die zulässigen Elemente für die Schema-Datei fest. Die zweite Zeile enthält das beginnende Tag der nächsten, dem Wurzelelement untergeordneten Ebene und deklariert einen komplexen Typ mit Namen *mail*. Die folgenden Zeilen sind selbsterklärend (sie geben die in der `mail` enthaltenen Elemente und deren Datentypen an).

Namespaces (manchmal auch als Namensräume bezeichnet) werden von den Menschen bereits seit einigen tausend Jahren verwendet. Jedoch wurden Namespaces erst in der jüngeren Geschichte wirklich wichtig. Ein gutes Beispiel einer Anwendung von Namensräumen ist die Post. Wenn Sie beispielsweise einen Brief an eine Ihnen bekannte Person senden wollen, schreiben Sie sicherlich nicht nur den Namen dieser Person auf den Umschlag. Da davon auszugehen ist, dass es auf dieser Welt mehr als einen Andreas Maslo oder Joerg Freiberger gibt, wird dem Namen noch die Straße und den Wohnort hinzugefügt. Mal abgesehen davon, dass Sie der Post dadurch natürlich eine Menge Arbeit ersparen, befinden sich die Namen der Personen in zwei Namespaces: Namespace 1 ist die Straße und Namespace 2 der Wohnort.

XML-Elemente (und Daten) können ebenfalls Namespaces zugeordnet werden, so dass innerhalb des selben Dokuments die Elemente <*Book*> und <*Author*> ein untergeordnetes Element <*Titel*> enthalten können:

```
<?xml version="1.0" encoding="UTF-8">
 [...]

<Book>
  <Titel>.NET Developer Guide</Titel>
  <Publisher>
    [...]
</Book>

<Author>
  <Titel>Dipl.-Ing.</Titel>
  <Name>
    [...]
</Author>

[...]
```

XSLT

XML Stylesheet Language Transformations (*XSLT*) ist eine Spezifikation des W3C und eine »Unter-Spezifikation« der *XML Stylesheet Language* (*XLS*). XSLT ist eine »aktive« Sprache und kann dazu verwendet werden, ein XML-Dokument in ein anderes textbasiertes Dokument wie z. B. eine HTML-Seite zu transformieren.

Sollen z. B. aus dem obigen Beispiel alle Betreffzeilen der Nachrichten innerhalb des XML-Dokuments als HTML-Datei ausgeben werden, könnte die entsprechende XSLT-Datei wie folgt aussehen:

Listing 1.4: XSLT-Datei mail.xsl zum Beispiel aus Listing 1.1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl_stylesheet
  xmlns:xsl=http://www.w3.org/1999/XSL/Transform
  version="1.0">
<xsl:template match="/">
  <html>
  <body>
    <p>Betreffzeilen Ihrer Nachrichten</p>
    <xsl:for-each select="mail/subject">
      <xsl:value-of select="."/><br/>
    </xsl:for-each>
  </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

> **QUELLTEXT ZU XSLT**
> Die in Listing 1.4 gezeigte XSLT-Datei *mail.xsl* ist auf der Buch-CD im Verzeichnis *\Programme\Kap1* enthalten.

SOAP

Das *Simple Object Access Protocoll* (*SOAP*) ist ein auf HTTP und XML basierendes Protokoll, das zum Austausch von Daten und Nachrichten im Internet dient. SOAP ist eine herstellerunabhängige Spezifikation, unterstützt lose verbundene verteilte Anwendungen und ermöglicht echte verteilte Interaktion und Interoperabilität. SOAP ist gut skalierbar, da es auf HTTP baut.

Doch SOAP ist nur ein Netzwerkprotokoll. Deshalb kümmert es sich nicht um den Status einer Verbindung oder die Speicherbereinigung. Auch zum Thema Sicherheitsmechanismen gibt es in SOAP selbst keine Lösung. Doch aufgrund seiner Herkunft und Abstammung von XML ist SOAP in all diesen Bereichen flexibel erweiterbar!

.NET als Entwicklungsumgebung

Anwendungen, die für .NET entwickelt wurden, enthalten managed Code und werden vom .NET Framework geladen, ausgeführt, verwaltet und kontrolliert. Dieser Vorgang wird als *Managed Execution* bezeichnet, der unter der Regie der CLR durchgeführt wird. Die CLR übernimmt dabei die Verwaltung und Kontrolle über den Speicher, Threads, Objekte und Ressourcen. Ein Garbage Collector räumt alle nicht mehr benötigten Ressourcen und Speicher auf, so dass sich ein Programmierer nicht mehr darüber den Kopf zerbrechen muss.

Alle Compilersprachen, die von .NET unterstützt werden, können die Dienste und Funktionalitäten des .NET Framework nutzen. Dies schließt auch die sprachenübergreifende Software-Integration ein. Damit können Klassen, die in einer der .NET-Sprachen implementiert wurden, von Klassen aufgerufen und sogar abgeleitet werden können, die in anderen Compilersprachen geschrieben wurden.

Aber auch die Interaktion von .NET Code mit unmanaged Code ist möglich. Unter *COM Interop* und *PInvoke* werden in .NET Lösungen und Technologien zusammengefasst, die eine Verwendung von COM-Objekten (*Component Object Model – COM*) in .NET und umgekehrt zulassen. Damit kann z.B. bestehender COM-Code in eine .NET Anwendung integriert werden.

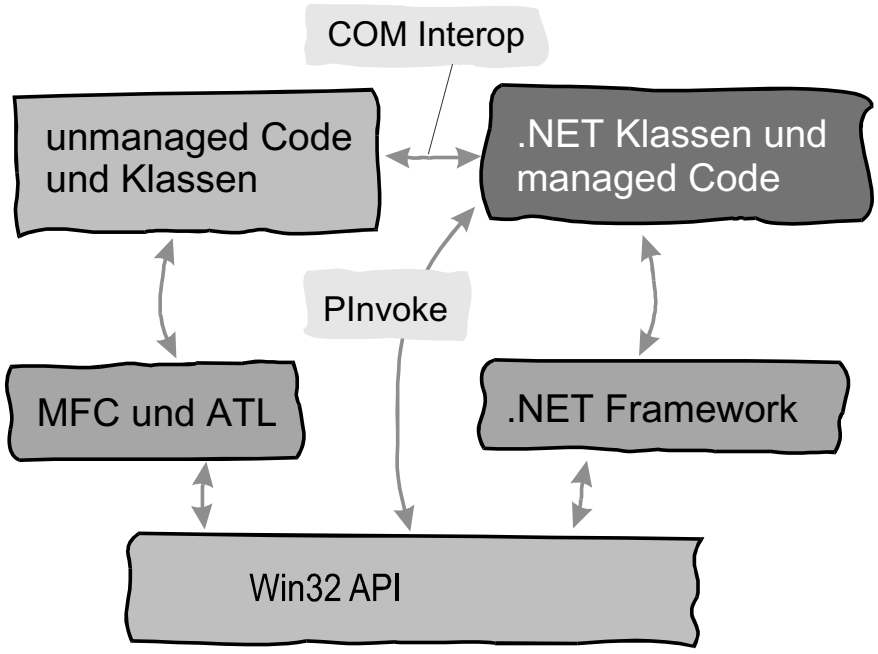

Abbildung 1.11: Mittels COM Interop und PInvoke können Ihre .NET Anwendungen direkt Aufrufe in die Win32 API machen oder mit (evtl. vorhandenem) unmanaged Code zusammenarbeiten.

Bei gleichzeitigem Einsatz von managed und unmanaged Code in ein und derselben Applikation ist jedoch Vorsicht geboten. Zum Beispiel können nicht (ohne weiteres) die Vorzüge des .NET Code-Sicherheitssystems für unmanaged Code verwendet werden. So muss auch z.B. bei der Konfiguration einer Web Applikation in ASP .NET darauf geachtet werden, dass die Einstellungen lediglich die ASP .NET Dateien und nicht die anderen Dateien (wie .jpg, .asp, htm, etc.) betrifft.

Typen repräsentieren eine Art Vertrag zwischen dem Anwender und dem System, das die Typen zur Verfügung stellt. Erfüllt ein Programm oder ein Stück Code einen solchen Vertrag in .NET, wird der Code als typensicher eingestuft. Dieser Code greift nur auf dem Speicher zu, der ihm explizit zugewiesen worden ist. Außerdem darf dieser Zugriff nur durch wohl definierte Interfaces (i.a. als Schnittstellen bezeichnet) erfolgen.

Der Garbage Collector – Ein Müllmann

Eine große Herausforderung in der Entwicklung mit C/C++ oder auch mit Visual Basic (z.B. Arbeiten mit Objekten) ist das Speichermanagement. Objekte, die im Speicher angelegt worden sind, müssen auch wieder sorgfältig entfernt werden. Dies liegt allein in der Verantwortung des Programmierers. Dynamisch angeforderter Speicher, der ab einem bestimmten Zeitpunkt nicht mehr benötigt wird, sollte (bis auf das letzte Bit) wieder freigegeben werden. Arbeitet ein Programmierer hier nicht sauber und sorgfältig, kann es zu sog. Speicherlöchern kommen.

Eine Anwendung auf einem Server, die unter Umständen einige Tage, Wochen oder sogar Monate läuft, nimmt dann immer mehr Speicher (Bit für Bit) in Anspruch, der nicht wieder freigegeben wird. Irgendwann, wenn kein Speicher mehr zur Verfügung steht, kommt es dann zum Absturz des Servers. Diese Art von Fehler (*memory leaks*) ist in den meisten Fällen sehr schwer zu finden und verursacht gegenwärtig in der Entwicklung allgemein sehr hohe Kosten und Zeitverluste.

Das .NET Laufzeitsystem (CLR) bietet eine Lösung für dieses Problem. Der sog. *Garbage Collector* übernimmt das gesamte Speichermanagement Ihrer .NET Anwendungen. Ein Programmierer muss sich nicht mehr den Kopf über die Freigabe des angeforderten Speichers machen; das wird automatisch vom Garbage Collector gemacht.

Jede .NET Anwendung hat eine Liste von Referenzen, mit (u.a.) folgenden Einträgen:

a. Objekte, die im managed Heap angelegt worden sind

b. Objekte, die nicht mehr benutzt werden und auf null gesetzt sind

Diese Referenzen werden als *Roots* (Wurzeln) bezeichnet. Der Garbage Collector hat Zugriff auf diese Liste, überprüft die Einträge und baut einen hierarchischen Baum aller Objekte, auf die noch zugegriffen werden kann. Alle Objekte die danach nicht Teil dieses Baumes sind, werden vom Garbage Collector als »Müll«

gekennzeichnet und später entsorgt. Später heißt hier wirklich später. Denn der Zeitpunkt, wann der Garbage Collector den Müll aufräumt, kann nicht genau spezifiziert werden – er macht dies, wenn »ihm gerade danach ist«. Bleiben nach dem Entsorgen Lücken im Aufbau des von der Applikation verwendeten Speichers, wird dieser auch noch defragmentiert. Dabei werden evtl. veränderte Adressen und Referenzen auf diese Adressen korrigiert.

Manch einer mag sich nun fragen: »Gibt es dann noch eine Lebensberechtigung für Destruktoren«?

Das Konzept des Garbage Collectors alleine macht einen Destruktor noch nicht unnötig. Denn ein Garbage Collector kümmert sich lediglich um die Verwaltung des Speichers. Andere Ressourcen, wie z.B. Verbindungen zu Datenbanken oder geöffnete Dateien müssen immer noch vom Programmierer aufgeräumt werden.

Unterstützt werden sie dabei durch die Methode *finalize*. Ein Programmierer kann diese Methode für eine Klasse implementieren und dort den für die Aufräumarbeiten notwendigen Code platzieren. Problem hierbei: der Garbage Collector ruft diese Methode automatisch kurz vor dem Tod eines Objekts auf (die letzte Ölung – quasi). *finalize* kann nicht explizit aufgerufen werden. Wie Sie eben erfahren haben, ist der Zeitpunkt, wann der Garbage Collector ein Objekt aufräumt, nicht vorhersagbar. Mehr Kontrolle darüber bietet die Verwendung der Methoden *Close* und *Dispose*.

Die beiden Methoden *Close* und *Dispose* können explizit aufgerufen werden. Dies kann auch zu Lebzeiten des Objekts geschehen. Darum kann ein Objekt auch nach *Close* und/oder *Dispose* wiederbelebt und verwendet werden. Dies ist ein großer Unterschied zur Verwendung von *finalize*.

Tools

Das entsprechende Tool für einen Programmierer, der all dieser Veränderungen und zahllosen Möglichkeiten in .NET Herr werden muss, ist die neue Version des Visual Studio. Visual Studio .NET erhöht dank seiner Integration in die .NET Plattform die Performanz, Zuverlässigkeit und Sicherheit der erstellten Anwendungen.

Rund um .NET gibt es aber auch noch eine Vielzahl an zusätzlichen Tools, die zu Konfigurations- und Verwaltungszwecken eingesetzt werden. Ob zur Administration und Konfiguration des Sicherheitssystems, Verwaltung der auf einem Computer installierten .NET Komponenten oder zum Zweck der »Erforschung« von Servern, auf denen sich Web Services befinden, viele kleine Tools sind in unterschiedlichen Subsystemen – auch des Betriebssystems – eingebunden oder als eigenständige Anwendungen aufrufbar.

.NET – Das bessere COM?

Bisherige Objektmodelle, wie z.B. das *Component Object Model* (*COM*), die *Common Object Request Broker Architecture* (*CORBA*) oder *Java RMI* (Remote

Methode Invocation) funktionieren nicht (wirklich) übers Internet. Sie setzen eine sehr enge Verbindung zwischen Client und Server voraus. Außerdem sind sie von einer hohen Ausfallsicherheit der Verbindung abhängig. Eine weitere Vorbedingung ist eine homogene Infrastruktur, in Bezug auf das Betriebssystem, die Programmiersprache und das Objektmodell.

Nun, bei einer Kommunikation übers Internet kann beruhigt von einer Nicht(!)-Erfüllung aller genannten Punkte ausgegangen werden. Die Distanz zwischen Client und Server kann sehr groß werden (in irdischen Maßstäben gemessen), und die Störungen der Verbindung sind sehr stark. Es gibt auch keine Garantie, dass ein Client das gleiche Objektmodell, die gleiche Programmiersprache oder auch nur das gleiche Betriebssystem wie der Server verwendet.

Das ist aber noch nicht alles: Werden in einer Anwendung Komponenten auf einem entfernten System eingesetzt, muss die Implementierung auf der einen Seite kompatibel zu der auf der anderen Seite sein. Wird auf einer der beiden Seiten der Code geändert, und zwar so, dass er inkompatibel zur vorhergehenden Version wird, wird der Code auf der anderen Seite nicht mehr stabil oder evtl. gar nicht mehr laufen.

COM

Das *Component Object Model* (*COM*) genießt eine umfangreiche Unterstützung auf der Windows Plattform. Auf anderen Betriebssystemen wie z. B. UNIX ist der Gebrauch von COM – wenn überhaupt – nur eingeschränkt möglich.

Ein anderer Nachteil von COM ist, das es keinen Standard für Typeninformationen gibt. Und da ein Standard nicht vorhanden ist, gibt es auch keine Standard-Methode, um Typen zu beschreiben. Immerhin gibt es in COM drei verschiedene Wege, Typeninformationen anzulegen: in *IDL* (*Interface Definition Language*), Typbibliotheken und /Oicf-Strings, und es ist durchaus möglich, in einem Format Informationen unterzubringen, auf die von einem anderen Format aus nicht zugegriffen werden kann.

CORBA

Die *Common Object Request Broker Architecture* (*CORBA*) wurde entworfen, das Problem der Kommunikation zwischen verschiedenen Computern zu lösen. Die verbundenen Maschinen bzw. Anwendungen sollen unabhängig vom Betriebssystem und der verwendeten Programmiersprache miteinander interagieren können. Aber CORBA ist kein Objektmodell, CORBA ermöglicht vielmehr einem Client den Aufruf von Methoden auf einem angebundenen Server (remote procedure calls).

Es gibt den Entwurf eines Komponentenmodells in CORBA (*CORBA Component Model – CCM*). Doch CORBA ist kein Produkt, sondern eine Spezifikation, die durch einen Anbieter implementiert werden muss. Gleiches gilt auch für das CCM. Leider hat sich bisher noch niemand dieses Problems (ein CCM für CORBA zu implementieren) angenommen.

Ein großer Nachteil entsteht dadurch, dass CORBA eben »nur« eine Spezifikation ist. Wenn Sie eine Anwendung mit CORBA schreiben und dabei eine Implementierung verwenden, die von einem bestimmten Anbieter entwickelt wurde, ketten Sie sich womöglich an diesen Anbieter. In der Regel werden dabei nämlich Anbieter-spezifische Leistungsmerkmale verwendet, womit der Code nicht mehr portabel ist.

Java RMI und EJB

Mit *Java RMI* (*Remote Method Invocation*) und *Enterprise JavaBeans* (*EJB*) arbeiten Sie mit einer (1) Sprache: JAVA.

Also – warum .NET?

Die .NET Plattform – der Bezug ist hier im Wesentlichen auf die Common Language Runtime – ist die konsequente Weiterentwicklung und Verschmelzung vorhandener, weit verbreiteter, guter (!), aber vielleicht auch gealterter Programmiermodelle. Einige Vorteile der CLR sind:

- Der Begriff *Komponente* wird mittlerweile für viele Dinge verwendet, wie z.B. DLLs, Module oder gar einzelne Objekte. In .NET wird die Komponente zum *Assembly*.[3] Ein Assembly ist eine Typmenge und bildet eine physikalische Einheit von einer oder (in der Regel) mehreren Dateien. Assemblies beinhalten auch die Metadaten über das Assembly selbst und alle darin enthaltenen Dateien. Zu den Metadaten gehören Informationen über Versionen, Typreferenzen oder sicherheitsrelevante Daten. Zum Beispiel kann ein 128 Byte langer Schlüssel zur eindeutigen Kennzeichnung der Herkunft eines Assemblies enthalten sein.
- Der Aufbau der Objekte im Speicher ist nicht bekannt und muss auch nicht bekannt sein. Alle Typen sind genau beschrieben, aber über die Position und Größe in Speicher muss sich kein Programmierer den Kopf zerbrechen.
- Alles ist ein Objekt, und alles stammt (direkt oder indirekt) von *System.Objekt* ab.
- XML und SOAP sind sehr wichtige Formate in der Computerindustrie der heutigen Zeit. .NET und die .NET Web Services basieren auf XML und SOAP; sie bieten nicht nur Unterstützung, vielmehr leben sie von und durch XML. Die genannten Vorzüge und Eigenschaften von XML/SOAP übertragen sich auf .NET und bieten die Grundlage für eine völlig neue Art der Softwareentwicklung – übers Internet!

.NET und das Laufzeitsystem sind entworfen worden, um die vorhandenen Entwicklungsumgebungen und Lösungen auf diesem Gebiet zu verbessern und zu erweitern. Dies wird dadurch erreicht, dass .NET verschiedene Programmiermodelle in sich vereint. Diese Vereinigung (und Vereinfachung) macht eine Verwen-

3 In deutschsprachigen Veröffentlichungen ist häufig der Begriff *Baugruppe* zu finden.

dung eines bestimmten Programmiermodells möglich, wobei der Zugriff durch ein einziges, konsistentes *API* (*Application Programming Interface*) erfolgt, ohne jegliche (programmier-)sprachliche Einschränkungen.

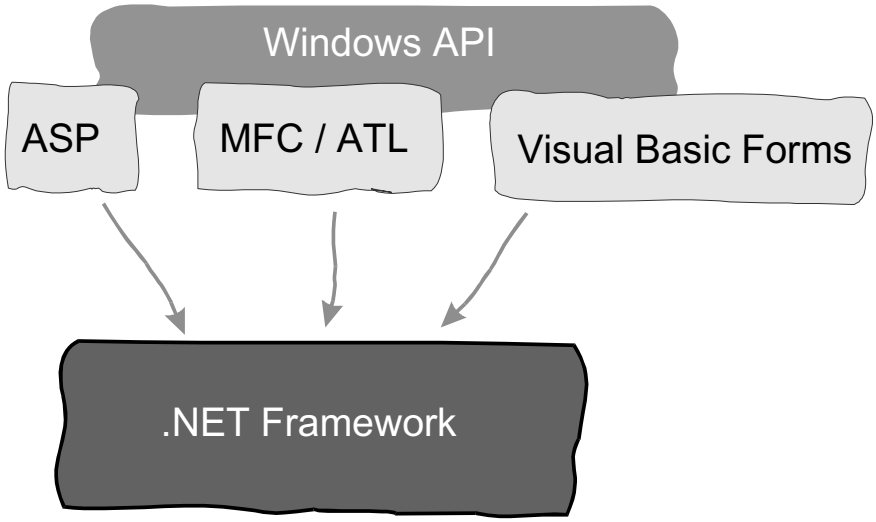

Abbildung 1.12: .NET – die Vereinigung verschiedener Programmiermodelle

1.2.4 Internetbasierte Programmierung

In den vorhergehenden Abschnitten wurden die Eigenschaften und Vorzüge von .NET und dem .NET Framework beschrieben. Die gesamte .NET Plattform wurde konzipiert und entwickelt, um eine neue Generation von Software und der Softwareentwicklung zu ermöglichen. Mit .NET können Anwendungen übers Internet miteinander kommunizieren, interagieren und Daten austauschen.

Wenn heute über Anwendungen im Bereich des Internets geredet wird, sind im Wesentlichen Web Seiten, die in einem Web Browser aufgerufen werden können, gemeint. Ein Besucher *kommuniziert* mit diesen Seiten, indem er Informationen liest und/oder Daten in Formulare einträgt. Diese Formulare werden dann zu einem Browser gesendet, auf dem anschließend in irgendeiner Art und Weise eine Verarbeitung der Daten erfolgt.

Soll eine Anwendung aber ohne die Interaktion mit dem Menschen mit einer anderen Anwendung zusammen arbeiten, wird von verteilten Applikationen geredet. Damit diese Anwendungen funktionieren, und um sie programmieren zu können, wird ein entsprechendes *verteiltes Objektmodell* (*distributed object model*) benötigt oder eine Architektur, die Aufrufe zu entfernten Systemen ermöglicht (*remoting architecture*). Bis hierhin noch kein Problem.

Doch die bekannten, aktuellen Modelle dieser Art wurden entwickelt, als noch von *Local Area Networks* (*LAN*) und nicht vom Internet gesprochen wurde, und sie setzen eine entsprechend enge und sichere (dauerhafte) Verbindung voraus. Außerdem verlangt es die heutigen Anwendungen (und deren Programmierer) nach Komponenten, die bestimmte Aufgaben übernehmen können und wiederverwendbar sind. Dazu ist ein Komponentenmodell notwendig.

Zusammen mit ASP .NET sind es speziell die Web Services in .NET die das internetbasierte Programmieren ermöglichen.

Web Services – Dienste im Internet

Web Services ermöglichen ein Modell, in dem verschiedene Anwendungen übers Internet miteinander verknüpft werden können. Dabei basiert dieses Modell auf vorhandener Infrastruktur und Applikationen, was bedeutet, dass dieses Modell unterschiedliche, offene Standards verwendet und sehr einfach ist. Stellen Sie sich Web Services als schwarze Kisten (oder gelbe) vor, die verschiedene Funktionalitäten bieten, die von Ihnen bzw. einem Programm genutzt werden können, ohne dass Sie Kenntnis über die Implementierung haben müssen. Eine recht gute Analogie wäre z.B. ein Briefkasten (daher die gelben Kisten), in den Sie einen Brief hineinwerfen, mit der Erwartung, dass er zum Empfänger transportiert wird, ohne dass Sie sich um den Transport kümmern müssen oder wissen müssen, wie der Transport erfolgt. Es ist auch egal, in welcher Sprache Sie Ihren Brief schreiben. Wichtig ist nur, dass »der Briefträger« die Adresse lesen kann und versteht!

Ein Web Service kann durchaus wiederverwendet werden – genau das ist seine Bestimmung. Webdienste liegen zwar nicht in binärer Form vor,[4] es können aber entsprechende binäre Komponenten genutzt werden und gegebenenfalls in darauf aufbauende Komponenten integriert werden.

Einfach ausgedrückt ist ein Web Service eine Ressource, auf die unter Verwendung eines URL zugegriffen werden kann, und die angeforderte Informationen zu einem Client liefern oder eine Modifikation in einem angeschlossenen Datenmodell vornehmen kann.

Nicht ohne Grund tragen diese Dienste den Begriff »Web« in ihrem Namen, denn ihre Funktionalitäten bieten sie übers Internet an, und sie können (und sollen) auch übers Internet genutzt werden. Dies kann durch verschiedene Clients unabhängig vom Betriebssystem und der Programmiersprache (und dem Komponentenmodell) geschehen; so können z.B. Web Browser, andere Web Services oder Web Applikationen die angebotenen Dienste nutzen.

4 sondern entweder als reine Quellcode-Dateien oder in Form von *Intermediate Language*.