Karl-Peter-Obermaier-Schule

Staatliche Berufsschule 1 Passau

Staatliche Fachschulen für Elektrotechnik und Maschinenbautechnik

Das .NET Framework

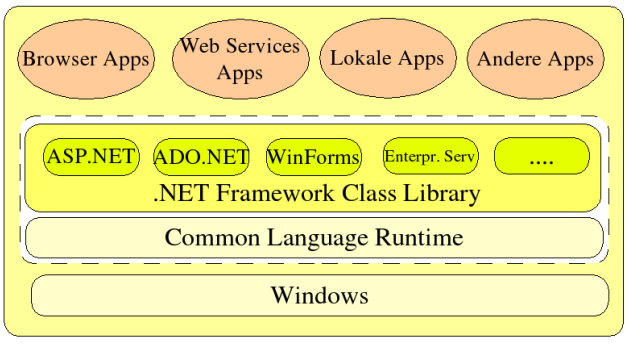
1 Das .NET - Framework

- ***Erläutern Sie die Gründe für die Entwicklung des .NET - Frameworks!***
- ***Nennen Sie Beispiele für Applikationen, die über .NET erstellt werden können!***
- ***Weshalb nutzen .NET Applikationen nicht mehr direkt die Windows API?***
- ***Was versteht man unter .NET Core?***

Die .NET- Initiative von Microsoft© steht für ein ganzes Bündel Technologien. Das .NET-Framework ist dabei zunächst für den Programmierer das zentrale Element. Um die Entwicklung zum .NET- Framework zu verstehen, muss man kurz einen Rückblick auf die vorhergehenden Technologien und dessen zentrale Kritikpunkte werfen:

- Zur Jahrtausendwende existierten verschiedene Programmiersprachen, deren Softwareprodukte nicht unbedingt als kompatibel zu bezeichnen sind,
- das lange genutzte Component Object Model (**COM**) erzielte nicht die angestrebte Versionsunabhängigkeit von Software, sondern führte bei der Installation verschiedener Programme und Programmversionen allzu oft in die so bezeichnete “dll- Hölle”,
- die Verbreitung des World Wide Webs machten die Entwicklung verteilter Applikationen mit Millionen von Benutzern über das Internet notwendig,
- Programmierumgebungen boten zu wenig Unterstützung für neue Technologien wie z.B. XML und wiesen Probleme bei der plattformunabhängigen Umsetzung in inhomogenen Hardwareumgebungen auf.

Es entstand die Idee, eine gemeinsame Grundlage zu schaffen, die von allen Programmiersprachen und über Plattformgrenzen hinweg genutzt werden kann, die Grundidee des .NET- Frameworks. Mit der **Common Language Runtime (CLR)** existiert eine grundlegende Bibliothek, die von allen Anwendungen genutzt werden kann und muss. Weiterhin besteht mit den Klassen des .NET- Frameworks eine Sammlung von Funktionalität, auf die von allen Programmiersprachen aus zugegriffen werden kann. Die wichtigsten Klassen zeigt nebenstehendes Schaubild.



Durch die Nutzung des .NET – Frameworks greifen Applikationen nicht mehr direkt auf die Windows API (Application Programming Interface) zu sondern benutzen die .NET Framework Class Library. Somit besteht eine einheitliche Grundlage zur Programmierung von Applikationen. Module können in unterschiedlichen Programmiersprachen erstellt und trotzdem gemeinsam in größeren Projekten kombiniert und wieder verwendet werden.

Die Verwaltung von Programmen in Assemblies mit genau festgelegten Regeln zur Versionierung ermöglichen so genannte **side by side** Installationen verschiedener Versionen eines einzigen Programms auf demselben Rechner.

Ab 2016 erschien unter Federführung von Microsoft parallel die Open-Source-Plattform .NET Core. Sie ist für die Entwicklung von Apps konzipiert. Dabei stellt .NET Core eine deutliche Modernisierung im Vergleich zum.NET-Framework dar. Es handelt sich um kei-

ne komplette Neuentwicklung, sondern um eine Hybridlösung aus Redesign und Neuimplementierung. Die .NET Core Plattform und das alte Framework wurden bis 2020 parallel entwickelt. Allerdings gab es dabei eine immer deutlichere Verschiebung hin zu .NET Core, nicht zuletzt wegen dessen Vorteilen Geschwindigkeit, Plattformunabhängigkeit, Open Source und dem modularen Aufbau.

Das Nebeneinander von .NET Core und .NET Framework endete im Herbst 2020. Hier flossen die beiden Plattformen sowie Mono zusammen. Genannt wird das Ganze .NET 5.0. Der Zusatz „Core“ entfällt. Für folgende Entwicklungsziele soll eine gemeinsame Basis geschaffen werden:

- Apps für Desktoprechner
- Anwendungen für Webserver
- mobile Apps
- Spiele

Dabei läuft .NET 5.0 auf Windows, macOS, Android, iOS und Linux. Technisch gesehen ist es der direkte Nachfolger von .NET Core. Dies gilt für die Werkzeuge sowie für das offene und agile Entwicklungsmodell, das auf GitHub basiert. Von .NET Standard wird die Klassenbibliothek übernommen. Aus Mono fließt z.B. die Lauffähigkeit auf iOS und Android ein.

2 Die Common Language Runtime

- ***Erklären Sie den Zweck der Common Language Runtime!***
- ***Beschreiben Sie den Weg eines Quellcodes bis zur Ausführung in einem Programm!***
- ***Erläutern Sie den Zweck der MSIL hinsichtlich der Programmiersprachen übergreifenden Entwicklung von Software!***

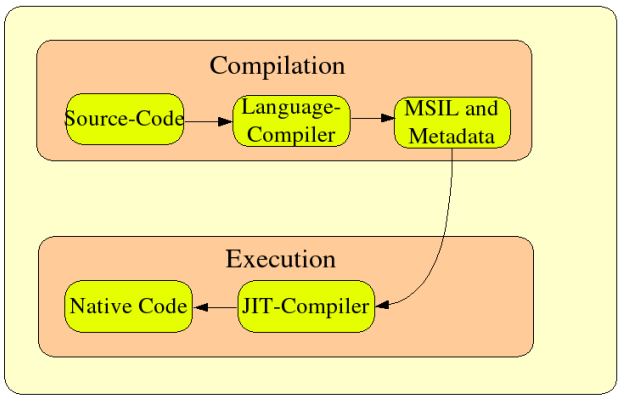
Die **Common Language Runtime (CLR)** stellt die Grundlage des gesamten Frameworks dar. Sie vereint alle Features einer modernen Programmiersprache hinsichtlich ihrer Semantik. Die Syntax bleibt dabei jedoch programmiersprachenspezifisch. In der CLR finden sich Regeln zur Erstellung von Datentypen, Interfaces und Klassen sowie zur Implementierung objektorientierter Konzepte wie der Vererbung. Darüber hinaus stellt sie eine Reihe von Diensten zur Verfügung wie z.B.

- die Garbage Collection zum Speichermanagement von Applikationen,
- ein Standard- Format für Metadaten zur Beschreibung der enthaltenen Komponenten mit Inhalt, Version und Herkunft sowie
- ein allgemeines Schema für die Organisation und Verwaltung von compiliertem Code, genannt **Assembly**.

Indem die CLR nur den Rahmen vorgibt, wird sie von einer Reihe von Programmiersprachen unterstützt, darunter natürlich die Microsoft Haussprachen Visual Basic.NET, C#, Visual C++ und J#. Damit sind auch keine signifikanten Geschwindigkeitsunterschiede bei der Ausführung eines Programms zwischen den Programmiersprachen mehr gegeben.

Wegen der Freigabe der wichtigsten Codeabschnitte und Schnittstellen ist es auch Drittanbietern möglich, eigene Language - Compiler für ihre Programmiersprachen zu

erstellen, was zur Unterstützung des .NET – Frameworks durch zahlreiche Programmiersprachen führt (z.B. COBOL, Pascal, APL, PEARL, Smalltalk,...).



Aber egal in welcher Programmiersprache der Quellcode auch geschrieben ist, er wird beim Compilieren durch den Language - Compiler immer erst in die sog. **Microsoft Intermediate Language (MSIL)** und nicht in einen maschinenspezifischen Binärcode umgewandelt. Darüber hinaus werden noch Metadaten mit hinzugefügt und in der gleichen Datei gespeichert.

Vor der Ausführung wird der MSIL - Code durch den **JIT – Compiler (Just In Time - Compiler)** in nativen Code übersetzt. Dies geschieht zur Optimierung der Ausführungszeit auf Methodenebene. Jede Methode wird beim ersten Gebrauch just in time kompiliert und bleibt übersetzt im Arbeitsspeicher. Bei einem erneuten Aufruf kann diese dann ohne Übersetzungszeit ausgeführt werden.

Durch die Verwendung der MSIL ist es möglich ganze Klassen oder Quellcodeabschnitte einer Programmiersprache in so genannte .netmodules zu kompilieren und in anderen Projekten mit anderen Programmiersprachen zu kombinieren.

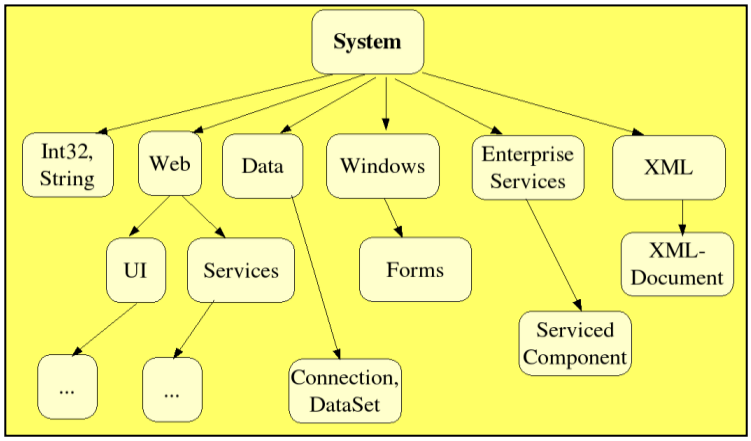
3 Die .NET - Klassenbibliothek

☞ ***Welche Vorteile bieten Klassenbibliotheken?***

☞ ***Erklären Sie den Unterschied zwischen Klassenbezeichnung und Namensraum!***

Die .NET - Klassenbibliothek ist eine Ansammlung von über 2000 Klassen, die das Programmiererleben erleichtern sollen. Sie unterstützen den Programmierer bei fast allen möglichen Anwendungsfällen.

Damit der Nutzer in der Vielzahl der Klassen nicht den Überblick verliert sind alle Klassen der .NET – Klassenbibliothek entsprechend ihrem Verwendungszweck in **Namespaces** (Namensräume) aufgeteilt.



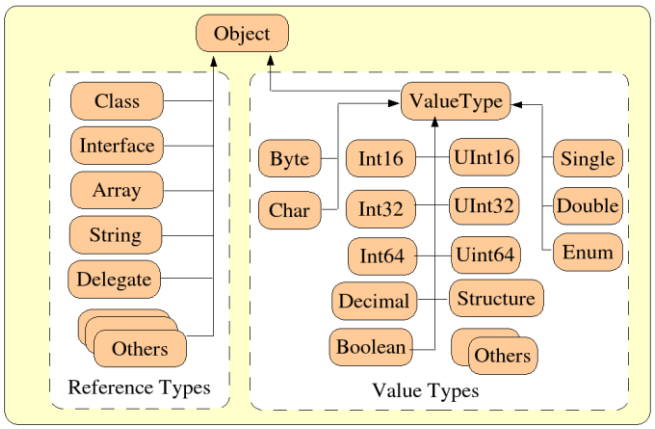
Folgende Abbildung gibt einen kleinen Überblick über die Struktur der Klassen und Namensräume:

Der Zugriff auf vordefinierte Klassen der Library erfolgt über die Angabe oder das Importieren des Namensraums im Programm. Namensräume und Klassenbezeichnung werden dabei durch den Punktoperator verknüpft. Will man beispielsweise die Klasse ComboBox in einer Windows Anwendung nutzen, so erfolgt der Zugriff darauf über die Spezifizierung **System.Windows.Forms.ComboBox**.

4 Das Common Type System

☞ ***Begründen Sie die Einführung des Common Type Systems!***

Die CLR ist der gemeinsame Nenner aller .NET - Programmiersprachen. Doch wie wird diese Gemeinsamkeit definiert. Ein Schlüssel zur Antwort auf diese Frage liegt im so genannten **Common Type System (CTS)**.



Basis aller Klassen bei der Programmierung unter .NET ist automatisch die Klasse Object. Alle anderen Klassen, auch selbst erstellte, sind davon abgeleitet („Everything is an object!“).

Über Vererbung und Polimorphie sind somit Typumwandlungen oder das speichern unterschiedlicher Datentypen in Containern (z.B. ArrayList) möglich.

Jede Programmiersprache, welche unter .NET mit anderen Programmiersprachen kombiniert werden soll muss diese Datentypen unterstützen.

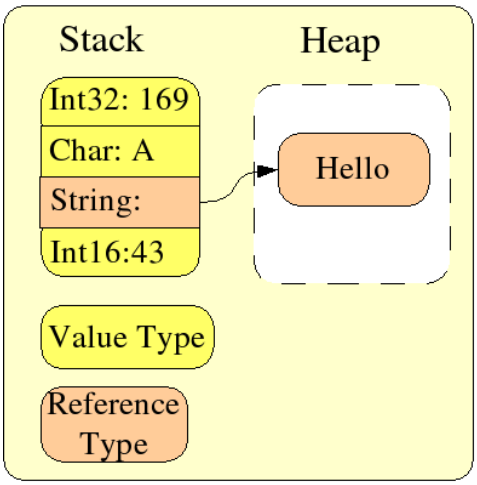
4.1 Refrence Types und Value Types

☞ ***Erklären Sie den Unterschied zwischen Reference- und Value Types!***

☞ ***Worin unterscheidet sich die Speicherverwaltung in Stack und Heap?***

☞ ***Was versteht man in .NET unter Boxing/ Unboxing?***

Hinsichtlich der Verwaltung im Speicher gliedert das CTS alle Datentypen in zwei Kategorien: **Reference Types** und **Value Types**.



Während die Value - Types die einfachen Datentypen wie Ganzzahlen, Buchstaben, Structures usw. darstellen, sind die Reference - Types komplexere Gebilde (Klassen, Interfaces, Arrays, usw.). Der Hauptunterschied zwischen beiden liegt in der Art der Verwaltung im Speicher:

- **Value - Typen** werden immer auf dem **Stack** verwaltet. Der Heap unterliegt der **Garbage - Collection**, während Speicher im Stack verwaltet und automatisch beim Beenden der Methode freigegeben wird.
- **Reference - Typen** werden über den Heap verwaltet. Eine Referenzvariable verweist auf den reservierten Speicher im Heap.

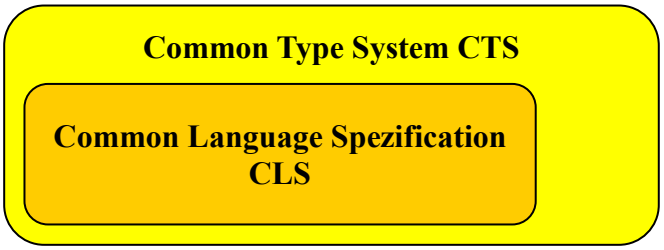
Da alle Datentypen von der Basisklasse Object abgeleitet sind ermöglicht der Mechanismus des Boxings/ Unboxings („in eine Schachtel packen...“) die Umwandlung von Value – Typen in Referenz – Typen und zurück.

4.2 Die Common Language Specification

☞ ***Begründen Sie die Einführung der Common Language Spezification!***

Jede Programmiersprache, die .NET - kompatibel sein will, muss die Anforderungen des CTS erfüllen. Dies ist nicht immer für alle Sprachen sinnvoll. Um dennoch die Kompatibilität und einen Datenaustausch mit anderen Programmiersprachen herstellen zu können, ist mit der **Common Language Specification (CLS)** ein Kompromiss für die Implementierung von Programmen unter .NET geschaffen worden.

Die CLS definiert einen breiten Subset des CTS. So müssen CLS - kompatible Sprachen die meisten Value - Types implementieren, nicht aber jedoch die ganzzahligen Typen UInt16, UInt32 und UInt64.



Die CLS hat noch eine Reihe weiterer Einschränkungen gegenüber dem CTS wie z.B. Konventionen für die Bezeichnung von Variablen, welche jedoch alle ein Ziel verfolgen: die effektive Zusammenarbeit zwischen verschiedenen Programmiersprachen ermöglichen zu können.

5 Das Arbeiten im Managed Environment

Mit der Einführung des .NET Frameworks bestehen nicht nur für den Programmierer neue Möglichkeiten und Konzepte bei der Softwareentwicklung. Zusätzlich zum Erstellen bietet das Framework im Managed Environment die Möglichkeit der einfachen Installation, Pflege, Verwaltung und Rechteadministration der Applikationen. Dazu müssen .NET Programme neben dem ausführbaren Quellcode noch Metadaten über Inhalt, Herkunft und Version beinhalten.

5.1 Das Kompilieren von Managed Code

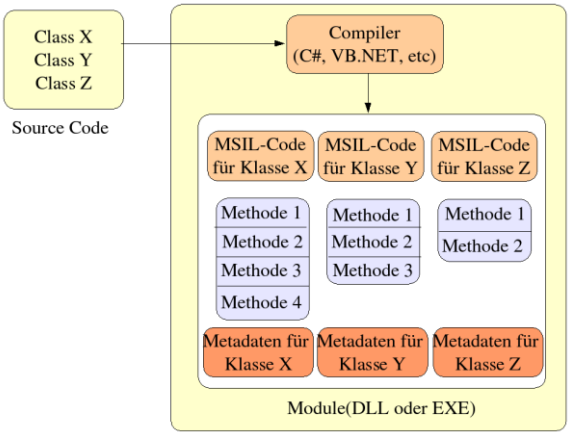
☞ ***Was versteht man unter Metadaten?***

Beim Kompilieren von Managed Code werden einem Assembly neben dem Quellcode in MSIL noch weitere Informationen in Form von Metadaten hinzugefügt.

Metadaten beschreiben unter Anderem die Typen, die in einem Modul enthalten sind. Zu den Informationen gehören z.B.

- der Name des Typs,
- die Sichtbarkeit (public oder auf Assembly - Ebene),
- Vererbungsstrukturen,
- Implementierte Interfaces und Methoden,
- Properties und Events,

Strong named assemblies beinhalten zusätzlich Informationen über Version und



Herkunftsort der Applikation, welche zur Verwaltung von Programmversionen und Sicherheitseinstellungen genutzt werden können

5.2 Das Verwalten von Managed Code (Assemblies)

☞ ***Beschreiben Sie die drei Arten von Assemblies!***

☞ ***Welche Informationen sind in einer Assembly gespeichert?***

Eine Applikation besteht häufig aus vielen Dateien wie zum Beispiel DLLs, EXE - Dateien, Ressource - Dateien etc. Im .NET - Framework werden alle Dateien, die eine logische Einheit bilden, in einer so genannten **Assembly** gruppiert.

Die Installation einer .NET - Applikation bedeutet deshalb vereinfacht dargestellt lediglich das Kopieren der Assemblies auf die Zielplattform. Sie müssen nicht in einer Registry oder Ähnlichem registriert werden.

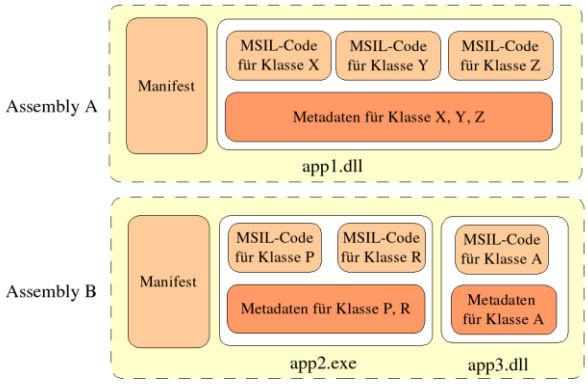
Bei den Assemblies unterscheidet man drei Arten:

- **private assembly**: wird nur von einer Applikation genutzt und liegt in der Regel im gleichen Ordner wie die Ausführbare *.exe Datei der Applikation.
- **strong named assembly**: besitzt einen strong name (eindeutige Kennzeichnung) mit Version und Herkunftsinformationen.
- **shared assembly**: muss ein strong named assembly sein und wird von mehreren Applikationen genutzt. Shared assemblies werden im so genannten **GAC**, dem **Global Assembly Cache** installiert.

Um einen Überblick zu erhalten, welche Dateien in einer Assembly organisiert sind ist das darin enthaltene **Manifest** zu untersuchen:

Ein Manifest enthält eine Reihe von Informationen zur Assembly. Die wichtigsten wären:

- Der Name, evtl. den sog. **Strong Name**
- Die Versions-Nummer der Assembly
- Die unterstützten Sprachen
- Eine Liste aller Dateien der Assembly, incl. deren HASH - Werte.
- Die Abhängigkeiten von anderen Assemblies (incl. deren Versionsnummer)



Eine **strong named assembly** definiert in ihrem Dateinamen nicht nur den Namen, sondern auch noch die Landeseinstellungen, die Versionsnummer und eine digitale Signatur (HASH). Strong named assemblies vermeiden damit die berüchtigte „*DLL – Hell*“, weil nun mehrere Versionen einer Assembly gleichzeitig auf einer Maschine existieren können (side by side installation).

5.3 Ausführen von Managed Code

☞ Beschreiben Sie den Suchprozess (Probing) beim Laden einer Assembly!

Wenn eine .NET - Applikation ausgeführt wird, müssen die Assemblies der Applikation gefunden und geladen werden. Geladen werden dabei nur die Assemblies, die auch wirklich zum Einsatz kommen. Werden Methoden nicht aufgerufen, dann müssen die sie beherbergenden Assemblies auch nicht geladen werden.

Aber wie werden nun die Assemblies gefunden? Die CLR verfolgt dabei überblicksartig folgende Strategie:

- Feststellen, ob die benötigte Assembly schon geladen ist. Wenn nicht, dann
- Suche im sog. **Global Assembly Cache (GAC)**. Beim GAC handelt es sich um ein spezielles Verzeichnis im Dateisystem. Es nimmt nur strong named assemblies auf. Findet Sie die benötigte Assembly nicht, dann
- Suche nach einem sog. **Codebase** Element in einer Konfigurationsdatei der Applikation (Applicationconfiguration – File). Wenn kein Codebase - Element-Eintrag gefunden wird, dann
- Suche in der sog. **Application Base** , sozusagen das root - Directory der Anwendung.

Der Suchprozess (**Probing**) kann evtl. weit komplizierter sein als hier dargestellt.