Seminararbeit

Die Architektur der Common Language Runtime

Gertraud Orthofer (0056324)

Inhaltsverzeichnis

1 Common Language Runtime (CLR)

Ein wesentlicher Bestandteil des .NET-Frameworks ist die Common Language Runtime (CLR). Sie steht direkt in Verbindung mit dem Betriebssystem und ist die Laufzeitumgebung. Eine Besonderheit der CLR ist, dass sie mit verschiedenen Sprachen arbeitet. Dies ist möglich, weil sich alle .NET Sprachen an die Common Language Specification (CLS) und an das Common Typ System (CTS) halten.

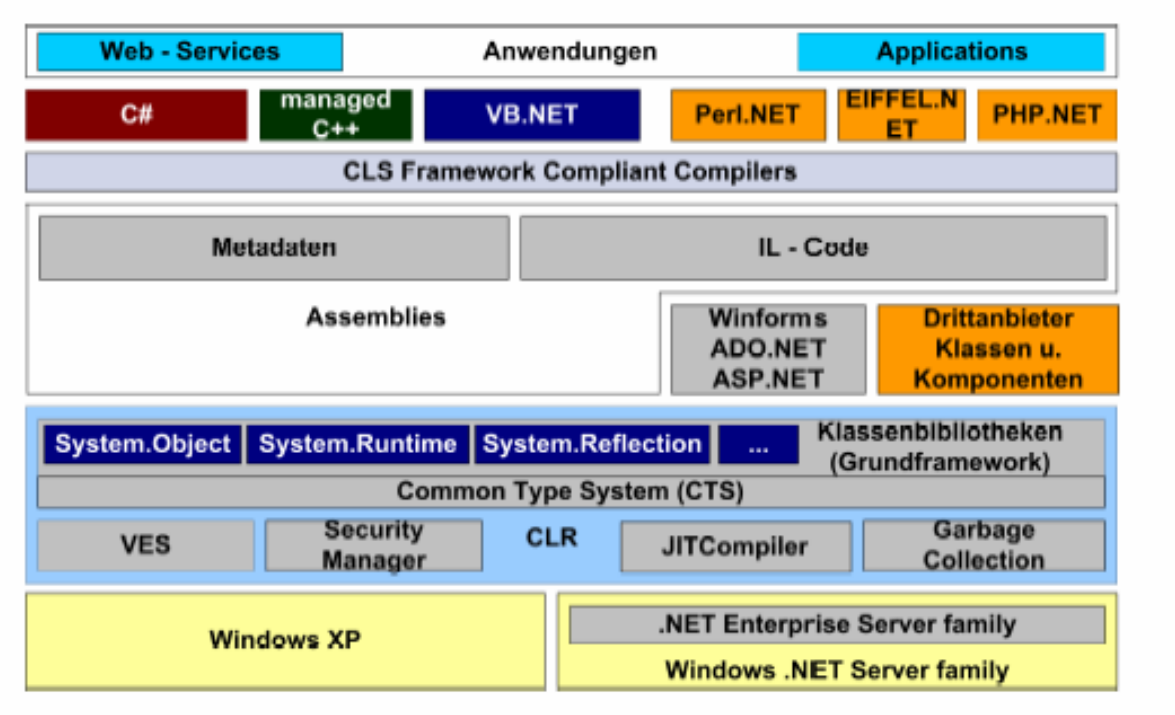


Abbildung 1: .Net-Framework

CLR arbeitet mit verwalteten Modulen (managed Moduls). Damit die CLR mit Programmen aus verschiedenen Sprachen arbeiten kann, bedient man sich einer Zwischensprache, vergleichbar mit dem Java Bytecode. Im .NET spricht man von der Common Intermediate Language (CIL). Jede Sprache des .NET Frameworks besitzt einen eigenen Compiler. Aber diese Compiler erzeugen nicht nur den CIL Code, sondern ein verwaltetes Modul (managed module), das aus dem CIL Code und Metadaten besteht.

Dies bittet einige Vorteile und zwar:

Typsicherheit

Ein Typ darf nur Funktionen ausführen, welche für ihn bestimmt sind. Weiters werden casts nicht erlaubt, deren Typen nicht kompatibel sind. Dies führt ansonsten zu einer InvalidCastException. Auch kommt es zur Überprüfung, ob nicht außerhalb des für den Objekt vorgesehenen Speicherbereichs zugegriffen wird. Dies kann beim Arbeiten mit Arrays der Fall sein.

Sicherheit

Typsicherheit spielt eine bedeutende Rolle bzgl. sicherer Applikation, aber damit ist der Punkt Sicherheit noch nicht abgedeckt. .Net unterstützt zwei Arten von Sicherheitskonzepten und zwar:

rollenbasierte Sicherheit: Diese Art von Sicherheit wird von Betriebsystemen unterstützt, welche den Benutzern eine Rolle und mit nur einer bestimmten Menge an Rechten zuteilen.
codebasierte Sicherheit: Anders als bei der rollenbasierten Sicherheit werden hier den Assemblies Rechte zugeordnet. Welche Rechte einem Assembly zugeordnet werden, hängt von der Information ab, welche die Assemblies selbst enthalten.

Unterstützung verschiedener Sprachen

Wie bereits erwähnt, arbeitet die CLR mit CIL-Code und ihr ist es egal in welcher Sprache der Code ursprünglich geschrieben wird. Von der CLR wird grundsätzlich jede Sprache unterstützt, welche Compiler anbieten, die CIL-Code produzieren und sich dabei an die CLS und das CTS halten.

Performanz

Auf den ersten Blick scheint der Begriff Performanz mit managed Code im Widerspruch zu stehen, denn es sind schließlich zwei Compilervorgänge erforderlich bis Maschinencode entsteht. Erwähnenswert ist, dass die CLR keinen Interpeter verwendet, sondern im Gegensatz zur JVM direkt übersetzt. Somit dauert die erste Ausführung etwas länger, aber beim wiederholten Ausführen ist der Maschinencode bereits vorhanden und kann direkt verwendet werden. Weiters erfolgt die Übersetzung von CIL in Maschinencode Just-In-Time, d.h. es werden nur jene Teile übersetzt, die auch tatsächlich verwendet werden.

DLL-Hölle (DLL-hell)

Von der DLL-Hölle spricht man, wenn ein neues Programm mit einer gleichnamigen .dll Datei eine bereits existierende ersetzt. Dies kann dazu führen, dass sich Programme nicht mehr ausführen lassen. Um dies zu vermeiden, wird jetzt eine .dll nicht nur durch den Namen identifiziert, sondern zusätzlich durch die Versionsnummer. Das heißt, es können mehrer .dll-Dateien mit dem gleichen Namen existieren, jedoch unterscheiden sich diese in ihren Versionsnummern. Jedes Programm kann somit, mit der richtigen .dll Datei in der richtigen Version arbeiten.

Metadaten

Ein Modul ist die kleinste physische Einheit, während ein Assembly die kleinste logische Einheit ist und mehrere Module enthalten kann. Jedes Modul enthält nicht nur den CIL-Code sondern auch Metadaten. Metadaten enthalten Informationen darüber, welche Typen, Felder, Methoden im Modul definiert sind. Somit kann die CLR leicht viel Informationen über das Modul erhalten ohne zuerst den CIL-Code betrachten zu müssen.

Garbage Collection

Vorteilhaft ist, dass die CLR die Speicherbereinigung übernimmt. Der Benutzer muss sich nicht mehr um die Allokation bzw. Deallokation des Speichers kümmern. Weiters werden unerlaubte Speicherzugriffe vermieden, die zu unangenehmen Problemen führen können.

2 Common Type System (CTS)

Das Common Type System (CTS) legt den Grundstein für die Sprachinteroperabilität zwischen den verschiedenen Programmiersprachen in .NET. Der Compiler für die jeweilige Programmiersprache unter .NET übernimmt das Abbilden der spracheigenen Typen auf die des CTSs, so dass die Verwendung für Entwickler völlig transparent ist.

In .NET stammen alle Typen von einer Wurzel, dem System.Object, ab. Alle Typen die man über das CTS definiert und damit von System.Object ableitet, sind Objekte und garantieren ein minimales gleiches Verhalten. Man unterscheidet in .NET zwei Arten von Typen, die Referenztypen, und die Wertetypen.

2.1 Vergleich des Werttyps mit dem Referenztyp:

Speicherort

Ein grundsätzlicher Unterschied zwischen diesen zwei Kategorien liegt darin, wo sie gespeichert werden. .NET verwendet zwei verschiedene physikalische Speicherblöcke um Daten zu speichern und zwar den Stack und den Heap. Werttypen werden am Stack nach dem Prinzip Last-In-First-Out (LIFO) gespeichert. Referenztypen werden am Heap gespeichert, aber der Verweis zur Adresse am Heap wird am Stack abgelegt.

Garbage Collection

Der Heap verwendet einen Garbage Collector, welcher nicht mehr referenzierte Objekte entfernt. Die verbleibenden Objekte werden so angeordnet, dass am Anfang des Speicherbereichs die belegten und im Anschluss die freien Speicherzellen sind. Damit wird vermieden, dass ein Objekt auf mehrere Speicherbereiche aufgeteilt werden muss. Wird ein neues Objekt am Stack eingefügt und gibt es dafür nicht mehr ausreichend Platz, so tritt der Garbage Collector in Aktion und räumt auf.

Vergleich von Typen

Wenn zwei Werttypen miteinander verglichen werden, wird deren Inhalt Bit für Bit verglichen. Anders sieht es bei den Referenztypen aus, denn bei einem Vergleich wird überprüft, ob sie auf die gleiche Adresse verweisen (sprich: gleiches Objekt). Äquivalent verhält es sich mit der Zuweisung, denn bei den Referenztypen spricht man vom passing by reference. Das heißt, der Zeiger zeigt auf die gleiche Speicherzelle im Speicher wie die des Zuweisers. Es wird somit nur die Adresse kopiert, während bei den Werttypen der Inhalt kopiert wird.

Initialsierung

Werttypen werden einfach mit einem entsprechenden Nullwert initialisiert, während Referenztypen mit „null“ initialisiert werden. Damit wird vermieden, dass auf keinen falschen Speicherplatz zugegriffen wird.

Vererbung

Wie bereits erwähnt, erbt jeder Typ von System.Object. Die Referenztypen erben direkt von System.Object im Gegensatz zu den Werttypen. Diese erben direkt von System.Value. Während Werttypen nicht als Superklasse fungieren können, ist dies bei Referenztypen möglich.

2.2 Beschreibung von Typen

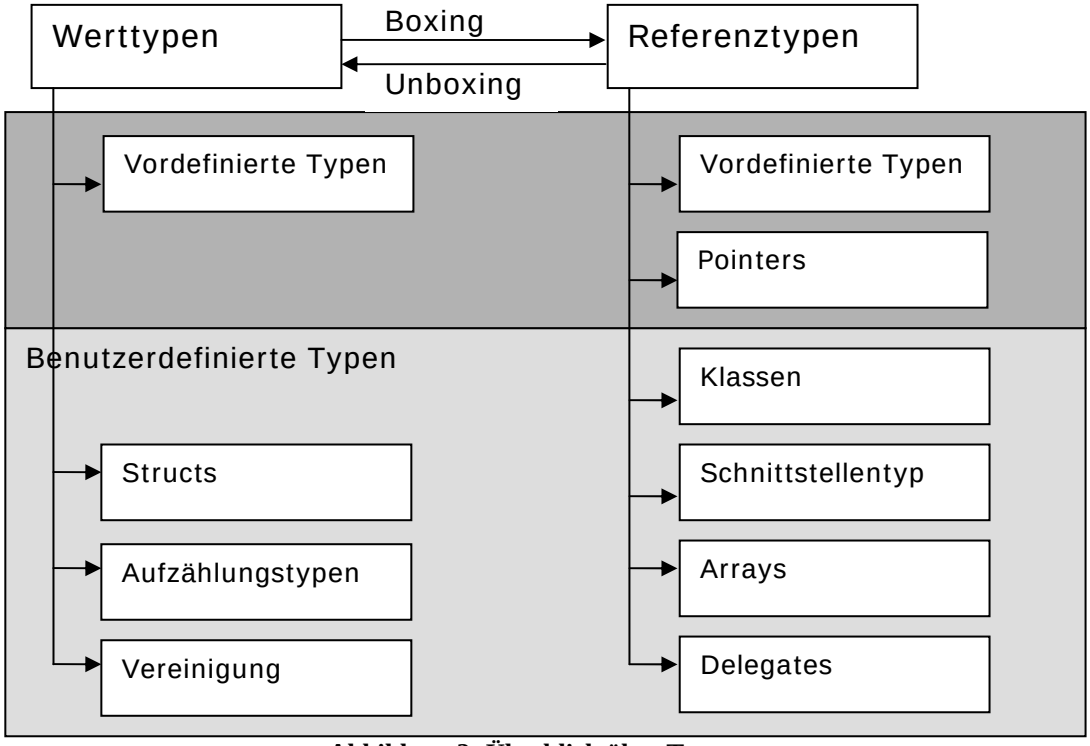


Abbildung 2: Überblick über Typen

2.2.1 Vordefinierte Typen

Die CLR unterstützt eine Reihe von vordefinierten Datentypen und zu den vordefinierten Werttypen gehören:

- Bool (1 Byte; 0 = false; 255 = true)
- Zeichen (16-Bit Unicode)
- Zwei-Komplement-Ganzzahlen (8,16, 32 und 64 Bit)
- Vorzeichenlose Ganzzahlen (8,16, 32 und 64 Bit)
- Gleitkommazahlen (32 und 64 Bit)
- Zahlen von maschinenabhängiger Größe

Weiters gibt es auch vordefinierte Referenztypen und zu diesen gehören die Zeichenketten (string) und die Objekte (object).

Weiters bietet das CTS auch verschiedene Typen (verwaltete, unverwaltete) von Zeigern, welche den vordefinierten Referenztypen angehören.

- *Verwaltete Zeiger:* Der Unterschied zu den Objektzeigern ist jener, dass sie nicht nur auf den Beginn eines Objekts verweisen können, sondern auf ein Feld eines Objekts bzw. auf ein Element eines Arrays.
- *Unverwaltete Zeiger:* Dies sind vorzeichenlose Integerwerte, die damit nicht die Zahl sondern die Adresse im Speicher meinen. Weiters werden sie von der Speicherverwaltung nicht kontrolliert, daher kommt auch der Name unverwalteter Zeiger. Es gibt in .Net eine Einschränkung für die unverwalteten Zeiger und zwar dass sie nicht auf Objekte am Heap verweisen dürfen, denn sonst würde der Garbage Collector in seiner Arbeit sehr eingeschränkt sein.

2.2.2 Benutzerdefinierte Werttypen

Im CTS gibt es die Möglichkeit selber Werttypen zu definieren und dies ist besonders sinnvoll wenn:

- sie sich wie primitive Typen verhalten
- sie von keinen anderen Typen (außer System.Value) erben wollen und wenn sie von keinen weiteren Typ geerbt werden sollten
- sie hauptsächlich als Parameter für Methoden verwendet werden
- sie meistens nur als Rückgabewerte verwendet werden

1. Structs

Bei den Strukturen handelt es sich um abgespeckte Klassentypen. Sie können Inferfaces implementieren, jedoch dürfen sie von keinem anderen Typ erben als von System.Value und können nicht von anderen Typen geerbt werden.

Structs dürfen definieren:

- Konstruktoren, aber keine parameterlosen
- Methoden
- Felder
- Properties
- Events.

Beispiel:

```
class (auto|sequential) sealed Person extends System.ValueType{
        .field string name
        .field int32 age
}
```

Das Attribut sealed bedeutet, dass der Typ nicht weitervererbt werden kann. Während die beiden anderen Attribute (auto, sequential) angeben, in welcher Reihenfolge die Felder gespeichert werden. Bei auto ist die Reihenfolge egal, im Gegensatz dazu, werden bei der Attributangabe sequential die Felder in der angegebenen Reihenfolge am heap angelegt.

2. Aufzählungstypen

Bei Aufzählungen handelt es sich um ein praktisches Programmierkonstrukt, mit dem sie Name-Wert-Paare unter einem bestimmten Namen gruppieren können, anstatt eine Reihe von Möglichkeiten mit aussagelosen numerischen Werten zu deklarieren.

Beispiel:

```
.class sealed Color extends System.Enum{
        .field static literal valuetype Color red = int32 (0x00000000)
        .field static literal valuetype Color green = int32 (0x00000001)
        .field static literal valuetype Color blue = int32 (0x00000002)
}
```

Das CTS fordert, dass Aufzählungstypen von der gemeinsamen Basisklasse System.Enum stammen. Das Attribut sealed im Beispiel deutet wieder darauf hin, dass Aufzählungstypen nicht von anderen Typen geerbt werden können.

3. Vereingigung

Bei der Vereinigung wird das Speicherlayout festgelegt, welches von der CLR nicht mehr verändert werden kann (Attribut explicit). Die Verwendung des Attributes explicit fordert, dass beim Anlegen von Feldern die Feldpositionen angegeben werden müssen. Es ist auch möglich Speicherbereiche mehrmals zu verwenden (siehe Beispiel). Wie die vorher genannten benutzerdefinierten Typen kann auch dieser Typ nicht von anderen Typen geerbt werden.

Beispiel:

```
.class explicit sealed IntFloat extends System.ValueType{
.field [0] float32 f
.field [0] int32 i
}
```

2.2.3 Benutzerdefinierte Referenztypen

Zu diesen Referenztypen zählen Klassen, Interfaces, Arrays und Delegates.

1. Klassen

Klassen sind der Grundbaustein für die objektorientierte Programmierung. Klassen können erben, jedoch ist eine Mehrfachvererbung nicht erlaubt.

Klassen dürfen definieren:

- Felder
- Methoden
- Properties (readonly, write only)
- Ereignisse

Somit werden sie auch als selbstbeschreibender Typ bezeichnet.

Beispiel:

```
.class Bar extends System.Object {
        .method instance void .ctor(){          // Konstruktor
                Ldarg.0
                call instance void System.Object::.ctor()
                ret
        }
        .method static void foo() {…}
        .method instance virtual void goo() {…}
        .method instance void hoo() {…}
}
```

Jede Klasse muss einen Konstruktor enthalten, ist im Quellcode keiner enthalten, dann wird im CIL-Code ein default-Konstrukor erzeugt.

Methoden:
Die Deklaration von Methoden kann folgende Attribute enthalten, static, instance und virtual. Static – wie bereits der Name aussagt - deutet darauf hin, dass es sich um eine statische Methode handelt. Während das Attribute instance für nicht statische Methoden verwendet wird. Soll es möglich sein, dass eine Unterklasse Methoden überschreibt, so muss das Attribut virtual verwendet werden.

Beim Methodenaufruf unterstützt das CTS sowohl die statische als auch die dynamische Bindung. Bei der statischen Bindung werden Methoden mit call und andernfalls mit callvirt aufgerufen.

Der Aufruf der Methoden des Beispiels würde wie folgt aussehen:

Statische Bindung	Dynamische Bindung
call void Bar::foo()	--- gibt es nicht ----
call instance void Bar::goo() (Attribute virtual fällt weg)	**callvirt** instance void Bar::goo()
call instance void Bar:: hoo()	**callvirt** instance void Bar::hoo()

2. Schnittstellentypen

Bei Schnittstellentypen handelt es sich um eine Sammlung abstrakter Methoden, Eigenschaften und Ereignisdefinitionen. Typen, welche diese Schnittstellentypen implementieren, werden Teile ihres Verhalten bestimmt. Schnittstellentypen können niemals instanziert werden. CTS erlaubt eine mehrfache Schnittstellen-vererbung.

Interfaces dürfen

definieren	nicht definieren
Klassenfelder	Objektfelder
Properties	nicht überschreibbare Objektmethoden
statische Methoden	innere Klassen
virtuelle Objektmethoden	Werttypen

Interfaces müssen bei der Deklaration die zwei Attribute interface und abstract enthalten. Bei der Deklaration von Methoden ist zu beachten, dass alle die Attribute public und abstract enthalten müssen. Dadurch wird gefordert, dass diese Methoden von der implementierenden Klasse überschrieben werden. Möchte eine Klasse ein bzw. mehrere Interface implementieren, erfolgt dies mit dem Aufruf: implements Interfacename

Beispiel:

```
.class interface abstract ILockable {
        .method public abstract instance virtual void Lock() {}
}
```

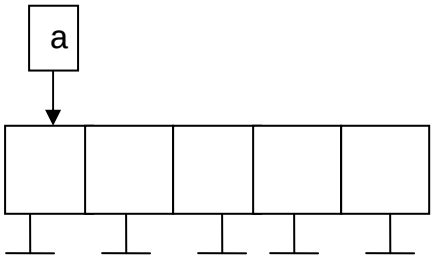
3. Arrays

Klassen enthalten Felder mit verschiedenen Typen und diese werden über die Feldnamen angesprochen. Anders verhält es sich bei den Arrays. Alle Objekte im Array sind vom gleichen Typ, der bei der Deklaration festgelegt wird. Wird auf ein Objekt im Array zugegriffen kann dies nur mit Indizes erfolgen. Für die Festlegung der Größe der Arrays, sowie für die Indizierung werden signed integers verwendet. In Arrays können nur Objekte gespeichert werden. Falls Werttyen in Arrays gespeichert werden, müssen sie zuerst zu einem Referenztyp transformiert (boxing) werden.

Es gibt zwei Arten von Arrays:

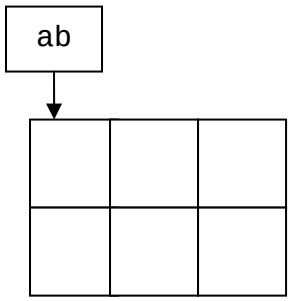
- *Vektoren*
  Hier handelt es sich um eindimensionale Arrays, deren Indizierung mit 0 beginnt. Auch die ausgefransten Arrays gehören zu dieser Gruppe, denn jedes Objekt eines Array kann wieder ein Array enthalten. Die Erzeugung erfolgt durch newarr im CIL-Code.

  Beispiel:
  int [][] a = new int [5][];



- *Arrays als Objekte*
  Dies sind alle jene Arrays, die laut Beschreibung nicht als Vektor betrachtet werden können, Beispiele dafür sind die mehrdimensionalen Arrays. Erzeugt werden diese mit dem Befehl newobj im CIL-Code.

  int [,] ab = new int [2,3]



Beide Arten erben vom abstrakten Typ System.Array.

4. Delegatetypen

Diese Art von Referenztypen sind eine typsichere Variante von Methodenzeigern. Sie wird erreicht, indem der Methodenzeiger in eine verwaltete Klasse gekapselt wird.

Deklaration: delegate int Adder (int a, int b);

Übersetzung der obigen Zeile führt zu folgendem CIL-Code.

```
.class sealed Adder extends System.MulticastDelegate{
        .method instance void .ctor (object receiver, native int method) runtime{}
        .method virtual instance int32 Invoke(int32 a, int32 b) runtive{}
        .method virtual instance class System.IAsyncResult BeginInvoke (int32 a,
int32 b, class System.AsyncCallback acb, object asyncState) runtive{}
        .method. virtual instance int32 EndInvoke (class System.IAsyncResult
result) runtime{}
}
```

Diese Delegatetypen erben von der Klasse System.MulticastDelegate und weiters darf kein anderer Typ von diesem Typ erben (Attribute: sealed) und jede Methode wird mit runtime gekennzeichnet. Die CLR unterstützt BeginInvoke und EndInvoke, damit ist ein Asynchroner Aufruf möglich.

2.3 Boxing und Unboxing

.NET kennt die Wertetypen, die direkt von der Runtime implementiert werden, vor allem aus Performancegründen. Grundsätzlich ist aber in .NET alles ein Objekt. Normalerweise befinden sich die Wertetypen auf dem Stack, aber sie können jederzeit in ein Referenztyp-Objekt konvertiert werden. Dieser Prozess wird als *Boxing* bezeichnet. Der umgekehrte Prozess wird als *Unboxing* bezeichnet.

Beispiel*:*

```
int i = 20;
object box = i;      // boxing


int j = (int)box;     // unboxing
```

Boxing wird durchgeführt, wenn die linke Seite von der Operation ein Objekttyp und die rechte Seite ein Werttyp ist. Weiters enthalten Arrays nur Objekt und wird ein Array mit Zahlen angelegt, so ist Boxing erforderlich. Bei sehr großen Arrays kann es passieren, dass ziemlich viel Performanz für Boxing und Unboxing verbraucht wird.

3 Common Language Specification (CLS)

Bei der Common Language Specification (CLS) handelt es sich um eine Reihe von Richtlinien, die detailliert die minimale und vollständige Menge der Funktionen beschreiben, die ein .NET-fähiger Compiler für die Erzeugung von Code, der von der CLR verarbeitet und gleichzeitig in allen Zielsprachen der .NET-Plattform einheitlich verwendet werden kann, unterstützen muss. Sollte ein Assembly auf CLS Konformität überprüft werden, so kann dies durch die Angabe von „[assembly: CLSCompliant(true)]“ erreicht werden.

Besonders wichtig ist die CLS-Regel 1:

CLS-Regeln betreffen lediglich die Teile eines Typs, die außerhalb des definierenden Assemblys zur Verfügung stehen.

Die nachfolgende Grafik soll verdeutlichen, dass die Programmiersprachen eine Teilmenge des CTS/CLR verwenden, denn das CTS möchte möglichst alle Programmierkonstrukte unterstützen. Jedoch sind die Programmiersprachen eine Obermenge des CLS. Da C# passend zum .NET Framework entwickelt wurde, bietet es fast alle Funktionen der CLR an. Also müsste in der Abbildung der C#-Kreis, der den Funktionsumfang darstellen soll, richtigerweise fast deckungsgleich mit dem äußeren sein!

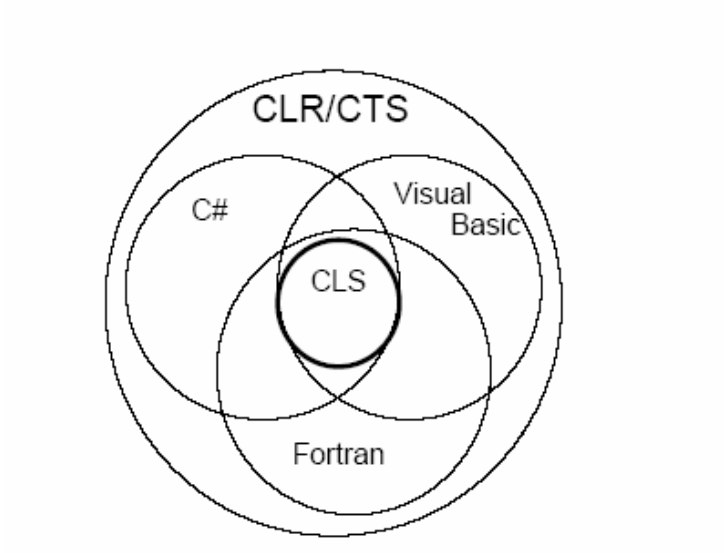


Abbildung 3: Zusammenhang zwischen CLR/CTS und CLS

Während CTS die Typen spezifiziert, welche in einem Programm verwendet werden können, spezifiziert die CLS wie diese Typen verwendet werden dürfen, um mit anderen Sprachen kompatibel zu sein. Daher geht es im folgenden darum, welche Eigenschaften noch erforderlich sind, um sich an die CLS zu halten.

Namensgebung
Namen sind erforderlich um Typen, Werte und Instanzen zu identifizieren und zu unterscheiden. Daher gibt es in CLS einige Richtlinien, die Namenskonflikte vermeiden soll.

Beispiele dafür sind:
- Namen dürfen sich nicht in der Groß- und Kleinschreibung unterscheiden.
- Der gleiche Name darf nicht für eine Methode und einen Feldnamen verwendet werden.

Attribute bzgl. Sichtbarkeit.

Bei der Sichtbarkeit von Typen werden folgende drei Arten unterschieden:

- *exported:* Der deklarierte Typ darf in anderen – nicht nur im deklarierten – Bereich verwendet werden. (In C# würde exported dem Sichtbarketisattribut public entsprechen).
- *not exported:* Ein Verwenden des Typs in anderen Bereichen ist verboten.
- *nested:* Eingebettete Typen dürfen auch die äußeren Typen verwenden.

Mitglieder von Typen können ebenfalls mit folgenden Sichtbarkeitsattributen ausgestattet sein:

- *Compiler-controlled:* diese Art von Typmitglieder dürfen nur vom Compiler verwendet werden.
- *Private:* Ausserhalb des deklarierten Bereichs sind sie nicht zugänglich.
- *Family:* Diese dürfen vom bezeichneten Typ und von den von ihm erbendnen Typen verwendet werden.
- *Assembly:* Sichtbarkeit im gesamten Assembly
- *Family and Assembly:* Sichtbarkeit im gesamten Assembly, sowie vom bezeichneten und dessen erbenden Typen.
- *Family or Assembly:* Sichtbarkeit entweder im Assembly oder im bezeichneten und dessen erbenden Typen.
- *Public:* Jeder darf diese Mitglieder verwenden, denn sie sind überall sichtbar.

4 Virtual Execution System (VES)

Die CLR ist die Laufzeitumgebung und ist verantwortlich für die Ausführung der Programme. Die erste Abbildung hat bereits gezeigt, dass für das Ausführen mehrere Teile (Garbage Collector, JIT-Compiler, Security Manager) eine wichtige Rolle spielen und all diese Komponenten werden unter den Begriff Virtual Execution System zusammengefasst.

Bevor die CLR mit der Ausführung einer .NET-Applikation beginnen kann, muss zuerst die CLR durch die .EXE oder .DLL gestartet werden.

Der Quelltext wird zum Beispiel mit dem C#-Compiler (csc.exe) übersetzt. Der Compiler erzeugt eine EXE-Datei im PE-Standardformat und diese ist ein Assembly (verwaltetes Modul mit Manifest (weitere Metadaten)). Der Compiler importiert dabei immer die Methode _CorExeMain aus der MSCorEE.dll.

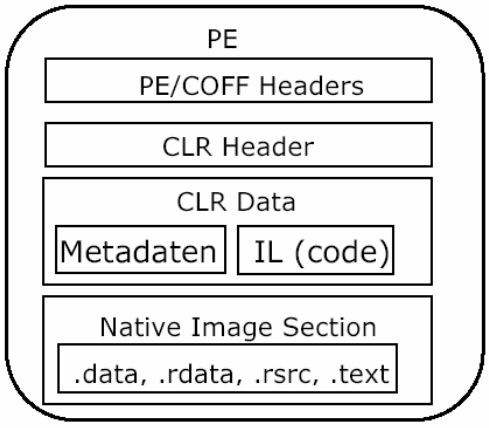
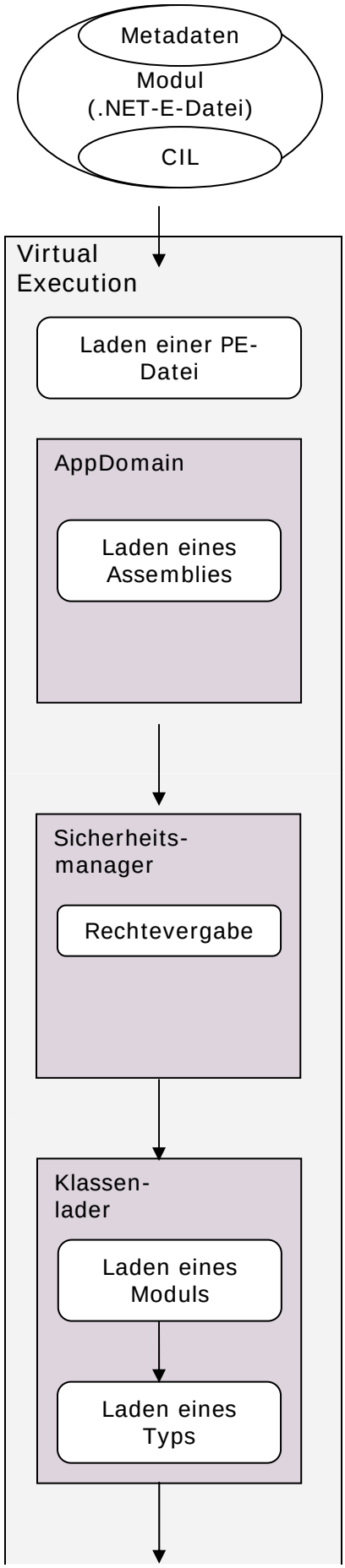


Abbildung 4: PE-Datei

Wenn diese Anwendung ausgeführt wird, behandelt Windows diese Datei wie eine ganz normale .EXE Datei. Der Windowslader lädt die Datei und sucht nach dem Absatz mit dem Namen .idata und erkennt, dass die MSCorEE.dll in den Adressbereich geladen werden muss. Dann sucht der Lader in der MSCorEE.dll nach der Adresse der Funktion _CorExeMain. Anschließend wird die Adresse im Absatz .text (JMP _CorExeMain) eingetragen und sofort ausgeführt. _CorExeMain hat eine wichtige Aufgabe, denn sie entscheidet, welche Version der CLR die .Net-Applikation ausführen soll. Jetzt ist der Zeitpunkt gekommen, wo die CLR gestartet ist und für die restliche Ausführung verantwortlich ist.



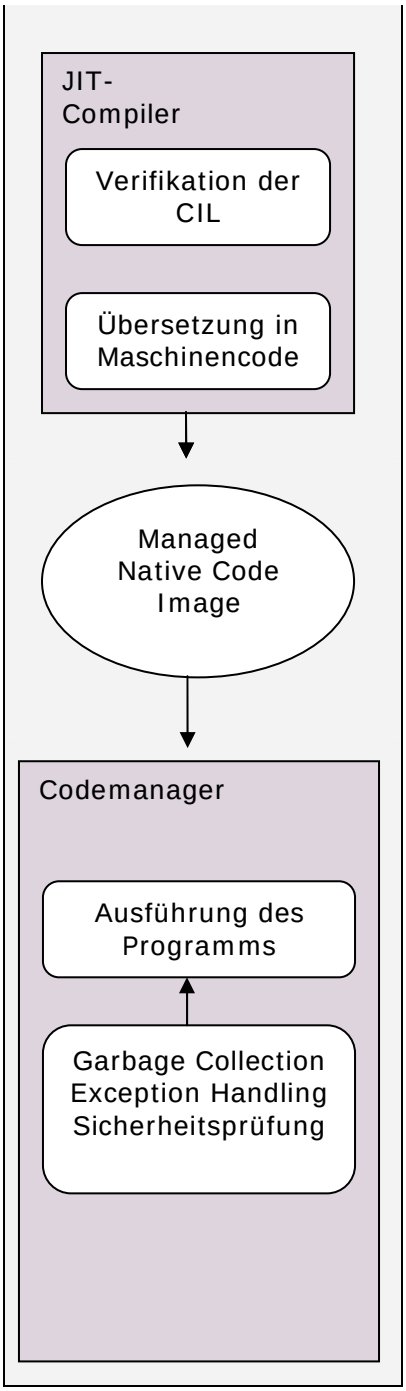
Die CLR muss zuerst den Eintrittspunkt - gekennzeichnet mit .entrypoint – im Assembly finden. Die auszuführende Methode gehört einem Typ an und ist einem Assembly zugeordnet. Die Implementierung befindet sich in einem Modul. Dieses Modul entspricht einer .NET-PE-Datei auf der Festplatte. Anschließend wird diese PE-Datei geladen.

Die nachfolgende Aufgabe und zwar das Laden eines Assemblies bzw. erzeugen eines Assemblyobjekts findet im Applikationsdomän statt und die dafür erforderlichen Informationen sind in der PE-Datei enthalten.

Weiters sollte erwähnt werden, dass jede Applikation einen eigenen Bereich im CLR-Prozess erhält, der als Applikationsdomän bezeichnet wird. Sobald die CLR gestartet wird, wird immer ein Default-AppDomain erzeugt. Der Vorteil ist, dass für die Ausführung mehrerer Applikationen nur ein Prozess erforderlich ist und somit die Ausführung rascher erfolgt. Dies ist möglich, weil der auszuführende Code typsicher ist und nicht auf unerlaubte Speicherbereiche zugreift. In solch einem Domän werden die Assemblies, Module, Typen und Methoden verwaltet.

Der nächste Schritt besteht darin, dass der Sicherheitsmanager überprüft, welche Rechte dem Assembly zugeordnet werden dürfen. (codebasierte Sicherheit)

Anschließend tritt der Klassenlader in Aktion. Dieser findet alle notwendigen Informationen im Assembly um ein Modulobjekt erzeugen zu können. An dieser Stelle kann es möglich sein, dass der Klassenlader ein weiteres Laden einer PE-Datei fordert. Nachdem der Klassenlader ein Modulobjekt erzeugt hat, kann er einen Typen laden, weil im Modul alle Informationen vorliegen, die für ein Typobjekt erforderlich sind.



Nun folgt der Zeitpunkt, an dem der JIT-Compiler beginnt den CIL-Code zu verifizieren. Das bedeutet es muss die Typsicherheit eines Programmes vor der Übersetzung bewiesen werden, bevor CIL-Code in Maschinencode übersetzt wird. An dieser Stelle sei erwähnt, dass ein Teil der Verifikation auch vom Klassenlader gemacht wird, beispielsweise überprüft dieser, ob die Eingangswerte mit der vorgeschriebenen Signatur übereinstimmen.

Nachdem nun der native Code vorhanden ist, kann mit dessen Ausführung begonnen werden. Dabei kann der Codemanager das Laden eines weiteren Typs, Moduls oder Assemblies anfordern, weil ein noch nicht vorhandener Typ verwendet wird. Aber es kann auch passieren, dass Methoden benötigt werden, welche vom JIT-Compiler noch nicht übersetzt wurden.

Der Codemanager ist verantwortlich für folgende Punkte:

- Speicherverwaltung
- Garbage Collection
- Exception Handling
- Sicherheitsprüungen
- Unterstützung von Entwicklerservices (Debugging, Profiling)

Es gibt noch zwei weitere Methoden zu der hier beschriebenen Standardvariante. Für Systeme mit wenig Speicher gibt es einen EconoJIT, der die Möglichkeit hat, den Methoden Cache während der Laufzeit wieder zu leeren, um Speicherplatz zu gewinnen. Natürlich müssen dann Methoden neu übersetzt werden, wenn sie aus dem Speicher gelöscht wurden. Eine andere Variante bietet die Codeerzeugung bei der Installation. Hier wird während der Installation der gesamte Code optimiert und in Maschinensprache übersetzt. Hierfür wird das Tool EGen.exe benutzt.

5 Literaturhinweis

Andrew Troelsen: C# und die .NET-Plattorm. 1 Auflage, mitp-Verlag, 2002

Wolfgang Beer, Dietrich Birngruber, Hanspeter Mössenböck, Albrecht Wöß: Die .NET-Technologie, 1.Auflage, dpunkt.verlag, 2003

Kevin Burton: NET Common Language Runtime Unleashed, Sams Publishing 2002

Jeffrey Richter: Applied .NET Framework Programming, Microsoft Press, 2002

ECMA-335 Standard: Common Language Infrastructure.
http://www.ecma-international.org/publications/standards/ecma-335.htm

http://msdn.microsoft.com/netframework/using/Understanding/default.aspx