Seminarbericht zum Vortrag

„Linker & Loader in .NET“

im Rahmen der Lehrveranstaltung:
„Seminar aus Softwareentwicklung (Inside Java and .NET)“
LVA Nr 338.177

Wintersemester 2003/04

Autor:

August Steinbacher
Matrikelnummer: 9955811
Studienkennzahl: 880
Email: august.steinbacher@students.jku.at

Betreuer:

o.Univ.-Prof. Dr. Hanspeter Mössenböck
Institut für Systemsoftware (www.ssw.uni-linz.ac.at)

Inhalt:

1 Einleitung und Motivation

Diese Arbeit beschäftigt sich mit Microsoft's .NET Framework, insbesondere mit den internen Abläufen, die für das Binden (Linken) und Laden von Code zuständig sind. Microsoft hat bei der Entwicklung des .NET Framworks versucht, den bereits von anderen Sprachen (zum Beispiel Java) eingeschlagenen Weg, bei der Codeerzeugung über eine Zwischensprache zu gehen, konsequent fortgesetzt.
Ein Compiler einer .NET-Sprache erzeugt nicht sofort maschinenspezifischen Code (native code), sondern einen plattformunabhängigen Zwischencode. Dieser Zwischencode heißt Microsoft Intermediate Language (MSIL) und wird in sogenannten Assemblies gespeichert. Erst zur Laufzeit wird dieser MSIL-Code in einen prozessorspezifischen Maschinencode umgewandelt. MSIL-Code wird – im Gegensatz zu Java-Bytecode – allerdings niemals interpretiert, sondern immer kompiliert. Diese finale Kompilation erledigt der Just-In-Time Compiler (JIT Compiler). Zur JIT-Kompilierung kommt es immer erst im allerletzten Moment, Methoden werden also erst kompiliert, wenn sie zum ersten Mal aufgerufen werden. Ebenso werden Typen und Klassen erst geladen, wenn sie zum ersten Mal benötigt werden. In der Regel werden also während eines Programmlaufs ständig Typen und Klassen nachgeladen bzw. Methoden kompiliert. Dieses Prinzip nennt man auch „deferred loading".
Nun kann leicht der Eindruck entstehen, dass der Aufwand, der betrieben wird, um das deferred loading zu realisieren sich sehr negativ auf das Laufzeitverhalten von .NET-Anwendungen auswirkt. Es ist aber eher das Gegenteil der Fall, denn dadurch, dass Komponenten erst geladen werden, wenn sie gebraucht werden, werden Ressourcen eingespart. Des weiteren wird durch die JIT-Kompilierung vermieden, dass nie aufgerufene Methoden unnötigerweise übersetzt werden.
Startet man eine .NET-Anwendung, so vergeht zwar eine gewisse Setup-Zeit, wo die verschiedenen Komponenten geladen, Einsprungpunkte gesucht, und die ersten Methoden übersetzt werden. Je länger die Anwendung dann läuft, desto wahrscheinlicher ist, dass benötigte Typen schon im Speicher vorhanden sind und Methoden als native code vorliegen. Hier liegt auch einer der Hauptvorteile der Common Language Runtime des .NET Framworks gegenüber der Java Virtual Machine: Kompilierter Code kann viel schneller ausgeführt werden als interpretierter.
Ziel dieser Seminararbeit ist es, die komplexen Vorgänge innerhalb der CLR, die dieses Ausführungsmodell ermöglichen, möglichst anschaulich darzustellen. Es kann aber nicht auf alle Feinheiten und Spezialitäten des .NET Framworks eingegangen werden, hierzu sei auf die am Ende der Arbeit angegebenen Referenzen verwiesen.

2 Vergleich Traditioneller Ansatz - .NET Modell

2.1 Traditioneller Ansatz

Traditionellerweise wird der Compiler unterteilt in Compiler-Frontend und Compiler-Backend. Aufgabe des Frontends ist, den High Level Code zu parsen und Zwischencode für das Backend zu erzeugen. Das Backend des Compilers nimmt diesen Zwischencode und erzeugt daraus den maschinenabhängigen native code. Dabei müssen Platzhalter für die einzelnen imports & exports geschaffen werden. Anschließend wird der Linker aktiv, der diese Platzhalter auflöst und aus den verschiedenen native code – Modulen eine einzelne ausführbare Datei erzeugt. Diese Datei ist dann auch die physische Repräsentation des Programms, die schließlich zu den Kunden ausgeliefert wird. Startet man das Programm beim

Kunden, ist der Lader dafür verantwortlich, die Datei in den Speicher zu laden und anschließend die Kontrolle an das Betriebssystem zu übergeben.
Das problematische an diesem Ansatz ist, dass schon sehr früh der endgültige native code erzeugt wird, nämlich noch zum Entwicklungszeitpunkt. Dadurch kann man nur sehr schwer Applikationen entwickeln, die auf verschiedenen Architekturen gleichwertig laufen sollen. Des weiteren ist es praktisch unmöglich, zur Laufzeit auf aktuellere Bibliotheken umzusteigen, da Bibliotheken ja schon am Entwicklungsrechner fix gebunden werden. Unter Win32 hat man dieses Problem mithilfe der dynamically loadable libraries (DLLs) teilweise entschärft, allerdings entstanden dadurch auch viele neue Probleme (Stichwort „DLL-Hell“).

2.2 .NET Modell

Microsoft versuchte beim .NET Framwork die zuvor genannten Probleme zu vermeiden, indem das Compiler Backend, Linken und Laden erst dort durchgeführt wird, wo das Programm auch laufen soll: auf der Zielmaschine. Der Entwickler der Software erzeugt mit dem Compiler einen Zwischencode, der durch die Microsoft Intermediate Language (MSIL) repräsentiert wird. Dieser Zwischencode wird, angereichert mit Metainformationen, in Assemblies gespeichert. Die Metainformationen werden später von der Laufzeitumgebung benötigt, um Informationen über die verschiedenen Typen des Programms abzufragen. Dazu gehören Informationen über die Namen und Ableitungsbeziehungen von Typen, Methodensignaturen und Abhängigkeiten der Typen untereinander.
Assemblies sind die Einheiten, in denen die Software ausgeliefert und auf der Zielmaschine installiert wird. Es gibt verschiedene Orte, wo Assemblies zur Laufzeit aufzufinden sind. Dies wird erledigt von der CLR, die auch für das JIT-kompilieren, Linken und Laden verantwortlich ist, wie in den späteren Kapiteln noch genauer beschrieben wird.

3 „The Big Picture“

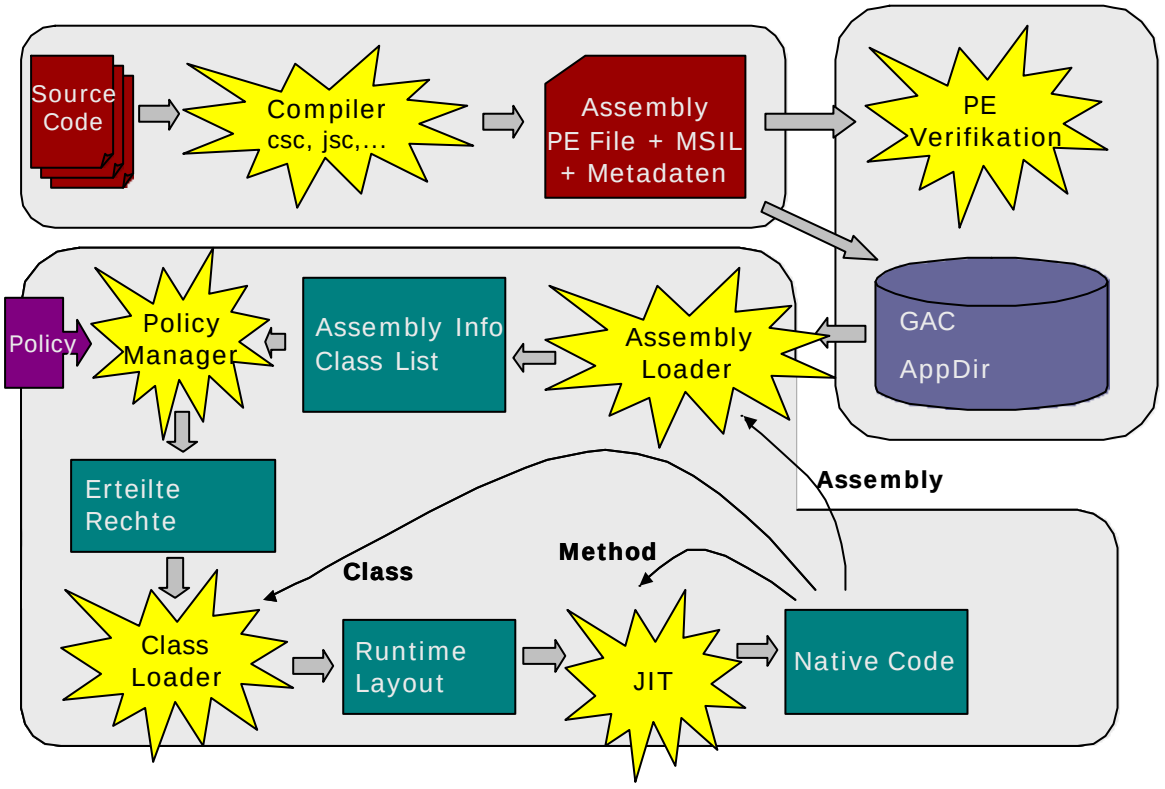

Abbildung 1: "The Big Picture"

Abbildung 1 zeigt den Lebenszyklus einer .NET Applikation. Dieser kann in 3 wesentliche Bereiche unterteilt werden, die im folgenden kurz erläutert werden.

3.1 Development (Entwicklung)

Der Entwickler schreibt die Applikation in einer Hochsprache, die .NET – fähig ist (C#, J#, VB.NET, Managed C++,...) und kompiliert die Quellcodes in die Zwischensprache, die Microsoft Intermediate Language (MSIL). Die Resultate der Kompilierung werden in Assemblies gespeichert, die anschließend ausgeliefert werden.

3.2 Deployment (Auslieferung)

Beim Kunden werden nun die Assemblies installiert. Dies geschieht entweder im Global Assembly Cache oder direkt im Applikationsverzeichnis. Wo genau ein Assembly installiert werden darf, hängt primär ab von seiner Vertrauenswürdigkeit, die durch öffentliche beziehungsweise private Schlüssel festgelegt wird. Jedes zu installierende Assembly wird darum zunächst vom PE Verifier auf seine Vertrauenswürdigkeit überprüft. Außerdem prüft der PE Verifier noch, ob die Datei eine gültige .NET PE-Datei ist. PE bedeutet „portable executable“ und ist ein Standarddateiformat für unter Windows ausführbare Dateien.

3.3 Execution (Ausführung)

Zu Beginn der Ausführung einer .NET Applikation muss der Assembly Loader die benötigten Assemblies lokalisieren (vgl. Kapitel 6: „Assemblies“) und im Speicher ein Abbild von ihnen erzeugen, mitsamt der Metainformationen, die von der Common Language Runtime während der Laufzeit benötigt werden. Der Policy Manager hat die Aufgabe, den geladenen Assemblies Rechte zu erteilen beziehungsweise nicht berechtigten Assemblies den „Zugang“ zur laufenden Applikation zu verwehren. Diese Rechte werden geprüft aufgrund verschiedener Policies, die in den Konfigurationsdateien definiert sind.
Wurden alle benötigten Rechte erteilt, so kann der Klassenlader die einzelnen Typen aus den Assemblies laden, Objektinstanzen am Heap erzeugen und die eigentliche Laufzeitumgebung aufbauen (vgl. Kapitel 8: „Erzeugen von Objektinstanzen“ und Kapitel 9: „Das .NET Runtime Layout).
Der JIT-Compiler schließlich erledigt die finale Übersetzung der Applikation in Maschinencode. Wie in Kapitel 1 schon erwähnt, werden immer nur jener MSIL-Code übersetzt, der auch tatsächlich in diesem Moment benötigt wird. Daher kommt es laufend vor, dass neue Methoden übersetzt werden, Klassen oder Assemblies nachgeladen werden müssen. Dies erledigen dann wieder der Assembly Loader, der Class Loader und der JIT-Compiler.
Es gibt auch die Möglichkeit, den Maschinencode schon bei der Installation eines Assemblies zu erzeugen, und zwar mit einem beim .NET Framework mitgelieferten Werkzeug, dem Native Image Generator (`ngen.exe`). Dieser ist in der Lage, bereits zum Installationszeitpunkt den Maschinencode zu erzeugen und ihn im Native Image Cache (NAC, Teil des GAC) abzulegen. Solche vorkompilierten Programme haben eine verkürzte Startzeit, sind aber nicht mehr so flexibel.
Bei der Installation des .NET Frameworks werden folgende sehr oft benutzte Assemblies automatisch vorkompiliert und im NAC gespeichert: `CustomMarshallers`, `mscorlib`,

System, System.Design, System.Drawing, System.Drawing.Design, System.Windows.Forms, System.XML.

4 Ausführen von .NET Applikationen

4.1 Starten einer .NET Applikation

Unter Windows werden .NET Applikationen ausgeführt, indem die .exe – Datei gestartet wird. Im Gegensatz zu Java (Starten von Java-Programmen: `java MyApp`) muss Windows nicht mitgeteilt werden, welche Runtime gestartet werden soll, sondern der Windows-Lader erkennt, dass es sich um eine .NET Applikation handelt und startet die CLR. Ab diesem Zeitpunkt fungiert die CLR als virtuelle Maschine und hat alleine die volle Kontrolle über die Applikation.

4.2 Applikationsdomänen

Jede .NET Applikation hat einen eigenen geschützten Bereich, innerhalb dem sie abläuft - die Applikationsdomäne. Applikationsdomänen sind isolierte Bereiche innerhalb eines CLR-Prozesses (ein CLR-Prozess entspricht einem Prozess des Betriebssystems). Innerhalb einer Applikationsdomäne befinden sich alle Informationen und Daten, die eine Applikation zur Laufzeit braucht (Assemblies, Module, Typen und ihre Methodentabellen, statische Felder und Objekte). Applikationsdomänen isolieren Applikationen voneinander, um zu verhindern, dass sich einzelne Applikationen gegenseitig beeinflussen. Dadurch wird es möglich, mehrere Instanzen ein und der selben Applikation innerhalb einer CLR parallel laufen zu lassen, ohne dass es zu Beeinträchtigungen kommt.
Zusätzlich zu den applikationsspezifischen Applikationsdomänen gibt es auch noch einen Bereich, auf den alle Applikationen innerhalb einer CLR zugreifen können: die shared Application Domain. Diese beinhaltet Komponenten, die von allen Applikationen benötigt werden (z.B. `mscorlib.dll`).
Es gibt auch die Möglichkeit, Komponenten die Überquerung der Grenzen von Applikationsdomänen zu gestatten. Solche Komponente heißen „agile Komponenten“. Sie müssen höchst vertrauenswürdig sein, d.h. sie müssen einen starken Namen haben und mit einem public key signiert sein.

5 Der Ladevorgang

Der Ladevorgang kann grob in drei Schritte unterteilt werden:

1. Laden der PE-Datei
2. Laden der Assemblies
3. Laden der Typen

Als erstes wird die PE-Datei von der Festplatte geladen und eine Repräsentation der darin enthaltenen Daten im Speicher erzeugt. Dies ist auch der einzige physische Festplattenzugriff während des Ladeprozesses. Aufgrund der Metainformationen in der PE-Datei können anschließend die einzelnen Assemblies lokalisiert werden. Nachdem alle benötigten Assemblies geladen sind, kann der Klassenlader aus ihnen die Typen laden. Im Gegensatz zu Java können unter .NET keine eigenen Klassenlader verwendet werden.

6 Assemblies

6.1 Auffinden von Assemblies

Für das Lokalisieren von Assemblies ist der Assembly Resolver zuständig. Dieser wird aktiv, wenn die CLR auf den Befehl Assembly.Load(name, culture, version, public key token) stößt. Anhand der hier übergebenen Parameter versucht der Assembly Resolver das gewünschte Assembly aufzufinden und seinen Ort dem Assembly Loader mitzuteilen, der für das eigentliche Laden zuständig ist. Der Ort kann entweder ein Pfad im Dateisystem oder ein Verweis ins Internet sein. Falls das Assembly aus dem Internet geladen werden soll, muss die ladende Klasse spezielle Web-Zugriffsrechte haben, ansonsten wird eine SecurityExeption geworfen.
Generell ist zwischen zwei Arten von Assemblies zu unterscheiden: öffentliche Assemblies und private Assemblies.
Öffentliche Assemblies werden zunächst im Global Assembly Cache (GAC) gesucht. Der GAC ist ein systemweites Verzeichnis für öffentliche Assemblies. Das Besondere am GAC ist, dass er im Gegensatz zu einem normalen Verzeichnis mehrere verschiedene Versionen einer Datei enthalten kann. Im GAC dürfen sich nur höchst vertrauenswürdige Assemblies befinden, das sind jene, die mit einem öffentlichen Schlüssel (public key) signiert sind.
Befindet sich das gesuchte Assembly nicht im GAC, so werden die einzelnen Konfigurationsdateien des .NET-Frameworks (application.config, publisher.config, machine.config) nach Hinweisen auf den Speicherort des Assemblies durchsucht. Solche Hinweise sind durch <CodeBase> - XML-Tags gekennzeichnet. Falls solche Hinweise vorhanden sind und die dadurch referenzierte Datei den Erwartungen entspricht, wird das Assembly geladen. Falls keine <CodeBase> - Verweise vorhanden sind oder es sich um ein privates Assembly handelt, wird versucht, das gewünschte Assembly im Applikationsverzeichnis zu finden. Diese Suche wird „Probing“ genannt.

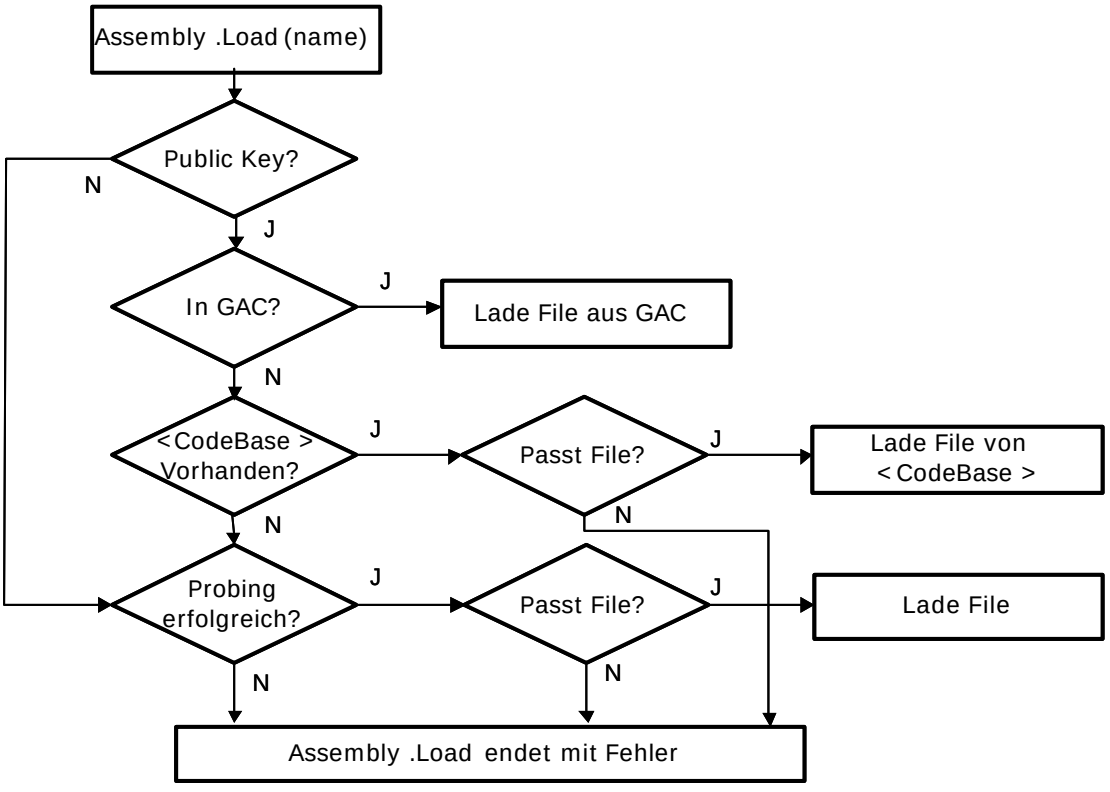

Abbildung 2: Auffinden von Assemblies

6.2 Probing

Beim Probing werden, ausgehend vom Wurzelverzeichnis der Applikation, gewisse Verzeichnisse nach der passenden Datei durchsucht. Die Auswahl der zu durchsuchenden Verzeichnisse basiert auf vier Kriterien:

- Basisverzeichnis der Applikation: Wurzelverzeichnis, in dem die Applikation ausgeführt wird.
- Name: der Name des referenzierten Assemblies.
- Kultur-Attribut: Kulturattribut, das im Assemblynamen optional angegeben werden kann.
- Privater Binpfad: kann vom Benutzer in einer Konfigurationsdatei (application.config) eingestellt werden und muss ein Unterverzeichnis des Wurzelverzeichnis der Applikation sein.

Assemblies, die als Kulturattribut „neutral“ haben, werden auf folgende Weise gesucht:

1) Als erstes wird im Wurzelverzeichnis der Applikation nachgesehen ([AppDir] \ [AssemblyName].dll).
2) Als nächstes wird geprüft, ob ein Unterverzeichnis mit dem Namen des Assemblies existiert ([AppDir] \ [AssemblyName] \ [AssemblyName].dll).
3) Falls in der Konfigurationsdatei ein privater Binpfad angegeben wurde, wird dann noch dieser Pfad überprüft ([AppDir] \ [BinPfad] \ [AssemblyName}.dll).

Bei Assemblies, die mit einem Kulturattribut versehen sind, sieht der Ablauf ähnlich aus, es wird aber noch ein zusätzliches Verzeichnis berücksichtigt, das dem Kulturattribut entspricht:

1) [AppDir] \ [Culture] \ [AssemblyName].dll
2) [AppDir] \ [Culture] \ [AssemblyName] \ [AssemblyName].dll
3) [AppDir] \ [BinPfad] \ [Culture] \ [AssemblyName].dll
4) [AppDir] \ [BinPfad] \ [Culture] \ [AssemblyName] \ [AssemblyName].dll

Da der Name des Assemblys keine Informationen über die Dateierweiterung enthält, muss diese erraten werden. Es wird zuerst nach .dll-Dateien und anschließend nach .exe-Dateien gesucht, weitere Endungen werden nicht überprüft.

6.3 Probing-Beispiel

Name des referenzierten Assemblies: myAssembly
Wurzelverzeichnis der Applikation: c:\myApp
Binpfad (application.config): \bin
Kulturattribut: de
Es werden die folgenden Orte durchsucht:

```
c:\myApp\de\myAssembly.dll
c:\myApp\de\myAssembly\myAssembly.dll
c:\myApp\bin\de\myAssembly.dll
c:\myApp\bin\de\myAssembly\myAssembly.dll

c:\myApp\de\myAssembly.exe
c:\myApp\de\myAssembly\myAssembly.exe
```

```
c:\myApp\bin\de\myAssembly.exe
c:\myApp\bin\de\myAssembly\myAssembly.exe
```

Wurde an keinem dieser Orte das referenzierte Assembly gefunden, so schlug das Probing fehl und `Assembly.Load(..)` endet mit einem Fehler. Falls das Assembly gefunden wurde und es den Erwartungen entspricht, wird dem Klassenlader der Ort mitgeteilt und dieser beginnt damit, die Typen zu laden.

7 Laden eines Typs

Für das Laden von Typen ist der Klassenlader zuständig. Während des Ladens eines Typs T müssen einige Aufgaben erledigt werden:

- Bestimmung des benötigten Speicherplatzes.
- Bestimmung des Speicherlayouts der T-Objekte.
- Auflösen der Referenzen von T auf bereits geladene Typen.
- Auflösen der Referenzen von T auf noch nicht geladene Typen: entweder werden die referenzierten Typen sofort nachgeladen, oder die Referenzen werden registriert sodass sie beim späteren Laden der referenzierten Typen problemlos aufgelöst werden können.
- Erzeugung von Stubs für die implementierten Methoden von T. Ein Stub löst beim ersten Aufruf der Methode die JIT-Kompilierung aus

Eine weitere Prüfung, die beim Laden und Übersetzen von Typen durchgeführt wird, ist die Prüfung auf Typsicherheit. Diese Prüfung übernimmt der Verifizierer, der Teil des JIT-Compilers ist. Typsichere Programme greifen nur auf für sie eingerichteten Speicherbereiche zu und verwenden Objekte nur über deren Schnittstelle. Zur Verifikation der Typsicherheit kommt ein konservativer Algorithmus zum Einsatz, der garantiert, dass von ihm für typsicher befundene Programme dies auch wirklich sind. Das bedeutet allerdings, dass möglicherweise typsichere Programme als nicht typsicher klassifiziert werden. Es kann aber niemals vorkommen, dass ein nicht typsicheres Programm fälschlicherweise als typsicher klassifiziert wird. Es ergibt sich eine Einteilung in vier Kategorien:

- Ungültig: Jene CIL-Programme, für die der JIT-Compiler keinen Maschinencode erzeugen kann, weil der Code entweder nicht dem CIL-Format entspricht oder undefinierte Befehle enthält
- Gültig: Alle CIL-Programme, die vom JIT-Compiler übersetzbar sind
- Typsicher: Alle gültigen CIL-Programme, die typsicher sind.
- Verifizierbar: Typsichere CIL-Programme, für die bewiesen werden kann, dass sie typsicher sind.

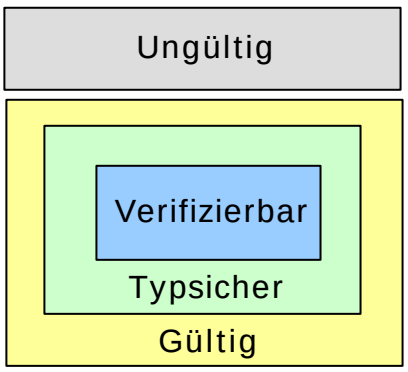

Abbildung 3: 4 Code-Kategorien

Der Verifizierer kann allerdings den Kontext der Codestücke nicht prüfen, da er immer nur einzelne Methoden sieht. Daher ist nicht prüfbar, ob alle Eingangswerte die vorgeschriebene Typsignatur aufweisen. Diese Prüfung übernimmt der Klassenlader, der seine Informationen aus den Metadaten des Assemblies erhält.

Beispiel für gültigen aber nicht typsicheren CIL Code ist die von C++ bekannte Pointerarithmetik. Will man ein CIL-Programm erzeugen, das solche Befehle enthält, so muss man dem Compiler mittels einer Compileroption mitteilen, dass er das Programm trotzdem übersetzen soll, oder man bettet den unsicheren Code von vornherein in einen `unsafe-Block`.

Nachdem ein Typ geladen wurde, können Instanzen des Typs erzeugt werden, die sogenannten Objektinstanzen. Näheres dazu im nächsten Kapitel.

8 Erzeugen von Objektinstanzen

Objektinstanzen werden am Heap erzeugt. Jede Objektinstanz hat einen Zeiger auf die Methodentabelle ihres Typs. In der Methodentabelle stehen Verweise, wo im Speicher die jeweiligen Methoden als Maschinencode gespeichert sind. Methodentabellen sind typrelevant, pro Typ existiert genau eine Tabelle, die dann von den Instanzen des jeweiligen Typs referenziert wird. Statische Felder (z.B. Klassenvariablen) werden ebenfalls in der Methodentabelle gespeichert, alle anderen instanzspezifischen Daten befinden sich direkt am Heap.

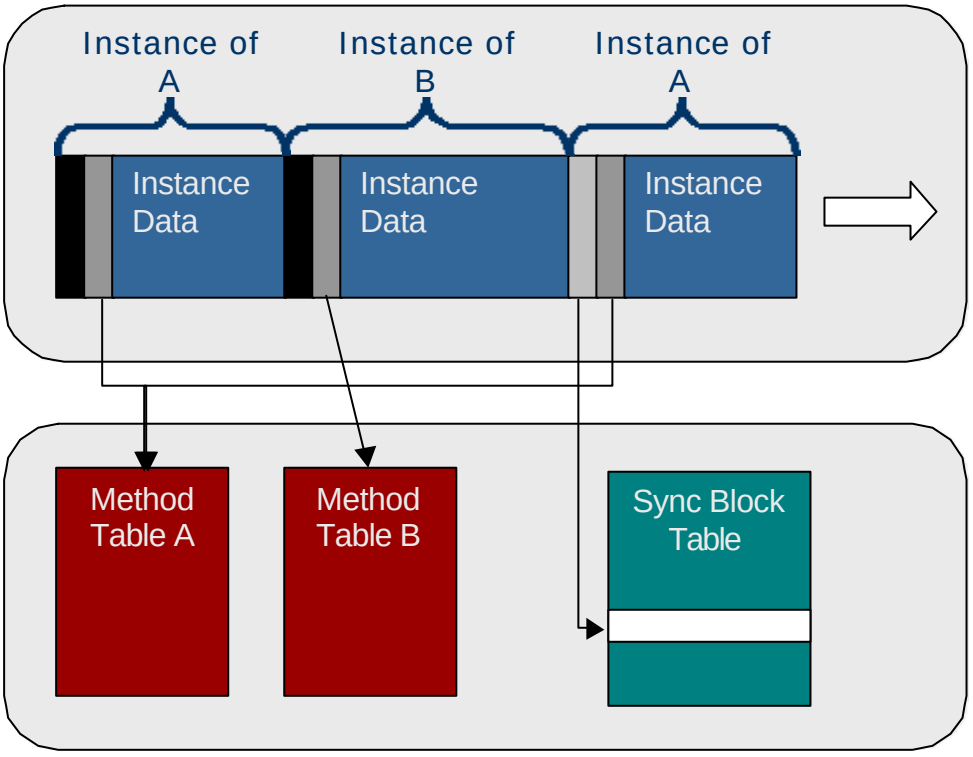

Abbildung 4: Objektinstanzen am Heap

Weiters hat jede Objektinstanz am Heap einen optionalen Zeiger auf eine Tabelle, die zu Synchronisationszwecken verwendet wird („Sync Block Table“). Diese Tabelle existiert genau einmal pro CLR und alle Prozesse innerhalb der CLR haben darauf Zugriff. Sie wird verwendet, um Threadsynchronisation zu betreiben. Nicht alle Objekte haben einen Eintrag in dieser Tabelle, nur jene, auf die synchroner Zugriff gefordert wird.

9 Das .NET Runtime Layout

Die nächste Abbildung zeigt das vollständige Layout eines Objekts, wie es zur Laufzeit aussieht. Dieses Runtime Layout wird vom Klassenlader beim Laden eines Typs erzeugt. Die Informationen die er dazu braucht (Anzahl der Felder, Anzahl der Methoden und Interfaces, Größe der Datentypen,...) gehen hervor aus den Metadaten des Assemblies, in dem der Typ deklariert ist. Im folgenden werden die einzelnen Komponenten genauer beschrieben.

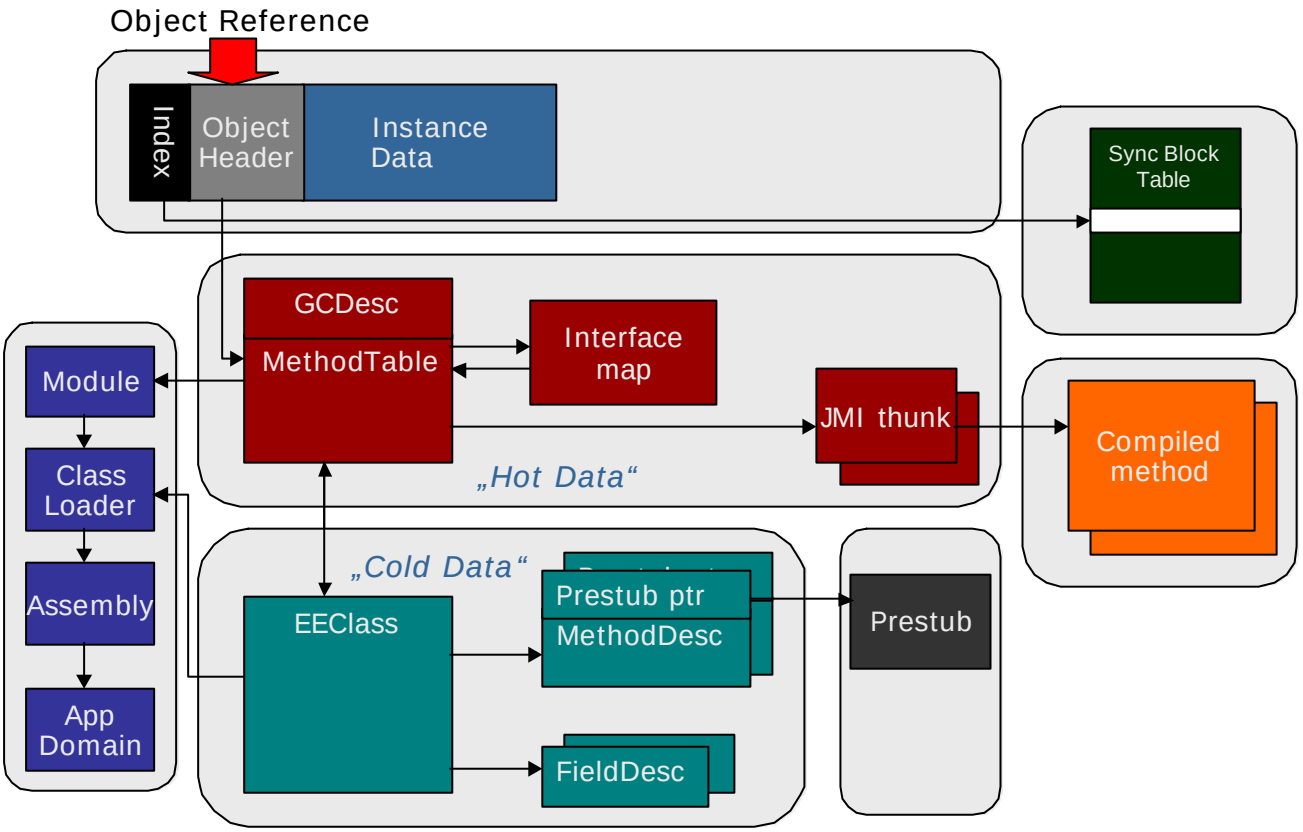

Abbildung 5: Runtime Layout

- *Object Reference*: eine Referenz eines anderen Objekts auf dieses Objekt.
- *Index*: Index des Objekts am Heap; kann optional einen Zeiger auf die Synchronisationstabelle enthalten.
- *Object Header*: enthält einen Zeiger auf die Methodentabelle des Typs.
- *Instance Data*: Instanzspezifische Daten des Objekts (z.B. lokale Variablen)
- *Sync Block Table*: Tabelle, die Informationen zum Synchronisieren von Objekten enthält.
- *MethodTable*: Methodentabelle. Die Methodentabelle ist unterteilt in Slots, wobei es für jede Methode des Typs einen Slot gibt. Der Slot zeigt auf den *JMI thunk* der jeweiligen Methode, und dieser wiederum enthält die Adresse, wo im physischen Speicher sich der Maschinencode der Methode befindet. Beim erstmaligen Laden eines Typs stehen in der Regel die zugehörigen Methoden noch nicht in kompilierter Form zur Verfügung, sondern die Kompilierung wird erst ausgelöst, wenn die Methoden das erste Mal aufgerufen werden. Darum werden die Slots der

Methodentabelle initialisiert mit Verweisen auf Deskriptoren der Methoden (*MethodDesc*, siehe unten).

- *GCDesc:* Garbage Collector Descriptor. Teil der Methodentabelle, der Informationen über das Objektlayout enthält, die vom Garbage Collector zur Speicherbereinigung benötigt werden.
- *JMI thunk*: Jittet Method Info. Enthält die Speicheradresse der zugehörigen Methode. Diese zusätzliche Dereferenzierung beim Auffinden der kompilierten Methode ist deswegen notwendig, weil es passieren kann, dass der Garbage Collector im Zuge der Speicherbereinigung den kompilierten Methodencode löscht. Somit würde, wenn die Slots in der Methodentabelle direkt auf den kompilierten Code zeigen würden, und der Code vom Garbage Collector inzwischen entfernt wurde, es hier zu einem unerlaubten Speicherzugriff kommen, was den Abbruch der Applikation zur Folge hätte. Darum wurde der Umweg über den JMI thunk eingeführt. Falls eine Methode vom Garbage Collector aus dem Speicher entfernt wurde, wird dies im JMI thunk vermerkt. Bei einem Aufruf einer auf solcherart entfernten Methode merkt nun die CLR, dass der Maschinencode der Methode nicht mehr existiert und veranlasst den JIT-Compiler, die Methode zu kompilieren und die Speicheradresse des Maschinencodes im JMI thunk einzutragen.
  Nun stellt sich die Frage, warum man diesen Umweg geht, man könnte ja auch die einzelnen JMI Thunks direkt in den Slots der Methodentabelle unterbringen und sich somit eine Dereferenzierung ersparen, was sich positiv auf die Laufzeit auswirken würde. Die Antwort auf diese Frage ist offensichtlich: Durch die Vererbungshierarchie werden viele Methoden von verschiedenen Typen verwendet. Da in .NET alle Typen von System.Object abgeleitet sind, sind auch alle Methoden, die in System.Object implementiert sind, auf alle Typen anwendbar (z.B. System.Object.Equals()). Nun wäre es sehr speicherverschwendend, wenn diese Methoden für jeden einzelnen Typ separat kompiliert und in eigenen Speicherbereichen abgelegt werden würden. Stattdessen existiert jede Methode – und somit auch der zugehörige JMI thunk - genau einmal pro CLR als Maschinencode, und jeder Typ der die Methode kennt verweist in seiner Methodentabelle auf diese eine kompilierte Methode. Somit entsteht eine zentrale Anlaufstelle für die Methoden. Wird nun der Maschinencode einer Methode vom Garbage Collector entfernt, so muss dies nur an dieser einen Stelle vermerkt werden, und alle Objekte, die mit dieser Methode arbeiten, wissen Bescheid. Gäbe es den *JMI thunk* nicht, so müsste die Laufzeitumgebung die komplette Vererbungshierarchie bis System.Object durchgehen und in allen Methodentabellen den entsprechenden Slot ändern, was sich wiederum sehr negativ auf das Laufzeitverhalten auswirken würde.
- *Interface Map:* Die Interface Map existiert einmal pro CLR und wird von allen geladenen Typen gemeinsam verwendet. Es sind hier alle applikationsweit implementierten Interfaces vermerkt und eindeutig durchnummeriert. Weil ein Interface keine Anweisungen enthalten darf, sondern nur die Schnittstellen für die implementierenden Methoden definiert, gibt es hier auch keine Zeiger zu Maschinencode. Stattdessen verweisen die Einträge in der Interface Map in die Methodentabelle, und zwar genau auf die Slots jener Methoden, die die jeweilige Schnittstelle implementieren. Wie dies genau funktioniert, wird weiter unten noch ein Beispiel zeigen.
- *Compiled Method:* Bereich im Speicher, in dem sich der Maschinencode der jeweiligen Methode befindet.
- *EEClass:* Execution Engine Class. Hier finden sich Informationen zur Struktur des Typs: die Anzahl der Interfaces, die Anzahl der Objektreferenzen die eine Instanz des Typs enthält (dies wird benötigt, um GCDesc zu erzeugen) und die Anzahl der

statischen Felder. Jede EEClass ist mit der EEClass ihres Supertyps verlinkt, um an Informationen über geerbte Felder und Methoden zu kommen. Die EEClass hat auch eine Referenz auf den Klassenlader, um das Nachladen benötigter Typen anzufordern. Hauptaufgabe der EEClass ist es aber, Auskunft über die strukturelle Beschaffenheit der Felder und Methoden des jeweiligen Typs zu geben. Dies wird realisiert durch Zeiger auf die Deskriptoren der Felder (*FieldDesc*) und Methoden (*MethodDesc*).

- *FieldDesc*: Deskriptor eines Feldes. Für jedes Feld wird hier vermerkt, welche Eigenschaften es hat (Name, ist es statisch, Thread-lokal, Context-lokal, protected, ist es ein agiles Feld (agile Felder dürfen Application Domains überqueren),...).
- *MethodDesc:* Deskriptor einer Methode. Ähnlich wie FieldDesc, zusätzlich ist hier vermerkt, welche Stelle (Slot) in der Methodentabelle dieser Methode zugewiesen wurde. Außerdem können über den MethodDesc die Argumente, die eine Methode erwartet, abgefragt werden. Dies ist wichtig, wenn man Reflection betreiben will.
- *Prestub Ptr:* Für jeden Methodendeskriptor existiert ein prestub pointer. Dieser enthält Anweisungen, die die JIT-Kompilierung der Methode auslösen. Beim erstmaligen Laden eines Typs zeigen die Pointer in den Slots der Methodentabelle auf diese prestub pointer. Wird nun eine noch nicht kompilierte Methode aufgerufen, findet ein Stubcall zum prestub pointer statt, wodurch die Methode JIT-kompiliert und ein JMI-Thunk erzeugt wird, der auf den erzeugten Maschinencode zeigt. Anschließend wird dann noch der entsprechende MethodTable – Slot modifiziert, so dass er auf den erzeugten JMI-Thunk zeigt.

Zum besseren Verständnis folgt nun eine kurze Zusammenfassung:
Objektinstanzen werden immer am Heap erzeugt. Jede Objektinstanz hat einen Zeiger auf die Methodentabelle ihres Typs (MethodTable), und jede Methodentabelle hat einen Zeiger auf die Deskriptorklasse des Typs (EEClass).
Die Auftrennung in Methodentabelle und Deskriptorklasse erfolgte aus Laufzeitgründen. Method Table, Interface Map und den Prestub Pointer nennt man auch „Hot Data“ oder „Runtime Info“, da diese während der Ausführung eines Programms ständig von der Runtime benötigt werden. Darum wurde ein Hauptaugenmerk darauf gelegt, diese Daten möglichst kompakt zu halten, um möglichst schnell und ohne Umwege damit arbeiten zu können.
EEClass, MethodDesc und FieldDesc nennt man auch „Cold Data“ oder „Reflection Info“. Auf diese Daten wird nicht ständig zugegriffen, nur beim Erzeugen eines Objektes, während der JIT-Kompilierung und wenn man Reflection betreibt.
Der Maschinencode für die Methoden wird erst dann erzeugt, wenn eine Methode das erste Mal aufgerufen wird (JIT-Kompilierung). Der kompilierte Maschinencode der Methoden kann bei Bedarf vom Garbage Collector entfernt werden, um zu vermeiden, dass der Speicher am Heap ausgeht. Damit in diesem Fall der Methodenzeiger in der Methodentabelle nicht ins Leere zeigt, gibt es den JMI Thunk, der dafür verantwortlich ist, solche ins Leere gehende Methodenaufrufe abzufangen und den JIT-Compiler zu beauftragen, die Methode neu zu kompilieren.

10 Methodentabelle & Interface Map

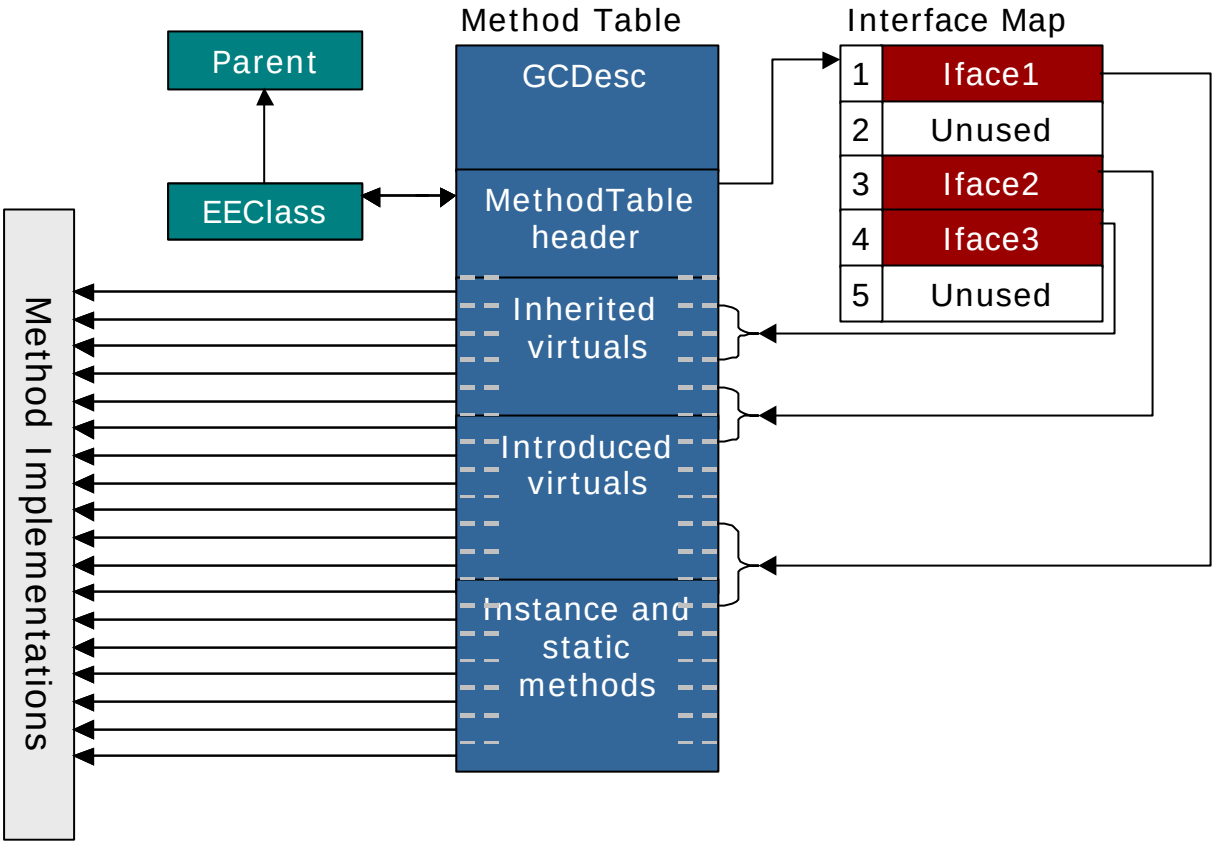

Abbildung 6: Method Table & Interface Map

Die Methodentabelle dient dazu, zur Laufzeit die Einsprungpunkte in die einzelnen Methoden zu finden. Wie weiter oben schon erwähnt, ist diese Tabelle in verschiedene Bereiche unterteilt. Am Anfang befinden sich Garbage Collector-spezifische Informationen, danach kommt der Header mit einem Pointer auf die zugehörige EEClass und auf die InterfaceMap. Anschließend folgen die Slots, in denen die Adressen der einzelnen Methoden stehen, wobei die Methoden in drei verschiedene Arten unterteilt sind:

- *Inherited Virtuals:* Geerbte Methoden („overrides“); Dies sind jene Methoden, die aus einer Superklasse geerbt wurden. Ähnlich wie bei der Java Virtual Machine hält jeder Typ alle auf ihn anwendbaren Methoden in seiner Methodentabelle, auch alle geerbten Methoden haben einen Eintrag. Man könnte zwar auch die geerbten Methoden auffinden, indem man mithilfe des parent-Zeigers in der EEClass die Vererbungshierarchie hinauf traversiert bis System.Object, jedoch würde dies erheblichen Laufzeitaufwand bedeuten. Stattdessen nimmt man lieber in Kauf, dass die Methodentabellen etwas länger werden. Nur System.Object hat keine Einträge in diesem Bereich der Methodentabelle, da System.Object der Basistyp aller .NET Typen ist und somit von niemandem erbt.
- *Introduced Virtuals:* Vom Typ definierte, vererbbare Methoden („public virtual method()“); Jene Methoden, die der Typ neu definiert und als virtuell (vererbbar) deklariert.
- *Instance & Static Methods:* Statische (Klassenmethoden) und nicht vererbbare Methoden.

In der InterfaceMap sind alle applikationsweit implementierten Interfaces eingetragen. Jede Zeile repräsentiert ein Interface und enthält einen Zeiger zurück auf die entsprechenden Slots der Methodentabelle des Typs, der das Interface implementiert. Da ein Interface mehrere Methoden haben kann und es pro Interface aber nur einen Zeiger auf die Methodentabelle gibt, müssen die Methoden-Slots in der Methodentabelle genau in der Reihenfolge vorkommen, die das Interface vorgibt, was meistens auch der Fall ist. Ist keine solche Überlagerung möglich, so werden einfach zusätzliche Slots in die Methodentabelle eingefügt, um die richtige Reihenfolge herzustellen. Dadurch entsteht der Effekt, dass manche Slots der Methodentabelle auf die selbe Methode zeigen, was aber nur einen geringen Speicher-Overhead darstellt.
Obwohl jeder Typ aufgrund von Einträgen in seiner EEClass die Position seiner Interfaces in der InterfaceMap genau kennt und somit der Aufruf einer Methode über ein Interface in konstanter Zeit durchgeführt werden kann, stellt die zusätzliche Indirektion einen Laufzeitoverhead dar. Dies ist auch der Grund, warum Methodenaufrufe über Interfaces etwas langsamer sind als solche Aufrufe, die direkt über das jeweilige Objekt erfolgen.

Nachfolgendes Beispiel veranschaulicht das Überlagern der Interface-Methoden in den Methodentabellen:

Gegeben seien zwei Interfaces `I1` und `I2` mit jeweils zwei Methodenschnittstellen und drei Klassen `A`, `B` und `C`, die jeweils voneinander erben und Methodenimplementierungen für die Interfaces zur Verfügung stellen:

```
interface I1 { m1(); m2(); }
interface I2 { m2(); m3(); }

class A : I1 {
        m1() {..};
        m2() {..};
}

class B : A, I2 {
        m3() {..};
}

class C : B {
        I2.m2() {..};
}
```

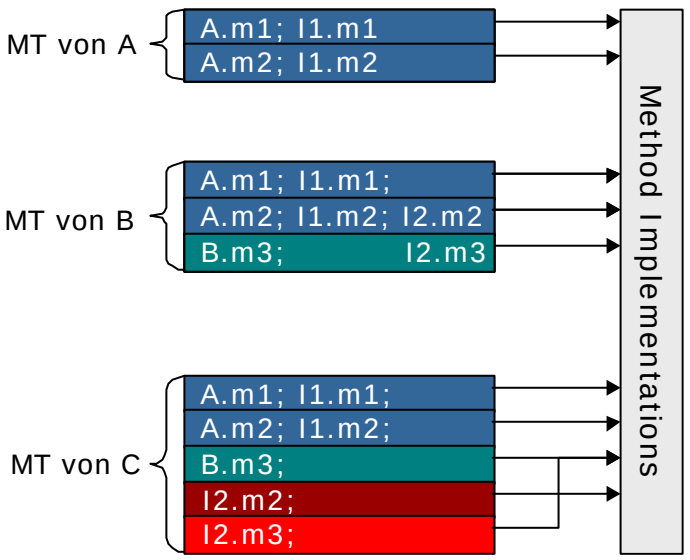

Abbildung 7: Mapping von Interface-Methoden

Als erstes wird hier die Klasse `A` erzeugt. In der Methodentabelle von `A` werden die Adressen der Methoden `m1()` und `m2()` vermerkt. Weil `A` das Interface `I1` implementiert, wird anschließend versucht, die Methoden von `I1` auf die schon vorhandenen Slots der Methoden von `A` abzubilden, was auch erfolgreich ist.
Als nächstes folgt die Erzeugung der Klasse `B`. `B` erweitert `A`, also erbt `B` alle im MethodTable von `A` eingetragenen Methoden-Slots. Zusätzlich hat `B` eine Methode `m3()`, die wieder im MethodTable eingetragen wird. Danach wird wieder das Überlagern der Methoden der implementierten Interfaces durchgeführt, in diesem Fall für die Methoden der Interfaces

I1 und I2. Da im MethodTable von B die Methoden m1() und m2() beziehungsweise m2() und m3() direkt aufeinander folgen, ist dies wieder ohne Probleme möglich.
Die Klasse C erbt wiederum die Einträge im MethodTable von B. Die Methoden von I1 können wieder exakt überlagert werden. Allerdings implementiert C explizit die Methode I2.m2(), wodurch diese Methode einen eigenen Slot im MethodTable von C erhält. Also muss I2 in der InterfaceMap auf die Methode I2.M2() im MethodTable von C zeigen. Wie oben schon erwähnt, erwartet die Runtime, dass die Methoden eines Interfaces unmittelbar aufeinander im MethodTable der implementierenden Klasse folgen. Darum wird im MethodTable von C unmittelbar nach dem Slot für I2.m2() ein Slot für I2.m3() erzeugt, der dann auf die selbe Methodenimplementierung zeigt wie der schon vorhandenen Slot B.m3().

11 Literatur & Links

- Don Box, Chris Sells: Essential .NET, The Common Language Runtime, Addison-Wesley 2003
- Dave Stutz, Ted Neward, Geoff Shilling: Shared Source CLI Essentials. O'Reilly 2003
- W.Beer, D.Birngruber, H.Mössenböck, A.Wöß: Die .NET-Technologie, dpunkt.verlag 2002
- Jeffrey Richter: Applied Microsoft .NET Framework Programming, Microsoft Press, 2002
- http://www.msdn.microsoft.com/netframework/
- http://www.dotnetframework.de
- http://dotnet.di.unipi.it/
- http://dotnet.jku.at
- http://www.sscli.net/

12 Abbildungsverzeichnis