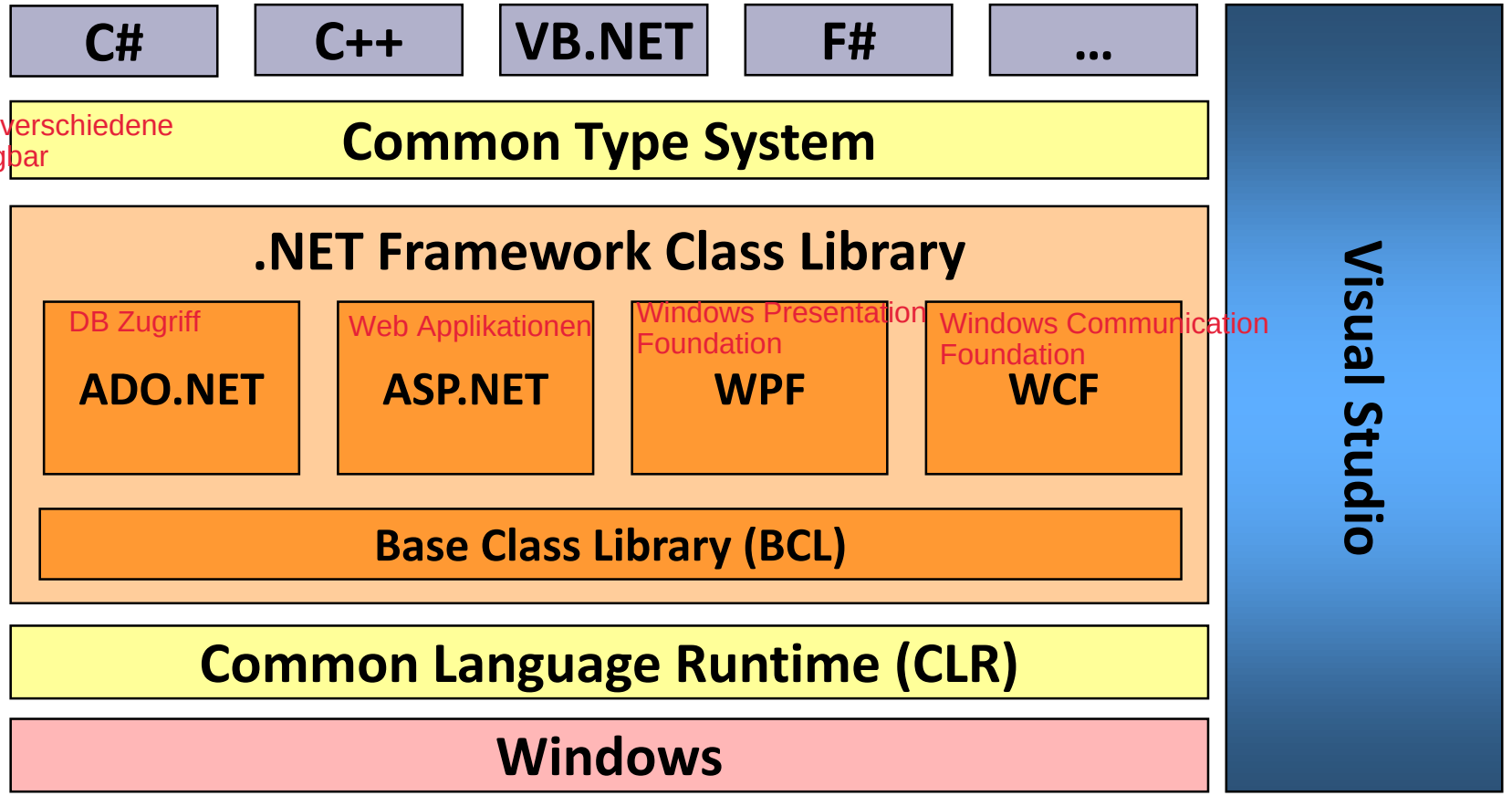
.NET: Architektur

© J. Heinzelreiter

Version 5.8

.NET Framework 2.0 – 4.7 (Full Framework)

nicht alle Anforderungen der modernen Web Welt deckt von Funktionalität her sehr viel ab



alle Klassen für verschiedene Sprachen verfügbar

Framework: CLR (Laufzeitumgebung (JVM)), Bibliothek
Framework - kein fertige bestehende Anwendung, die nur angepasst werden muss

Varianten von .NET

Full Framework läuft unter Windows

- Das **.NET-Framework** steht ausschließlich für Windows zur Verfügung.
- **.NET Core**
  - Open-Source-Projekt unter Führung von Microsoft
  - *CoreFX* enthält Basisfunktionalität der .NET-Framework-Bibliothek
  - *CoreCLR* ist die Laufzeitumgebung
  - Unterstützte Plattformen: Linux, Mac OS X, Windows
- **Mono** umfassender als .NET Core

  WPF nicht enthalten - nicht ganzes Full Framework aber mehr als Core
  - Open-Source-Projekt,
  - Laufzeitumgebung zu .NET kompatibel,
  - stellt große Teile der Funktionalität des .NET-Framework zur Verfügung
  - Unterstütze Plattformen: Linux, Mac OS X, Windows
- Die **Xamarin**-Plattform
  - basiert auf Mono und
  - ermöglicht die Entwicklung von nativen mobilen Anwendungen für iOS, Android und Windows Phone.
  - Xamarin wurde Anfang 2016 von Microsoft übernommen.

.NET Core

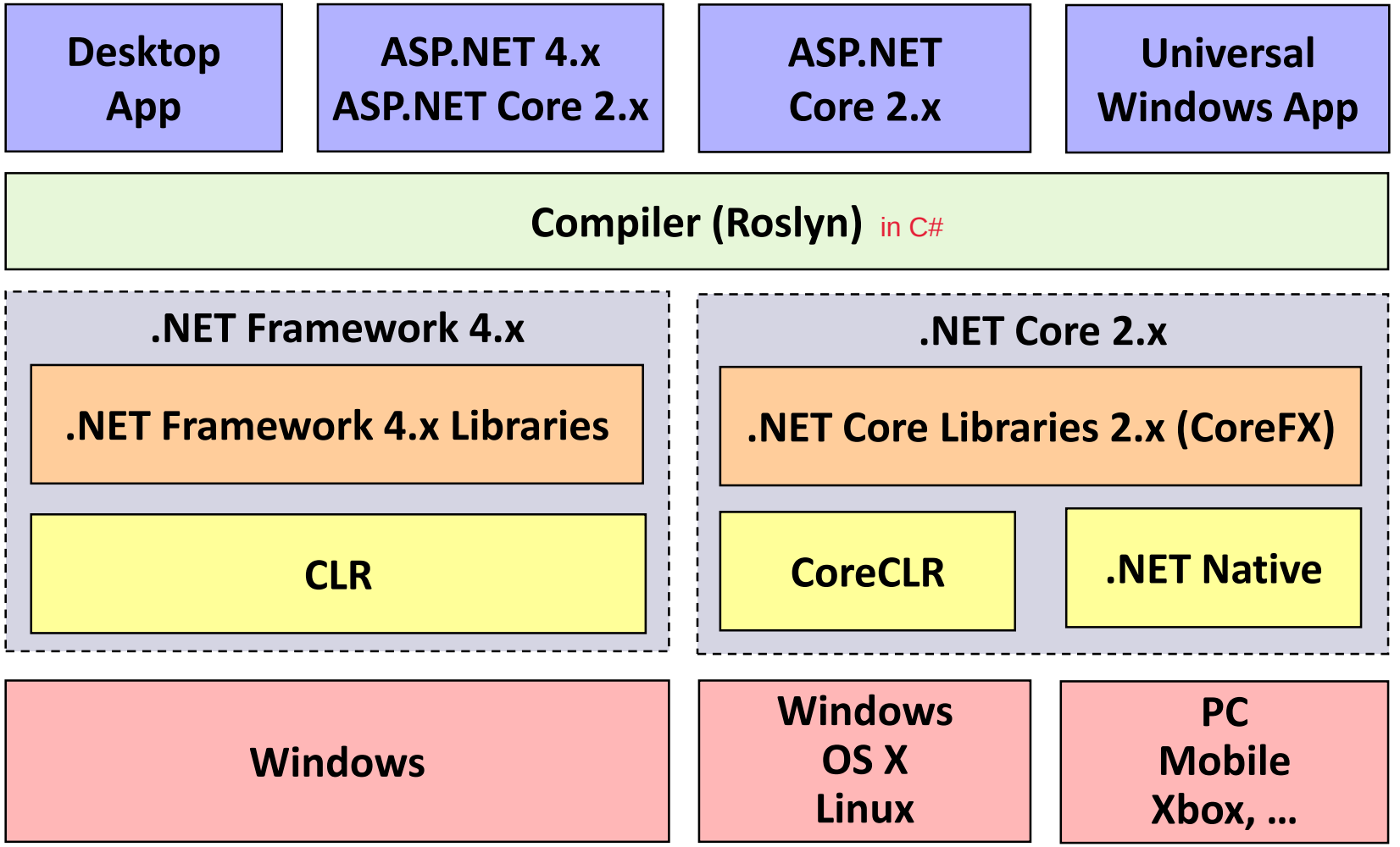
Portable class libraries can help you reduce the time and costs of developing and testing code. Use this project type to write and build portable .NET Framework assemblies, and then reference those assemblies from apps that target multiple platforms such as the .NET Framework

- Motivation
  - Verschiedene Varianten des .NET-Frameworks für Desktop, Store Apps, Windows Phone.
  - Entwicklung Framework-übergreifender Anwendung ist schwierig (→ Portable Class Libraries).
  - Maschinenweite Installation: Verschiedene Versionen beeinflussen sich gegenseitig.

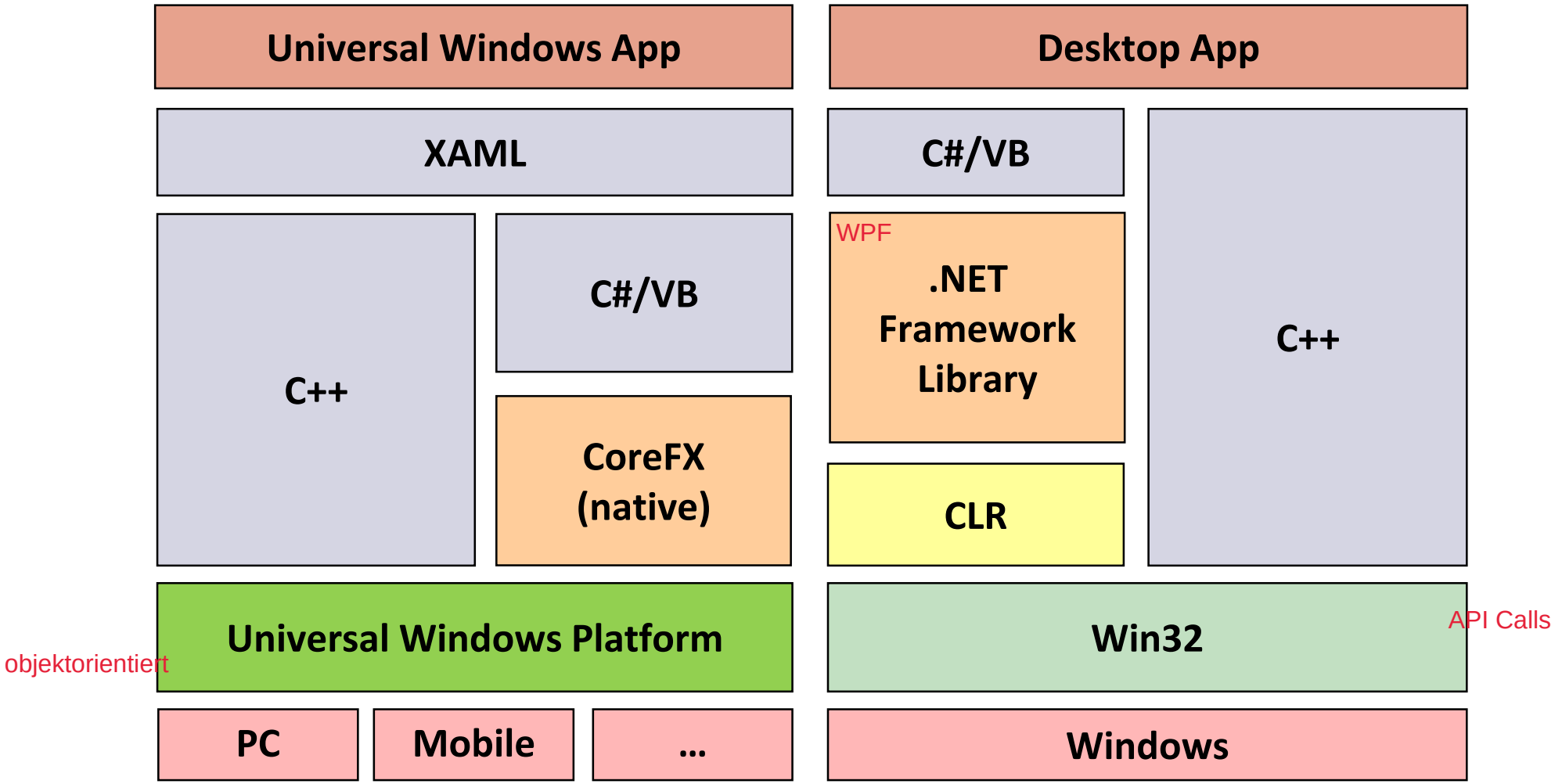
bestimmte Laufzeitversion - verursacht Probleme

- Eigenschaften
  - Plattformübergreifende Implementierung
  - Gemeinsame Codebasis für unterschiedliche Anwendungsgebiete (Windows Apps, Web-Anwendungen)
  - Komponenten werden über Nuget bereitgestellt nuget - Packet Manager - modular
  - Häufigere Releasezyklen (4 Snapshots pro Jahr), einzelne Assemblys können ausgetauscht werden.
  - Anwendung und Assemblys können gemeinsam deployt werden.

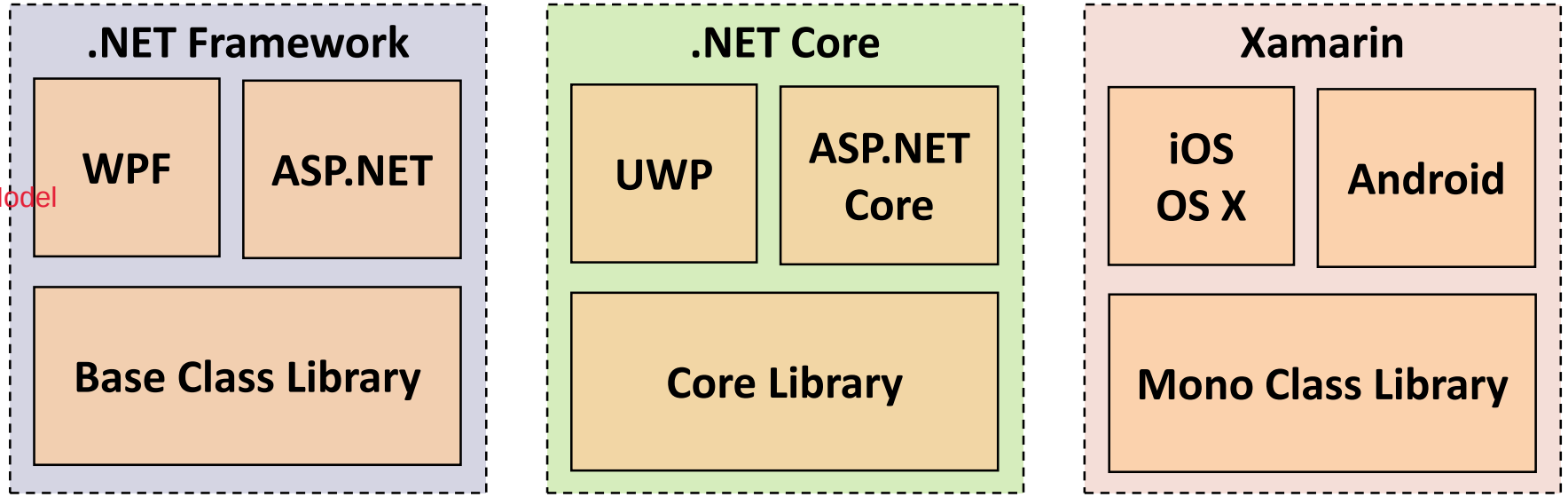
Die .NET-Plattform



.NET Framework und die UWP (Windows 10)

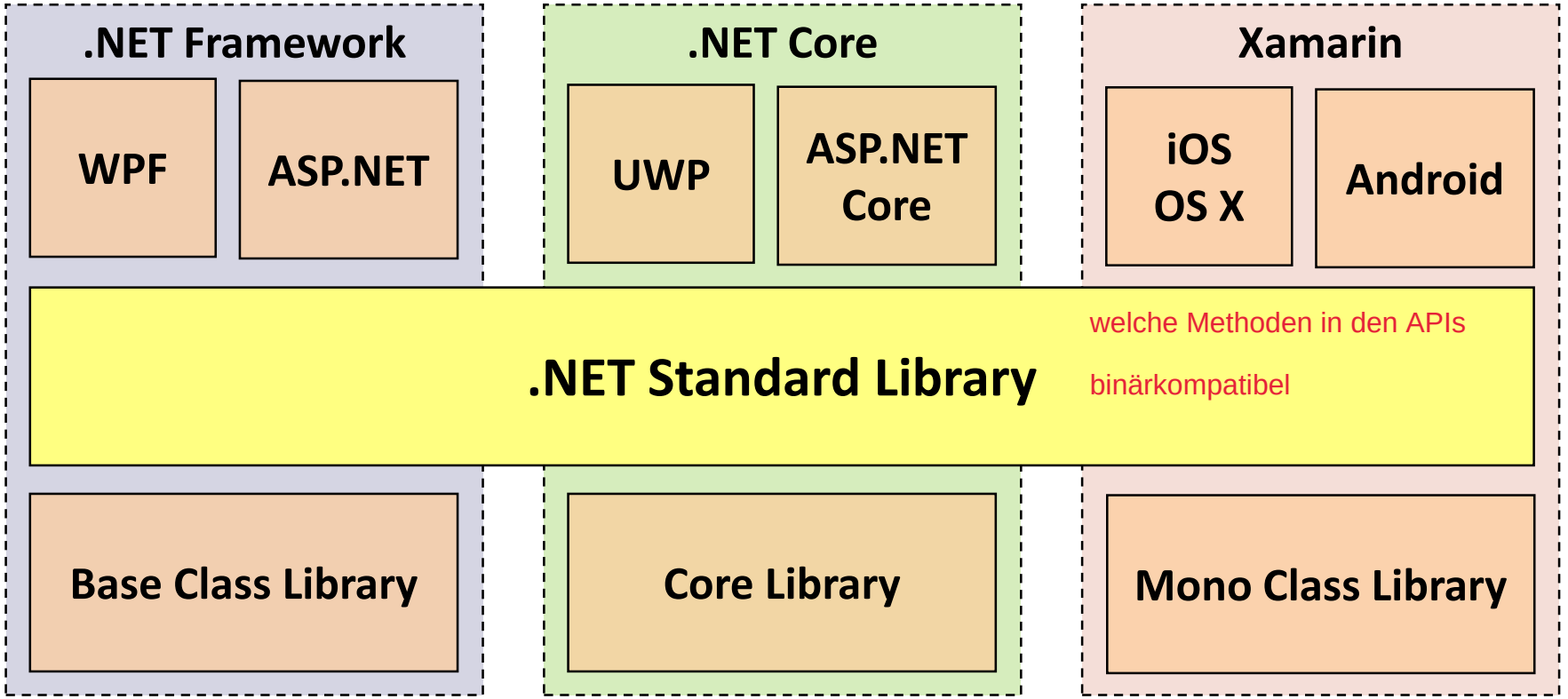


.NET Standard (1)



- Es existieren viele verschiedene Varianten von .NET.
- Portieren von Code ist aufwändig, da sich auch die Basis-Bibliotheken unterscheiden.
- Portable Class Libraries (PCL) sind unbefriedigend, da Entwickler auf größte gemeinsame Funktionalität eingeschränkt ist.

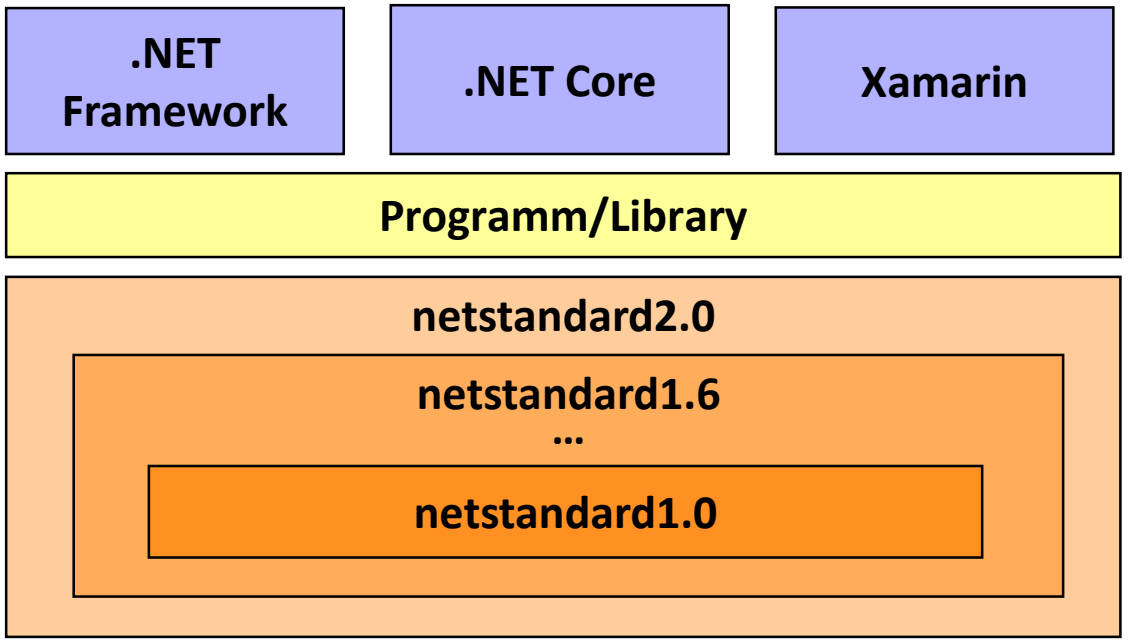
.NET Standard (2)



- Die *.NET-Standard-Library* ist eine Spezifikation von .NET-APIs, die von mehreren Laufzeitumgebungen unterstützt werden.

.NET Standard (3)

- Gemeinsame APIs entstehen nicht zufällig, sondern im Rahmen eines Standardisierungsprozesses.
- Lineare Versionierung:
  $1.0 \subset 1.2 \subset \ldots \subset 2.0$
- Binärkompatible Komponenten



- Kompatibilitätsmatrix:

Komponenten eher niedriger ??

Plattform-Name	Alias								
.NET Standard	netstandard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0
.NET Core	netcoreapp	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
.NET Framework	net	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1
Mono		4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4
Universal Windows Platform	uap	10.0	10.0	10.0	10.0	10.0	10.0. 16299	10.0. 16299	10.0. 16299

Version 4.6.1 vom .NET Framework unterstützt .NET Standard 2.0

NET Framework Compatibility Mode

nuget: Sammlung verschiedener Pakete von vielen Frameworks

- Rückwärtskompatibilität
  - Anwendungen, die mit Framework x entwickelt wurden, funktionieren mit Laufzeitumgebung von Framework y (y > x).
- Full Framework
  - Anwendung läuft auf jenem Framework, mit dem es entwickelt wurde.
  - Mehrere Frameworks können parallel installiert sein („side-by-side execution“)
  - Anwendungen für .NET-Framework ≥ 4.5 sind rückwärtskompatibel.
- .NET Standard 2.0 → **NET Framework Compatibility Mode**
  - Beliebige *binäre* .NET-Komponenten können referenziert werden.
  - Warnung, dass Komponente möglicherweise nicht kompatibel ist.
  - Viele NuGet-Komponenten können unverändert verwendet werden.
  - Komponenten, die nicht unterstützte Bibliotheken verwenden (z. B. WPF) verursachen einen Laufzeitfehler.

Common Language Runtime (CLR) (1)

- Jedes Framework enthält eine Implementierung der Laufzeitumgebung:
  - Full Framework → *CLR* CLR hat ISO Standard
  - .NET Core → *CoreCLR*
- CLR führt .NET-Anwendungen aus: Aufgaben: Code ausführen
  - führt Sicherheitsüberprüfungen aus.
  - übernimmt Speicherverwaltung und Fehlerbehandlung.
  - lädt dynamisch Komponenten (richtige Version).
- CLR stellt Verbindung zum Betriebssystem her.
- CLR versteht eine Zwischensprache , in die alle .NET-Programme übersetzt werden→ IL. Intermediate Language
- Die Typen der Programme müssen sich an gewisse Spielregeln halten→ CTS.
- Zwischensprache und Informationen über Programme werden in Assemblys verpackt.

Common Language Runtime (CLR) (2)

- CLR ist eine *virtuelle Maschine* = mit Software realisierter Prozessor.
- Vorteile: zur Laufzeit kennt man Hardwarespezifikationen (Prozessor)
  - Plattformunabhängigkeit: CLR kann auf andere reale Maschinen portiert werden.
  - Sprachunabhängigkeit: Compiler übersetzen in Sprache der CLR.
  - Kompakter Code.
  - Optimierter Code: CLR kann Spezifika der Zielmaschine berücksichtigen.
- CLR ist eine Stackmachine.
  - Einfache Codegenerierung

Unterschiede zur Java Virtual Machine

- Kompilation/Interpretation
  - Bytecode wird von JVM interpretiert (Bytecode ist darauf ausgelegt).
  - Hotspot-JVMs übersetzen Teile des Bytecodes, wenn dieser häufig aufgerufen wird.
  - CLR übersetzt Zwischencode immer → JIT-Compiler.
- Unterstützung verschiedener Sprachen und –Paradigmen (OO, prozedurale, funktionale Sprachen)
- Selbstdefinierte Wertetype (Strukturen)
- Aufruf per Referenz
- Typsichere Methodenzeiger (Delegates)

in nativen Code

methodenweise (alle Methoden die aufgerufen werden)

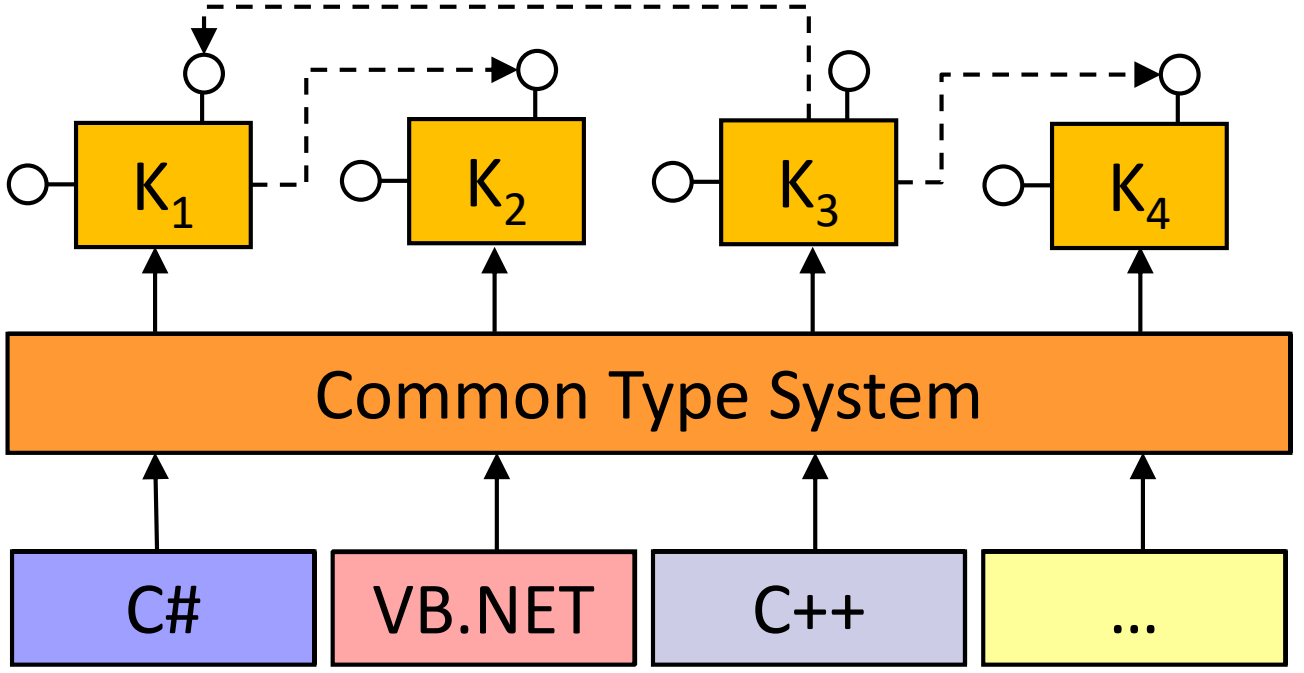
Komponenten der CLR

- *CTS:* Common Type System
- *CLS:* Common Language Specification — Typen sprachübergreifend verwendbar
- *CIL:* Common Intermediate Language — IL - C wegen Standard
- *JIT:* Just in Time Compiler
- *VES:* Virtual Execution System — führt Code aus

Common Type System (CTS)

- CTS legt fest, wie Typen im Speicher repräsentiert werden:
  - Objektorientiertes Programmiermodell,
  - ermöglicht sprachübergreifende Verwendung von Typen.

Binärlayout festgelegt



Komponenten

stellt sicher das man selbst Typen erstellen kann und diese in verschiedenen Sprachen dann verwenden kann

.NET Compiler

Common Type System – Beispiel

VB.NET

```
Public Class Person
  Private name As String

  Public Sub New(ByVal n
                 As String)
    name = n
  End Sub

  Public Function
      GetName() As String
    GetName = name
  End Function
End Class
```

vbc /target:library Person.vb

mit idlasm kann man in bspw. .dll hineinsehen (Klassen, Methoden, ...)

C#

```
public class Student : Person {
  private string id;

  public Student(string name,
                 String id):
    base(name) {
    this.id = id;
  }

  public string GetId() {
    return id;
  }
}
```

csc /r:Person.dll
/target:exe Student.cs

Common Language Specification (CLS)

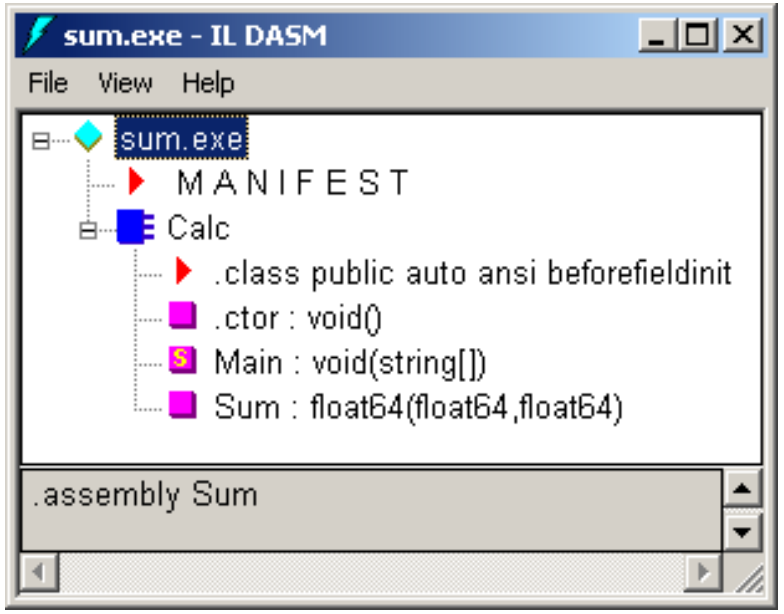
- Programmierrichtlinien für *Entwickler*, die garantieren, dass Klassen *sprachübergreifend* eingesetzt werden können:
  - Als Parametertypen bei öffentlichen Methoden darf nur eine Untermenge der CTS-Standardtypen verwendet werden.
  - Groß-/Kleinschreibung nicht ausnutzen.
  - Verschiedene Namen für Methoden und Felder.
- Regeln, die *Compilerbauer* einhalten müssen:
  - Vorschriften, gewisse Metadaten zu generieren.
- Compiler kann CLS-Konformität überprüfen.

Intermediate Language (IL)

- Zwischencode, der von allen .NET-Compilern erzeugt wird.
- Zwischencode wird von CLR ausgeführt.
- IL entspricht dem Bytecode von Java.
- C++-Compiler kann IL- oder nativen Code generieren.
- Andere Bezeichnungen:
  - CIL: Common Intermediate Language (ECMA, ISO).
  - MSIL: Microsoft Intermediate Language.

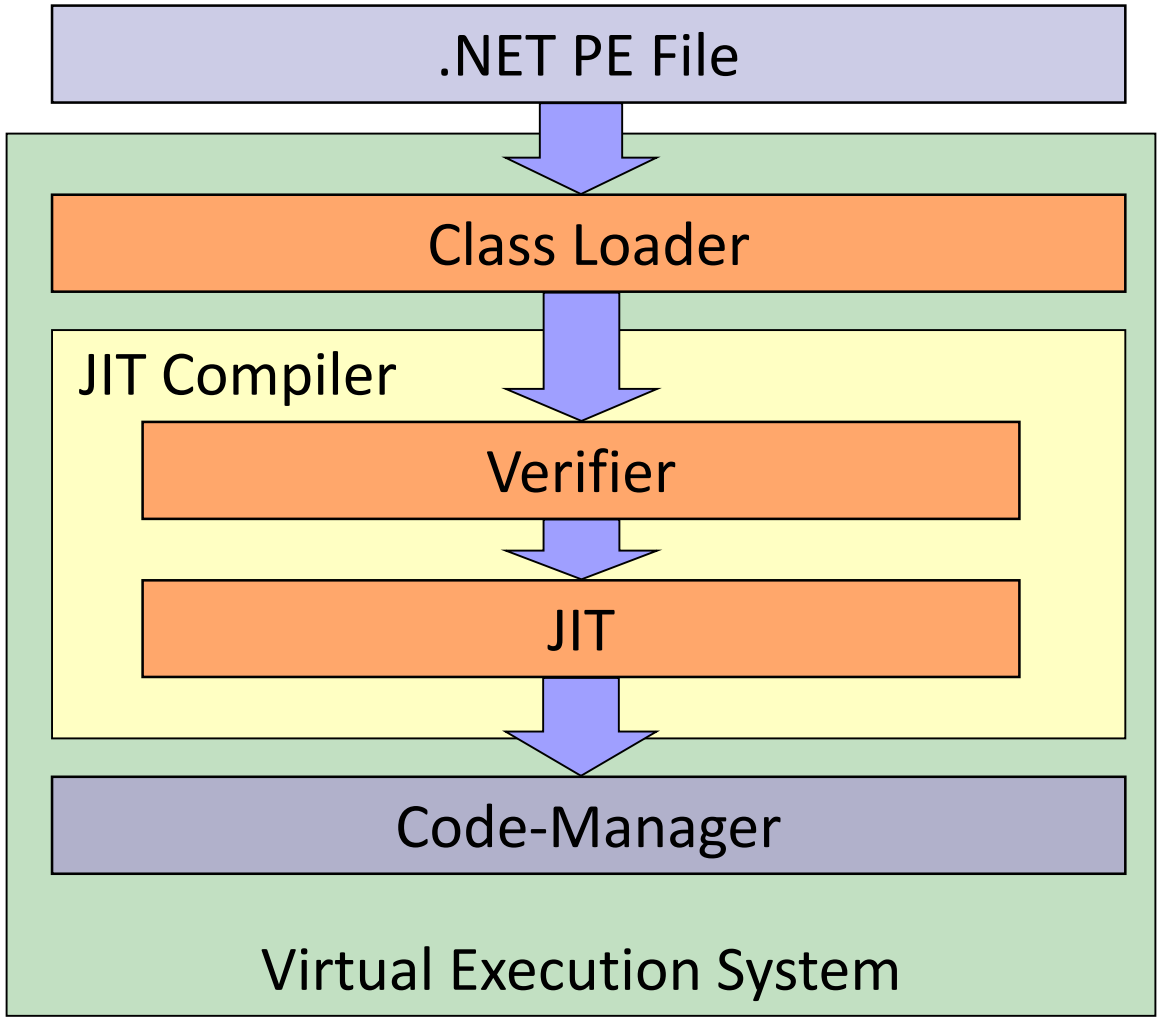
IL DASM

```
public class Calc {
  public  double Sum(
    double d1, double d2) {
     return d1+d2;
  }
  …
}
```




```
.method public hidebysig instance
float64 Sum(float64 d1,
            float64 d2) cil managed {
  // Code size       8 (0x8)
  .maxstack  2
  .locals init (float64 V_0)
  IL_0000:  ldarg.1
  IL_0001:  ldarg.2
  IL_0002:  add
  IL_0003:  stloc.0
  IL_0004:  br.s IL_0006
  IL_0006:  ldloc.0
  IL_0007:  ret
} // end of method Calc::Sum
```

Virtual Execution System (VES)



typ wird nur geladen wenn diese referenziert werden

Aufgaben des VES (1)

- Class-Loader
  - Suchen der Assemblys (Arbeitsverzeichnis oder GAC).
  - Vorbereitung zur Ausführung: Einfügen von Stubs für JIT-Kompilierung.
- Verifier:
  - Überprüfung der Typsicherheit
    - Code darf nur auf berechtige Bereiche zugreifen.
    - Objekte werden nur über deren Schnittstelle angesprochen.
  - Für typsichere Assemblys kann garantiert werden, dass sie sich nicht gegenseitig beeinflussen (→ *Application Domains*).

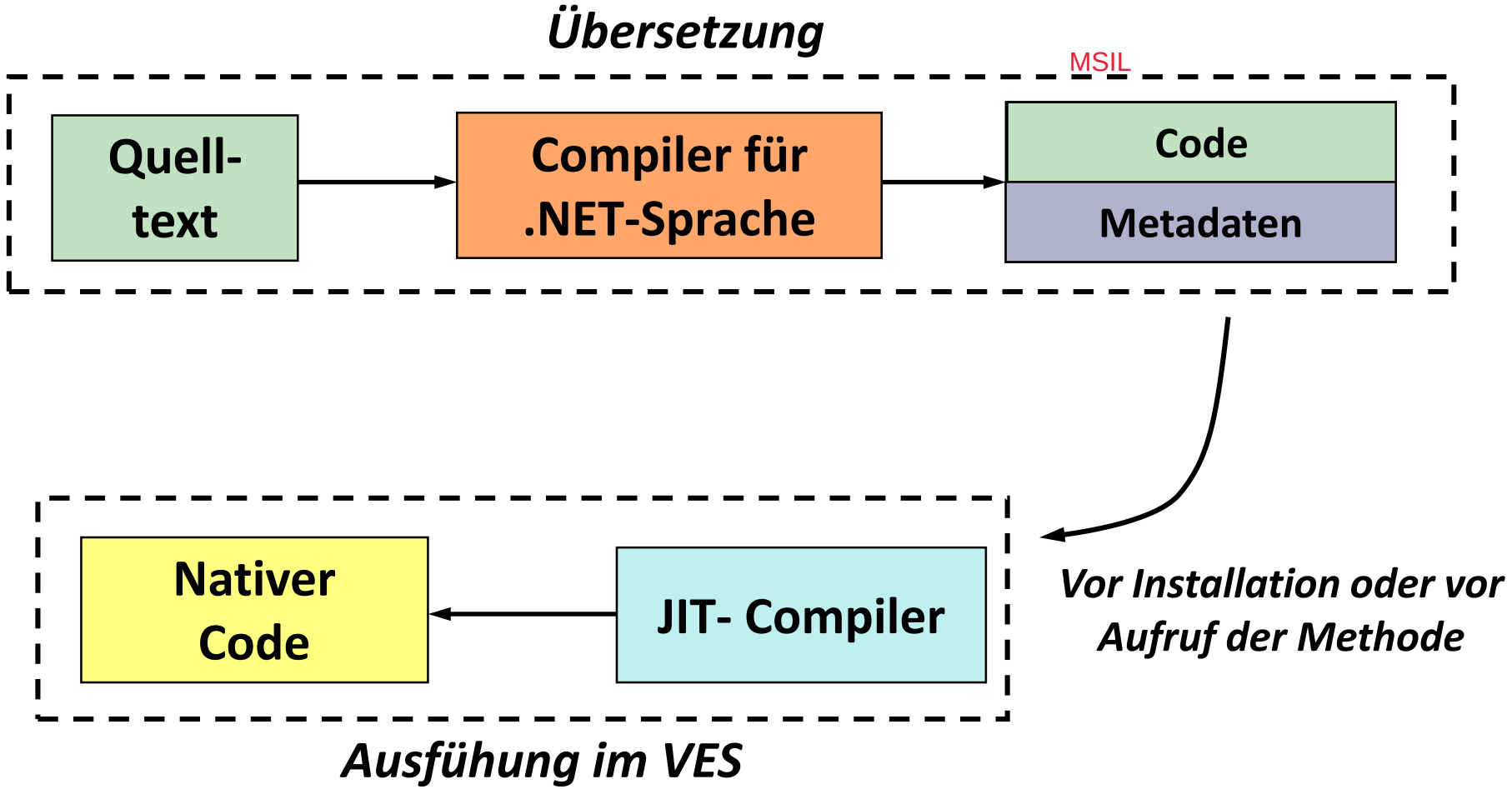
Aufgaben des VES (2)

- Kategorien von IL-Code
  - Ungültiger Code
    - Code enthält faschen IL-Code.
  - Gültiger Code
    - Code kann nicht typsichere Anweisungen (kann durch Zeigerarithmetik entstehen) enthalten.
  - Typsicherer Code
    - Objekte halten sich an Schnittstellen.
  - Verifizierbarer Code
    - Typsicherheit kann bewiesen werden.
    - Nicht verifizierbarer Code kann aber typsicher sein.
  - Manche Compiler garantieren die Erzeugung von typsicherem Code (C#, nicht C++).

Aufgaben des VES (3)

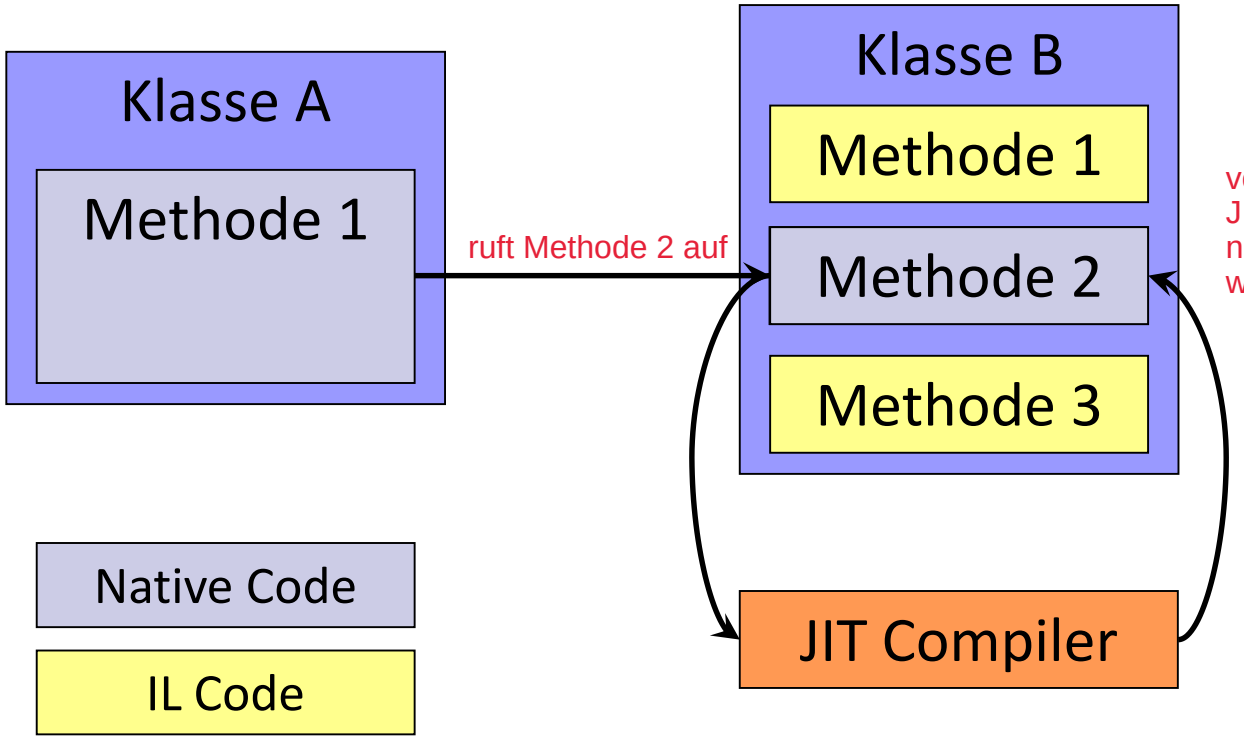
- JIT: Übersetzung von IL- in nativen Code.
- Code-Manager: Ausführung des nativen Codes
  - Garbage Collection von „Managed Types“,
  - Ausnahmebehandlung,
  - Security:
    - Code-Access Security: Berechtigungen sind von der „Herkunft“ des Codes abhängig.
    - Große Veränderungen in .NET 4.0
  - Debugging und Profiling,
  - Platform Invoke (P/Invoke):
    - Aufruf von Win32 API-Funktionen

Übersetzung und Ausführung



Just in Time Compiler (JIT)

- IL-Code wird immer kompiliert.
- Code-Generierung bei erstem Methodenaufruf.



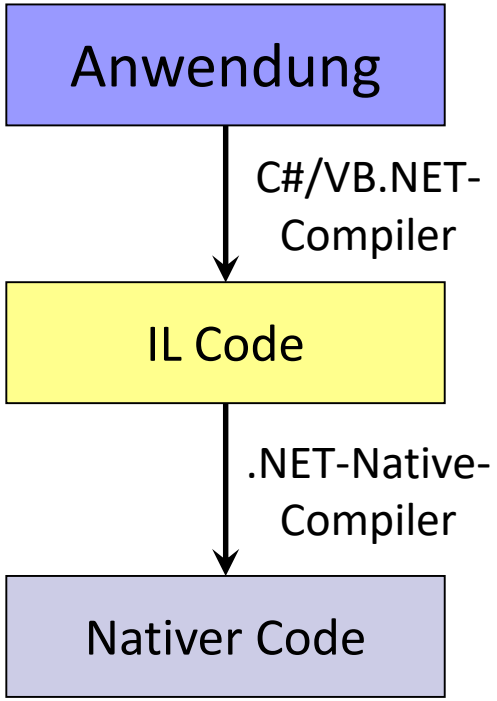
vorher gelb (nicht von JIT übersetzt)
JIT übersetzt dann Methode 2 und liegt dann in nativem Code vor (falls später nochmal aufgerufen wird nativer Code ausgeführt)

IL-Code wird durch *Maschinencode (nativem Code)* ersetzt.

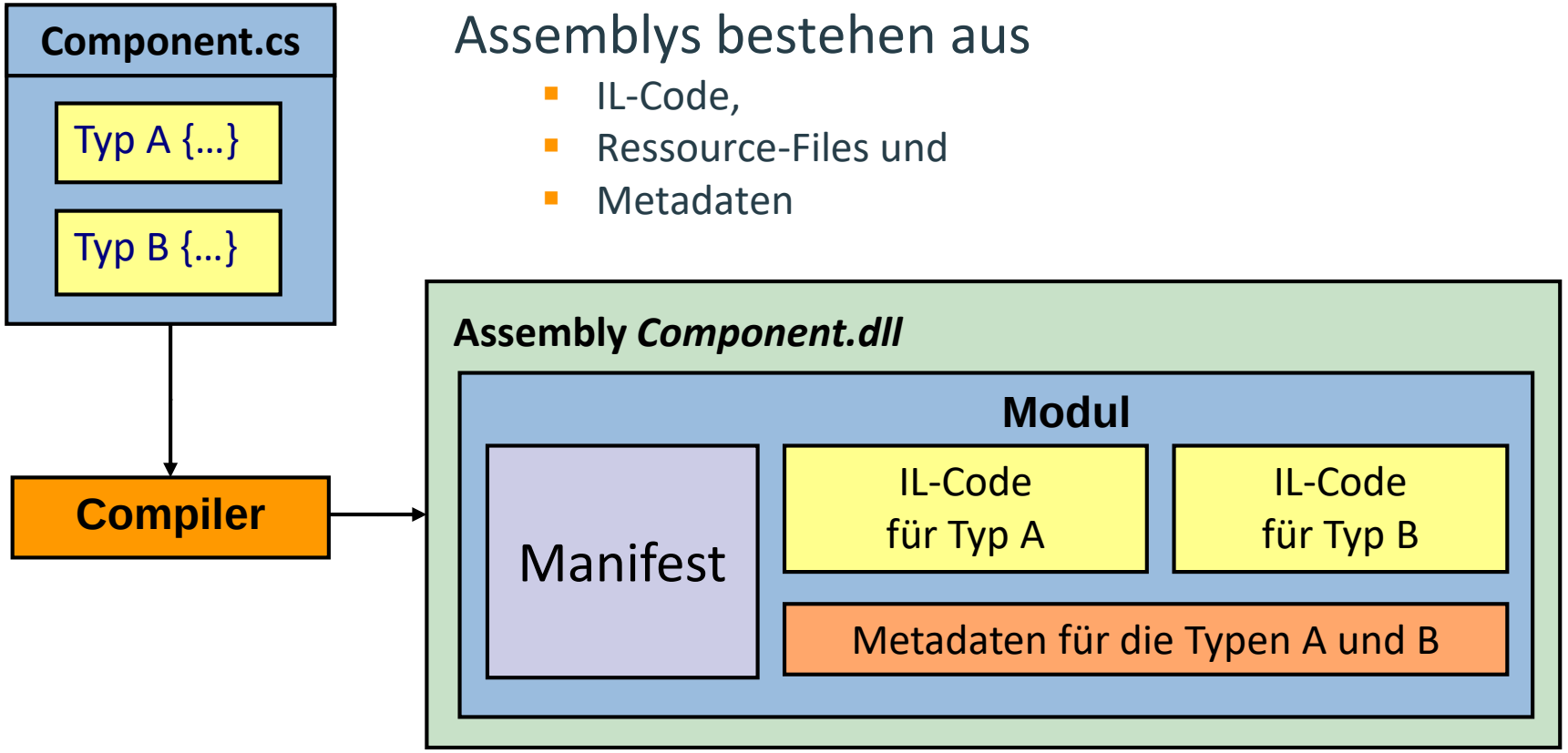
wenn UWP im Store wird Code in Cloud kompiliert

.NET Native (.NET Core)

- Der JIT-Compiler übersetzt IL-Code zur Laufzeit des Programms → Just In Time.
- Mit .NET Native kann IL-Code bereits zur Übersetzungszeit des Programms in nativen Code kompiliert werden.
- .NET Native nutzt das Compiler-Backend von C++.
- Vorteile
  - Schnellere Ausführungszeiten
  - Schnellerer Programmstart
  - Geringerer Hauptspeicherbedarf
  - Kleinere Deployment-Pakete
- Produktiv wird .NET Native dzt. nur für Universal-Windows-Plattform-Apps (Windows 10) genutzt.



Assemblys



Assemblys bestehen aus

- IL-Code,
- Ressource-Files und
- Metadaten

Managed/Unmanged Code/Types

- *Managed Code* wird von der CLR ausgeführt.
- *Unmanaged Code* wird direkt vom Prozessor ausgeführt.
- Alle .NET-Sprachen, außer C++, werden in *Managed Code* übersetzt.
- C++-Code kann entweder in nativen oder *Managed Code* übersetzt werden.
- *Managed Code* darf nicht mit *Managed Types* verwechselt werden. Nur *Managed Types* werden vom Garbage Collector automatisch freigegeben.
- Nur *Managed Types* können über Assembly-Grenzen hinweg verwendet werden.

Managed/Unmanaged Code

Person.cpp

```
class Person {
};

Person* p = new Person;
P->SetAge(20);
delete p;
```

Person.cs

```
public class Person {
}

Person p = new Person();
p.SetAge(20);
```

cl Person.cpp

cl **/clr** Person.cpp

csc /t:exe Person.cs

```
push  4
call  operator new (41117Ch)
add   esp,4
mov   dword ptr [ebp-0E0h],eax
mov   eax,dword ptr [ebp-0E0h]
mov   dword ptr [p],eax
push  14h
…
```

Unmanaged Code

```
L_0000: newobj instance void
        Person::.ctor()
L_0005: stloc.0
L_0006: ldloc.0
L_0007: ldc.i4.s 20
L_0009: callvirt instance void
        Person::SetAge(int32)
L_000e: …
```

Managed Code

Managed/Unmanaged Types

PersonUM.cpp

```
class Person {
};

Person* p =
  new Person;
P->SetAge(20);
delete p;
```

cl **/clr** PersonUM.cpp

Virtual Execution System

(Person)

Unmanaged Type
Managed Code

Person.cs

```
class Person {
}

Person p =
  new Person();
p.SetAge(20);
```

csc /t:exe Person.cs

PersonM.cpp

```
ref class Person {
};

Person^ p =
  gcnew Person;
p->SetAge(20);
```

cl **/clr** PersonM.cpp

Virtual Execution System

Garbage Collector

(Person)

Managed Type
Managed Code