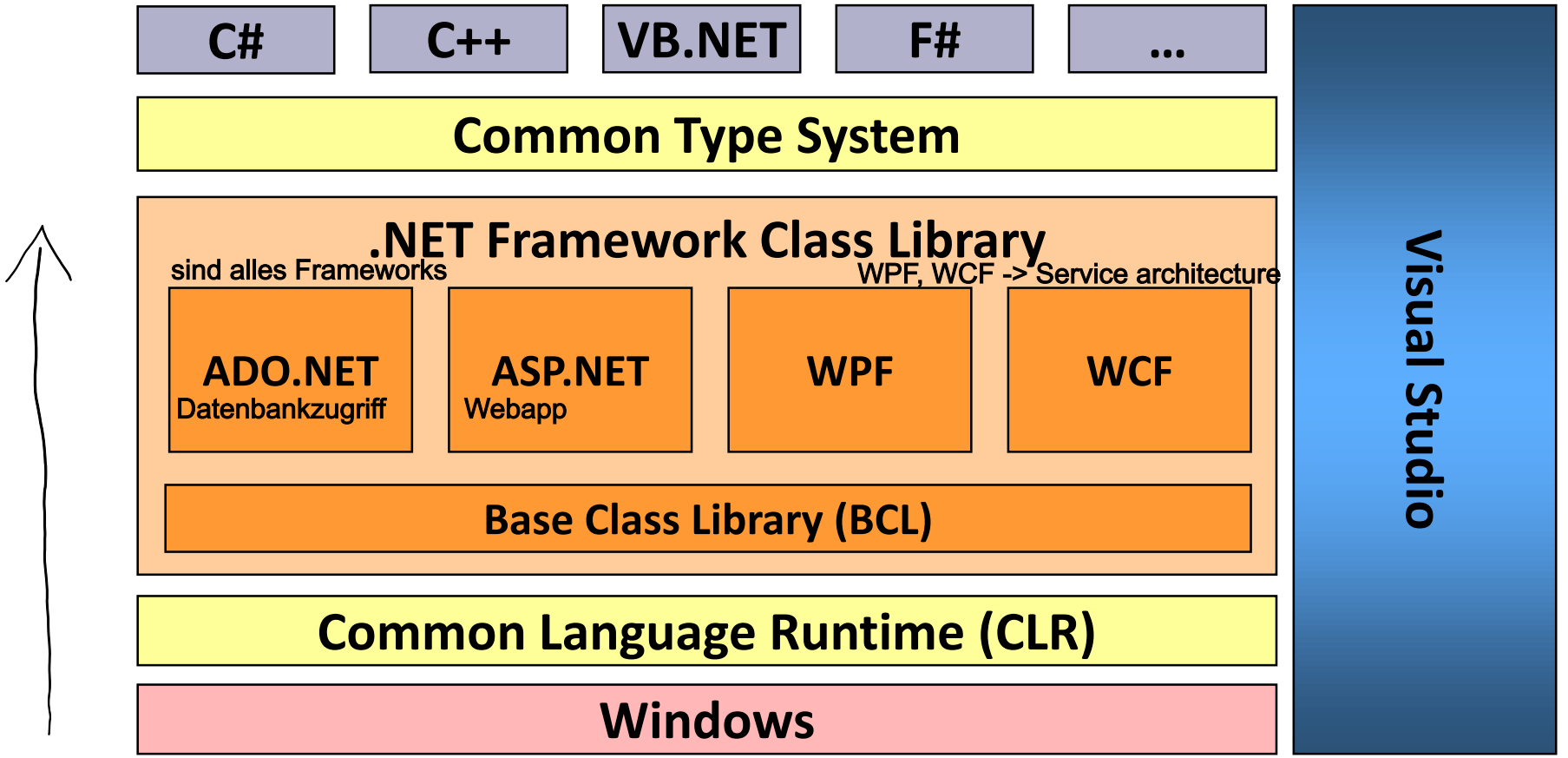
.NET: Architektur

Version 5.8

.NET Framework 2.0 – 4.7 (Full Framework)



Windows gehört nicht zum Framework; CLR schon

CLR führt den Code aus
Serialisierung,
Objektarrays in der Base Class Library

gibt Compilerplattform für alle

Varianten von .NET

- Das **.NET-Framework** steht ausschließlich für Windows zur Verfügung.
- **.NET Core** in den letzten 2-3 Jahren entwickelt
  - Open-Source-Projekt unter Führung von Microsoft
  - *CoreFX* enthält Basisfunktionalität der .NET-Framework-Bibliothek Gegenstück zu Class Framework
  - *CoreCLR* ist die Laufzeitumgebung
  - Unterstützte Plattformen: Linux, Mac OS X, Windows
- **Mono** seit 2001
  FullFramework als OpenSource, WPF ist nicht drinnen, Windows Forms zb schon
  - 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

- 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.

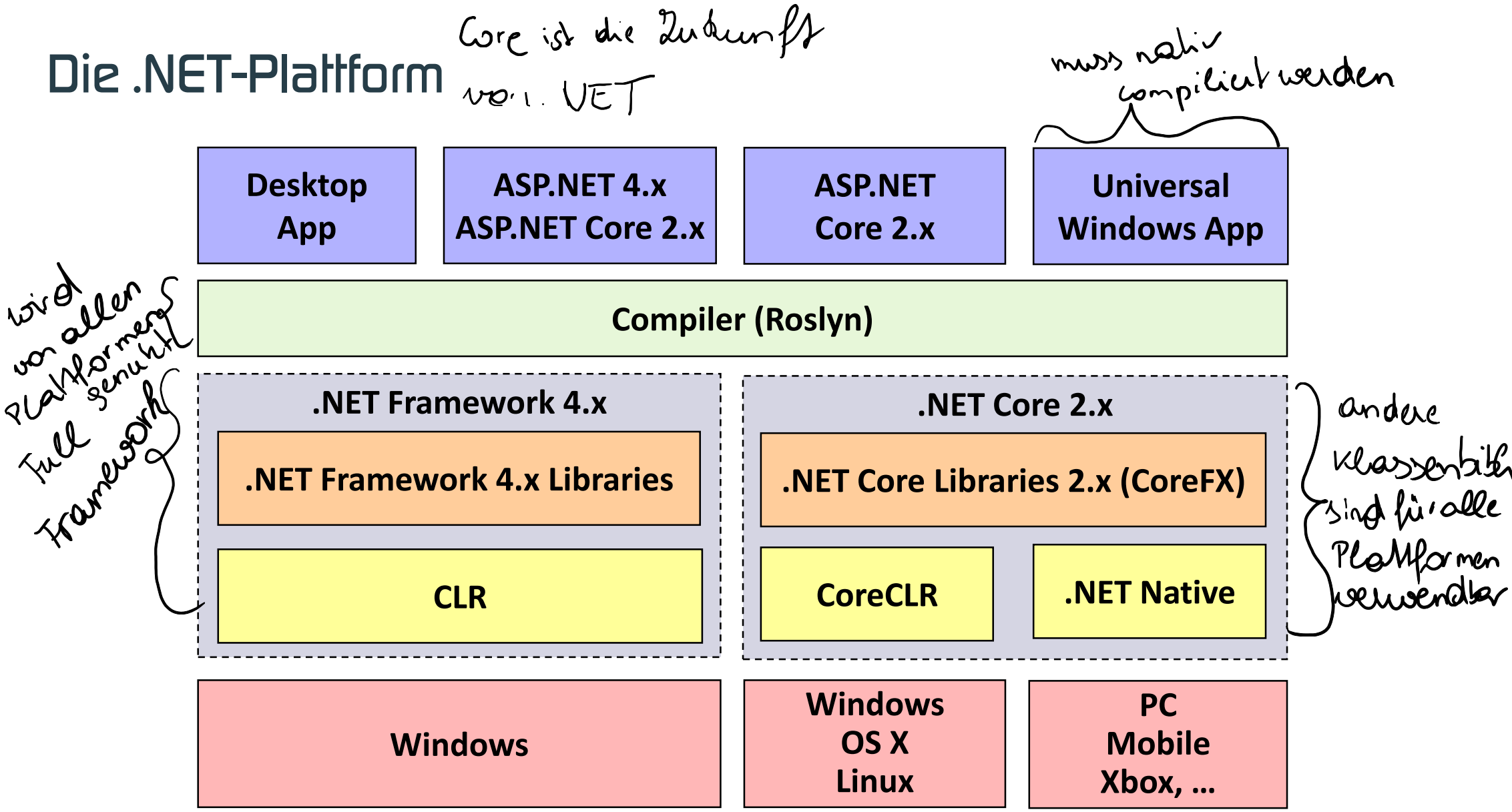
Entwickler soll nicht mehr von der Version abhängig sein.

- Eigenschaften

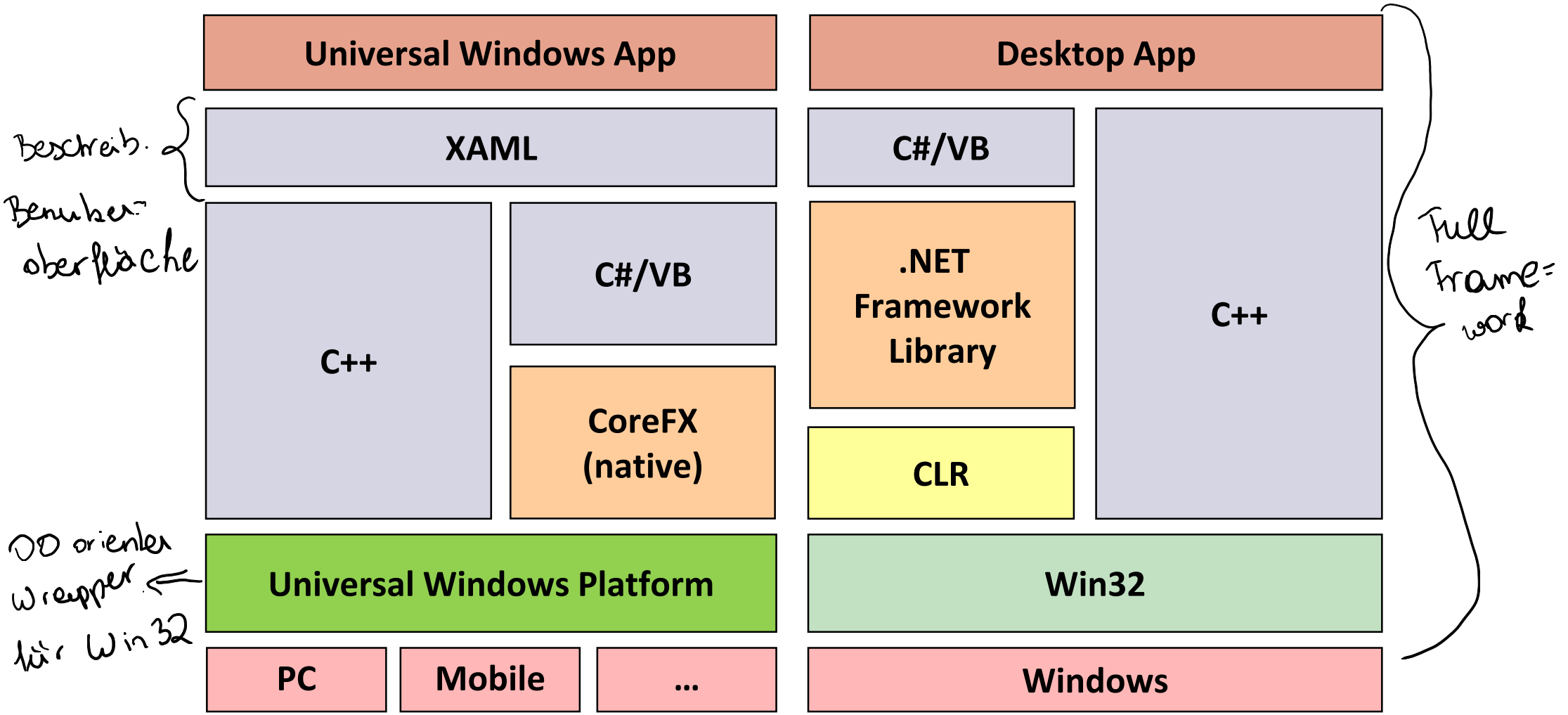
nuget ist er packagemanager für .NET

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

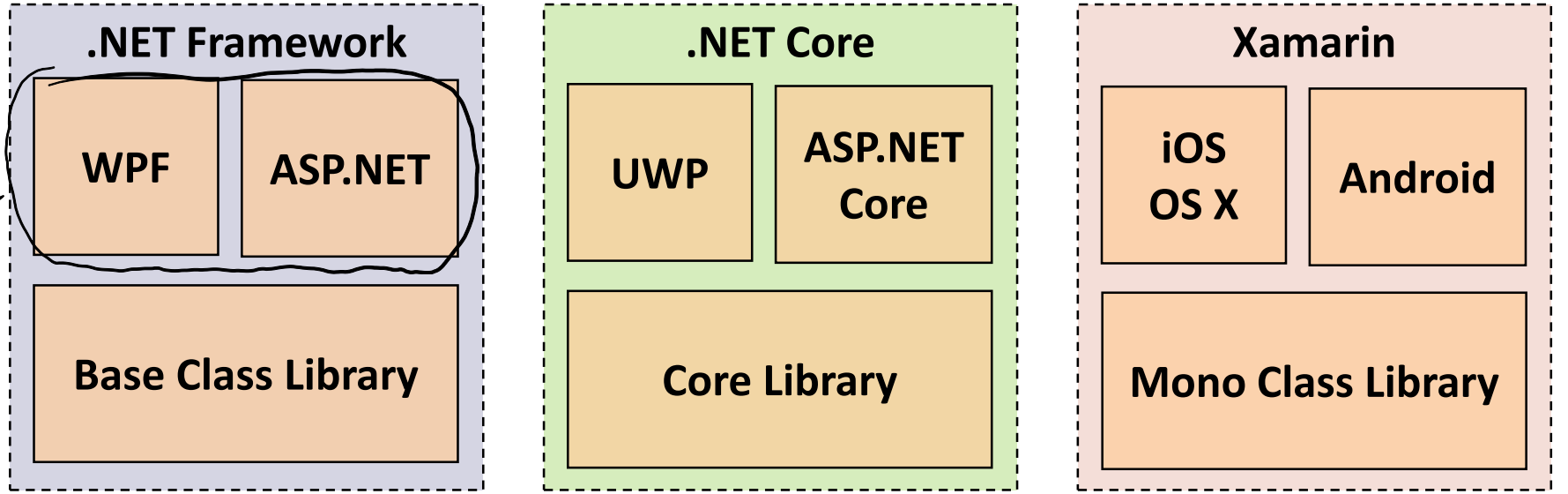
Die .NET-Plattform



.NET Framework und die UWP (Windows 10)

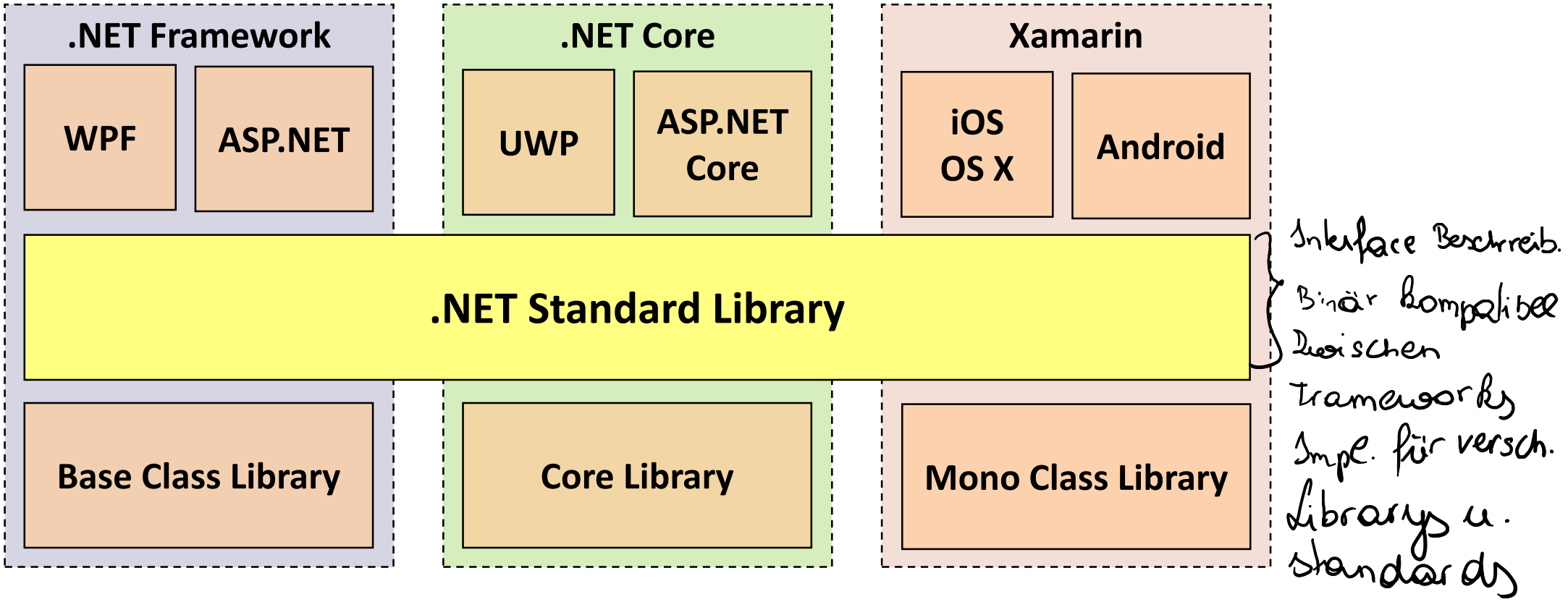


.NET Standard (1)



- 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.

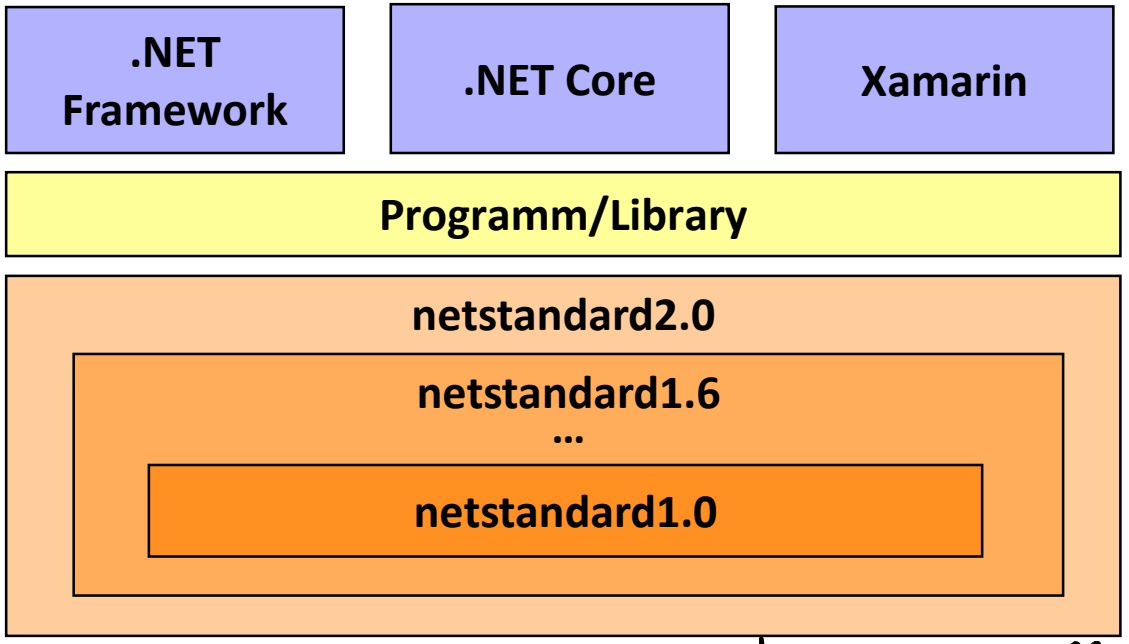
.NET Standard (2)



- 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:

nicht immer neuste Version verwenden -> wird nicht von allen unterstützt

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

unterstützt

Vereinfachung plattformübergreifende Entwicklung

NET Framework Compatibility Mode

- 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.

überprüft dll ob funktioniert

alle DLL's können somit unter Linux, ... verwendet werden auch wenn sie für Windows entwickelt wurde

Common Language Runtime (CLR) (1) gibt ISO-Standard dazu

- Jedes Framework enthält eine Implementierung der Laufzeitumgebung:
  - Full Framework → *CLR*
  - .NET Core → *CoreCLR*
- CLR führt .NET-Anwendungen aus:
  - 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.
- 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:
  - 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)

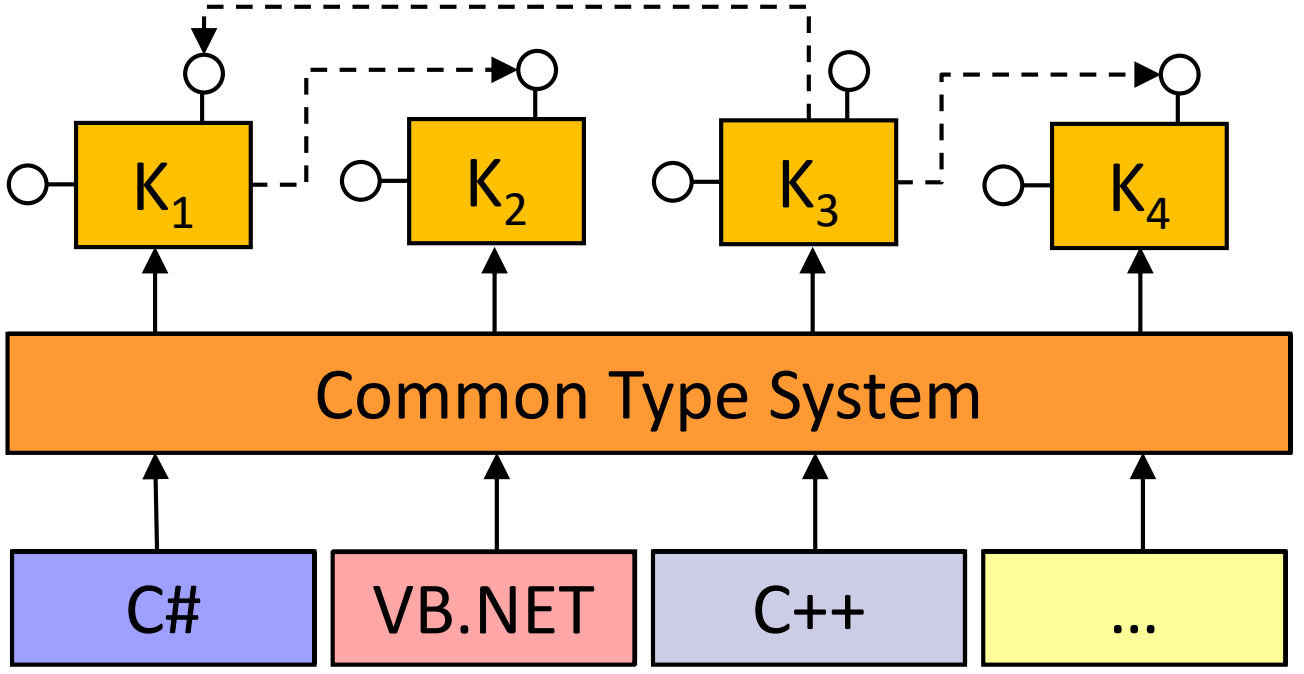
JVM: Hotspot compiler übersetzt Hotspots in nativen Code

Komponenten der CLR

- *CTS:* Common Type System
- *CLS:* Common Language Specification damit Typen Sprachübergreifend
- *CIL:* Common Intermediate Language
- *JIT:* Just in Time Compiler
- *VES:* Virtual Execution System

Common Type System (CTS)

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



Komponenten

stellt sicher dass man Komponenten in anderen Sprachen auf verwendet werden können

.NET Compiler

Common Type System – Beispiel

über references können andere Sprachen hinzugefügt

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

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)

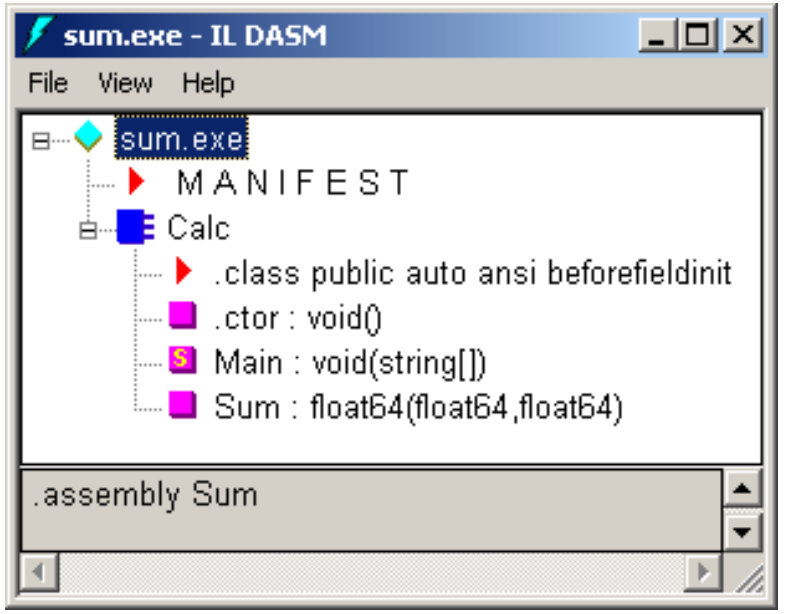
- Programmierrichtlinien für *Entwickler*, die garantieren, dass Klassen *sprachübergreifend* eingesetzt werden können:
  - Als Parametertypen bei öffentlichen Methoden darf nur eine Untermenge der CTS-Standardtypen verwendet werden.
  - Groß-/Kleinschreibung nicht ausnutzen.
  - Verschiedene Namen für 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
```

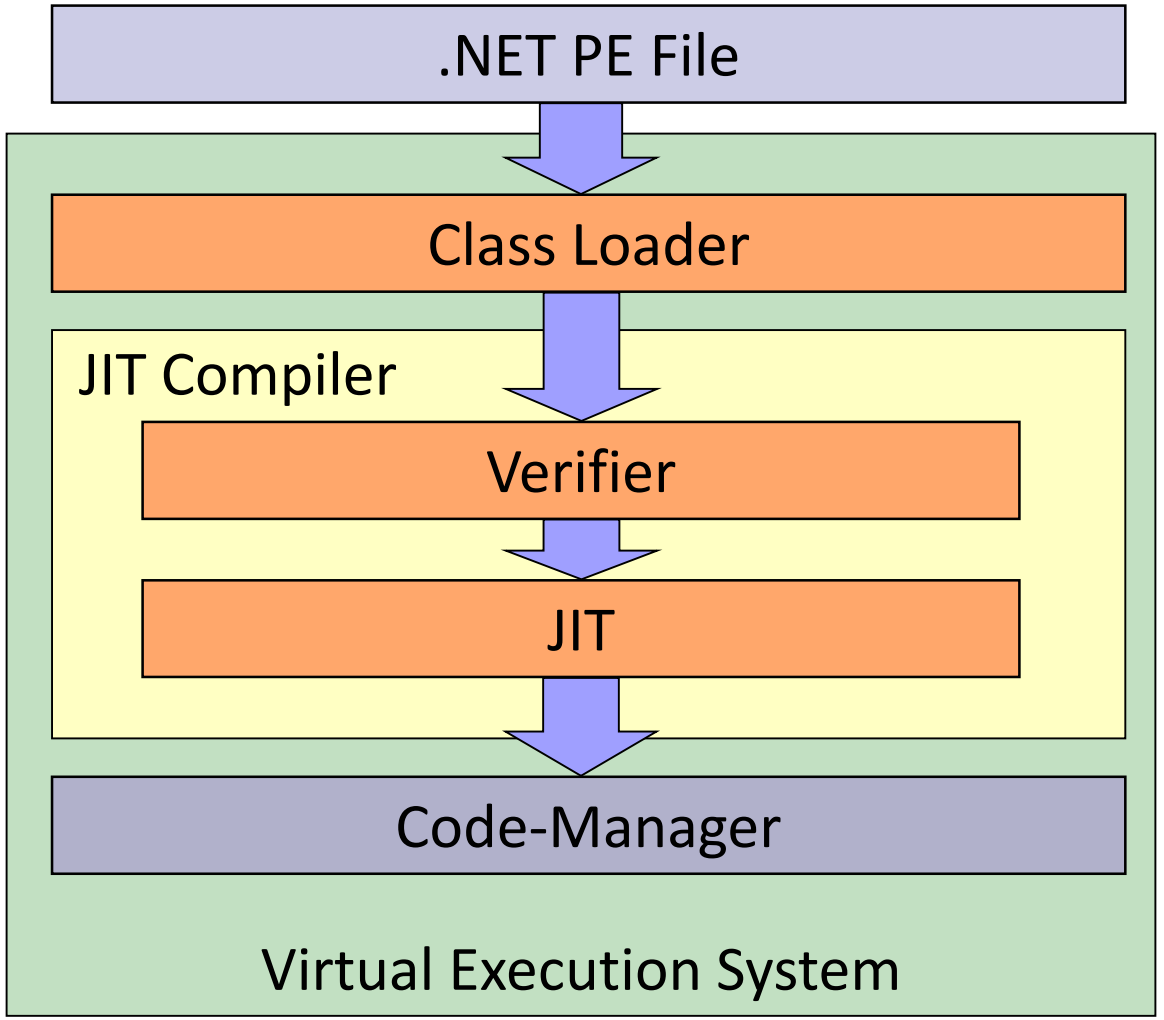
geht noch kürzer mit optimierter version gibt es nur;
ldarg.0 ldarg.1 add ret

Virtual Execution System (VES)

kommet eine exe heraus kann es also als Windows Programm verwendenden

Verifier werden Typen vertragskonform verwendet (bufferoverflows zb)

JIT wird nur gemacht wenn Methode aufgerufen wird



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)

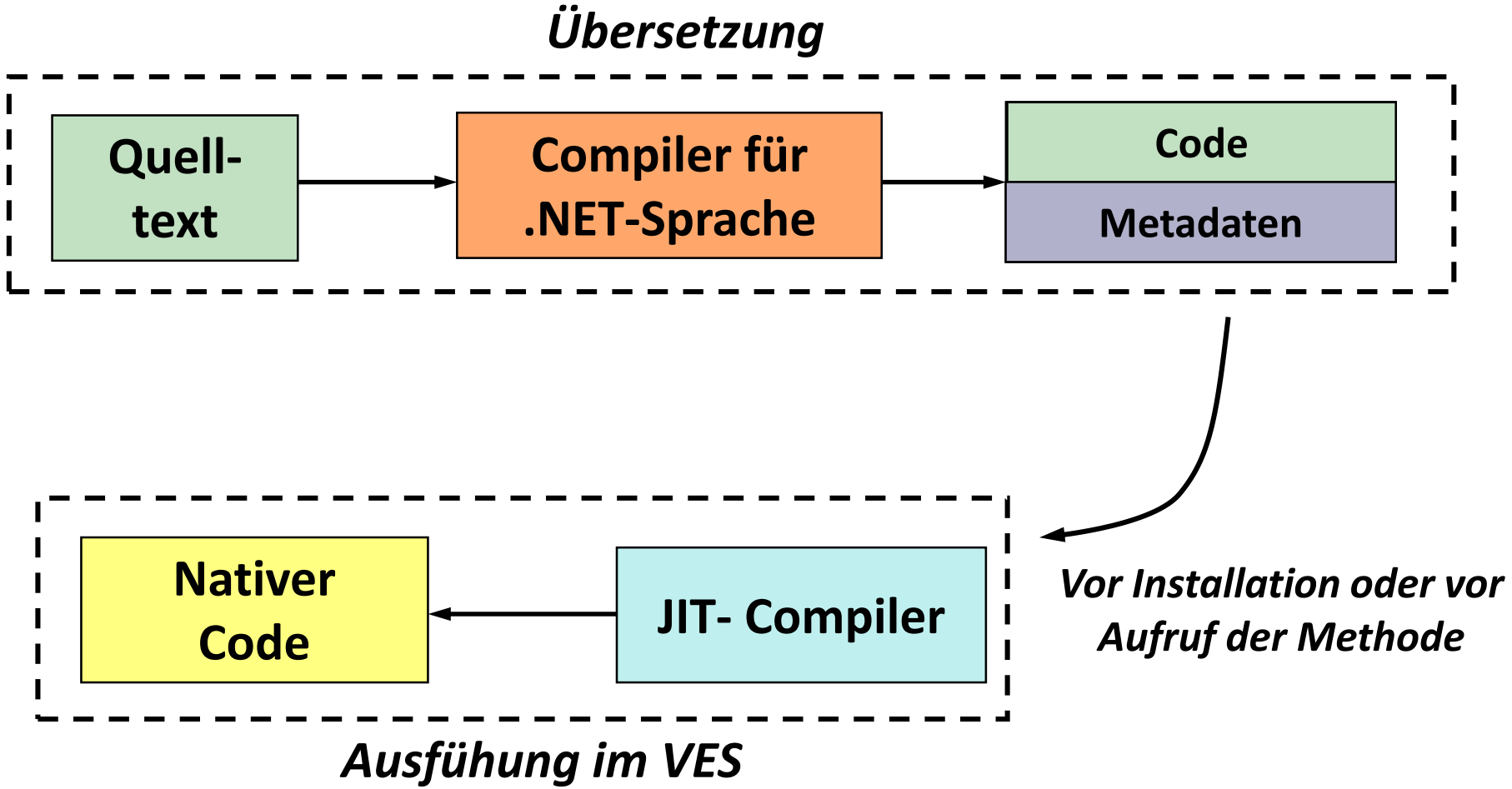
- Kategorien von IL-Code
  - Ungültiger Code
    - Code enthält faschen IL-Code.
  - Gültiger Code
    - Code kann nicht typsichere Anweisungen (kann durch Zeigerarithmetik entstehen) enthalten.
  - Typsicherer Code
    - Objekte halten sich an Schnittstellen.
  - Verifizierbarer Code
    - Typsicherheit 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

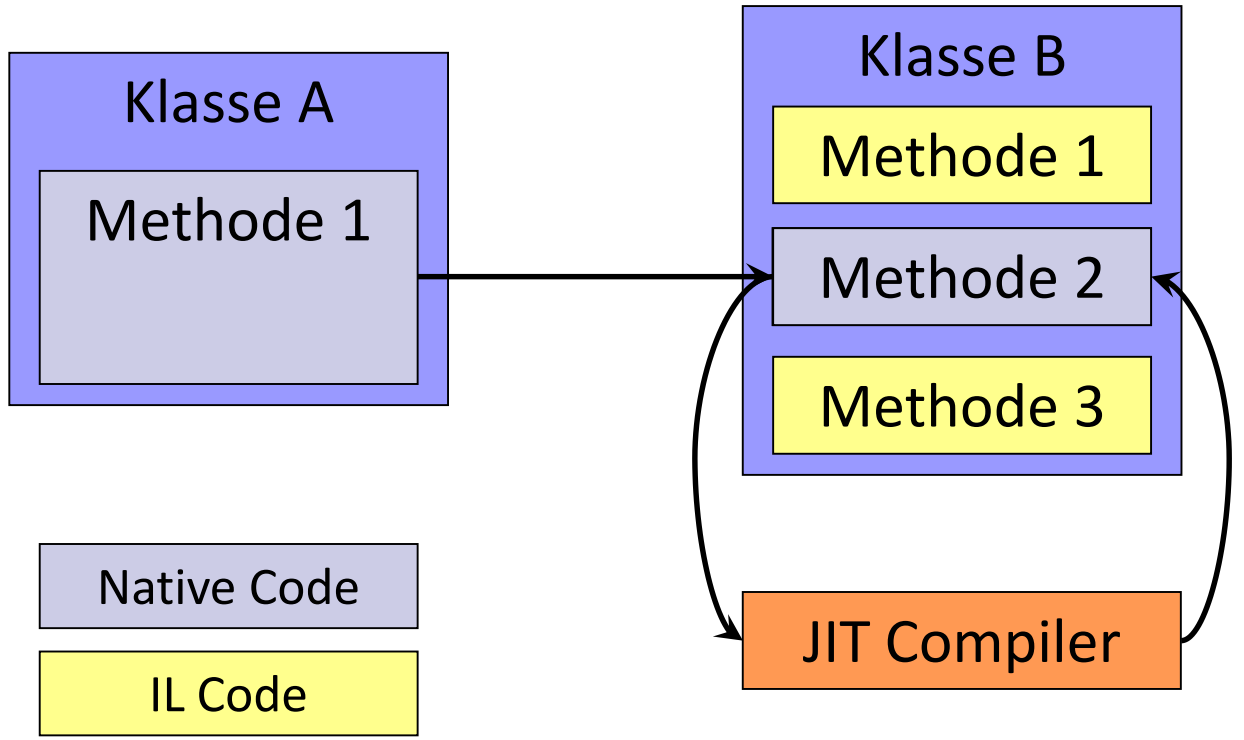
Übersetzung und Ausführung

MSIL - > Microsoft Implementierung



Just in Time Compiler (JIT)

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



Bei Aufruf einer Methode also zb Methode 1 ruft Methode 2 auf, dann wird automatisch der JIT Compiler aufgerufen
Bei jedem weiteren Aufruf wird der native Code verwendet
nur bei jenen die Aufgerufen werden wird nicht gespeichert, bei Neustart gibt es wieder eine erste Verwendung der Methode, also wird der native Code nie gespeichert

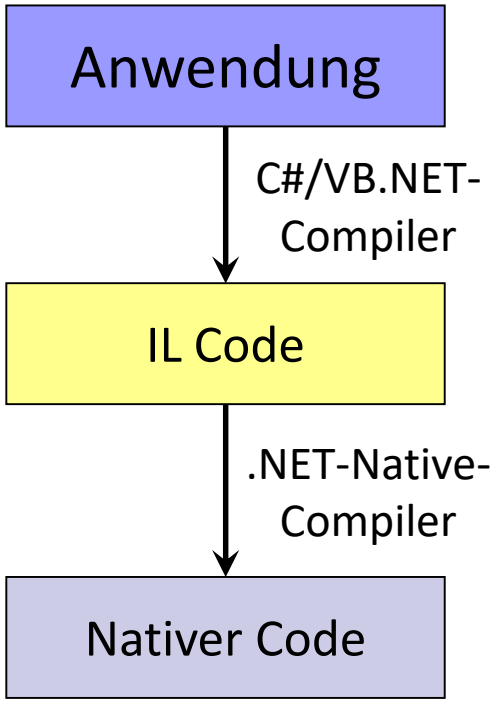
IL-Code wird durch *Maschinencode (nativem Code)* ersetzt.

.NET Native (.NET Core)

Intermediate Language

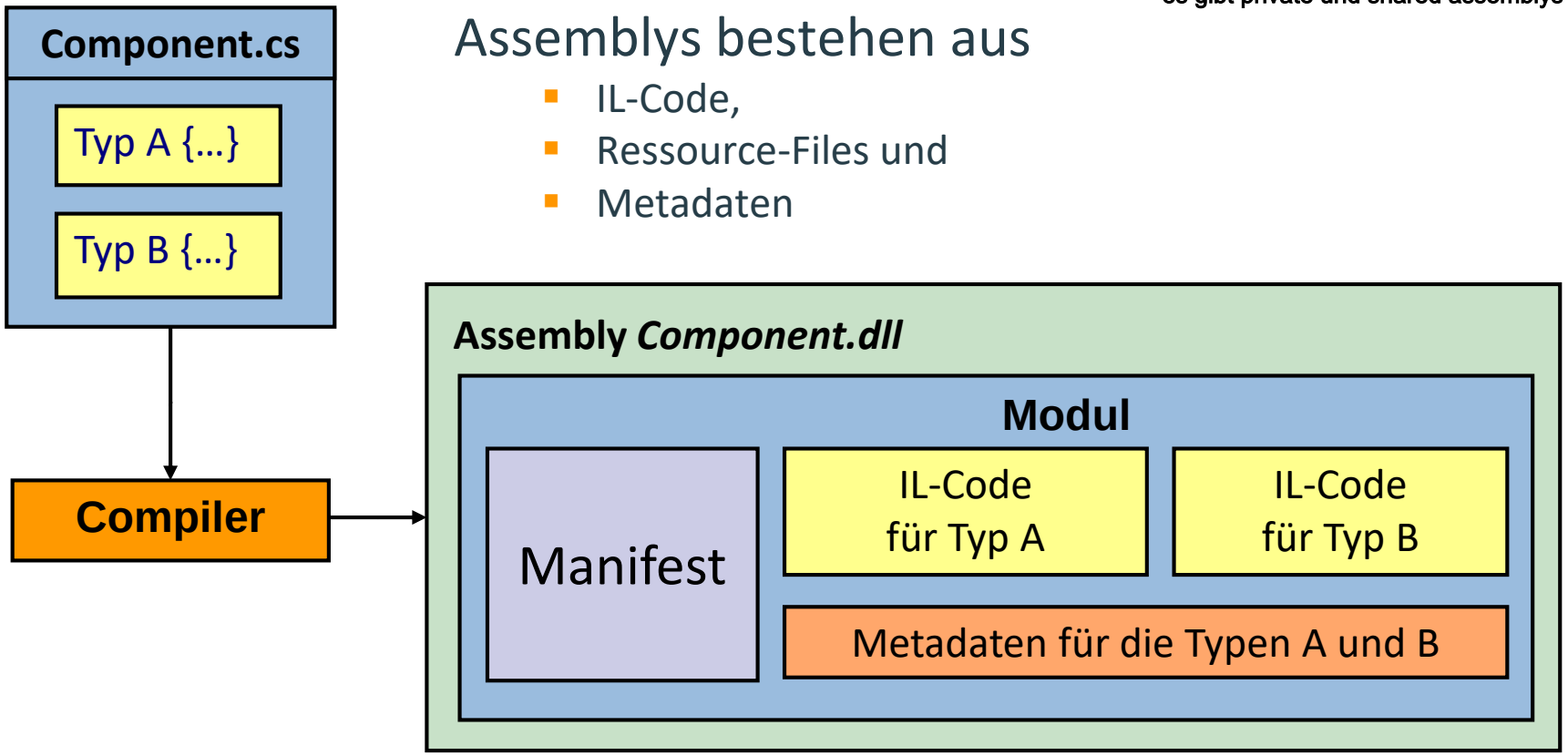
- Der JIT-Compiler übersetzt IL-Code zur Laufzeit des Programms → Just In Time.
- Mit .NET Native kann IL-Code bereits zur Übersetzungszeit des Programms in nativen Code kompiliert werden.
- .NET Native nutzt das Compiler-Backend von C++.
- Vorteile
  - Schnellere Ausführungszeiten
  - Schnellerer Programmstart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

Programmstart beschleunigt sich dadurch weil weniger Daten geladen werden müssen



Assemblys

exe und dlls sind assemblys
es werden .NET Typen darin gespeichert
es gibt private und shared 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