

Praktische Informatik

Vorlesung 02

Projektmanagement

Zuletzt haben wir gelernt...

- Dass unterschiedliche Bedienkonzepte für die Mensch-Maschine-Interaktion existieren.
- Welche Kriterien eine qualitativ hochwertige Benutzerschnittstelle ausmacht.
- Was eine grafische Benutzeroberfläche ist und woraus sie besteht.
- Wie uns Frameworks bei der Entwicklung einer grafischen Benutzeroberfläche helfen.

Inhalt heute

- Projektmanagement
- Compiler
- Assemblies
- Bibliotheken und Abhängigkeiten
- Buildwerkzeuge
- Paketmanager
- NuGet

Projektmanagement

- Wir wollen gute Programmierer werden.
 - Dazu gehört auch ein Verständnis, wie größere Softwareprojekte aufgebaut sind.
- Wir benötigen ein Verständnis von einigen Dingen, die eher indirekt etwas mit dem eigentlichen Programmieren zu tun haben.
 - Wie wird Quellcode in z.B. C# in Maschinencode übersetzt?
 - Aus welchen Elementen kann ein Programmierprojekt aufgebaut werden?
 - Wie können manche Teile, z.B. Klassen in anderen Projekten wiederverwendet werden?
 - Wie verteilt man Software auf die Zielrechner?
- Diesen Fragen werden wir uns heute widmen.

Compiler

- Im ersten Semester haben wir bereits gelernt, dass ein Prozessor lediglich Maschinencode versteht.
 - Unsere Anweisungen in C#, Java oder C++ müssen entsprechend **immer** erst in Maschinencode übersetzt werden.
- Ein solcher Übersetzer wird **Compiler** genannt.
 - Er überprüft die Syntax und die Grammatik der Anweisungen und erzeugt ggf. Warnungen bzw. Fehlermeldungen.
 - Er optimiert die Programme und entfernt z.B. überflüssige Anweisungen oder ersetzt sie durch bessere.
- Als Ergebnis wird der Maschinencode, das **Kompilat** erzeugt.
 - Obwohl der Maschinencode immer entstehen muss, existieren ganz unterschiedliche Strategien, wann dies passiert.
 - Alle diese Strategien haben eigene Vor- und Nachteile.

C/C++

- Bei C/C++ ist der Maschinencode das direkt Ergebnis der Programmierung.
 - Der Maschinencode liegt als Datei nach der Kompilierung vor und wird auch in dieser Form auf die Zielrechner verteilt.
- **Vorteile:**
 - Auf dem Zielrechner wird der Maschinencode direkt und ohne irgendeine bremsende Schicht ausgeführt: Hohe Performance!
 - Für eingebettete Systeme ist dies (oft) die einzig mögliche Strategie!
- **Nachteile:**
 - Plattformabhängig: Das Kompilat ist nur auf dem Prozessor lauffähig, für den es auch übersetzt wurde.
 - Auf dem Zielrechner läuft keine Schicht, die Einfluss auf das Programm nehmen könnte: Keine Prüfung von Sicherheit, Speicherbereinigung, ...

JavaScript

- Viele Scriptsprachen, wie z.B. JavaScript, werden erst dann in Maschinencode überführt, wenn das Programm gestartet wird.
 - Der Programmcode selbst wird auf die Zielrechner verteilt.
 - Dies ist das genaue Gegenteil der Strategie von C/C++.
- **Vorteile:**
 - Plattformunabhängigkeit: Der Entwickler muss nicht darauf achten, für welche Plattform er gerade entwickelt.
 - Für Anwendungen im Web-Browser wäre Plattformabhängigkeit fatal, da der Entwickler nie weiß, welches Betriebssystem der Nutzer besitzt.
- **Nachteile:**
 - Der Zielrechner muss genügend Performance haben, den Programmcode (just-in-time) zu übersetzen.
 - Fehler treten wirklich erst zur Laufzeit auf → Sicherheit?

Bytecode

- C# und Java setzen auf eine Mischform.
 - Der Compiler erzeugt als Ergebnis eine Zwischensprache.
 - In .Net wird diese **Intermediate Language (IL)**, bei Java **Bytecode** genannt.
- Die IL/Bytecode ist kein Maschinencode, ist aber schon nah dran.
 - Nicht nur C# kann in IL überführt werden, sondern auch VB.Net, F#, ...
 - In .Net können in Projekten mehrere Programmiersprachen gemischt werden.
- Die IL/Bytecode wird auf den Zielrechner verteilt und dort von einer Ausführungsschicht in Maschinencode überführt und ausgeführt.
 - In .Net: Common Language Runtime (CLR).
 - In Java: Java Runtime Environment (JRE).

Vor- und Nachteile von Bytecode

- **Vorteile:**

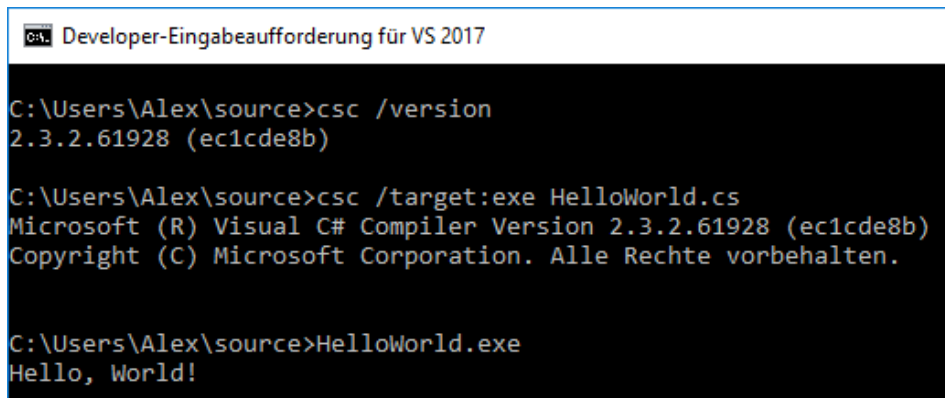
- Der Bytecode ist selbst ist plattformunabhängig → Der Programmierer muss nicht für unterschiedliche Plattformen übersetzen.
- Dieses Vorgehen ist i.d.R. performanter als die Interpretation zur Laufzeit wie bei Scriptsprachen.
- Die Ausführungsschicht kann die Ausführung des Programms überwachen, z.B. Sicherheitsregeln durchsetzen und den Speicher aufräumen.

- **Nachteile:**

- Auf dem System, welches den Bytecode ausführen soll, muss die entsprechende Ausführungsschicht installiert sein.
- Die Ausführungsschicht kostet Performanz zur Laufzeit!

C# Compiler

- Der C# Compiler kann auch auf der Kommandozeile ausgeführt werden.
 - Normalerweise wird er dadurch angestoßen, wenn wir im Visual Studio den Knopf „Starten“ betätigen.
 - Am besten startet man (Windows) die **Developer Eingabeaufforderung**, die mit dem Visual Studio installiert wird.
- Der Befehl für den Compiler ist **csc**.
 - Über entsprechende Parameter muss dem Compiler mitgeteilt werden, welchen Programmcode (Datei) er wie übersetzen soll.

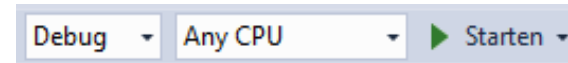


```
C:\Users\Alex\source>csc /version
2.3.2.61928 (ec1cde8b)

C:\Users\Alex\source>csc /target:exe HelloWorld.cs
Microsoft (R) Visual C# Compiler Version 2.3.2.61928 (ec1cde8b)
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\Alex\source>HelloWorld.exe
Hello, World!
```

Konfigurationen



- Im Visual Studio ist links neben dem Starten-Knopf auch noch eine Auswahlleiste zu sehen, in der zwischen „Debug“ und „Release“ gewechselt werden kann.
 - Dies sind sog. **Übersetzungskonfigurationen**.
- Die Debug-Konfiguration ist dabei standardmäßig ausgewählt.
 - Wird diese Konfiguration zur Übersetzung genutzt, enthält der Bytecode zusätzliche Informationen für den Debugger.
- In Übersetzungskonfigurationen können im Allgemeinen Einstellungen für die Übersetzung des Projektes zusammengefasst werden.
 - Zu den wichtigsten gehört sicher, in welchem Verzeichnis die Ergebnisse der Übersetzung abgelegt werden sollen.
 - In der Konfiguration „Debug“ ist dies standardmäßig das Verzeichnis „bin/Debug“, welches relativ zum Projektverzeichnis erzeugt wird.

Präprozessoranweisungen

- In vielen Programmiersprachen kann der Übersetzungsprozess mit zusätzlichen Anweisungen im Programmcode gesteuert werden.
 - So können manche Bereiche z.B. nur dann in Maschinencode übersetzt werden, wenn bestimmte Bedingungen erfüllt sind.
- Solche Anweisungen nennt man **Präprozessoranweisungen**.
 - Sie werden lediglich vom Compiler verarbeitet und spielen keine Rolle zur Laufzeit des Programms.
- In den meisten Programmiersprachen erkennt man solche Anweisungen damit, dass sie mit dem Doppelkreuz beginnen.
 - z.B. #define, #if, ...

Beispiel

- Stellen wir uns vor, unser Programm soll nur in einer Premium Version bestimmte Features aufweisen.
 - In der „normalen“ Version sollen bestimmte Funktionen gar nicht erst mit übersetzt werden.
 - Hierbei hilft die bedingte Übersetzung mit Präprozessoranweisungen.

- Am Kopf einer Quellcode-Datei definieren wir dazu das Symbol PREMIUM:

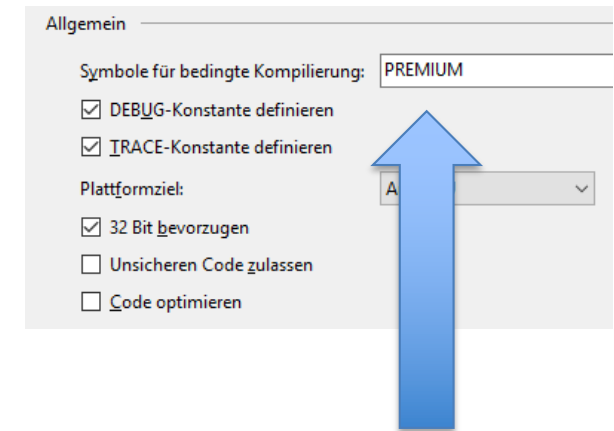
```
#define PREMIUM
```

- In Abhängigkeit, ob dieses Symbol definiert wurde, können wir nun bestimmte Teile bei der Übersetzung einbinden, oder ausschließen.

```
#if PREMIUM  
    Console.WriteLine("Premiumfunktion!");  
#endif
```

Globale Symbole

- Normalerweise sind Symbole, die mit `#define` eingeführt wurden, nur pro Datei bekannt.
 - Besonders sinnvoll sind Symbole aber, wenn sie projektweit definiert wurden.
- Dafür kann im Visual Studio in den Projekteigenschaften gesorgt werden.
 - Im Bereich „Build“ können für jede Konfiguration global für das ganze Projekt eigene Symbole definiert werden.
- In den standardmäßig vorhandenen Konfigurationen „Debug“ und „Release“ werden bereits automatisch die Symbole „DEBUG“ und „RELEASE“ definiert.
 - Diese können dafür genutzt werden, um bestimmten Programmcode nur in der Debug-Variante eines Programms zu inkludieren.



Weitere Präprozessoranweisungen

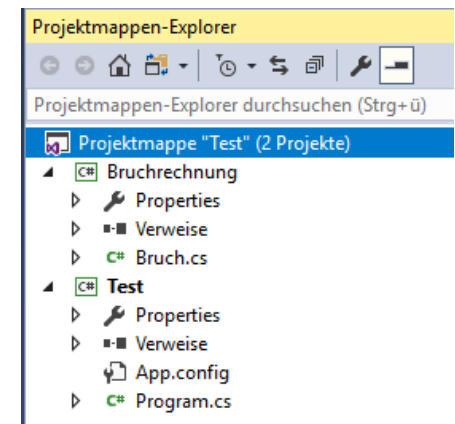
- Neben `#define`, `#if` und `#endif` existieren noch einige weitere Präprozessoranweisungen.
 - Alle steuern die Übersetzung und werden nicht zur Laufzeit des Programms ausgeführt.
- Für C/C++ ist insbesondere auch die Anweisung `#include` wichtig.
 - Mit `#include` kann der Inhalt einer anderen Datei in einer Quellcode-Datei eingebunden werden, bevor die Übersetzung beginnt.
- Wenn in C++ eine Funktion benutzt werden soll, die in einer anderen Datei definiert wurde, muss der Compiler Informationen über die Signatur der Funktion erhalten.
 - Dies wird über sog. Header-Dateien und `#include` gelöst.
 - Wir werden dies noch in den Vorlesungen zu C++ sehen.

Assemblies

- Auf der .Net Plattform wird das Resultat der Übersetzung in sog. **Assemblies** verpackt.
 - Eine Assembly ist eine einzelne Datei und kann Ressourcen, wie Bilder usw. und den Bytecode vieler Klassen enthalten.
 - Zusätzlich sind in einer Assembly beschreibende Meta-Daten, z.B. Autor, Version und Sicherheitsinformationen hinterlegt.
- Eine Assembly kann direkt ausführbar sein.
 - Sie hat dann die Dateiendung EXE und besitzt genau einen Einsprungspunkt für den Start der Anwendung (eine Main-Methode).
- Eine Assembly kann aber auch eine **Bibliothek** sein.
 - Eine Bibliothek verpackt Klassen und andere Ressourcen, die man in vielen anderen Projekten wiederverwenden möchte.
 - Eine Bibliothek hat die Dateiendung DLL.

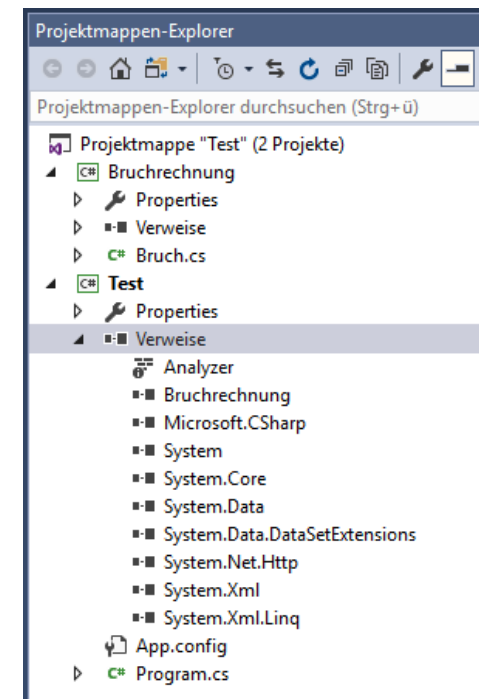
Bibliothek erzeugen

- Im Visual Studio ist es sehr leicht, eine Bibliothek zu erzeugen und einzubinden.
 - Es existiert eine entsprechende Projektvorlage: Klassenbibliothek.
- Ein solches Projekt kann nur übersetzt, aber nicht gestartet werden.
 - Nur öffentliche Klassen (public) können von anderen Projekten wiederverwendet werden.
 - Es ist zudem darauf zu achten, in welchem Namespace (Namensraum) die zu nutzenden Klassen liegen.
- Testweise erstellen wir eine neue Konsolenanwendung.
 - Wir fügen der Projektmappe ein weiteres Projekt vom Typ Klassenbibliothek mit dem Namen Bruchrechnung hinzu.



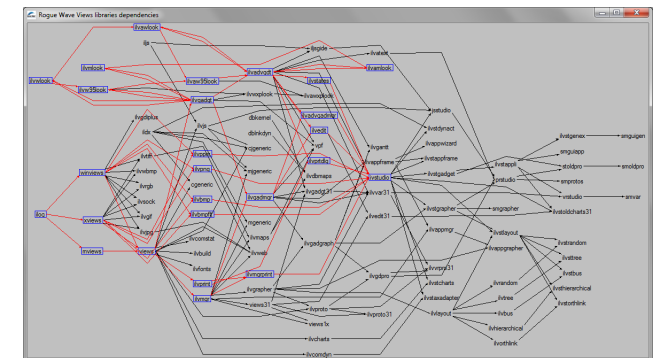
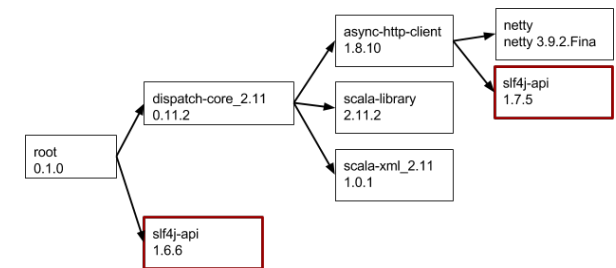
Bibliothek einbinden

- Jedes Projekt verfügt im Visual Studio über den Abschnitt **Verweise**.
 - Dort sind alle Bibliotheken aufgeführt, die ein Projekt benötigt.
 - Im Normalfall sind hier bereits Abhängigkeiten zu .Net eigenen Bibliotheken aufgeführt.
- Über einen Rechtsklick kann dem Projekt ein Verweis hinzugefügt werden.
 - Für das Projekt „Test“ können wir nun ein Verweis auf die Projektmappe „Bruchrechnung“ hinzufügen.
- Im Projekt „Test“ können nun alle öffentlichen Klassen aus der Bibliothek „Bruchrechnung“ benutzt werden.
 - Auch die Erstellungsreihenfolge wird beachtet.
 - Bevor „Test“ übersetzt werden kann, muss „Bruchrechnung“ übersetzt werden.
 - Anschließend wird die neue DLL als Bibliothek im Projekt „Test“ eingebunden.



Abhängigkeiten

- Ein Projekt, welches eine Bibliothek einbindet, hat zu dieser Bibliothek eine **Abhängigkeit**.
 - Das Projekt kann nicht ohne diese Bibliothek übersetzt und ausgeführt werden.
- Auch Bibliotheken können wiederum Abhängigkeiten zu weiteren Bibliotheken besitzen.
 - Dadurch kann ein kompliziertes Geflecht entstehen.
- Auf der einen Seite ist es sinnvoll, Klassen und Funktionen, die man häufig benötigt, in Bibliotheken zu sammeln.
 - Man darf es aber nicht übertreiben.



Global Assembly Cache

- Wenn ein Programm auf einen anderen Rechner übertragen wird, müssen alle notwendigen Bibliotheken mitgeliefert werden.
 - Die benötigten Bibliotheken werden beim Start von .Net Programmen immer zuerst im **lokalen Verzeichnis** gesucht.
- Werden Bibliotheken dort nicht gefunden, wird an einem speziellen Ort danach gesucht.
 - Der sog. **Global Assembly Cache (GAC)** stellt Bibliotheken systemweit für alle .Net Anwendungen zur Verfügung.
 - Das Verzeichnis findet sich meist unter c:\windows\assembly.
- Man kann dort auch selber Bibliotheken installieren.
 - Dazu muss die Bibliothek mit einem kryptographischen Schlüssel signiert werden.
 - Das Kommandozeilenwerkzeug gacutil.exe hilft dann bei der Installation im GC.
 - Vorsicht: Es gibt nur wenige Gründe, warum man eine eigene Bibliothek im GAC installieren sollte!

Paketmanager

- Eine Bibliothek von einem anderen Entwickler im eigenen Projekt einzubinden, kann mitunter aufwändig sein.
 - Man muss den Quellcode bzw. die DLL-Datei und alle notwendigen Abhängigkeiten herunterladen.
 - Im Projekt müssen die Verweise hinterlegt werden.
 - Auch Updates kosten viel Aufwand.
- Um die öffentliche Bereitstellung und Einbindung von Bibliotheken zu vereinfachen, existieren in vielen Programmiersprachen sog. **Paketmanager**.
 - Ein Paketmanager verbindet sich i.d.R. mit einem **öffentlich zugänglichen Verzeichnis**, in dem eine Vielzahl von wiederverwendbaren Bibliotheken zugänglich sind.
 - Ein Paketmanager kann von dort Bibliotheken samt ihrer Abhängigkeiten installieren, updaten bzw. auch eigene Pakete dort ablegen.
- Paketmanager existieren für viele (nicht alle) Plattformen.
 - JavaScript: npm
 - .Net: Nuget

NuGet

- Auf der .Net Plattform hat sich der Paketmanager **NuGet** etabliert.
 - NuGet ist eine OpenSource Software unter der Apache Lizenz, die u.a. von Microsoft entwickelt wurde.
- Das zentrale Verzeichnis von NuGet ist unter nuget.org erreichbar.
 - Dort sind aktuell mehr als 2,6 Millionen unterschiedliche Pakete registriert.
- NuGet ist eigentlich ein Kommandozeilenwerkzeug.
 - Es besitzt aber eine gute Integration in das Visual Studio.
 - Die NuGet-Konsole kann über den Menüpunkt **Extras** → **NuGet-Paket Manager** erreicht werden.

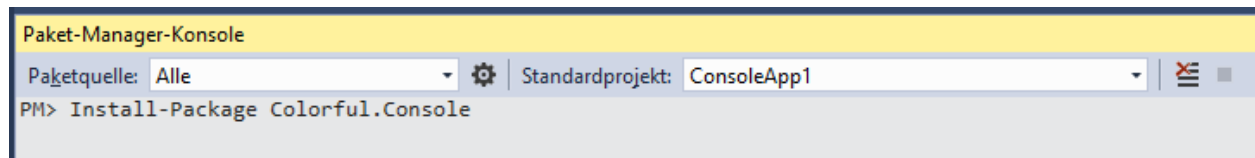
NuGet Kommandos

- NuGet wird in der Paket-Manager-Konsole über Kommandos gesteuert.

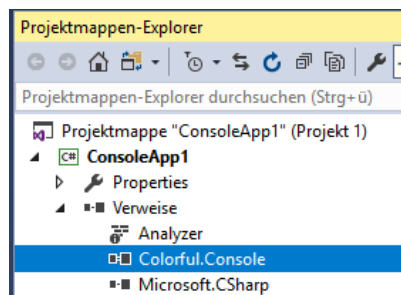
Kommando	Bedeutung
Install-Package	Ein Paket wird heruntergeladen und die Bibliotheken im aktuellen Projekt als Verweis eingefügt.
Update-Package	Das Paket wird auf den neusten Stand erneuert.
Uninstall-Package	Das Paket wird aus dem Projekt entfernt.

Beispiel Colorful.Console

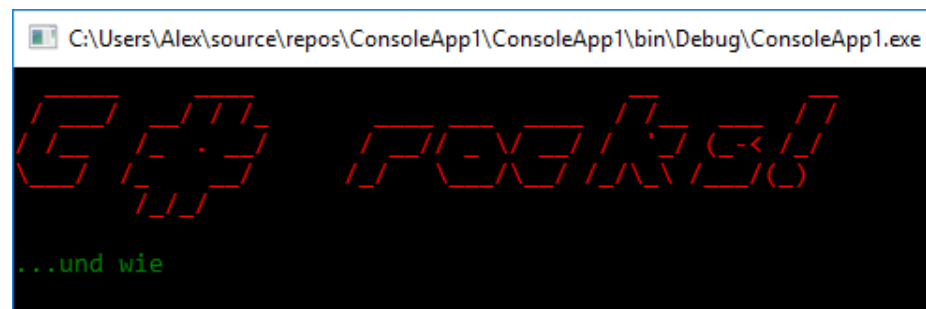
- Das Projekt Colorful.Console ermöglicht es, farbigen Text in Konsolenprojekten auszugeben.
 - Dazu kann das Projekt über Nuget in der Paket-Manager-Konsole installiert werden.



- Anschließend ist das Projekt eingebunden und kann im Quellcode genutzt werden...



```
Console.WriteLine("C# rocks!", Color.Red);
Console.WriteLine("...und wie", Color.Green);
```

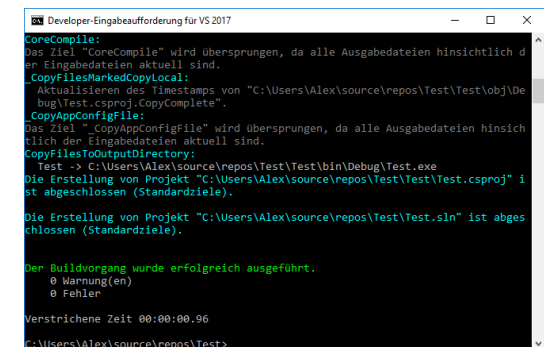


Buildwerkzeuge

- Wie wir sehen, bestehen Programmierprojekte oft aus vielen Quellcodedateien.
 - Diese müssen in einer bestimmten Reihenfolge übersetzt und zu Bibliotheken oder anderen Projektergebnissen zusammengefasst werden.
 - Auch automatisierte Tests, die Erzeugung von Installationsskripten und die Verteilung auf Zielrechner fällt oft als Arbeit an.
- Bei derartigen Aufgaben helfen sog. Buildwerkzeuge.
 - Sie steuern den Erzeugungsvorgang eines Projektes und nutzen weitere Werkzeuge, wie z.B. den Compiler.
- Auf jeder Plattform existieren derartige Tools.
 - C/C++: make, cmake, qmake, ...
 - Java: ant, maven, ...
 - C#: msbuild

msbuild

- Bei .Net kann **msbuild** als Buildwerkzeug eingesetzt werden.
 - Msbuild ist ein Werkzeug für die Kommandozeile.
 - Msbuild erwartet eine der folgenden Dateien als Eingabe:
- Eine **Projektdatei (Dateiendung „csproj“)**
 - Sie legt fest, aus welchen Quellcodedateien ein Projekt besteht, welche Abhängigkeiten zu Bibliotheken existieren und welche Übersetzungskonfigurationen festgelegt wurden.
- Eine **Projektmappe (Dateiendung „sln“)**
 - Sie verweist auf mehrere Projektdateien und bindet dieses zu einem großen Entwicklungsprojekt zusammen.
- Diese Dateien dienen als Steuerinformationen, so dass msbuild den Compiler in der richtigen Reihenfolge und Einstellungen auf die definierten Dateien anwenden kann.
 - Das Visual Studio erzeugt solche Dateien automatisch.
 - In C/C++ Projekten ist es mitunter notwendig die entsprechende Dateien manuell anzulegen.



```

Developer: Eingabeaufforderung für VS 2017
CoreCompile:
Das Ziel "CoreCompile" wird übersprungen, da alle Ausgabedateien hinsichtlich d
er Eingabedateien aktuell sind.
CopyFilesMarkedCopyLocal:
Aktualisieren des Timestamps von "C:\Users\Alex\source\repos\Test\Test\obj\De
bug\Test.csproj.CopyComplete".
CopyAppConfigFile:
Das Ziel "CopyAppConfigFile" wird übersprungen, da alle Ausgabedateien hinsich
tlich der Eingabedateien aktuell sind.
CopyFilesToOutputDirectory:
Test -> C:\Users\Alex\source\repos\Test\Test\bin\Debug\Test.exe
Die Erstellung von Projekt "C:\Users\Alex\source\repos\Test\Test\Test.csproj" i
st abgeschlossen (Standardziele).
Die Erstellung von Projekt "C:\Users\Alex\source\repos\Test\Test.sln" ist abges
chlossen (Standardziele).

Der Buildvorgang wurde erfolgreich ausgeführt.
0 Warnung(en)
0 Fehler

Verstrichene Zeit 00:00:00.96
C:\Users\Alex\source\repos\Test>
  
```

Wir haben heute gelernt...

- Welche Strategien für die Übersetzung von Quellcode existieren und wie dies auf der .Net Plattform geschieht.
- Was Assemblies sind.
- Wie man Bibliotheken erstellt und diese in Projekten zur Wiederverwendung von Programmcode einsetzt.
- Was ein Paketmanager leistet.
- Wie man NuGet einsetzt, um eine externe Bibliothek zu einem Projekt hinzuzufügen.
- Was Buildwerkzeuge sind und warum diese gebraucht werden.