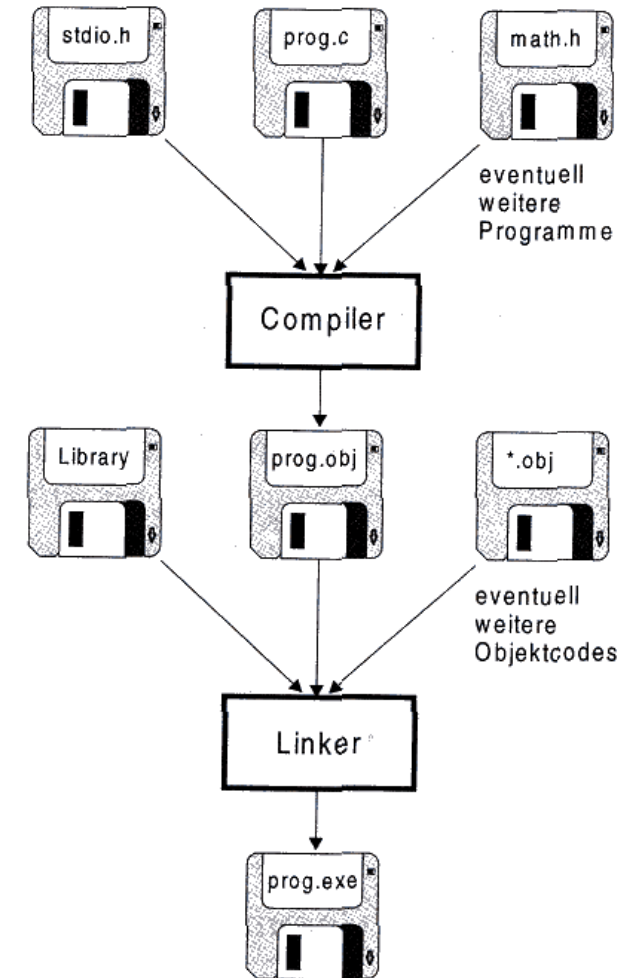




.NET und COM Bibliotheken

Allgemeines

- Ein Programm (EXE) kann als einzige Datei bereitgestellt werden. Dabei wird jegliche Funktionalität in das Programm eingebunden
- Bibliotheken werden statisch in die EXE gelinkt
- Vorteile: Das Programm läuft ohne dass eine Installation erforderlich würde, keine Seiteneffekte
- Nachteile: ???



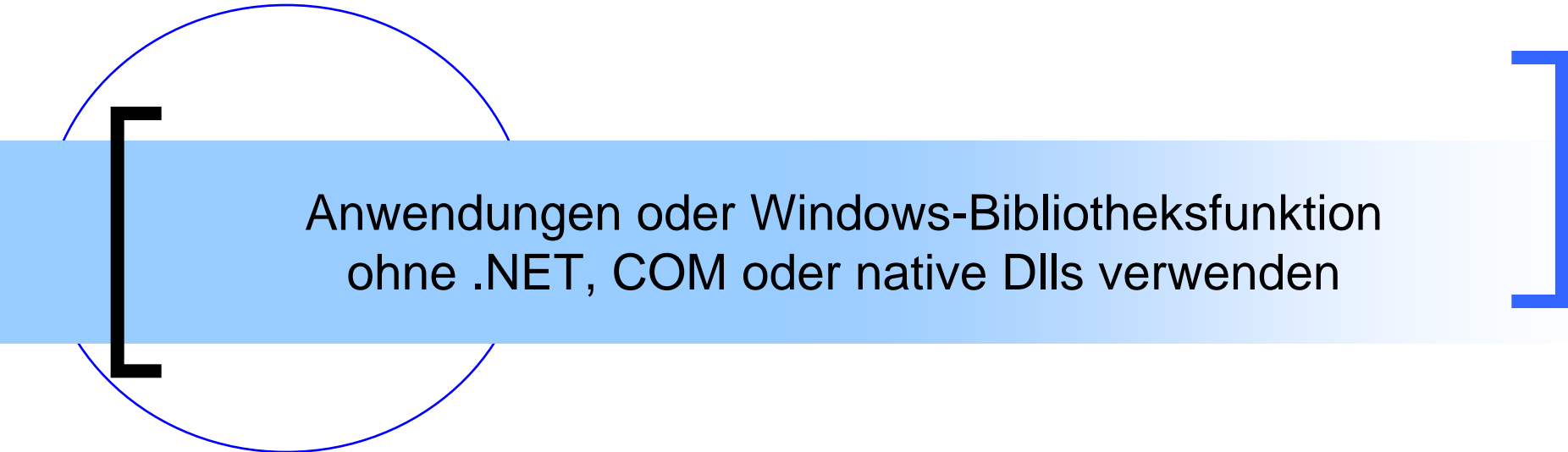
Quelle: Herrmann

- Soll eine Bibliothek nicht in das Programm eingebunden werden, muss sie folglich als eigenständige Datei in kompilierter Form zur Verfügung stehen.
- Der Hauptprozess (EXE) muss die Dll zu bestimmter Zeit in den Adressraum der Anwendung laden, wobei dann kein Unterschied mehr zu einer statisch gebundenen Bibliothek bestehen soll
- Probleme dabei sind
 - das Finden der Datei auf dem Rechner
 - die Verwendung einer Dll in der richtigen Version
- Für das Finden werden abhängig vom Dll-Typ verschiedene Verfahren eingesetzt:
 - Ordner, der über Suchpfade eingebunden ist (z.B. Programme\Gemeinsame Dateien)
 - Registrieren von Dlls, die dabei Ihren Pfad in die Windows-Registry schreiben und somit unabhängig von Installationsordnern gefunden werden (Registrieren von COM-Dlls über regsvr32.exe, .NET-Dlls über regasm.exe)
- Wird eine Dll in falscher Version geladen, führt das bei Aufruf inkompatibler Funktionen oder bereits beim Laden zu Laufzeitfehlern.
 - Dieses Problem kann nur durch entsprechendes Fehlermanagement erkannt werden
 - In .NET werden Dlls verschiedener Versionen parallel auf dem Rechner gehalten und über den global assembly cache abhängig von Dateiversion, Betriebssystem usw. bereit gestellt

- In gleicher Form wie eine Dll kann auch eine Anwendung Dll-Funktionalität bereitstellen
- Hierbei kann der laufende Prozess nicht in den eigenen Prozess integriert werden, weshalb eine Inter-Prozess-Kommunikation erforderlich wird. Dabei müssen Daten zwischen den Prozessen ausgetauscht werden. Diese werden dazu in ein Paket verpackt und auf Empfängerseite wieder entpackt. Das Verpacken wird als Marshalling bezeichnet und verlangsamt den Aufruf insbesondere bei großen zu teilenden Datenmengen
- Derartige Schnittstellen und Architekturen ermöglichen beispielsweise die Automatisierung von Anwendungen durch Drittsysteme, wie die Steuerung eines CAD-Systems zur Datenerzeugung oder –manipulation
- Besteht andererseits die Möglichkeit, ein System selbst durch Dlls zur Laufzeit zu erweitern, so wird diese häufig als AddOn oder AddIn bezeichnet
- Derartige Dlls müssen den von der Hostapplikation gestellten Anforderungen entsprechen, um ladbar zu sein, d.h. es müssen z.B. bestimmte Klassen mit bestimmter Vererbung (durch eine Bibliothek angeboten) verwendet werden, um die Anbindung zu initiieren oder es müssen bestimmte Methoden verfügbar sein, die beim Laden aufgerufen werden und z.B. den Funktionsumfang der Hostapplikation erweitern oder andere Dinge in die Wege leiten (siehe Skript Automatisierung) oder ...

- Bibliotheken werden in C# über Verweise eingebunden. Das Laden und Entladen der Bibliotheken erfolgt ohne weiteres Zutun automatisch durch von der Umgebung bereit gestellte Basisfunktionalität
- Beim Setzen eines Verweises wird durch die Entwicklungsumgebung Visual Studio der Typ der Dll unterschieden:
 - .NET-Dll nutzen die gleichen Datentypen wie C#, die Daten zwischen Dll und C#-Exe können direkt ohne Marshalling genutzt werden
 - COM-Dlls basieren auf anderen Basis-Datentypen, hier muss eine Umwandlung der Daten erfolgen, was aber durch Visual Studio automatisch in die Anwendung hineingebaut wird (managed/unmanaged Code)
- Soll mit C# eine Bibliothek erstellt werden, so muss dazu lediglich der Projekttyp auf Klassenbibliothek umgestellt werden, dadurch wird eine Eintittspunktsroutine wie static void main überflüssig. In Grenzen kann eine Klassenbibliothek auch über eine COM-Schnittstelle genutzt werden
- In COM ist die Kompatibilität der Dll und die Versionsverwaltung ein großes Thema, in .NET können beliebig viele Dlls in unterschiedlicher Version vorhanden sein (im GAC) oder aber im Programmverzeichnis ohne Registrierung geladen werden (dann mit Versionsproblematik)

- Die Wiederverwendbarkeit von Code steigt enorm
- Eine Kooperation im Team läßt sich leichter umsetzen
- Mitarbeiter lassen sich besser gemäß ihren Fähigkeiten einsetzen
- Kapsel-Dlls machen das Arbeiten mit komplexen Problemen für ein Entwicklungsteam einfacher
- Tests auf unteren Ebenen lassen sich leichter durchführen, indem die Dlls mit einfachen Testprojekten auf ihre Funktion durchgeprüft werden können
- Fehler lassen sich leichter beheben, wirken sich aber auch stärker aus
- Ladezeiten verkürzen sich, da nicht immer alles benötigt wird und nur bei Bedarf hinzugeladen wird
- Funktionalität von Applikationen kann je nach Kundenwunsch individuell gestaltet werden, Konfigurierbarkeit von Software und ihrer Module wird damit leichter möglich, aber auch die Anbindung anderer Basissysteme durch Austausch einer Kapsel-Dll



Anwendungen oder Windows-Bibliotheksfunktion
ohne .NET, COM oder native DLLs verwenden

- Sollte eine Anwendung nicht über COM oder .NET gesteuert werden können, sind prinzipiell noch andere Arten der Fernsteuerung denkbar:
 - Simulation von Tastaturanschlägen, Start der Applikation durch direkten Aufruf der Programmdatei
 - Nutzen der (veralteten) DDE (Dynamic Data Exchange) Technik, die bei älteren Applikationen häufig noch aus Kompatibilitätsgründen unterstützt wird
 - Einsatz einer Standard-Dll, die von der Applikation als Schnittstelle angeboten wird
- Die Simulation von Tastaturanschlägen ist die letzte Möglichkeit und stellt insbesondere hinsichtlich der Kontrolle des Ablaufs kaum Möglichkeiten bereit
- DDE bedient sich einer speziellen Syntax, mit der eine Applikation im Rahmen der dort bereitgestellten DDE-Funktionen gesteuert werden kann.

Tastenfolgen senden

- Applikationen, die nicht über .NET, COM oder andere Schnittstellen fernsteuerbar sind, können auch direkt angesprochen werden, indem Tastenfolgen an den Prozeß geschickt werden
- Diese Vorgehensweise sollte nur in Ausnahmen angewandt werden, da sie sehr instabil bei Benutzer-Interaktionen ist

```
//using System.Threading erforderlichlich
public static void Main()
{

    // Notepad starten
    Process.Start("Notepad");

    // Etwas warten, bis Notepad gestartet ist
    Thread.Sleep(500);

    // Einen kleinen Text schreiben
    SendKeys.SendWait("Das ist ein Text, der von .NET kommt");

    // STRG+S simulieren
    SendKeys.SendWait("^s");

    // Überprüfen, ob die zu erzeugende Datei existiert, und
    // diese gegebenenfalls löschen
    string filename = @"C:\SendKeys_Demo.txt";
    if (File.Exists(filename))
    {
        File.Delete(filename);
    }

    // Dateinamen eingeben
    SendKeys.SendWait(filename);

    // Return simulieren
    SendKeys.SendWait("{ENTER}");

    // Notepad über ALT+F4 beenden
    SendKeys.SendWait("%{F4}");

}
```

- Tastenbetätigungen können über `SendKeys.Send` oder `.SendWait` simuliert werden.
 - Zeichenfolge Bedeutung
 - + Shift-Taste
 - ^ Strg-Taste
 - % Alt-Taste
 - {BACKSPACE}, {BS} oder {BKSP} Backspace-Taste
 - {BREAK} Pause-Taste
 - {CAPSLOCK} Feststelltaste
 - {DELETE} oder {DEL} Lösch-Taste (Entf)
 - {END} Ende-Taste
 - {ENTER} oder ~ Return-(Enter-)Taste
 - {ESC} Escape-Taste (Esc)
 - {HELP} Hilfe-Taste (F1)
 - {HOME} Home-Taste (Pos1)
 - {INSERT} oder {INS} Einfügen-Taste (Einfg)
 - {NUMLOCK} Num-Lock-Taste
 - {PGDN}/{PGUP} Bild-nach-unten/oben-Taste
 - {PRTSC} Druck-Taste (für zukünftige Versionen reserviert)
 - {SCROLLLOCK} Rollen-Taste
 - {TAB} Tabulator-Taste
 - {UP}/{DOWN}/{LEFT}/{RIGHT} Cursor-nach-oben/unten/links/rechts-Taste
 - {F1} ... {F16} Die Funktionstasten F1 bis F16
 - {ADD} Additionstaste auf der Zehnertastatur
 - {SUBTRACT} Subtraktionstaste auf der Zehnertastatur
 - {MULTIPLY} Multiplikationstaste auf der Zehnertastatur
 - {DIVIDE} Divisionstaste auf der Zehnertastatur



Nutzen nativer DLLs

- Eine Funktion in einem C-Header kann z.B. mit
`double DLLSTATEMENT sin(double _X);`
definiert werden. Hier erfolgt der Export einer einzelnen Funktion (ohne Klassenhintergrund), es ist aber in C++ auch möglich, eine Klasse zu exportieren.
- Mit
`#define DLLSTATEMENT __declspec(dllimport)`
wird im Headerfile festgelegt, ob die Funktion in das aktuelle Projekt importiert oder exportiert werden soll
- Dabei können sogenannte Defines auch mit Verzweigungen verwendet werden, um ein Header-File mehrfach nutzen zu können (in der exportierenden Dll und dem verwendenden Projekt)
`#ifndef _EXPORT`
`#define DLLSTATEMENT __declspec(dllimport)`
`#else`
`#define DLLSTATEMENT __declspec(dllexport)`
`#endif`
- Die Funktion `sin` liegt in einer nativen Dll vor und kann in C# genutzt werden. Dazu muß geklärt werden, welche Dll heranzuziehen ist und wie die Funktion angesprochen werden muß

- Die Funktion `GetKeyState` befindet sich in einer „normalen“ Windows-Bibliothek (`user32.dll`) und soll in einem C-Programm verwendet werden. Da es sich hierbei weder um eine .NET- noch eine COM-Dll handelt, die über einen Verweis eingebunden werden kann, sondern eine Standard-Dll, die im Gegensatz zu den genannten Dll-Typen keine unmittelbar verwendbaren Informationen bzgl. der enthaltenen Funktionen mitbringt und insbesondere keine Speicherautomatismen wie C# unterstützt, muß hier anders vorgegangen werden
- Um derartige Dll trotzdem nutzen und aufrufen zu können, gibt es in C# die Klasse `DllImportAttribute`, bei der mit folgendem Statement die `user32.dll` in den Adressraum geladen wird.

```
[DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]
```

- Zusätzlich muss definiert werden, wie eine Funktion aus der Dll aufgerufen werden soll. Dazu ist die Funktion über ein weiteres Statement in C# bekannt zu machen:

```
/*private/public*/ static extern short GetKeyState(int virtualKeyCode);
```
- Sollen Funktionsname in der Dll und in C# unterschiedlich sein, kann mit dem Attribut `EntryPoint="Name der Funktion in der Dll"` ein anderer Funktionsname in C# gewählt werden

- Die importierte Funktion kann in der Klasse, in der der Import erfolgt, als public, private oder auch internal (nur innerhalb des Namensraums) deklariert werden
- Da die Funktion aus der nativen Dll rein prozeduralen Charakter hat, gibt es nur die Möglichkeit einer statischen Methode, ohne Bindung an ein Objekt
- Die Funktion wird als extern deklariert, um zu zeigen, daß sie in der externen Dll implementiert ist.
- Die Funktion selbst muß wie immer in C#, einer Klasse zugeordnet sein
- Stimmen die Parameter der Funktion mit denen der Funktion in der Dll nicht überein, tritt beim Aufruf ein Laufzeitfehler auf
- Das CharSet-Argument dient dazu, in der Bibliothek die korrekte Funktion mit den richtigen Parametern zuzuordnen (es könnten z.B. zwei Varianten einer Funktion existieren, eine mit Ascii-, eine andere mit Unicode-Codierung), bei einem ExactSpelling-Attribut von true wird nach einer Funktion des angegebenen Namens ohne Ansi A oder Unicode W gesucht
- Durch die andere Speicherverwaltung (managed/unmanaged Code müssen sämtliche Parameter in entsprechende Datentypen umgewandelt werden

Aufruf einer Methode aus den Windows-System-Bibliotheken

```
/* Deklaration der API-Funktion GetWindowsDirectory, die einen String zurückgibt
 * GetWindowsDirectory entstammt der Dynamic Link Library Kernel32.dll,
 * die im System32-Ordner liegt (und damit auch gefunden wird
 * hier wird außerdem eingestellt, daß der Windows-Fehler gesetzt wird (kann
 * anschließend mit GetLastError wieder abgefragt werden), CharSet
 * wird auf auto gesetzt, um bei Umlauten zu funktionieren die importierte
 * Funktion steht in der aktuellen Klasse statisch zur Verfügung
 */
[DllImport("Kernel32.dll", SetLastError = true, CharSet = CharSet.Auto)]
public static extern uint GetWindowsDirectory(
StringBuilder lpBuffer, uint uSize);

[STAThread] /* Single Threaded Apartment
 * bedeutet, daß Methoden dieses Bereichs immer nur von einem
 * Prozess gleichzeitig gerufen werden können - das Problem des
 * Mehrfachzugriffs wird damit vermieden
 */
static void Main(string[] args)
{
    // Aufruf der Funktion GetWindowsDirectory
    StringBuilder buffer = new StringBuilder(261);
    if (GetWindowsDirectory(buffer, 261) > 0)
    {
        Console.WriteLine("Windows-Ordner: {0}", buffer.ToString());
    }
    else
    {
        Console.WriteLine("Fehler bei der Ermittlung des Windows-Ordners");
    }
}
```

- Durch eine C#-Klasse, die Dll-Funktionen aus einer nativen Dll statisch einbindet, können die sonst nur über DllImport-Statements verfügbaren Funktionen in einem .NET-Projekt einfach nutzbar gemacht werden.
- In einfachster Form werden die vorhandenen Funktionen einfach durch die C#-Klasse an die Dll delegiert und stehen ohne Kenntnisse von Dllimport und den genauen Aufrufkonventionen allen zur Verfügung
- Auch Intellisense funktioniert dann mit diesen Klassen
- Schließlich läßt sich die rein statische Funktionalität der externen Dll durch die objektorientierten Möglichkeiten von C# erweitern, so daß die Aufrufe der externen Dll nicht mehr mit endlosen Parametern, sondern – durch die C#-Klasse gekapselt – mit Objekten und an Objekten durchgeführt werden können, wenn dies für das Projekt sinnvoll sein sollte
- In diesem Fall ist die native Dll zwar auf dem jeweiligen Entwicklungs- und Ziel-Rechner zu installieren, die Nutzung durch die anderen Entwickler basiert dann aber auf der neuen Klasse