Includedateien und Units von Siegfried Thoß

Einstieg

Zu Beginn einer Unterrichtsstunde in einem Informatik-Lehrgang einer Volkshochschule wurde die Frage gestellt:

Was ist eine Sammlung von Konstanten, Datentypen und Variablen?

Was hättest du da geantwortet?

Du hättest zwei Antworten geben können:

a. es ist eine Includedatei,
b. es ist eine Unit.

Beide Antworten wären richtig gewesen.

Eine Sammlung von Konstanten, Datentypen und Variablen

Stellen wir uns doch einmal eine solche Sammlung zusammen:

```
{Unsere Sammlung}
CONST
  SpAnz = 24;
  ZlAnz = 14;

TYPE
  tStein =
    (leer, null, stern, kreuz);
  tFeld = ARRAY[0..SpAnz-1,0..ZlAnz-1] of tStein;

//Wenn Spielfeld.Font = Wingdings dann sollen diese gelten
{1} CONST (* Konstanten für Wingdings *)
  stLeer = 'ž';  stNull = #86;  stStern = 'J'; stKreuz = 'Ó';

//Wenn Spielfeld.Font = Wingdings 2 dann sollen diese gelten
{2} CONST (* Konstanten für Wingdings 2 *)
  stLeer = 'o';  stNull = #86;  stStern = 'Ù'; stKreuz = 'Ó';

CONST
  Stein: ARRAY [tStein] of Char =
    (stLeer, stNull, stStern, stKreuz);

VAR
  Count: Integer;
  Namen: String;
```

Das ist so eine Sammlung von Konstanten, Datentypen und Variablen.

Dieser Quelltextausschnitt stammt aus einem Programm, das verschiedene Aufgaben mit einer Anzahl von Steinen (keine grafische Darstellung) ausführt. Je nachdem, ob auf dem System des Endnutzers die Fonts Wingdings oder Wingdings 2 vorhanden sind, soll das Programm an den jeweils eingestellten Schriftfont angepasst werden. Das geschieht durch Verwendung der entsprechenden Konstanten. Die jeweils andere Konstantengruppe ({1} oder {2}) soll nicht zur Verwendung kommen.

Verwendung der Sammlung im Quelltext

Eine **Bedingung** ist in der Sammlung genannt:

Da nicht auf jedem Computer der Schriftfont Wingdings 2 vorhanden ist, gilt: Die Const-Gruppe {2} soll nur dann verwendet werden, wenn im Objektinspektor der Font Wingdings 2 einzustellen ist. Ist nur Wingdings vorhanden, muss die Const-Gruppe {1} verwendet werden.
Diese Festlegung dient dazu, dass in jedem der beiden Fälle eine ordnungsgemäße Bildschirmdarstellung erfolgen kann.

1. Welche Möglichkeiten zur Erfüllung der Bedingung gibt es?

Entweder verwende ich eine entsprechende Programmstruktur und wähle damit die Gruppe aus oder, was mir programmtechnisch lieber ist, ich schreibe die Codegruppen in eine andere Datei.

Wenden wir uns der zweiten Möglichkeit zu.
Und hier können wir die beiden schon genannten Alternativen vom Anfang anführen; nämlich eine Includedatei oder eine Unit.

2. Unsere Lösung: Includedatei oder Unit

Sowohl die Includedatei als auch die Delphi-Unit sind reine Textdateien, wie sie jeder gewöhnliche Texteditor erstellen kann.

Die Includedatei bildet einen Ausschnitt aus dem Quellcode des jeweiligen Programms. Mit ihr wird ein fortlaufender Quelltext in eine weitere Datei (oder mehr) aufgespaltet. *Eine Zerlegung des Projektes in Module ist damit nicht möglich.*

```
CONST
  stLeer = 'ž';  stNull = #86;  stStern = 'J'; stKreuz = 'Ó';
```

Das könnte zum Beispiel der Inhalt einer **Includedatei** mit dem Namen **IncWing2.inc** sein.

Die Unit kann durch das Programm erstellt oder vom Nutzer als eigenständige Datei angelegt werden.
Die grundlegende Struktur ist immer gleich.

```
unit <Bezeichner>
interface
uses <Liste von Units>        { optional }
{ public-Deklarationen }
implementation
uses <Liste von Units>        { optional }
```

```
{ private-Deklarationen }
{ Implementierung von Prozeduren und Funktionen }
initialization
{ optionaler Initialisierungs-Quelltext }
finitialization
{ optionaler Quelltext für "Aufräumungsarbeiten" }
end;
```

Im Gegensatz zur Includedatei kann ein Delphi-Projekt *mit Units in einzelne Module* aufgeteilt werden.
Aber dazu später mehr.

Arbeiten mit den "Sammlungs"dateien

1. Anlegen einer Includedatei

Das Anlegen einer Includedatei ist ganz einfach:

1. Datei|Neu|Textdatei aufrufen. Es wird eine leere Textdatei angezeigt.
2. In diese leere Datei gibst du einfach den Quelltext ein, der in dieser Datei verwaltet werden soll. Ein Beispiel hast du einige Zeilen weiter oben.
3. Danach speicherst du die Includedatei ab. Günstig ist es, wenn du ihr einen Namen mit der Endung .inc gibst. Ich mache das so. Du kannst es ganz anders machen. Jede Endung wird angenommen (auf vorhandene Dateiendungen achten!).

Also:
Zuerst rufst Du *Speichern unter* auf.
Hier stellst Du den *Dateityp Alle Dateien (*.*)* ein.
Du gibst den Namen mit der Endung ein (siehe oben).
Du klickst jetzt auf *Speichern*.

Um die Includedatei als Beispiel einsetzen zu können, fügen wir nicht unsere Sammlung von Daten vom Anfang (aus (1)) ein. Wir schreiben vielmehr an dieser Stelle:

```
CONST
  stLeer = 'ž';  stNull = #86;  stStern = 'J'; stKreuz = 'Ó';
```

und speichern diese Datei unter **IncWing2.inc** ab.

Jetzt ist die Includedatei zur Verwendung bereit.

2. Einbinden einer Includedatei in das Delphi-Projekt

Das Einbinden der Includedatei in den Quelltext des Programms ist ganz einfach:

Da der Inhalt der Includedatei ein Ausschnitt aus dem fortlaufenden Quelltext des Programms ist, wird der Compiler mit der Compilerdirektive
{$I Dateiname.inc}
an der Stelle in den Quelltext eingebunden, an der sonst der Inhalt dieser *.inc-Datei stehen würde.

Ein Ausschnitt aus einem Programm:

```
CONST
  SpAnz = 24;  (* Wert nicht verändern !! *)
  ZlAnz = 14;  (* Wert nicht verändern !! *)

TYPE
  tStein =
    (leer, null, stern, kreuz);
  tFeld = ARRAY[0..SpAnz-1,0..ZlAnz-1] of tStein;

(* ---------------------------------------------------------------
   Originaleinstellung:
   Spielfeld.Font = Wingdings 2 + {$I IncWing2.inc}
   Ersatzeinstellung (wenn Wingdings 2 nicht vorhanden):
   Spielfeld.Font = Wingdings + {$I IncWing1.inc}
   --------------------------------------------------------------- *)

//Wenn Spielfeld.Font = Wingdings 2 dann

{$I IncWing2.inc}

(* ---------------------------------------------------------------
   Includedateien verteilen den Quellcode einer Unit auf mehrere
   Dateien. Units dagegen dienen zur modularen Gestaltung eines
   Delphi-Programmprojektes.
   Die Parameter-Direktive $I weist den Compiler an, die angegebene
   Datei in die Compilierung aufzunehmen. Diese Datei wird direkt
   nach der Direktive {$I Dateiname} in den Text eingefügt.
   --------------------------------------------------------------- *)

CONST
  Stein: ARRAY [tStein] of Char =
    (stLeer, stNull, stStern, stKreuz);

VAR
  Count: Integer;
  Namen: String;
```

Damit ist es leicht möglich, vor dem endgültigen Compilieren durch Auswahl der entsprechenden Inc-ludedatei die Bildschirmausgabe an die Gegebenheiten des Computers anzupassen.

3. Anlegen einer Unit-Datei

Wenn du Formulare für deine Anwendung entwickelst, erzeugt Delphi ohne dein Zutun, also automatisch, neue Units, die mit dem Formular verbunden sind. Wenn du beispielsweise ein Dialogfeld in das Formular einfügst, setzt Delphi die Unit Dialogs in die Uses-Liste unterhalb von Interface in der Formu-larUnit ein. Damit werden alle Programmdaten bereitgestellt, die zum Funktionieren dieses Dialogfeldes gebraucht werden. Auf die wenigen Ausnahmen, bei denen du selbst Unitnamen in die Liste einfügen musst, kommen wir noch.

Units müssen nicht immer mit Formularen verbunden sein. Du kannst auch eigene Units erzeugen, in denen du ausschließlich deine Daten verwalten kannst. Mehr dazu im weiteren Verlauf dieses Tutorials.

So eine eigenentwickelte Unit könnte vielleicht dazu dienen, die unter (2) angeführte Bedingung in deinem Programm umzusetzen?

Ehe wir hier viel Herumprobieren, will ich folgendes dazu schreiben:
Da Units - wie weiter vorn bereits beschrieben - ein Programm in einzelne Module aufteilen, es also in bestimmter Weise gliedern, würden wir bei der Lösung unseres Problems "mit Kanonen auf Spatzen schießen". Es gilt ja, eine Auswahl aus zwei Möglichkeiten zu treffen. Eine Gliederung soll nicht erreicht werden. - An dieser Stelle nicht mehr dazu.

Im folgenden Abschnitt wollen wir uns mit Wesen und Wirklichkeit der Unit-Dateien etwas genauer befassen.

Wesen und Wirklichkeit der Unit-Dateien

1. Units, die Delphi selbst erzeugt

Zur Erinnerung noch einmal die Struktur einer Unit:

```
unit <Bezeichner>
interface
uses <Liste von Units>        { optional }
{ public-Deklarationen }
implementation
uses <Liste von Units>        { optional }
{ private-Deklarationen }
{ Implementierung von Prozeduren und Funktionen }
initialization
{ optionaler Initialisierungs-Quelltext }
finitialization
{ optionaler Quelltext für „Aufräumungsarbeiten“ }
end;
```

Die Kopfzeile einer Unit beginnt mit dem reservierten Wort **UNIT**, dem der Name der Unit als Bezeichner folgt.
Das nächste Element in einer Unit ist das reservierte Wort **INTERFACE**, das den Anfang des Interface-Teils einer Unit anzeigt - das ist der Teil einer Unit, der für andere Units oder für Anwendungen, die die Unit verwenden, sichtbar ist.

Eine Unit kann die Deklarationen in anderen Units verwenden, indem sie diese Units in einer **USES-ANWEISUNG** angibt. Die uses-Anweisung kann an zwei Stellen verwendet werden:

a. Uses unmittelbar hinter dem reservierten Wort INTERFACE

Aller Quelltext in der aktuellen Unit kann die in den Interface-Teilen deklarierten Deklarationen der in der Uses-Anweisung angegebenen Units verwenden.

Nehmen wir beispielsweise an, du schreibst Quelltext in der Unit A, und du möchtest eine Prozedur aufrufen, die im Interface-Teil von Unit B deklariert ist. Sobald du den Namen von Unit B in die Uses-Anweisung im Interface-Teil von Unit A aufnimmst, kann jeglicher Quelltext in Unit A die in Unit B deklarierte Prozedur aufrufen.

Ist Unit B in der Uses-Anweisung im Interface-Teil von Unit A enthalten, dann kann Unit

A nicht in der Uses-Anweisung im Interface-Teil von Unit B auftauchen. Andernfalls würde eine ***zirkuläre Unit-Referenz*** vorliegen, und Delphi würde beim Versuch, eine Compilierung durchzuführen, eine Fehlermeldung ausgeben.

b. Uses unmittelbar hinter dem reservierten Wort IMPLEMENTATION

Nur Deklarationen in der aktuellen Unit können die Deklarationen in den Interface-Teilen der hier angegebenen Units verwenden. Alle anderen Units, die die aktuelle Unit verwenden, haben keinen Zugriff auf die in dieser Uses-Anweisung verzeichneten Units.

Ist zum Beispiel Unit C in einer Uses-Anweisung im Implementation-Teil von Unit B aufgeführt und Unit B in einer Uses-Anweisung von Unit A, dann kann Unit A nur die Deklarationen in den Interface-Teilen von Unit B verwenden. Unit A kann nicht auf die Deklarationen in Unit C zugreifen, es sei denn, Unit C steht in einer Uses-Anweisung von Unit A.

Wenn Unit B in der Uses-Anweisung im Implementation-Teil von Unit A verzeichnet ist, kann Unit A in der Uses-Anweisung im Implementation-Teil von Unit B stehen.

Zirkuläre Unit-Referenzen im Interface-Teil sind unzulässig, aber im Implementation-Teil problemlos möglich.

c. Der Interface-Teil

Der Interface-Teil einer Unit beginnt mit dem reservierten Wort **INTERFACE**, das hinter der Kopfzeile einer Unit steht, und er endet mit dem reservierten Wort **IMPLEMENTATION**. Die Schnittstelle (interface) bestimmt, was für eine Anwendung oder eine andere Unit, die diese Unit verwendet, zugänglich ist.

In der Schnittstelle einer Unit kannst du all das deklarieren, was du an Konstanten, Datentypen und Variablen in deinem Programm verwenden möchtest (denke an unsere Sammlung aus Seite 2), dazu noch alle gewünschten Prozeduren und Funktionen.

Diese Prozeduren und Funktionen, die also für eine beliebige Unit oder Anwendung sichtbar sind, sind im Interface-Teil der Unit nur mit ihren Kopfzeilen aufgeführt. Der eigentliche Quelltext, also deren Implementierung, befindet sich dagegen im Implementations-Teil der Unit.

In der Schnittstelle kann eine optionale Uses-Anweisung verwendet werden; sie muss direkt auf das reservierte Wort Interface folgen.

d. Der Implementation-Teil

Der Implementation-Teil einer Unit beginnt mit dem reservierten Wort **IMPLEMENTATION**. Alles, was im Interface-Teil deklariert ist, ist für den Quelltext im Implementation-Abschnitt zugänglich.

Die Implementation kann über eigene, zusätzliche Deklarationen verfügen, die aber für andere Units oder Anwendungen nicht zur Verfügung stehen.
"Aus dem Implementation-Teil dringt nichts nach außen."
Die Deklarationen hier im Implementation-Teil werden von den in dieser Unit deklarierten Prozeduren, Funktionen und Ereignisbehandlungsroutinen verwendet.

Optional kann im Implementation-Teil eine Uses-Anweisung eingesetzt werden, die direkt auf das reservierte Wort IMPLEMENTATION folgen muss.

Der Quelltext der in der Schnittstelle Interface deklarierten Routinen muss im

Implementation-Teil stehen. In diesem Abschnitt können aber auch Deklarationen von "Sammlungsdaten" wie in Schritt 2 und von Prozeduren und Funktionen erfolgen, die nur hier (und nicht im Interface-Teil) aufgeführt werden. Der Gültigkeitsbereich bezieht sich dann dementsprechend nur auf diesen Abschnitt. Wir wissen ja: "Aus dem Implementation-Teil dringt nichts nach außen."

e. Der Initialization-Teil

Wenn Daten, die eine Unit benutzt, auf korrekte Anfangswerte gesetzt werden sollen (initialisieren), dann erreicht man das durch Einfügen eines INITIALIZATION-TEIL in die Unit. Das reservierte Wort **INITIALIZATION** wird - wie in Schritt 3 aufgezeigt - in die Unit eingesetzt. Zwischen diesen beiden Wörtern INITIALIZATION und END schreibst du deinen Quelltext, der die Daten initialisiert.

```
initialization
  {Hier steht der Initialisierungs-Quellcode}
end.
```

Wenn eine Anwendung eine Unit verwendet, wird zuerst der Quelltext im Initialization-Abschnitt der Unit aufgerufen, bevor anderer Quelltext der Anwendung ausgeführt wird. Wenn die Anwendung mehr als eine Unit verwendet, wird der Quelltext aller Initialisierungsteile aufgerufen, ehe der Rest der Anwendung ausgeführt wird.

f. Der Finalization-Teil

Wenn die Unit einige Aufräumarbeiten ausführen muss, wenn einige Ressourcen freigegeben werden müssen, die im Initialization-Teil initialisiert wurden, kann das in einem anschließenden **FINALIZATION**-Teil ausgeführt werden.

Dieser Abschnitt wird - wie unter Schritt 3 zu ersehen - ans Ende angefügt. Wo ein finalization-Abschnitt verwendet werden soll, muss vorher ein initialization-Teil vorhanden sein.

```
initialization
  { Hier steht der Quelltext zur Initialisierung }
finalization
  { Hier kommt der Quelltext für die "Aufräumarbeiten" hin }
end.
```

Wenn eine Anwendung unterbrochen wird, führt sie die finalization-Abschnitte der Unit in umgekehrter Reihenfolge aus.

2. Units, die der Nutzer erzeugt

Du kannst grundsätzlich jeder Unit, die Delphi erzeugt, Quelltext hinzufügen. Das ist kein Problem. *Aber du solltest den von Delphi generierten Quelltext auf keinen Fall verändern.* Nur wenn Delphi diesen Text verändert, wird das System diese Änderung registrieren und entsprechend darauf reagieren. Im anderen Fall weiß Delphi nichts von deiner Änderung: Die Folge kann ein Absturz des Systems sein.

Um eine eigene Unit zu erzeugen, kannst Du folgendermaßen vorgehen:
Wähle *Datei|Neu|Unit* aus und klicke auf <OK>.
Delphi stellt Dir daraufhin ein Gerüst für Deine Unit zur Verfügung.
So sieht der Quelltext "deiner" Unit jetzt aus:

```
unit Unit2;
interface
implementation
initialization
end.
```

Der Name, den Delphi der neuen Unit gibt, richtet sich nach der Anzahl der in deinem Projekt bereits vorhandenen Units. Du kannst dieses Grundgerüst anschließend mit dem von dir gewünschten Quelltext auffüllen.

Bitte sieh dir dazu auch noch einmal das an, was zu Anfang dieses Abschnittes über Units gesagt wurde, die Delphi selbst erzeugt. So wirst du manches besser verstehen.

Um dieses Dateiengerüst erst einmal zu sichern, speicherst du es unter uDefin.pas in Deinem Projekt-verzeichnis ab.

Wenn du dein Projekt jetzt compilierst, wirst du in dem entsprechenden Verzeichnis eine neue Datei uDefin.dcu vorfinden. Diese neu erzeugte Datei mit der Endung *.dcu (Delphi-compilierte-Unit) enthält Maschinencode, der zur Programmdatei deines Projekts hinzugelinkt wird.

Wenn du diese Unit mit Uses in ein Programm einfügen würdest, gäbe es keinerlei Reaktion; denn die *.pas-Datei uDefin.pas ist ja noch leer.

Rufen wir sie also wieder auf und fügen wir die "Sammlung" aus Abschnitt 2 - mit nur einer (!) Inc-ludedatei - in den Abschnitt nach interface ein. Wenn wir das Programm jetzt nochmals compilieren, kann diese Definitionsdatei in ein Programm mit Uses eingebunden werden, um dort von verschiedenen Stellen des Projekts aus angesprochen zu werden.

Wir wollen die Gelegenheit nutzen, um an dieser Stelle einen ganz praktischen Hinweis zu geben:
Wenn ich des öfteren Programme mit Delphi schreibe, werde ich ja oftmals auch die gleichen Prozeduren und Funktionen verwenden. Um diese nicht jedes Mal neu schreiben zu müssen, kann ich diese - wie zuvor beschrieben - in eine eigene Unit einfügen. Damit schaffe ich mir mein eigenes "OwnDelphi 1.0".

Die Datei OwnDelph.pas könnte dann so aussehen:

```
unit OwnDelph;

interface

uses
  Windows, Forms, SysUtils, Controls, ...;

procedure Pause (Zeit: LongInt);
// Realisiert eine Pause von <Zeit> Millisekunden

implementation

procedure Pause (Zeit: LongInt);
// Pause in Millisekunden
var
  ZeitVar: LongInt;
begin
  ZeitVar := GetTickCount;   // eine API-Funktion ohne Parameter
```

```
  repeat
    Application.ProcessMessages
  until (GetTickCount - ZeitVar > Zeit)
end {Pause};

end.
```

Das als Beispiel für eine eigene Unit, die nach und nach entsprechend erweitert werden kann.

In einem Programm eingebunden sieht das dann so aus:

```
unit Formular;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, OwnDelph;

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

end.
```

Noch einige "Schnäppchen"

Um es an einem anderen Beispiel zu zeigen:
Wenn du vielleicht ein Farbgitter in dein Formular einfügst, hängt Delphi automatisch die Bezeichnung "ColorGrd" an das Ende der uses-Anweisung der Unit an. Auf diese Weise steht dem Programm der volle Funktionsumfang der Unit ColorGrd zur Verfügung.

In Units sind jedoch auch einige Objekte deklariert,
die keine Entsprechung in Form visueller Komponenten in der Delphi-Umgebung haben. Wenn du beispielsweise vordefinierte Meldungsfenster im Quelltext einer Unit verwenden möchtest, musst du die Unit MsgDlg selbst in die entsprechende uses-Anweisung einfügen. Das gleiche trifft zu, wenn du das Objekt TPrinter aus der Unit Printers einsetzen möchtest.

Um eine Routine zu verwenden, die in einer anderen Unit deklariert ist:
Stell dem Namen der Routine den Namen der Unit voran, in der sie deklariert ist. Um zum Beispiel aus UnitForm eine Funktion BerechneZahlung aufzurufen, die in UnitNew deklariert ist, rufe auf:

```
Anzahl := UnitNew.BerechneZahlung (1000, .100, 50);
```

Du kannst jedem beliebigen Bezeichner - Eigenschaft, Konstante, Datentyp, Variable oder Routine - den Unit-Namen voranstellen.

Die Anweisung uses im Implementation-Teil:
Object Pascal gestattet das Einfügen einer optionalen uses-Anweisung im Implementation-Teil einer Unit. Wenn das geschieht, muss es direkt nach dem reservierten Wort implementation stehen.

Mit einer uses-Anweisung im Implementation-Teil kannst du die inneren Details einer Unit verbergen, weil Units, die im Implementation-Teil angegeben sind, für die Anwender einer Unit nicht zugänglich sind.

Abschluss

Warum ich dieses Tutorial geschrieben habe

Eine Frage im Forum in der letzten Zeit und eine Antwort dazu sowie ein Beitrag im Internet, der die Angelegenheit Units sehr lückenhaft darstellte, haben mich bewogen, doch einmal ausführlicher dazu Stellung zu nehmen.

Vor allem war es aber ein Tutorial, das ich schon seit einiger Zeit überarbeite. Dem füge ich einige neue Fakten und Beispieldateien hinzu. Dabei ergab sich die Verwendung der Includedateien zur Lösung des beschriebenen Problems.

Außerdem geisterten in manchen Überlegungen einige der hier verwendeten Gedanken herum, so dass die nun auch eine "Heimat" gefunden haben.

Ich hoffe nun, du kannst einiges aus diesem Tutorial als Denkanstoß gebrauchen. Das wäre mir schon die Mühe wert gewesen.

Autor:
Dieses Tutorial wurde uns freundlicherweise von
Siegfried Thoß zur Verfügung gestellt.
E-Mail: siegfried.thoss@t-online.de
