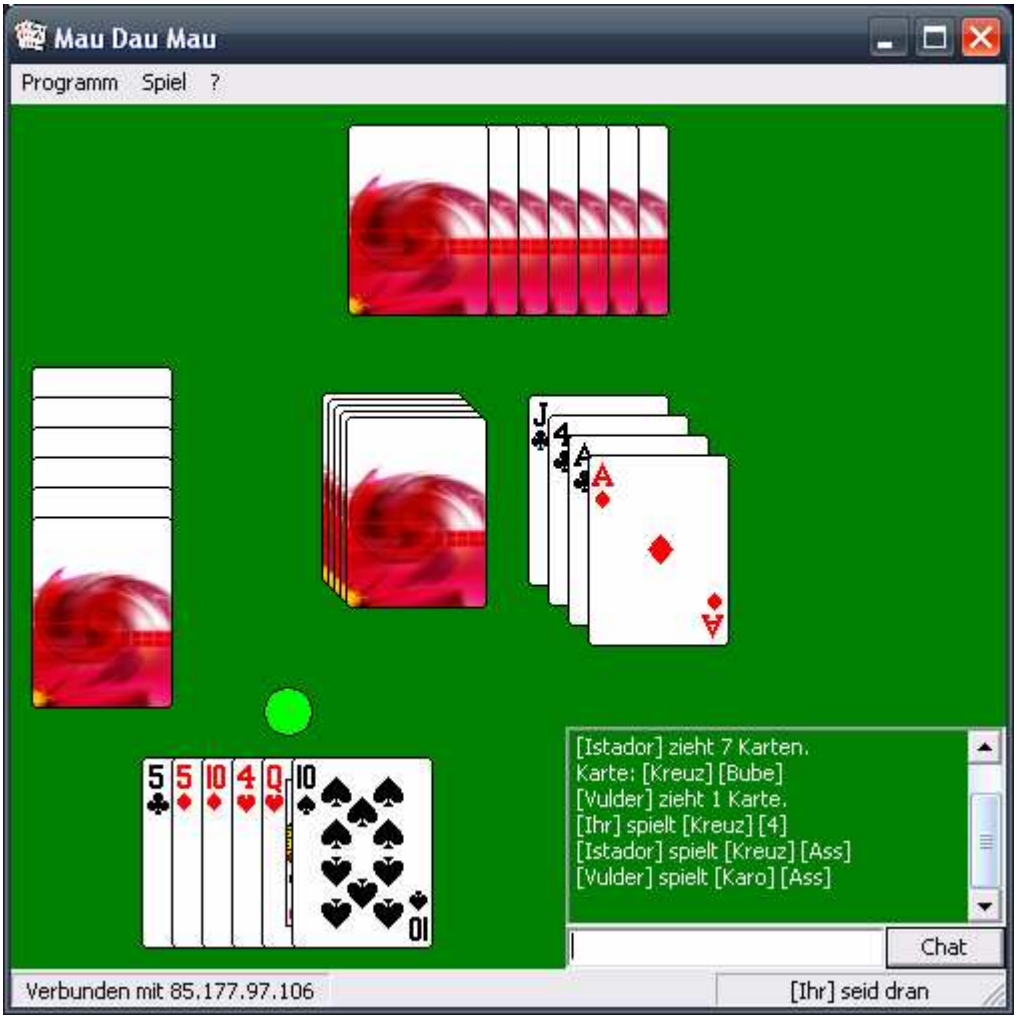
Techniklehre Projekt

Mau Dau Mau



von

Robin Christopher Ladiges

Schüler der Klasse FOH8b

Inhaltsverzeichnis

1. Einleitung

Dieses Schriftstück ist an andere Programmierer gerichtet und erläutert den grundlegenden Aufbau meines Projekts und reißt einige programmiertechnische Themen ausführlicher an.

Die Anwendungen dieses Projektes wurden mit Borlands „Turbo Delphi“ erstellt, die Anleitung mit Microsofts „HTML Help Workshop“.
Verwendete Software für die Erstellung des Projektes befindet sich zu Teilen auf der CD im Ordner „Software“.

Alle Quelldateien zu meinem Projekt befinden sich auf der CD im Ordner „Quelldateien“. In ihm befindet sich ein Ordner „Ressourcen“, in dem selbst erstellte Bilder zu finden sind und ein Ordner „Programm“, in dem sich die einzelnen Programmteile befinden.
Nachfolgendes bezieht sich auf Unterordner des Ordners „Programm“ im Ordner „Quelldateien“.

Im Ordner „bin“ befinden sich die kompilierten Dateien.

Im Ordner „executable“ befinden sich die Quelldateien zur „*MauDauMau.exe*“, der Spiel-Anwendung.

Der Ordner „dll“ beinhaltet die Quelldateien zur „*MauDauMau.dll*“ meiner eigenen DLL, in der ich einige Methoden der „*MauDauMau.exe*“ auslagere.

Die Quelldateien der „*MauDauMau.chm*“, der Anleitung des Spieles, befinden sich im Ordner „Hilfe“ und lassen sich mit dem „HTML Help Workshop“ von Microsoft öffnen. Zusätzlich lässt sich mit der Datei *„index.html“* eine Vorschau der Hilfe anzeigen.

Beim Einlegen der CD wird die „*Autostart.exe*“ ausgeführt, dessen Quelldateien sich im Ordner „autostart“ befinden.

Die „*Autostart.exe*“ öffnet unter anderem die *„Install.exe“*, um das Spiel zu installieren. Die Quelldateien der *„Install.exe“* befinden sich im Ordner „install“. Um das installierte Programm vom Rechner zu deinstallieren, existiert die *„Uninstall.exe“,* dessen Quelldateien sich im Ordner „uninstall“ befinden.

Für Probleme mit der Anleitung, die durch den Microsoft Sicherheits-Patch KB896358 auftreten können, hilft die *„chmfix.exe“*, ihre Quelldateien sind im Ordner „chmfix“ zu finden.

2. Aufbau der Spiel-Anwendung

Die Spiel-Anwendung *„MauDauMau.exe“* öffnet beim Start die **Form1** in der **FMainform** Unit. Die **Form1** ist das Haupt-Anwendungsfenster, von dem unter anderem auch gespielt wird. Von der **Form1** aus werden die meisten anderen Formulare und Units eingebunden, initialisiert und aufgerufen.

Der Großteil der Spiel-Anwendung besteht aus eigenen Klassen. Jede Klasse hat meistens eine eigene Unit, um das Projekt übersichtlich zu halten.

Damit die Dialogfelder der Methoden **ShowMessage** und **InputQuery** in der Mitte des Formulars, von wo aus die Methoden aufgerufen werden, und nicht in der Mitte des Bildschirms angezeigt werden, habe ich Abwandlungen der beiden Methoden in der **UDialogs** Unit, zusammen mit den von ihnen benötigten Methoden.

Die **Udll** Unit bindet die Methoden der *„MauDauMau.dll“* sowie der *„Cards.dll“* von Microsoft ein und bietet anderen Units diese Methoden an durch Einbinden der **Udll** Unit in deren „uses“ Klausel.

Zum Lesen und Speichern der Einstellungen in der *„MauDauMau.ini“* Datei wie z.B. des ausgewählten Deckblattes, habe ich die **Uini** Unit. Sie bietet anderen Units ein Objekt namens **config**, über welches sie auf die Einstellungen bei Bedarf zugreifen können.

Die **ULanguage** Unit lädt Zeichenketten aus der *„language.rcl“* Datei entsprechend der Sprache, die eingestellt wurde. Sie bietet anderen Units ein Objekt namens **lang,** über das sie die Zeichenketten bei Bedarf auslesen und entsprechend ausgeben können.

In der **UClient** Unit habe ich ein Objekt namens **Client** vom Typ **TClient**.
Der Typ **TClient** ist eine Klasse, welche die Verbindung mit dem Server regelt. Sie liest die Nachrichten, die vom Server kommen und führt bei bestimmten Anweisungen den dafür bestimmten Code aus wie z.B., eingehende Textnachrichten in die Chatfelder zu schreiben. Die **TClient** Klasse beinhaltet Eigenschaften wie z.B., die Regeln, die auf diesem Server gelten oder die Namen der verbundenen Spieler.
In der **TClient** Klasse befindet sich ein Unterobjekt namens **game** vom Typ **TClientGame,** das erstellt wird, wenn eine Runde gestartet wird.

TClientGame ist eine Klasse in der **UClientGame** Unit und stellt abstrakt das Spiel dar. Sie enthält Eigenschaften wie z.B. ob man selbst gerade dran ist oder wie viele Karten gezogen werden müssen.
Die **TClientGame** Klasse beinhaltet unter anderem ein Objekt namens **Form** vom Typ **TClientForm**. Diese Klasse beinhaltet Unterobjekte der **UClientPos** Unit, welche im Einzelnen beschreiben, wo sich die sie betreffenden sichtbaren Objekte auf der Form befinden. So ist in der **TClientForm** z.B. ein Objekt namens **Player0** vom Typ **TClientPosPlayer0** aus der **UClientPos** Unit, welches die Karten vom Typ **TClientCard** aus der **UClientCards** Unit, die der Spieler 1 hat, enthält und verwaltet.

Auf Server-Seite ist die **TServer** Klasse in der **UServer** Unit das Gegenstück zur **TClient** Klasse der **UClient** Unit und verwaltet alle Clients, die mit ihm verbunden sind sowie die Nachrichtenübertragung mit ihnen.
Anhand der Nachrichten, die die **TServer** Klasse von den Clients bekommt, entscheidet sie was getan werden muss. So leitet sie den Spielablauf betreffende Nachrichten an ein ihr untergestelltes Objekt namens **game** vom Typ **TServerGame** in der **UServerGame** Unit weiter.
Die **TServerGame** Klasse stellt das Herz des Spieles dar. Sie speichert alle für die Spiellogik entscheidenden Informationen (z.B. welcher Spieler dran ist oder welche Karte wem gehört) und entscheidet mit Hilfe von Methoden ob Spielzüge, die die Clients ausführen möchten, gültig sind oder nicht. Sie teilt ihre Entschlüsse über den Server den Clients mit.

3. Sichtbare Karten auf der Form

3.1 Vererbung

Um Karten in meiner Spiel-Anwendung auf dem Haupt-Anwendungsfenster anzeigen zu lassen, benutze ich **TImage** Objekte.
Genauer gesagt, benutze ich Objekte vom Typ meiner eigenen Klasse namens **TClientCards** in der **UClientCards** Unit, welche eine Tochterklasse von **TImage** ist. Das bedeutet, die **TClientCards** Klasse hat von Beginn an die gleichen Komponenten (Felder, Methoden und Eigenschaften) wie die **TImage** Klasse.

Der Vorteil hierbei ist, dass ich die vorhandenen Komponenten als gegeben und funktionierend hinnehmen kann. Die Klasse **TClientCards** kann ich so selbst beliebig und unkompliziert um weitere Komponenten ergänzen.

So speichere ich in meinen Objekten vom Typ **TClientCards** z.B. die Information welche Karte es ist. (Mit Karte meine ich, welche Farbe und welchen Wert sie hat)

3.2 Deklaration der TClientCards Klasse

Hier ein verkürzter Auszug der Deklaration:

```
//## TClientCards Deklaration aus units/UClientCards.pas ##################
type TClientCards = class(TImage)
     public
      card: Integer;
      side: 0..1;
      position: 0..4;
      player: -1..4;
      constructor create(AOwner:TComponent; card,side,width,
                         height:integer); reintroduce;
      procedure imagerefresh;
      //(...)
     end;
//#########################################################################
```

Das Feld(Variable) **card** gibt an welche Karte es ist, also ob es sich z.B. um eine Karo Zehn oder eine Kreuz Sieben handelt. **Card** arbeitet hier mit dem gleichen Karten-Index, den auch die *„cards.dll“* Prozedur **cdtDraw** benutzt (siehe den Abschnitt „3.3 Cards.dll“).

Für die Deckblätter ist das Feld **side** wichtig, da hiermit differenziert wird, ob es sich um eine aufgedeckte oder eine verdeckte Karte handelt.

Die Felder **position** und **player** dienen zur Unterscheidung, ob die Karte einem Spieler gehört oder ob sie keinem Spieler gehört. Kurz: wenn **player** Null ist, gehört sie keinem Spieler, ist also eine offene Karte oder eine Karte, die sich im Stapel befindet. Ist **player** gleich Eins, ist die Karte unsichtbar (die Eigenschaft **Visible** wird auf *false* gesetzt), was lediglich zu Beginn eines Spieles bei den offenen Karten vorkommt. Wenn **player** zwischen Eins und Vier ist, gehört die Karte einem Spieler. Wobei hier beim Client gilt, dass der Benutzer selbst Player1 ist und die anderen Mitspieler aufsteigend nach Reihenfolge auf dem Spielfeld Player2, Player3 oder Player4.
Das Feld **position** gibt an wo sich die Karte befindet. Wenn **position** Null ist, befindet Sie sich entweder auf der Hand eines Spielers oder (abhängig von **player**) ist eine Deckkarte. Wenn **position** zwischen Eins und Vier ist, ist sie eine der offenen Karten.

#	0	1	2	3	4	position
-1	n/a	invisible	invisible	invisible	invisible	
0	im Deck	Offen-1*	n/a	n/a	n/a	
1	Handkarte	Offen-1	Offen-2	Offen-3	Offen-4	
2	Handkarte	Offen-1	Offen-2	Offen-3	Offen-4	
3	Handkarte	Offen-1	Offen-2	Offen-3	Offen-4	
4	Handkarte	Offen-1	Offen-2	Offen-3	Offen-4	

player

invisible = unsichtbar

n/a = not available
kommt nie vor

Offen-1* = erste Karte

Als Nachfahre von **TImage** erbt die Klasse **TClientCards** unter anderem den constructor **create**. Da ich den constructor jedoch gerne verändern möchte, deklariere ich ihn in meiner Klasse **TClientCards** neu, um Objekten vom Typ **TClientCards** beim Erstellen gleich Informationen, welche Karte sie darstellen soll oder welchem Spieler sie gehört, mitzugeben.

Die Prozedur **imagerefresh** zeichnet die eigentliche Karte mit Hilfe der *„cards.dll“* Prozedur **cdtDraw** aufgrund der Werte der Felder des Objektes.

Rausgekürzt sind hier die Prozeduren für die Ereignisse, auf die ich in dieser Dokumentation aus Platzgründen nicht weiter eingehen werde.

3.3 Cards.dll

Die *„Cards.dll“* ist eine Microsoft-Programmbibliothek, die es verschiedenen Programmen erlaubt, auf die in ihr gespeicherten Bilder von Spielkarten zuzugreifen. Da die DLL nicht den direkten Zugang zu den Bildern ermöglicht, bietet sie nach außen verschiedene Methoden an, um die Karten anzuzeigen.

Die für mein Projekt wichtigste Methode aus der *„Cards.dll“* ist die **cdtDraw** Funktion. Sie zeichnet auf einen ihr übergebenen *Canvas Handle* die von ihr verlangte Karte.

```
//## cdtDraw Einbindung aus units/Udll ##################################
function cdtDraw(hdc:HDC; x,y,card,typ:integer; color:DWORD):boolean;
         stdcall; external 'cards.dll';
//#####################################################################
```

Der Funktion müssen einige Werte übergeben werden.
Zum einen den schon erwähnten *Canvas Handle* namens **hdc**. Da jedes grafische Objekt in Delphi einen *Canvas Handle* besitzt und unsere Objekte vom Typ **TClientCards** alle von der **TImage** Klasse, eine für grafische Objekte bekannte Klasse, erben, übergeben wir der Funktion den eigenen *Canvas Handle*.

Die Übergabeparameter **x** und **y** sagen der Funktion wo die Karte auf dem *Canvas Handle* gezeichnet werden soll.

Card gibt an, welche Karte gezeichnet werden soll und setzt sich aus folgender Formel zusammen:

Karte = Farbe + Wert * 4

Für Farbe gilt: Kreuz=0, Karo=1, Herz=2, Pik=3
Und für Wert gilt: Ass=0, zwei=1, drei=2, vier=3, fünf=4, sechs=5, sieben=6, acht=7, neun=8, zehn=9, Bube=10, Dame=11, König=12.
Daraus folgt, dass eine Karo Drei den Wert 9 hat (1+2*4=9).
Die Deckblätter haben **card** Werte von 54 bis 65.

Typ sagt der Funktion, ob die Vorder- oder Rückseite gezeichnet werden soll. Wobei 0 die Vorder- und 1 die Rückseite ist, was für Deckblätter relevant ist.

Color hat nur Einfluss auf die Hintergrundfarbe von **card** 53, die ich nirgends verwende.

3.4 Zeichnen der Karte

Wie bereits zuvor erwähnt, zeichnet die **imagerefresh** Prozedur die Karten mit Hilfe der **cdtDraw** Funktion auf das eigene Objekt.

Hierbei gibt es eine Besonderheit von **TImage** Objekten zu beachten.
TImage Objekte sind nicht sichtbar solange nichts auf ihnen gezeichnet wurde. Sobald etwas auf ihnen gezeichnet wird, werden die Objekte sichtbar und wir erkennen, dass der gesamte Hintergrund der Objekte weiß ist.
Wenn auf diesem weißen Hintergrund nun die Karte mit den abgerundeten Ecken gezeichnet wird, haben alle Karten weiße Ecken.

Nun haben **TImage** Objekte ein nützliches Feld, mit dem sich eine bestimmte Farbe transparent darstellen lässt. Dieses Feld heißt **Transparent**, welches ich im constructor für alle Karten automatisch auf *true* setze. Jetzt muss man jedoch wissen wie **TImage** Objekte mit diesem Feld umgehen. Ist dieses Feld gesetzt, so wird die Farbe transparent, die das 1. Pixel links oben hat.
Da der Hintergrund der **TImage** Objekte weiß und die Ecken der Karten abgerundet sind, wird die Farbe weiß transparent.
Da ich das nicht möchte, weil fast alle Karten überwiegend weiß sind, bemale ich den Hintergrund der Objekte in einer anderen, vorher im constructor definierten, Farbe (clLime), bevor ich die Karten von der Funktion **cdtDraw** auf das Objekt zeichnen lasse.

```
//## TClientCards.imagerefresh Prozedur aus units/UCLientCards.pas ########
procedure TClientCards.imagerefresh;
begin
 self.Repaint;
 self.Canvas.Rectangle(0, 0, self.Width, self.Height);
 cdtDraw(self.Canvas.Handle, 0, 0, self.card, self.side, 0);
 self.Refresh;
end;
//############################################################################
```

Überzeichnen erlauben
Eigenen Hintergrund "clLime" einfärben
Karte mit DLL-Funktion "cdtDraw" aufgrund der Werte der eigenen Felder zeichnen
Änderungen durch "refresh" sichtbar machen

4. Nachrichtenübertragung Server Client

4.1 TServerSocket und TClientSocket

Für meine Netzwerkumsetzung benutze ich eigene Tochterklassen der **TClientSocket** und **TServerSocket** Klassen von Borland, um sie mit eigenen Komponenten (Felder, Methoden und Eigenschaften) zu ergänzen.

Die **TClientSocket** Klasse hat drei wichtige Felder (Variablen), die für die Verbindung zum Server von Bedeutung sind.

1. **Port**: Der Port, der bei dem Server zur entsprechenden Anwendung führt.
2. **Host**: Hostname oder IP-Adresse des Servers, zu dem die Verbindung aufgenommen werden soll.
3. **active**: Startet beim Setzen auf *true* die Verbindung oder beendet sie beim Setzen auf *false*.

Der Client und der Server können den jeweils anderen bei bestehender Verbindung einen String mit der Prozedur „**Socket.SendText(text: string);**“ senden. Wenn der Server dem Client eine Nachricht schickt, erfährt der Client das über das Ereignis **OnRead** und kann die Nachricht über die Funktion „**Socket.ReceiveText;**“ auffangen, welche die entsprechende Zeichenkette als Funktionswert zurückliefert. Der Server bekommt über das Ereignis **OnClientRead** mitgeteilt, dass ein Client ihm etwas schickt und fängt es über die Funktion „**Socket.ReceiveText;**“ auf.

4.2 Part Funktion

Die vom Server oder Client versendeten Nachrichten sollten klar definierte Befehle sein, die man in der weiteren Programmierung mit Vergleichen differenziert und entsprechendem Code ausführt.
Zusätzlich dazu sollte man sich eine bestimmte Schreibweise überlegen wie man verschiedene Parameter, die man mit einem Befehl übergibt, voneinander trennt.

Ich habe mir überlegt, dass Nachrichten, die ich als Zeichenketten zwischen Server und Client schicke, mit einem Befehlsnamen anfangen und dann mit einem Semikolon enden. Parameter werden hinten angehängt. Jeder einzelne Parameter endet mit einem Semikolon. Um Fehlern vorzubeugen muss darauf geachtet werden, dass der Benutzer nicht die Möglichkeit besitzt Semikolons in Eingabefelder einzugeben.

So ist z.B. der Befehl, mit dem der Server den Clients Textnachrichten von anderen Spielern mitteilt, folgendermaßen aufgebaut:

text;Benutzername;Stunde;Minute;Nachricht;

index: 0 1 2 3 4

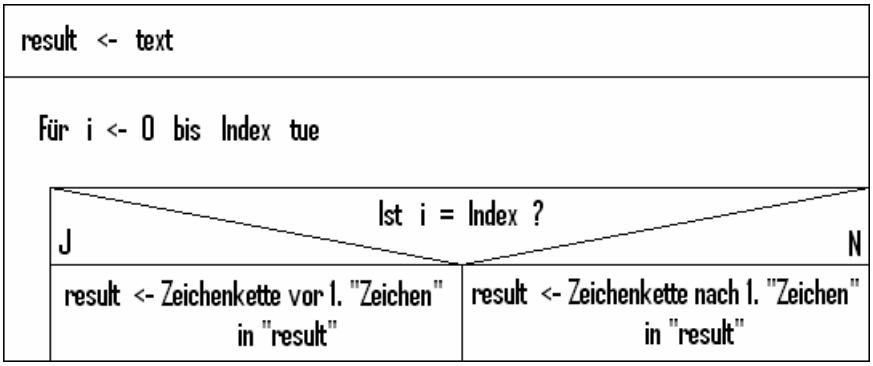
text ist der Befehlsname.
Benutzername ist der Name des Absenders der Textnachricht.
Stunde und **Minute** bilden zusammen die Uhrzeit zu der die Textnachricht beim Server eingegangen ist.
Nachricht ist die Textnachricht, die der Absender der Textnachricht eingegeben hat.

Wenn z.B. der Spieler „Spieler1“ in den Chat um 22:11 Uhr die Nachricht „Hallo Welt“ eingibt und abschickt, so geht bei den Clients die folgende Nachricht vom Server ein:

text;Spieler1;22;11;Hallo Welt;

Diese eingehenden Nachrichten, müssen ausgewertet werden, um zu differenzieren welcher Befehl eingeht und wie vorgegangen werden muss. Dafür habe ich eine Funktion namens **part**, der die gesamte Zeichenkette, das trennende Zeichen (in diesem Fall das Semikolon) und der Index des Parameters, den die Funktion zurückgeben soll, übergeben wird.

```
//## part Funktion ###############################################################
function part(Zeichen:Char; Index:Integer; text:string):string;
var i: Integer;
begin
 result:=text;
 for i := 0 to Index do
  if i = Index then
   result:= Copy(result, 1, Pos(Zeichen,result)-1)
  else
   result:=Copy(result,(Pos(Zeichen,result)+1),
                (length(result)-Pos(Zeichen,result)));
end;
//#############################################################################
```



Die Funktion beginnt damit, die ihr übergebene Zeichenkette in die **result** Variable zu übergeben, mit der im weiteren Funktions-Verlauf gearbeitet wird und die nach Ablauf der Funktion zurückgegeben wird.

Danach startet eine Schleife, die nach und nach jeden Zeichenkettenblock durchgeht bis zum Block mit dem übergebenen **Index**.
Jeder Block steht für einen anderen Parameter bzw. Befehlsnamen.
Bei jedem Schleifendurchlauf wird ein Vergleich durchgeführt, der überprüft ob der aktuelle Schleifenindex **i** mit dem gesuchten **Index** des auszugebenden Zeichenkettenblockes übereinstimmt.

Stimmt der Schleifen**i**ndex mit dem Zeichenkettenblock-**Index** überein (der „if“-Abschnitt), was der Fall ist, wenn die Schleife das letzte Mal ausgeführt wird, wird die **result** Variable mit dem Zeichenkettenabschnitt vor dem ersten trennendem **Zeichen** (dem Semikolon) beschrieben. Dies geschieht mit der **Copy** Funktion von Borland aus der **System** Unit, welche aus einer Zeichenkette einen bestimmten Bereich herausschneidet und zurückgibt. Der Bereich, der herausgeschnitten wird, ergibt sich aus den Positionen von wo aus herausgeschnitten werden soll sowie der Länge der herauszuschneidenden Zeichenkette. So wird der **Copy** Funktion die Position 1 für das 1. Zeichen sowie die Position des letzten zurückzugebenden Zeichen übergeben, welches gleichzeitig, da wir beim 1. Zeichen beginnen, auch die Länge der herauszuschneidenden Zeichenkette ist.
Das zuletzt zurückzugebende Zeichen ist das Zeichen vor dem ersten trennenden **Zeichen** (Semikolon). Die Position des trennenden **Zeichen**s (Semikolon) bekommen wir über die **Pos** Funktion in der **System** Unit, welche die erste Position einer ihr übergebenen Zeichenkette (in diesem Fall nur eines **Zeichen**s) in einer anderen ihr übergebenen Zeichenkette zurückgibt, die wir dann mit 1 subtrahieren.

Wenn der Schleifen**i**ndex nicht mit dem Zeichenkettenblock-**Index** übereinstimmt (der „else“-Abschnitt), was jedes Mal der Fall ist, wenn wir nicht im gesuchten Block sind, wird die **result** Variable mit dem Zeichenkettenabschnitt nach dem ersten trennenden Zeichen (dem Semikolon) beschrieben.
Hierbei wird mit der **Copy** Funktion nicht vom 1. Zeichen aus zurückgegeben, sondern vom 1. Zeichen nach dem ersten trennenden **Zeichen**.
Die Position nach dem ersten trennenden **Zeichen** erhalten wir, wenn wir die Position des trennenden **Zeichen**s, die wir von der **Pos** Funktion als Rückgabewert erhalten, per Addition mit 1 erhöhen.

Da der **Copy** Funktion die Länge der herauszuschneidenden Zeichenkette übergeben werden muss, subtrahieren wir von der gesamten Länge der Zeichenkette, die wir von der **length** Funktion zurückgegeben bekommen, die Länge der Zeichenkette, die wir nicht herausschneiden (den ersten Parameter-Block inklusive Semikolon), welche gleich ist mit der Position des ersten trennenden **Zeichen**s (Semikolon), weil die nicht herauszuschneidende Zeichenkette beim ersten Zeichen beginnt.

Der erste Block wird quasi nach und nach herausgenommen bis wir beim gesuchten Block sind. Beim gesuchten Block wird alles hinter ihm abgeschnitten, und der gesuchte Block wird von der Funktion zurückgegeben.

4.3 Part Funktion zur DLL auslagern

Die **part** Funktion habe ich in der *„MauDauMau.dll“* ausgelagert. Bei DLL - Zugriffen muss darauf geachtet werden, dass keine Zeichenketten vom Datentyp **string** übergeben werden, da es ansonsten zu Laufzeitfehlern führen kann. Deshalb habe ich die **part** Funktion auf den Datentyp **PChar** abgeändert und in **partpchar** umbenannt:

```
//## partpchar Funktion aus dll/MauDauMau.dpr ##############################
function partpchar(zeichen:Char; index:Integer; text:PChar):PChar; stdcall;
var i: integer;
begin
 result:=text;
 for i:=0 to index do
  if i = index then
   result:=PChar(Copy(result, 1, Pos(zeichen,result)-1))
  else
   result:=PChar(Copy(result, (Pos(zeichen,result)+1),
                 (length(result)-Pos(zeichen,result))));
end;
//########################################################################
```

Die **PChar** Funktion dient hier zur Umwandlung des Rückgabewertes der **Copy** Funktion vom Datentyp **string** zum Datentyp **PChar**.
Da ich in meinem Projekt fast ausschließlich mit Zeichenketten des Datentyps **string** arbeite und nicht jedes Mal, wenn ich die Funktion aufrufe, manuell die Übergabeparameter in den Datentyp **PChar** und den Rückgabeparameter in den Datentyp **string** umwandeln will, habe ich in der Spiel-Anwendung eine Funktion namens **part.** welche die Umwandlung vornimmt und die **partpchar** Funktion aufruft.

```
//## part Funktion aus /executable/units/Udll ##############################
function part(zeichen:char; index:integer; text:string):string;
begin
 result:=String(partpchar(zeichen, index, PChar(text)));
end;
//########################################################################
```

4.4 Verschachtelte Befehle

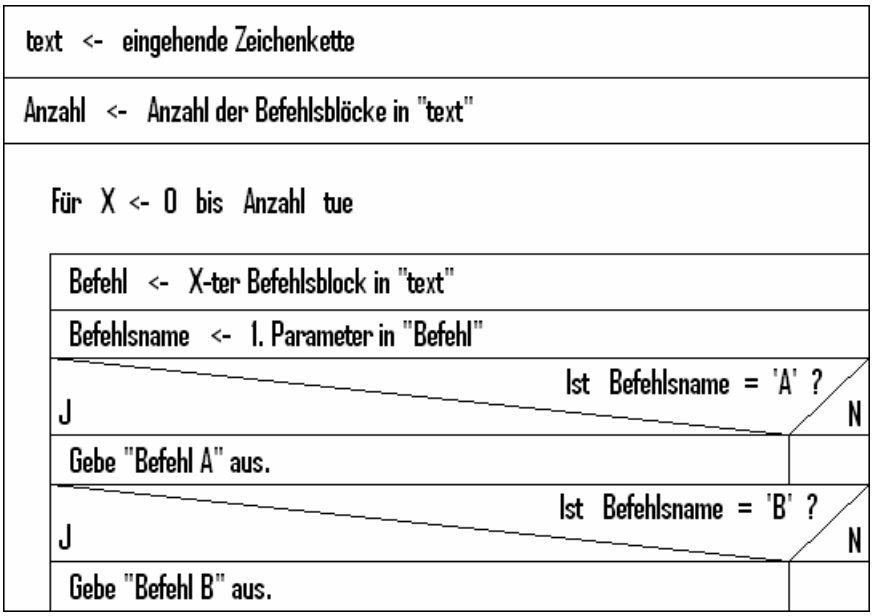
Weil der Server mitunter mehrere Nachrichten auf einmal den Clients schicken möchte, habe ich zusätzlich zur Trennung der Parameter innerhalb eines Befehles mit einem Semikolon noch eine weitere Trennung zur Differenzierung mehrerer Befehlsblöcke (Befehlsname plus Parameter mit Semikolon getrennt) ausgedacht. Diese Trennung beginnt mit der Anzahl der Befehle, damit der Client weiß, wie viele Befehle er abarbeiten muss, gefolgt von einer Raute und Befehlsblöcke, jeder Block mit einer Raute am Ende.

Möchte z.B. der Server dem Client die fiktiven Befehle „A;3;6;“ und „B;8;4;“ senden, so sähe das so aus:

2#A;3;6;#B;8;4;#

Mit der **part** Funktion, der wir anstelle des Semikolons eine Raute als Zeichen übergeben, können wir beim Index 0 die Anzahl der Befehle auslesen und mit dieser Anzahl eine Schleife aufbauen, die nach und nach alle Befehle abarbeitet.

```
//## Beispielquelltext ########################################################
text:= '2#A;3;6;#B;8;4;#';
Anzahl:= StrToInt(part('#', 0, text));
for X:=1 to Anzahl do
 begin
  befehl:= part('#', X, text);
  befehlsname:= part(';', 0, befehl);
  if befehlsname = 'A' then
   ShowMessage('Befehl A');
  if befehlsname = 'B' then
   ShowMessage('Befehl B');
 end;
//############################################################################
```



5. Rückblick

Rückblickend kann ich behaupten, die Zielsetzung umgesetzt zu haben.
Ich habe eine Spiel-Anwendung, die ziemlich übersichtlich und einfach zu bedienen ist. Sie fängt zumindest, die mir bekannten möglichen Fehlerquellen auf und verhindert Eingaben der Anwender, die zu Fehlern führen könnten.
Bei falschen Eingaben, Spielzügen oder bei Verbindungsproblemen teilt das Programm dies dem Anwender mit.
Es ist möglich, das Spiel mit bis zu vier Spielern stabil und ohne Fehler (soweit mir bekannt) im Netzwerk oder Internet zu spielen.
Die möglichen verschiedenen Regeln werden durch den Server korrekt umgesetzt.
Zusätzlich zu den aufgestellten Zielen ist noch die Multilingualität umgesetzt, die es erlaubt das Spiel in Deutsch, Englisch und Französisch zu spielen. (Letztere Sprache mit Übersetzung von Christian Stoffers).

Die in der Zielsetzung als Möglichkeit ausgesetzten anderen Erweiterungen für das Programm, die Sprachausgabe und das Spielen gegen Computergegner, wurde nicht umgesetzt. Gründe für die Nicht-Umsetzung der Sprachausgabe sind zum einen, dass sich einige Sätze, die ich ausprobiert habe, sich im Spiel als unpassend und unstimmig ergeben haben und zum anderen die Sprachausgabe in anderen Sprachen wohl von der Benutzung des Spieles abschrecken würde.
Die Möglichkeit gegen den Computer zu spielen ist nicht umgesetzt, da dieses Feature sehr zeitaufwändig wäre und viele Änderungen an den bereits vorhandenen Teilen des Programms erfordern würde. Die Zeit hierfür blieb mir nach der Umsetzung, der in der Zielsetzung festgelegten Ziele, nicht mehr zum Zeitpunkt der Projektabgabe.

Schwierigkeiten hatte ich bei der Umsetzung meines Projekts wenig.
Eigentlich hatte ich kaum bis gar keine Schwierigkeiten. Meistens waren es kleine Flüchtigkeits- oder Tippfehler, die mich stundenlang beschäftigt haben. Auf den Schul- und Nachhausewegen drehten sich meine Gedanken fast nur um mein Projekt. Ich überlegte wie ich bestimmte Methoden zu Hause am Rechner umsetzen kann, wodurch vieles gleich auf Anhieb funktionierte.

Das wohl größte Problem hatte ich wohl mit der Auslagerung der **part** Funktion in meine eigene DLL, da ich zuerst der Funktion Zeichenketten vom Datentyp **string** übergeben habe. Dies schien auf dem ersten Blick fehlerfrei zu funktionieren, führte aber bei der Anwendung unregelmäßig zu Fehlermeldungen. Die Fehlermeldungen konnte ich zwar auf die DLL und die **part** Funktion zurückführen, aber nicht deren Ursachen.
Das Problem zu identifizieren kostete mich viel Zeit, da ich davon ausging die Übergabe von Zeichenketten vom Datentyp **string** verlaufe (trotz Warnung von Borland) fehlerfrei, und ich die Fehler bei den zu übergebenen Parametern vermutete.

Die Idee zu diesem Projekt sowie Gedanken zu dessen Umsetzung in C++ hatte ich schon vor etwa eineinhalb Jahren während meiner Ausbildung zum Technischen Assistenten für Informatik. Jedoch fehlte es mir an Motivation und Zeit zur Umsetzung.
Angefangen mit der Umsetzung des Projektes in Delphi habe ich schon am ersten Tag, an dem ich erfuhr, dass wir ein Projekt in Delphi machen sollen. Von da an habe ich fast täglich mehrere Stunden daran gearbeitet.
Ich habe in dieses Projekt viel meiner Freizeit gesteckt, weil es mir Spaß macht zu programmieren, und ich bisher noch nie ein solch großes Projekt entwickelt habe, aber schon immer einmal machen wollte. Das dieses Projekt sehr zeitaufwändig und umfangreich sein würde, war mir beim Schreiben der Zielsetzung voll bewusst.
Rückblickend hatte ich viel Spaß und Freude an der Entwicklung des Projektes und bin stolz auf meine erbrachte Leistung.

6. Verwendete Quellen und Hilfen

Mir hat bei meinem Projekt keiner geholfen, abgesehen von Christian Stoffers bei der französischen Übersetzung sowie meiner Mutter durch Korrekturlesen meiner Texte. Programmiertechnisch habe ich alleine gearbeitet, in dem ich mir Lösungen selbst ausgedacht habe oder (z.B. die korrekte Schreibweisen für Klassendeklarationen) in der Entwicklungsumgebung beiliegenden Units nachgelesen habe.
Ab und zu habe ich im Internet nach bestimmten Teillösungen gesucht. Auf die dabei verwendeten Quellen möchte ich jetzt genauer eingehen:

„TTcpClient/TTcpServer“ auf entwickler-forum.de von Manfred Pawelzik
http://entwickler-forum.de/showthread.php?t=5325 aufgerufen am 06.09.2008
Die Information, dass das Paket „dclsockets100.bpl“ die Komponenten für die **TClientSocket** und **TServerSocket** enthält. (Das Paket ist bei Turbo Delphi nicht standardmäßig installiert)

„HTML-Hilfe“ auf delphi-treff.de von Martin Strohal
http://www.delphi-treff.de/tutorials/tools/html-hilfe/ aufgerufen am 07.09.2008
Dieses Tutorial lehrte mich wie man mit Microsofts „HTML Help Workshop“ chm-Hilfedateien erstellt sowie diese, mit Hilfe des "HTML Help Kit for Delphi" von der „Helpware Group“ (http://www.helpware.net/delhi), in Delphi einbindet.

„Delphi: DLLs einbinden“ auf fatman98.fa.funpic.de von „Fatman98“
http://fatman98.fa.funpic.de/delphi_dlls.php aufgerufen am 07.09.2008
Von dieser Quelle habe ich gelernt wie ich Methoden von DLLs in Delphi einbinde und eigene DLLs in Delphi erstelle.

„Cards.dll API“ auf catch22.net von „James“
http://www.catch22.net/tuts/cards aufgerufen am 10.09.2008
Durch diese Quelle erfuhr ich welche Methoden die Cards.dll hat und wie sie funktionieren. Da in der Quelle die Befehle nur als C++ Befehle angegeben sind, habe ich sie selbst für Delphi daraus abgeleitet.

„Delphi: Ini-Files bearbeiten“ auf fatman98.fa.funpic.de von „Fatman98“
http://fatman98.fa.funpic.de/delphi_ini.php aufgerufen am 13.09.2008
Diese Quelle lehrte mich die Existenz und den Umgang mit der **TIniFile** Klasse zum Zugriff auf „ini“-Dateien fürs Speichern und Lesen von Optionen, Sprache und Statistiken in meinem Projekt.

„Bei DragMode dmAutomatic keine Auswahl möglich“ auf delphipraxis.net
http://www.delphipraxis.net/topic96971.html aufgerufen am 14.09.2008
Der gesamte Forums-Thread half mir dabei zu verstehen wie ich meine **TClientCards** Karten per Drag & Drop bewegen und ablegen kann.

„Erstellen + Löschen von Komponenten zur Laufzeit“ auf wer-weiss-was.de
http://www.wer-weiss-was.de/theme159/article1957127.html aufgerufen am 16.09.2008
Die korrekte Schreibweise des Erstellens von Komponenten zur Laufzeit sowie die Information beim Zerstören die Variable auf **nil** zu setzen um überprüfen zu können ob die Komponente existiert oder nicht habe ich von dieser Quelle.

„Versions-abfrage in Delphi“ auf tutorials.de von „NIC140903“
http://www.tutorials.de/forum/delphi-kylix-pascal/14227-versions-abfrage-delphi.html
aufgerufen am 04.10.2008
Die **GetVersion** Funktion von „nexus“ um die Versionsnummer der Spiel-Anwendung auszulesen in der **FAuthor** Unit habe ich 1:1 übernommen. (Vorher hatte ich das nicht dynamisch sondern musste bei jeder neuen Versions-Nr. die Zeichenkette im Quelltext ändern)

„Problem beim Anzeigen von .chm-Dateien“ auf wintotal.de
http://www.wintotal.de/Tipps/index.php?id=1179 aufgerufen am 26.11.2008
Die Information warum bei einem Rechner auf dem ich meine Hilfedatei getestet habe der Inhalt der Hilfethemen nicht angezeigt wurde.

„Delphi: Zugriff auf die Registry“ auf fatman98.fa.funpic.de von „Fatman98“
http://fatman98.fa.funpic.de/delphi_registry.php aufgerufen am 04.12.2008
Von dieser Quelle erfuhr ich wie man mit der **TRegistry** Klasse Informationen in der Registry auslesen bzw. schrieben kann. Dies benötige ich zum einen beim Autostarter um zu überprüfen ob das Spiel bereits installiert wurde, sowie in der *„chmfix.exe“* um meiner Hilfedatei explizit zu erlauben Hilfethemen zu laden.

„Verknüpfung (*.lnk) zu einer Datei erstellen“ auf delphi-treff.de von Martin Strohal
http://www.delphi-treff.de/tipps/dateienverzeichnisse/wiki/Verkn%C3%BCpfung%20(*.lnk)%20zu%20einer%20Datei%20erstellen/ aufgerufen am 09.03.2009
CreateLink Funktion zur Erstellung von Verknüpfungen auf dem Desktop und im Startmenü. Habe ich 1:1 übernommen und verwende ich bei der Installation.

Erstellt und kompiliert habe ich die Anwendungen mit Borland Turbo Delphi:
Borland® Delphi® für Microsoft® Windows™ Version 10.0.2288.42451 Update 2
http://www.turboexplorer.com/delphi

Die Hilfe Datei *„MauDauMau.chm“* habe ich erstellt und kompiliert mit dem Microsoft® HTML Help Workshop Version 4.74.8702.0
http://www.microsoft.com/downloads/details.aspx?familyid=00535334-c8a6-452f-9aa0-d597d16580cc
und in Delphi eingebunden mit dem The Helpware Group Delphi HTML Help Kit Version 1.10 http://helpware.net/delphi/

Bilder sind entweder selbst mit Microsoft® Paint Version 5.1 gezeichnet, von Microsofts *„Cards.dll“* gezeichnet oder sind Screenshots meines Projektes.

Das LightScribe CD-Cover sowie das DVD-Inlay wurden mit dem Nero Cover Designer 2.10.1.1 Essentials von mir erstellt.

Diese schriftliche Ausarbeitung wurde mit Microsoft® Office Word 2003 Version 11.5604.5606 verfasst.