Das Scripten

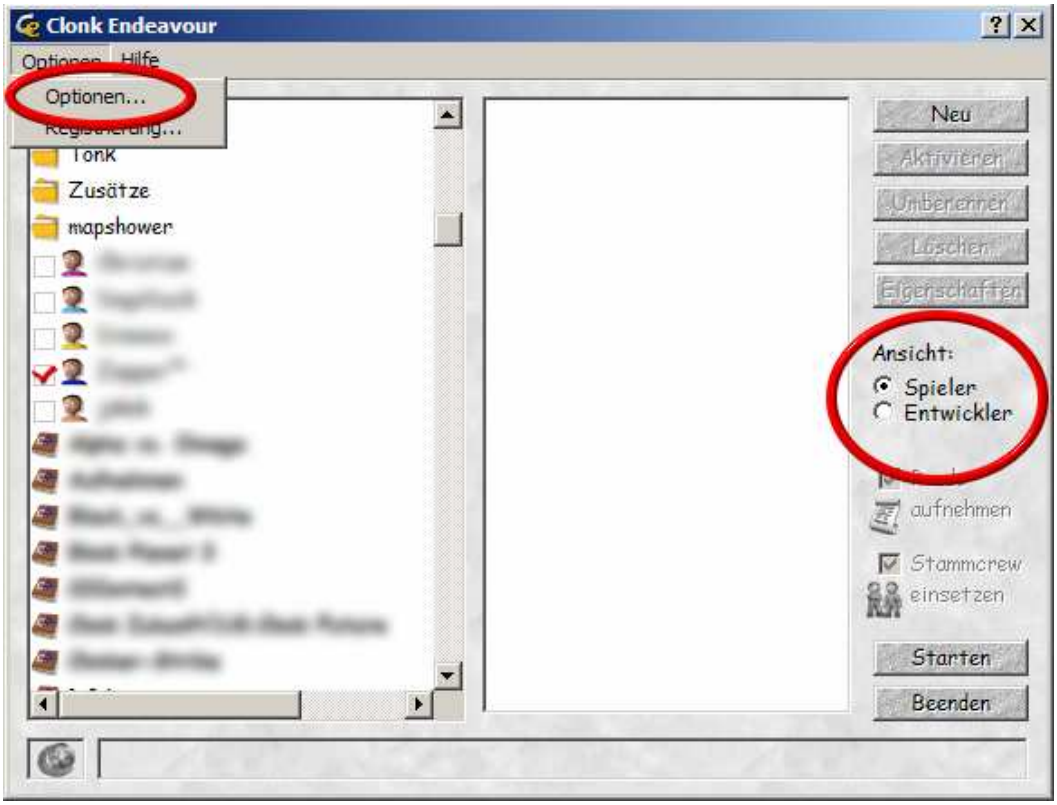
Grundkenntnisse

Ich versuche das Scripten hier so leicht, wie möglich zu erklären. Wer schon Erfahrungen mit höheren Sprachen wie z.B. C++ hat, dem sollte es nicht allzu schwer fallen auch in dieser Sprache (C4Script) bald Ergebnisse zu erzielen, die sich sehen lassen können.
Und die anderen natürlich auch ;)

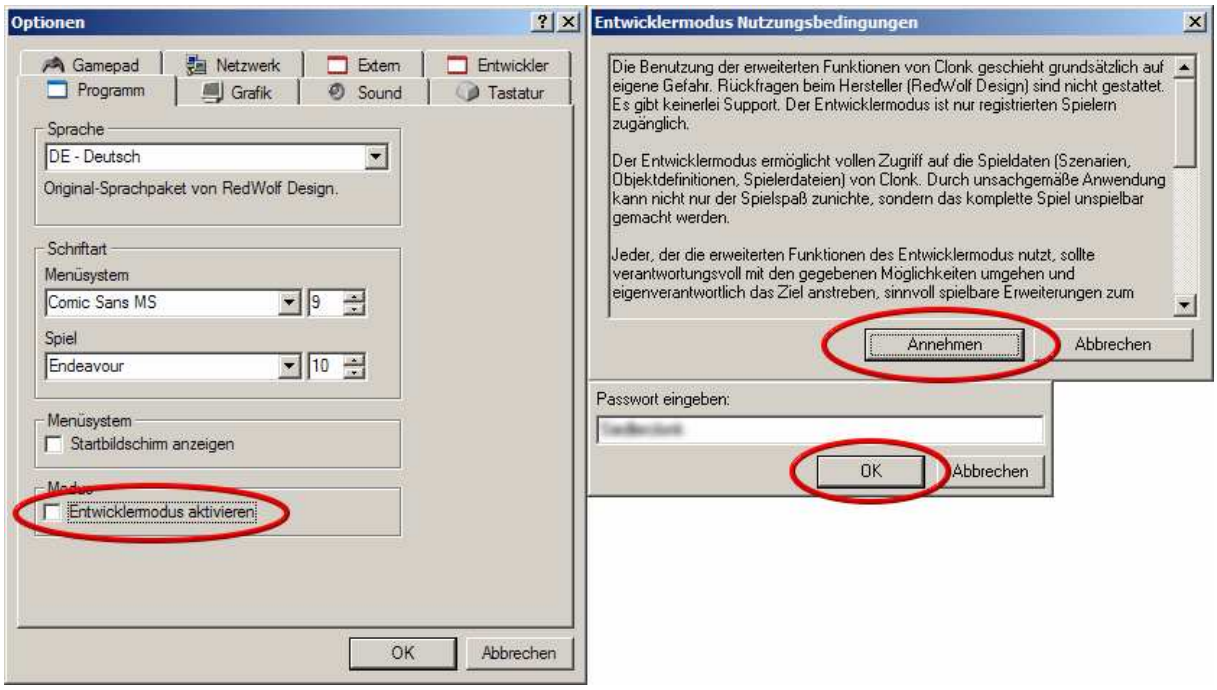
Ein paar Sachen sind im Text kursiv geschrieben. Das heißt, dass es entweder wichtige Tipps oder Warnungen vor Fehlerquellen oder ähnliches sind.

Einstieg:

Hier zeige ich erstmal, wie man in den Entwicklermodus kommt, und wie man Programme für bestimmte Aufgaben definiert:

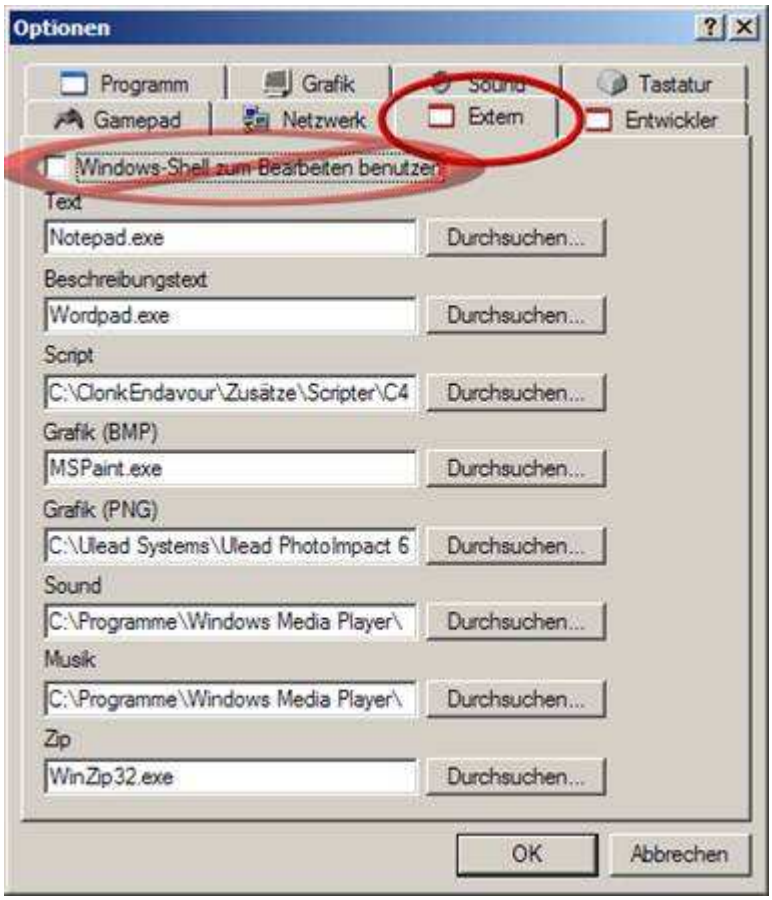


Oben links kann man die Optionen ändern und wenn man den Entwicklermodus aktiviert hat, kann man rechts bei „Ansicht“ zwischen Spieler-
und Entwicklermodus wechseln.



Zum Aktivieren muss man in den Optionen unter „Programm“ ein Häkchen bei „Entwicklermodus aktivieren“ machen.
Dann wird man aufgefordert die Nutzungsbedingungen zu akzeptieren, wenn man dort auf „Annehmen“ klickt muss man noch ein Passwort eingeben.
Das Passwort sage ich hier nicht, aber man findet es in der offiziellen Clonk Endeavour-Doku.

Jetzt muss man nur noch einstellen, mit was man die Dateien bearbeiten will:



In den Optionen klickt man danach auf „Extern" dann muss man „Windows-Shell zum Bearbeiten benutzen" ausstellen.
Dann muss man nur noch den Pfad zu der *.exe Datei angeben.
Ich würde sagen, jeder sollte sich die Programme heraussuchen, mit denen er am liebsten arbeitet. Viele Programme, die man benötigt stellt auch Windows bereits.

So, danach bestätigt man mit OK.

Jetzt kommen wir zu den Grundlagen des C4-Scripts:

Also: In einem Script der neuen Engine muss (am Anfang) immer #strict stehen.

Kommen wir erstmal kurz zu den Kommentaren. Kommentare werden vollständig ignoriert. Man sollte sie z.B. Benutzen, um einen Script, den man auch noch in einigen Jahren verstehen will, näher zu beschreiben.
Es gibt auch im C4-Script Möglichkeiten Kommentare in den Script einzufügen. Wie in C++ gibt es zwei Kommentararten:

```
// Ich bin ein Kommentar. Meine gesamte Zeile wird ignoriert

/*Ich bin auch ein Kommentar. Ich werde von den Zeichen links und rechts begrenzt und kann ohne Probleme über mehrere Zeilen gehen.*/
```

Die erste Kommentarart, bei der „//" an den Zeilenbeginn gestellt wird, wird häufiger zum Kommentieren an sich benutzt.
Die zweite Art, bei der das Kommentar mit „/*" und „*/" begrenzt wird, ist sehr beliebt um in einem Script Teile auszukommentieren. D.h., dass man einfach, wenn man einen bestimmten Teil des Scriptes „abschalten", ihn zu einem Kommentar machen kann.

Ein Script ist immer in Funktionen gegliedert. Funktionen sind Bereiche, die man durch bestimmte Befehle aufrufen kann.
In einem Script können beliebig viele Funktionen aneinander gereiht werden.
Das Schlüsselwort „func" gibt an, dass ein Funktionsname folgt.

Schlüsselwörter dürfen für keinen anderen Zweck, als den Vorgesehenen verwendet werden. So sind Schlüsselwörter als z.B. Variablennamen unzulässig.

```
#strict

protected func HalloFunktion()
{

}
```

„protected" ist eine von 4 Aufrufarten. Sie bestimmen, von wo man die Funktion aufrufen kann.

protected	Nur vom eigenen Script oder von der Engine
public	Von allen Scripten (Standart)
private	Nur vom eigenen Script
global	Von jedem Script als normaler Befehl

Wenn man die Aufruferegeln verletzt, wird man von der Engine mit einer Meldung bestraft.

Diese Funktion hier oben ist noch etwas trist. Wenn sie aufgerufen wird, würde nichts passieren.
Jetzt muss man die Funktion noch mit Befehlen spicken.
Nehmen wir mal den Befehl Message(). Message() gibt eine Nachricht aus, die in Anführungsstrichen als erster Parameter da steht.

In Anführungszeichen wird eigentlich jeder String
(jede Zeichenfolge - z.B.: „Ich heiße Twonky") im Script geschrieben.

Parameter geben einer Funktion extra Informationen. Die verschiedenen Parameter sind durch Kommata getrennt.

```
#strict

protected func HalloFunktion()
{
Message("Hallo");
}
```

Semikolons:
Eine Anweisung muss durch ein Semikolon geschlossen werden.
Als Anweisung gilt z.B. ein Befehl, eine Zuweisung oder eine sonstige Berechnung oder ähnliches.
Das Semikolon gibt an, dass die Anweisung ausgewertet wird. Hinter ihm steht nichts, zur Anweisung gehörendes, mehr.
Man kann also davon ausgehen, dass hinter jeden Befehl ein Semikolon gehört. Damit kann man eigentlich nicht viel falsch machen.

Jetzt würde beim Aufrufen der Funktion HalloFunktion() die Nachricht „Hallo" ausgegeben. So könnte jetzt im selben Script irgendwo „HalloFunktion()“ als Befehl stehen.

Leider wird ‚Hallo’ in der Mitte des Bildschirms ausgegeben.
Jetzt brauchen wir auch noch ein Objekt. Das Objekt, über dem die Nachricht ausgegeben werden soll, ist bei „Message()“ der 2.Parameter.

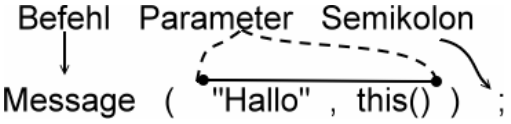
```
#strict

protected func HalloFunktion()
{
Message("Hallo",this());
}
```

this() gibt das Objekt zurück von dem es aufgerufen wird. Das heißt, dass wenn man this() schreibt immer das Objekt „gemeint" ist, in dem der Script steht.

Obwohl im Klammerpaar von this() nichts steht muss es gesetzt sein!

Wenn 2 Befehle verschachtelt sind muss man nur hinter dem Letzten ein Semikolon machen.



So kann man in einer Funktion beliebig viele Befehle aneinander reihen.

Es ist egal ob der Script untereinander steht oder nicht. Man könnte es auch so machen:
#strict protected func HalloFunktion() { Message("Hallo",this()); }
Aber dann würde es kaum mehr lesbar sein.

Aber wie ruft man Funktionen auf?

Es gibt Funktionen, die automatisch aufgerufen werden, wenn etwas Bestimmtes mit dem Objekt passiert.
Z.B. Wird Initialize() automatisch aufgerufen, wenn das Objekt „erschaffen" wird.

Um eigene Funktionen wie unser HalloFunktion() aufzurufen gibt eine andere Art.

Man kann im eigenen Script eine Funktion einfach wie einen anderen Befehl aufrufen:

```
#strict

protected func Initialize()
{
HalloFunktion();
}

protected func HalloFunktion()
{
Message("Hallo",this());
}
```

Jetzt wird, wenn das Objekt erzeugt wird Initialize() aufgerufen. Initialize() ruft dann HalloFunktion() auf.

In einem Script gibt es keine Zeitverzögerung. Das heißt, dass alle Befehle, die in einer Funktion stehen, zwar nacheinander aufgerufen werden, es aber im Spiel keine Verzögerung zwischen den einzelnen Befehlen gibt.

Will man eine Funktion eines anderen Objektes aufrufen kann man dies mit den Befehlen „PrivateCall()", „Call()" oder „ProtectedCall()" machen. Auch zulässig ist die Schreibweise obj->Funktion(), welche oftmals viel einfacher ist.

Da haben wir also schon einen richtigen kleinen Script.

Kommen wir zu den extrem wichtigen Variablen:

Jetzt können wir schon einfache Scripts erstellen.
Doch wie speichert man Daten. Z.B. Den Namen eines Clonks oder eine Zahl, wie oft ein Objekt schon aktiviert wurde?

Mit Variablen. Eigentlich ganz einfach.

Es gibt verschiedene Variablentypen.
Erstmal kommen die, in den meisten Fällen praktischeren benannten Variablen. Sie verbessern z.B. die Übersichtlichkeit eines Scriptes gewaltig.

Name	Eigenschaft	Beispiel
Funktionslokale Variablen	Sind nur für eine Funktion gespeichert	var timer; var objekt; var clonk;
Objektlokale Variablen	Sind für das ganze Objekt unbegrenzt gespeichert	local aktiviert; local ziel; local timer;
Globale Variablen	Sind für einen Script einer Landkarte gespeichert	static meteor; static spieler; static relaunchs;

Erstmal die Funktionslokalen Variablen:

Hit() wird aufgerufen, wenn ein Objekt mit der Landschaft kollidiert.

FindObject() gibt ein Objekt zurück, das nach den Kriterien (Parameter) gefunden wird. Der erste Parameter bei FindObject() ist die ID also eine 4-Zeichen-Folge. Jedes Objekt hat eine eigene ID. Wer nähere Erklärungen zu IDs benötigt, der sollte sich z.B. an die Originaldoku wenden.

RemoveObject() entfernt ein Objekt. Nämlich das, das an der Stelle des ersten Parameters angegeben ist.

Das Gleichheitszeichen speichert in einer beliebigen Variable einen beliebigen Wert oder, wie hier, ein Objekt.

```
#strict

protected func Hit()
{
ClonksWeg();
}

protected func ClonksWeg()
{
var clonk;
clonk = FindObject(CLNK);
RemoveObject(clonk);

}
```

Hier wird in „clonk" ein gefundener Clonk gespeichert. (CLNK ist die ID für einen normalen Clonk).
"clonk" ist eine Funktionslokale Variable, wie man es an dem Schlüsselwort „var" erkennen kann.
Funktionslokale Variablen müssen direkt in der Funktion, in der sie benötigt werden, deklariert werden.

Wenn kein Objekt, welches den Suchkriterien von FindObject() entspricht gefunden wird, wird 0 zurückgegeben.
Also würde dann statt, dass ein Clonk in „clonk" gespeichert wird, 0 in „clonk" gespeichert werden.

Danach wird „clonk" - also der Clonk (wenn es einen gab) - bei RemoveObject() einfach gelöscht.

Wenn als Parameter 0 steht entspricht das meist this(). Also kann man statt this() in den meisten Fällen auch einfach 0 schreiben. Und da man 0 auch weglassen kann, schreibt man meist gar nichts hin:
RemoveObject(this()) == RemoveObject(0) == RemoveObject()

Dieses klappt bei den meisten Funktionen, man sollte aber mit this() auf Nummer sicher gehen.

Die Funktionslokale Variable „clonk" wird aber, wenn die Funktion beendet ist einfach gelöscht. Was, wenn man die Daten behalten will?

Dafür gibt es die Objektlokale Variable:
Objektlokale Variablen kann man im gesamten Script abfragen
man muss sie außerhalb einer Funktion deklarieren.

```
#strict

local clonk;

protected func Initialize()
{
clonk = FindObject(CLNK);
}

protected func Hit()
{
RemoveObject(clonk);
}
```

Hier wird gleich, wenn das Objekt erschaffen wird, in „clonk" ein Clonk gespeichert. Diesmal ist „clonk" eine Objektlokale Variable. Dies kann man an dem Schlüsselwort „local" erkennen, das dem Variablennamen vorangestellt wird.
Bei Hit() - Also wenn das Objekt mit der Landschaft kollidiert- wird dieser Clonk gelöscht.

Globale Variablen sind Variablen, die im gesamten Szenario gültig sind. Sie können in jedem beliebigen Objekt oder im Szenarienscript selbst erzeugt werden.
Das wirft natürlich auch einige Probleme auf, da alle Objekte, die auf diese Variable zugreifen, dieselbe Variable ändern. So könnte es zu Problemen bei Objekten kommen, die Informationen in einer globalen Variable speichern. Man sollte aufpassen, dass diese sich nicht in die Quere kommen. Deshalb sollte man sich darauf beschränken, globale Variablen nur im Szenarienscript zu verwenden. So beugt man Problemen schon mal vor.

Globale Variablen werden von lokalen überschrieben.
Also, wenn es eine globale Variable namens „zahl" gibt und ein anderes Objekt eine lokale Variable, die auch „zahl" heißt besitzt, kommt es nicht zu einem Fehler.
Wenn das Objekt seine lokale Variable ändert, wird die globale dadurch nicht beeinflusst.

```
#strict

static der_eine_clonk;

protected func Initialize()
{
der_eine_clonk = FindObject(CLNK);
}
```

Jetzt würde jedes Objekt mit „der_eine_clonk" auf denselben Clonk zugreifen können.
Globale Variablen werden mit dem Schlüsselwort „static" gekenneichnet.

Kommen wir jetzt noch mal zu den **unbenannten Variablen**.
Diese Variablen benötigt man vor allem für dynamische Scripte.
Was wäre z.B., wenn man die Relaunchzahl für die Spieler speichern will?
Man könnte dies so angehen, dass man für jeden möglichen Spieler eine globale Variable anlegt.

```
static spieler1; static spieler 2; static spieler3;….
```

Hier kann man jedoch schnell erkennen, dass dieses sehr mühsam wird.
Vor allem, wenn man das Szenario für bis zu unendlich Spieler spielbar machen will.

Dann benötigt man die unbenannten dynamischen Variablen.
Diese können im Grunde wie benannte eingesetzt werden. Es gibt nur einen kleinen Unterschied:
Diese Variablen sehen aus wie ein Befehl und benötigen keinen Namen, sondern nur eine Zahl.
Hier sehen wir mal das Beispiel mit dem Clonk-finden-und-entfernen mit unbenannten Variablen:

```
#strict

protected func Initialize()
{
Local(0) = FindObject(CLNK);
}
```

```
protected func Hit()
{
RemoveObject(Local(0));
}
```

Statt Local(0) könnte man auch Local(1), Local(2) usw schreiben.
Jede Zahl greift auf eine andere Variable zu.
Jetzt ist es auch nicht schwer, sich das Beispiel mit den Relaunchs vorzustellen.
Die Nummer der Variable ist dann einfach die Spielernummer.
Aber diese Art von Variablen benötigt man eher selten.

Unbenannte Variablen:

Name	Eigenschaft	Beispiel
Funktionslokale Variablen	Sind nur für eine Funktion gespeichert	Var(0); Var(1); Var(2)
Objektlokale Variablen	Sind für das ganze Objekt unbegrenzt gespeichert	Local(0); Local(1); Local(2);
Globale Variablen	Sind für einen Script einer Landkarte gespeichert	Global(0); Global(1); Global(2);

Jetzt kommen die Bedingungen und Schleifen

Was man eigentlich fast überall braucht sind Bedingungen.
Man will z.B. fragen, ob in der Nähe ein Objekt ist, ob eine Variable 0 enthält oder anderes dergleichen.

Die einzige Bedingungsabfrage ist im C4-Script „if()".
"if()" testet, ob eine Aussage ‚true' ist. Eine Aussage ist ‚true', wenn sie ungleich 0 ist. Also sind alle Zahlen, alle Zeichen und alle Objekte ‚true'.

Nehmen wir mal unser Beispiel mit den localen Variablen. Es wurde immer etwas entfernt, wenn kein Clonk da war, dann wurde eben das Objekt selbst entfernt. Wir erinnern uns: 0 entspräche hier „this()". Und, wenn der Clonk nicht vorhanden wäre, dann würde in „clonk" 0 gespeichert werden. Man kann nun testen, ob die Variable ‚true' also nicht 0 enthält.

```
#strict

local clonk;

protected func Initialize()
{
clonk = FindObject(CLNK);
}

protected func Hit()
{
if(clonk)RemoveObject(clonk);
}
```

Das "if()" hat hier den Effekt, dass getestet wird, ob „clonk" etwas anderes als 0 enthält.
Also wird der Clonk nur entfernt, wenn es ihn gibt. Sinnvoll, oder?

Mit „if()" kann man auch überprüfen, was Variablen enthalten.

```
#strict

protected func VariablenCheck()
{
var nummer=3;
if(nummer==3)Message("nummer ist 3");
}
```

Das doppelte Gleichheitszeichen vergleicht zwei Werte miteinander. Es ist etwas völlig anderes als das einfache Gleichheitszeichen und sollte daher nicht mit diesem verwechselt werden.

Hier wird erst in „nummer“ 3 gespeichert und dann wird getestet, ob „nummer“ auch 3 enthält. Man kann davon ausgehen, dass „nummer“ immer drei enthält, und dass deshalb auch immer „Message()“ aufgerufen wird.
So kann man mit Bedingungen aber immer nur einen Befehl ausführen lassen. Wenn man mehrere Befehle mit einer Bedingung koppeln will, muss man die geschweiften Klammern zur Hilfe nehmen. Diese machen es möglich Befehle in einen Block einzuschließen.

```
#strict

protected func VariablenCheck()
{
var nummer=3;
if(nummer==3)

	{//Diese Klammer beginnt einen Block

	Message(“nummer ist 3”);
	Message(“nummer ist nicht 5”, this());
	//Hier können noch beliebig viele andere Befehle aneinandergereiht
	//werden.

	}//Diese Klammer beendet den Block

}//Diese Klammer beendet die Funktion
```

Hier wird also, wenn „if()“ ‚true’ ist, gleich mehrere Befehle ausgeführt.

Es gibt auch noch viele andere Vergleichsoperatoren.
Zum Beispiel kann man auch auf ungleich prüfen.

```
#strict

protected func VariablenCheck()
{
var nummer=2;
if(nummer!=3)Message(“nummer ist nicht 3”);
}
```

Diesmal gibt die Bedingung nur ‘true’ zurück, wenn “nummer” nicht 3 ist.

Was soll man aber machen, wenn man etwas machen will, wenn die Funktion ‚true’ ist und etwas anderes, wenn die Funktion nicht ‚true’ ist.
Man könnte das natürlich so machen:

```
#strict

protected func VariablenCheck()
{
var nummer=3;
if(nummer == 3)Message(“nummer ist 3”);

if(nummer != 3) Message(“nummer ist doch nicht 3”);
}
```

Das könnte man natürlich wie oben machen, aber es gibt eine viel einfachere Möglichkeit. Nämlich mit „else“.
Mit „else“ kann man angeben was passiert, wenn eine Bedingung nicht zutrifft.
Das Beispiel wie oben sähe dann so aus:

```
#strict

protected func VariablenCheck()
{
var nummer=3;
if(nummer == 3)Message(“nummer ist 3”);
else Message(“nummer ist doch nicht 3”);
}
```

Wenn hier die Bedingung nicht zutrifft, also wenn „nummer“ nicht 3 ist. Wird die Bedingung hinter „else“ ausgeführt.

Jetzt kommen wir zu Schleifen.
Mit „if()“ kann man Wenn-dann-Abfragen machen aber was, wenn du Solange-wie-…-mache-das: haben willst?
Also einfach einige Sachen mehrmals ausführen willst?

Dazu gibt es zwei Möglichkeiten. Nehmen wir erstmal die while-Schleife. Die while-Schleife benimmt sich eigentlich so wie „if()“ außer, dass solange wie die Bedingung zutrifft die Befehle ausgeführt werden.

Man sollte sich vor Endlosschleifen hüten. Also sollte man dafür sorgen, dass die Bedingung irgendwann nicht mehr ‚true‘ ist.

So kann man z.B. alle Clonks entfernen.

```
#strict

protected func RemoveAllClonks()
{
var clonk;
while(clonk=FindObject(CLNK))RemoveObject(clonk);
}
```

Solange, wie in „clonk" ein Clonk gespeichert werden kann –also solange es einen gibt-, wird dieser Clonk entfernt. Hier ergibt die Schleife irgendwann nicht mehr ‚true', weil einfach kein Clonk mehr gefunden werden kann. Sie werden ja schließlich alle entfernt.
Schleifen sind tolle Sachen, die man eigentlich genauso oft braucht, wie das normale „if()".
Durch Schleifen kann man auch z.B. einen Befehl mehrmals ausführen.

```
#strict

protected func Make10Gold()
{
var goldzahl=10;
while(goldzahl != 0)
	{
	CreateObject(GOLD);
	goldzahl--;
	}
}
```

CreateObject() erzeugt ein Objekt von der angegebenen ID. Für mehr Informationen zu den Befehlen sollte man die Oridinaldoku zur Hilfe nehmen.

„--„ verringert eine Variable um eins.
"++" erhöht sie dagegen um eins.

In „goldzahl" wird die Zahl 10 gespeichert.
Es wird dann solange ein Gold erzeugt und „goldzahl" um eins verringert bis „goldzahl" nicht mehr ungleich 0 ist.

Kommen wir zu der zweiten Art von Schleife.
Der for-Schleife. Diese hat eine Besonderheit - sie ist in drei Teile aufgeteilt:

for(Vor der Schleife ; Bedingung ; Jedes Mal am Ende)

Die Befehle vor dem ersten Semikolon werden bevor die Schleife beginnt ausgeführt. So kann man z.B. Variablen auf bestimmte Werte setzen.
Vor dem zweiten Semikolon steht die Bedingung, die abgefragt wird und sich genauso wie die in der while-Schleife verhält.
nach dem zweiten Semikolon können Befehle stehen, die jedes Mal am Ende der Schleife durchgeführt werden. So kann man z.B. Variablen bei jedem Schleifendurchlauf verringern oder erhöhen.
Hier ist noch mal das Beispiel von der while-Schleife mit „for".

```
#strict

protected func Make10Gold()
{
for(var goldzahl=10;goldzahl != 0;goldzahl--)CreateObject(GOLD);
}
```

Mit einer for-Schleife kann man sehr schön alle Befehle in die Schleife einbinden. Eine for-Schleife ist einfach bequemer als eine while-Schleife. Aber man muss darauf achten keines der Semikolons zu vergessen. Selbst, wenn Stellen leer gelassen werden, müssen die Semikolons vorhanden sein.

```
#strict

protected func Make10Gold()
{
var goldzahl=10;
for(;goldzahl != 0;)
	{
	CreateObject(GOLD);
	goldzahl--;
	}
}
```

Wenn man ein Semikolon weglässt gibt es eine Fehlermeldung und die Schleife funktioniert nicht.

Jetzt weiß man eigentlich schon das wichtigste.

Kommen wir zu den Timern

Als ich schon einigermaßen scripten konnte, wusste ich leider noch nicht, wie man Timer sinnvoll erzeugt. Das stand in keiner Doku. Deshalb versuche ich es hier zu erklären.

Es gibt wieder verschiedene Möglichkeiten für Timer:

Timer	Nutzen
TimerCall	Regelmäßige Aufrufe bei Objekten (zB. für eine Mine)
Aktion	Nützlich für einmalige Timer (zB. für einen Zeitzünder)
Szenarienscripttimer	Aufrufe in Szenarienscripts (zB. für eine Kontrolle der Punktzahl)
ScheduleCall()	Für einmalige oder (kurzweilige) mehrfache Timer

Erstmal TimerCall:

Der TimerCall wird in der DefCore gesetzt. Z.B. könnte es dann in der DefCore so aussehen:

```
[DefCore]
id=TEST
Name=Beispiel
Category=1
TimerCall=Check
Timer=10
```

Das ist jetzt zwar insgesamt ein sehr spartanisches Beispiel einer DefCore aber für uns reicht es.
TimerCall heißt, dass die Funktion, die nach dem Gleichheitszeichen steht aufgerufen wird. Also wird hier Check aufgerufen.
Timer gibt an, wie oft der TimerCall aufgerufen wird. Hier also alle 10 Frames.

DefCore	Script

[DefCore]
id=TEST
Name=Beispiel
Category=1
TimerCall=Check
Timer=10
	#strict
	
	protected func Check()
	{
	Message("Schon wieder 10 Frames um",this());
	}

Hier wird alle 10 Frames die Funktion Check aufgerufen.
Dies kann man gut für z.B. eine Mine nutzen, die alle paar Frames überprüfen soll, ob sich ein potenzieller Auslöser nähert.

Eine Aktion als Timer:

Man kann auch eine normale Aktion als Timer verwenden. Man muss die Aktion nur irgendwo starten.

Z.B. so:

ActMap	Script

[Action]
Name=Timer
Procedure=NONE
FacetBase=1
Length=1
Delay=50
EndCall=TimeOver
	#strict
	public func Activate()
	{
	SetAction("Timer");
	}
	
	protected func TimeOver()
	{
	Explode(20);
	}

Activate() wird zB. aufgerufen, wenn ein Clonk dieses Objekt in der Hand hat und „DoppelGraben" drückt.

Wenn eine Aktion zu Ende ist wird die Funktion, die hinter EndCall steht, aufgerufen.

Wenn Activate() aufgerufen wird, wird die Aktion „Timer" gestartet und wenn diese beendet ist (genau nach 50 Frames - siehe Delay), wird TimeOver aufgerufen.
Wobei es hierbei natürlich völlig egal ist, wie man die Aktion oder den EndCall nennt.

Durch die Aktionstimer kann man schöne einmalige Timer kreieren.
Ein gutes Beispiel dafür findet man beim T-Flint.

Szenarienscript-Timer:

Wenn man will, dass z.B. in einem Szenario überprüft wird, ob schon jemand gewonnen hat oder eine bestimmte Geldmenge erreicht wurde, geht am besten diese Timerart.

```
#strict

protected func Initialize()
{
//Startet den Scriptzähler. Nicht vergessen, das ist ein Kommentar
ScriptGo(1);
}

protected func Script10()
{
var pclonk;
pclonk=FindObject(CLNK);
RemoveObject(pclonk);

//Setzt den Scriptzähler wieder auf 5. Das ist auch ein Kommentar
goto(5);
}
```

ScriptGo(1) startet einen Zähler im Szenarioscript.
goto setzt den Scriptzähler wieder auf eine bestimmte Zahl.

Hier wird immer, wenn der Zähler auf 10 ist, die Funktion Script10() aufgerufen (das geht natürlich auch mit Script15(), Script16(), Script100, usw...).

Der interne Zähler wird alle 10 Frames um eins erhöht. Also wird die Funktion „Script10()" nach genau 100 Frames aufgerufen.

Script10() entfernt einen gefundenen Clonk und setzt danach den Scriptzähler wieder auf 5 von wo er erneut hoch auf 10 läuft usw...
Wenn man nicht goto(5) macht, dann wird die Funktion nur einmalig aufgerufen.

ScheduleCall()

Der Befehl ScheduleCall() ist neu und arbeitet mit Effekten. Für eine nähere Erklärung zu Effekten muss man wohl in die Originaldoku schauen. Aber insgesamt sind Effekte optimal dafür geeignet Timer zu erzeugen.

Dieser Befehl ruft eine Funktion einmal, mehrfach und/oder mit Verzögerung auf.
Das ist dasselbe Beispiel wie bei den Aktionstimern. Diesmal mit ScheduleCall():

```
public func Activate()
{
ScheduleCall(this(),"TimeOver",50,0);
}

protected func TimeOver()
{
Explode(20);
}
```

Hier wird mit einer Verzögerung von 50 die Funktion TimeOver() aufgerufen.

Ich erklär mal die Parameter von ScheduleCall. Angegeben ist das so:
object pObj, string strFunction, int iInterval, int iRepeat, par0, par1, par2, par3, par4

D.h.: der erste Parameter (also object pObj) gibt an bei welchem Objekt string strFunction aufgerufen wird.
string strFunction gibt an, welche Funktion aufgerufen wird.
int iInterval gibt an mit welcher Verzögerung string strFunction aufgerufen wird.
int Repeat gibt an wie oft das Aufrufen wiederholt werden soll.
Die restlichen Parameter werden an string strFunction übergeben.

Kommen wir abschließend noch mal zu Fehlermeldungen:

Es erscheinen am Anfang der Runde manchmal komische Fehlermeldungen. Hier erklär' ich ein paar (die häufigsten):

Meldung	Fehler	Beispiel
::Hit[22;2] unexpected token	Kein Semikolon?	SetAction("Wait")
::Activate[27;18] ';' must be on top-level, forgot ')'?	) oder ; oder so vergessen?	SetAction("Wait";
func "SetAction" par 1: "id" cannot be converted to "object"	ein falscher Typ (par 1: gibt an, wo sich der Fehler befindet- hier beim 1.Parameter (man fängt bei 0 an zu zählen))	SetAction("Wait",CLNK);
'func': expecting opening block ('{') after func declaration	Nach einer Funktion fehlt die geschweifte Klammer	protected func Activate() }

too many blocks closed ('}')	Es gibt ein ‘{‘ ohne zugehöriges‘}’	while(*bedingung*)
		mach was;
		}//gehört zu keinem ‚{‚

::unknown identifier: pClonk	‚pClonk’ wurde vorher nicht als Variable definiert	var clonk; pClonk=1;

Es gibt noch andere Möglichkeiten, was an einem Objekt nicht funktioniert, wenn es nicht oben aufgeführt ist (und es keiner der Fehler ist, die ich nicht beschrieben habe).

Problem	Mögliche Ursache	Behebung
Objekt erscheint nie im Spiel	Leere ActMap.txt ?	ActMap füllen oder löschen
	Falsche ID? Oder Überlagerung der ID mit andrem Objekt	ID ändern (**muss** 4 Zeichen haben)
	Overlay stimmt von der Größe her nicht mit Graphics überein	Overlay anpassen
Objekt fällt durch die Landschaft	Vertices falsch oder nicht vorhanden	Vertices anpassen (siehe Originaldoku)
Objekt sieht man nur halb	In der DefCore ist Width oder Height falsch gesetzt	Korrigieren

Aktion wird nicht vollständig abgespielt	Teile falsch geschrieben? ZB: Length, Delay	Richtig schreiben
Objekt schwebt in der Luft, obwohl es keine solche Aktion hat	Ist die Category (in der DefCore) auf 1 ?	Category ändern (für Gegenstände auf 16) - siehe Originaldoku
Man kennt das Passwort für den Entwicklermod. nicht.	Also wirklich.....(Originaldoku nicht gründlich gelesen?)	Gründlich lesen(ist ganz vorne in der Doku)
Das Objekt dupliziert sich beim bearbeiten und es entstehen im Arbeitsverzeichnis Duplikate mit der Endung: *.000 ; *.001 ; *.002 ; usw.	Läuft die Engine beim bearbeiten?	Man sollte Objekte vor dem bearbeiten mit Rechtsklick „zerlegen“ und vor dem hochladen wieder „packen“
Die Engine stürzt völlig ohne ersichtlichen Grund und ohne Fehlermeldung ab.	Ruft eine Funktion sich immer selber auf? func Check(){Check();}	Man sollte so etwas vermeiden
	Ist zB. in Message ein Platzhalter falsch? Message(„%d“,0,10);	Die Platzhalter müssen vom Typ her stimmen

Ich hoffe, dass ihr in dieser kleinen Grunddokumentation etwas gelernt habt und nun eigene Objekte entwickeln könnt.
Wer jetzt nicht wie, wie er weitermachen soll, der kann sich ja einfach Scripte anderer Objekte vornehmen und versuchen, ihre Funktionsweise zu verstehen.
Natürlich sollte man auch spätestens, wenn man bestimmte Befehle sucht einen Blick in die Originaldoku werfen.
Und mit ein bisschen Geschick ist alles möglich. Man sollte sich wenn man Fragen hat entweder im Clonk Forum auf clonk.de oder in einem der anderen Foren umschauen. Oftmals wurden Probleme/Fragen bereits gelöst.

Wer noch weitere Anregungen oder Beschwerden hat, kann mir ja Bescheid geben.
Diese Dokumentation wurde von David Dormagen aka Zapper geschrieben.