11.5.0.0 Fehlerbehandlung 1

Wenn Sie Software entwickeln, so ist korrekte Software sicher das erklärte Ziel, um damit eine bestimmte Aufgabe oder eins Klasse von Aufgaben zu jeder Zeit fehlerfrei zu lösen. Sie kennen sicher aus eigener Erfahrung, dass das ein beschwerlicher Weg sein kann. In unterschiedlichen Phasen der Softwareentwicklung werden Sie es immer wieder mit Fehlern zu tun haben.

Um diesen Fehlern auf die Spur zu kommen hilft es, die möglichen Fehlerklassen zu kennen, um so den Fehlern auf die Spur zu kommen. Wer dabei nur an Fehler im Quelltext denkt, dem helfen sicher die folgende Übersicht und die Beiträge zum Fehler-Management in diesem und in den folgenden Kapiteln.

Die Instruktionen TRY, CATCH und FINALLY werden im Kapitel 11.5.1_Try_Finally_Catch beschrieben; ebenso im Kapitel 11.5.1.4 das Ereignis Application_Error(). Auf mögliche Fehlerquellen beim Programmieren wird im Kapitel 11.5.2 hinzuweisen. Diese detaillierten Hinweise sollen Ihnen dabei helfen, die den Programmen zugrunde liegenden Algorithmen fehlerfrei in Gambas-Quelltext umzusetzen. Im Kapitel 11.5.3 finden Sie grundlegende Informationen zum Einsatz des GNU Debuggers und des Programms Valgrind, damit Sie Laufzeit-Fehlern auf die Spur kommen.

11.5.0.0.1 Software-Fehler

In Anlehnung an https://de.wikipedia.org/wiki/Programmfehler#Arten_von_Programmfehlern wird hier nur eine kurze Übersicht zu Software-Fehlern eingefügt und ansonsten auf die angegebene Quelle verwiesen.

- Lexikalische Fehler sind nicht interpretierbare Zeichenketten wie sie bei simplen Schreibfehlern oder der Nichtbeachtung der Groß- und Kleinschreibung ($DBConnection.Type = "sqlite3" ' The database server type must be in lower case!) entstehen.
- Syntaxfehler sind Verstöße gegen die grammatischen Regeln der benutzten Programmiersprache.
  Nutzen Sie die IDE von Gambas oder das Program gbc3, so werden lexikalische Fehler und Syntaxfehler frühzeitig erkannt, weil bei diesen Fehlern eine Kompilierung des fehlerhaften Quelltextes verhindert wird und detaillierte Fehlermeldungen ausgegeben werden.

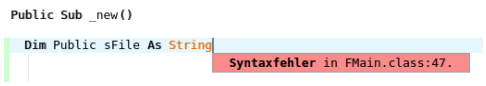

Abbildung 11.5.0.0.1: Syntax-Fehler (IDE)

- Semantische Fehler sind Fehler, in denen eine programmierte Anweisung zwar syntaktisch fehlerfrei, aber inhaltlich trotzdem fehlerhaft ist.
- Logische Fehler bestehen in einem im Detail falschen Lösungsansatz durch fehlerhafte Algorithmen.

Von diesen Fehlern sind die so genannten Laufzeitfehler zu unterscheiden: Während die o.a. Fehler ein fehlerhaftes Programm bedeuten, das entweder nicht ausführbar ist oder fehlerhafte Ergebnisse liefert, kann auch ein `korrektes` Programm bei seiner Ausführung zu Fehlern führen. Laufzeitfehler sind alle Arten von Fehlern, wie falsche oder fehlerhafte Laufzeitumgebung, falsche Eingabedaten (Typ, Wert) oder fehlerhafte Bibliotheken, die auftreten, während das korrekte Programm abgearbeitet wird. Laufzeitfehler können sich zum Beispiel in unerwünschtem Verhalten zeigen, einen Programm-Absturz hervorrufen oder das Programm als nicht mehr bedienbar "einfrieren" lassen.

Die nächsten beiden Fehler-Kategorien sind nur im erweiterten Sinne Fehler:

- Designfehler sind Fehler im Grundkonzept, auf dessen Grundlage das Programm entwickelt wird. Ein typischer Designfehler ist die Codewiederholung, die zwar nicht unmittelbar zu Programmfehlern führt, aber bei der Softwarewartung, der Modifikation oder der Erweiterung von Programm-Quelltext sehr leicht übersehen werden kann und dann zu unerwünschten Effekten führt.
- Fehler im Bedienkonzept. Das Programm verhält sich anders, als es einzelne oder viele Anwender erwarten, obwohl es fehlerfrei arbeitet. Bezogen auf grafische Oberflächen kann hier auf die Grundsätze der Dialoggestaltung nach ISO 9241-110 verwiesen werden.

11.5.0.0.2 Paradigmen der Fehlerbehandlung

Was Behandlung von Fehlern angeht, lassen sich die meisten Programmiersprachen in zwei Lager unterteilen. Einerseits hat man die C-artige Fehlerbehandlung, bei der eine Funktion einen Rückgabewert hat, der vom Aufrufer untersucht werden muss und dessen Wert den Erfolg vom Fehler der Operation unterscheidet. Möglicherweise enthält der Rückgabewert auch nähere Informationen über den Erfolg zum Beispiel wie viele Dateien gelöscht wurden oder den Fehler, der beim Löschen auftrat. Funktionen haben hier keinen Rückgabewert oder einen Rückgabewert nur um den Erfolg der Operation näher zu beschreiben. Tritt ein Fehler bei der Operation auf, so wird eine Ausnahme – eine Exception – ausgelöst. Das äußert sich in einem Laufzeitfehler. Der Programmierer muss diesen Laufzeitfehler abfangen – in Gambas mit Try oder Catch – ihn behandeln und damit beseitigen.

Moderne Sprachen neigen wohl zum Ausnahmen-Paradigma. Der Vorteil von Ausnahmen besteht darin, dass der Rückgabewert einer Funktion nicht in zwei Konzepte (Erfolg und Fehler) unterteilt werden muss. Oft findet man im ersten Lager die Konvention, dass eine Funktion einen Integer-Wert zurück gibt, bei dem nicht-negative Werte für Erfolg und negative für einen Fehler stehen. Gleichzeitig ist der numerische Rückgabewert im Fehlerfall ein Index auf ein Array von Fehlermeldungen, beschreibt also den Fehler genau. Was will man aber tun, wenn man eine Funktion hat, die im Erfolgsfall auch bestimmte negative Werte zurückgeben kann?

Ein weiterer Vorteil von Ausnahmen besteht darin, dass man den Programmierer zwingt, sie zu behandeln. Während man den Rückgabewert einer Funktion absichtlich oder unabsichtlich ignorieren kann, lässt sich ein Laufzeitfehler nicht ignorieren. Man müsste dazu explizit ein Try vor die Anweisung schreiben - es kann also nicht unabsichtlich vergessen werden, den Fehlerfall zu behandeln. Der Gewinn besteht darin, dass ein Programm, das in einer ausnahmen-basierten Fehler-Behandlung läuft, durch *unerwartete* Fehler immer beendet wird und nicht unter der falschen Annahme weiter läuft, dass kein Fehler passiert ist, wie es beim Ignorieren von fehleranzeigenden Rückgabewerten passieren würde.

Gambas unterstützt beide Paradigmen, verwendet aber häufiger Ausnahmen, die man in Gambas mit Error.Raise(...) auslöst. Wenn Sie zum Beispiel eine Datei mit der Anweisung KILL filepath nicht löschen können, dann wird Gambas einen Laufzeitfehler auslösen, den Sie mit Try oder Catch abfangen müssen – sonst geht es für das Programm nicht weiter.

Allerdings ist Gambas nicht konsequent beim Ausnahmen-Paradigma. Error.Raise("string") können Sie nur mit einem einzigen String als individuelle Fehlermeldung aufrufen. Wenn Sie genau wissen möchte, was im Fehlerfall passiert ist, so müssen Sie diese Informationen aus dem String herausparsen. Was passiert jedoch, wenn der Entwickler Inhalt und Format der Fehlermeldung ändert oder wenn das Projekt und damit auch die Fehlermeldung übersetzt wird? In anderen Sprachen wie Python kann man Ausnahme-Klassen erzeugen. Statt der Methode Error.Raise("string") kann man ein ganzes Objekt mit spezifischen Informationen (in nativen Daten-Typen der Sprache) füllen und dieses Objekt in Error.Raise(oObject) einfügen. Der Programmierer fängt dieses Objekt ab und hat die komplexen Daten des Fehlers sauber verpackt in nativen Daten-Typen.
In Gambas können Sie Gleiches erreichen – nur etwas umständlicher. Angenommen Sie verwenden die Klasse Memcached. Spendieren Sie dieser Klasse eine Eigenschaft mit dem Namen "LastError". Diese Eigenschaft kann vom Typ eine beliebige Klasse sein. Erzeugen Sie deshalb eine Klasse _Memcached_Error. Die Unterstriche sind Konvention in Gambas: Der führende Unterstrich versteckt die Klasse und der zweite Unterstrich zwischen "Memcached" und "Error" zeigt die Zugehörigkeit zur Klasse Memcached an. Dort deklarieren Sie Eigenschaften, die nötig sind um Memcached-Fehler detailliert zu beschreiben.

Bevor Sie dann in Memcached einen Fehler mit Error.Raise() auslösen, erzeugen Sie ein neues Objekt vom Typ _Memcached_Error, füllen es mit Daten über den aktuell aufgetretenen Fehler und weisen es der LastError-Eigenschaft zu. So könnte eine Erweiterung der Komponente aussehen:

```
' Memcached.class aus gb.memcached
  Property Read LastError As _Memcached_Error

  Private $hLastError As _Memcached_Error

  Private Sub LastError_Read() As _Memcached_Error
    Return $hLastError
  End
```

```
Public Sub Delete(sKey As String)
  ...
  If _HadError(...) Then
     $hLastError = New _Memcached_Error
   ' $hLastError mit Informationen füllen
     Error.Raise(...)
  Endif
  ...
End
```

In einem Projekt setzen Sie dann die neue Eigenschaft ein:

```
Try $hMemcached.Delete(...)
If Error Then
 ' Nutzen Sie $hMemcached.LastError um den Fehler zu behandeln oder
 ' um eine aussagekräftige Fehlermeldung zusammenzustellen.
Endif
```

Dieses Vorgehen reicht wahrscheinlich schon aus. Bedenken Sie: Wenn Sie nur eine einzige _Memcached_Error-Klasse verwenden, müssten Sie alle Fehlereigenschaften in dieser einen Klasse beschreiben. Informationen, die für bestimmte Fehler relevant sind, werden für andere Fehler nicht gebraucht, wodurch die _Memcached_Error-Klasse unnötig groß und allgemein wird. Wenn Sie aber noch einen Schritt weiter gehen wollen mit einer konsequenten, objekt-orientierten Sicht auf Klassen, so können Sie aus _Memcached_Error spezielle Klassen ableiten, die einzelne Fehler beschreiben. Ein Fehler beim Löschen eines Schlüssels hat eine eigene Klasse _Memcached_Error_NotFound genau für den Fall, dass das Löschen unmöglich war, weil der Schlüssel nicht vorhanden ist. Man könnte sich aber auch vorstellen, dass der Schlüssel existiert, aber nicht zum Löschen freigegeben wurde. Das Löschen schlägt deshalb wegen fehlender Rechte fehl. Für diesen Fall wäre eine weitere Klasse _Memcached_Error_Permission zu planen.

Der erhöhte Aufwand bringt zwei Vorteile mit sich. Zum einen werden sich die verfügbaren Informationen je nach Fehler unterscheiden und damit detaillierter sein. Der andere Vorteil dieser vielen Fehlerklassen besteht darin, dass man den IS-Operator nutzen kann, um die Fehlerart festzustellen:

```
Try $hMemcached.Delete(...)
If Error Then
  If $hMemcached.LastError Is _Memcached_Error_NotFound Then
   ' -> Löschen war unmöglich, weil die Datei nicht gefunden wurde
  Else If $hMemcached.LastError Is _Memecahed_Error_Permissions Then
     ' -> Es fehlt die Berechtigung, den Schlüssel zu löschen
  Else
     ...
  Endif
Endif
```

Die Klasse HttpClient von gb.net.curl besitzt zum Beispiel anstelle eines Ausnahme-Objekts die beiden Eigenschaften *ErrorText* und *Status*, die Sie für die Diagnose von Ausnahmen (Laufzeitfehler oder im Error-Ereignis) nutzen können.