Sibelius 2

Using the ManuScript™ language

Deutsche Fassung

Edition 2.1
Verfasst von Jonathan Finn, James Laccombe, Yasir Hassan, Simon Whiteside und Graham Westlake, mit Beiträgen von Andrew Davis und Daniel Sreadbury.
Die deutsche Übersetzung besorgte Adrine Nazaryan mit Korrekturen und Anmerkungen von Thomas Buchholz.

Inhalt

Einführung

ManuScript™, eine einfache und auf Musik basierende Programmiersprache, wurde entwickelt, um Plug-ins für den Sibelius Musikprozessor zu schreiben.

Sie beruht auf Simkin, die eine Schriftsprache ist, die von Simon Whiteseide entwickelt und dann von ihm und von Graham Westlake erweitert wurde (Simkin ist ein Kosename für Simon, der manchmal in den viktorianischen Novellen zu finden ist.). Mehr Information über Simkin und weitere Hilfen für die Sprache und Syntax finden sie auf der Simkin`s Webseite unter **www.simkin.co.uk.**

Begründung

Da die Plug-in Sprache zu Sibelius hinzugefügt wurde, haben wir einige Vorüberlegungen getroffen:

- Die Musiknotation ist vielschichtig und grenzenlos ausgedehnt, so dass einige Benutzer manchmal Hinzufügungen zum Musiknotationsprogramm machen können, um das Musiknotationsprogramm mit diesen Erweiterungen zu bewältigen.
- Es ist nützlich, häufig wiederholende Vorgänge automatisieren zu lassen (z. B eine MIDI- Datei öffnen und als eine Partitur speichern), in dem man das Schrift- oder Makrosystem benutzt.
- Natürlich können noch viele komplexe Techniken, die für die Komposition und Arrangierung notwendig sind, automatisiert werden, aber es gibt noch mehrere Techniken, die als Standartfeatures in Sibelius aufgenommen werden können.

Es gab einige Bedienungen, die wir in Fragen des Sprachgebrauchs festlegen wollten.

- Die Sprache sollte einfach sein, da wir möchten, dass der normaler Benutzer (nicht nur ein erfahrener Programmierer) im Stande ist, sie zu gebrauchen.
- Wir wollten, dass ein Plug-in auf jedem Computer ausführbar ist, da der Gebrauch von PC-s, Mac-s und anderen Plattformen in der Musikwelt sehr verbreitet ist.
- Wir wollten, dass die Hilfsprogramme in der Sprache mit Sibelius zusammen geliefert werden.
- Wir wollten, dass die Musikbegriffe (Tonhöhe, Noten und Takte) sprachlich einfach ausgedrückt werden.
- Wir wollten, dass die Programme im Stande wären, mit Sibelius leicht zu konfrontieren (das Hinzufügen und Zurückholen der Information von den Partituren).
- Wir wollten, dass die einfachen Dialogfenster und andere brauchbare Elemente der Nahtstelle leicht zu programmieren wären.

C/C++ wäre nicht die Standartprogrammiersprache der Welt geworden, wenn sie für den Nichtfachmann nicht leicht zu benutzen wäre, aber diese Sprachen benötigen einen separaten Compiler, und sie müssen für jede verschiedenartige Plattform neu übersetzt werden (und auf diese Weise entstünden vielfältige Versionen für jeden Plug-in).

Die Sprache Java war viel versprechend, da sie verhältnismäßig einfach ist und auf jeder verschiedenen Plattform ohne neue Übersetzung gestartet werden kann. Wie auch immer, wir würden immer noch für den Gebrauch des Benutzers einen Compiler liefern müssen, und wir würden Musikbegriffe in Java nicht so genau ausdrücken können, wie wir das mit einer neuen Sprache tun können.

Deshalb entschieden wir uns dafür, unsere eigene Sprache zu entwickeln, die so interpretiert ist, dass sie auf verschiedenen Plattformen gestartet werden kann. Sie ist in Sibelius eingegliedert, ohne dass man dafür weitere separate Compiler benötigt. Sie kann jederzeit mit neuen Musikbegriffen erweitert werden.

Das Ergebnis ist die leicht zu erlernende ManuScript-Sprache. Die Syntax und viele Begriffe werden für die Programmierer von C/C++ oder Java bekannt sein. Der Aufbau der Sprache arbeitet mit englischen Musikbegriffen (Score = Partitur, Staff = Notenzeile, Bar = Takt, Clef = Schlüssel, NoteRest = Note/Akkord/Pause), die sofort verständlich sind, wenn man ihre Bedeutung kennt.

Muss ich ManuSkript-Sprache benutzen?

Theoretisch gesehen nein, falls sie nicht die ManuScript- Sprache verwenden wollen, können sie Plug-ins mit einer anderen Sprache schreiben, aber es muss vorausgesetzt werden, dass als Endergebnis ein Dynamically Linked Library (DLL) im Sibelius Plug-in-Verzeichnis steht. Zum Beispiel das Sibeliusdatei-Plugin, das Finale-, Allegro-, PrintMusic- und Scoredateien konvertiert, ist auf diese Weise programmiert.
Wie auch immer, um eine DLL-Datei für ein Plug-in zu erstellen, wird ein Software Development Kit (SDK) erforderlich, welches im Moment nicht verfügbar ist. Deshalb können Plug-ins in der Praxis momentan nur mit ManuScript geschrieben werden.

Technische Unterstützung

Da die ManuScript- Sprache mehr zum Aufgabenbereich unserer Programmierer gehört als zu den Aufgaben unseres technischen Unterstützungsteams (in dem kein Programmierer arbeitet), können wir ihnen keine detaillierte technische Hilfe zur Verfügung stellen. Dieses Dokument und die Plug-in-Beispiele sollen lediglich den nötigen Durchblick darüber geben, wie man einfache Programmierungen sehr schnell erstellen kann.
Wir freuen uns über alle an uns geschickten nutzbaren Plug-ins. Ihre neuen Plug-ins können sie an folgende Adresse senden: **helpuk@sibelius.com**, und wir nehmen sie vielleicht in unsere Website auf. Falls wir ihr Plug-in mit Sibelius absetzen, werden wir sie für dafür bezahlen.

Tutorial

Das Erstellen von Plug-ins

Ein einfaches Plug-in erstellen

Fangen wir mit einem einfachen Plug-in an. Es wird angenommen, dass sie einige Grundkenntnisse in Programmierung haben (Z. B. in Basic oder in C), und deshalb sollte ihnen solche Begriffe wie Variable, Loop usw. bekannt sein.

- Sibelius starten.
- Im Menü **Datei > Plug-ins** wählen sie **Plug- ins bearbeiten**, dann klicken sie auf **Neu.**
- Sie werden aufgefordert, den internen Namen ihres Plug-in einzugeben, der Name, der im Menü angezeigt werden soll und den Namen der Kategorie, in der ihr Plug-in im Plug-in-Menü angezeigt werden soll. Zwei einzelne Namen für den Plug-in sind deshalb dafür notwendig, weil die Dateinamen nicht mehr als 31 Buchstaben haben dürfen (auf dem Mac, der Classic Mac OS), hingegen können die Menünamen so lang sein, wie sie es möchten.
- Tippen sie **Test** als interner Name, **Test plug-in** als Menüname und **Tests** als Kategoriename, dann klicken sie auf **Ok**.
- Sie werden sehen, **Test** wird in die Liste von **Edit Plug-ins** hinzugefügt. Klicken sie auf **Ok**.
- Wenn sie wieder in die **Plug-in-Menüs** sehen, werden sie ein **Tests-Untermenü** mit einem **Test-Plug-in** am Ende sehen.
- Klicken sie auf **Test Plug-in** und das Plug-in wird starten. Was passiert? Es öffnet sich ein Fenster auf, das **Test** heißt (Wann immer sie ein neues Plug-in definieren, Sibelius es wird in die erste Reihe des Programms einsetzen).
- Klicken sie auf **Ok** in dem Fenster und das Plug-in wird anhalten.

Drei Arten der Information

Lassen sie uns ansehen, was bis jetzt im neuen Plug-in zu finden ist. Wählen sie wieder **Datei > Plug-ins > Plug- ins bearbeiten,** dann klicken sie im Dialogfenster auf **Test** und auf **Bearbeiten** (stattdessen können sie nur zwei Mal auf **Test** klicken, um es zu bearbeiten). Sie werden ein Fenster mit drei Arten von Informationen finden, die in einem Plug-in vorkommen können.

- **Methoden**: sind ähnlich wie die Vorgehensweisen, Funktionen oder Routinen in anderen Programmiersprachen.
- **Dialoge**: enthält das Layout jedes einzelnen Dialogfensters, das sie für ihr Plug-in entwerfen.
- **Daten**: sind die Variablen, deren Wert zwischendurch festgehalten wird, wenn das Plug-in startet. Sie sind ganz wie die globalen Variablen in den Sprachen wie C, aber meistens sind sie wie die Teilvariablen in C++ und Java oder wie die Eigenschaften in Visual BASIC.

Methoden

Das eigentliche Programm besteht aus Methoden. Wie sie sehen können, haben die Plug-ins normalerweise wenigstens zwei Methoden, die für sie automatisch erstellt werden, wenn sie ein neues Plug-in erstellen möchten:

- **Initialize**: diese Methode wird automatisch aufgerufen, immer wenn sie Sibelius starten. Normalerweise bedeutet es nichts anderes als Hinzufügung des Plug-in-Namens in das **Plug-ins-Menü**, obwohl sie bemerken werden (wenn sie sich einige mitgelieferte Plug-ins ansehen), dass diese Methode manchmal auch dazu verwendet wird, um die Ausgangswerte für die Datenvariablen zu bestimmen.

- **Run**: diese Methode wird aufgerufen, wenn sie das Plug-in starten, sie werden überrascht sein zu hören, dass es wie **main ( )** in C/C++ und Java ist. Mit anderen Worten, wenn sie auf den Eintrag **Test** im Menü **Plug-ins** klicken, wird diese **Run**-Methode aufgerufen, andernfalls würden die Plug-ins nicht im Stande sein, etwas zu tun.

Klicken sie auf **Run** und dann auf **Bearbeiten** (oder sie können gleich zwei Mal auf **Run** klicken, um zu bearbeiten), dann werden sie ein Dialogfenster sehen, wo sie die **Run** Methode einfügen können. Den Namen können sie in dem oberen Feld eingeben, im nächsten Feld können sie Parameter eingeben (d.h. die Variablen, wo die Werte sich an die Methode anpassen, werden gespeichert), und unten ist selbst der Code:

```
Sibelius.MessageBox("Test");
```

Dieses ruft die Methode **`MessageBox`** auf, die das Dialogfenster aufmacht, das **Test** heißt, wenn sie das Plug-in starten. Merken sie, dass der Methodenname nach dem Parameterverzeichnis in Klammern folgt. In diesem Fall gibt es nur einen Parameter, weil es ein string (bzw. Text) ist, d. h. ein Text in Gänsefüßchen. Sie merken weiter, dass die Zeile mit der Anweisung mit einem Semikolon endet, wie in C/C++ und Java. Fall sie vergessen haben, ein Semikolon zu tippen, bekommen sie eine Fehlmeldung, wenn sie das Plug-in starten.

Was bedeutet das Wort Sibelius in **`Sibelius.MessageBox`**? Im eigentlichen Sinne ist es eine Variable, die das Sibelius-Programm präsentiert. Diese Anweisung besagt, dass Sibelius das Dialogfenster aufrufen soll (C++ und Java Programmierer werden erkennen, dass diese Variable zu einem 'Object' gehört). Falls sie das noch nicht ganz verstehen, können sie es ignorieren, wir werden noch später darauf zurückkommen.

Einfügen eines Codes

Jetzt versuchen wir den Code langsam zu verändern. Sie können den Code wie in einem Textverarbeitungsprogramm einfügen, indem sie die Maus und die Pfeiltasten benutzen, sie können noch ⌘**X/C/V** oder **Ctrl(Strg)+X/C/V** fürs Ausschneiden, Kopieren und Einfügen benutzen, wenn sie die rechte Maustaste klicken, bekommen sie ein Kontext-Menü mit grundlegenden Optionen.
Ändern sie jetzt den Code auf folgende Weise:

```
x = 1;
      x = x + 1;
            Sibelius.MessageBox("1 + 1 = " & x);
```

Sie können überprüfen, ob das einen Sinn hat (oder wenigstens irgendeinen Sinn ergibt), indem sie auf die Schaltfläche **Syntax prüfen** klicken. Falls dort irgendwelche eklatante Fehler auftreten (z.B. fehlende Semikolons), werden sie darauf hingewiesen, wo diese sind.
Dann schließen nacheinander alle drei Dialogfenster, indem sie jedes Mal auf **OK** klicken. Starten sie ihr verbessertes Plug-in aus dem Menü Plug-ins, und ein Dialogfenster mit der Antwort **1 + 1 = 2** wird sich öffnen.

Wie funktioniert das? Die ersten zwei Zeilen werden einleuchtend sein. In der letzten Zeile wird **&** benutzt, um zwei strings (Texte in Gänsefüßchen) miteinander zu verbinden. Sie können kein **+** verwenden, da das Zeichen nur mit Zahlen funktioniert (wenn sie das Pluszeichen in oben genanntem Beispiel einsetzen, bekommen sie eine interessante Antwort).

Eine Falle: versuchen sie die zweite Zeile folgenderweise zu ändern:

```
x += 1;
```

dann klicken sie auf **Syntax prüfen**. Sie bekommen eine Fehlmeldung: diese Syntax (und die Syntax **`x++`**) ist in verschiedenen Sprachen erlaubt, aber nicht in ManuScript. Sie müssen daher eingeben:

```
x = x+1;
```

Wo die Plug-ins gespeichert sind

Die Plug-ins sind im **Plug-ins** - Unterordner gespeichert, der sich im **Sibelius-**Programmordner befindet. (**Sibelius Software\Sibelius 2\Plugins**). Das lohnt sich zu wissen, falls sie ein Plug-in einem Anderen geben möchten. So wie die Plug-ins in dem Unterordner erscheinen, stimmen sie mit dem Untermenü

Plug-ins im Menü **Datei** überein. Der Dateiname des Plug-ins, ist zugleich der interne Name des Plug-ins mit der Dateiendung **.plg**, in unserem Falle also **Test.plg**.

Zeilenumbrüche und Kommentare

Wie bei C/C++ und Java, können sie auch hier eine neue Zeile anfangen, wo sie möchten (nur in der Mitte des Wortes ist ein Umbruch nicht möglich), aber vergessen sie nicht, nach jeder Anweisung ein Semikolon zu setzen. Sie können mehrere Anweisungen auf eine Zeile setzen oder umgekehrt, d. h. eine Anweisung auf mehrere Zeilen verteilen.
Sie können Kommentare zu ihrem Programm einfügen, so wie bei C/C++ und Java. Alles, was nach `//` kommt, wird am Ende der Zeile ignoriert. Alles, was zwischen `/*` und `*/` steht, wird ignoriert, falls es ein Teil von einer Zeile oder von mehreren Zeilen ist.

```
// der Kommentar dauert bis zum Ende der Zeile an

/* sie können hier einige
Kommentarzeilen setzen
*/
```

z. B.

```
Sibelius.MessageBox("Hi!");   //druckt die aktive Partitur
```

oder:

```
Sibelius /* das enthält die Anwendung */ .MessageBox("Hi!");
```

Variablen

x ist im **Test**-Plug-in eine Variable. In ManuScript kann eine Variable eine beliebige Folge von Buchstaben, Zahlen oder `_` (der Unterstrich) sein, aber sie darf nicht mit einer Zahl anfangen.

Eine Variable kann eine ganze Zahl, eine Bruchzahl, einen Text oder ein Objekt (z. B. eine Note) beinhalten, mehr über Objekte wird später erklärt. Im Gegensatz zu den anderen Sprachen kann in ManuScript eine Variable alle Arten von Daten beinhalten, sie müssen nicht umschreiben, welche Art von Daten sie möchten. Auf diese Weise können sie eine Zahl in einer Variable speichern, dann anstatt einer Zahl einen Text, dann einen Objekt. Versuchen sie folgendes:

```
x = 56; x = x+1;
Sibelius.MessageBox(x); //druckt ‚57' im Dialogfenster
x = " das ist der Text "; //die Zahl, die es enthielt, ist weggefallen
Sibelius.MessageBox(x); //druckt 'das ist der Text' im Dialogfenster
x = Sibelius.ActiveScore; //jetzt enthält es eine Partitur
Sibelius.MessageBox(x); //druckt nichts im Dialogfenster
```

Die Variablen, die innerhalb einer ManuScript-Methode beschrieben sind, sind örtlich beschränkt, d.h. sie können mit anderen Methoden im selben Plug-in nicht verwendet werden. Auf alle globalen **Daten**variablen, die der Gebrauch des Plug-in-Editors definiert, kann mit allen Methoden im Plug-in zugegriffen werden.

Eine schnelle Übersicht der Texte in ManuScript stellt sich folgendemaßen dar: Wie die vielen anderen Programmiersprachen, so verwenden auch die ManuScript-Texte den Backslash `\` als 'Escape-Zeichen', um bestimmte Sachen zu repräsentieren. Falls sie ein einzelnes Anführungszeichen in ihrem Text einfügen möchten, verwenden sie `\'`, und wenn sie eine neue Zeile einfügen möchten, dann sollen sie folgendes setzen `\n`. Das bedeutet auch: um einen Backslash in ManuScript-Text einzufügen, muss man einen doppelten Backslash schreiben `\\` .

Umwandlung der Nummern, Texte und Objekte

Merken sie, dass die **`MessageBox`** Methode dafür ist, um einen Text darzustellen. Wenn sie eine Zahl stattdessen eingegeben haben (siehe oben, wenn **`MessageBox`** zum ersten Mal aufgerufen wird), wird die

Zahl in einen Text umgewandelt. Wenn sie stattdessen ein Objekt eingeben, (z. B. eine Partitur) wird kein Text hergestellt.

Ähnlich ist es, wenn die Berechnung eine Zahl verlangt, aber irgendein Text eingegeben wurde, dann wird der Text in eine Zahl umgewandelt:

```
x = 1 + "1";                    //das + Zeichen verlangt eine Zahl
      Sibelius.MessageBox(x);    //zeigt '2' an
```

Falls der Text nicht mit einer Zahl anfängt (oder falls die Variable aus einem Objekt und nicht aus einem Text besteht), dann wird der Text als 0 gewertet.

```
x = 1 + "fred";
      Sibelius.MessageBox(x); //zeigt '1' an
```

Schleifen (Loops)

‘for’ und ‘while’

Die ManuScript-Sprache hat eine **`while`**-Schleife, die einen Codeblock immer wieder ausführt, bis eine bestimmte Aussage **True** *(= wahr)* getroffen wird. Erstellen sie ein neues Plug-in, das **Potato** heißt. Das wird sie amüsieren, und für alle wird es lustig, wenn sie anfangen die Wörter des bekannten Songs ‘1 potato, 2 potato, 3 potato, 4’ zu schreiben. Tippen sie folgendes für die **Run**-Methode des neuen Plug-ins ein:

```
x = 1;
while (x<5)
{
      text = x & " potato, ";
      Sibelius.MessageBox(text);
      x = x+1;
}
```

Starten sie das Plug-in. Jetzt muss der Text ‘1 potato’, ‘2 potato’, ‘3 potato’, ‘4 potato’ angezeigt werden, der sich als Start versteht. Obwohl es lästig ist, aber sie müssen noch nach jeder Aussage (message) auf **OK** anklicken.

Nach der **`while`**-Anweisung folgt eine Bedingung, die in Klammern **`( )`** steht, dann folgt noch eine Blockanweisung, die in Mengenklammern **`{ }`** steht (nach der letzten Mengenklammer **`}`** müssen sie kein Semikolon setzten). Wenn die Bedingung richtig ist, wird der Block ausgeführt. Im Gegensatz zu den anderen Sprachen, sind die Mengenklammern obligatorisch (sie können sie nicht weglassen, auch wenn sie nur eine Anweisung erhält). Außerdem muss jeder Block wenigstens eine Anweisung enthalten. Daraus ergibt sich, dass die ManuScript eine einfache Sprache ist.

Im folgenden Beispiel können sie sehen, dass wir den Wert des **`x`** am Anfang der Schleife überprüfen, und dann am Ende den Wert des **`x`** erweitern. Diese allgemeine Konstruktion kann noch kürzer und klarer in ManuScript zum Ausdruck gebracht werden, wenn sie eine **`for`**-Schleife benutzen. Das obere Beispiel kann auch auf folgende Weise geschrieben werden:

```
for x = 1 to 5
{
      text = x & " potato,";
      Sibelius.MessageBox(text);
}
```

Hier fängt die Variable **`x`** mit dem Wert (1) an und hört bei dem Wert (5) auf, dann bleibt die Variable einen Schritt vor dem letzten Wert stehen. Gewöhnlich, ‘der Schritt’, der angewendet wurde, ist 1, aber wir könnten auch den Schritt 2 anwenden, in dem wir die folgende Syntax eingeben **`for x = 1 to 5 step 2`**. Das Ergebnis, das nachher gedruckt wird, zeigt uns nur ‘1 potato’ und ‘3 potato’!

Merken sie, dass das Zeichen **`&`** verwendet wird, um Strings (Texte) einzugeben. Da ein String auf der anderen Seite erwartet wird, wird der Wert des **`x`** in einen Text umgewandelt.

Bemerken sie bitte, dass ich die **Tab**-Taste gedrückt habe, um die Anweisung innerhalb der Schleife einzurücken. Es wäre gut, sich daran zu gewöhnen, da auf diese Weise die Struktur deutlicher wird. Falls sie Schleifen in Schleifen haben, dann sollten sie die inneren Schleifen wieder mit Tab-Taste einrücken.

Die if Anweisung

Jetzt können wir eine **if**-Anweisung hinzufügen, damit die letzte Phrase nur '4' lautet und nicht '4 potato':

```
x = 1;
while  (x<5)
{
      if(x=4)
      {
            text = x & ".";
      }
      else
      {
            text = x & " potato,";
      }
      Sibelius.MessageBox(text);
      x = x+1;
}
```

Die Regel für **if** nimmt die Form **if (Bedingung) {Objekte}** ein. Sie können noch fakultativ **else {Objekte}** hinzufügen, was angewendet wird, falls die Bedingung falsch ist. Genauso wie mit while, sind auch hier die Klammern und die Mengenklammern obligatorisch, außerdem können sie noch das Programm noch kürzer machen, indem sie die Mengenklammern in die gleiche Zeile einsetzen, wo die anderen Anweisungen sich befinden:

```
x = 1;
while (x<5)
{
      if(x=4)    {
            text = x & ".";
      } else {
            text = x & " potato,";
      }
      Sibelius.MessageBox(text);
      x = x+1;
}
```

Die Position der Mengenklammern ist reine Geschmacksache.
Lassen sie uns jetzt aus diesem Plug-in etwas wirklich Großartiges machen. Wir können alle 4 Aussagen in einer Variable aufbauen, die Text heißt, und dann die sie nur am Ende anzeigen lassen, indem wir die nützliche Haltbarkeit auf ihrer Maustaste absichern. Wir können noch um **if** und **else** Blocks austauschen, um der Gebrauch von **not** hervorzuheben. Schließlich kehren wir zurück zur Syntax **for**, die wir uns vorher angesehen haben.

```
text = "";        //fängt nicht mit dem Text an
for x = 1 to 5
{
      if (not(x=4))  {
            text = text & x & " potato,"; //irgendeinen Text hinzufügen
      }   else   {
            text = text & x & ".";  //fügt Nr. 4 ein
      }
}
Sibelius.MessageBox(text);          //zum Schluss anzeigen
```

Arithmetik

Wir haben das + Zeichen ohne Kommentare benutzt, hier nun ist die gesamte Liste der verfügbaren Rechen-zeichen:

`a + b`	addieren
`a - b`	subtrahieren
`a * b`	multiplizieren
`a / b`	dividieren
`a % b`	der Rest
`-a`	verneinen
`(a)`	wird zuerst bewertet

Der normale Vorgang dieser Zahlenaufgabe rechnet aus: **`2+3*4`** macht 14 und nicht 20, da das **`*`** Zeichen vor dem **`+`** Zeichen berechnet wird. Um das deutlicher zu machen, können sie schreiben **`2+(3*4)`**. Wenn sie die Zahl 20 als Ergebnis haben wollen, dann müssen sie **`(2+3)*4`** schreiben.

ManuScript unterstürzt auch die Umbruchzahlen, so während in den vorherigen Versionen **`3/2`** zu **`1`** abgerundet wurde, jetzt wird es zu 1.5 abgerechnet. Die Umwandlung von Bruchzahlen zu Ganzzahlen wird mit **`RoundUp`** **(***expr***)**, **`RoundUp`** **(***expr***)** und **`Round`** **(***expr***)** Funktionen erreicht, die mit jeder Aussage angewendet werden können.

(expr = expression = Aussage)

Objekte

Jetzt betrachten wir den Hauptaspekt der objektorientierten Sprachen wie ManuScript, C++ oder Java, die sich von den traditionellen Sprachen wie BASIC, Fortan und C unterscheiden. Die Variablen in traditionellen Sprachen können nur bestimmte Arten von Daten festhalten: Ganzzahlen, Bruchzahlen, Strings *(= Texte)* usw. Jede Art von Daten übt bestimmte Tätigkeiten aus, die angewendet werden können: Zahlen können multipliziert und dividiert werden, z. B. Strings können zusammengefügt werden, dann in eine Zahl oder von einer Zahl umgewandelt werden, nach anderen Strings suchen usw. Aber wenn ihr Programm mit vielen komplexen Datenarten arbeitet, (die im Prinzip verglichen werden können, in dem man folgende Zeichen verwendet **=, <** und **>**, in einen Text oder von einem Text umgewandelt werden können und sogar subtrahiert werden), dann bleibt ihnen nichts anderes übrig, als für sich selbst zu sorgen.

Objektorientierte Sprachen können direkt mit vielen komplexen Datenarten arbeiten. Somit können sie in ManuScript eine Variable setzen, z. B. **`this_chord`**, damit in ihrer Partitur ein Akkord erscheint und mehrere Noten hinzugefügt werden:

```
thischord.AddNote(60);  //fügt mittlere Note C hin (Note Nr. 60)
thischord.AddNote(64);  //fügt E hin (Note Nr. 64)
```

(Anmerkung des Übersetzers: chord = Akkord)

Das ist keine Magie. Das gleiche wird auch in BASIC mit Texten (Strings) angewendet, wo viele bestimmte Arbeitsabläufe nur für einen Text gelten.

```
A$ = "1"
A$ = AS + " potato, ":        REM add strings
X = ASC(A$):                  REM get first letter code
```

In ManuScript können sie eine Variable setzen, die ein Akkord, eine Note im Akkord, ein Takt, eine Notenzeile oder sogar eine ganze Partitur sein kann und dann mit ihnen arbeiten. Wieso nicht? Vielleicht würden sie auch den Wunsch haben, eine Variable als eine ganze Partitur zu setzen. Die können sie dann zum Beispiel speichern oder ein neues Instrument hinzufügen.

Die Funktion der Objekte

Wir werden uns noch ansehen wie die Musik in ManuScript momentan dargestellt ist, aber um eine kleine Probe zu machen, lasst uns unser Text **Potato** bearbeiten und aus ihm eine Partitur erstellen.

```
x = 1;
text = "";        //fängt nicht mit dem Text an
while   (x <5)
{
      if   (not (x=4) )  {
            text = text & x & " potato, "; //fügt irgendeinen Text hinzu
      } else {
            text = text & x & ".";        //fügt Nr 4 hinzu
      }
      x = x+1;
}

Sibelius.New();                     //erstellt eine neue Partitur
newscore = Sibelius.ActiveScore;    //setzt die in eine Variable
newscor.CreateInstument("Piano");
staff = newsore.NthStaff(1);        //bestimmt die obere Notenzeile
bar = staff.NthBar(1);        //bestimmt den ersten Takt dieser Notenzeile

bar.AddText(0,text,"Technique");    //benutzt den Textstil Technik
```

Dieses Plug-in erstellt eine Partitur mit Klavier und tippt unseren Potato-Text im ersten Takt als Techniktext ein.

Der Code benutzt den Punkt '**.**' mehrere Male, immer in der Form **`variable.variable`** oder **`variable.method()`**. Das zeigt, dass die Variable vor dem Punkt ein Objekt erhalten soll.

- Wenn nach dem Punkt der Name einer Variable steht, dann bekommen wir eine von den Sub-Variablen des Objekts (in einigen Sprachen heißt es 'fields' oder 'member variables'). Z. B. Wenn **`n`** eine Variable ist, die eine Note enthält, dann ist **`n.Pitch`** eine Zahl, die ihre MIDI pitch = Tonhöhe (z.B. 60 für den mittleren Ton C) repräsentiert, und **`n.Name`** ist ein String, der seine Pitch beschreibt (z. B. 'C4' für den mittleren C). Die Variablen, die für jede Objektart verfügbar sind, werden nachher aufgezählt.
- Wenn nach dem Punkt (in Klammern **`()`**) der Name einer Methode steht, dann wird eine von diesen Methoden, die für diese Art des Objekts zulässig ist, aufgerufen. Auf diese Weise aufgerufene Methode wird entweder das Objekt vertauschen oder den Wert zurückbringen. Z. B. wenn **`s`** eine Variable ist, die eine Partitur enthält, dann fügt **`s.CreateInstument("Flute")`** eine Flöte hinzu (die Partitur wird dadurch verändert), aber **`s.NthStaff(1)`** bringt den Wert zurück, nämlich ein Objekt, das die erste Notenzeile enthält.

Lassen sie uns jetzt den neuen Code im Einzelnen betrachten. Es gibt eine Variable, die vorher definiert wurde, die heißt Sibelius und enthält einen Objekt, das das Sibelius Programm selbst repräsentiert. Wir haben uns die Methode **`Sibelius.MessageBox()`** angesehen. Die Methode, die **`Sibelius.New()`** heißt, sagt, dass Sibelius eine neue Partitur erstellen soll. Jetzt wollen wir irgendetwas mit dieser Partitur machen, deshalb müssen wir die in eine Variable setzen.

Zum Glück wird die neu erstellte Partitur aktualisiert (z. B. ihre Titelfläche wird hervorgehoben und alle anderen Partituren deaktiviert), deshalb können wir Sibelius gleich um die aktive Partitur bitten und die in die Variable setzen:
`Newscore = Sibelius.ActiveScore`.

Dann können wir der Partitur den Befehl geben, ein Klavier zu erstellen:
`newscore.CreateInstument("Piano")`.

Aber um einen Text in die Partitur hinzuzufügen, müssen sie verstehen wie das Layout dargestellt wird.

(Übers.: Score = Partitur
Staff = Notenzeile
create = erstellen)

Darstellung in einer Partitur

Eine Partitur wird als eine Hierarchie betrachtet: jede Partitur enthält 0 oder mehrere Notenzeilen, jede Notenzeile enthält Takte (obwohl jede Notenzeile dieselbe Taktanzahl hat) und jeder Takt enthält 'Taktobjekte'. Schlüssel, Text und Akkorde sind alles verschiedene Arten von Taktobjekten.

Ein Taktobjekt (d. h. ein Objekt, das zu einem Takt gehört) wird genauso wie ein Text der Partitur hinzugefügt. Zuerst müssen sie bestimmen, welche Notenzeile sie möchten (und dieses in eine Variable setzen):

```
staff = newscore.NthStaff(1);
```

Dann müssen sie den Takt, den sie in dieser Notenzeile möchten, bestimmen (und die Bestimmung in eine Variable setzen):

```
bar = staff.NthBar(1); //bar bedeutet Takt
```

Zum Schluss geben sie den Befehl, dem Takt noch einen Text hinzuzufügen:

```
bar.AddText(0,text,"Technique").
```

Sie müssen den Namen des Textstils (oder die Indexnummer - für die Einzelheiten schauen sie sich die Verweisstelle an) angeben, der ein Notenzeilentext sein muss, da wir den Text in eine Notenzeile hinzufügen.

Merken sie, dass die Takte und die Notenzeilen ab 1 aufwärts gezählt werden; das gilt für die Takte unabhängig von den Taktzahländerungen, die in der Partitur zu finden sind, deshalb ist die Nummerierung immer eindeutig. Was den Notenzeilen angeht, ist die obere Notenzeile die Nr. 1, und weiter werden so alle Notenzeilen gezählt, sogar wenn sie ausgeblendet sind, deshalb hat eine bestimmte Noteinzeile die selbe Nummer, unabhängig davon, wo sie in der Partitur erscheint.

Für die Takte wird später die **`AddText`**-Methode erläutert, aber der erste Parameter, den sie annimmt, ist die rhythmische Position im Takt. Jede Note hat im Takt eine rhythmische Position, die die Stelle der Note angibt (am Anfang, ein Viertel nach dem Anfang usw.), das gleiche gilt auch für alle anderen Objekte der Takte. Das zeigt, wo das Objekt verankert ist, das gleiche gilt auch für den Techniktext, d. h. der Techniktext wird verankert dort, wo die linke Seite des Textes anfängt. Somit, um unseren Text am Anfang des Taktes zu setzen, verwenden wir die Wert 0, um den Text ein Viertel vom Taktanfang entfernt zu setzen, verwenden wir 256 (die Einheiten sind 1024, so ein Vierten davon macht 256, das ist gar nicht so schwer).

```
bar.Addtext(256,text,"Technique");
```

Um unklare Zahlen wie 256 in ihrem Programm zu vermeiden, gibt es schon eine bereits definierte Variable, die verschiedene Notenwerte darstellt (sie werden später aufgezählt), also können sie schreiben:

```
bar.AddText(Quarter,text,"Technique");
```

oder um kurios zu sein, können sie das britische Äquivalent benutzen:

```
bar.AddText(Crotchet,text,"Technique");
```

Für ein punktiertes Viertel, können sie anstatt 384 eine andere schon bereits definierte Variable setzen:

```
bar.AddText(DottedQuarter,text,"Technique");
```

oder zwei Variablen setzen:

```
bar.AddText(Quarter+Eighth,text,"Technique");
```

Das ist viel deutlicher, als Zahlen zu verwenden.

Die Systemnotenzeile

Wie sie schon von der Anwendung des Sibeliusprogramms wissen, gehören einige Objekte zu einer Notenzeile, sondern zu allen Notenzeilen. Dazu zählen Titel, Tempotext, spezielle Taktstriche; sie können es so anordnen, dass Objekte zu allen Notenzeilen gehören, das ist zum Beispiel dafür gut, wenn sie in allen Stimmen extrahiert werden sollen.

Alle diesen Objekte werden natürlich in einer ausgeblendeten Notenzeile gespeichert, die Systemnotenzeile heißt. Sie können sie als eine unsichtbare Notenzeile betrachten, die immer über den anderen Notenzeilen in einem System steht. Die Systemnotenzeile besteht wieder aus Takten wie eine normale Notenzeile. Um so den Titel 'Potato' zu unserer Partitur hinzuzufügen, brauchen wir den folgenden Code in unserem Plug-in.

```
sys = newscore.SystemStaff ;  //Systemnotenzeile ist eine Variable
bar = sys.NthBar(1);
bar . AddText(0,"POTATO SONG","Subtitle");
```

Wie sie sehen, ist **`SystemStaff`** eine Variable, die sie direkt aus einer Partitur ergreifen können. Vergessen sie nicht, dass sie einen Systemtextstil verwenden müssen (hier habe ich Untertitel verwendet), wenn sie einen Text in einen Takt einer Systemnotenzeile setzen. Ein Textstil wie **`"Technique"`** wird nicht funktionieren. So müssen sie den Takt und die Position im Takt bestimmen; das ist natürlich überflüssig für einen Text, der wie Titel zentriert wird, aber für Tempo und Metronomangabe sind diese Angaben erforderlich.

Darstellung der Noten, Pausen, Akkorde und anderer musikalischer Begriffe

Sibelius repräsentiert einheitlich Noten, Pausen und Akkorde. Eine Pause hat keinen Notenkopf, eine Note hat einen Notenkopf und ein Akkord hat zwei oder mehrere Notenköpfe. Das schafft eine neue Hierarchie: die meisten Schnörkel, die sie in der Partitur sehen, sind natürlich spezielle Objektarten eines Taktes, die sogar noch kleinere Dinge enthalten (nämlich Noteköpfe). Im Großen und Ganzen gibt es also keinen Namen, der gleichzeitig eine Pause, Note und ein Akkord sein kann, deshalb haben wir uns den Namen NoteRest (NotePause) ausgedacht. Ein NoteRest mit 0,1 oder 2 Notenköpfen ist das, was sie normalerweise eine Pause, eine Note oder einen Akkord nennen.

Wenn **`n`** eine Variable ist, die Note/Akkord/Pause enthält, dann wird es eine Variable geben, die **`n.NoteCount`** heißt, die Anzahl der Noten enthält und **`n.Duration`**, die den Notenwert in 1/256 eines Viertel bestimmt. Sie können noch **`n.Highest`** und **`n.Lowest`** erstellen, die die hohen und tiefen Noten enthalten (angenommen, dass **`n.NoteCount`** keine 0 ist). Wenn sie die **`lownote = n.Lowest`** erstellen, dann können sie einiges über die tiefen Noten herausfinden, wie zum Beispiel **`lownote.Pitsch`** (eine Zahl) und **`lownote.Name`** (ein Text).
Alle Einzelheiten über diese Methoden und Variablen finden sie in der Verweisstelle.

Andere Musikbegriffe wie Schlüsseln, Linien, Liedtext haben entsprechende Objekte in ManuScript, die wieder verschiedene Variablen und Methoden haben. Z. B. wenn sie eine Linienvariable **`ln`** haben, dann wird die **`ln.EndPosition`** die rhythmische Position angeben, wo die Linie enden soll.

Die ‘for each’ Schleife

Die allgemeinen Aufgaben einer Schleife sind Arbeitsgänge, die an jeder Notenzeile in einer Partitur oder an jedem Takt in einer Notenzeile oder an jedem Taktobjekt in einem Takt oder an jeder Note in einer Note/Akkord/Pause-Darstellung ausgeführt wird. Es gibt noch mehrere andere komplexe Aufgaben allgemeiner Natur, wie zum Beispiel einige Arbeitsabläufe, die an jedem Taktobjekt in einer Partitur in chronologischer Anordnung oder an jedem Taktobjekt in vielfacher Auswahl ausgeführt werden. ManuScript hat eine **for each** (= für jeden) Schleife, die jeden von diesen Arbeitsabläufen mit einer einzelnen Anweisung erreichen kann.

Die einfache Form der **for each** Schleife stellt folgendes Bespiel dar:

```
thisscore = Sibelius.ActibeScore;
for each s in thisscore // setzt s zu jeder Notenzeile der Reihe nach ein
{
            // . . . mache etwas mit s
}
```

Hier ist **thisscore** eine Variable, die eine Partitur enthält, die Variable *s* wird in jede Notenzeile in **thisscore** (in dieser Partitur) der Reihe nach eingesetzt. Das ist so, weil die Notenzeilen Objektarten in der nächsten Hierarchieebene der Objekte darstellen. (schauen sie sich **die Hierarchie der Objekte** im Abschnitt **Referenz** an). Für jede Notenzeile in der Partitur wird die Anweisung in Mengenklammern **{}** gesetzt.
Wie wir schon gesehen haben enthalten die Partiturobjekte Notenzeilen, aber sie können auch ein Auswahlobjekt enthalten, d. h. wenn man vor dem Start des Plug-ins eine Musikpassage ausgewählt. Das Auswahlobjekt ist ein besonderer Fall: es reagiert nicht auf die **for each** Schleife, weil es sich nur um ein einzelnes Auswahlobjekt handelt und nicht um die gesamte Partitur. Wenn sie das Auswahlobjekt in einer **for each** Schleife benutzen, dann bekommen sie gewöhnlich ein Taktobjekt zurück (und nicht eine Notenzeile, einen Takt oder etwas anderes!).

Wir nehmen einen anderen Beispiel, diesmal für Noten innerhalb einer NoteRest (= Note/Akkord/Pause-Hirarchie):

```
noterest = bar.NthBarObject(1);
for each n in noterest              // füge  n zu jeder Note hinzu
{
            Sibelius.MessageBox("Pitch is" & n.Name);
}
```

n ist an jede Note im Akkord der Reihe nach angebunden und ihr Notenname wird angezeigt. Das funktioniert so, weil die Noten die nächsten Objekte unter der Hierarchie Note/Akkord/Pause sind. Und zwar, wenn NoteRest (Note/Akkord/Pause) eine Pause ist (weder eine Note noch ein Akkord), dann kann die Schleife niemals ausgeführt werden. Sie müssen es nicht extra ausprobieren.
Die selbe Form von Schleifen kann Takte aus einer Notenzeile oder einem Notenzeilesystem und Taktobjekte aus einem Takt erhalten. Diese Schleifen sind oft ineinander geschachtelt, wodurch sie zum Beispiel mehrere Takte aus mehreren Notenzeilen erhalten können.

Die erste Form der **for each** Schleifen bekommt die Objekte in der Reihenfolge aus einem Objekt der nächsten Objekthierarchieebene. Die zweite Form der **for each** Schleifen ermöglicht ihnen die Ebenen in der Hierarchie zu überspringen, indem sie bestimmen welche Objektart sie erhalten möchten. Das sichern mehrere geschachtelte Schleifen:

```
thisscore = Sibelius.ActiveScore;
for each NoteRest n in thisscore
{
      n.AddNote(60);          // füge das mittlere C hinzu
}
```

Nachdem **for each** die **NoteRest** bestimmt, weiß dann Sibelius genau, dass jede Note/Akkord/Pause in jedem Takt und jeder Notenzeile der Partitur gemeint ist, andernfalls wird Sibelius nur jede Notenzeile in der Partitur ansteuern, weil ein Notenzeilenobjekt eine Objektart in der nächsten Objekthierarchieebene ist. Die Note/Akkord/Pause-Hirarchie wird in einer nützlichen Anordnung ausgeführt, nämlich von der oberen bis zur unteren Notenzeile, dann von links nach rechts in den Takten. Das ist die chronologische Anordnung. Wenn sie eine andere Anordnung möchten (z. B. alle Noten/Akkorde/Pausen in dem ersten Takt jeder Notenzeile, dann alle Noten/Akkorde/Pausen in dem zweiten Takt jeder Notenzeile usw.), dann müssen sie geschachtelte Schleifen verwenden.

Hier ist ein nützlicher Code, der jeder Note in der Partitur eine Oktave hinzufügt:

```
score = Sibelius.ActiveScore;
for each NoteRest chord in score{
            if(not(chord.NoteCount = 0)){  // ignoriert die Pausen
      note = chord.Highhest;        //fügt über die obere Note ein
      chord.AddNote(note.Pitch+12)  // 12 ist die Intervallzahl der Oktave
                                    // in Halbschritten
                                    // (semitones)meint den Halbton
      }
}
```

Wir können das Beispiel sehr leicht verbessern, indem wir die Oktavverdopplung der Noten nur in bestimmten Takten oder Notenzeilen definieren, natürlich nur in dem Fall, wenn die Noten bestimmte Tonhöhen oder Längen usw. haben.

Solch eine Schleife ist im Zusammenhang mit der aktuellen Auswahl des Benutzers sehr nützlich. Diese Auswahl kann durch eine Variable erreicht werden, die ein Partiturobjekt wie folgt enthält:

```
selechtion = score.Selection; // übersetzt: Auswahl = Partitur.Auswahl
```

Wir können dann testen, ob dieses eine Passageauswahl ist, und so können wir uns alle ausgewählten Takte mit einer **for each** Schleife ansehen:

```
if  (selechtion.IsPassage) {  // Übers.: falls Auswahl.IstPassage
      for each Bar b in selechtion { // Übers.: für jeden Takt in Auswahl
            // mache etwas mit diesem Takt
            ...
      }
}
```

Umwege, Datenfelder (arrays) und hashes –Tabellen

Umwege (Indirection)

Wenn sie das Zeichen **@** vor den Namen einer Textvariablen setzen, dann wird der *Wert* der Variable als Name einer Variable oder einer Methode angewendet, z. B.:

```
var="Name";
x=@var;     // setzt x für die Inhalt des Variablennamens
mymethod="Show";
@mymethod();  // ruft die Showmethode auf
```

Dies hat viele Vorteile, aber wenn es im Übermaß gebraucht wird, dann kann es zu Gehirnschädigungen führen. Z.B. können sie das Zeichen `@` gebrauchen, um 'unbegrenzte' Datenfelde vorzutäuschen. Wenn der Name eine Variable ist, die den folgenden String enthält **`"x1"`**, dann wird **`@name`** dem Gebrach der Variable **`x1`** direkt entsprechen. Somit:

```
i = 10;
name = "x" & i;
@name = 0;
```

Die Variable **`x10`** wird auf 0 gesetzt. Die letzen zwei Zeilen sind äquivalent zu **`x[i] = 0;`** in der C-Sprache. Hier gibt es viele Anwendungsmöglichkeiten; wie auch immer, sie können sich noch für die Verwendung von Datenfelder (und hashes-Tabellen) entscheiden, die unten beschrieben werden.

Datenfelder (arrays) und Haschee-Tabellen (hashes)

Die oben beschriebene Methode, die für die Anwendung von 'unechten' Datenfelden durch Indirektion (Umwege) erfolgt, ist in der Praxis ein bisschen knifflig. Die ManuScript_Sprache bietet auch noch eingebaute Datenfeldobjekte an. Um ein Datenfeld zu erstellen, verwenden sie die Einbaumethode **`CreateArray()`**,wie folgt:

```
array = CreateArray();
```

Die Elemente des Datenfeldes können dann angesetzt und gelesen werden, indem der **`[]`** Datenfeldindexoperator verwendet wird. Hier bemerken sie, dass es nicht notwendig ist, die Größe des Datenfeldes zu bestimmen, denn die richtige Größe wird erweitert, indem sie weitere Eintragungen hinzufügen. Alle Variablen der Datenfelder haben eine angeschlossene Mitgliedervariablen (member variable), die **`NumChildren`** heißen, sie geben die aktuelle Größe des Datenfeldes an. Ein einfaches Beispiel soll das verdeutlichen:

```
array = CreateArray();
array[0] = "one";
array[1] = "two";
array[2] = "three";

for x = 0 to array.NumChildren {
      Sibelius.MessageBox(array[x]);
}
```

Leider können diese Einbaudatenfelder nur Zahlen und Strings speichern (nicht beliebige Objekte), deshalb gibt es noch die Anwendung für die 'unechten' Datenfelder, die oben erklärt wurden, wenn wir z.B. Datenfelder wie Noten erstellen möchten. Daher sind die Einbaudatenfelder viel praktischer, wenn sie nur Strings und Zahlen speichern möchten.
Ein ähnlicher Begriff für die Datenfelder sind die Haschee-Tabellen. Hier werden die Partiturwerte in den Tabellen gespeichert, so dass sie von einem 'key string' (Schlüsseltext) karteimäßig erfasst werden können. (das ist eher einfacher bei Zahlen, als im Falle von Datenfeldern). Dieses kann sehr nützlich sein, wenn wir Werte mit bestimmten speziellen 'key strings' vereinigen möchten. Um eine Hasch-Tabelle in ManuScript zu erstellen, verwenden sie einfach die Einbaufunktion **`Create-Hash()`** genau so, wie wir es

mit **`CreateArray()`** oben gemacht haben. Diese können dann abgefragt werden, indem der **`[]`** Operator wie vorher verwendet wird. Hier ist ein Beipiel:

```
table = CreateHash();
table["key one"] = "first value";
table["key two"] = "second value";
table["key three"] = "third value";

Sibelius.MessageBox(table["key one"]);
x(table["key one"]);
```

Das wird den ersten Wert (**`"first value")`** in einer Dialogbox ausgeben. Wenn wir versuchen auf ein Objekt der Hasch- Tabellen oder des Datenfeldes in einem Verzeichnis zuzugreifen, das augenblicklich keine Verbindung hat (z.B. wenn wir versuchen auf **`table["key four"]`** im oberen Beispiel zuzugreifen) dann bekommen wir den speziellen Wert n**`ull`** zurück.

Datenfelder und Hasch-Tabellen sind Objekte und nicht Strings, deshalb können sie nicht ausgedruckt und in einer Messagebox angezeigt werden u. s. w. Wie auch immer, wenn sie den Wert des Datenfeldes oder der Hasch-Tabelle als einen String verwenden möchten, können sie die **`WriteToString()`**-Methode benutzen, beispielsweise: *all_array* = *arrayname* **`.WriteToString.()`**.

Dialogeditor

Um noch kompliziertere Plug-ins als die vorherigen zu erstellen, kann es nützlich sein, den Benutzer zu verschiedenen Optionen zu veranlassen. Dieses kann durch den Gebrauch des eingebauten Dialogeditors in ManuScript funktionieren (der leider nur in der Windowsversion von Sibelius verfügbar ist). Die Dialoge können ähnlich wie die Methoden und Datenvariablen im Plug-in Editor erstellt werden, hier wird nur das 'Dialog' Fenster gewählt und darin klickt man auf die 'Add' Schaltfläche.

Um einen Dialog mit der Methode von ManuScript zu zeigen, verwenden wir den Einbauabruf:

`Sibelius.ShowDialog(dialogName, Self)` *(show – zeigen, self – selbst)*

Der **`dialogName`** ist der Name des Dialogs, den wir zeigen möchten, und **`Self`** ist eine 'spezielle' Variable, die diesem Plug-in angehört (Sibelius bekommt den Befehl, was dem Dialog angehört). Die Steuerung wird nur dann zur Methode zurückkehren, sobald der Benutzer den Dialog schließt.

Natürlich sagt das nicht viel, solange wir die Erscheinung des Dialoges nicht verstehen können. Um zu sehen wie der Editor funktioniert, editieren sie das **StreicherfingersatzHinzufügen**-Plug-in, wählen sie den Dialog, der **`window`** heißt, und klicken Sie **Edit**. Der Plug-in Dialog wird zusammen mit einer langen 'Palette' der verfügbaren Steuerungen erscheinen. Dies sind folgende:

- Radio button
- Check box
- Button
- Static text
- Editable text
- Combo text
- List box

Der **StreicherfingersatzHinzufügen** *(Add String Fingering)* - Dialog verwendet nicht alle verfügbaren Steuerungen, aber er demonstriert vier von ihnen, nämlich checkboxes, static text, combo boxes und buttons. Jede Steuerung in diesem Dialog kann für das Editieren gewählt werden, indem sie einfach angeklickt werden, ein kleiner, schwarzer 'Griff ' wird erscheinen, durch den die Steuerungselemente in der Größe angepasst werden können. Die Steuerungselemente können innerhalb des Dialogs hin und her bewegt werden, indem man sie anklickt und zieht. Um neue Steuerungen zu erstellen, ziehen sie eine von diesen Symbolen von der Steuerungspalette hinüber zu dem Dialog selbst, und eine neue Steuerung von dieser Art wird erstellt.

Das Hauptmerkmal des Dialogeditors ist das Fenster **Properties** (Eigenschaften), auf das zugegriffen werden kann, indem man auf dem **Control (Strg)** oder rechts klickt und dann **Properties** aus dem Dialogmenü wählt. Falls keine Steuerungen gewählt sind, kann man verschiedene Optionen über den Dialog selbst setzen, wie z.B. Höhe, Breite, Titel. Wenn eine andere Steuerung gewählt ist, dann variiert das Eigenschaftsfenster, in Abhängigkeit von der Art der Steuerung. Die meisten Optionen aber sind für alle Steuerungen gleich, dies sind folgende:

- **Text**: Der Text erscheint in der Steuerung
- **Position (X, Y)**: wo die Steuerung im Dialog erscheint, oben links angeordnet
- **Size (width, height)**: *(Sitz(Breite, Höhe))* die Größe der Steuerung
- **Variable storing contol`s value** *(Variable speichert den Wert der Steuerung)*: die ManuScript **Data**-Variable, die mit dem Wert der Steuerung übereinstimmt, kontrolliert, wenn das Plug-in startet
- **Method called when cklicked** *(die Methode wird mit dem Klick aufgerufen)*: Die ManuScript Methode wird aufgerufen, wann immer der Benutzer diese Steuerung anklickt (lassen sie es weg, wenn man nicht zu wissen braucht, wie der Benutzer auf die Steuerung klicken soll)
- **Click closes dialog** *(mit dem Klick wird der Dialog geschlossen)*: wählen sie diese Option, wenn sie den Dialog schließen möchten. Die zusätzlichen Optionen 'returning True / False' (richtig/ falsch zurückbekommen) bestimmen die Werte, die die **`Sibelius.ShowDialog`** Methode rückgängig machen soll, wenn das Fenster auf diese Weise geschlossen wird.
- **Give this control focus** *(geben sie dieser Steuerung die Bildschärfe ein)*: wählen sie diese Option, falls die 'input focus' Eingabebildschärfe zu dieser Steuerung hinzugefügt werden soll, wenn der

Dialog geöffnet ist, d.h. falls es die Steuerung sein soll, zu der die Tastatur des Benutzers zutrifft, wenn der Dialog geöffnet ist. Hauptsächlich ist es für die Steuerungen der Editable Text nützlich.

Combo-boxes und list-boxes haben zusätzliche Eigenschaften; sie können eine Variable setzen, von der die Steuerungsliste der Werte nehmen kann. Da der Wert den aktuellen Wert der Steuerung speichert, muss es eine globale Datenvariable sein. Wie auch immer, in diesem Fall haben sie eine ziemlich spezielles Format, um eine Liste von Strings leichter als einen Einzeltext zu bestimmen. Schauen sie sich die Variable an, z.B. **`_ComboItems`** in **`StreicherfingersatzHinzufügen`**, das wird so aussehen:

```
_ComboItems
{
     "1"
     "2"
     "3"
     "4"
     "1 and 3"
     "2 and 4"
}
```

Die Radioschaltfläche hat auch in der neuen Version 2.1 von Sibelius eine zusätzliche Eigenschaft, die zulässt, dass man die Gruppen der Radioschaltflächen im Plug-in Dialog bestimmt. Wenn der Benutzer auf eine Radioschaltfläche innerhalb einer Gruppe klickt, dann werden die anderen Radioschaltflächen, die zu dieser Gruppe gehören, deaktiviert; alles andere im Dialog bleibt so wie es ist. Diese Eigenschaft wird hauptsächlich für die komplexeren Dialoge benutzt, beispielsweise das AkkordSymboleHinzufügen-Plug-in verwendet diese Eigenschaft.

Um eine Radiogruppe festzulegen, wählen sie eine Steuerung von jeder Gruppe aus, die die erste Schaltfläche der Gruppe repräsentiert, und für diese Steuerungen stellen sie dann sicher, dass das Checkfenster "Start a new radio group" (Starten sie eine neue Radiogruppe) im Eigenschaftsdialog der Steuerung angekreuzt ist. Dann setzen sie den Erstellungsauftrag (creation order) der Steuerungen an, indem sie auf den "Set Creation Order" (Ansetzen Erstellungsauftrag) im Dialogmenü, der mit dem rechten Mausklick oder Steuerungsklick durch den der Plug-in-Dialog erreicht werden kann, klicken. Nachdem das alles getan ist, werden sie über jeder Steuerung eine Zahl sehen, genau in der Reihenfolge wie die Steuerungen erstellt worden sind, als der Dialog geöffnet war. Das kann aber verändert werden, indem man auf jede Steuerung in der Reihenfolge anklickt, die man sich wünscht. Eine Radioschaltflächengruppe wird wie alle Radioschaltflächen definiert, die zwischen zwei Schaltflächen, die den "Start a new radio group" Zeichensatz haben, (oder zwischen einer von diesen Schaltflächen und dem Dialogende) erstellt worden sind. Damit die Radiogruppe richtig arbeitet, stellen sie sicher, ob jede Gruppe in der Reihenfolge erstellt worden ist, mit der zuerst die Schaltfläche, die am Anfang der Gruppe und erst dann alle anderen Radios in der Gruppe erstellt worden sind. Um es fertig zu stellen, klicken sie wieder auf das "Set Creation Order" Menü und dieses Verfahren wird deaktiviert.

Andere Eigenschaften sind für die Steuerungen static text (damit kann man festsetzen, ob der Text links oder rechts ausgerichtet werden soll) und für die Steuerungsbuttons *(= Schaltflächen)* (wo eingestellt wird, ob die Steuerung Defaultschaltfläche sein kann oder nicht) verfügbar. Wenn sie sich Beispiele ansehen möchten, dann schauen sie die mitgelieferten Plug-ins an, einige von ihnen beinhalten komplexe Dialoge.

Auf etwas anderes gefasst sein

Das Quint- und Oktavparallelen-Plug-in (**QuintUndOktavParallelen**) zeigt einige Methoden, die wir bis jetzt nicht verwendet haben. Das **Proof-read** Plug-in erläutert, dass ein Plug-in ein anderes Plug-in aufrufen kann - dieses Plug-in macht selber nicht anderes als nur andere Plug-ins wie **PizzicatoPrüfen**, **SchlüsselPrüfen**, **CheckRepeats** und **HarfenPedalPrüfen** aufzurufen. Auf diese Weise können sie Meta-Plug-ins bauen, die die Bibliotheken der anderen Plug-ins verwenden. Cool, nicht wahr?

(Übersetzung: Clef - Schlüssel, Repeat - Wiederholung, Suspect - verdächtig, suspekt)

(Als objektorientierte Programmierer sollen sie informiert sein, dass das Programm in dieser Weise funktioniert, weil jedes Plug-in ein Objekt ist, genau so wie die anderen Objekte in der Partitur, auf diese Weise kann jedes Plug-in die Methoden und die Variablen eines Anderen verwenden).

Fehlerbeseitigung (Debugging) in Plug-ins

Wenn ein Computerprogramm entwickelt wird, ist es mit großer Leichtigkeit möglich, dass unbedeutende (und nicht so unbedeutende!) Fehler oder Defekte passieren können. Da die ManuScipt-Sprache ein einfaches und leichtes System hat, ist die Fehlersuche und Behebung nicht notwendig, trotzdem können sie das Plug-in Fenster der Fehlerbeseitigung verwenden (verfügbar durch das **Plug-ins** Menü), um Fehler in ihrem Plug-ins zu finden.

Ein spezieller ManuScript-Befehl, **`trace(`***string*`)`**, druckt den festgelegten Text *(= string)* in das Fehlerbeseitigungsfenster. Das ist nützlich, wenn man sich ansehen will, was das Plug-in an bestimmten Stellen macht. Diese Befehle können dann entfernt werden, wenn man mit der Fehlerbeseitigung fertig ist. Ein wichtiges Merkmal für das Fehlerbeseitigungsfenster ist die Funktion, die die Ablaufverfolgung aufruft. Wenn diese startet, dann zeigt das Protokoll an, welche Funktionen durch das Plug-ins aufgerufen werden.

(Übersetzung: trace = Fehlerbeseitigung)

Bei der Fehlerbeseitigung kann uns der Befehl **`trace(`***string*`)`** zu einer Falle bringen. Es wurde schon erwähnt, dass die eingebauten Hasch-Tabellen und die Objekte des Datenfeldes keine strings sind, deshalb können sie auch nicht im Fehlerbeseitigungsfenster angezeigt werden. Das Problem kann durch die entsprechende Methode **`WriteToString()`**, die beide Objekte haben, verhindert werden, weil sie diese Methode in ein String umwandelt, dadurch wird die ganze Struktur des Datenfeldes oder der Hasch-Tabelle dargestellt. Auf folgende Weise können wir die aktuellen Werte einer Datenfeldvariablen schrittweise verfolgen:

```
trace("array variable = " & array.WriteToString();
```

Referenz

Syntax

Das hier ist ein formloser Bericht der Syntax von ManuScript.

Eine Methode besteht aus einer Liste von Anweisungen in folgender Darstellung:

block `{`*Anweisungen*`}`

z. B.

```
{

      a = 4 ;

}
```

while `while` `(`*expression*`)` *block*

z. B.

```
while    (i<3) {
Sibelius.MessageBox(i);
i=i+1;
}
```

if else `if` `(`*expression*`)` *block* `[ else` *block* `]`

z. B.

```
if (found) {
      Application.ShowFindResults(found);
} else {
Application.NotFindResults();
}
```

for each *(für jeden)* `for each` *variable* `in` *expression*
block

Das setzt die *Variable* zu jedem Unterobjekt (sub-object) innerhalb des Objekts an, der durch den Ausdruck angegeben wird.
Normalerweise gibt es nur eine Art des Unterobjekts, die den Objekt aufnehmen kann. Zum Beispiel eine Note/Akkord/Pause (wie z. B. ein Akkord) kann nur Notenobjekte enthalten. Wie auch immer, wenn mehr als eine Art des Unterobjekts möglich wird, dann können sie die Art bestimmen:

```
for each Type variable in expression
<block>
```

Zum Beispiel:

```
for each NotRest n in thisstaff {
      n.AddNote(60); // fügt c1 hinzu
}
```

for `for` *variable* **=** *value* `to` *value* `[step` *value*`]`
block

Die Variable geht von dem ersten Wert nach oben oder nach unten zum letzten Wert in Schrittwerten. Einen Schritt vor der letzten Wert bleibt sie stehen.

Dieses Beispiel:

```
for  x=1 to note . NoteCount {
      ...
}
```

arbeitet richtig.

assignment
(Anweisung)

variable = expression ;

z. B.

```
value=value+1;
```

oder

variable. variable = expression ;

z. B.

```
Question.CurrentAnswer=True;
```

method call
(Aufrufmethode)

variable . identifier **(***comma-separated exprssions* **);**
Variable . Identifizierungsmerkmal (mit Komma getrennte Ausdrücke) ;

z. B.

```
thisbar.AddText(0,"Jonathan Finn","Composer");
```

self method call
(Selbstaufrufmethode)

identifier **(***comma-separated expressions* **);**

Ruft eine Methode in diesen Plug-in auf, z. B.

```
CheckIntervals();
```

Expressions (Ausdrücke)

Hier sind die Operators, Buchstaben, die sie in ihren Ausdrücken verwenden dürfen.

self
(selbst)

das ist ein Passwort, das auf den Plug-in, der die Methode hat, hinweist. Sie können selbst zu den anderen Methoden übergehen:

z. B. `other.Introduce(Self);`

Identifier
(Identifizierungsmerkmal)

das ist der Name einer Variable oder Methode (Buchstaben, Zahlen oder Unterstrich, und es soll nicht mit einer Zahl anfangen), sie können den Identifizierungsmerkmal mit dem Zeichen @ (wieder ungefähr) einleiten, um Indirektion zu beschaffen; das Identifizierungsmerkmal wird dann zu einer Stringvariable, deren Wert als der Name einer Variable oder Methode gebraucht wird.

member variable
(Teilvariable)

variable . variable

Das greift eine Variable in einem anderen Objekt zu

integer
(ganze Zahl)

z. B. 1, 100, -1

floating point number
(Bruchzahl)

z. B. 1.5, 3.15, -1.8

string

ein Text in Anführungszeichen, z. B. `"some text"`.
Für Strings, die als ein Teil der Partitur bei Sibelius dienen (z. B, der Inhalt einiger Textobjekte), gibt es eine kleine, aber nützliche Formatierungssprache, um zu bestimmen, wie der Text erscheinen soll. Diese 'entworfenen Strings' enthalten Befehle, die den Textstil leiten. Alle Befehle starten und enden mit einem Backslash ("\"). Die ganze Liste der verfügbaren Befehlstile ist die folgende:

`\n\`	Neue Zeile
`\B\`	Fett an
`\b\`	Fett aus
`\I\`	Kursiv an
`\i\`	Kursiv aus
`\U\`	Unterstichen an
`\u\`	Unterstrichen aus
`\fArial\`	Die Schrift wechselt zu Arial (z. B)
`\f_\`	Die Schrift wechselt zur Standardschrift des Textstils
`\s123\`	Die Größe wechselt zu 123 (Einheiten sind 1/32stel eines Leezeichens und nicht Punkte)

(Daraus folgt, dass die Backslashs selbst durch dieses Zeichen \\ dargestellt werden, um Konflikte mit den oberen Befehlen zu vermeiden.)

not
(nein)

not *Ausdruck*

Der Ausdruck verneint logischerweise, z. B.

```
not (x=0)
```

and
(und)

Ausdruck **and** *Ausdruck*

Logischerweise hinzu, z. B.

```
FoxFound and BadgerFound
```
(FuchsFinden und DachsFinden)

or
(oder)

Ausdruck **or** *Ausdruck*

Logischerweise oder, z. B.

```
FoxFound or BadgerFound
```
(FuchsFinden oder DachsFinden)

equality
(Gleichheit)

Ausdruck **=** *Ausdruck*

Gleichheit testen, z. B.

```
Name='Clock'
```

subtract
(subtrahieren)

Ausdruck **-** *Ausdruck*

ganze Zahlen subtrahieren, z. B.

```
12 - 1
```

add
(addieren)

Ausdruck **+** *Ausdruck*

ganze Zahlen addieren, z. B.

```
12 + 1
```

minus
(minus)

- Ausdruck

Ganzzahlinversion, z. B.

```
-1
```

concatenation
(Kette)

Ausdruck **&** *Ausdruck*

Addiert zwei strings, z. B.

```
Name="Fred" & "Bloggs";  //'Fred Bloggs'
```

Sie können das **+** Zeichen hier nicht verwenden, da es mit Zahlen gebraucht wird und meistens auch erfolgreich. z.B.:

```
x = "2" + "2";  // dasselbe wie x = 4
```

subexpression
(Unterausdruck)

(*Ausdruck***)**

Für gruppierende Ausdrücke und für die Erzwingung des Vorrangs.

```
(4+1)*5
```

method call
(Aufrufmethode)

variable . identifier **(**mit Komma getrennte Ausdrücke**)** ;

z. B.

```
x = monkey.CountBananas();
```
(x = Affe . ZählenBananen ( ) ;)

self method call
(Selbstaufrufmethode)

Identifier **(**mit Komma getrennte Ausdrücke **) ;**

Ruft eine Methode in diesen Plug-in auf, z. B.

```
x = CountBananas();
```

Operators (Die Operatoren)

Die Operatorbegriffe

Sie können beliebige Ausdrücke nach einer **if** oder **while** Anweisung in Klammern setzen, aber es ist typisch für sie solche Begriffe wie **=** und **<** zu beinhalten. Die verfügbaren Begriffe sind sehr einfach:

`a = b`	gleich (für die Zahlen, Texte oder Objekte)
`a < b`	kleiner als (für die Zahlen)
`a > b`	größer als (für die Zahlen)
`c and d`	beide sind richtig
`c or d`	jeder von zweien ist richtig
`not c`	kehrt den Begriff um, z. B. `not(n=4)`
`<=`	kleiner als oder gleich
`>=`	größer als oder gleich
`!=`	nicht gleich sein, ungleich

Merken sie, dass wir das Zeichen **=** benutzen, für die Gleichheit, und nicht **==** wie es in C/C++ und Java zu finden ist.

Arithmetik

`a + b`	addieren
`a - b`	subtrahieren
`a * b`	multiplizieren
`a / b`	dividieren
`a % b`	der Rest
`-a`	verneinen
`(a)`	wird zuerst bewertet

Der normale Vorgang dieser Zahlenaufgabe rechnet aus: `2+3*4` macht 14 und nicht 20, da das `*` Zeichen vor dem + Zeichen berechnet wird. Um das deutlicher zu machen, können sie schreiben `2+(3*4)`. Wenn sie die Zahl 20 als Ergebnis haben wollen, dann müssen sie `(2+3)*4` schreiben.

ManuScript unterstürzt auch die Bruchzahlen, so während in den vorherigen Versionen `3/2` zu 1 abgerundet wurde, jetzt wird es zu 1,5 abgerechnet. Die Umwandlung von Bruchzahlen zu Ganzzahlen wird mit den `RoundUp(`*expr*`)`, `RoundUp(`*expr*`)` und `Round(`*expr*`)` Funktionen erreicht, die in jeder Aussage angewendet werden können.

Objektarten

Hier sind die wichtigen Objektarten und die wichtigen Methoden und Variablen, die sie unterstützen:

Beliebiges Objekt

Methoden:

`IsObject(`*expression***`)`**
gibt 1 (oder True) *(=richtig)* zurück, wenn *expression* zu einem Objekt oder eher zu einem String oder zu einer Ganzzahl berechnet wird.
(Seien sie vorsichtig, dass sie es nicht mit der Variable **`IsPassage`** der ausgewählten Objekte verwechseln!)

`Length(`*expression***`)`** *(=Länge)*
gibt die charakteristische Zahl in den Wert des Ausdrucks (expression)

`Substring(`*expression,start,***`[`***length***`])`** *(Teilzeichenfolge)*
das gibt die Teilzeichenfolge des Ausdrucks zurück, die von der angegebenen Anfangsposition (Nullpunkteinstellung) startet und geht bis zur Endposition des Ausdrucks,
z. B. aus **`Substring("Potato",2)`** wird **`"tato"`** ausgegeben. Wenn wir den optionalen *length*parameter verwenden, dann gibt **`Substring`** eine Teilzeichenfolge des Ausdrucks zurück, die von der angegebenen Startposition (Nullpunkteinstellung) der angegebenen Länge (lenght) startet,
z. B. aus **`Substring("Potato",2,2)`** wird **`"ta"`** ausgegeben.

`CharAt(`*expression,position***`)`** *(Char = Charakter = das Schriftzeichen, Buchstabe)*
gibt das angegebene Schriftzeichen von *expression* an die Position zurück (ab Null zählend), z. B. bei **`CharAt("Potato",3)`** wird **`"a"`** ausgegeben.

`Trace(`*expression***`)`**
sendet einen Teil des Textes, damit es im Abfolgefenster des Plug-ins angezeigt wird,
z. B. **`Trace("Here`s a trace");`**

`User(`*expression***`)`** *(Benutzer)*
das zeigt einen Teil des Textes in einem Dialogfenster;
z. B. **`User("Here`s a massage!")`**
Abgelehnt; anstatt dieser Methode verwenden sie **`Sibelius.MessageBox(`***expression***`)`**

`CreateArray()` *(Datenfeld erstellen)*
bringt ein neues Datenfeldobjekt zurück.

`CreateHash()` *(Hash-Tabelle erstellen)*
bringt ein neues Hash-Tabellenobjekt zurück.

`AddToPluginsMenu (`*"Menütext", "Funktionsname"***`)`**
fügt einen neuen Menübegriff in das Menü **Plug-ins** ein. Wenn der Menübegriff gewählt ist, dann wird die angegebene Funktion aufgerufen. Das ist normalerweise nur bei Plug-ins möglich.

`RoundUp(`*expression***`)`**
gibt die nächste Ganzzahl zurück, die größer ist als der Wert des Ausdrucks (expression),
z. B. bei **`RoundUp(1.5)`** wird **`"2"`** ausgegeben.

`RoundDown(`*expression***`)`**
gibt die nächste Ganzzahl zurück, die kleiner ist als der Wert des Ausdrucks (expression),
z. B. bei **`RoundDown(1.5)`** wird **`"1"`** ausgegeben.

`Round(`*expression***`)`**

gibt die Ganzzahl zurück, der dem Wert des Ausdrucks am nächsten steht,
z. B. bei **`Round(1.5)`** wird **`"2"`** und bei **`Round(1.3)`** wird **`"1"`** ausgegeben.

Systemnotenzeile, Notenzeile, Selektion, Takt und alle Taktobjekte

sind abgeleitete Objekte.

Variablen:

`Type` Ein String gibt den Namenstyp eines Objekts an. Die Strings für die ersten vier Arten, die oben genannt wurden, sind "Systemnotenzeilen", "Notenzeilen", "MusicSelectionListe", und "Takt". Merken sie, dass diese Variable ein Teil (oder Mitglied) aller Objekte ist, die in den Takten vorkommen.

`IsALine` Diese Variable wird angewendet, wenn das Objekt ein Linienobjekt ist. (Merken sie, dass es sich um eine Variable handelt und nicht eine Methode, im Gegensatz zur **`IsObject()`**-Methode, die für alle Objekte gilt.)

Sibeliusobjekt

Es gibt eine schon vorher definierende Variable, die das Sibeliusprogramm repräsentiert. Sie können Sibelius benutzen, um Partituren zu öffnen, zu schließen, um Dialoge anzuzeigen oder (am häufigsten) um die aktuellen geöffneten Partiturobjekte zu bekommen.

Methoden:

`MessageBox(`*string***`)`**
zeigt ein Dialogfenster mit dem String und der **OK** Schaltfläche.

`YesNoMessageBox(`*string***`)`**
zeigt ein Dialogfenster mit **Yes** und **No** Schaltflächen. Das kommt nur infrage, wenn **Yes** gewählt werden kann, ansonsten macht es keinen Sinn.

`ShowDialog (`*Scriptname, Objekt***`)`**
zeigt einen Dialog von einer Dialogbeschreibung und sendet Nachrichten und Werte zu dem angegebenen Objekt. Der Wert ist **`True`** (1) oder **`False`** (0). Das hängt davon ab, auf welche Schaltfläche sie klicken, um den Dialog zu beenden (kennzeichnend sind **OK** oder **Abbrechen**). *(true = richtig, false = falsch)*

`Debug(`*string***`)`**
sendet eine Nachricht zum trace windows.

`New()`
Eine neue Partitur wird erstellt und angezeigt. (Dabei wird dasselbe Manuskriptpapier verwendet, das zuletzt angewendet wurde)
gibt das Partiturobjekt entsprechend der neuen Partitur zurück

`Play()`
spielt die aktuelle Partitur vor

`Stop()`
beendet das Vorspiel der aktuellen Partitur

`Print()`
druckt die aktuelle Partitur in den Standardeinstellungen des Druckers

`Close()`
schließt die aktuelle Partitur

`Close(`*show dialogs***`)`**
schließt die aktuelle Partitur; falls der mitgelieferte Bitschalter richtig ist, werden sie gewarnt, die aktive Partitur zu speichern, und falls er falsch ist, dann erscheint auch keine Warnung (und die Partituren werden auch nicht gespeichert).

`Open(`*Dateiname***`)`**
öffnet und zeigt die angegebene Datei an. Der Dateinamen muss mit Erweiterung angegeben werden, z. B. 'Song. sib'.

`SelectFolder()`
erlaubt dem Benutzer, einen Ordner auszuwählen und ein Ordnerobjekt anzulegen

`SelectFolder(`*Titel***`)`**
wie oben, aber mit Angabe eines Ordnernamens (Titel)

`GetFolder(`*Dateipfad***`)`**
legt ein neues Ordnerobjekt an, entsprechend des angegebenen Dateipfades,
z. B. **`folder=Sibelius.GetFolder("c:\\temp");`**

`GetFile(`*Dateipfad***`)`**
legt ein neues Dateiobjekt an, entsprechend des angegebenen Dateipfades,
z. B. **`file= Sibelius.GetFile("c:\\temp\\textfile.txt");`**

`SelectFileToSave(`*Titel, Datei,initial_dir,Standardnamenerweiterung,Standardtyp, Standardbeschreibung***`)`**
zeigt einen Dialog, der den Benutzer veranlasst, eine Datei auszuwählen, um sie zu speichern. Alle Parameter sind optional. Die Methode gibt ein Dateiobjekt entsprechend der Auswahl aus.

Mögliche Dateiarten und Dateiendungen:

Beschreibung	**Type**	**Dateiendung**
EMF Graphics	**`"EMF"`**	.emf
Windows Bitmap	**`"BMP"`**	.bmp
Macintosh PICT Bitmap	**`"PICT"`**	.pict
Sibelius Datei	**`"SIBE"`**	.sib
MIDI Datei	**`"Midi"`**	.mid
Sibelius Stile Datei	**`"SIBS"`**	.lib
PhotoScore Datei	**`"SCMS"`**	.opt
Web Page *(Webseite)*	**`"TEXT"`**	.html
TIFF Graphics	**`"TIFF"`**	.tif

`SelectFileToOpen(`*Titel, Datei, initial_dir,Standardnamenerweiterung,Standardtyp, Standardbeschreibung***`)`**
zeigt einen Dialog, der den Benutzer veranlasst, eine Datei auszuwählen, um sie zu öffnen. Alle Parameter sind optional. Die Methode gibt ein Dateiobjekt aus, das die Auswahl beschreibt, z. B.
`file=Sibelius.SelectFileToOpen("Save Score","*.sib","c:\\my documents","sib","SIBE","Sibelius File");`

`CreateProgressDialog(`*Titel,Mindestwert,Maximalwert***`)`**
erstellt den Fortschrittsdialog, der den Fortgang während einer langen Operation anzeigt

`UpdateProgressDialog(`*Fortschrittsposition,Statusinformation***`)`**
gibt 0 zurück, wenn der Benutzer auf **Abbrechen** oder **Schließen** klickt

`DestroyProgressDialog()` *(destroy = löschen)*
löscht den Fortschrittsdialog

`MakeSafeFileName(`*Dateiname***`)`**

gibt eine gespeicherte Version der Datei des angegebenen Dateinamens aus. Die Funktion entfernt Buchstaben, die auf Windows oder auf Unix illegal sind, und verkürzt den Name auf maximal 31 Buchstaben, damit es auch auf Macintosh sichtbar sein kann.

`RandomNumber()` *(Zufallzahl)*
gibt eine Zufallzahl aus

`RandomSeedTime()` *(random seed = Anfangswert für Zufallgenerator)*
startet die Zufallzahlsequenz neu auf Grund der aktuellen Zeit

`RandomSeed(`*Startzahl***`)`**
startet die Zufallzahlsequenz neu ab der angegebenen Zahl

`ResetStopWatch(`*Zeitgeberzahl***`)`** *(timer = der Zeitgeber, stop watch = die Stoppuhr)*
stellt die angegebene Stoppuhr um

`GetElapsedSeconds(`*Zeitgeberzahl***`)`** *(elapsed = die Ablaufzeit)*
gibt die Zeit in Sekunden aus, wie **`ResetStopWatch`** für die angegebene Stoppuhr aufgerufen wurde

`GetElapsedCentiSeconds(`*Zeitgeberzahl***`)`**
gibt die Zeit in 100stel-Sekunden aus, wie **`ResetStopWatch`** für die angegebene Stoppuhr aufgerufen wurde

`GetElapsedMilliSeconds(`*Zeitgeberzahl***`)`**
gibt die Zeit in 1000stel-Sekunden aus, wie **`ResetStopWatch`** für die angegebene Stoppuhr aufgerufen wurde

Variablen:

`ActiveScore`	ist das aktuelle Partiturobjekt
`ScoreCount`	ist die Zahl der Partituren, die bearbeitet werden
`Playing`	ist True, wenn eine Partitur im Augenblick gespielt wird
`ViewHighlights`	ist True, wenn View > Highlights (Ansicht > Besonderheiten) eingestellt ist (lesen/schreiben)

Partiturobjekte

Eine Partitur enthält eine Systemnotenzeile und eine oder mehrere Notenzeilenobjekte.

`for each in` gibt jede Notenzeile in der Partitur in der Reihenfolge aus (nicht die Systemnotenzeile)

`for each` *type* **`in`** gibt die Objekte in die Partitur in chronologischer Reihenfolge aus, von der oberen Notenzeile bis zur unteren Notenzeile (für gleichartige Objekte) und dann von links nach rechts (wieder wird die Systemnotenzeile nicht einbezogen)

Methoden:

`Score[` *Datenfeldelement***`]`**
Gibt die *n*te Notenzeile an (Notenzeilenindex von 0 an), z. B. Partitur [0]

`NthStaff(`*staff index from 1***`)`** **(***Notenzeilenindex von 1 an***)**
gibt die *n*te Notenzeile aus

`Save(`*Dateiname***`)`**
speichert die Partitur, wobei jede Datei mit demselben Namen überschrieben wird.

`SaveAs(`*Dateiname,Type* **`[`***,use_defaults,Ordnername***`])`**

speichert die Partitur nicht in einem Sibeliusformat, wobei jede Datei mit demselben Namen überschrieben wird. Das optionale Argument *use_defaults* stellt fest, ob es die Standardeinstellungen oder nicht die Standardeinstellungen zu verwenden sind, und der optionale *Ordnername* stellt fest in welchem Ordner die Datei gespeichert werden soll.

Die möglichen Werte für die Typen sind:

"EMF"	emf
"BMP"	windows bitmap
"PICT"	pict format
"Midi"	midi format
"EPSF"	EPS format
"TIFF"	tiff format

CreateInstrument(*Name*, **[***is default name***])**
erstellt ein neues Instrument. Der optionale zweite Parameter *is default name* bestimmt, ob der mitgelieferte Name als ein 'Standard' Instrumentenname angewendet werden soll, der immer in allen Sprachversionen von Sibelius derselbe ist *(englisch)*.

ApplyStyle(Pfad der Stildateien, "*style*", **[**"*style*"**])**
importiert die genannten Stile entsprechende der angegebenen 'Exportstils'-Datei in die aktuelle Partitur. Sie können so viele **"style"** Elemente, wie sie möchten nach der selben Methode importieren.
Stilnamen sind folgende:
House, Text, Symbole, Linien, Notenköpfe, Schlüsseln, Notenzeilenarten, Wörterbuch, Abstandsmaßstab oder alle Stille.

z. B.:
```
score2.ApplySyle("C:\NewStyle.lib","HOUSE","TEXT");
```

Merken sie, dass die Konstante HOUSE aus historischen Gründen nur auf die Optionen der Dialoge 'Gravierungsregeln' und 'Dokument Setup' hinweist, und nicht den ganzen Hausstil. Um den ganzen Hausstil zu importieren, verwenden sie die Konstante ALLESTILE.

RenameTextSyle("*alter Name*","*neuer Name*"**)**
ein Textstil bekommt einen neuen Namen, wird umbenannt

AddBars(*n***)**
fügt n Takte am Ende der Partitur hinzu

InsertBars(*n,b***)**
fügt n Takte nach dem Takt b ein

SetBreakType(*b,t***)**
nach dem Takt b, wird eine Umbruchart t gesetzt, wobei t eine von den Konstanten **NoBreak** (1) *(=Kein Umbruch)*, **SystemBreak** (2) *(=Systemumbruch)* oder **PageBreak** (4) *(=Seitenumbruch)* ist.

ExtractParts([*show_dialogs***],[***Stimmenpfad***])**
extrahiert Stimmen aus der Partitur. Der erste optionale Parameter kann falsch sein, da die Stimmen, ohne einen Optionsdialog anzuzeigen, extrahiert werden. Der zweite optionale Parameter bestimmt einen Ordner, wohin die Stimmen extrahiert werden sollen (das muss ein separater Ordner sein).

SystemCount(*Seitenzahl***)**
die Systemzahl auf der angegebenen Seite (die erste Seite einer Partitur ist immer die Seite 1)

LineStyleIndex(*Name des Linienstils***)**
gibt den Index des Linienstils mit dem angegebenen Namen an, oder -1, wenn ein solcher Linienstil nicht existiert

`TextStyleIndex(`*Name des Textstils***`)`**
gibt den Index des Textstils mit dem angegebenen Namen an, oder -1,wenn ein solcher Textstil nicht existiert

`RemoveAllHighlights()`
entfernt alle Highlights in der Partitur

Variablen:

`FileName`	der Dateiname für die Partitur
`SystemStaff`	das Systemnotenzeile-Objekt für die Partitur
`Selection`	das ausgewählte Objekt für die Partitur, d.h. eine Liste der ausgewählten Objekte
`Redraw`	(*Writeable*) *(=beschreibbar)* Wird als *True* (1) gesetzt, um die Partitur neu zu entwerfen, nachdem dort alle Veränderungen gemacht wurden, es ist *False* (0), wenn die Partitur nicht neu entworfen werden soll
`StaffCount`	die Anzahl der Notenzahlen in einer Partitur zählen
`PageCount`	die Anzahl der Seiten in einer Partitur zählen
`SystemCount`	die Anzahl der Systeme in einer Partitur zählen
`TransposingScore`	ist *True* (1), falls die Partitur transponiert ist
`ShowMultiRests`	ist *True* (1) oder *False* (0), entsprechend der Menüeinstellung im Menü Layouts > Mehrtaktpausen anzeigen
`Title`	Titel der Partitur (entsprechend Menü Datei > Datei-Informationen)
`Composer`	Komponist der Partitur (entspr. Menü Datei > Datei-Informationen)
`Lyricist`	Textdichter der Partitur (entspr. Menü Datei > Datei-Informationen)
`Arranger`	Bearbeiter der Partitur (entspr. Menü Datei > Datei-Informationen)
`Artist`	Interpret der Partitur (entspr. Menü Datei > Datei-Informationen)
`Copyright`	Copyright der Partitur (entspr. Menü Datei > Datei-Informationen)
`ScoreWidth`	Seitenbreite der Partitur, in Millimeter
`ScoreHeight`	Seitenhöhe der Partitur, in Millimeter
`StaffHeight`	Höhe der Notenzeile, in Millimeter (Rastral)

Ordnerobjekte

Auch zu finden in den Methoden des Sibeliusobjekts.

`for each` erstellt Sibeliusdateien im Ordner, wie Dateiobjekte.

`for each` *Type* **`in`** erstellt Dateien des Typs *Type* im Ordner, wo **`Type`** eine Windowserweiterung ist. Nützliche Werte sind SIB (Sibelius Dateien), MID (MIDI Dateien) oder OPT (PhotoScore Dateien), weil sie

alle direkt von Sibelius geöffnet werden können. Die entsprechenden Mac OS Type Dateien werden auf Macintosh auch ausgegeben (so zum Beispiel: **`"for each MID"`** wird alle Dateien, deren Namen mit MID enden, und alle Dateien des ‘ Midi ‘ Typs, ausgeben).

Diese beiden Anweisungen geben Unterordner an.

Methoden:

`FileCount(`Type**`)`**	gibt die Anzahl der Dateien des Typs *Type* im Ordner an, auch hier sind die nützlichen Werte SIB, MID oder OPT

Variablen:

`Name`	Name des Ordners
`FileCount`	benennt die Anzahl der Sibeliusdateien in einem Ordner
`Type`	ein *string*, der den Namen des Objekttyps angibt; **`Folder`** für Ordnerobjekte

Dateiobjekte

Auch möglich, indem **`for each`** auf einem Ordner angewendet wird.

Methoden:

Keine

Variablen:

`Name`	der ganze Pfadname der Datei (keine Erweiterung)
`NameWithExt`	der ganze Pfadname der Datei (mit der Erweiterung)
`NameNoPath`	nur der Dateiname (keine Erweiterung)
`Path`	gibt nur den Dateipfad an (kein Dateiname)
`CreationDateAndTime`	ein *string* gibt das Datum und die Zeit der Erstellung der Datei an
`Type`	ein *string* gibt den Namenstyp des Objekts an; **`File`** für Dateiobjekte

Selektierte Objekte

`for each in` gibt jedes Taktobjekt (d.h. eines Objekt innerhalb eines Taktes) in der Auswahl an

`for each` *Type* **`in`** erstellt jedes Objekt des Typs *Type* in der Auswahl. Merken sie, wenn die Selektion eine Systemselektion ist (d.h. in Sibelius mit einem doppelten Rechteck umrandet), dann werden die Objekte in der Systemnotenzeile in einer Schleife (Loop) ausgegeben.

Methoden:

Keine

Variablen:

`IstPassage`	ist True, wenn die Auswahl eine Passage betrifft (im Gegensatz zu einer mehrfachen Auswahl)

TopStaff die Zahl der oberen Notenzeile einer Passage

BottomStaff die Zahl der unteren Notenzeile einer Passage

FirstBarNumber die erste Taktzahl einer Passage

LastBarNumber **d**ie letzte Taktzahl einer Passage

Notenzeilenobjekte

Notenzeilenobjekte können normale Notenzeilen oder Systemnotenzeilen sein. Die Systemnotenzeile enthält Objekte, die zu allen Notenzeilen gehören, solche wie spezielle Taktlinien und der Text, der einen Systemtextstil verwendet.
Eine Notenzeile enthält Taktobjekte.

for each in gibt jedes Objekt in der Notenzeile aus
for each *Type* **in** gibt jedes Element des Typs *Type* in der Notenzeile in chronologischer Anordnung an (d.h. nach der rhythmischen Position in jedem Takt)

Methoden:

Staff[*Datenfeldelement***]**
gibt den *n*ten Takt aus (wird von 0 gezählt) z. B. **Staff[0]**

NthBar(*n***)**
Gibt den *n*ten Takt in die Notenzeile aus, wird von 1 gezählt

AddNote(*Position,klingendeTonhöhe,Dauer,***[***angebunden***],[***Stimme***],[***diatonische Tonfolge***])**
fügt eine Note in die Notenzeile hinzu, die an der Stelle der schon existierenden Note/Akkord/Pause hinzugefügt wird (in diesem Fall wird die Dauer nicht beachtet); sonst wird eine neue Note/Akkord/Pause erstellt. Falls es notwendig ist, wird ein neuer Takt am Ende der Notenzeile hinzugefügt. Die Position ist ein Viertel (1/256) vom Anfang der Partitur entfernt. Der optionale Bindungsparameter soll richtig sein, wenn sie möchten, dass die Noten gebunden werden. Die 1. Stimme wird vorausgesetzt, es sei denn, dass der optionale Stimmenparameter (mit einem Wert von 1, 2, 3 oder 4) bestimmt wurde. Sie können noch die diatonische Tonfolge festsetzen, d. h. die Zahl des 'Notennamens' zu der diese Note entspricht, 7 per Oktave (35 = mittlere C, 36=D, 37 = E usw.).

AddLine(*Position,Dauer,IndexdesLinienstils***[***dx***],[***dy***],[***dieZahlderStimmen***],[***versteckt***])**
fügt eine Linie in die Notenzeile ein (bitte sehen sie sich die Dokumentation über die **Taktobjekte** weiter unten an)

AddClef(*Position, Schlüsselstil***)**
fügt einen Schlüssel an der angegebenen Position in die Notenzeile ein

CurrentKeySignature(*bar_number***)**
gibt eine **KeySignature** *(=Tonartvorzeichnung)* an der Stelle der angegebenen Taktzahl ein

Variablen:

BarCount Taktzahlen in der Notenzeile

InstumentName Instrumentname der Notenzeile

DefaultInstrumentName der (einzige) Name des Standardinstrumentes, von dem diese Instrumentennotenzeile erstellt wurde. Das gilt für alle Sprachversionen von Sibelius gleich. Der Standardname ist also

immer jener, den wir wählen, wenn wir ein neues Instrument erstellen. Jede Kopie von dieser Notenzeile ist mit dem gleichen Instrumentennamen definiert. Um ein Instrument von solch einem Namen zu erstellen, geben sie 'True' als zweites Argument hinzu **`Score.CreateInstrument`** .

`NumStavesInSameInstrument` diese Abzahl der Notenzeilen gehört zu dem Standard-instrument, von dem aus diese Notenzeile erstellt worden ist.

`IsSystemStaff` ob es *True* oder *False* ist hängt davon ab, ob diese Notenzeile eine Systemnotenzeile ist oder nicht

`InitialClefStyle` der Typ des Anfangsschlüssels auf der Notenzeile

Systemnotenzeile

Sie können für die normalen Notenzeilenobjekte der Systemnotenzeile die benannten Methoden und Variablen verwenden und noch das Folgende:

Methoden:

`CurrentTimeSignature(`*bar_number***`)`**
gibt eine **Taktart**, die an der Taktzahl erstellt wird

Variablen:
Keine

Taktobjekte

Ein Takt enthält Taktobjekte des Objekts.

`for each in` erstellt Taktobjekte im Takt
`for each` *Type* **`in`** erstellt bestimmte Arten von Taktobjekte (Type) im Takt

Methoden:

`Bar[`*Datenfeldelement***`]`**
gibt das *n*te Element in den Takt (von 0 zählen) aus, z. B. Takt [0]

`NthBarObject(`*n***`)`**
gibt das *n*te Objekt in den Takt aus, von 0 zählen

`Respace()`
die Noten werden in den Takt zurückplatziert

`AddText(`*Position,Text,Stil***`)`**
der Text wird an der angegebenen Position und im benannten Textstil eingefügt
Ein Notenzeilentextstil muss für eine normale Notenzeile verwendet werden, und ein Systemtextstil für eine Systemnotenzeile. Stile können durch die Werte, die in **Globals.txt** zu finden sind oder durch den Namen, definiert werden.

`AddNote(`*Positon,klingendeTonhöhe,Dauer***`,[`***angebunden***`],[`***Stimme***`],[`***diatonischeTonhöhe***`])`**
fügt eine Note in die Notenzeile ein, die an der Stelle der schon existierenden Note/Akkord/Pause eingefügt wird (in diesem Fall wird die Dauer nicht beachtet); sonst wird eine neue Note/Akkord/Pause erstellt. Falls es notwendig ist, wird ein neuer Takt am Ende der Notenzeile hinzugefügt. Die Position ist ein Viertel (1/256) vom Anfang der Partitur entfernt. Der optionale Bindungsparameter soll richtig sein, wenn sie möchten, dass die Noten gebunden werden. Die 1. Stimme wird vorausgesetzt, es sei denn, dass der optionale Stimmenparameter (mit einem Wert von 1, 2, 3 oder 4) bestimmt wurde. Sie

können noch die diatonische Tonfolge festsetzen, d. h. die Zahl des 'Notennamens' der dieser Note entspricht, 7 per Oktave (35 = mittlere C, 36=D, 37 = E usw.).

AddTimeSignature(*top, bottom, allow_cautionary,rewrite_music***)**

gibt einen Fehler*string* zurück (welcher leer sein wird, wenn kein Fehler vorliegt), welcher dem Benutzer angezeigt wird, wenn er nicht leer ist. Die ersten zwei Parameter sind der Anfang und das Ende der neuen Taktarten. Der dritte sagt Sibelius, dass wenn die warnenden Taktarten angezeigt werden, dann sollen sie von dieser Taktart ausgehen. Wenn *rewrite_music* True ist, dann werden alle Takte nach der eingesetzten Taktart neu geschrieben.

AddLyric(*Position, Dauer, Text,* **[***Typ der Silbe, die Zahl der Noten***])**

diese Methode fügt ein Liedtext in den Taktein, die Position ist ein Viertel (1/256) vom Anfang des Taktes entfernt, und die Dauer ist ein Viertel

AddLyric(*Position, Dauer, Text, Silbentyp***)**

das Gleiche wie oben, außer dass sie bestimmen können, ob der Liedtext am Ende eines Wortes (der Wert ist "1", und ist Normalwert) oder am Anfang eines Wortes (der Wert ist "0") sein soll

AddLyric(*Position, Dauer, Text, Silbentyp, Zahl der Noten***)**

das Gleiche wie oben, außer dass sie bestimmen können, dass der Liedtext eine Anzahl von Noten einnimmt. Der angenommene Wert ist 1, aber sie können hier auch verschiedenartige Werte eingeben.

AddGuitarFrame(*Position, Akkordname, Name des Notenzeilentyps***)**

dies fügt ein Gitarren-Akkordraster für den angegebenen Akkord in den Takt hinzu. Der Notenzeilentyp soll auf eine vorhandene Tabulatur des Notenzeilentyps (wie Gitarre) hinweisen. Die Position ist ein Viertel (1/256) vom Anfang des Taktes entfernt.

AddLine(*Position, Dauer, Linienstil-Index,* **[***dx, dy, Stimmenzahl, versteckt***])**

fügt eine Linie in den Takt ein. Der Linienstil soll auf einen Linienstils hinweisen, der in der Partitur definiert ist. Bitte sehen sie sich am Ende dieses Dokumentes die Liste von Stil-Indexen an, die in jeder Sibeliuspartitur definiert ist. Um zusätzliche Stile hinzuzufügen, finden sie den Linienstilindex, indem sie die **LineStilIndex** -Methode im Partiturobjekt verwenden.

AddKeySignatureFromText(*Position, Tonartname, Durtonart***)**

fügt eine Tonartvorzeichnung in den Takt ein. Die Tonartvorzeichnung wird durch den Textnamen bestimmt, z. B. "Cb" oder "C#". Der dritte Parameter ist ein boolescher Bittschalter, der angibt, ob dieTonart Dur (oder Moll) ist.

AddKeySignature(*Position, die Zahl der Kreuze, Durtonart***)**

fügt eine Tonartvorzeichnung in den Takt ein. Die Tonartvorzeichnung wird durch die Zahl der Kreuze (+1 bis +7), Been (-1 bis -7), ohne Versetzungszeichen (0)oder atonal (-8) bestimmt. Der dritte Parameter ist ein boolescher Bittschalter, der angibt, ob die Tonart Dur (oder Moll) ist.

AddClef(*Position, Schlüsselstil***)**

fügt einen bestimmten Schlüssel an angegebener Position in den Takt ein

Variablen:

BarObjectsCount	die Anzahl der Objekte im Takt
Length	die rhythmische Dauer
ParentStaff	die Notenzeile enthält diesen Takt
BarNumber	die Taktzahl dieses Taktes

Objekte des Taktobjekts

Taktobjekte erhalten Schlüssel, Linien, Note/Akkord/Pause-Objekte und Textobjekte. Die Variablen **`Position`** und **`Type`** werden für alle bestimmten Typen eines Taktobjekts verwendet; sie sind hier zusammen aufgelistet, um nicht jeden Typ zu trennen. (Für objektorientierte Programme werden die Typen wie Note/Akkord/Pause, Schlüssel usw. von den abstrakten Taktobjektarten abgeleitet.)

Methoden:
Keine

Variablen:

`Position`	**r**hythmische Position des Objekts im Takt
`Type`	ein *string*, das die Objektart beschreibt, z. B. 'Note/Akkord/Pause', 'Schlüssel'. Das ist nützlich, wenn sie einen bestimmten Typ des Objekts in einem Takt suchen. Bitte sehen sie sich den 'Taktobjekttypen'-Abschnitt am Ende dieses Dokumentes für die möglichen Werte an.
`VoiceNumber`	ist 0, wenn die Betrachtungseinheit mehreren Stimmen gehört (viele Betrachtungseinheiten gehören mehr als einer Stimme) und 1 bis 4 sind für die Betrachtungseinheiten, die den Stimmen von 1 bis 4 angehören
`ParentBar`	der Takt enthält dieses Taktobjekt
`Dx`	der horizontale Grafikabstand des Objekts von der Position aus, der bei dem Positionsfeld impliziert ist (Lesen/Schreiben)
`Dy`	der vertikale Grafikabstand des Objekts von der Mitte der Notenzeile (das Positive geht aufwärts) (Lesen/Schreiben)

Linien

Alles, was sie von dem Dialog **Erstellen > Linie** erstellen können, ist ein Linienobjekt, z. B. Crescendolinie, Diminuendolinie usw. Diese Objekte sind von einem Taktobjekt abgeleitet.

Methoden:
Keine

Variablen:

`EndBarNumber`	die Taktzahl, wo die Linie endet
`EndPosition`	die Position innerhalb des letzten Taktes, wo die Linie endet
`Duration`	die absolute Dauer der Linie (in 1/256stel Viertel)
`Style`	ein Zahlenindex des Linienstils, der mit dieser Linie verbunden ist
`StyleAsText`	der Name des Linienstils, der mit diese Linie verbunden ist

Tonartvorzeichnung

Abgeleitet von einem Taktobjekt.

Methoden:
Keine

Variablen:

`Sharps`	die Anzahl der Kreuze (positive) oder der Bees in dieser Tonartvorzeichnung
`AsText`	der Name der Tonartvorzeichnung wie ein *string*
`Major`	True ist, wenn diese Tonartvorzeichnung eine Durtonart ist

Taktart

Von einem Taktobjekt abgeleitet.

Methoden:
Keine

Variablen:

`Numerator`	die obere Zahl der Taktart
`Denominator`	die untere Zahl der Taktart

Schlüssel

Von einem Taktobjekt abgeleitet.

Methoden:
Keine

Variablen:

`ClefStyle`	ein Name, der den Schlüsseltyp bezeichnet. Das kann der Methode **`Bar.AddClef`** übergegeben werden, um ein Schlüssel von diesem Stil zu erstellen.

Text

Von einem Taktobjekt abgeleitet. Für den Systemtext (d. h. der Text, der der Systemnotenzeile angehört, bereits bekannt durch **`for each`** bezogen auf das Systemnotenzeilenobjekt) ist die Systemtexteinheit und nicht die Texteinheit des Typs des Textobjekts gemeint.

Mehtoden:
Keine

Variablen:

`Text`	der Text als ein *string* - der Wert kann verändert werden
`Style`	die Nummer des Textstils
`StyleAsText`	der Name des Textstils

Liedtexteinheit

Von einem Objekt abgeleitet

Methoden:
Keine

Variablen:

`Text`	der Text als ein *string* - der Wert kann verändert werden
`Style`	die Nummer des Textstils
`StyleAsText`	der Name des Textstils
`SyllableTyp`	eine ganze Zahl, die anzeigt, ob der Liedtext am Ende (1) oder in der Mitte (0) eines Wortes steht
`NumNotes`	gibt die Anzahl der Noten an, die diesem Liedtext angehören (eine Liedtextlinie wird unter den Noten in der Partitur gezeichnet)

SpezielleTaktlinien

Von einem Taktobjekt abgeleitet. Diese können nur in den Systemnotenzeilen gefunden werden.

Methoden:
Keine.

Variablen:

`BarlineType`	Der Name des speziellen Taktlinientyps: Wiederholungs-Anfang, Wiederholungs-Ende, Gestrichelt, Doppelstrich, Schlussstrich, Unsichtbar, Normal, Zwischen Notenzeilen, Häkchen, Kurz; sie werden alle als globale Konstante definiert (siehe **Globale Konstanten** unten).

N-tole (Tuplet)

Von einem Taktobjekt abgeleitet

Methoden:
Keine

Variablen:

`PlayedDuration`	die richtige rhythmische Dauer der N-tole, z. B. Triole mit Viertelnoten wird in der Zeit von einer Halben Note gespielt.
`Text`	diesewr Text erscheint über der N-tole
`Unit`	die Einheit, die für die N-tole verwendet wird, z.B 256 für eine Triole von Viertelnoten
`Left`	die linke Seite der N-tole , z. B. 3 in 3:2
`Right`	die rechte Seite der N-tole, z. B. 2 in 3:2

Note/Akkord/Pause (NoteRest)

Von einem Taktobjekt abgeleitet. Eine Note/Akkord/Pause enthält Notenobjekte.
`for each` gibt die Noten in die Note/Akkord/Pause zurück.

Methoden:

`NoteRest [`*Datenfeldelement***`]`**

gibt die *n*te Note in den Akkord aus (von 0 zählen), z.B. **`NoteRest[0]`** gibt die höchste Note zurück

`AddNote(`*pitch,* **`[`***tied***`])`**

fügt eine Note mit der angegebenen MIDI pitch (60 = mittlere C), z.B. um einen Akkord zu erstellen. Der optionale zweite Parameter bestimmt, ob diese Note verbunden oder nicht verbunden sein soll (*True* oder *False*).

`RemoveNote(`*Note***`)`**

entfernt das angegebene Notenobjekt

`FlipStem()`

Den Hals dieser Note/Akkord/Pause umdrehen - das funktioniert wie ein Umschalter.

Variablen:

`NoteCount`	die Anzahl der Noten im Akkord
`Duration`	die Dauer der Note/Akkord/Pause
`Lowest`	die Tonhöhe der höchsten Note im Akkord
`Highest`	die Tonhöhe der niedrigsten Note im Akkord
`Beam`	nimmt die Werte **`StartBeam`** *(=Balkenanfang)* **`ContinueBeam`** *(=Balkenmitte)*, **`NoBeam`** *(=kein Balken)* und **`SingleBeam`** *(=Einzelbalken)* auf (siehe **Globale Konstanten** unten) Diese entsprechen den Tasten 4, 5, 6 und 8 auf dem dritten Keypadlayout.
`GraceNote`	ist *True*, wenn es eine Verzierungsnote ist
`StemFlipped`	ist *True*, wenn der Notenhals umgedreht ist

Notenobjekte

Sind nur in Note/Akkord/Pause zu finden. Entspricht den individuellen Notenköpfen.

Methoden:

Keine

Variablen:

`Pitch`	MIDI pitch der Note (in Halbtönen, 60 = mittlere C)
`DiatonicPitch`	die diatonische Tonfolge der Noten, d. h, die Nummer des 'Notennamens' , dem diese Note entspricht, 7 per Oktave (35 = mittlere C, 36 = D, 37 = E usw.)
`WrittenPitch`	der geschriebene MIDI Pitch der Note, der die Transposition in Betracht zieht, wenn **`Score.TransposingScore=True`** (60 = mittlere C) ist
`Name`	die Tonhöhe der Note als ein *string*
`WrittenName`	der geschriebene Pitch der Note als ein *string* (die Transposition in Betracht ziehen!)

`Accidental`	das Versetzungszeichen (für das die globalen Konstanten wie Kreuze, Been usw. definiert sind; sehen sie sich den Abschnitt ‘globale Konstanten’ unten an)
`WrittenAccidental`	das Versetzungszeichen, das die Transposition in Betracht zieht
`Tied`	ist *True*, wenn die Note mit der nächsten Note verbunden ist

Globale Konstanten

Sie sind nützliche Variablen, die sich in der Datei **Global.txt** im **Plugins** Ordner innerhalb ihres Sibeliusordners befinden. Sie können zu dieser Datei etwas hinzufügen, wenn sie es möchten. Diese Variablen sind von jedem Plug-in greifbar. Sie heißen 'Konstanten', weil sie nicht in der Lage sind, sich zu verändern.

Viele von diesen Konstanten sind Namen von Notenwerten, die sie verwenden können, um eine Position in einem Takt leicht zu bestimmen. So, anstatt 320 zu schreiben, können sie Viertel + Sechzehntel (Quarter + Sixteenth) oder das Gleiche Achtel + Sechzehntel (Crotchet + Semiquaver) verwenden.

Richtige Werte

True *(Richtig)*	1
False *(Falsch)*	0

Abmessungen

Space (Raum)	32
StaffHeight *(Höhe der Notenzeile)*	128

Position und Dauer

Long *(Longa)*	4096
Breve *(Brevis)*	2048
DottedBreve *(Punktierte Brevis)*	3072
Whole *(Ganze)*	1024
Semibreve (Semibrevis)	1024
DottedWhole *(Punktierte Ganze)*	1536
DottedSemibreve *(Punktierte Semibrevis)*	1536
Half *(Halbe)*	512
Minim *(Minima)*	512
DottedHalf *(Punktierte Halbe)*	768
DottedMinim *(Punktierte Minima)*	768
Quarter *(Viertel)*	256
Crotchet *(Semiminima)*	256
DottedQuarter *(Punktierte Viertel)*	384
DottedCrotchet *(Punktierte Viertel)*	384
Eighth *(Achtel)*	128
Quaver *(Fusa)*	128
DottedEighth *(Punktierte Achtel)*	192
DottedQuaver *(Punktierte Achtel)*	192
Sixteenth *(Sechzehntel)*	64
SemiQuaver *(Semifusa)*	64
DottedSixteenth *(Punktierte Sechzehntel)*	96
DottedSemiquaver *(Punktierte semifusa)*	96
ThirtySecond *(Zweiunddreißigstel)*	32
Demisemiquaver *(Demisemifusa)*	32
DottedThirtySecond *(Punktierte Zweiunddreißigstel)*	48
DottedDemisemiquaver *(Punktierte Demisemifusa)*	48
SixtyFourth *(Vierundsechzigstel)*	16
Hemidemisemiquaver *(Hemidemisemifusa)*	16
DottedSixtyFourth *(Punktierte Vierundsechzigstel)*	24
DottedHemidemisemiquaver *(Punktierte. Hemidemisemifusa)*	24
OneHundredTwentyEighth *(Hundertachtundzwanzigstel)*	8
Semihemidemisemiquaver *(Semihemidemisemiquaverfusa)*	8
DottedOneHundredTwentyEighth *(Punktierte Hundertachtundzwanzigstel)*	12
DottedSemihemidemisemiquaver *(Punktierte Semihemidemisemiquaverfusa)*	12

Stilnamen

Für die **`ApplyStyle()`**–Methode der Partiturobjekte. Anstatt der groß geschriebenen Strings in Anführungszeichen, können sie die entsprechenden Variablen gemischt wie der obere und der untere Fall verwenden. Merken sie wieder, dass die Konstante HOUSE nur auf die Optionen der Stilvorlagen und Dokumenteinrichtung hinweist; um den ganzen Stil anzuwenden, müssen sie die Konstante AllStyles gebrauchen.

House *(Haus)*	"HOUSE"
Text *(Text)*	"TEXT"
Symbols *(Symbole)*	"SYMBOLS"
Lines *(Linien)*	"LINES"
Noteheads *(Notenköpfe)*	"NOTEHEADS"
Clefs *(Schlüssel)*	"CLEFS"
StaffTypes *(Notenzeilentypen)*	"STAFFTYPES"
Dictionary *(Wörterbuch)*	"DICTIONARY"
SpacingRule *(Abstandsmaße)*	"SPACINGRULE"
AllStyles *(Alle Stile)*	"ALLSTYLES"

Indexe des Linienstils

Für die **`AddLine()`**–Methode der Taktobjekte. Verwenden sie die globale Variable oder Zahl, um auf die eingebauten Linienstile des Dokuments hinzuweisen.

HighlightLineStyle *(HighlightLinienStil)*	"-1"
OctavaPlus8LineStyle	"0"
OctavaMinus8LineStyle	"1"
OctavaPlus15LineStyle	"2"
OctavaMinus15LineStyle	"3"
PedalLineStyle	"4"
FirstRepeatLineStyle *(ErsteWiederholungLinienStil)*	"5"
SecondRepeatLineStyle *(ZweiteWiederholungLinienStil)*	"6"
OpenRepeatLineStyle *(OffeneWiederholungLinienStil)*	"7"
ClosedRepeatLineStyle *(GeschlosseneWiederholungLinienStil)*	"8"
TrillLineStyle *(TrillerLinienStil)*	"9"
WavyGlissLineStyle *(GewellteGlissandoLinienStil)*	"10"
UpSlurLineStyle *(ObenBindebogenLinienStil)*	"11"
DownSlurLineStyle *(UntenBindebogenLinienStil)*	"12"
TieLineStyle *(BindebogenLinienStil)*	"13"
CrescendoLineStyle *(CrescendoLinienStil)*	"14"
DiminuendoLineStyle *(DiminuendoLinienStil)*	"15"
ArpeggioLineStyle *(ArpeggioLinienStil)*	"16"
BendLineStyle *(BogenLinienStil)*	"18"
LineLineStyle *(LinieLinienStil)*	"19"
UpArpeggioLineStyle *(ObenArpeggioLinienStil)*	"21"
DownArpeggioLineStyle *(UntenArpeggioLinienStil)*	"22"
StraightGlissLineStyle *(GeradeGlissLinienStil)*	"23"
BoxLineStyle *(RechteckLinienStil)*	"25"

Namen des Textstils

Für die **`AddText()`**– Methode des Taktobjekts. Sie sollten diese Werte verwenden, um auf die Basistextstile, die immer vorhanden sind, hinzuweisen. Verwenden sie den aktuellen Namen der anderen Textstile als einen *string* in Anführungszeichen.

TitleTextStyle *(TitelTextStil)*	"3"
SubtitleTextStyle *(UntertitelTextStil)*	"4"
DedicationTextStyle *(WidmungTextStil)*	"5"
ComposerTextStyle *(KomponistTextStil)*	"6"
LyricistTextStyle *(TextdichterTextStil)*	"7"
CopyrightTextStyle *(CopyrightTextStil)*	"8"
HeaderTextStyle (KopfzeileTextStil)	"9"
HeaderAfterFirstPageTextStyle *(KopfzeileNachDerErstenSeiteTextStil)*	"10"

FooterTextStyle *(FußzeileTextStil)*	"11"
Footer2TextStyle *(Fußzeile2TextStil)*	"12"
TempoTextStyle *(TempoTextStil)*	"13"
MetronomeMarkTextStyle *(MetronomangabeTextStil)*	"14"
ExpressionTextStyle *(ExpressionTextStil)*	"15"
TechniqueTextStyle *(TechnikTextStil)*	"16"
LyricsTextStyle *(LiedTextStil)*	"17"
LyricsVerse2TextStyle *(Liedtext2StropheStil)*	"18"
ChordSymbolTextStyle *(AkkordsymbolTextStil)*	"19"
PageNumbersTextStyle *(SeitenzahlTextStil)*	"20"
BarNumbersTextStyle *(TaktzahlTextStil)*	"21"
imeSignaturesTextStyle *(TaktartTextStil)*	"22"
InstrumentNamesTextStyle *(InstrumentenameTextStil)*	"24"
TupletsTextStyle *(N-toleTextStil)*	"25"
RehearsalMarksTextStyle *(StudierzeichenTextStil)*	"26"
TimeSignaturesLargeTextStyle *(TaktartbezeichnungenGroßTextStil)*	"27"
TimeSignaturesHugeTextStyle *(TaktartbezeichnungenRiesigTextStil)*	"28"
GuitarFrameFretTextStyle *(AkkordrasterBundTextStil)*	"30"
TablatureNumbersTextStyle *(TabulaturZiffernTextStil)*	"31"
TablatureLettersTextStyle *(TabulaturBuchstabenTextStil)*	"32"
MultirestsTextStyle *(MehrtaktpausenTextStil)*	"33"
InstrumentNameAtTopLeftTextStyle *(InstrumentennamenLinksObenTextStil)*	"34"

Balkenoptionen

Für die Balkenvariable der Note/Akkord/Pause-Objekte.

NoBeam *(Kein Balken)*	"1"
StartBeam *(Balkenanfang)*	"2"
ContinueBeam (Balken verbinden)	"3"
SingleBeam *(Einzelbalken)*	"4"

Umbrüche

Für die **`SetBreakType()`** – Methode der Partiturobjekte.

NoBreak *(Kein Umbruch)*	"1"
SystemBreak *(Systemumbruch)*	"2"
PageBreak *(Seitenumbruch)*	"4"

Versetzungszeichen

Für die Versetzungszeichenvariable der Notenobjekte.

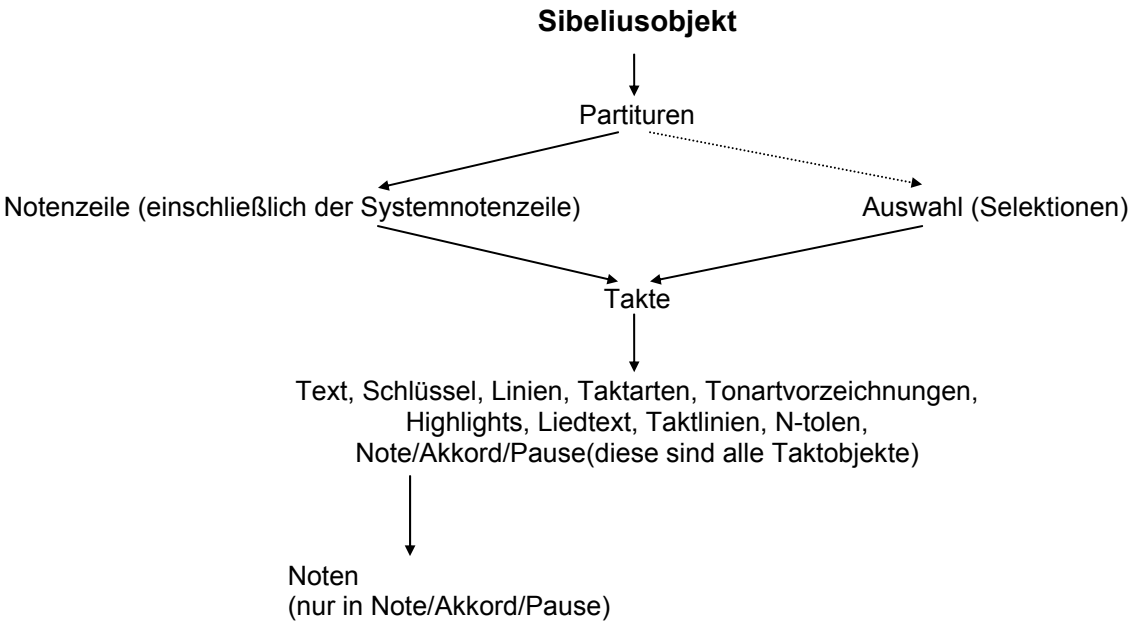
Sharp *(Doppelkreuz)*	2
DoubleSharp *(Kreuz)*	1
Natural *(Auflösungszeichen)*	0
Flat *(Be)*	-1
DoubleFlat *(Doppel Be)*	-2

Objekttypen in einem Takt

Der Feldtyp für die Objekte in einem Takt kann eine von den folgenden Werten zurückgeben:
Schlüssel, Spezielle Taktlinien, Taktart, Tonartvorzeichnung
Linie, Arpeggio -Linie, Crescendolinie, Diminuendolinie, Glissandolinie, Oktavlinie, Pedallinie, Wiederholungslinie, Bindebogen, Triller, Rechteck, N-tole, Ritardando-/Accellerandolinie

Liedtext-Betrachtungseinheit, Text, Akkordraster, Transposition, Studierzeichen, Notenzeilentypänderung
Taktpause, Note/Akkord/Pause, Graphik, Taktlinie

Hierarchie der Objekte



SIBELIUS 2.1 – Plug-Ins

Folgende Plug-ins sind im Lieferumfang von Sibelius Version 2.1 enthalten. Sie befinden sich im Ordner **Plugins** im Sibeliusverzeichnis.

1. AkkordnamenHinzufügen
2. AlleMarkierungenEntfernen
3. AufEineTonhöheSetzen
4. BlechbläserFingersatz
5. BsInKreuzeUmdeuten
6. CrescUndDim
7. DynamikKopieren
8. EinheitlichesLayout
9. Erinnerungsvorzeichen
10. FictaHinzufügen
11. FinaleConverter
12. FinaleDateienKonv
13. HarfenPedalPrüfen
14. KorrekturLesen
15. Krebs
16. KreuzeInBsUmdeuten
17. MIDIsKonvertieren
18. MotivFinden
19. NIFFConverter
20. NotennamenHinzufügen
21. NotenvwerteHalbieren
22. NotenwerteVerdoppeln
23. OrdnerDrucken
24. OrdnerZuGrafikKonv
25. PausenEntfernen
26. PizzicatoPrüfen
27. PunktViertelPausenAuft
28. QuintUndOktavParallelen
29. SchlüsselPrüfen
30. SCOREConverter
31. Sibelius7Converter
32. Sibelius7DateienKonv
33. Spiegeln
34. StilvorlagenFürOrdner
35. StreicherFingersatz
36. TonhöhenVersetzen
37. TonikaDoHinzufügen
38. Tonumfang
39. ViertelTonMidi
40. WiederholungPrüfen
41. ZählzeitenSchreiben
42. ÜberhängendeHaltebögenEntf