HTL - Perg
Höhere Abteilung für Informatik

Diplomarbeit

Objektorientierte Darstellung von Keba TeachTalk Programmen

Projektteam: Niklas Lempradl
Jan Kaltenböck
Moritz Leonhardsberger

Projektbetreuer: OStR Dipl.-Ing. Roland Eggetsberger

In Zusammenarbeit mit Fa. Engel Austria GmbH

Betreuer Herr Hackl Helmut

Bearbeitungszeitraum: 30.06.2023 – 03.04.2024

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der von uns angegebenen Quellen angefertigt zu haben. Alle Stellen, die wörtlich oder sinngemäß aus fremden Quellen direkt oder indirekt übernommen wurden, sind als solche gekennzeichnet.

Perg, 02.04.2024 Unterschrift

Jan Kaltenböck

Perg, 02.04.2024 Unterschrift

Moritz Leonhardsberger

Perg, 02.04.2024 Unterschrift

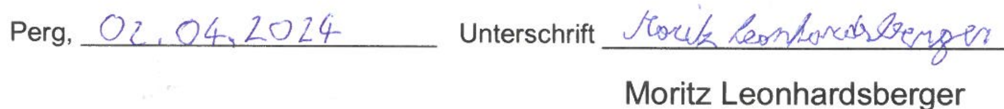
Niklas Lempradl

Gendererklärung

Für die bessere Lesbarkeit werden in dieser Diplomarbeit personenbezogene Bezeichnungen, die sich zugleich auf Frauen und Männer beziehen, generell nur in der Deutschen üblichen maskulinen Form angeführt. Dies soll jedoch keinesfalls eine Geschlechterdiskriminierung oder eine Verletzung des Gleichheitsgrundsatzes zum Ausdruck bringen.

Perg, 02.04.2024 Unterschrift

Jan Kaltenböck



Perg, 02.04.2024 Unterschrift

Moritz Leonhardsberger

Perg, 02.04.2024 Unterschrift

Niklas Lempradl

Danksagung

Wir möchten uns bei jenen Personen bedanken, die uns im Verlauf der Erstellung der Diplomarbeit in jeglicher Hinsicht unterstütz haben.

Besonders möchten wir uns bei unseren Ansprechpartner Helmut Hackl bedanken.
Im Laufe der Entwicklung, bot er seine Hilfe an, um uns bei der Behebung von Fehlern und Problemen zu unterstützen und hatte immer ein offenes Ohr für uns.

Ein weiterer Dank gebührt unserem Betreuungslehrer, Herrn Professor OStR Dipl.-Ing. Roland Eggetsberger, der uns während der Entwicklung und Planung stets unterstützt hat.

Danke auch an unsere Familien und Freunden, die uns während dieser Zeit unterstützt haben.

Kurzfassung

Die Diplomarbeit Objektorientierte Darstellung von Keba Teachtalk Programmen besteht aus einem Parser für Teachtalk Projekte und einem Strukturbaum, der die einzelnen Dateien hierarchisch darstellt. Diese Dateien werden nach Routinen, Variablen, Konstanten, Vererbungen und Typendeklarationen durchsucht. Dafür werden Regular Expressions verwendet, mithilfe derer der Parser den Programmcode in die einzelnen Bestandteile zerlegen kann. Diese Bestandteile werden in C# Klassen gespeichert, mit denen der Strukturbaum aufgebaut wird. Der Strukturbaum ist ähnlich einer verketteten Liste und erstellt aufgrund der Vererbung zwischen teachtalk Dateien eine Hierarchie. Zur visuellen Darstellung des gesamten Strukturbaumes wird eine WPF-App mit einer Baumansicht verwendet.

Abstract

The diploma thesis Object-oriented representation of Keba TeachTalk programs consists of a Parser for teachtalk projects and a structured tree, which visualise single files hierarchical. These files are searched for Routines, Variables, Constants, Inherits, and Type declarations. Regular expressions are used for this purpose so that the parser can break down the code into individual components. These components are saved in C# classes in which the structured tree is build up. The structured tree is like a linked list and creates, because of the inherits between teachtalk files a hierarchy. To visual represent the whole structured tree a WPF-App is being used which contains a tree view.

Inhaltsverzeichnis

1 Allgemein 7
1.1 Teachtalk 7
1.1.1 Bezeichner 7
1.1.2 Schlüsselwörter 7
1.1.3 Datentypen 8
1.1.4 Attribute 8
1.1.5 ROUTINE 10
1.1.6 Variable 11
1.1.7 Konstante 12
1.1.8 Type 13
1.1.9 Vererbung 13
2 Erklärung Regex 14
2.1 Verwendung Regex 17
2.2 ROUTINE 17
2.2.1 Routine Parameter 19
2.2.2 Routine ohne Parameter 21
2.3 VARIABLE 22
2.3.1 Variable Kompletter Block 22
2.3.2 Variable Name und Typ 24
2.4 Konstante 27
2.4.1 Konstante Kompletter Block 27
2.4.2 CONSTANT Name und Typ 29
2.5 TYPE 31
2.5.1 Type Kompletter Block 34
2.5.2 Strukturierte TYPE 36
2.5.3 Mehrzeilige Type 37
2.5.4 TYPE Name und Typ 38
2.6 Vererbungen 39
3 Parser 41
3.1 C# Regex Klasse 41
3.2 TeachtalkFile 43
3.3 Parser Methoden 43

3.3.1 Parse Parent 44
3.3.2 Parse -Routines 44
3.3.4 Parse Variables 47
3.3.5 Parse Constants 49
3.3.6 Parse Types 50
4 Baum Ansicht 54
4.1 TeachTalk File 56
4.2 Children 57
4.3 Routinen 59
4.3.1 Return Type 60
4.3.2 Attribut 60
4.3.3 Parameter 61
4.3.4 Attribut 62
4.4 Typ 63
4.5 Variablen 64
4.6 Konstanten 66
4.7 Suchfunktion 67
4.7.1 Benutzeroberfläche 67
4.7.2 Programmcode 68
5 Unit Test 71
6 Technologien 75
6.1 Windows Presentation Foundation (WPF) 75
6.2 Visual Studio 77
6.3 Unit Test 77
6.4 Regex101 78
7 Resümee 78
8 Planung 79
Literaturverzeichnis 81

1 Allgemein

Teachtalk ist eine objektorientierte Programmiersprache, die von der Firma KEBA für die Steuerung der KeMotion Robotik-Software entwickelt wurde und von der Firma Engel dafür verwendet wird. Für diese Diplomarbeit wurde ein Parser entwickelt, der Teachtalk Projekte mit Hilfe von Regular Expressions einliest und daraus einen Strukturbaum erstellt. Der Programmcode befindet sich in Textdateien und ein Projekt besteht aus mehreren Dateien und Unterverzeichnissen.

1.1 Teachtalk

Ein Teachtalk Programm besteht aus Deklarationen für Variablen, Konstanten und Typen und aus Anweisungen, die in Routinen stehen. Eine Teachtalk Klasse kann Eigenschaften wie Variablen und Routinen von einer anderen Klasse erben. Das wird mit dem Schlüsselwort INHERIT angegeben. Aufgrund dieser Vererbung wird der Strukturbaum aufgebaut.

1.1.1 Bezeichner

Diese dienen in Teachtalk zur Identifikation von Projekten, Programmen, Bausteinen, Routinen, Variablen, Konstanten, wie auch Typen. Diese Bezeichner sind eine Folge von Buchstaben, Ziffern und Zeichensymbolen.

1.1.2 Schlüsselwörter

Schlüsselwörter, wie beispielsweise ROUTINE oder VAR, werden für Deklarationen und Anweisungen verwendet. Sie werden in Teachtalk immer großgeschrieben und dürfen nicht als Bezeichner für Programme, Routinen, Variablen oder Typen verwendet werden

1.1.3 Datentypen

In Teachtalk gibt es elf Basisdatentypen.

- BOOL für Wahrheitswerte
- SINT für 8-Bit Ganzzahlen
- INT für 16-Bit Ganzzahlen
- DINT für 32-Bit Ganzzahlen
- LINT für 64-Bit Ganzzahlen
- BYTE für 8-Bit Bitmuster
- WORD für 16-Bit Bitmuster
- DWORD für 32-Bit Bitmuster
- LWORD für 64-Bit Bitmuster
- REAL für 32-Bit Fließkomma Zahlen
- STRING für Zeichenketten mit einer maximalen Länge von 255 Zeichen

Je nach Datentyp können verschiedene Operatoren wie logische Operatoren (AND, OR, XOR, NOT), arithmetische und Vergleichsoperatoren oder bitweise Operatoren verwendet werden.

In Teachtalk können auch eigene Typen deklariert werden, die aus einem oder mehreren Basisdatentypen bestehen. Dabei wird unterschieden zwischen einfachen Typen, Enumerations-Typen, Feld-Typen und Struktur-Typen.

1.1.4 Attribute

Variablen, Konstanten, Typen und Routinen in Teachtalk können Attribute besitzen, mit denen zusätzliche Eigenschaften festgelegt werden.

PRIVATE

Wenn etwas mit dem Attribut PRIVATE versehen wird, kann dieses Objekt nur in der Datei, in der es deklariert wurde, verwendet werden.

GLOBAL

Global ist das Gegenteil von PRIVATE. Alle mit GLOBAL deklarierten Konstanten, Typen, Variablen und Routinen sind im ganzen Projekt verfügbar. Globale Deklarationen im Systemprojekt (ein teachtalk Basisverzeichnis hat ein Systemprojekt und ein Globalprojekt, die den anderen Projekten im Verzeichnis hierarchisch übergeordnet sind) sind in allen Projekten verfügbar.

SAVE

Das Attribut SAVE kann nur für Variablen und Typen verwendet werden. Wird das Programm beendet, werden die Werte von Variablen mit SAVE-Attribut oder Variablen von Typen, die mit dem SAVE Attribut deklariert wurden in einer Datei gesichert

SAVE PROJECT

Wenn man sowohl das Attribut SAVE als auch PROJECT für Variablen im System- oder Globalprojekt verwendet, werden die durch SAVE gesicherten Werte in den Applikationsprojekten abgelegt. Wenn so ein Projekt dann geladen wird, werden die Variablen mit den gespeicherten Werten befüllt.

CONST

Das CONST Attribut kann für Variablen, Strukturelemente und Bausteinelemente verwendet werden und gibt an, dass die darin enthaltenen Werte durch das Programm nicht mehr verändert werden können. Diesen Elementen sollte bei der Deklarierung gleich ein Wert zugewiesen werden, weil es danach nicht mehr möglich ist.

READONLY

Elemente mit diesem Attribut können nur von innerhalb der Datei verändert werden. Anders als beim Attribut PRIVATE, wo der Lese- und Schreibzugriff von außerhalb gesperrt ist, können READONLY Elemente von außen gelesen, aber nicht verändert werden.

USER

Mit dem Attribut USER deklarierte Variablen, Typen, Routinen, Bausteine und Programme werden zu User-Objekten und können dadurch in der End-User-Ebene verwendet werden.Die End-User-Ebene eines Projekts ist die Deklarationsebene, die aus

NOINIT

Werte von Elementen mit dem USER-Attribut können von den Enduserdateien in die Variablendeklaration übernommen werden. Wird das NOINIT Attribut hinzugefügt, werden sie nicht in die Variablendeklaration geschrieben.

NAME

Der Name eines Teachtalk Bausteins leitet sich normalerweise aus dem Dateinamen ab, der kleingeschrieben ist. Möchte man die Groß- und Kleinschreibung des Namens nachträglich ändern, kann man das NAME-Attribut verwenden. Es kann aber nur die Groß- und Kleinschreibung verändert werden, ansonsten muss der Name gleichbleiben.

ABSTRACT

Mit dem Attribut ABSTRACT kann man Bausteine und Routinen als abstrakt deklarieren. Diese können nicht instanziiert werden und abstrakte Routinen können nur in abstrakten Bausteinen erstellt werden. Wenn ein nicht-abstrakter Baustein von einem abstrakten Baustein erbt, müssen die abstrakten Routinen überschrieben werden.

EXPORT

Dieses Attribut kann nur für Typen im Systemprojekt verwendet werden. Dieser Typ darf nicht für routinenlokale Variablen verwendet werden, darf keine Referenzen enthalten und kann nicht gemeinsam mit SAVE verwendet werden. Eine Variable dieses Typs kann steuerungsweit als Shared-Memory-Objekt verwendet werden.

(vgl. KEBA, 2010)

1.1.5 ROUTINE

Routinen sind Anweisungen in einem Programm die immer wieder aufgerufen werden können sie sind mit Methoden in Java vergleichbar. Sie enthalten den ausführbaren Programmcode. Diese bestehen aus einen Routinenkopf mit den Namen dieser Routine, einer Parameterliste, dem Rückgabewert sowie anderen optionalen Attributen. Sie können routinenlokale Variablen besitzen, welcher nur in dieser Routine verwendet, werden können. Die Parameter einer Routine können auch mit dem Schlüssel Wort „CONST“ als schreibgeschützt gekennzeichnet werden. Durch die Angabe von „VAR_IN“ können Parameter als Wertparameter gekennzeichnet werden, jedoch können „CONST“ und „VAR_IN“ nicht für denselben Parameter angegeben werden. Alle Parameter die nicht mit „VAR_IN“ gekennzeichnet sind werden als Referenz übergeben.

```
ROUTINE Bremsweg(CONST v : REAL; CONST a : REAL) : REAL // Routinenkopf
```

```
VAR
s : REAL;                                         // routinenlokale
END_VAR                                           //Variablendeklaration
s := (v * v) / (2.0 * a);                         // Beginn Anweisungsblock
RETURN s;
END_ROUTINE
```

(KEBA, 2010)

Wenn Routinen keinen Rückgabewert besitzen, muss man darauf achten, dass dies nicht explizit im Routinenkopf angegeben ist wie beispielsweiße in Java mit dem Schlüsselwort void.

```
ROUTINE HochN(x : REAL; CONST n : DINT) // Routinenkopf
VAR
p : REAL;                               // routinenlokale
i : DINT;
END_VAR                                 //Variablendeklaration
p := 1;
FOR i := 1 TO n DO
p := p * x;
END_FOR;
x := p;
END_ROUTINE
```

(KEBA, 2010)

1.1.6 Variable

Variablen sind Elemente, auf welche man immer wieder im Programmcode zugreifen kann. Mit dem Schlüsselwort VAR beginnt man einen Variablendeklaration, in welcher man den Namen oder Datentypen, wie auch Attribute festlegen kann. Mithilfe eines „:“ trennt man den Variablennamen von den Datentypen. Hierbei kann man entweder einen vordeklarierten Datentyp verwenden oder einen selbst erstellten Datentyp, welcher jedoch zuvor in einer Typendeklaration definiert werden muss.

```
VAR
k : REAL;
a : ARRAY [10] OF REAL;
mk99 : BOOL;
name : STRING;
END_VAR
```

(KEBA, 2010)

1.1.7 Konstante

Konstanten sind ähnlich wie Variablen besitzen aber den entscheidenden Unterschied, dass dessen Wert nicht geändert werden kann. Konstantendeklarationen beginnt man mit dem Schlüsselwort CONSTANT. In dieser Deklaration kann man den Namen der Konstante festlegen. Der Deklarationsblock endet mit END_CONSTANT. Es ist wichtig zu beachten, dass den Konstanten in diesem Block ein Wert zugewiesen wird. Diese Werte können Ausdrücke sein, in denen aber nur andere benutzerdefinierte benannte Konstanten und konstante Literale vom selben Typ verwendet werden können.

Konstanten können von einem Basistyp sein, es gibt also nur Boolsche-, Integer-, Real- und String-Konstanten. Neben Basistypen sind auch Enumerationstypen als Konstantentypen erlaubt. Konstanten belegen keinen Speicher und dürfen im Allgemeinen überall verwendet werden, wo konstante Literale vorkommen dürfen.

```
CONSTANT
ON : BOOL := TRUE;
PI : REAL := 3.14159;
MAX : DINT := 10 + 10;
NAME : STRING := "Entnahmeroboter";
offset : DINT := 1000;
```

```
first : DINT := offset + 1;
END_CONSTANT
```

(KEBA, 2010)

1.1.8 Type

Mit dem Schlüsselwort TYPE kann man Typendeklarationen beginnen. Typdeklaration sind vom Anwender definierte Datentypen mit Namen.

Typdeklarationen werden mit den Schlüsselwörtern TYPE und END_TYPE

begrenzt.

Beispiel für eine Typdeklarationen:

```
TYPE
token : DINT; // einfacher Typ
tCoord : (AXIS, WORLD); // Enumerations-Typ
tVector : ARRAY [3] OF REAL; // Feld-Typ
tPoint : STRUCT // Struktur-Typ
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

(KEBA, 2010)

1.1.9 Vererbung

Die INHERIT-Anweisung kann nur einmal im Programm verwendet werden. Es ist auch wichtig, dass dies die erste Anweisung im Programm ist. Sie gibt den Typ an, von dem das Programm abgeleitet ist und dessen Variablen und Routinen es erben soll.

2 Erklärung Regex

Regex (eine Abkürzung für regular expression, auf Deutsch Reguläre Ausdrücke) sind eine Art Filterkriterium, mit deren Hilfe man Zeichenketten analysieren und manipulieren kann.

Ein regulärer Ausdruck ist eine Folge von Buchstaben und Symbolen, die ein Suchmuster bilden. Sie werden oft in Programmen verwendet, welche Text Segmente suchen oder bearbeiten, um Zeichenfolgen oder Wörter zu finden. Reguläre Ausdrücke werden häufig in den Pattern-Matching-Systemen von Unix-ähnlichen Betriebssystemen sowie in vielen Programmiersprachen wie Phyton und C# verwendet.

Sie sind im Grunde eine Abfolge von Zeichen, die ein Suchmuster definieren. Diese Suchmuster können mit Mustern erstellt werden, welche Metazeichen genannt werden.

Verwendete Muster:

*	steht für mehrere oder kein Zeichen
+	steht für ein oder mehrere Zeichen
?	steht für eine optionale Bedingung
[]	steht für eine Character Class. Alle Zeichen in dieser Klasse können als nächstes kommen. Man kann dies auch mit anderen

<table>
<tr><td></td><td>Metazeichen verknüpfen beispielsweiße mit „*“ nun können alle Zeichen in dieser Klasse kein Mal oder mehrmals vorkommen</td></tr>
<tr><td>.</td><td>Steht für jedes einzelne Zeichen außer einen Zeilenumbruch</td></tr>
<tr><td>$</td><td>Steht für das Ende einer Zeile</td></tr>
<tr><td>\</td><td>Wird verwendet, um spezielles Verhalten von Metazeichen zu definieren oder deaktivieren beispielsweiße steht \n für einen Zeilenumbruch</td></tr>
<tr><td>|</td><td>Alternative Operator steht für oder</td></tr>
<tr><td>()</td><td>Definiert eine Gruppe</td></tr>
<tr><td>{}</td><td>Definiert die Anzahl der Wiederholungen oder den Bereich von Wiederholungen eines Zeichens oder Musters</td></tr>
<tr><td>^</td><td>Innerhalb einer Character Class „[]“ steht dieses Metazeichen für nicht. Außerhalb von dieser steht er für den Beginn einer Zeile</td></tr>
<tr><td>(?=…)</td><td>Diese Zeichenkette steht für einen Positive Lookahead. Hier findet man nur Zeichen wenn diese Bedingung in diesem erfüllt wurde, man würde beispielsweiße hier: bar(?=bar) nur ein bar finden auf welches auch ein bar folgt.</td></tr>
<tr><td>(?!...)</td><td>Diese Zeichenkette steht für einen Negative Lookahead. Hier findet man nur Zeichen wenn diese Bedingung in diesem erfüllt wurde,</td></tr>
</table>

	man würde beispielsweiße hier: bar(?!bar) nur ein bar finden auf welches kein bar folgt.
(?<=…)	Diese Zeichenkette steht für einen Positive Lookbehind. Hier findet man nur Zeichen wenn diese Bedingung in diesem erfüllt wurde, man würde beispielsweiße hier: (?<=foo)bar nur ein bar finden wenn ein foo davor steht
(?<!...)	Diese Zeichenkette steht für einen Negative Lookbehind. Hier findet man nur Zeichen wenn diese Bedingung in diesem erfüllt wurde, man würde beispielsweiße hier: (?<!foo)bar nur ein bar finden wenn kein foo davor steht

Jedes dieser Metazeichen kann miteinander verknüpft werden so ist es möglich komplexe REGEX Anweisungen zu schreiben und spezielle Teile aus einem Text zu filtern.

Beispiel:

```
^([A-Za-z0-9ÄÖÜäöüß]{6,250})$
```

Der Ausdruck erlaubt Passwörter mit sechs bis 250 Zeichen, welches Buchstaben aus dem Alphabet (in Groß und Kleinschreibung) sowie Umlaute (ebenfalls Groß und Klein) und „ß“ benutzt.

REGEX Anweisungen haben viele verschiedene Einsatzmöglichkeiten einige dieser sind:

- Suchen und ersetzen von Text in Dateien
- Dursuchen von Logdateien, vor allem, um Fehler oder andere bestimmte Ereignisse zu finden
- Durchsuchen von großen Datenmengen, um bestimmte Dateien zu finden
- Validierung von Formularfeldern auf Websites. Hierbei kann überprüft werden ob in einem Passwort nicht nur Wörter, sondern auch Zahlen verwendet werden

(Spriesterbach, 2023)

2.1 Verwendung Regex

In dieser Arbeit werden REGEX Anweisungen verwendet um Routinen, Variablen, Konstanten und TYPE-Blöcke aus den Teachtalk Programmcode zu filtern. Mithilfe von komplexen Anweisungen ist es möglich, all diese gesuchten Blöcke aus dem Code zu filtern und alle benötigten Werte und Attribute dieser in extra Gruppen zu speichern. Einiger dieser REGEX Anweisungen wurden auf mehrere Anweisungen aufgeteilt, um diese beim Vorgang des Parsen einfacher verwenden zu können.

2.2 ROUTINE

Möglicher Aufbau:

Es ist wichtig zu beachten das in jedem Programm eine unbenannte Routine existieren kann. Im Gegensatz zu benannten Routinen welche, wenn sie einen neuen Namen besitzen, keine begrenzte Anzahl haben, können Routinen ohne Namen nur ein mal im Programm vorkommen. Zunächst muss man bestimmen, ob die gewählte Routine eine Ereignis Routine ist oder nicht, falls dies der Fall ist, muss auch die Ereignisbedingung erfasst werden. Da Routinen ohne Rückgabewerte dies

nicht explizit angeben, erhält man bei diesen in unserem Regex eine leere Gruppe 3 zurück.

```
ROUTINE NEW()
a := MAP(c);
FOR i:= 0 TO 9 DO
b[i] := MAP(c[9-i]);
END_FOR;
END_ROUTINE
```

Es muss auch auf die Attribute einer Routine geachtet werden, bzw. dass diese auch explizit angegeben werden.

```
ROUTINE DORUN(CONST name : STRING) GLOBAL
START _DORUN(name);
END_ROUTINE
```

Unter Berücksichtigung aller dieser Möglichkeiten kommt man zu folgendem Regex:

```
"^ *ROUTINE\s*(\w*)[\ \(]*?([\w\,\s\:\.\;\[\]\ \(\)'/]*)
[/\w '\ \(\)]*?(?<!\(\w*)\)(?! *;) ?\:?[ ]?[ \)/]?((?<= *: *)[\w\[\]]*)?
 *([\w ]*)?[\s\S]*?(?=END_ROUTINE)END_ROUTINE"gm
```

Dieser Regex gibt alle Routinen im Programm zurück. Er besitzt fünf Gruppen, welche sich folgend aufteilen:

Group 0: in dieser Gruppe wird der gesamte Routinenkopf ausgewählt.

Group 1: Diese Gruppe erfasst den Namen einer Routine. Falls diese Routine jedoch keinen Namen besitz ist diese Gruppe null

Group 2: gibt alle Parameter einer Routine an (Bei Ereignisroutinen ist in dieser Gruppe auch die Ereignisbedingung enthalten)

Group 3: in dieser Gruppe wird der Rückgabewert der Routine zurückgegeben

Group 4: in dieser Gruppe werden die Attribute einer Routine zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 2 matches (2.5ms)

```
@" ^ *ROUTINE\s*(\w*)[\ \(]*?([\w\,\s\:\.\;\[\]\ \(\)'/]*)[/\w '\ \(\)]*?(?<!\(\w*)\))(?! *;) ?\:?[ ]?[ \)/]?((?<= *: *)[\w\[\]]*)? *([\w ]*)?[\s\S]*?(?=END_ROUTINE)END_ROUTINE
" gm
```

TEST STRING

```
ROUTINE Bremsweg(CONST v : REAL; CONST a : REAL) : REAL // Routinenkopf
VAR
s : REAL;                               // routinenlokale
// Variablendeklaration
END_VAR
s := (v * v) / (2.0 * a);               // Beginn Anweisungsblock
RETURN s;
END_ROUTINE 

ROUTINE DORUN(CONST name : STRING) GLOBAL
START _DORUN(name);
END_ROUTINE
```

2.2.1 Routine Parameter

Dieser REGEX gibt alle Parameter einer Routine zurück. Für die Parameter einer Routine wurde ein eigener Regex erstellt, um im Prozess des Parsen besser auf die Attribute der Elemente im Parameter zugreifen zu können.

```
"(?<![/ ])([\w\ ]*)[\ ;(](\w+) *(?=[ :]*)[:=] *([\w\ \[\:\]\(\;]*)\)?"gm
```

Dieser Regex gibt alle Routinen im Programm zurück. Er besitzt vier Gruppen, welche sich folgend aufteilen:

Group 0: in dieser Gruppe wird der Gesamte Block der Parameter zurückgegeben

Group 1: in dieser Gruppe werden alle Attribute eines Parameters zurückgegeben

Group 2: in dieser Gruppe wird der Name eines Parameters zurückgegeben

Group 3: in dieser Gruppe wird der Typ eines Parameters zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 3 matches (2.1

```
@" (?<![/ ])([\w\ ]*)[\ ;(]( \w+) *(?=[ :]*)[:=] *([\w\ \[\:\]\(]*)\)?  " gm
```

TEST STRING

```
ROUTINE Bremsweg(CONST v : REAL; CONST a : REAL) : REAL // Routinenkopf
VAR
s : REAL;                                   // routinenlokale
// Variablendeklaration
END_VAR
s := (v * v) / (2.0 * a);                   // Beginn Anweisungsblock
RETURN s;
END_ROUTINE

ROUTINE DORUN(CONST name : STRING) GLOBAL
START _DORUN(name);
END_ROUTINE
```

2.2.2 Routine ohne Parameter

```
"^ *ROUTINE\s*(\w*)\( *\)(?! *;) ?\:?[ ]?[ \)/]?((?<= *: *)[\w\[\]]*)?
 *([\w ]*)?[\s\S]*?(?=END_ROUTINE)END_ROUTINE"gm
```

Dieser Regex gibt alle Routinen ohne Parameter im Programm zurück. Er wurde auf vier Gruppen aufgeteilt, welche sich folgend aufteilen:

Group 0: in dieser Gruppe wird der gesamte Routinenkopf ausgewählt.

Group 1: Diese Gruppe erfasst den Namen einer Routine. Falls diese Routine jedoch keinen Namen besitz ist diese Gruppe null

Group 2: in dieser Gruppe wird der Rückgabewert der Routine zurückgegeben

Group 3: in dieser Gruppe werden die Attribute einer Routine zurückgegeben

2.3 VARIABLE

Der Regex für Variablen wurden auf zwei verschiedene Regexes aufgeteilt, da man ihn auf diese Weise effizienter nutzen kann. Hierbei wird der erste Regex „Variable Kompletter Block" dazu verwendet, um den gesamten Block der Variablendeklaration zurückzugeben. Der zweite Regex wird dazu verwendet, Datentypnamen und Varibalennamen sowie Attribute usw. aus dem Code herauszufiltern.

2.3.1 Variable Kompletter Block

Da Variablendeklarationen auf verschiedene Arten geschrieben werden können, muss man all diese verschiedenen Möglichkeiten berücksichtigen. Wenn man eine Variable mit dem User-Attribut deklariert, wird sie zu einem User-Objekt. Mit diesen ist die Verwendung auf der End-User-Ebene erlaubt.

```
VAR GLOBAL USER
i : SINT;
END_VAR
```

(KEBA, 2010)

In Deklarations-Blöcken kann das User-Attribut alleine stehen oder gemeinsam mit dem GLOBAL Attribut auftreten. Diese Attribute wie GLOBAL oder READONLY können aber auch allein in einem Variablendeklarationskopf stehen.

```
VAR PRIVATE
queued : BOOL;
confirm : BOOL;
END_VAR
```

(KEBA, 2010)

Variablen können auch kein Attribut besitzen also nur einen Namen.

```
VAR DEMO
result : BOOL;
END_VAR
```

(KEBA, 2010)

Unter Berücksichtigung aller dieser Möglichkeiten kommt man zu folgendem Regex:

```
"^\ *VAR *(?<=VAR) ?(\w*) *([\w\ ]*)([\s\S]*?)(?=END_VAR)END_VAR"gm
```

Dieser Regex gibt alle Variablen (VAR) zurück. Er besitzt drei Gruppen, welche folgende Werte enthalten

Group 0: Diese Gruppe gibt den gesamten Block der Variable zurück.

Group 1: Diese Gruppe gibt den Namen der Variable zurück. Falls jedoch ein Attribut für die Variabel explizit gegeben ist, wird dieser Attribut in dieser Gruppe zurückgegeben.

Group 2: Diese Gruppe ist optional. In ihr wird, falls ein Attribut im Variablenkopf gegeben ist, der Name zurückgegeben. Hier befindet sich auch das Schlüsselwort „User“, falls eine Variable mit dem User-Attribut zu einen User-Objekt deklariert wurde.

Test mir regex101.com:

REGULAR EXPRESSION 2 matches (0.5ms)

```
@" ^\ *VAR *(?<=VAR) ?(\w*) *([\w\ ]*)([\s\S]*?)(?=END_VAR)END_VAR " gm
```

TEST STRING

```
VAR PRIVATE
queued : BOOL;
confirm : BOOL;
END_VAR

VAR GLOBAL USER
i : SINT;
END_VAR
```

2.3.2 Variable Name und Typ

Einzelne Variablen können auch als Array definiert, werden. Dies bedarf auch einer speziellen Behandlung in unserem Regex.

```
VAR
k : REAL;
a : ARRAY [10] OF REAL;
mk99 : BOOL;
name : STRING;
END_VAR
```

(KEBA, 2010)

Sie können auch direkt bei der Initialisierung einen Wert besitzen. Dies gilt auch für Arrays.

```
VAR
a : ARRAY [3] OF INT := (4, 5, 6); // a[0] = 4, a[1] = 5, a[2] = 6
text : STRING[5] := "HELLO WORLD"; // text := "HELLO"
END_VAR
```

(KEBA, 2010)

Man kann Variablen Deklarationen nicht nur mit dem Schlüsselwort „VAR“ markieren, sondern auch mit dem Schlüsselwort „VAR_IN“. Somit wird der gekennzeichnete Wertparameter zu einem reinen Eingangsparameter.

```
VAR
m : MAPTO BOOL;
b : BOOL;
r : MAPTO ROUTINE() : BOOL;
END_VAR
```

(KEBA, 2010)

Mit Variablen vom Typ ANY können außer der Übergabe an eine Routine mit ANY-parameter keine Operationen durchgeführt werden. Es ist aber möglich mit IS_INSTANCE den Aktualtyp abzufragen sowie explizite Typumwandlungen zur Laufzeit durchzuführen. So kann es Zustande kommen das Konstanten in einer Variablendeklaration zu finden sind.

```
VAR GLOBAL
CONST msgClass : KMSG_Class; // In: Meldungsklasse
CONST compNr : DINT; // In: Komponentennummer
CONST msgNr : DINT; // In: Meldungsnummer
CONST instNr : DINT; // In: Instanznummer
CONST OPTIONAL param1 : ANY; // In: Parameter1
CONST OPTIONAL param2 : ANY; // In: Parameter2
CONST OPTIONAL param3 : ANY; // In: Parameter3
CONST OPTIONAL param4 : ANY // In: Parameter4
) : KMSG_Status;
END_VAR
```

(KEBA, 2010)

Variablentypen können auch ein Attribut besitzen wie beispielsweise CONST oder READONLY.

```
VAR
i : SINT CONST := 15;
s : TS;
END_VAR
```

(KEBA, 2010)

Unter Berücksichtig aller dieser Möglichkeiten kommt man zu folgendem Regex:

```
"(?<![/ ])([\w\ ]*)[\s;(](\w+) *(?=[ :]*)[:] *([\w\s\[\]\:\=]*)"gm
```

Group 0: in dieser Gruppe wird der gesamte Block der Variable zurückgegeben

Group 1: in dieser Gruppe werden alle Attribute einer Variable zurückgegeben

Group 2: in dieser Gruppe wird der Name der Variable zurückgegeben

Group 3: in dieser Gruppe wird der Typ einer Variable zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION

5 matches (0.5ms)

```
@"(?<![/ ])([\w\ ]*)[\s;(](\w+) *(?=[ :]*)[:] *([\w\s\[\]\:\=]*)" gm
```

TEST STRING

```
VAR PRIVATE
queued : BOOL; 
confirm : BOOL; 
i : SINT CONST := 15;
s : TS;
```

2.4 Konstante

Der Regex für Konstanten wurde auf zwei verschiedene Regex aufgeteilt, da man ihn auf diese Weise effizienter nutzen kann. Hierbei ist der erste Regex „Konstante Kompletter Block“ dazu da, um den gesamten Block der Konstantendeklarationen zurückzugeben. Der zweite Regex wird dazu verwendet, Datentypnamen und Konstantennamen sowie Attribute usw. aus dem Code herauszufiltern.

2.4.1 Konstante Kompletter Block

Konstanten können mit dem GLOBAL Attribut versehen werden. Dadurch werden sie im ganzen Projekt verfügbar.

```
CONSTANT GLOBAL
pi = 3.14159;
END_CONSTANT
```

(KEBA, 2010)

Unter Berücksichtigung aller dieser Möglichkeiten kommt man zu folgendem Regex:

```
"^\ *?CONSTANT *([\w\ ]*)[\s\S]*?(?=END_CONSTANT)END_CONSTANT"gm
```

Dieser Regex gibt alle Konstanten (CONSTANT) zurück. Er besitzt zwei Gruppen welche, folgende Werte enthalten:

Group 0: In dieser Gruppe wird der gesamte Block der CONSTANT Deklaration zurückgegeben.

Group 1: In dieser Gruppe wird das Attribut des CONSTANT zurückgegeben.

Test mit regex101.com:

REGULAR EXPRESSION 2 matches (0.4ms)

```
@" ^\ *?CONSTANT *([\w\ ]*)[\s\S]*?(?=END_CONSTANT)END_CONSTANT " gm
```

TEST STRING

```
CONSTANT GLOBAL
pi = 3.14159;
END_CONSTANT 

CONSTANT
ON : BOOL := TRUE;
PI : REAL := 3.14159;
MAX : DINT := 10 + 10;
NAME : STRING := "Entnahmeroboter";
offset : DINT := 1000;
first : DINT := offset + 1;
END_CONSTANT
```

2.4.2 CONSTANT Name und Typ

Bei Konstanten ist es auch möglich, das Attribut dieser anzugeben.

```
CONSTANT
ON : BOOL := TRUE;
PI : REAL := 3.14159;
MAX : DINT := 10 + 10;
NAME : STRING := "Entnahmeroboter";
offset : DINT := 1000;
first : DINT := offset + 1;
END_CONSTANT
```

(KEBA, 2010)

Wenn der Typ jedoch klar erkenntlich ist, muss man dieses nicht machen.

```
CONSTANT GLOBAL
pi = 3.14159;
END_CONSTANT
```

(KEBA, 2010)

```
"(?<![/ ])([\w\ ]*)[\s;(](\w+) *(?=[ :]*)[:=] *([\w\ ]*)
[ :=]*([\w\ ""+\-[\]:\.]*)"gm
```

Dieser Regex gibt alle Typen und Attribute von Konstanten (CONSTANT) zurück. Er besitzt fünf Gruppen, welche folgende Werte enthalten

Group 0: In dieser Gruppe wird der gesamte Block der Konstante zurückgegeben

Group 1: In dieser Gruppe wird das Attribut der Konstante zurückgegeben

Group 2: In dieser Gruppe wird der Name der Konstante zurückgegeben

Group 3: In dieser Gruppe wird der Typ einer Konstante zurückgegeben

Group 4: In dieser Gruppe wird der Wert einer Konstante zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 7 matches (1.2ms)

```
@"(?<![/ ])([\w\ ]*)[\s;(](\w+) *(?=[ :]*)[:=] *([\w\ ]*)[ :=]*([\w\ ""+\-[\]:]*)" gm
```

TEST STRING

```
CONSTANT GLOBAL
pi := 3.14159;
END_CONSTANT

CONSTANT
ON : BOOL := TRUE;
PI : REAL := 3.14159;
MAX : DINT := 10 + 10;
NAME : STRING := "Entnahmeroboter";
offset : DINT := 1000;
first : DINT := offset + 1;
END_CONSTANT
```

2.5 TYPE

Es ist auch möglich strukturierte Typendeklartionen zu erstellen, diese strukturierte Datentypen Enthalten gleichartige oder zusammengehörige Elemente.

Als strukturierte Datentypen stehen zur Verfügung:

- Arrays: Enthält Datenfelder mit gleichartigen Elementen,
- Strukturen: Enthält zusammengehörige Elemente

Beispiel einer ARRAY Deklaration:

```
TYPE
// Feld mit drei REAL Elementen, Indizes 0, 1 und 2 :
tVector : ARRAY [3] OF REAL;
```

```
// Feld mit drei STRING Elementen, Indizes 1, 2 und 3 :
tList : ARRAY [1..3] OF STRING;
// 3x3 Matrix mit 9 DINT Elementen:
tMatrix1 : ARRAY [3] OF ARRAY [3] OF DINT;
// oder auch:
tMatrix2 : ARRAY [3, 3] OF DINT;
END_TYPE
```

(KEBA, 2010)

Beispiel einer Strukturdeklaration:

```
TYPE
tPoint : STRUCT // Struktur-Typ mit drei Elementen
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

(KEBA, 2010)

In strukturierten Datentypen ist es möglich, das letzte Element mit Varianten zu versehen. Abhängig von der Art des beschreibenden Elements für die Varianten erfolgt der Zugriff auf die gleichen Daten auf unterschiedliche Weise. Während der Laufzeit des Programms wird überprüft, ob das beschreibende Element und die Zugriffsweise übereinstimmen. Das beschreibende Element für Varianten muss eine Aufzählung sein. Varianten können keine strukturellen Typen enthalten. Der Speicherbedarf für verschiedene Varianten muss nicht gleich sein. Es wird so viel Speicher reserviert, wie die größte Variante benötigt.

Beispiel einer Struktur-Deklaration mit Varianten:

```
TYPE
```

```
tPoint : STRUCT
CASE coord : tCoord OF // coord beschreibt die Variante
AXIS : (axis : tVector); // für coord = AXIS
WORLD : (x : REAL; // für coord = WORLD
y : REAL;
z : REAL)
END_STRUCT;
END_TYPE
```

(KEBA, 2010)

Eine weitere Art der Typendeklaration ist die Deklaration von Aufzählungstypen. Diese stellen eine Aufzählung (Enumeration) in einer Liste von eindeutigen Namen dar und definieren einen Unterbereich des ganzzahligen Typs DINT. Die aufgezählten Namen werden im Programm als Ganzzahlkonstanten verwendet. Besitzen die Namen keine Initialwerte, wird dem ersten Namen der Liste der Wert 0 zugeordnet. Jeder weitere Name erhält einen um 1 höheren Wert als sein Vorgänger. Verschiedene Namen dürfen nicht denselben Wert repräsentieren.

Einer Enumerationsvariablen kann nur mit einer expliziten Typumwandlung Wert von einem Ganzzahltyp wie INT oder DINT zugewiesen werden. Erwartet eine Routine einen nicht als CONST deklarierten Referenzparameter von einem Ganzzahltyp so kann ihr keine Enumerationsvariable übergeben werden. Eine Referenzvariable auf einen Ganzzahltyp kann nicht auf eine Enumerationsvariable gemappt werden. Beim Programmstart erhalten Enumerationsvariablen den Wert des ersten Namens der Liste der Enumeration.

```
TYPE
tCoord : (AXIS, WORLD); // AXIS=0, WORLD=1
tColor : (RED:=2, GREEN:=16, BLUE); // RED=2 GREEN=16 BLUE=17
END_TYPE
```

(KEBA, 2010)

Durch Angabe des kleinsten und des größten möglichen Wertes kann ein Typ als Unterbereich des ganzzahligen Typs DINT deklariert werden. Der kleinste mögliche Wert muss dabei zuerst angegeben werden. Bei Zuweisungen wird der gültige Wertebereich für die Variable überprüft. Besitzt eine Variablendeklaration dieses Typs keinen Initialwert, so wird mit dem Wert der unteren Schranke initialisiert.

Beispiele von Bereichstypen:

```
TYPE
tRange : DINT(1..10); // Wertebereich 1..10
END_TYPE
```

(KEBA, 2010)

2.5.1 Type Kompletter Block

Typen können mit dem Attribut Global versehen werden, mit welchen sie im ganzen Projekt verwendet werden können

```
TYPE GLOBAL
tPos : STRUCT
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

(KEBA, 2010)

```
"^\ *TYPE *(?<=TYPE) ?([\w\ ]*)([\s\S]*?)(?=END_TYPE)END_TYPE"gm
```

Dieser Regex gibt alle Variablen (VAR) zurück. Er besitzt drei Gruppen, welche folgende Werte enthalten

Group 0: In dieser Gruppe wird der Gesamte Block des TYPE zurückgegeben

Group 1: In dieser Gruppe wird das Attribut des Typen zurückgegeben

Group 2: In dieser Gruppe werden alle Elemente des Typen zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 2 matches (1.4ms)

```
@" ^\ *TYPE *(?<=TYPE) ?([\w\ ]*)([\s\S]*?)(?=END_TYPE)END_TYPE " gm
```

TEST STRING

```
TYPE
// Feld mit drei REAL Elementen, Indizes 0, 1 und 2 :
tVector : ARRAY [3] OF REAL;
// Feld mit drei STRING Elementen, Indizes 1, 2 und 3 :
tList : ARRAY [1..3] OF STRING;
// 3x3 Matrix mit 9 DINT Elementen:
tMatrix1 : ARRAY [3] OF ARRAY [3] OF DINT;
// oder auch:
tMatrix2 : ARRAY [3, 3] OF DINT;
END_TYPE

TYPE GLOBAL
tPos : STRUCT
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

Um im Parser eine vereinfachte Verarbeitung von Typen zu gewährleisten, werden spezielle Arten von Typendeklarationen zuerst gefiltert.

2.5.2 Strukturierte TYPE

Um in der Darstellung der Typen nicht jedes Element einer Struktur als eigen Typen darzustellen, wird jedes Element einer Struktur in einer Gruppe ausgewählt. Dieser Block wird dann im Programm aus dem Block des Typen entfernt.

```
"^ *(\w*) *:* *(STRUCT)([\s\S]*?)(?=END_STRUCT)END_STRUCT"gm
```

Dieser REGEX gibt alle Strukturen innerhalb eines TYPE zurück. Er besitzt vier Gruppen. Diese sind folgend unterteilt:

Group 0: in dieser Gruppe wird der gesamte Block der Struktur zurückgegeben

Group 1: in dieser Gruppe wird der Name einer Struktur zurückgegeben

Group 2: in dieser Gruppe wird der Typ des TYPE zurückgegen

Group 3: in dieser Gruppe werden alle Elemente der Struktur zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 1 match (0.3ms)

```
@" ^ *(\w*) *:* *(STRUCT)([\s\S]*?)(?=END_STRUCT)END_STRUCT " gm
```

TEST STRING

```

TYPE GLOBAL
tPos : STRUCT
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

2.5.3 Mehrzeilige Type

Um in der Darstellung der Typen nicht jedes Element eines Mehrzeiligen Typen als eigen Typen darzustellen wird jedes Element dieser in einer Gruppe ausgewählt. Dieser Block wird dann im Programm aus dem Block des Typen entfernt.

```
"^(?<![ //*]) *([\w,\ ]*)[:;(\ ] *(\w+)
 +(?=[ :(]*)[:=]+\s*\(\s*([\,\:\=\[\w\.\ \-\]\{\}""\|\s]*);?"gm
```

Dieser REGEX gibt alle Mehrzeiligen Typen innerhalb eines TYPE zurück. Er besitzt vier Gruppen. Diese sind folgend unterteilt:

Group 0: in dieser Gruppe wird der Gesamte Block der Mehrzeiligen Typen zurückgegeben

Group 1: in dieser Gruppe wird der Name einer Mehrzeiligen Typen zurückgegeben

Group 2: in dieser Gruppe wird das Attribut des Mehrzeiligen Typen zurückgegeben

Group 3: in dieser Gruppe werden alle Elemente des Mehrzeiligen Typen zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION 1 match (0.4ms)

```
@" ^(?<![ //*]) *([\w,\ ]*)[:;(\ ] *(\w+) +(?=[ :(]*)[:=]+\s*\(\s*([\,\:\=\[\w\.\ \-\]\{\}""\|\s]*);? " gm
```

TEST STRING

```

  t1files :
(
  a1_megafile,
  a1_logfile,
  a2_logfile,
);
```

2.5.4 TYPE Name und Typ

```
"^ *([\s\w]*?) *(\w+)\ *:\ *([\(\)\,\:\=\[\w\.\ \-\]\{\}""\|]*);?"gm
```

Dieser REGEX gibt alle definierten Variablen eines TYPE zurück. Er besitzt vier Gruppen. Diese sind folgend unterteilt:

Group 0: in dieser Gruppe wird der gesamte Block des Elements zurückgegeben

Group 1: in dieser Gruppe werden alle Attribute des Elements zurückgegeben

Group 2: in dieser Gruppe wird der Name des Elements zurückgegeben

Group 3: in dieser Gruppe wird der Typ des Elements zurückgegeben

Test mit regex101.com:

REGULAR EXPRESSION — 8 matches (15.0ms)

```
@" ^ *([\s\w]*?) *(\w+)\ *:\ *([\(\)\,\:\=\[\w\.\ \-\]\{\}""\|]*);?" gm
```

TEST STRING

```
TYPE
// Feld mit drei REAL Elementen, Indizes 0, 1 und 2 :
tVector : ARRAY [3] OF REAL;
// Feld mit drei STRING Elementen, Indizes 1, 2 und 3 :
tList : ARRAY [1..3] OF STRING;
// 3x3 Matrix mit 9 DINT Elementen:
tMatrix1 : ARRAY [3] OF ARRAY [3] OF DINT;
// oder auch:
tMatrix2 : ARRAY [3, 3] OF DINT;
END_TYPE

TYPE GLOBAL
tPos : STRUCT
x : REAL;
y : REAL;
z : REAL;
END_STRUCT;
END_TYPE
```

2.6 Vererbungen

Beispiel für Vererbungs-Anweisungen:

```
// Bausteindatei Tderived.tts
INHERIT Tbase;
```

(KEBA, 2010)

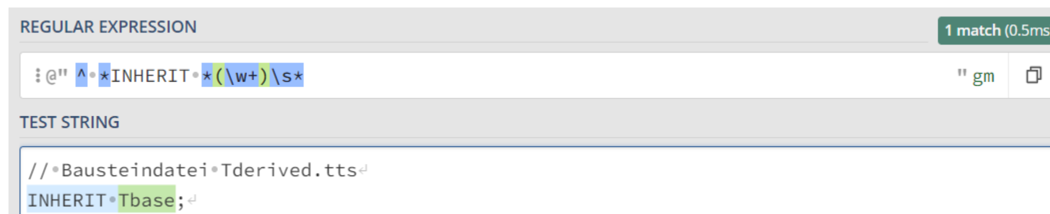
```
"^ *INHERIT *(\w+)\s*"gm
```

Dieser Regex gibt alle Vererbungen (Inherit) zurück. Er besitzt zwei Gruppen welche, folgende Werte enthalten

Group 0: in dieser Gruppe wird der gesamte Block der Vererbung zurückgegeben

Group 1: in dieser Gruppe wird der Name der zu erbenden Klasse zurückgegeben

Test mit regex101.com:



3 Parser

Der Parser ist eine C# Konsolenanwendung, der alle Teachtalk-Programmcodedateien aus einem bestimmten Verzeichnis und dessen Unterverzeichnissen einliest. Der Code wird dann mit Hilfe der Regular Expressions aus dem vorigen Kapitel in die einzelnen Bestandteile zerlegt und diese werden mit ihren Eigenschaften in Listen gespeichert. Dieser Vorgang wird als Parsen bezeichnet.

3.1 C# Regex Klasse

Um mit Regular Expressions zu arbeiten, stellt C# die Klasse Regex zur Verfügung. Um eine neue Regex zu erstellen, wird dem Konstruktor der Regex Klasse ein String, der den Regular Expression beinhaltet und optional eine oder mehrere RegexOptions übergeben.
Beispiel für die Erstellung einer Regex in C#:

```
public static readonly Regex variables = new Regex(@"(?<![/ ])([\w\
]*)[\s;(](\w+) *(?=[ :]*)[:] *([\w\s\[\]]*)", RegexOptions.Multiline);
```

Hier wird RegexOptions.Multiline verwendet, um mehrzeilige Strings richtig zu matchen.

Die Regex Klasse enthält auch viele Methoden, um mit Regular Expressions zu arbeiten. Die folgenden Methoden werden vom Parser verwendet:

- Match(String)
  gibt das erste Vorkommen des Strings, der der Regular Expression entspricht in dem übergebenen String zurück
- Matches(String)
  sucht den übergebenen String nach allen Vorkommen der Regular Expression und gibt diese zurück
- Replace(String, String)
  Ersetzt alle Textstellen die durch die Regular Expression vorgegeben werden innerhalb des ersten übergebenen String und ersetzt diese durch den zweiten String

(vgl. Regex Class, kein Datum)

Der Rückgabewert der Methoden Match und Matches ist eine Instanz der Klasse Match bzw. eine MatchCollection, die alle Matches enthält. Dieses Objekt enthält die Ergebnisse einer einzelnen gefundenen Übereinstimmung. Ein Match enthält folgende Eigenschaften:

- Captures
  enthält eine Gruppe mehrere Matches, bekommt man von Captures eine Collection aller dieser Matches
- Empty
  enthält alle fehlgeschlagenen Matches
- Groups
  enthält alle Gruppen eines Matches
- Index
  Die Position im originalen String, an der das Match gefunden wurde
- Length
  Die Länge des Strings der gematcht wurde
- Name
  Der Name der Gruppe
- Success
  gibt an, ob der Match erfolgreich war
- Value
  gibt den String an, der gematcht wurde
- ValueSpan
  gibt die Spanne des Matches im originalen String an

(vgl. Match Class, kein Datum)

3.2 TeachtalkFile

Wenn ein neues TeachtalkFile erstellt wird, wird eine Datei übergeben, die im Konstruktor eingelesen wird. Danach wird eine TeachtalkClass erstellt und die Methoden zum Parsen der Datei aufgerufen.

```
242    Public TeachtalkFile(FileInfo file)
243    {
244        File = file;
245        var reader = new StreamReader(File.FullName, Encoding.UTF8);
246        FileContent = reader.ReadToEnd();
247        reader.Close();
248        TTClass = new
       TeachtalkClass(Path.GetFileNameWithoutExtension(File.FullName));
249        TTClass.ParentString = parseParent();
250        TTClass.Routines = parseRoutines();
251        TTClass.Variables = parseVariables();
252        TTClass.Constants = parseConstants();
253        TTClass.Types = parseTypes();
254
255        testPrintTTClass();
256    }
```

Eine TeachtalkClass besteht aus den folgenden Bestandteilen:

- Name
  ist der Dateiname der Teachtalkdatei und wird als String gespeichert
- ParentString
- Children
  werden beim Erstellen des Strukturbaums in die Liste gespeichert
- Variables
- Constants
- Routines
- Types

3.3 Parser Methoden

Für die Bestandteile Parent, Variables, Constants, Routines und Types der TeachtalkClass gibt es in der Klasse TeachtalkFile jeweils eine eigene Methode zum Parsen

3.3.1 Parse Parent

Wenn eine Klasse Eigenschaften einer anderen Klasse erbt, wird das in teachtalk mit dem Schlüsselwort INHERIT angegeben. Die Klasse von der geerbt wird, wird als Parent oder Elternklasse bezeichnet.

```
18      public string parseParent()
19      {
20          var match = RegularExpressions.inherit.Match(FileContent);
21
22          return match.Groups[1].Value;
23
24      }
```

In der Methode parseParent wird mit der Regular Expression nach dem Schlüsselwort gesucht und der Name der Elternklasse, der sich in Gruppe 1 der Elternklasse befindet zurückgegeben.

3.3.2 Parse -Routines

Eine TeachtalkRoutine besitzt die folgenden Eigenschaften:

- Name
- Returntype
- Parameters
- Attributes

Wichtig zu berücksichtigen bei Routinen ist auch, dass alle diese Eigenschaften optional sind. Es kann also zum Beispiel eine Routine ohne Namen, Rückgabewert, Parameter und Attribute geben. Von den Parametern und Attributen kann eine Routine aber auch mehrere haben. Damit alle diese Fälle berücksichtigt werden können, parst der erste Teil der parseRoutines Methode nur die Routinen ohne Parameter.

```
27     List<TeachtalkRoutine> routines = new List<TeachtalkRoutine>();
28     var matches = RegularExpressions.routineNoParam.Matches(FileContent);
29     foreach (Match match in matches)
30     {
31         var routine = new TeachtalkRoutine();
32         routine.Name = match.Groups[1].Value;
33         routine.ReturnType = match.Groups[3].Value;
34         routine.Attributes.AddRange(match.Groups[4].Value.Split(' '));
35         foreach (string attribute in routine.Attributes.ToList())
36         {
37             if (String.IsNullOrWhiteSpace(attribute))
38             {
39                 routine.Attributes.Remove(attribute);
40             }
41             else
42             {
43                 attribute.Trim();
44             }
45         }
46         routines.Add(routine);
47     }
```

Hier wird zuerst die Regular Expression verwendet, die nur nach Routinen ohne Parameter sucht. Danach wird für jedes Match eine neue TeachtalkRoutine erstellt und der Name und Rückgabetyp zugewiesen. Die Regex gibt alle für die Attribute einen String zurück, wo die Attribute mit Leerzeichen getrennt stehen, weil man nicht weiß, wie viele Attribute eine Routine hat. Deshalb wird die String-Methode Split mit dem Parameter (‘ ‘) aufgerufen, die den String nach Leerzeichen trennt und eine Liste von Strings zurückgibt. Um keine ungewollten Leerzeichen in den Strings zu haben, wird die ganze Liste mit einer foreach-Schleife durchlaufen. Dafür muss eine Kopie der Liste verwendet werden, die mit ToList erstellt wird, weil man in der foreach-Schleife nichts aus der Liste entfernen darf. Im ersten Schritt wird gefragt, ob ein String leer ist oder nur aus Leerzeichen besteht. Ist das der Fall, wird er wieder aus der Liste entfernt. Wenn das nicht der Fall ist, wird die String-Methode Trim aufgerufen. Diese entfernt Leerzeichen am Anfang und am Ende eines Strings.

```
48     string fileWithoutNoParamRoutines =
       RegularExpressions.routineNoParam.Replace(FileContent,String.Empty);
49     matches = RegularExpressions.routine.Matches(fileWithoutNoParamRoutines);
```

Nachdem alle Routinen ohne Parameter gefunden wurden, wird ein neuer String fileWithoutNoParamRoutines erstellt. Mit der Regex Methode Replace werden die Routinen, die bereits eingelesen wurden, durch einen leeren String ersetzt, damit sie nicht von der nächsten Regex noch einmal gematcht werden. Der Dateiinhalt ohne diese Routinen wird in dem neuen String gespeichert, damit der FileContent nicht verändert wird. Die Regular Expression für Routinen mit Parameter sucht im neu

erstellten String nach allen restlichen Routinen. Die nächsten Schritte für Name, Rückgabewert und Attribute der Routine sind gleich wie bei den Routinen ohne Parameter. Der Unterschied ist, dass am Ende noch die Parameter in die Parameterliste der TeachtalkRoutine gespeichert werden.

```
68    var MatchParameters =
      RegularExpressions.parameters.Matches(match.Groups[2].Value);
69    foreach (Match parameter in MatchParameters)
70    {
71        var parameterVariable = new TeachtalkVariable();
72        parameterVariable.Attributes.AddRange(parameter.Groups[1].Value.Split('
      '));
73        foreach (string attribute in parameterVariable.Attributes.ToList())
74        {
75            if (String.IsNullOrWhiteSpace(attribute))
76            {
77                parameterVariable.Attributes.Remove(attribute);
78            }
79            else
80            {
81                attribute.Trim();
82            }
83        }
84        if (parameter.Groups[1].Value.IndexOf("OPTIONAL",
      StringComparison.OrdinalIgnoreCase) >= 0)
85        {
86            parameterVariable.IsOptional = true;
87        }
88        else
88        {
90            parameterVariable.IsOptional = false;
91        }
92
93        parameterVariable.Name = parameter.Groups[2].Value;
94        parameterVariable.Type = parameter.Groups[3].Value;
95        routine.Parameters.Add(parameterVariable);
96    }
```

Für Parameter gibt es wieder einen eigenen Regex, weil sie aus drei Teilen bestehen: Attribute, Name und Datentyp, wobei die Attribute wieder optional sind. Ein String, der alle Parameter enthält, befindet sich in Gruppe 2 des Match, das man von der Routinen Regex bekommt. Auf den Wert dieser Gruppe wird die Parameter Regex verwendet, um die einzelnen Parameter und ihre Bestandteile zu bekommen. Die Parameter werden in der TeachtalkRoutine als Liste von Variablen gespeichert, weil sie die gleichen Eigenschaften wie Variablen haben. Die Attribute der Parameter werden auch hier wieder in einem String zurückgegeben und müssen getrennt und von Leerzeichen befreit werde. In der folgenden Abfrage wird mit der String Methode IndexOf gefragt, ob das Attribut OPTIONAL gefunden wurde. Diese Methode gibt eine positive Zahl zurück, wenn es gefunden wurde und sonst -1. Der zusätzliche

Parameter StringComparison.OrdinalIgnoreCase, der der IndexOf Methode übergeben wird, stellt sicher, dass das gesuchte Attribut unabhängig von Groß- und Kleinschreibung gefunden wird. Wenn ein Parameter das Optional Attribut hat, muss er beim Routinenaufruf nicht verwendet werden. Ob ein Parameter optional ist, wird nicht nur in den Attributen, sondern auch in der parameterVariable als bool Wert IsOptional gespeichert. Am Schluss werden der Name und Typ des Parameters aus den Regex Gruppen 2 bzw. 3 gespeichert und der Parameter zu der Parameterliste, die sich in TeachtalkRoutine befinden, hinzugefügt.

3.3.4 Parse Variables

Die Variablen werden in teachtalk in einem Block deklariert, der mit VAR beginnt und mit END_VAR endet. Die Attribute können direkt für den ganzen Block angegeben werden, beispielsweise VAR USER PRIVATE. Eine Variable kann aber auch eigene Attribute besitzen, die nicht am Beginn des Blocks angegeben werden.

Eine TeachtalkVariable besitzt folgende Eigenschaften:

- Name
- Type
- Attributes
- IsOptional
  wird nur für Routinenparameter verwendet, die auch als Variable gespeichert werden
- Value
  wird nur für Konstanten verwendet

Für die Variablen werden zwei Regular Expressions benötigt: Eine für alle Variablenblöcke und eine für die Variablen und Eigenschaften der Variablen innerhalb der Blöcke.

```
101    public List<TeachtalkVariable> parseVariables ()
102    {
103        List<TeachtalkVariable> variables = new List<TeachtalkVariable>();
104        string fileNoRoutines = RegularExpressions.routine.Replace(FileContent,
       "");
105        var matches = RegularExpressions.varBlock.Matches(fileNoRoutines);
106        foreach (Match match in matches)
107        {
108            var variablesMatches =
       RegularExpressions.variables.Matches(match.Groups[3].Value);
109            foreach (Match variableMatch in variablesMatches)
110            {
111                TeachtalkVariable variable = new TeachtalkVariable();
112                variable.Attributes.AddRange(match.Groups[1].Value.Split(' '));
113                variable.Attributes.AddRange(variableMatch.Groups[1].Value.
       Split(' '));
114                foreach (string attribute in variable.Attributes.ToList())
115                {
116                    if (String.IsNullOrWhiteSpace(attribute))
117                    {
118                        variable.Attributes.Remove(attribute);
119                    }
120                    else
121                    {
122                        attribute.Trim();
123                    }
124                }
125                variable.Name = variableMatch.Groups[2].Value;
126                variable.Type = variableMatch.Groups[3].Value;
127                variables.Add(variable);
128            }
129
130        }
131        return variables;
132    }
```

Zu Beginn werden alle Routinen mit der Routinen-Regex und der Replace Methode durch einen leeren String ersetzt, damit nur die Variablenblöcke auf Klassenebene gefunden werden und nicht die Variablen, die innerhalb von Routinen deklariert werden und nur innerhalb dieser verwendet werden können. Auf diesen String ohne Routinen wird die Regex varBlock angewendet, die alle Variablenblöcke zurückgibt. In der Schleife werden alle Variablenblöcke durchlaufen und darauf die Regex variables angewendet, die die einzelnen Variablen innerhalb der Blöcke matcht. Diese werden auch in einer Schleife durchlaufen. Bei den Attributen werden zuerst die Attribute des Variablenblocks hinzugefügt. Diese befinden sich in Gruppe 1 der varBlock Regular Expression. In der darauffolgenden Zeile werden die Attribute der einzelnen Variable hinzugefügt, die sich in Gruppe 1 der variables Regex befinden. Beide Gruppen geben einen Attributstring mit möglicherweise mehreren Attributen zurück, der wieder mit Split getrennt werden muss. In der darauffolgenden foreach Schleife werden wieder alle unerwünschten Leerzeichen und leere Strings entfernt.

Am Schluss wird für die Variablen noch der Name und Datentyp aus Gruppe 2 und 3 der variables Regex hinzugefügt

3.3.5 Parse Constants

Konstanten sind ähnlich wie Variablen mit dem Unterschied, dass sie in einem Konstantenblock deklariert werden, der mit CONSTANT beginnt und mit END_CONSTANT endet. Außerdem bekommen sie beim Erstellen einen Wert zugewiesen, der sich nicht mehr ändern kann. In TeachtalkClass werden die Konstanten in einer eigenen Liste gespeichert. Für die Konstanten werden wie bei den Variablen zwei Regular Expressions verwendet: eine für den Konstantenblock und eine für die einzelnen Konstanten und deren Eigenschaften.

```
134    public List<TeachtalkVariable> parseConstants ()
135    {
136        List<TeachtalkVariable> constants = new List<TeachtalkVariable> ();
137        string fileNoRoutines = RegularExpressions.routine.Replace(FileContent,
       "");
138        var matches = RegularExpressions.constBlock.Matches(fileNoRoutines);
139        foreach (Match match in matches)
140        {
141            var constantMatches =
       RegularExpressions.constant.Matches(match.Value);
142            foreach (Match constantMatch in constantMatches)
143            {
144                TeachtalkVariable constant = new TeachtalkVariable();
145                constant.Attributes.AddRange(match.Groups[1].Value.Split(' '));
146                constant.Attributes.AddRange(constantMatch.Groups[1].Value.
       Split(' '));
147                foreach (string attribute in constant.Attributes.ToList())
148                {
149                    if (String.IsNullOrWhiteSpace(attribute))
150                    {
151                        constant.Attributes.Remove(attribute);
152                    }
153                    else
154                    {
155                        attribute.Trim();
156                    }
157                }
158                constant.Name = constantMatch.Groups[2].Value;
159                constant.Type = constantMatch.Groups[3].Value;
160                constant.Value = constantMatch.Groups[4].Value;
161                constants.Add(constant);
162            }
163
164        }
165        return constants;
166    }
```

Zu Beginn werden wie bei den Variablen die Routinen aus dem String entfernt, in dem nach Konstanten gesucht wird, weil auch Konstanten innerhalb von Routinen deklariert werden können, die außerhalb der Routinen nicht existieren. Für die

Konstantenblöcke wird die Regular Expression constBlock verwendet. Die Konstanten werden in einer Liste von TeachtalkVariable gespeichert. Es werden alle Konstantenblöcke in einer foreach-Schleife durchlaufen und auf jeden Block die Regex constants angewendet, die die einzelnen Konstanten und deren Eigenschaften zurückgibt. Diese werden auch in einer darunterliegenden Schleife durchlaufen. Für Konstanten werden die Attribute entweder am Beginn des Blocks angegeben, zum Beispiel CONSTANT USER oder für die einzelnen Konstanten vor dem Namen angegeben. In der parseConstants Methode werden zuerst die Attribute, die den ganzen Block umfassen zur Attributliste hinzugefügt. Diese bekommt man von Gruppe 1 der constBlock Regex. In der nächsten Zeile werden die Attribute der einzelnen Konstante hinzugefügt, die in Gruppe 1 der constants Regex zu finden sind. Diese Strings müssen mit der String-Split Methode getrennt werden, weil sie mehrere Attribute enthalten können. In der darauffolgenden foreach-Schleife werden alle Attribute durchlaufen und aus der Liste entfernt, wenn sie leer sind, oder mit der String-Trim Methode die Leerzeichen am Beginn und Ende des Strings entfernt. Am Schluss wird der Name und Datentyp zugewiesen, die sich in Gruppe 2 und 3 der constants Regex befinden. Anders als bei Variablen wird bei Konstanten auch der Wert eingelesen und im Objekt der Konstante gespeichert, weil jede Konstante zu Beginn einen Wert hat, der sich im Rest des Programms nicht mehr ändern kann. Diese befindet sich in Gruppe 4 der constant Regex.

3.3.6 Parse Types

In teachtalk können eigene Typen erstellt werden, die aus einer oder mehreren Variablen bestehen können. Wie Variablen und Konstanten werden auch Typen in Blöcken deklariert. Diese beginnen mit TYPE und enden mit END_TYPE.

Ein TechtalkType Objekt besitzt die folgenden Eigenschaften:

- Name
- Attributes
- Declaration

Wegen der unterschiedlichen Arten von Typen und den Unterschieden bei der Deklaration dieser, ist die Methode parseTypes die längste und verwendet vier verschiedene Regular Expressions: Eine für alle Typenblöcke, eine für Strukturtypen,

eine für sonstige Typen, die aus mehreren Elementen bestehen und eine für alle restlichen Typen.

```
168    public List<TeachtalkType> parseTypes ()
169    {
170        List<TeachtalkType> types = new List<TeachtalkType> ();
171        var typeBlocks = RegularExpressions.typeBlock.Matches(FileContent);
172        foreach(Match typeBlock in typeBlocks)
173        {
174            string typeBlockString = typeBlock.Value;
175            var structMatches =
       RegularExpressions.structType.Matches(typeBlockString);
176            foreach(Match structMatch in structMatches)
177            {
178                TeachtalkType teachtalkType = new TeachtalkType();
179                teachtalkType.Name = structMatch.Groups[1].Value;
180                teachtalkType.Declaration = structMatch.Value;
181                teachtalkType.Attributes.AddRange(typeBlock.Groups[1].Value.
       Split(' '));
182                foreach (string attribute in teachtalkType.Attributes.ToList())
183                {
184                    if (String.IsNullOrWhiteSpace(attribute))
185                    {
186                        teachtalkType.Attributes.Remove(attribute);
187                    }
188                    else
189                    {
190                        attribute.Trim();
191                    }
192                }
193                types.Add(teachtalkType);
194            }
```

Im obenstehenden Codeblock ist der erste Teil der parseTypes Methode, für das Parsen von Strukturtypen. Zuerst werden die Typenblöcke mit der Regex typeBlock gesucht. In der Schleife wird der Typblock in einen neuen String typeBlockString geschrieben, weil der Inhalt später verändert wird, ohne dass der FileContent verändert werden soll. Auf diesen String wird der Regex structType angewendet, die alle Strukturen innerhalb eines Typblocks sucht. Für jede Struktur wird am Beginn der nächsten Schleife eine teachtalkType Variable angelegt. Diese bekommt einen Namen und die Deklaration der Struktur zugewiesen, die sich in Gruppe 1 und 0 befinden. Die Attribute bekommt man vom Match der typeBlock Regex weil diese am Beginn des Typblocks für alle Typen angegeben werden. Die Gruppe 1 dieses Match enthält alle Attribute in einem String. Diese werden mit der Split Methode in die einzelnen Attribute getrennt und zur Liste von Attributen des Typs hinzugefügt. In der Schleife wird diese Liste durchlaufen und leere Strings oder Leerzeichen am Beginn und Ende eines Attributs entfernt. Diese Struktur wird dann zu der anfangs erstellten Liste von Typen hinzugefügt

```
195   typeBlockString = RegularExpressions.structType.Replace(typeBlockString,
      string.Empty);
196   var multilineTypeMatches =
      RegularExpressions.multilineType.Matches(typeBlockString);
197   foreach( Match multilineMatch in multilineTypeMatches)
198   {
199       TeachtalkType teachtalkType = new TeachtalkType ();
200       teachtalkType.Name = multilineMatch.Groups[1].Value;
201       teachtalkType.Declaration = multilineMatch.Value;
202       teachtalkType.Attributes.AddRange(typeBlock.Groups[1].Value.Split('
      '));
203       foreach (string attribute in teachtalkType.Attributes.ToList())
204       {
205           if (String.IsNullOrWhiteSpace(attribute))
206           {
207               teachtalkType.Attributes.Remove(attribute);
208           }
209           else
210           {
211               attribute.Trim();
212           }
213       }
214       types.Add(teachtalkType);
215   }
```

Im zweiten Teil der parseTypes Methode werden im typeBlockString mit der structType Regex alle Strukturen durch leere Strings ersetzt. Die wurden bereits im ersten Teil der der Methode zur Liste hinzugefügt und sollen nicht noch einmal gematcht werden. Im veränderten String werden hier mit der Regex multilineType alle Typen gesucht, die aus mehreren Elementen bestehen, aber keine Strukturen sind. Das können zum Beispiel Enumerationstypen sein. In einer foreach Schleife über alle Matches wird eine neue TeachtalkType Variable erstellt und der Name, die Deklaration und die Attribute des Typen zugewiesen. Dieser Vorgang ist gleich wie im ersten Teil der Methode bei den Strukturen. Zum Schluss wird der teachtalkType zu der Liste von Typen zugewiesen, die im ersten Teil der Methode erstellt wurde.

```
216            typeBlockString =
     RegularExpressions.multilineType.Replace(typeBlockString, string.Empty);
217            var typeMatches = RegularExpressions.type.Matches
     (typeBlockString);
218            foreach(Match typeMatch in typeMatches)
219            {
220                TeachtalkType teachtalkType = new TeachtalkType();
221                teachtalkType.Name = typeMatch.Groups[2].Value;
222                teachtalkType.Declaration= typeMatch.Value;
223                teachtalkType.Attributes.AddRange(typeBlock.Groups[1].Value.
     Split(' '));
224                foreach (string attribute in teachtalkType.Attributes.ToList())
225                {
226                    if (String.IsNullOrWhiteSpace(attribute))
227                    {
228                        teachtalkType.Attributes.Remove(attribute);
229                    }
230                    else
231                    {
232                        attribute.Trim();
233                    }
234                }
235                types.Add(teachtalkType);
236            }
237
238        }
239        return types;
240    }
```

Im dritten Teil der parseTypes Methode werden vom typeBlockString von dem im zweiten Teil bereits die Strukturen entfernt wurden, auch alle mehrteiligen Typen entfernten, die bereits geparst wurden. Dafür wird die multilineType Regex und die Replace Methode der Regex Klasse verwendet. Um alle restlichen Typen zu bekommen, wird die Regex type verwendet. In der Schleife über alle Matches wird ein neuer teachtalkType erstellt, dem Name und Deklaration zugewiesen werden. Der Name ist dieses Mal in der zweiten Gruppe der type Regex. Das Parsen der Attribute ist wieder gleich wie bei den Strukturen und mehrteiligen Typen am Anfang der parseTypes Methode

4 Baum Ansicht

Es soll eine Benutzeroberfläche (GUI) für die Ansicht eines Strukturbaumes welcher TeachTalkKlassen, TeachTalkRoutinen, TeachTalkTypes und TeachTalkVariablen enthalten, erstellt werden. Eine Baumstruktur ist eine Ansammlung von Elementen, welche als Knoten bezeichnet werden. Jeder Knoten ist mit einen anderen Unterknoten verbunden. Der Wurzelknoten, welcher auch Stamm oder Wurzel genannt wird, soll die Startebene des Baumes sein. Von dem Wurzelknoten breiten sich Verbindungswege zu den anderen Knoten, ähnlich den Ästen eines Baumes. Die Knoten repräsentieren verschiedene Arten von Elementen, darunter TeachTalk-Klassen, -Routinen, -Typen und -Variablen.

Das Ganze wurde in C# entwickelt, unter der Verwendung des WPF-Frameworks. Zusätzlich zur Anzeige der Baumstruktur wurde noch eine Suchfunktion implementiert, welche das Suchen von Knotenpunkten um ein Vielfaches erleichtert, da tausende von Knotenpunkten vorhanden sein können.

(vgl. Baumstruktur, kein Datum)

```
20    public partial class MainWindow : Window
21        {
22            private Parser parser = new Parser();
23            public MainWindow()
24            {
25                InitializeComponent();
26                parser.Parse(new DirectoryInfo(Path));
27                parser.Tree(); // Baumstruktur erstellen
```

In der WPF Anwendung wird erstmals ein neues Parser Objekt erstellt, sodass man auf die dazugehörigen Methoden (Parse, Tree,…) zugreifen kann. InitializeComponent(); ist eine Methode, die automatisch vom Designer generiert wird, wenn man eine Benutzeroberfläche(GUI) visuell in Visual Studio entwirft. Sie initialisiert und konfiguriert alle visuellen Elemente (zum Beispiel Buttons, Textboxen, Labels etc.), welche man im Fenster platziert hat.

In Zeile 26 wird die Methode Parse() mit dem Pfad, in dem sich alle TeachTalk Dateien befinden, aufgerufen. Die DirectoryInfo Klasse ermöglicht es, Informationen über ein Verzeichnis auf dem Dateisystem zu erhalten und das Erstellen, Löschen oder auch Navigieren durch Verzeichnisse zuzulassen.

```
7      public void Parse(DirectoryInfo currentDir)
8              {
9                  if(TeachtalkFiles == null)
10                 {
11                     TeachtalkFiles = new List<TeachtalkFile>();
12                 }
13                 var files = currentDir.GetFiles("*.tts");
14
15                 foreach(var file in files)
16                 {
17                     var ttf = new TeachtalkFile(file);
18                     TeachtalkFiles.Add(ttf);
19                 }
20                 var directories = currentDir.GetDirectories();
21
22                 foreach(var directory in directories)
23                 {
24                     Parse(directory);
25                 }
26             }
```

Die Methode Parse() durchläuft rekursiv alle Dateien im Verzeichnis und in allen Unterverzeichnissen, die die Dateiendung „.tts“ haben. Für jede gefundene TeachTalk Datei wird ein Objekt(TeachtalkFile) erstellt, wie es in Zeile 17 zu sehen ist. Danach wird dieses Objekt in die Liste TeachTalkFiles gespeichert. Abschließend werden alle Unterverzeichnisse des aktuellen Verzeichnisses rekursiv durchlaufen, um alle TeachTalk Dateien und Unterverzeichnisse im gesamten Verzeichnisbaum zu durchsuchen.

Nun wird als nächstes die Methode Tree() aufgerufen.

```
31    public void Tree()
32            {
33                foreach(var file in TeachtalkFiles)
34                {
35                    foreach(var child in TeachtalkFiles)
36                    {
                      if(child.TTClass.ParentString.Equals(file.TTClass.Name))
38                    {
39                        file.TTClass.addChild(child.TTClass);
40                    }
41                    }
42                }
43            }
```

Hier wird eine hierarchische Struktur erstellt, indem man durch jedes Element in der Liste „TeachtalkFiles“ iteriert. Für jedes Element in der Liste wird erneut durch die gesamte Liste iteriert, um zu überprüfen, ob das aktuelle Element ein Child eines anderen Elements ist. Falls das zutrifft, wird das aktuelle Element als Child zum anderen Element gespeichert.

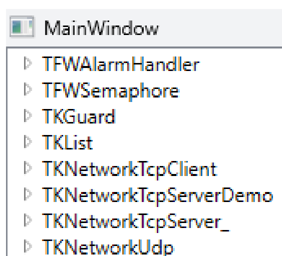
4.1 TeachTalk File

In der WPF-Anwendung geht es nun mit einer foreach Schleife weiter. Für jedes File in der Liste TeachtalkFiles welche vom Parser kommen, werden die gleichen Schritte ausgeführt.

```
28    foreach(var file in parser.TeachtalkFiles)
29            {
30                var item = new TreeViewItem();
31                item.Header = file.TTClass.Name;
32
```

Ein TreeViewItem ist ein grafisches Element, welches in Baumstrukturen verwendet wird, um Daten hierarchisch darzustellen. Vergleichen kann man dies mit einer Ordnerstruktur im Datei-Explorer. Ein TreeViewItem kann selbst Kinder-TreeViewItems haben, die wiederum untergeordnete Elemente darstellen.

In Zeile 31 weisen wir den TreeViewItem einen Namen zu, welcher dann im Fenster angezeigt wird. Da die oberste Schicht die TeachTalk Klasse ist, weisen wir dem Item den Namen der Klasse zu. Das sieht dann wie folgt aus:



4.2 Children

Die nächste Schicht zeigt die Children der Parent Klasse an, also, welche Klasse von dieser TeachTalk Klasse erbt. Erben bedeutet, dass eine Klasse die Eigenschaften und Methoden einer anderen Klasse übernimmt, um sie dann zu verwenden oder zu erweitern.

```
32
33                        //children
34                      if(file.TTClass.Children.Count > 0)
35                      {
36                          var itemChildren = new TreeViewItem();
37                          itemChildren.Header= "Children";
38                          foreach(var child in file.TTClass.Children)
49                          {
40                              var children = new TreeViewItem();
41                              children.Header = child.Name;
42                              itemChildren.Items.Add(children);
43                          }
44                          item.Items.Add(itemChildren);
45                      }
46                      else
47                      {
```

```
48                      var itemChildren = new TreeViewItem();
49                      itemChildren.Header = "Children sind nicht vorhanden";
50                      item.Items.Add(itemChildren);
51                      }
52
```

Bei der If Abfrage wird zunächst geschaut, ob die Klasse mehr als 0 Childen besitzt. Wenn das zutrifft, wird ein eigener Überpunkt mit den Namen „Children“ erstellt. In der darauffolgenden foreach Schleife werden alle Children der Parent Klasse durchlaufen. Für jedes Child wird ein neues TreeViewItem erstellt. Dem Header wird der Name des Childs zugewiesen. Dieses TreeViewItem wird nun dem TreeViewItem der Parent Klasse angehängt.

Falls die Parent Klasse keine Children besitzt, wird bei dem else Block ein TreeViewItem erstellt, welches dann in der Benutzeroberfläche „Children sind nicht vorhanden“ ausgibt.

Diese Zwei Fälle sehen wie folgt aus:

file.TTClass.Children.Count > 0

```
◢ OAnalogIn
   ◢ Children
        OUserAnalogIn
        OAnalogInMon
```

file.TTClass.Children.Count == 0

```
◢ TFWAlarmHandler
     Children sind nicht vorhanden
```

4.3 Routinen

Eine weitere Unterschicht von den TeachTalk Klassen sind Routinen. Routinen besitzen einen Namen, ReturnType, Attribute und eine Liste von TeachtalkVariablen Parameter.

```
53                      //routines
54                      if (file.TTClass.Routines.Count > 0)
55
56                      {
57                          var itemRoutines = new TreeViewItem();
58                          itemRoutines.Header = "Routines";
59                          foreach (var routine in file.TTClass.Routines)
60                          {
61                              var routines = new TreeViewItem();
62                              routines.Header = routine.Name;
63                              itemRoutines.Items.Add(routines);
64
```

Zuerst wird erstmals überprüft, ob die Anzahl von Routinen einer Klasse mehr als 0 ist. Danach wird wieder ein TreeViewItem mit dem Namen „Routines“ erstellt. Unter dieser Schicht sind wieder alle Routinen einer Klasse untergeordnet. Diese werden mit dem Namen der Routine ausgegeben.

4.3.1 Return Type

Um den Unterpunkt ReturnType von einer Routine anzulegen, wird auch hier erstmals überprüft, ob die Routine überhaupt ReturnTypes aufweist.

```
65          //Return Type
66          if (routine.ReturnType.Length == 0)
67          {
68            var routineReturnType = new TreeViewItem();
69            routineReturnType.Header = "Return Parameter existiert nicht";
70            routines.Items.Add(routineReturnType);
71          }
72          else
73          {
74            var routineReturnType = new TreeViewItem();
75            routineReturnType.Header="Return Parameter: "+routine.ReturnType;
76              routines.Items.Add(routineReturnType);
77          }
```

Wenn das zutrifft, wird der „Return Parameter: “ + der ReturnType ausgegeben. Wenn es keine Return Types aufweist, gibt es in der Benutzeroberfläche wieder nur den Punkt „Return Parameter existiert nicht“.

TFWAlarmHandler
Children sind nicht vorhanden
Routines
GetFirst
Return Parameter: BOOL

4.3.2 Attribut

Die Logik der Ausgabe des zweiten Unterpunkt Attribute unterscheidet sich darin, dass eine Routine mehrerer Attribute beinhalten können.

```
78            //Attribute
79            var routineAttributes = new TreeViewItem();
80            if (routine.Attributes.Count > 0)
81            {
```

```
82                  routineAttribute.Header = "Attributes";
83              }
84              else
85              {
86                  routineAttribute.Header = "Attributes existieren nicht";
87              }
88              foreach(var attribute in routine.Attributes)
89              {
90                  var routineAttribute = new TreeViewItem();
91                  routineAttribute.Header = "Attribute: " + attribute;
92                  routineAttributes.Items.Add(routineAttribute);
93              }
94
```

4.3.3 Parameter

Ähnlich sieht es bei dem dritten Unterpunkt Parameters aus. Ein Parameter besitzt einen Namen, Type, Attribute und isOptional. Eine Routine kann mehrere Parameter besitzen, da diese wie oben beschrieben in einer Liste gespeichert sind.

Anfangs wird geprüft, ob sich überhaupt Parameter in der Liste befinden. Wenn mindestens ein Parameter in der Liste gespeichert ist, wird der Überpunkt Parameters erstellt, ist das aber nicht der Fall, wird nur „Parameter existieren nicht“ ausgegeben.

```
96              //Parameters
97              var ParameterItem = new TreeViewItem();
98              if (routine.Parameters.Count > 0)
99              {
100                 ParameterItem.Header = "Parameters";
101             }
102             else
103             {
104             ParameterItem.Header = "Parameter existieren nicht";
105             }
106             routines.Items.Add(ParameterItem);
```

Darauffolgend wird mithilfe einer foreach Schleife, alle Parameter in der Liste routine.Parameters ermittelt. Unter dem Unterpunkt Parameters wird nun ein neues TreeViewItem mit dem Namen des Parameters eingefügt.

```
108         foreach (var Parameter in routine.Parameters)
109         {
110
111             //Parameter Name
112             var routineParameter = new TreeViewItem();
113             routineParameter.Header = Parameter.Name;
114             ParameterItem.Items.Add(routineParameter);
115
```

4.3.4 Attribut

Das Gleiche wird auch bei den weiteren Unterpunkten Type und isOptional gemacht. Bei Attribute von den Routines ist der Ablauf etwas anders.

Zuerst wird wie üblich geprüft, ob sich überhaupt Attribute in der Routine befinden.

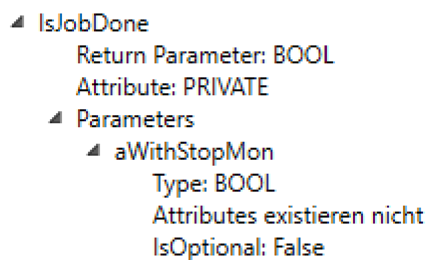
```
119         var routineParameterAttribute = new TreeViewItem();
120
121         if (Parameter.Attributes.Count > 0)
122         {
123            routineParameterAttribute.Header = "Attributes";
124         }
125         else
126         {
127           routineParameterAttribute.Header = "Attributes existieren nicht";
128         }
129            routineParameter.Items.Add(routineParameterAttribute);
```

Nach der If Abfrage wird in der Liste Parameter.Attributes nach allen Attributen gesucht. Für jedes Attribute in Parameter.Attributes wird erstmals ein neues TreeViewItem erstellt. Dann wird der Name des TreeViewItems auf den Namen des

Attributes gesetzt. Schlussendlich wird dieses TreeViewItem unter das TreeViewItems des Überpunktes „Attributes:“ eingefügt.

```
130         foreach (var attribute in Parameter.Attributes)
131         {
132          var routineParameterAttributes = new TreeViewItem();
133          routineParameterAttributes.Header = "Attribute: " + attribute;
134          routineParameterAttribute.Items.Add(routineParameterAttributes);
135         }
136
```

In der Benutzeroberfläche sieht das wie folgt aus:



4.4 Typ

Weiter geht es mit den Types. Types besitzen einen Namen, Deklaration und eine Liste von Attributen. Anfangs wird geprüft, ob überhaupt Typen in der Liste sich befinden. Wenn das der Fall ist, wird ein Knotenpunkt mit den Namen „Types“ erstellt. Unter diesen Knotenpunkt werden zwei Unterknotenpunkte mit den Namen Attribute und Declaration erstellt. Da Typen mehrere Attribute besitzen können, wird hier wieder zuerst geprüft, ob die Liste nicht leer ist. Danach werden alle Attribute in der Liste Attributes als Knotenpunkt ausgegeben. Der Name des Knotenpunktes ist der gleiche Name, den das Attribut besitzt. Ein Typ kann nur eine Deklaration besitzen, deshalb ist diese Implementierung etwas einfacher. Hier wird einfach die Deklaration vom Typen als Unterknotenpunkt gespeichert. Der Knotenpunktname ist

identisch mit der Deklaration. Falls aber kein Typ in der Liste existiert, wird dieser ganze Vorgang übersprungen und ein einfacher Knotenpunkt mit dem Namen „Types existieren nicht" ausgegeben. Beide Fälle sehen in der Benutzeroberfläche folgendermaßen aus:

Fall 1 | Die Liste ist leer:

```
◢ TFWAlarmHandler
      Children sind nicht vorhanden
   ▷ Routines
      Types existieren nicht
```

Fall 2 | Die Liste beinhaltet Typen

```
◢ OAxisChannelSigmatek1
      Children sind nicht vorhanden
   ▷ Routines
   ◢ Types
      ▷ tConfiguratedAxis
      ▷
      ◢ eUsedAxisX
         ◢ Attributes
               Attribute: GLOBAL
               Attribute: USER
            Declaration:      eUsedAxisX := 0,
```

4.5 Variablen

Um Variablen im Strukturbaum anzeigen zu können, wird nach der Überprüfung, ob die Liste von Variablen leer ist, ein neues TreeViewItem erstellt. Der Name des TreeViewItems wird auf „Variables" gesetzt. Als Unterknotenpunkt wird der Name der Variable eingefügt. Dieser Knotenpunkt hat wieder mehrere Unterknotenpunkte wie Type und Attribute. Da Variablen nur einen Type besitzen können, wird dieser mit den Typennamen als Unterknotenpunkt gespeichert. Anders sieht es bei den Attributen aus. Eine Variable kann wieder mehr Attribute besitzen, welche in einer Liste gespeichert sind. Die Überprüfung sieht wie folgt aus:

```
221                 //Attribute
222                 var variablesAttribute = new TreeViewItem();
223                 if(variable.Attributes.Count > 0)
224                 {
225                    variablesAttribute.Header = "Attributes";
226                 }
227                 else
228                 {
229                    variablesAttribute.Header = "Attributes existieren nicht";
230                 }
231                 variables.Items.Add(variablesAttribute);
```

```
232
```

In Zeile 222 wird das TreeViewItem erstellt und in Zeile 223 erfolgt die Überprüfung der Liste von Attributen. Falls die Liste Attribute beinhaltet, geht es in der Zeile 225 weiter. Hier wird der Name des Knotenpunktes auf „Attributes“ gesetzt. Dann wird der else Block übersprungen (Zeile 229). Dieser else Block wird nur berücksichtigt, wenn die Liste leer ist und somit die If Abfrage in den else Block springt. Abschließend wird dieses TreeViewItem mit dem Namen „Attributes“ oder „Attributes existieren nicht“ als Unterknotenpunkt der Variable eingefügt.

Jetzt wurde der Knotenpunkt erstellt, nun müssen nur mehr die Attribute darunter hineingespeichert werden. Dies erfolgt in folgender foreach Schleife.

```
233                 foreach(var attribute in variable.Attributes)
234                 {
235                    var VariableAttribute = new TreeViewItem();
236                    VariableAttribute.Header = attribute;
237                    variablesAttribute.Items.Add(VariableAttribute);
238
239                 }
240
```

(Zeile 233) Für jedes Attribut in der Liste variable.Attributes wird folgendes gemacht:

Zuerst wird ein Knotenpunkt also ein TreeViewItem erstellt. Darauffolgend wird der Header, also der Name, der in der Benutzeroberfläche angezeigt wird auf den Namen des Attributes gesetzt. Schlussendlich wird das gerade erstellte TreeViewItem unter den Knotenpunkt „Attributes“ gespeichert.

In der Benutzeroberfläche sieht die Ansicht so aus:

- OAxisChannelSigmatek1
  - Children sind nicht vorhanden
  - Routines
  - Types
  - Variables
    - Sigmatek
      - Type: Achsrekonfiguration implementieren
      - Attributes
        - PRIVATE
        - ODO
    - axisLink
    - args
    - chg
  - Constants existieren nicht

4.6 Konstanten

Konstanten besitzen einen Typ und eine Liste von Attributen. Zuerst wird die Liste von Konstanten geprüft, ob diese Konstanten beinhaltet. Wenn die Liste gefüllt ist, wird ein Knotenpunkt mit dem Namen „Constants“ erstellt, aber falls die Liste leer ist, wird der Knotenpunkt mit dem Namen „Constants existieren nicht“ ausgegeben. Angenommen die Liste von Konstanten ist nicht leer, wird ein Knotenpunkt mit dem Konstantennamen unter „Constants“ gespeichert. Unter den Konstantennamen wird der passende Typ dazu als Unterknotenpunkt ausgegeben. Um die Liste der Attributen auszugeben, wird die Liste geprüft, ob sie gefüllt ist. Ist dies der Fall wird ein Knoten mit dem Header(Namen) „Attributes“ ausgegeben. Dann wird für jedes Attribute in der constant.Attributes Liste ein eigener Knoten, mit dem passenden Attributnamen erstellt. Schlussendlich werden diese Knoten zu den Überknoten hinzugefügt.

In der Benutzeroberfläche sieht das folgendermaßen aus:

- OAntiCrashSystem
  - Children sind nicht vorhanden
  - Routines
  - Types existieren nicht
  - Variables
  - Constants
    - cMinVelocityDiff
      - Type: REAL
    - cMinDistanceDiff
      - Type: REAL
    - cMaxMcuUpdateErrorCount
    - cMinMoveOverride

4.7 Suchfunktion

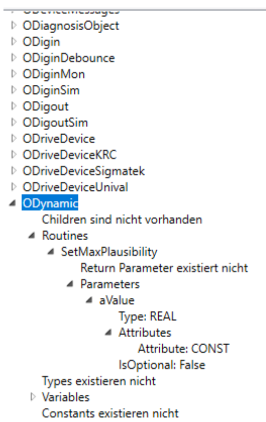
4.7.1 Benutzeroberfläche

Um den Nutzern eine leichtere Bedienung der Benutzeroberfläche zu gewähren, wurde noch eine Suchfunktion für Knotenpunkten implementiert. Im Fenster wird eine einfache TextBox und ein Button angezeigt. In die TextBox wird der Name des gesuchten Knotenpunktes geschrieben und mit dem Button, wird dieser Knoten gesucht. Wenn dieser Knoten gefunden wurde, springt die Anzeige zu der Position, wo sich dieser befindet.

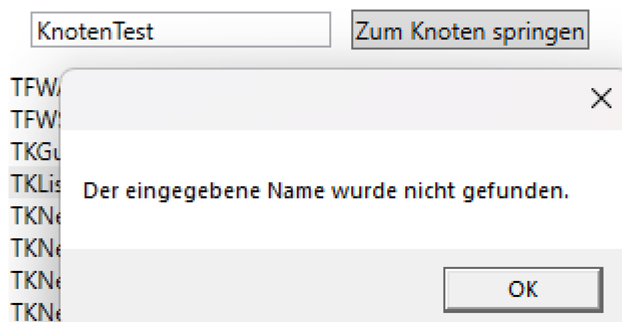
In der Benutzeroberfläche sieht diese Funktion wie folgt aus:

Angenommen mal will nach den Knoten ODynamic suchen. Der Nutzer gibt diesen Knotennamen in die TextBox ein und drückt auf „Zum Knoten springen“.

Nach dem Buttonklick springt die Ansicht zu dem gesuchten Knoten „ODynamic".



Falls der eingegebene Knoten nicht gefunden wird, sieht das folgendermaßen in der Benutzeroberfläche aus:



4.7.2 Programmcode

4.7.2.1 XAML

Im Code sieht das ganze so aus:

```
<Grid>
    <TreeView x:Name="TreeViewFiles">
        <StackPanel Grid.Row="1" Orientation="Horizontal" Margin="5">
            <TextBox x:Name="textBoxName" Width="150" Margin="5" VerticalAlignment="Center" />
            <Button Content="Zum Knoten springen" Click="Button_Click" Margin="5"
             VerticalAlignment="Center" />
        </StackPanel>
    </TreeView>
</Grid>
```

Dieser Code befindet sich in der MainWindow.xaml Klasse. Hier wird ein TreeView erstellt, welcher die Baumstruktur darstellt. Innerhalb des TreeView befindet sich ein StackPanel. Ein StackPanel ist ein Layout-Container, der dazu dient, Elemente in einer bestimmten Ausrichtung anzuordnen. In diesen Fall ist es die TextBox und der Button welche horizontal angeordnet sind. Danach wird die TextBox mit dem Namen textBoxName erstellt, wo der Nutzer den Suchbegriff eingeben kann. In der nächsten Zeile ist der Button, welcher definiert, dass die Methode „Button_Click“ aufgerufen wird, falls dieser geklickt wird (Click="Button_Click").

4.7.2.2 Button_Click Methode

Die Button_Click Methode sieht folgendermaßen aus:

```
302 private void Button_Click(object sender, RoutedEventArgs e)
303 {
304     // Namen aus dem Textfeld abrufen
305     string gesuchterName = textBoxName.Text;
306
307     // Methode aufrufen, um den Knotenpunkt im TreeView zu finden
308     TreeViewItem gesuchterKnoten = FindNodeByName(TreeViewFiles.Items,
309     gesuchterName);
310
311     // Wenn der Knotenpunkt gefunden wurde, scrollen Sie ihn in den
312     sichtbaren Bereich
313     if (gesuchterKnoten != null)
314     {
315         gesuchterKnoten.BringIntoView();
316     }
317     else
318     {
319         MessageBox.Show("Der eingegebene Name wurde nicht gefunden.");
320     }
321 }
```

Zuerst wird der Name, der im Textfeld eingegeben wurde, abgerufen und in der Variable „gesuchterName“ gespeichert. Danach wird die Methode

„FindNodeByName“ aufgerufen, um den Knoten im TreeView-Element mit dem eingegebenen gesuchten Namen zu finden. Diese Methode erhält die Parameter: Liste der Elemente im TreeView und den gesuchten Namen. Wenn der gesuchte Knoten gefunden wurde, wird dieser Knoten in den sichtbaren Bereich gescrollt. Falls der Knoten nicht gefunden werden konnte, wird die Meldung "Der eingegebene Name wurde nicht gefunden." in einer MessageBox angezeigt.

4.7.2.3 FindNodeByName Methode

Die FindNodeByName Methode sieht wie folgt aus:

```
322 private TreeViewItem FindNodeByName(ItemCollection nodes, string name)
323    {
324        foreach (object obj in nodes)
325        {
326            if (obj is TreeViewItem node)
327            {
328                if (node.Header.ToString() == name)
329                {
330                    return node;
331                }
332                // Überprüfen, ob der aktuelle Knoten Unterknoten hat
333                if (node.HasItems)
334                {
335                 // Unterknoten rekursiv durchsuchen
336                 TreeViewItem foundNode = FindNodeByName(node.Items, name);
337                 if (foundNode != null)
338                 {
339                    return foundNode;
340                 }
341                }
342            }
343        }
```

```
344         return null;
345     }
```

Die Methode “FindNodeByName” sucht rekursiv nach einem Knoten in einem TreeView Element anhand seines Namens und gibt diesen zurück, falss dieser gefunden wird.

Diese Methode bekommt die Parameter nodes und den gesuchten Namen von der Methode Button_Click übergeben. „FindNodeByName“ durchläuft jedes Element in der Liste „nodes“. Da diese Liste eine allgemeine Auflistung von Objekten ist, wird erstmals jedes Elemt auf den Datentyp „TreeViewItem“ überprüft, da dies die Klasse der Knoten im TreeView darstellt. Wenn das aktuelle Element ein TreeViewItem ist, wird der Name des TreeViewItems mit den gesuchten Namen verglichen. Falls dieser übereinstimmt, wird dieser Knoten zurückgegeben. Falls der aktuelle Knoten Unterknoten hat (wird mit der Methode „HasItems“ überprüft) wird die Methode „FindNodeByName“ rekursiv aufgerufen, um die Unterknoten zu durchsuchen. Falls der gesuchte Knoten in den Unterknoten gefunden wird, wird dieser Knoten zurückgegeben. Falls kein Knoten mit den gesuchten Namen übereinstimmen, wird null zurückgegeben.

5 Unit Test

Die ganze Anwendung wird mit Unit Tests getestet. Zuerst musste man zu dem bestehenden Projekt ein neues Unit Test Projekt hinzufügen. Zu diesen Unit Test Projekt, wurde ein Verweis auf die Parser Anwendung erstellt, sodass die Unit Test Klassen auf die Parser Methoden und Klassen zugreifen kann.

Eine Unit Test Klasse erkennt man daran, dass über den Klassen Namen [TestClass] steht.

```
    [TestClass]
    public class UnitTest1
    {
```

Um die ganzen TeachTalk Dateien nicht in jeder Testmethode erneut parsen zu müssen, wurde die Methode „ClassInitialize“ erstellt. Diese wird vor allen Testmethoden aufgerufen, um die Teachtalk Dateien zu parsen und diese in Objekte/Listen zu speichern.

```
16        [ClassInitialize]
17        public static void ClassInitialize(TestContext context)
18        {
19            parser = new Parser();
20            parser.Parse(new DirectoryInfo("C:\\Users\\Niklas\\Documents\\Diplomarbeit2023\\WorkSpace"));
21            parser.Tree();
22            teachtalkFiles = parser.TeachtalkFiles;
23            foreach (var file in teachtalkFiles)
24            {
25                if (file.TTClass.Name.Equals("OKinematicSync", StringComparison.OrdinalIgnoreCase))
26                {
27                    teachtalkFile = file;
28                }
29            }
```

Hier werden in die Routinen, Variablen, Typen und Klassen des TeachTalk files „OkinematicSync“ in das Testobjekt eingefügt.

Eine Testmethode überprüft, ob die Routine „NEW“ keinen Parameter aufweist. Dazu muss man zuerst in der Datei nachschauen, ob Parameter in der Routine vorhanden sind. In diesen Fall gibt es keinen Parameter, somit ist der erwartete Wert 0. Um den tatsächlichen Wert herausfinden zu können, wird mit Hilfe einer foreach Schleife, alle Routinen des TeachTalk Files durchlaufen. Mit einer If Abfrage überprüft man, ob der aktuelle Routinen Name den Namen der gesuchten Routine(„NEW“) entspricht. Ist das der Fall wird der Test erstellt. Hier gibt man zuerst den erwarteten Wert, dann den tatsächlichen Wert und schlussendlich, die Fehlermeldung, welche beim Scheitern der Test Methode angezeigt wird, ein.

Assert.AreEqual(erwarteter Wert, tatsächlichert Wert, „Fehlermeldung“);

Die Methode sieht wie folgt aus:

```
[TestMethod]
        public void RoutineNoParameter()
        {
         foreach (var routine in teachtalkFile.TTClass.Routines)
         {
           if (routine.Name.Equals("NEW", StringComparison.OrdinalIgnoreCase))
```

```
                {
                    Assert.AreEqual(0,routine.Parameters.Count, "Der Rückgabe
                    Parameter stimmt nicht überein");
                }
            }
        }
```

Um genau das Gegenteil zu prüfen, also ob der Parametername den erwarteten Namen entspricht, wird statt der Anzahl der Parameter der Parametername abgefragt:

```
        [TestMethod]
        public void RoutineParameterName()
        {
            foreach (var routine in teachtalkFile.TTClass.Routines)
            {
                if (routine.Name.Equals("UnMaskServoAxisInMcu",
                    StringComparison.OrdinalIgnoreCase))
                {
                    Assert.AreEqual("aBase", routine.Parameters[0].Name, "Der
                    Rückgabe Parameter(Name) stimmt nicht überein");

                }
            }
        }
```

Hier wird in der Routine „UnMaskServoAxisInMcu“ nach den Parametern gesucht. Beim ersten Parameter wird der Name „aBase“ erwartet. Falls der tatsächliche Name „aBase“ ist, wird die Methode mit einem grünen Haken gekennzeichnet. Wenn aber der tatsächliche Name nicht mit den erwarteten Namen nicht übereinstimmt, wird die Fehlermeldung ausgegeben.

Bei der Testmethode Variables Attribute, wird geprüft, ob das erste Attribut der Variable „ubError“, „USER“ entspricht. Das sieht wie folgt aus:

```
        [TestMethod]
        public void VariablesAttribute()
        {
            foreach (var variable in teachtalkFile.TTClass.Variables)
            {
                if (variable.Name.Equals("ubError",
```

```
                StringComparison.OrdinalIgnoreCase))
            {
                Assert.AreEqual("USER", variable.Attributes[0], "Das
                Variablenattribut stimmt nicht überein");
            }
        }
    }
```

In dieser Methode wird jede Variable in der Liste von Variablen überprüft, ob diese den Namen „ubError“ besitzt. Danach wird geschaut, ob das erste Attribute der Variable den erwarteten Namen „USER“ besitzt.

Solche Testfälle gibt es für:

- Routine
  - ParameterName,
  - ParameterTyp
  - ParameterAttribut
- Variable
  - Attribut
  - Typ
- Konstante
  - Anzahl
  - Attribut
  - Typ
  - Value
- Typ
  - Anzahl
  - Attribut

6 Technologien

6.1 Windows Presentation Foundation (WPF)

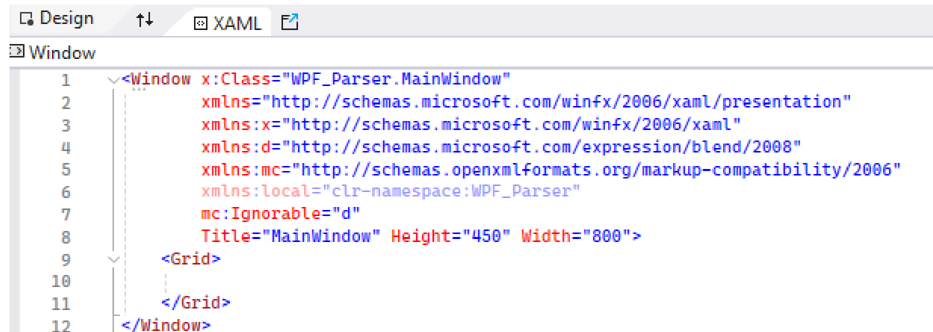
Windows Presentation Foundation ist ein Framework von Microsoft, welches für die Erstellung von Dekstopanwendungen für das Windows-Betriebssystem entwickelt wurde. Es bietet eine leistungsfähige Plattform, auf welcher man grafische Anwendungen erstellen kann. Windows Presentation Foundation basiert auf XAML (Extensible Application Markup Language). XAML ist eine deklarative XML-basierte Sprache, die die Benutzeroberfläche (GUI) definiert. Es trennt die Logik von der Gestaltung.

Wichtige Merkmale von WPF sind:

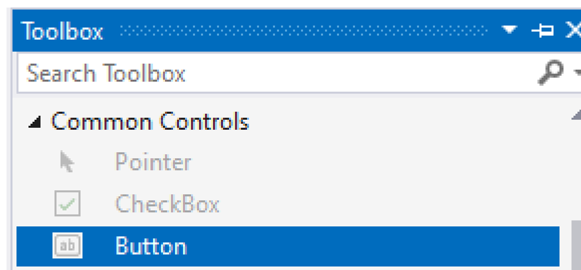
- Data binding:
    - Da sich oftmals Information ändern, die angezeigt werden sollen, verbindet man in WPF die Daten mit der Sicht, sodass diese automatisch aktualisiert werden.
- Anpassungsmöglichkeiten
    - Mit WPF kann man die Ansicht der Anwendung genau so gestalten, wie man es möchte. Farben ändern, Schriftarten anpassen und spezielle Effekte hinzufügen, ist leicht umzusetzen.
- 2D- und 3D-Grafiken.
    - WPF unterstützt die Erstellung von 2D- und 3D Grafiken und bietet eine API für die Manipulation von Grafikelementen. Eine API (Application Programming Interface (Programmierschnittstelle)) ist eine Brücke zwischen zwei Programmen oder Systemen. Es beschreibt, wie Programme miteinander kommunizieren können.
- Skalierbarkeit
    - In WPF kann man Anwendungen an diverse Bildschirmauflösungen und -größen anpassen.
- Integration mit .NET-Technologien

- Windows Presentation Foundation kann mit ASP.Net, Windows Forms und Windows Communication Foundation) integriert warden.

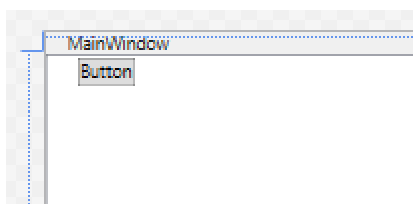
Eine automatisch generierte XAML Datei sieht wiefolgt aus:



Um Buttons, Pointer, TextBoxen usw. anzuzeigen zu können , kann man in Visual Studio aus der Toolbox so ein Element mit Drag and Drop ins Fenster ziehen.



Hier im Fenster ist durch Drag and Drop der Button eingefügt worden.



Wenn man dann diesen Button im Designer doppelklickt, öffnet sich eine Klasse wo folgende Methode sich darin befindet:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

```
}
```

In dieser Methode kann man jetzt festlegen, welche Aktion getätigt werden soll, wenn der Benutzer/Benutzerin diesen Button drückt. Bei Textboxen, ComboBoxen usw. sind solche und noch viele andere Methoden, um jeden Fall zu berücksichtigen, vorhanden.

(vgl. Grafiken und Multimedia, kein Datum)

(vgl. Windows Presentation Foundation, kein Datum)

6.2 Visual Studio

Visual Studio wurde von Microsoft entwickelt. Es ist eine Entwicklungsumgebung, die es Programmierer und Programmiererinnen ermöglicht, in verschiedene Programmiersprachen (zum Beispiel C, C#, Python, TypeScript, SQL, …) zu programmieren. Es ermöglicht Klassische Desktop-Programme, Dynamische Webseiten, Webdienste, Windows-Apps und Apps für Windows Handy zu entwickeln.

(vgl. Hirtenmacher, 2023)

6.3 Unit Test

Komponententests (Unit Tests), dienen zur Überprüfung, ob die von Entwickler und Entwicklerinnen erstellten Komponenten gemäß ihren beabsichtigten Funktionen arbeiten. Die Tests sollten vollständig automatisiert sein, sodass man nur einen Knopf drücken muss. Dank den Tests können Fehler gefunden werden, die man vielleicht spät oder erst gar nicht erkennen würde.

(vgl. Was ist ein Unit-Test?, kein Datum)

6.4 Regex101

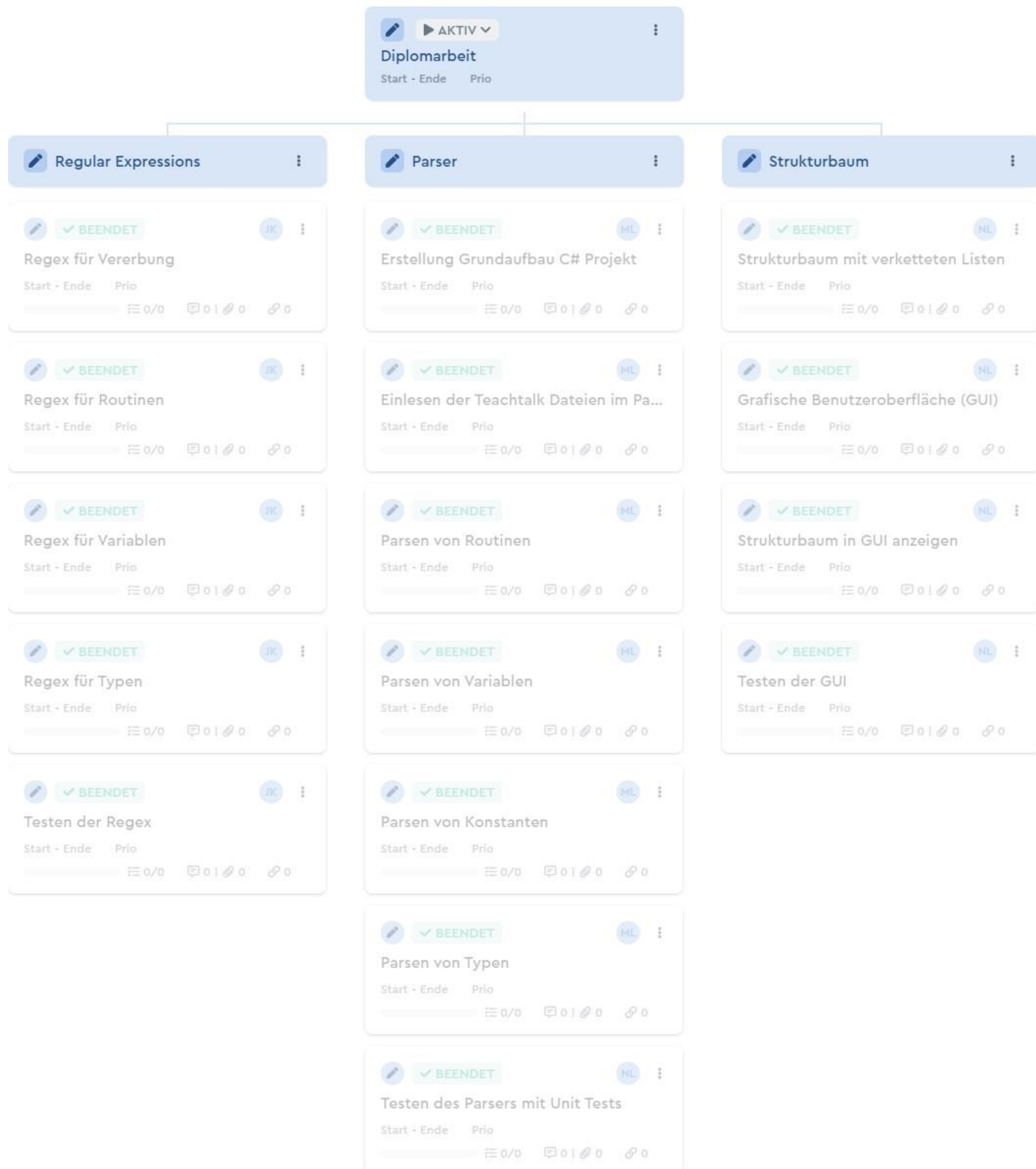
Regex101.com ist eine Website die speziell für Arbeiten mit REGEX entwickelt worden ist. Sie startete einst als Hobbyprojekt doch wurde nach und nach eine der Größten Test Services weltweit. Mit Regex101 kann man nicht nur seine Regex testen, sondern diese auch gleich für die gewünschte Programmiersprache formatieren. Es ist auch möglich in dieser Umgebung Tests für seine regulären ausdrücke zu erstellen

7 Resümee

Unsere Diplomarbeit Objektorientierte Darstellung von Keba Teachtalk Programmen entstand in Zusammenarbeit mit der Firma Engel und wurde entwickelt, um ihren Programmeditor um die Funktion Darstellung der Klassenhierarchien zu erweitern. Durch die großen Überschneidungen unserer Aufgabenbereiche hat sich gezeigt, dass eine gute Kommunikation im Team sehr wichtig ist, damit alle auf dem gleichen Stand sind und die Teile des Projektes ohne Probleme zusammengefügt werden können. Das Parsen einer Programmiersprache hat sich vor allem beim Testen der Regular Expressions als sehr herausfordernd dargestellt, da man alle Regeln der Sprache beachten muss. Beim Testen haben wir oft Testfälle gefunden, die wir vorher nicht berücksichtigt haben, weil es viele Möglichkeiten gibt, wie zum Beispiel eine Routine in Teachtalk aufgebaut sein kann. Zu Beginn der Entwicklung des Strukturbaumes war es sehr kompliziert da wir in diesen Bereich wenig Erfahrungen gehabt haben, und wir mussten uns zunächst einen Überblick über diesen verschaffen. Wir konnten außerdem einige Erfahrungen mit WPF sammeln, um eine Benutzeroberfläche zu gestalten und deren wichtige Methoden zu implementieren.

8 Planung

	Kaltenböck	Leonhardsberger	Lempradl
Planung	M	M	V
Regex	V	M	I
Parser	I	V	I
Strukturbaum	I	M	V
GUI	I	I	V
UNIT Test	I	M	V


AKTIV
Diplomarbeit
Start - Ende Prio
Regular Expressions
Parser
Strukturbaum
BEENDET
Regex für Vererbung
Regex für Routinen
Regex für Variablen
Regex für Typen
Testen der Regex
Erstellung Grundaufbau C# Projekt
Einlesen der Teachtalk Dateien im Pa...
Parsen von Routinen
Parsen von Variablen
Parsen von Konstanten
Parsen von Typen
Testen des Parsers mit Unit Tests
Strukturbaum mit verketteten Listen
Grafische Benutzeroberfläche (GUI)
Strukturbaum in GUI anzeigen
Testen der GUI

Literaturverzeichnis

Baumstruktur. (kein Datum). Abgerufen am 25. 4 2024 von StudySmarter.de: https://www.studysmarter.de/schule/informatik/algorithmen-und-datenstrukturen/baumstruktur/

Grafiken und Multimedia. (kein Datum). Abgerufen am 25. 4 2024 von Microsoft Learn: https://learn.microsoft.com/de-de/dotnet/desktop/wpf/graphics-multimedia

Hirtenmacher, M. (30. 11 2023). *Einfach erklärt: Was ist Visual Studio?* Abgerufen am 25. 4 2024 von Biteno: https://www.biteno.com/was-ist-visual-studio

KEBA. (2010). *teachtalk LanguageReference Programmierhandbuch V1.35.*

Match Class. (kein Datum). Abgerufen am 27. 03 2024 von Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.match?view=net-8.0

Regex Class. (kein Datum). Abgerufen am 27. 03 2024 von Microsoft Learn: https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex?view=net-8.0

Spriesterbach, K. (6. Februar 2023). *Regex: Einführung in reguläre Ausdrücke, Syntax und deren Verwendung*. Abgerufen am 24. März 2024 von Webmasterpro: https://www.webmasterpro.de/coding/einfuehrung-in-regular-expressions/

Was ist ein Unit-Test? (kein Datum). Abgerufen am 25. März 2024 von it-agile.de: https://www.it-agile.de/agiles-wissen/agile-entwicklung/unit-tests

Windows Presentation Foundation. (kein Datum). Abgerufen am 25. 04 2024 von Assecor: https://www.assecor.de/glossar/windows-presentation-foundation-wpf