



TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Praktikumsbericht

Janina Bär

Chemnitz, den 30. September 2005

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	iv
1 Kurzübersicht über das Praktikum	1
2 Praktikumsschwerpunkt 1 – Arbeit an den Symbolen	3
2.1 Problemstellung	3
2.2 Ziel	3
2.3 Vorgehensweise	3
2.4 Realisierung	4
2.4.1 Grundsätzliches	4
Entwicklungsumgebung	4
Symbolbeschreibung	4
Programmbeschreibung	5
2.4.2 Analyse und Lösung der auftretenden Probleme	5
Fall 1: das Shape lässt sich gar nicht oder nicht wie gewünscht drehen .	5
Fall 2: die schon eingebauten Funktionalitäten des Shapes funktionieren nicht	6
Fall 3: das Programm zur automatischen Beschriftung der Klemmen funktioniert bei einigen Shapes nicht	6
2.4.3 Lösung im Detail	7
ShapeSheet-Programmierung	7
Koordinatensysteme	7
Beispiel: Klappe + Motor	8
Joker-Symbole	15
2.5 Ergebnisse	16
2.6 sonstige Programmieraufgaben	26
3 Praktikumsschwerpunkt 2 – Arbeit am Hilfesystem	27
3.1 Problemstellung	27
3.2 Ziel	27
3.3 Vorgehensweise	27
3.4 Realisierung	27

INHALTSVERZEICHNIS

3.4.1	Analyse der Möglichkeiten	27
3.4.2	Erstellung des Hilfesystems	29
	Vorverarbeitung	29
	Designrichtlinien	29
	Vorlagefile	30
	Kategorisierung, Index, Referenzen	30
	Zusammensetzen der Hilfe	30
3.5	Ergebnisse	31
4	Schlussfolgerung	33
A	Praktikumsbescheinigung	37

Abbildungsverzeichnis

2.1	das Symbol „Klappe + Motor“	9
2.2	die verschiedenen Luftklappentypen	10
2.3	Motorposition in Abhängigkeit vom Drehwinkel	12
2.4	die zwei Joker-Symbole	15
2.5	Schablone: CT_BAUGRUPPEN.VSS	17
2.6	Schablone: CT_ELT.VSS	18
2.7	Schablone: CT_FUEHLER.VSS	19
2.8	Schablone: CT_LUFT.VSS	20
2.9	Schablone: CT_MOTOR.VSS	21
2.10	Schablone: CT_STELL.VSS	21
2.11	Schablone: CT_ZEICH.VSS	22
2.12	Schablone: CT_WASSER.VSS	23
2.13	Schablone: Allgemein.VSS	23
2.14	Schablone: CT_KANAELE.VSS	24
2.15	Beispielregelschema	25
3.1	Ansicht des neuen Hilfesystems	32

Tabellenverzeichnis

1.1	Übersicht über die Praktikumsschwerpunkte	1
3.1	Schrifthervorhebungen	30

1 Kurzübersicht über das Praktikum

Als Bestandteil der praktischen Ausbildung im Studiengang Angewandte Informatik an der TU Chemnitz ist ein insgesamt 15-wöchiges Betriebspraktikum vorgesehen. Dieses Praktikum absolvierte ich vom 28.2.2005 bis 16.6.2005 bei der *JOHNSON CONTROLS JCI Regelungstechnik GmbH* in der Geschäftsstelle Leipzig.

Die Firma ist im MSR-Bereich tätig und arbeitet unter anderem mit Microsoft Visio. Zur Erstellung von Regelschemen werden Symbole und VBA-Funktionen benutzt. Mein Aufgabenbereich lässt sich in zwei Teile gliedern: zum einen in die Neuprogrammierung der benötigten Symbole und Dokumentation dieses Prozesses, sowie die Erstellung einer Anleitung zur Arbeit mit Microsoft Visio und zum anderen in die Erweiterung der vorhandenen Funktionen, einschließlich Neugestaltung der Online-Hilfe. Die nachfolgende Tabelle zeigt die von mir bearbeiteten Praktikumsschwerpunkte in chronologischer Abfolge:

Praktikumsschwerpunkt	Zeitdauer in Werktagen
Schwerpunkt 1: Arbeit an den Symbolen	
Kennenlernen der Entwicklungsumgebungen Visio und VBA	8
Problemanalyse und Problemlösung	37
Dokumentation	10
sonstige Programmieraufgaben	6
Schwerpunkt 2: Arbeit am Hilfesystem	
Überarbeitung des Hilfe-Systems	14
Summe	75

Tabelle 1.1: Übersicht über die Praktikumsschwerpunkte

2 Praktikumsschwerpunkt 1 – Arbeit an den Symbolen

2.1 Problemstellung

Die Firma JCI arbeitete bis zu meinem Praktikum mit Visio-Symbolen, die in der Vergangenheit von einem Mitarbeiter gezeichnet und programmiert wurden, der nun nicht mehr verfügbar ist. Die vorhandenen Funktionen der Symbole waren nur schlecht – abgesehen von einigen wenigen Kommentaren gar nicht – dokumentiert. Es kamen mehr und mehr Symbole hinzu, welche teilweise von anderen Mitarbeitern programmiert wurden. Die meisten dieser vorhandenen oder neuen Symbole samt Funktionen wurden aber den aktuellen Ansprüchen nicht mehr gerecht. Denn mittlerweile gibt es automatisierte Funktionen, die das Erstellen von Regelschemen erleichtern. Manche Symbole waren dabei voll funktionsfähig, bei vielen jedoch traten Fehler auf.

2.2 Ziel

Das Ziel meines ersten Praktikumsschwerpunktes war es festzustellen, warum diese Fehler auftreten und wie sie sich auflösen lassen. Als nächstes war herauszufinden, welche Symbole überhaupt benötigt werden. Auf dieser Wissensgrundlage sollte ich alle benötigten Symbole entsprechend ändern, ggf. neue Symbole erstellen und evtl. um Funktionen erweitern.

Des weiteren sollten alle Änderungen gut dokumentiert und eine umfassende Anleitung verfasst werden, die es den Anwendern erlaubt die Symbole besser zu „verstehen“ und zu nutzen. Sozusagen sollten sich die User mit Hilfe der Dokumentation Wissen aneignen, mit dem sie mehr der mächtigen Funktionen von Visio nutzen und auch eigene Symbole erstellen können, die den Anforderungen entsprechen.

2.3 Vorgehensweise

Zunächst musste ich mich mit der ShapeSheet-Programmierung in Visio beschäftigen. Auf der Grundlage zweier Bücher, in Gesprächen mit meinem Praktikumsbetreuer, durch Konsultieren der entsprechenden Online-Hilfe und durch Probieren habe ich mir dies erarbeitet.

Als nächstes folgte die Analyse, welche Symbole überhaupt noch benötigt werden. Dies war durch Befragung der Nutzer der Symbole möglich. Es stellte sich heraus, dass es einige Symbole gab, die überhaupt nicht mehr benutzt werden, andere Symbole in den verschiedenen Kategorien doppelt oder nur in anderer Größe vorkamen und dass es Bedarf an neuen Symbolen gab.

Insgesamt waren also die 94 vorhandenen Symbole auf Funktionalität zu überprüfen und zu bearbeiten und ggf. neue Symbole zu erstellen. Die fertigen Symbole mussten danach neu in Kategorien eingeteilt werden, da sich die alte Einteilung als nicht mehr verwendbar erwies.

Meine Erkenntnisse dokumentierte ich schrittweise während der Analyse und Überarbeitung der Symbole, um sie nach Beendigung der Arbeit zusammenfassend als Dokumentation und Anleitung nieder zu schreiben.

2.4 Realisierung

2.4.1 Grundsätzliches

Entwicklungsumgebung

Microsoft Visio ist ein Programm, mit dem komplexe Zeichnungen erstellt werden können. Der Anwender hat ein Zeichenblatt vor sich und eine Reihe von Visio-eigenen und evtl. auch importierten Symbolen, welche in Schablonen zusammengefasst werden. Alle grafischen Objekte werden in Visio als Shapes bezeichnet, auch das Zeichenblatt an sich oder Gruppen von Shapes. Alle Eigenschaften dieser Shapes – angefangen bei einfachen Merkmalen wie Größe oder Position auf dem Zeichenblatt bis hin zu Schutzfunktionen oder Formatierungen – werden in sog. ShapeSheets festgehalten. Die ShapeSheets haben verschiedene Abschnitte, in denen die Eigenschaften festgelegt sind. Solch ein ShapeSheet kann man öffnen und in den einzelnen tabellenartigen Zellen Änderungen vornehmen. Wie bei allen Microsoft-Produkten liegt auch hinter Visio die Programmiersprache VBA (Visual Basic for Applications), mit welcher man direkt auf die ShapeSheets zugreifen und somit die Shapes programmieren kann.

Auch stellt Visio über 150 verschiedene Funktionen bereit, mit denen gearbeitet werden kann. Zu den Funktionen gehören beispielsweise Mathematische Funktionen, Fehlerfunktionen, Logische Funktionen, Ereignisfunktionen oder Textfunktionen. Mit Hilfe dieser Funktionen und der Zellbeschriftungen kann innerhalb eines ShapeSheets oder aber auf andere ShapeSheets referenziert werden. Somit ist es möglich sehr intelligente Symbole zu schaffen, die ihrer entsprechenden Einsatzform optimal angepasst werden können – der Anwender bekommt also eine für seine Arbeit optimierte Symbolbibliothek (Schablone).

Symbolbeschreibung

Die für die Regelschemen verwendeten Symbole sind alle nach folgendem Schema aufgebaut: das Symbol an sich, ein Verbinder, der vom Shape mit maximal zwei Knicken senkrecht nach unten laufen soll und eine Klemme am unteren Ende des Binders, an die später per Programm die Beschriftung der Ein- und Ausgänge automatisch angefügt wird.

Außerdem gibt es noch einige Sonderfälle – wie beispielsweise Kanäle, die als Ausrichtungsobjekt auf dem Zeichenblatt dienen und immer an der selben Position sitzen müssen. Diese

Symbole haben keinerlei Verbinder und sind am einfachsten zu bearbeiten, da das folgende Programm keine Änderungen an ihnen vornimmt.

Programmbeschreibung

Wie schon erwähnt, gibt es ein fertiges Programm, dass das Erstellen der Regelschemen erleichtert. Ausgangspunkt sind die Symbole auf dem Zeichenblatt und eine geöffnete Excel-Datei, in welcher sich die Geräteliste und die Belegungen für Ein- und Ausgänge der entsprechenden Geräte befinden. Die Symbole haben eine bestimmte Bezeichnung, die als Referenz für den Eintrag im Excel-Sheet dient. Durch Starten des Programms über das Hauptmenü aus Visio heraus, werden zu allen auf dem Zeichenblatt befindlichen und bezeichneten Symbolen die entsprechenden Belegungen der Ein- und Ausgänge in der Excel-Datei gesucht, besondere Symbole in Visio mit den entsprechenden Einträgen beschriftet und dem Zeichenblatt hinzugefügt. Im Anschluss daran werden automatisch die Klemmen wie vorgegeben horizontal ausgerichtet (die vertikale Ausrichtung bleibt identisch) – sozusagen auf dem Zeichenblatt nach unten gezogen – und danach entsprechend beschriftet. Der Anwender betrachtet das Ergebnis und fügt ggf. Positionsänderungen der Symbole durch (falls sich die Beschriftungen überlappen etc.) und startet das Programm erneut, bis das Ergebnis zufriedenstellend ist.

2.4.2 Analyse und Lösung der auftretenden Probleme

Zur Problemanalyse verwendete ich ein funktionierendes Shape und ein nicht funktionierendes. Nach und nach konnte ich feststellen, dass es mehrere Fehlerfälle gab:

- Fall 1: das Shape lässt sich gar nicht oder nicht wie gewünscht drehen
- Fall 2: die schon eingebauten Funktionalitäten des Shapes funktionieren nicht
- Fall 3: das Programm zur automatischen Beschriftung der Klemmen funktioniert bei einigen Shapes nicht

Meine Analyse ergab folgendes:

Fall 1: das Shape lässt sich gar nicht oder nicht wie gewünscht drehen

Dieser Fall trat auf, wenn gar keine Drehfunktion im Kontextmenü des Shapes implementiert worden war oder aber wenn die Standard-Visio-Drehfunktion verwendet wurde. Diese dreht das komplette Shape, also das Symbol samt Verbinder und Klemme. Dies war aber nicht erwünscht. Es sollte sich nur das Symbol drehen, die Klemme sollte an ihrer Position auf dem Zeichenblatt bleiben und der Verbinder sollte Symbol und Klemme verknüpfen.

Zur Lösung dieses Problems musste eine passende Drehfunktion implementiert werden und ggf. die existierende Funktion durch die neue ersetzt werden.

Fall 2: die schon eingebauten Funktionalitäten des Shapes funktionieren nicht

Dieses falsche Verhalten der Shapes kann verschiedene Ursachen haben, die jedoch alle auf Visio an sich zurückzuführen sind. Zum einen gibt es verschiedene Funktionsnamen und Abschnitts- bzw. Zellenbezeichnungen in deutschen und in englischen Visio-Versionen. Normalerweise werden die deutschen Namen automatisch ins Englische übersetzt – dies funktioniert selbstverständlich nicht, wenn ein Ausdruck in Hochkomma steht. Und genau hier liegt das Problem: die Syntax der Funktionen ist teilweise so, dass der entsprechende Name der Zelle, die von der Änderung betroffen ist, in Hochkomma geschrieben werden muss.

Zur Lösung dieses Problems müssen bei der Verwendung von Funktionen generell die englischen Funktionsnamen verwendet werden, damit diese auch in einer deutschen Visio-Version funktionieren.

Zum anderen ist die Punktierung in den verschiedenen Visio-Versionen unterschiedlich. Einmal werden die Ausdrücke innerhalb von Funktionen durch Komma, mal durch Semikolon von einander getrennt.

Dieses Problem lässt sich nicht lösen. Es kann lediglich der Anwender darauf hingewiesen werden, dass in diesem Fall die Punktierung geändert werden muss. Eine Alternative wäre die Entwicklung verschiedener Ausführungen der Schablonen, die an die entsprechenden Visio-Versionen angepasst sind. Es ist aber so, dass bei der Entwicklung der Shapes in Visio 2002 derzeit noch keine Probleme mit anderen in der Firma verwendeten Versionen aufgetreten sind. Wahrscheinlich sind die verschiedenen Versionen zueinander kompatibel. Allerdings besteht seit Visio 2003 das Problem der nicht mehr vorhandenen Abwärtskompatibilität. Höchstwahrscheinlich werden die Daten in einem neuen Format gespeichert. Es besteht aber die Möglichkeit die Daten im „alten“ Format abzulegen.

Aufgrund des hohen Mehraufwands bei verhältnismäßig geringem Nutzen wurde diese Alternative nicht verfolgt.

Fall 3: das Programm zur automatischen Beschriftung der Klemmen funktioniert bei einigen Shapes nicht

In diesem Fall wurden die falschen Kontrollpunkte nach unten gezogen, sodass das Shape nach dem Durchlauf des Programms verzerrt war und sich die Klemme immer noch nicht an der richtigen Position auf dem Zeichenblatt befand.

Die Analyse des Quellcodes ergab, dass fest auf eine bestimmte Zeile im Abschnitt Controls zugegriffen wurde. Manchmal befand sich die Klemme aber nicht in der entsprechenden Zeile und in diesem Fall trat der Fehler auf. Es musste also ein allgemein gültiges Kriterium für die Identifikation der Klemme gefunden werden.

Dieser Fehler konnte behoben werden, indem der Quellcode angepasst wurde und die Zeile nun über den Hinweis an der Klemme identifiziert wird.

2.4.3 Lösung im Detail

Bei der Überarbeitung der vorhandenen Shapes habe ich festgestellt, dass jedes auf eine andere Art und Weise aufgebaut ist. Außerdem gibt es unterschiedliche Anforderungen an die einzelnen Symbole – bei einem Fühler beispielsweise muss sich nur das Fühlerelement per Maussteuerung drehen lassen, bei Klappen mit Motoren hingegen muss sich das komplette Symbol per Kontextmenü drehen lassen und der Motor muss durch Klicken immer wieder auf den Standard- abstand zur Klappe zurückgesetzt werden können. Somit war es nicht möglich ein Programm zu schreiben, dass alle benötigten Shapes bearbeitet. Ich arbeitete deshalb meist direkt in den ShapeSheets und nutzte die Visio-Funktionen.

ShapeSheet-Programmierung

Wie schon erwähnt, besteht jedes ShapeSheet aus Abschnitten. Es gibt grundsätzlich fünf verschiedene Arten von Shapes, die sich in der „Zusammensetzung“ der Abschnitte unterscheiden – es gibt Abschnitte, die Sie in jedem ShapeSheet finden und andere, die sozusagen „artspezifisch“ sind. Die Abschnitte enthalten *alle* Eigenschaften der Symbole – angefangen bei der Geometrie, über die Einträge und Funktionen im Kontextmenü bis hin zur Festlegung der Kontrollpunkte¹. Die entsprechenden Abschnitte können hinzugefügt oder um Zeilen erweitert werden.

Auf die Abschnitte kann untereinander über die Zellen-Namen zugegriffen bzw. referenziert werden. Zusammen mit den Visio-Funktionen, vor allem den logischen Funktionen, lassen sich programmiersprachenähnliche Konstrukte schaffen. Somit können die Shapes „programmiert“ werden.

Koordinatensysteme

Visio verwendet für die Lage der Shapes auf dem Zeichenblatt zwei verschiedene Koordinatensysteme – ein globales und ein lokales. Das globale Koordinatensystem kann folgendermaßen beschrieben werden: der Ursprung befindet sich in der linken unteren Ecke des Zeichenblattes, dieser Punkt hat die Koordinaten $(0, 0)$. Die x-Achse ist die horizontale Linie, die durch den Ursprung geht, die y-Achse die vertikale.

Zu diesem Koordinatensystem wird nun jeder Punkt auf dem Zeichenblatt, und auch außerhalb davon, exakt bestimmt – Punkte links neben dem Zeichenblatt haben dann negative x-Koordinaten. Jedes Shape hat genau einen Punkt, dessen Lage global angegeben wird: der Pin oder Drehpunkt.

Die X-Koordinate wird als `PinX`, die y-Koordinate als `PinY` bezeichnet. Das Shape selbst wird nun um diesen Punkt herum aufgebaut – mit anderen Worten: die Lage des Pins innerhalb des Shapes wird durch das Koordinaten-Paar `LocPinX` und `LocPinY` festgelegt. Nor-

¹ Kontrollpunkte, auch Steuerelemente oder Controls sind kleine gelbe „Punkte“, die Symbolen hinzugefügt werden können und die der Anwender mit der Maus bewegen kann. Dadurch lassen sich Symbole verändern und somit eine gewisse Interaktivität erreichen.

malerweise befindet sich der Pin in der Mitte des Shapes, weswegen die Koordinaten meistens folgende Werte besitzen:

$$\begin{aligned} \text{LocPinX} &= \text{Width} * 0,5 \\ \text{LocPinY} &= \text{Height} * 0,5 \end{aligned}$$

Zwei einfache Rechtecke auf dem Zeichenblatt könnten folgende Zelleinträge haben: Lage auf dem Zeichenblatt und Größe des Rechtecks wären identisch.

$\text{PinX} = 140 \text{ mm}$	$\text{PinX} = 100 \text{ mm}$
$\text{PinY} = 200 \text{ mm}$	$\text{PinY} = 170 \text{ mm}$
$\text{LocPinX} = \text{Width} * 0,5$	$\text{LocPinX} = \text{Width} * 0$
$\text{LocPinY} = \text{Height} * 0,5$	$\text{LocPinY} = \text{Height} * 0$
$\text{Width} = 80 \text{ mm}$	$\text{Width} = 80 \text{ mm}$
$\text{Height} = 60 \text{ mm}$	$\text{Height} = 60 \text{ mm}$

Beispiel: Klappe + Motor

Betrachten wir nachfolgend den Überarbeitungsprozess am Beispiel des Symbols *Klappe + Motor* aus der Schablone *CT_STELL*².

Abbildung 2.1 stellt auf der linken Seite das Symbol an sich und auf der rechten Seite das Symbol mit Kontroll- und Verbindungspunkten³ dar. Das Symbol besteht aus der Klappe (bestehend aus drei „EinzelShapes“), dem Motor, der Verbindung zwischen Klappe und Motor, der Klemme, dem Verbinder und einem Textfeld zur Beschriftung.

Die Anforderungen an dieses Symbol waren:

- Einstellungen (Betriebsmittelkennzeichen angeben und Luftklappentyp festlegen)
- Drehen um 90 Grad nach links
- Klemme und Verbinder unsichtbar schalten
- Symbol unsichtbar schalten
- Motor ausblenden
- Motor und Verbindung ausblenden
- Motor in Standardentfernung zur Klappe bringen

²Anmerkung: Ursprünglich war das Symbol als „Luftklappe“ bezeichnet und befand sich in der Schablone *tpl_klt*.

³blaue Kreuze

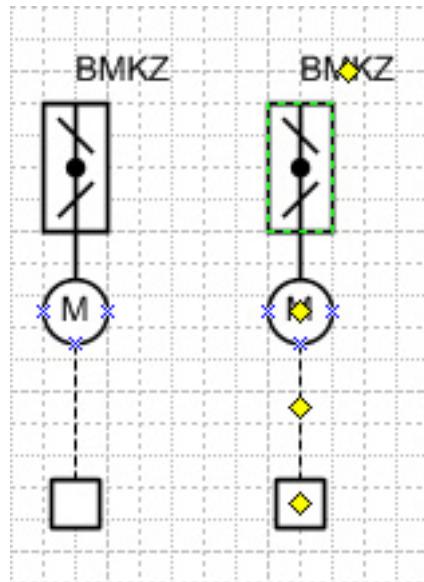


Abbildung 2.1: das Symbol „Klappe + Motor“

Einstellungen

Durch Klicken auf den Punkt „Einstellungen“ im Kontextmenü konnte die Betriebsmittelkennzeichnung geändert werden. Außerdem ließ sich der Luftklappentyp festlegen – zur Auswahl standen hierbei: „Klappe gegenläufig“, „Klappe gleichläufig“, „Klappe allgemein“, „Rauchschutzklappe“ oder „Klappe luftdicht“. Diese Einstellungen funktionierten auch ohne Probleme, sodass ich daran nichts mehr ändern musste.

Bei der Auflistung aller zu bearbeitenden Symbole fiel mir auf, dass es auch noch eine Brandschutzklappe mit Motor gab. Dieses eigenständige Symbol sieht genau so aus, wie die Klappe mit Motor bis auf geringfügige Änderungen an der Klappe. Deswegen fügte ich beide Symbole zusammen und nun kann über das Menü „Einstellungen“ auch der Luftklappentyp „Brandenschutzklappe“ ausgewählt werden.

Die unterschiedlichen Auswahlmöglichkeiten des Luftklappentyps können – in oben genannter Reihenfolge von links nach rechts – in Abbildung 2.2 betrachtet werden.

Drehen um 90 Grad nach links

Dieses Symbol ließ sich ursprünglich nur im Ganzen, also samt Klemme und Verbinder drehen. Dies war aber wie erwähnt nicht erwünscht.

Die Drehfunktion von Visio, die das ganze Symbol dreht musste nun ersetzt werden. Der Kontextmenüeintrag *um 90 Grad nach links drehen* konnte erhalten bleiben, es musste nur die

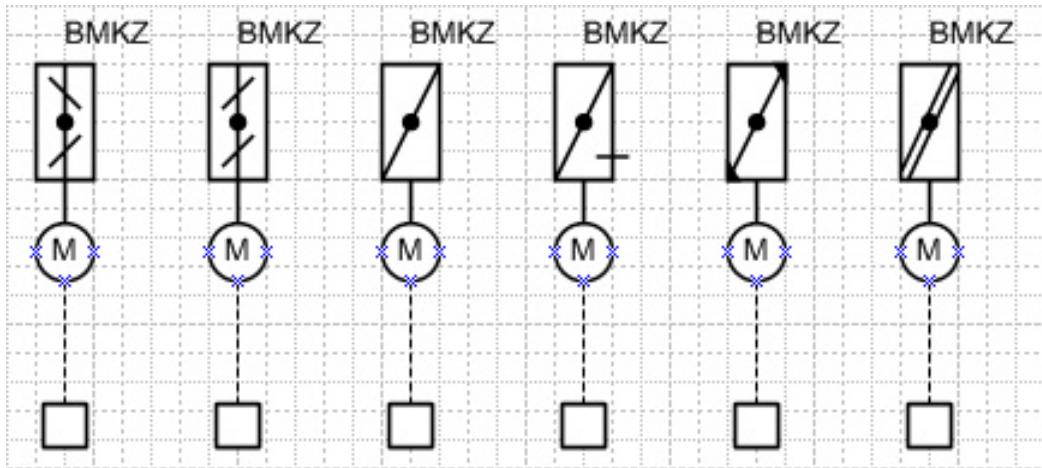


Abbildung 2.2: die verschiedenen Luftklappentypen

entsprechende Funktionalität geändert werden: im Abschnitt *Actions* des in der Hierarchie am weitsten oben liegenden ShapeShetts musste `DOCMD(1312)` durch die neue Drehfunktion ersetzt werden. Dazu musste zuerst im Abschnitt *User-defined Cells* eine neue Zeile hinzugefügt und diese in `User.wink` umbenannt werden. Diese neue Zelle speichert den aktuellen Drehwinkel – also 0 Grad, 90 Grad, 180 Grad oder 270 Grad, initialisiert mit 0 Grad. Nun zurück zum Abschnitt *Actions*, in dem in der ersten Spalte die neue Drehfunktion eingefügt wird:

```
IF (User.wink = 0 deg; SETF ("User.wink"; "90 deg");
IF (User.wink = 90 deg; SETF ("User.wink"; "180 deg");
IF (User.wink = 180 deg; SETF ("User.wink"; "270 deg");
SETF ("User.wink"; "0 deg")))
```

Bemerkung: Die Visio-Funktion `SETF()` sorgt dafür, dass die Zelle (erstes Argument) auf den neuen Wert (zweites Argument) gesetzt wird.

Die nun fertige Drehfunktion hat noch keinerlei Wirkung auf das Symbol an sich. Im ShapeSheet ändert sich durch Auswählen des entsprechenden Kontextmenüeintrags der Wert in der Zeile `User.wink` von 0 deg über 90 deg, 180 deg und 270 deg wieder zurück auf 0 deg. Nun müssen also die untergeordneten ShapeSheets des Symbols angepasst werden. Wechseln wir daher zum ShapeSheet der Klappe (das äußere Viereck der Klappe). Im Abschnitt *Shape Transform* gibt es eine Zelle *Angle*. Dort muss folgende Referenz eingetragen werden:

`Sheet.x!User.wink4`

⁴Visio benennt alle Shapes auf dem Zeichenblatt standardmäßig mit „Sheet.X“, wobei das „X“ eine ganze Zahl beginnend mit 1 ist (das Zeichenblatt selbst hat die Nummer 0). Auf einem sonst leeren Zeichenblatt hat das in der Hierarchie oberste ShapeSheet, also das komplette Symbol die Nummer 1.

Analog muss auch noch im ShapeSheet für die unterschiedlichen Klappen-Geometrien diese Änderung vorgenommen werden, da sich diese genau wie die Klappe verhalten müssen, im Beispiel aber in einem eigenen ShapeSheet vorzufinden sind. (Es wäre auch möglich diese Geometrien mit in das ShapeSheet der Klappe einzutragen.)

Besteht das ganze Symbol nur aus dem obersten GruppenShape, den ShapeSheets für Klemme und Verbinder und noch genau einem weiteren ShapeSheet für das eigentliche Symbol – dieses ShapeSheet kann auch ein GruppenShape sein – so lässt sich das Symbol ohne Probleme richtig drehen. Im Falle des Symbols *Klappe + Motor* dreht sich nun aber nur die Klappe richtig. Motor und Verbindung zwischen Motor und Klappe, sowie Klemme und Verbinder verbleiben am Ausgangspunkt.

Im Falle von Klemme und Verbinder ist dies gewollt, aber der Motor und die Verbindung zur Klappe sollen sich mitdrehen. Deshalb muss jetzt die Position des Motors in Abhängigkeit des Drehwinkels berechnet werden. Betrachten wir dazu die Abbildung 2.3: das grüne lokale Koordinatensystem des Symbols ändert sich nicht, gestrichelt sind eingetragen: `width * 1` auf der x-Achse und `height * 1` auf der y-Achse. Gesucht ist die neue Position des Mittelpunktes des Motors (rotes Kreuz) in Abhängigkeit des Drehwinkels. Die roten Werte sind die x- und y-Koordinaten dieses Punktes.

Die Werte sind im passenden ShapeSheet einzutragen. Die allgemeine Formel dafür lautet:

```
IF (User.wink = 0 deg; <Ausdruck1>; IF (User.wink = 90 deg;  
<Ausdruck2>; IF (User.wink = 180 deg; <Ausdruck3>;  
<Ausdruck4>)) )
```

Für `<Ausdruck1>` bis `<Ausdruck4>` muss ein entsprechender Ausdruck, in diesem Fall also die Position, angegeben werden.

Die Formeln können auch im Abschnitt *User-defined Cells* eingetragen werden, wenn anschließend mit der Funktion `SETF()` gearbeitet wird. Diese Methode sollte immer dann verwendet werden, wenn das Shape Funktionen besitzt, die es dem Benutzer erlauben mit der Maus Veränderungen der Position vorzunehmen – beispielsweise bei Steuerelementen. (Im Beispiel ist die Motorposition abhängig von einem Steuerelement. Wäre die Formel im Abschnitt *Controls* eingefügt worden, wird sie bei Bewegung des Steuerelements mit der Maus mit einem absoluten Wert überschrieben. Wenn also nun das Shape wieder gedreht wird, ist die Funktion weg und die Position ändert sich nicht mehr.) Ich habe mich für diese Art der Umsetzung für das Symbol „Klappe + Motor“ entschieden.

Für den Mittelpunkt des Motors gelten also folgende Formeln:

```
x = IF (User.wink = 0 deg; SETF ("Controls.X3"; "=GUARD (Width*0.5)");  
IF (User.wink = 90 deg; SETF ("Controls.X3"; "=Width*3");  
IF (User.wink = 180 deg; SETF ("Controls.X3"; "=GUARD (Width*0.5)");  
SETF ("Controls.X3"; "=Width*-2"))))
```

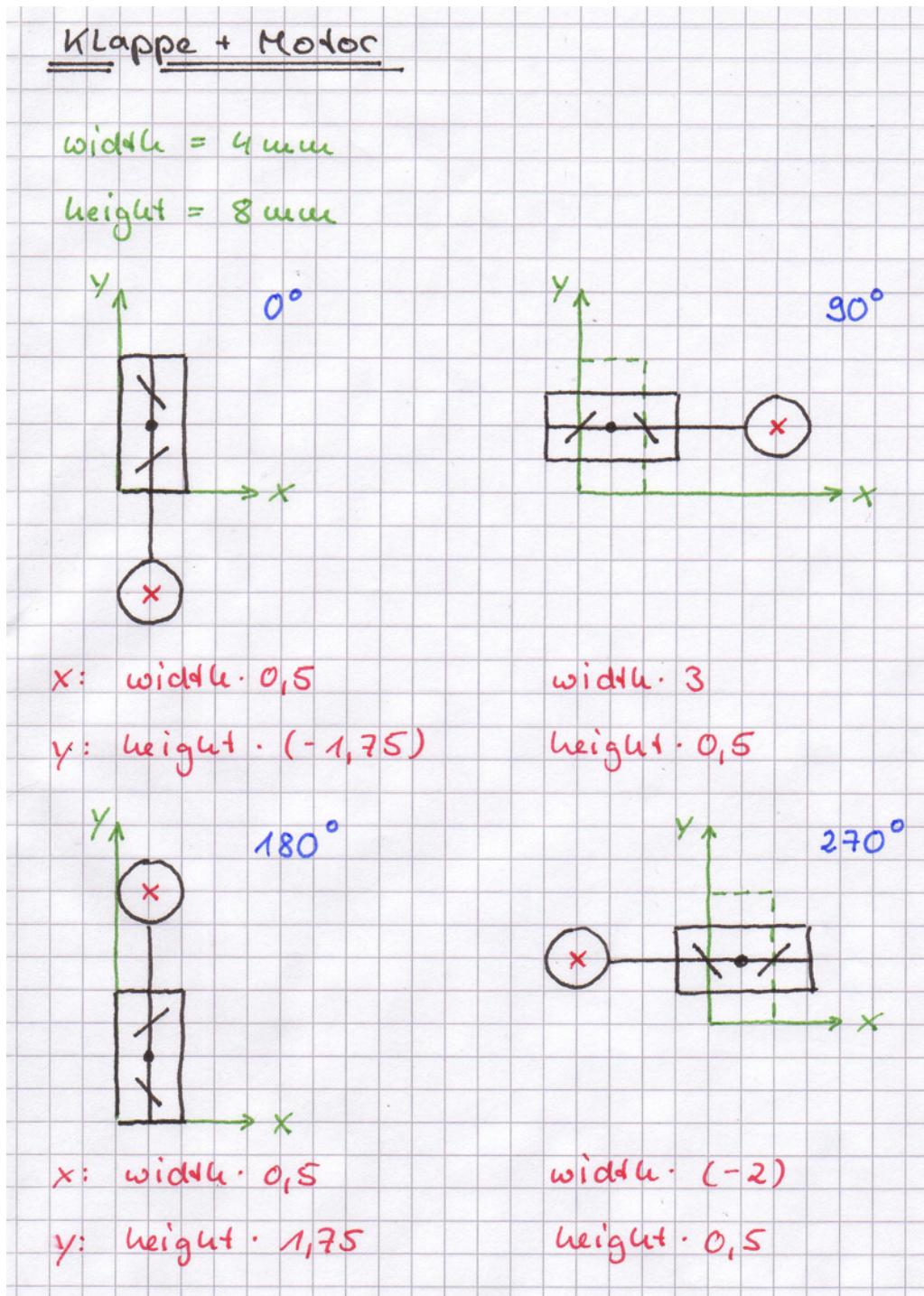


Abbildung 2.3: Motorposition in Abhängigkeit vom Drehwinkel

```
y = IF(User.wink = 0 deg; SETF("Controls.Y3"; "Height*-1,75");
IF(User.wink = 90 deg; SETF("Controls.Y3"; "=GUARD(Height*0,5)");
IF(User.wink = 180 deg; SETF("Controls.Y3"; "Height*1,75");
SETF("Controls.Y3"; "=GUARD(Height*0,5))))
```

Bemerkungen: Controls.X3 und Controls.Y3 sind bei diesem Symbol diejenigen Kontrollpunkte, an denen der Mittelpunkt des Motors „festgeklebt“ ist. Somit bewirkt eine Änderung der Kontrollpunktposition eine Änderung des Motors. Die Funktion GUARD () sorgt dafür, dass der Anwender den Wert des Zelleneintrags nicht ändern kann. Wenn bei einem Drehwinkel von 0 Grad der x-Wert des Kontrollpunktes geschützt ist, kann der Kontrollpunkt nicht mehr in x-Richtung – also horizontal – bewegt werden, bei einem geschützten y-Wert bei 90 Grad, kann der Kontrollpunkt nicht mehr in y-Richtung – also vertikal – bewegt werden. Dies ist beim Beispiel-Symbol so gewünscht.

Analog sind diese Änderungen für jede andere Geometrie und jeden Verbindungspunkt o.Ä. vorzunehmen. Dabei sind die Änderungen immer „an Ort und Stelle“ einzutragen, also nicht im übergeordneten Shape-Sheet.

Unsichtbarkeiten

Beim Ausgangssymbol waren die Funktionen „Klemme und Verbinder unsichtbar“ und „Shape unsichtbar“, sowie „Motor ausblenden“ und „Motor und Verbindung ausblenden“ bereits im Kontextmenü integriert, funktionierten aber nicht. Deshalb musste ich diese überarbeiten.

Geometrien können unsichtbar und danach wieder sichtbar „geschaltet“ werden. Zunächst muss die Sichtbar- Unsichtbar- Funktion in das Kontextmenü eingetragen werden – im Abschnitt *User-defined Cells* die Zeile *User.sicht1* einfügen und mit Null initialisieren. Danach sind analog zu oben die Änderungen im Abschnitt *Actions* vorzunehmen:

```
Action = IF(User.sicht = 1; SETF("User.sicht"; 0);
SETF("User.sicht"; 1))
Menu = IF(User.sicht = 0; "Unsichtbar"; "Sichtbar")
```

Nun muss noch die Geometrie so eingestellt werden, dass sie auch unsichtbar bzw. wieder sichtbar wird. Hierzu muss für jede Geometrie, die an der Funktion beteiligt ist, das entsprechende ShapeSheet geöffnet und folgende Änderung vorgenommen werden (Abschnitt *Geometry*, Zeile *GeometryX.NoShow*):

```
Sheet.X!User.sicht = 1
```

Im Falle des Symbols Klappe + Motor gibt es mehrere solcher Sichtbar- Unsichtbar- Funktionen, sodass mehrere Zeilen im Abschnitt *User-defined Cells* eingefügt und unterschiedlich benannt (*User.sicht1*, *User.sicht2* etc.) werden, sowie im Abschnitt *Actions* mehrere Einträge für

das Kontextmenü eingefügt werden mussten. Außerdem waren die Geometrien an die entsprechenden Variablen anzupassen und ggf. die Funktionen mit OR zu verknüpfen.

Standardentfernung Motor

Beim Ursprungssymbol war diese Funktion im Kontextmenü vorgesehen, jedoch nicht implementiert.

Bei dieser Funktionen muss der Wert des Kontrollpunktes des Motors immer wieder auf den Standardabstandswert gesetzt werden und zwar in Abhängigkeit vom aktuellen Drehwinkel. Dies lässt sich nur über einen kleinen Trick bewerkstelligen: mit einem Schalter. Nach dem Auslösen der Aktion „Standardabstand Motor“ muss der Schalter geschaltet werden, da sonst keine „Zustandsänderung“ eintritt und die Funktion, die den Standardabstand wiederherstellt nicht angestoßen wird. Immer wenn sich die Schalterstellung ändert, muss der Standardabstand in Abhängigkeit vom aktuellen Drehwinkel gesetzt werden. Angenommen der Schalter (User.motor_ausrichten) schaltet immer von Null auf Eins und dann wieder zurück, müsste die Formel für das Ausrichten des Motors folgendermaßen aussehen:

```
IF (User.motor_ausrichten <> 2; <Ausdruck>; 0)
```

Bemerkung: für <Ausdruck> müssen die entsprechenden Winkel und Positionen als IF-Anweisung angegeben werden.

Joker-Symbole

Damit die Anwender auch einfach eigene neue Symbole schaffen können, kreierte ich zwei Joker-Symbole (ein Rechteck und ein Oval), die als Ausgangsbasis zur Erstellung neuer Symbole genutzt werden können. In der Dokumentation beschrieb ich unter anderem genau, wie diesen Ausgangssymbolen neue Geometrien hinzugefügt werden können oder wie das Kontextmenü mit Funktionen erweitert wird. Komplizierte Symbole lassen sich eventuell nicht auf diese Art erstellen, aber in der Dokumentation ist ausführlich beschrieben, wie Visio arbeitet und deswegen sollte auf der Grundlage der Joker-Symbole mit entsprechenden Erweiterungen und Kniffen annähernd jedes gewünschte Symbol samt Funktionen kreiert werden können.

Abbildung 2.4 zeigt die beiden Joker-Symbole.

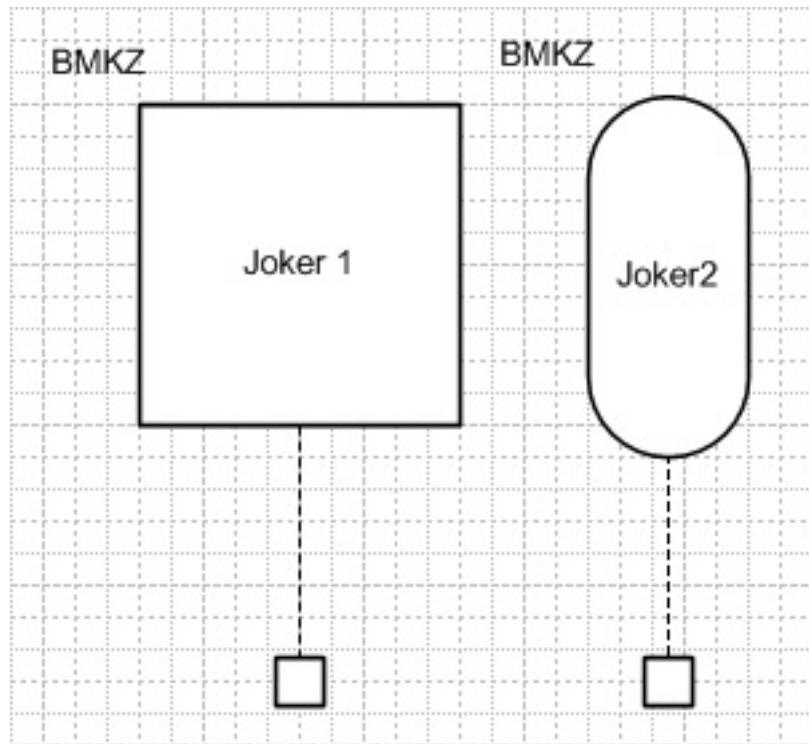


Abbildung 2.4: die zwei Joker-Symbole

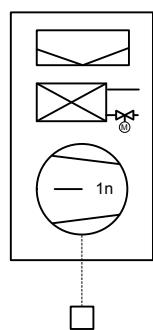
2.5 Ergebnisse

Nachfolgend eine Auflistung der Ergebnisse meiner Arbeit in diesem Problembereich:

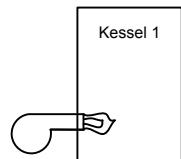
- 85 bearbeitete und neun neu geschaffene Symbole
- aus anfänglich sieben Schablonen wurden am Ende neun neue Schablonen und eine in der nur einige Shapes geändert wurden
- 80-seitige Dokumentation / Anleitung zur Arbeit mit Microsoft Visio

Abbildungen 2.5 bis 2.12 zeigen die neuen Symbolbibliotheken und Abbildung 2.13 stellt die geänderten Symbole aus der Schablone Allgemein.VSS dar. Die “Grundstrukturen“ aller Regelschemen, die Kanäle, um die herum alle anderen Symbole angeordnet werden müssen, sind in Abbildung 2.14 dargestellt. In Abbildung 2.15 ist ein Beispielregelschema – erstellt aus den neuen Symbolen – zu sehen.

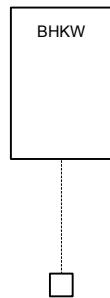
CT_BAUGRUPPEN



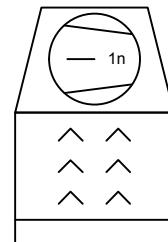
UMK



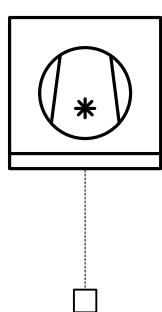
Kessel



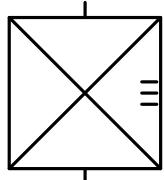
BHKW



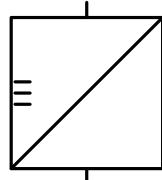
Kühlturm



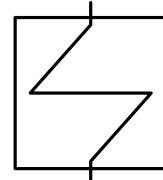
Kompressor



Luft 01



Luft 02



Wasser

Abbildung 2.5: Schablone: CT_BAUGRUPPEN.VSS

CT_ELT

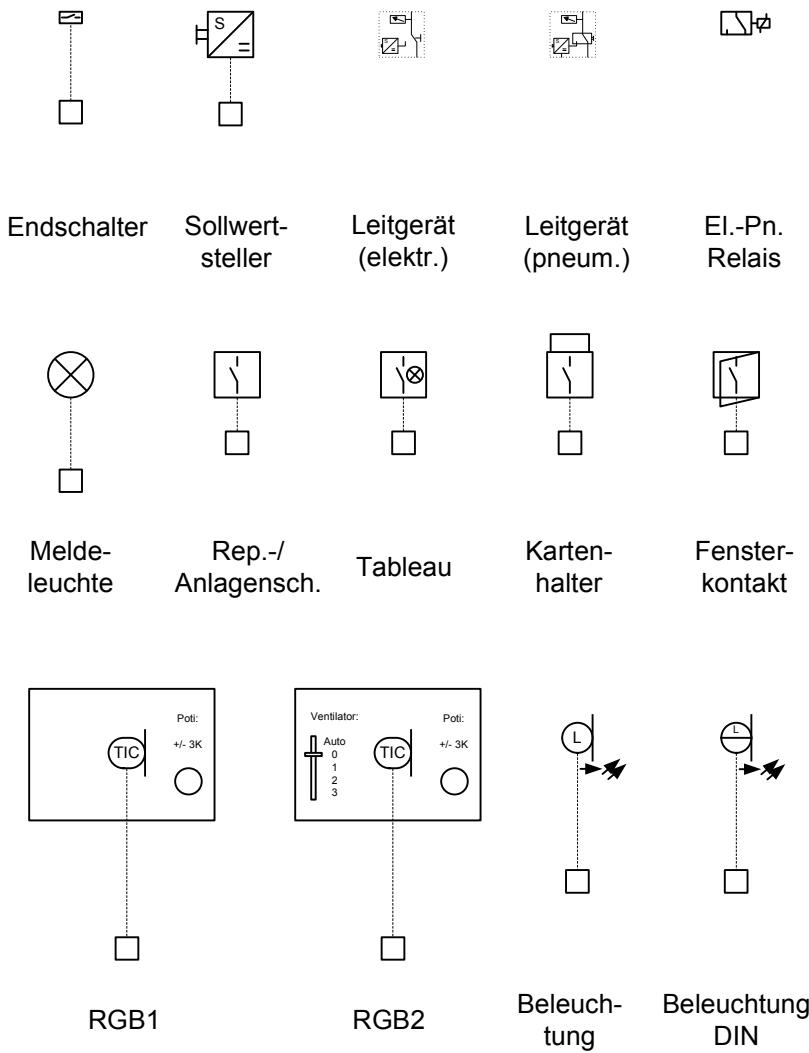


Abbildung 2.6: Schablone: CT_ELT.VSS

CT_FUEHLER

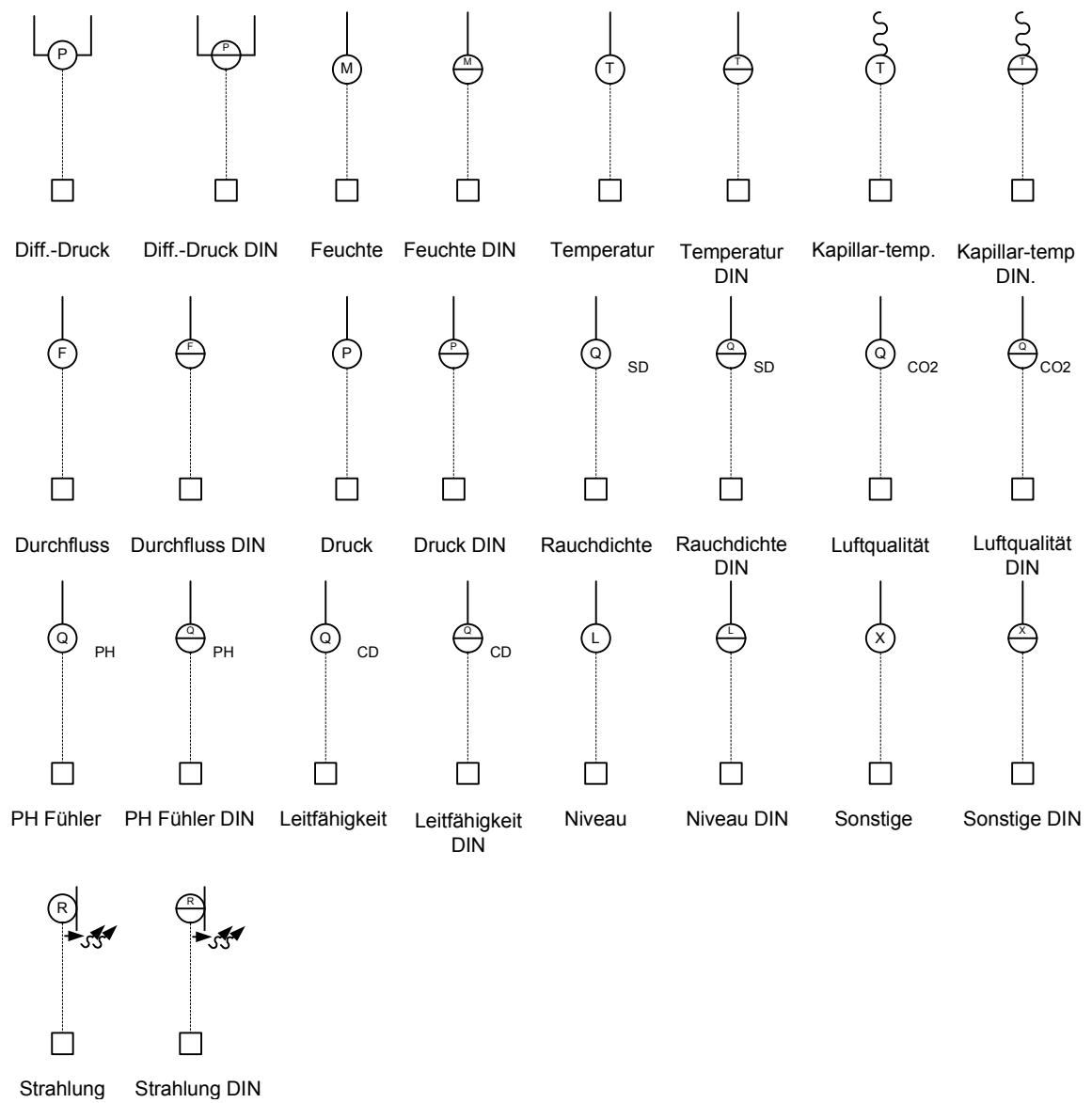
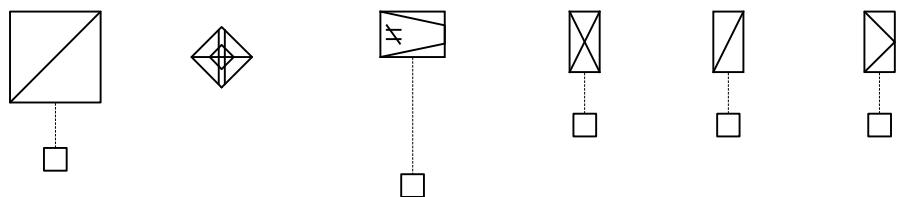


Abbildung 2.7: Schablone: CT_FUEHLER.VSS

CT_LUFT



Wärme-tauscher

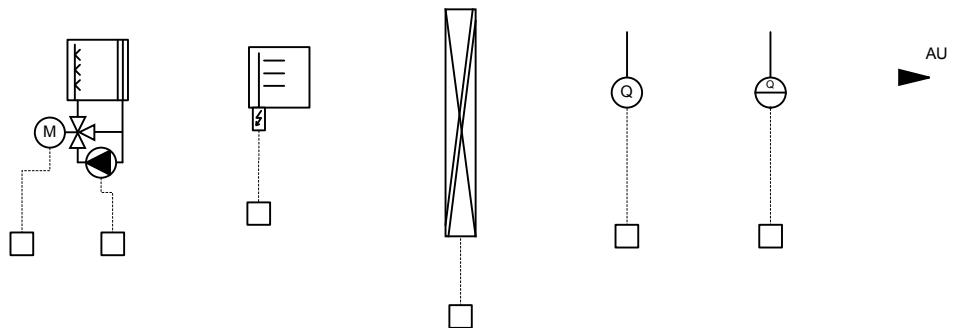
Kreuzstrom-wärmetauscher

Volumenstrom-regler

Kühler

Erhitzer

Luftfilter



Wäscher

Dampf-befeuchter

Wärmerück-gewinnung

Rauchmelder

Rauchmelder DIN

Lüftungspfeil

Abbildung 2.8: Schablone: CT_LUFT.VSS

CT_MOTOR

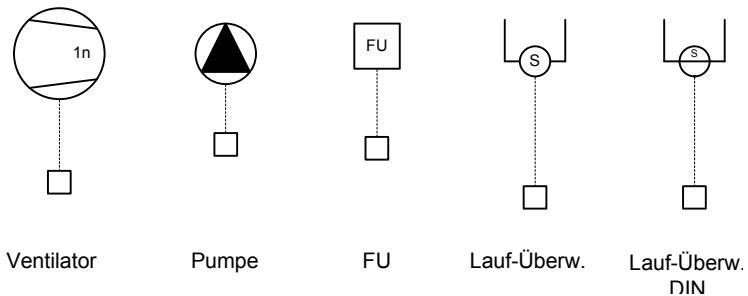


Abbildung 2.9: Schablonen: CT_MOTOR.VSS

CT_STELL

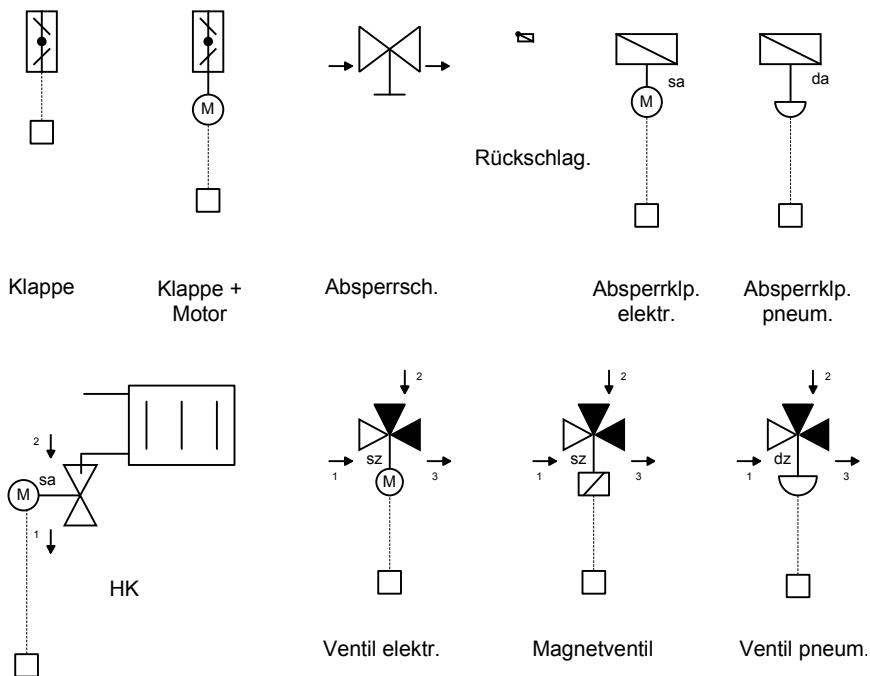


Abbildung 2.10: Schablonen: CT_STELL.VSS

CT_ZEICH

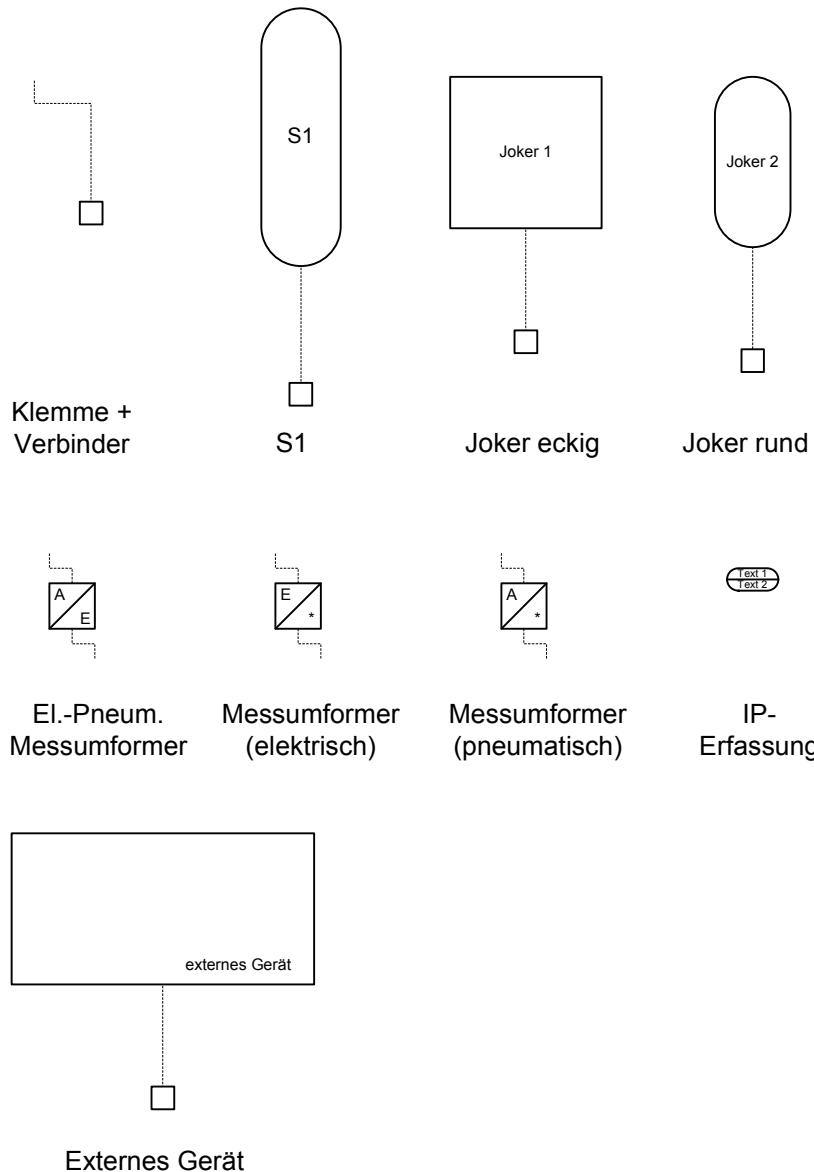


Abbildung 2.11: Schablone: CT_ZEICH.VSS

CT_WASSER

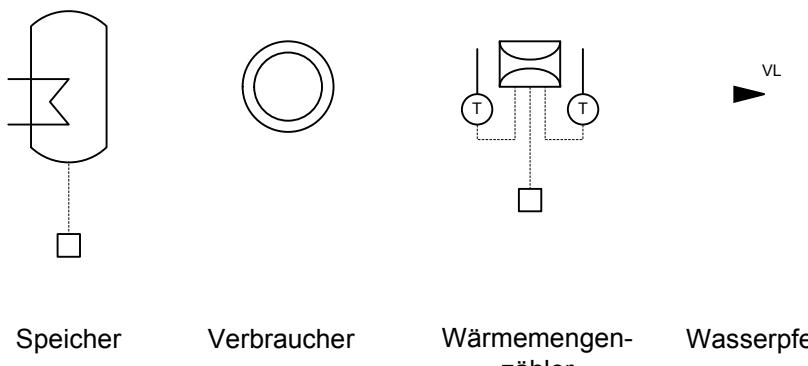


Abbildung 2.12: Schablone: CT_WASSER.VSS

Allgemein

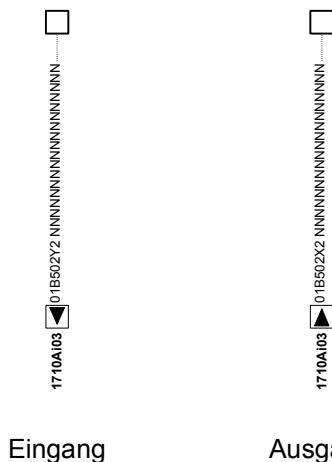


Abbildung 2.13: Schablone: Allgemein.VSS

2 PRAKTIKUMSSCHWERPUNKT 1 – ARBEIT AN DEN SYMBOLEN

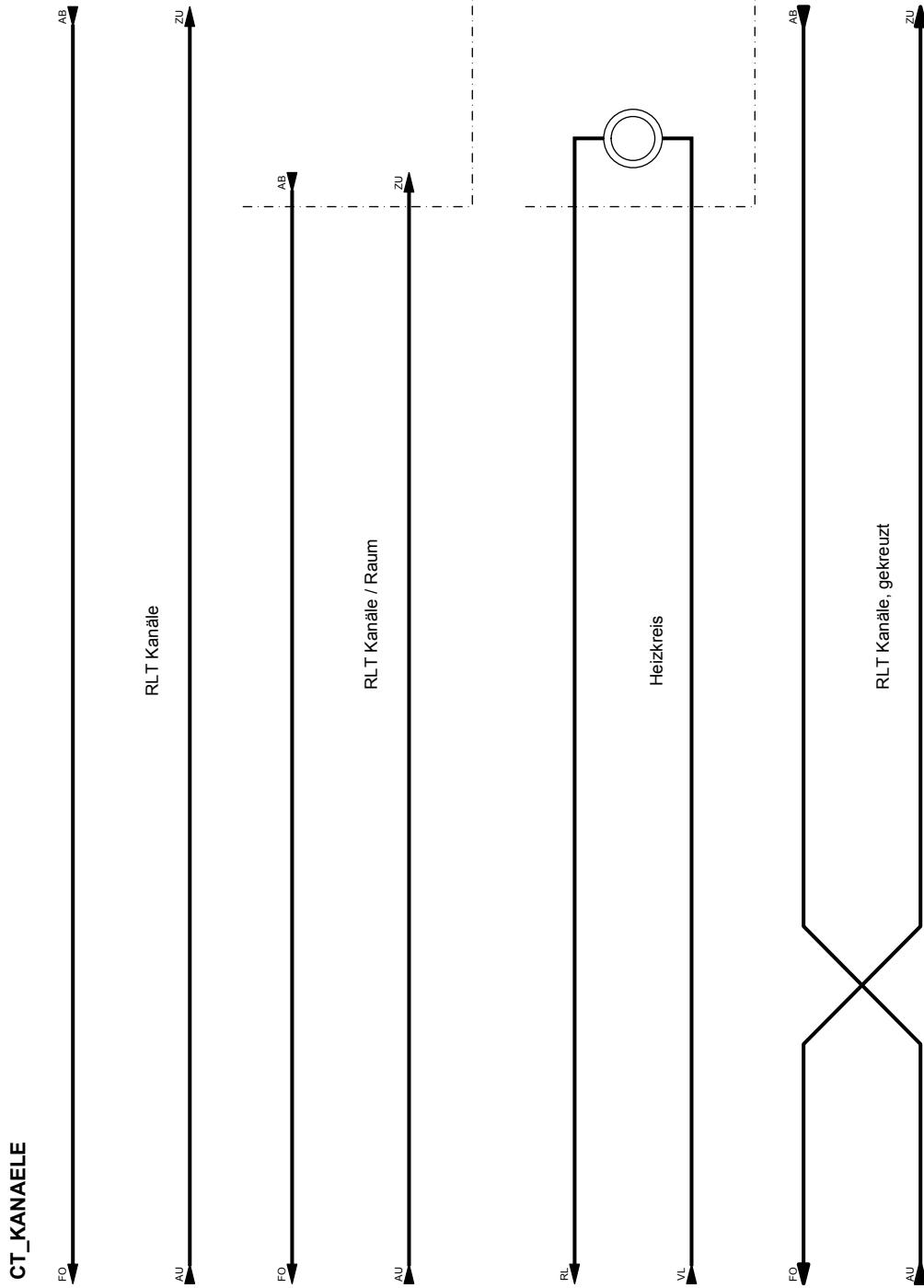


Abbildung 2.14: Schablone: CT_KANAELE.VSS

2 PRAKTIKUMSSCHWERPUNKT 1 – ARBEIT AN DEN SYMBOLEN

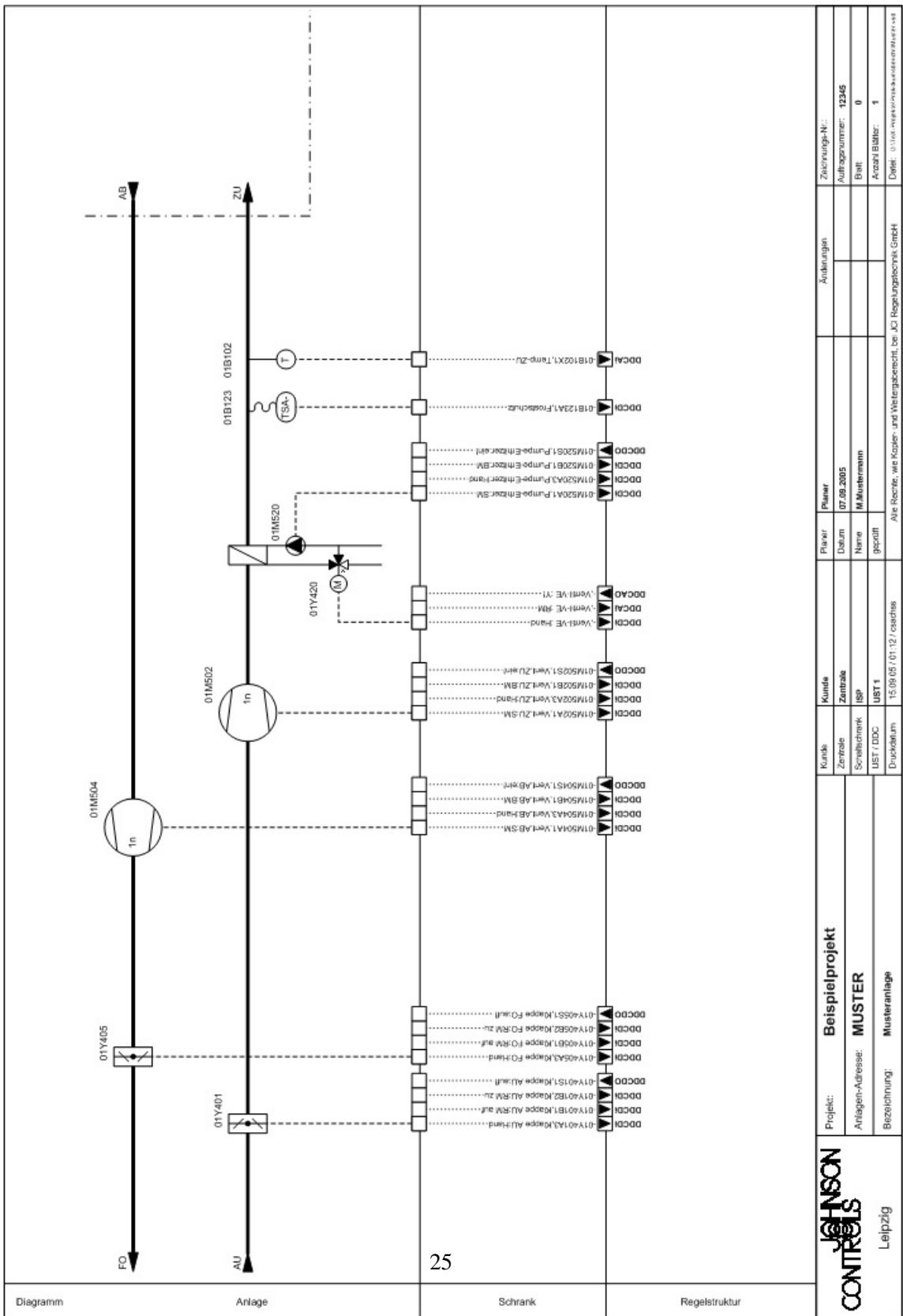


Abbildung 2.15: Beispielregelschema

2.6 sonstige Programmieraufgaben

Während meines gesamten Praktikums programmierte ich ab und an in VBA kleinere Funktionen für Excel und Visio. Dies waren keinerlei geplante Arbeiten, ich erledigte das, was gerade anfiel – Fehlerbeseitigung oder Implementation von Funktionen, die den Mitarbeitern das Arbeiten mit den entsprechenden Programmen erleichtern.

Ein Beispiel für eine solche Funktion ist das automatische Sortieren von Tabellenblättern in einer Excel-Arbeitsmappe. Die Mitarbeiter, die Regelschemen zeichnen arbeiten mit mehr als 15 Tabellenblättern, die ungeordnet nebeneinander liegen. Benötigt der Mitarbeiter eine bestimmte Tabelle, muss er nun nach rechts und links scrollen, um die Tabelle zu finden. Ein einfacher in VBA implementierter Bubblesort, der beabsichtigt Groß- und Kleinschreibung missachtet löste das Problem. Die Funktion wurde mit in das Hauptmenü integriert.

Eine Funktion für Visio, die die verschiedenen ShapeSheets auf einem Zeichenblatt und deren Gruppenbeziehungen untereinander anzeigt wurde auch benötigt. Mit einem entsprechenden rekursiven Aufruf des Unterprogramms ließ sich eine hierarchische Darstellung erzeugen.

Außerdem informierte ich mich über die Möglichkeiten des Passwort-Schutzes von Internetseiten, da sowohl das Visio-Programm als auch Symbole bis dahin noch „ungesichert“ auf einem privaten Webserver lagen. Ich richtete diesen Schutz zusammen mit dem Mitarbeiter ein. Nun gelangen nur noch autorisierte Nutzer an die Daten.

3 Praktikumsschwerpunkt 2 – Arbeit am Hilfesystem

3.1 Problemstellung

Für das Visio-Programm gab es eine Online-Hilfe, die aus dem Programm kontextsensitiv aufgerufen werden konnte. Dies waren .rtf-Dateien, die dann im vorhandenen Editor geöffnet wurden. Das Problem an dieser Art der Hilfe war, dass jeder User in den Dateien ändern und somit auch wichtige Informationen versehentlich löschen konnte. Außerdem waren die Hilfeseiten vom Design her nicht einheitlich.

3.2 Ziel

Das Ziel meines zweiten Praktikumsschwerpunktes war es herauszufinden, auf welche Art und Weise sich Hilfesysteme realisieren lassen und welches davon passend wäre. Schließlich sollte die alte Hilfe dann transformiert werden. Dabei musste sie auf Verständlichkeit, Einheitlichkeit und richtige Schreibung überprüft werden. Eine weitere Anforderung an das neue Hilfesystem war, dass sich die Hilfe vom Entwickler des Programms leicht erweitern und anpassen lässt.

3.3 Vorgehensweise

Zunächst informierte ich mich im Internet und bei vorhandenen Programmen auf meinem Computer darüber, wie Hilfesysteme realisiert werden. Im Anschluss daran erkundigte ich mich, wie die Systeme realisiert werden und wog ab, welche Methode sich am besten für die Überarbeitung der alten Hilfe eignet. Ich entschied mich für die Umsetzung in HTML-Help.

3.4 Realisierung

3.4.1 Analyse der Möglichkeiten

Die Anforderungen an das neue bzw. überarbeitete Hilfesystem waren vor allem, dass die einzelnen Hilfeseiten kontextsensitiv aus Visio per VBA angesprungen werden können. Die Hilfeseiten mussten auf jedem Rechner, an dem mit Visio gearbeitet wird, betrachtbar sein und die Einarbeitungszeit in das neue System sollte so gering wie möglich sein.

Nach meiner umfangreichen Recherche gelangte ich zu dem Ergebnis, dass ich die Wahl zwischen vier möglichen Hilfesystemen hatte. Realisierung der Hilfe

- in einer eigenständigen Programmierprache (z.B. Java) als „externes Programm“
- mittels Content-Management-System
- mit Win-Help
- mit HTML-Help

Zur Entscheidungsfindung betrachtete ich die einzelnen Möglichkeiten separat und wog die Kosten, sowie Benutzerfreundlichkeit und Änderbarkeit gegen den Nutzen ab.

Betrachten wir also nun die vier Möglichkeiten:

Die Entwicklung des *Hilfesystems in einer Programmiersprache* bietet die Möglichkeit ein komplexes eigenständiges Programm zu schaffen, dass auch ohne die Benutzung von Visio aufgerufen werden kann. Zur Durchführung dieser Variante wird eine Entwicklungsumgebung benötigt, sowie werden Kenntnisse in der entsprechenden Programmiersprache vorausgesetzt. Außerdem muss es möglich sein, dass die Hilfethemen kontextsensitiv aufgerufen werden können, was die Wahl der Programmiersprache beeinflusst. Diese Möglichkeit ist für das vorhandene Hilfesystem deshalb nicht geeignet.

Ein *Content-Management-System* zu verwenden ist wohl die eleganteste Lösung für das Problem. Allerdings arbeitet nur eine Person an der Hilfe für Visio, weswegen zahlreiche Funktionen eines CMS nicht genutzt werden. Vor allem aber aus dem Grund, dass es sich um komplexe Systeme handelt, für die erhebliche Einarbeitungszeit notwendig ist, ist diese Lösung nicht geeignet.

Die Realisierung des Hilfesystems in *Win-Help* ist die am nahe liegendste. Ausgangspunkt für die Erstellung eines Hilfesystems sind .rtf-Dateien, die ja schon vorhanden sind. Mittels spezieller Formatierungen und Fußnoten werden in das Dokument Hyperlinks eingefügt und die einzelnen Seiten zusammengesetzt. Es entsteht ein Hilfesystem mit Inhaltsverzeichnis, welches wie gehabt mit VBA kontextsensitiv angesprungen werden kann. Zur Realisierung mit Win-Help ist der Microsoft Help Workshop notwendig, welcher kostenlos aus dem Internet heruntergeladen werden kann. Alles in allem ein für die Überarbeitung der Hilfe geeignetes System, allerdings ist dies eine ältere Methode, die von HTML-Help abgelöst wurde.

HTML-Help ist wie schon erwähnt der Nachfolger von Win-Help und funktioniert auf ähnliche Weise. Ausgangspunkt sind hier .html-Dateien, die auf bekannte Weise untereinander verlinkt sein können – alle aus HTML bekannten Konzepte lassen sich hierbei verwenden. Zusätzlich werden zahlreiche andere Formate, wie beispielsweise ActiveX, Java, Scriptssprachen (z.B. VBScript) und Bildformate unterstützt. Zur Entwicklung der Hilfe wird der HTML Help Work-

shop benötigt, welcher kostenlos aus dem Internet herunterladbar ist.¹ Mit diesem Werkzeug wird ein Inhaltsverzeichnis erstellt, es können die einzelnen Seiten zu „Themen“ zusammengefasst und es kann ein Suchindex erstellt werden. Das Anspringen der entsprechenden Seite aus Visio heraus lässt sich wie gehabt mit VBA realisieren. Zu einem vorhandenen Hilfesystem können einfach neue Seiten und Indizes hinzugefügt oder aber veraltete Seiten gelöscht oder geändert werden. Aus diesen Gründen entschied ich mich für die Realisierung des Hilfesystems in HTML-Help.

3.4.2 Erstellung des Hilfesystems

Vorverarbeitung

Zunächst mussten die vorhandenen .rft-Hilfedateien in .html-Dateien umgewandelt werden. Dies lässt sich zwar recht einfach über die in Microsoft Word vorhandene Funktion realisieren, was sich aber in meinem Fall als nicht brauchbar erwies, da die Hilfeseiten nicht einheitlich waren. Zum großen Teil hatte man sich zwar um Einheitlichkeit bemüht, es gab aber auch Seiten, die „nur schnell dahingeschrieben“ waren und so nicht ins neue Hilfesystem übernommen werden konnten.

Außerdem mussten die vorhandenen Seiten auf korrekte Rechtschreibung überprüft werden. Umlaute waren nicht als Umlaute geschrieben, sondern als ae, ue, oe was die Leserlichkeit beeinträchtigte. Also mussten die „richtigen“ Umlaute hinzugefügt werden. Dabei war zu beachten, dass in .html-Dateien Sonderzeichen und Umlaute aus Kompatibilitätsgründen besonders gekennzeichnet werden sollten.

Designrichtlinien

Bei der Realisierung als .html-Seiten sind bekanntermaßen die Möglichkeiten nicht begrenzt – von einfacher schwarzer Schrift auf weißem Grund bis hin zur flippig-bunten Seite mit Bildern, Tönen und Animationen ist alles möglich. Es handelt sich bei den Seiten aber um einzelne Hilfeseiten, weswegen diese gut lesbar und vor allem übersichtlich gehalten werden müssen.

Einfach nur schwarz-weiß war mir aber zu „langweilig“, weswegen ich eine Grafik geschaffen habe, die als Hintergrundbild für die jeweilige Überschrift der Hilfeseite dient. Passend zur Firmenfarbe „royalblau“ ist die Grafik blau mit weiß gehalten, worauf die schwarze Schriftfarbe der Überschrift ausreichend Kontrast gibt, also gut zu erkennen ist.

Die Breite des Textes ist auf maximal 500 Pixel festgelegt, da sich links neben den Hilfeseiten das Inhaltsverzeichnis befindet. Außerdem wird die Hilfe sehr wahrscheinlich oft im Zusammenhang mit Visio geöffnet sein und somit wäre es unvorteilhaft, wenn die Hilfeseiten die gesamte Bildschirmbreite ausfüllen.

¹<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/hwMicrosoftHTMLHelpDownloads.asp>

Anschließend legte ich Richtlinien für die Schrift und Hervorhebungen fest. Als Schriftart wird nun generell Verdana verwendet, da diese zu den Standard-Schriftarten gehört. Außerdem ist Verdana am Bildschirm sehr gut lesbar, da zu ihren Eigenschaften eine lange Schriftweite und Serifenlosigkeit gehören. Bei den Hervorhebungen orientierte ich mich an den vorhandenen Konventionen, vereinheitlichte aber alles:

Überschriften	schwarz und fett
Wichtiges	rot (#ff0000) oder rosa (#ff00ff)
Tastenbezeichnungen	blau (#0000ff)
Hinweise	grün (#009900) und evtl. fett
Hervorhebungen	orange (#ff9900)

Tabelle 3.1: Schrifthervorhebungen

Vorlagefile

Als nächstes schuf ich eine Vorlage für die .html-Seiten im Editor, um das generelle „gleiche Aussehen“ der einzelnen Hilfeseite zu gewährleisten. Außerdem wird dem Programmierer des Visio-Programms und somit auch des „Hilfe-Verantwortlichen“ die Arbeit erleichtert, denn er braucht die Vorlage lediglich mit Inhalt füllen und sich nicht darum kümmern, dass das Layout stimmt. Er muss lediglich die Einhaltung der Designrichtlinien für Hervorhebungen und farbliche Gestaltung beachten, weswegen die Regeln hierzu oben in der Vorlage als Kommentare mit angegeben sind.

Kategorisierung, Index, Referenzen

Die einzelnen, nun fertigen Hilfeseiten mussten noch geordnet werden, damit sie dann im Hilfe-File zu Kategorien zusammengefasst und in eine Reihenfolge gebracht werden können. Außerdem waren die „Schlüsselworte“ für den Index und die Referenzen der Seiten untereinander festzulegen. Diesen Schritt konnte ich nur in Zusammenarbeit mit dem Programmierer durchführen, da mir das Wissen über die technischen Zusammenhänge und Abläufe der Regelungstechnik fehlen.

Mit der Ordnung der einzelnen Hilfeseiten und Festlegung der Indizes kann das spätere Hilfesystem auch als eigenständiges System betrachtet und unabhängig vom Visio geöffnet werden. Die Verlinkung der Seiten untereinander und über das Inhaltsverzeichnis hilft dem Anwender bei der Beantwortung seiner Fragen durch die Hilfe.

Zusammensetzen der Hilfe

Ich verwendete ein Tutorial, um mich in HTML Help Workshop einzuarbeiten. Dazu nahm ich die .html-Seiten, auf denen die Arbeit mit HTML Help Workshop beschrieben ist und setzte

diese wie beschrieben zusammen. Somit hatte ich am Ende ein Tutorial über die Erstellung eines Hilfesystems mit HTML Help Workshop als Hilfesystem an sich. Dies war ein guter Weg das neue Programm kennenzulernen und nach kurzer Zeit war es mir möglich das neue Hilfesystem zusammenzusetzen und Indizes hinzuzufügen.

Nun musste nur noch das Programm an den entsprechenden Stellen modifiziert werden, um die neuen Hilfeseiten entsprechend „anspringen“ zu können.

Ich zeigte dem Programmierer, wie der Workshop zu verwenden ist und überließ ihm eine Kopie des Tutorials als Hilfesystem zur Gedankenstütze.

3.5 Ergebnisse

Nachfolgend eine Auflistung der Ergebnisse meiner Arbeit in diesem Problembereich:

- überarbeitetes Hilfesystem in HTML-Help
- Designvorschriften für die Hilfeseiten
- Vorlagefile für leichtes Ändern und Ergänzen für den Programmierer

Abbildung 3.1 zeigt die Ansicht des neuen Hilfesystems anhand einer Beispieleseite.

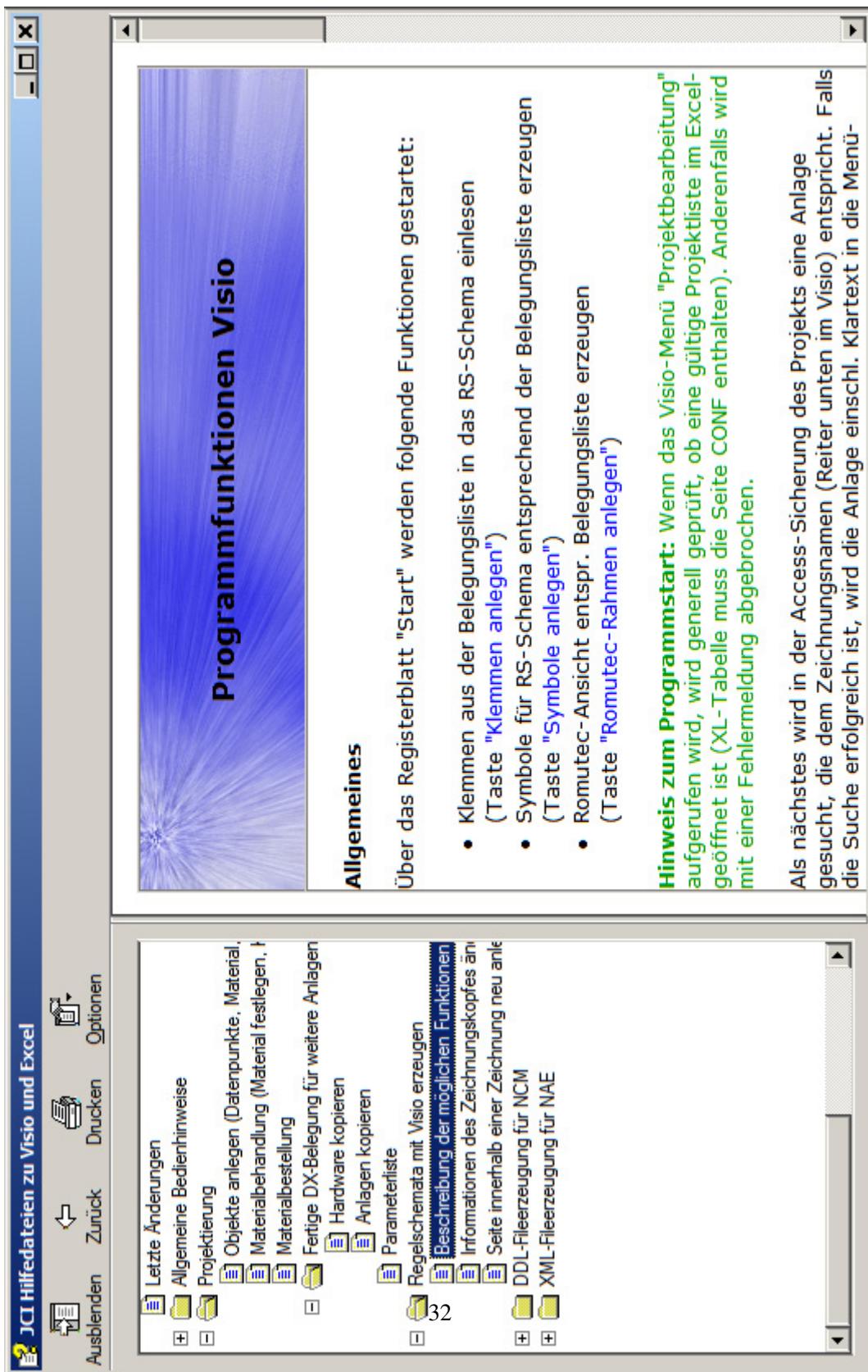


Abbildung 3.1: Ansicht des neuen Hilfesystems

4 Schlussfolgerung

Im Praktikum konnte ich mein erlerntes Wissen aus dem Studium in der Praxis anwenden und vertiefen.

Mein Wissen aus den Veranstaltungen „Mathematische Grundlagen der Computergrafik“ und „Computergrafik 1“ konnte ich bei der Drehung der Shapes anwenden. Außerdem war dies nötig für die Umrechnung der Koordinaten zwischen den verschiedenen Koordinatensystemen.

Wie in den Veranstaltungen „Softwaretechnologie 1“ und „Softwaretechnologie 2“ theoretisch kennengelernt, bin ich am Anfang meines Praktikums mit den Problemen eines sog. Alt-Systems konfrontiert worden: ein schon lange existierendes System ohne bzw. mit sehr wenigen Kommentaren mit dem nun Probleme auftreten und der ursprüngliche Programmierer ist nicht mehr erreichbar.

Normalerweise muss eine Firma in einem solchen Fall viel Geld bezahlen, bis sich der entsprechende neue Programmierer in das alte System eingearbeitet hat oder es werden Leute eingestellt, die den Quellcode analysieren. Aus diesem Grund habe ich meine Arbeit an den Shapes im ShapeSheet direkt kommentiert und separat sorgfältig dokumentiert.

Selbstverständlich konnte ich auch mein Wissen über Programmierung, welches an der TU Chemnitz seit der Veranstaltung „Algorithmen und Programmierung“ in vielen weiteren Vorlesungen erweitert bzw. gefordert wird bei der Programmierung mit Visual Basic for Applications anwenden.

Beim Verfassen der Dokumentation / Anleitung konnte ich das Wissen aus den Veranstaltungen „Typografie und Gestaltung“ für das Layout der Dokumentation und aus „Mediengestaltung“ für das Design der Hilfeseiten und die Userführung und das interaktive Lernen in der Anleitung anwenden.

Schlussendlich hat mir das Praktikum auch einen Einblick in die Arbeitswelt gegeben.

Anhang

A Praktikumsbescheinigung

Herr/Frau

geboren am in

wohnhaft in

wurde vom bis

als Praktikant wie folgt beschäftigt:

von

bis

Wochen

Art der Tätigkeit

Gesamte Wochenzahl:

Fehltage während der Beschäftigungsdauer, davon Tage Krankheit

Tage sonstiger Abwesenheit

Der Praktikumsbericht wird bestätigt.

.....
Firmenstempel und Unterschrift

Ort, Datum:

Dieses Praktikum wird mit Wochen anerkannt.

Chemnitz, den

.....
Stempel und Unterschrift
Vorsitzender des Prüfungsausschusses