DIPLOMARBEIT

zur Erlangung des akademischen Titels
DIPLOM-INFORMATIKER

DEKOMPOSITION VON SOFTWARE-SYSTEMEN IN FUNKTIONSKOMPONENTE UND NUTZEROBERFLÄCHE IM FORWARD UND REENGINEERING

Thomas Kullmann Günther Reinecker

Januar 2004

Gutachter: Prof. Dr. sc. nat. Klaus Bothe,
Dipl. Inf. Kay Schützler

Humboldt-Universität zu Berlin
Math.-Naturwiss. Fakultät II
Institut für Informatik
Lehrstuhl für Softwaretechnik und Theorie der Programmierung

Danksagung

Unser besonderer Dank gilt Herrn Prof. Bothe, Herrn Schützler und Herrn Sacklowski für die Betreuung sowie den Mitarbeitern des Physikalischen Institutes, Prof. Dr. Rolf Köhler, Dr. Hanke, Frau Richter und Herrn Panzner für die gute Zusammenarbeit. Sie unterstützten uns bei technischen Fragen und stellten uns, so oft es ging, die Messplätze zur Verfügung.

Außerdem möchten wir unseren Eltern, Geschwistern und Lebensgefährtinnen für ihre Unterstützung während des Studiums sowie Herrn Sändig recht herzlich für die orthographische Durchsicht danken.

Selbständigkeitserklärung

Hiermit versichern wir, dass wir die vorliegende Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Quellen erstellt haben.

Oranienburg, 28.01.2004 Thomas Kullmann, Günther Reinecker

Einverständniserklärung

Wir erklären hiermit unser Einverständnis unsere Diplomarbeit in der Bibliothek des Institutes für Informatik öffentlich auszustellen.

Oranienburg, 28.01.2004 Thomas Kullmann, Günther Reinecker

INHALT

TABELLENVERZEICHNIS II
ABBILDUNGSVERZEICHNIS III
ABKÜRZUNGSVERZEICHNIS V

I EINFÜHRUNG 1
I.1 Zusammenfassung 1
I.1.1 Aufgabenteilung 2
I.2 Kapitelübersicht 3
I.3 Fallbeispiel: das ‚XCtl'-System 4
I.3.1 Allgemeine Beschreibung 4
I.3.2 Subsystem ‚Manuelle Justage' 6
I.3.3 Subsystem ‚Topographie' 7
I.4 Motivation 9
I.4.1 Bewältigung der Programmkomplexität 9
I.4.2 Austausch, Flexibilität und Wartung der Nutzeroberfläche 12
I.4.3 Austausch und Wartung der Funktionskomponente 14
I.4.4 Kombinierbarkeit von Programmiersprachen 15
I.4.5 Makrosteuerung 16

II DEKOMPOSITION BEIM FORWARD ENGINEERING 17
II.1 Analysephase 18
Anwendungsfall: ‚Manuelle Justage' 18
II.2 Definitionsphase 21
II.2.1 Aufbau von Pflichtenheften 21
Anwendungsfall: ‚Manuelle Justage' 24
II.2.2 Oberflächen-Prototyping 28
Anwendungsfall: ‚Manuelle Justage' 29
II.2.3 Hilfe-System 31
Anwendungsfall: ‚Manuelle Justage' 31
II.2.4 Benutzerhandbuch 34
Anwendungsfall: ‚Manuelle Justage' 35
II.2.5 Objektorientierte Analyse 38
Anwendungsfall: ‚Manuelle Justage' 38
II.2.6 Zusammenfassung 41
II.3 Designphase 44
II.3.1 Verfeinerung der Funktionskomponente 44
Anwendungsfall: ‚Manuelle Justage' 44
II.3.2 Ableitung der Oberflächenfenster aus dem Pflichtenheft 48
Anwendungsfall: ‚Manuelle Justage' 48
II.3.3 Anbindung der Nutzeroberfläche an die Funktionskomponente 50
II.3.3–1 Ausgangspunkt: ADV-Modell 50
II.3.3–2 Anwendung bei der Dekomposition 51
II.3.3–3 Das Observer-Muster 51
Anwendungsfall: ‚Manuelle Justage' 53
II.3.3–4 Die Polling-Variante 56
Anwendungsfall: ‚Manuelle Justage' 57
II.3.3–5 Kardialitäten von Nutzeroberflächen und Funktionskomponenten 59
II.3.4 Aufbau von Designdokumenten 62
Anwendungsfall: ‚Manuelle Justage' 62
II.3.5 Zusammenfassung 64
II.3.5–1 Bewertung von Observer-Muster und Polling im Anwendungsfall 65

II.4 .... Implementierungsphase 68
II.4.1 Schwerpunkte bei der Funktionskomponente 68
Anwendungsfall: ‚Manuelle Justage' 68
II.4.2 Schwerpunkte bei der Nutzeroberfläche 74
Anwendungsfall: ‚Manuelle Justage' 74
II.4.3 Zusammenfassung 79
II.4.3–1 Bewertung von Observer-Muster und Polling im Anwendungsfall 80
II.5 Testphase 82
II.5.1 Test einer Funktionskomponente 82
Anwendungsfall: ‚Manuelle Justage' 83
II.5.2 Test einer Nutzeroberfläche 89
Anwendungsfall: ‚Manuelle Justage' 89
II.5.3 Zusammenfassung 94
II.6 Fazit und Ausblick 95

III DEKOMPOSITION BEIM REENGINEERING 97
III.1 Ist-Analyse 99
Anwendungsfall: ‚Manuelle Justage' 99
Anwendungsfall: ‚Topographie' 100
III.2 Restructuring 103
III.2.1 Allgemeine Schrittfolge für die Dekomposition 103
Anwendungsfall: ‚Manuelle Justage' 114
Anwendungsfall: ‚Topographie' 122
III.2.2 Zusammenfassung 130
Anwendungsfall ‚Manuelle Justage' 130
III.2.2–1 Bewertung von Observer-Muster und Polling im Anwendungsfall 133
Anwendungsfall ‚Topographie' 135
III.3 Fazit und Ausblick 137

IV GESAMTFAZIT 139
ANHANG A – LITERATURVERZEICHNIS 141
ANHANG B – VERWENDETE MATERIALIEN 142
ANHANG C – STICHWORTVERZEICHNIS 144
ANHANG D – PFLICHTENHEFT NEUE ‚MANUELLE JUSTAGE', VERSION 2.1 147
ANHANG E – BEWERTUNG DER NEUENTWÜRFE DES OBERFLÄCHENFENSTERS ZUR ‚MANUELLE JUSTAGE', VERSION 1.7 167
ANHANG F – XCTL-STEUERPROGRAMM, BENUTZER-LEITFADEN DER BASISANWENDUNG: NEUE ‚MANUELLE JUSTAGE', VERSION 1.6 175
ANHANG G – REGRESSIONS-TESTFÄLLE NEUE „MANUELLE JUSTAGE, VERSION 1.5 193
ANHANG H – DESIGNBESCHREIBUNG NEUE ‚MANUELLE JUSTAGE', VERSION 2.0 203
ANHANG I – DESIGNBESCHREIBUNG ZUM REENGINEERING DER ‚TOPOGRAPHIE', VERSION 1.4.. 221
ANHANG J – CTE-DIAGRAMME DER NEUEN ‚MANUELLEN JUSTAGE' 233
ANHANG K – CTE-DIAGRAMME DER NEUEN ‚TOPOGRAPHIE' 239

TABELLENVERZEICHNIS

Tab. II.1 Übersicht zur ursprünglichen ‚Manuellen Justage' 19
Tab. II.2 Ausgewählte Metriken für die Klassen der ursprünglichen ‚Manuellen Justage' 20
Tab. II.3 Ausgewählte Metriken für die neue ‚Manuelle Justage' (nach der Analyse- und Definitionsphase) 40
Tab. II.4 Abstrakte Methoden des Beobachters `TManJustage` 54
Tab. II.5 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Polling-Variante am Ende der Designphase) 66
Tab. II.6 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Observer-Muster am Ende der Designphase) 67
Tab. II.7 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Polling-Variante nach Abschluss der Implementierungsphase) 80
Tab. II.8 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Observer-Muster nach Abschluss der Implementierungsphase) 81
Tab. II.9 Ausgewählte Metriken der Testoberfläche 87

Tab. III.1 Ausgewählte Metriken für die Klassen der ursprünglichen ‚Topographie' 101
Tab. III.2 Übersicht zur ursprünglichen ‚Topographie' 102
Tab. III.3 Übersicht zur getrennten ‚Manuellen Justage' (Polling-Variante) 130
Tab. III.4 Ausgewählte Metriken für die Klassen bei der getrennten ‚Manuellen Justage' (Polling-Variante) 130
Tab. III.5 Identisches Funktionsangebot von `TManJustage` und `TAngleCtl`, Benennung wurde von `TManJustage` übernommen = 39 Methoden 132
Tab. III.6 Zusätzlich geforderte Funktionalität für `TManJustage` = 30 Methoden 132
Tab. III.7 Durch die Dekomposition entstandene Methoden von `TAngleCtl`, die beim Einsatz zu beachten sind = 5 Methoden 132
Tab. III.8 Ausgewählte Metriken für die Klassen bei der getrennten ‚Manuellen Justage' (Observer-Muster) 134
Tab. III.9 Übersicht zur getrennten ‚Topographie' 135
Tab. III.10 Ausgewählte Metriken für die Klassen der getrennten ‚Topographie' 136

ABBILDUNGSVERZEICHNIS

Abb. I.1 Schematischer Aufbau eines Topographiearbeitsplatzes (Quelle: nach [M2]) 4
Abb. I.2 ‚XCtl'-Programm mit Zähler- und AreaScan-Fenster und Dialog zur Konfiguration der Detektoren (Quelle: ‚XCtl'-Programm) 5
Abb. I.3 Use-Case-Diagramm des ‚XCtl'-Gesamtsystems, rot = in dieser Arbeit bearbeitete Subsysteme (Quelle: nach [M1]) 6
Abb. I.4 Freiheitsgrade der Probe bei der Topographie (Quelle: nach [M1]) 7
Abb. I.5 Arbeitsplatz RTK14 am Physikalischen Institut 8
Abb. I.6 Bewältigung der Programmkomplexität durch Zerlegung in zwei Subsysteme, Modularisierung und Dekomposition in GUI- und Funktionskomponentenklasse 9
Abb. I.7 Schichten-Architektur am Beispiel der getrennten ‚Topographie' 10
Abb. I.8 Evolution der Nutzeroberfläche – Oberflächen-Prototyping 12
Abb. I.9 Beispiel der Schriftformatierung über eine individualisierbare Symbolleiste (oben) im Vergleich zum Ausschnitt des Dialogfensters ‚Zeichen' (unten) in Microsoft Word 13
Abb. I.10 Übergang von der simulierten Funktionskomponente, über die Wartung, zum Endprodukt 14
Abb. I.11 OLE-Automation mit Microsoft Excel aus einem anderen Programm heraus: 1 – starten; 2 – Datenerfassung; 3 – Diagrammerstellung 16
Abb. II.1 Dialogressource zur ursprünglichen ‚Manuelle Justage' 18
Abb. II.2 Realisierung von Funktionen und Daten als Bedien- und Steuerelemente eines Oberflächenfensters am Beispiel des Dialogfensters ‚Offset für <Antrieb>' 23
Abb. II.3 Auszug aus der Gliederung des Pflichtenheftes [M16]; Kapitel ‚funktionale Beschreibung' 24
Abb. II.4 Auszug aus dem Funktionsteil des Pflichtenheftes [M16] 24
Abb. II.5 Auszug aus dem Datenteil des Pflichtenheftes [M16] 25
Abb. II.6 Auszug aus dem Anhang des Pflichtenheftes [M16] (Übersicht aller Datenelemente); Hervorhebung: schwarz die vorhandenen; blau die abgeleiteten und rot die neuen Elemente 26
Abb. II.7 Auszug aus der Gliederung des Pflichtenheftes [M16]; Kapitel ‚Daten' 26
Abb. II.8 Auszug aus der Gliederung des Pflichtenheftes [M16]; Kapitel ‚Benutzeroberfläche' . 27
Abb. II.9 Auszug aus dem Teil Benutzeroberfläche des Pflichtenheftes [M16] 28
Abb. II.10 Neuentwürfe des Hauptdialogs ‚Manuellen Justage NEU' (oben); Dialog ‚Offset für <PSD>' (links) und Dialog ‚Offset für <Antrieb>' (rechts) 30
Abb. II.11 Entwicklung der ‚XCtl'-Hilfe 32
Abb. II.12 Verlinkung des Hauptdialogs mit den Hilfe-Seiten für die jeweiligen Steuerelemente am Beispiel der Funktion: Relative Null setzen; ein Rahmen entspricht einer Hilfe-Seite .. 33
Abb. II.13 gekürzter Auszug aus dem Benutzer-Leitfaden [M17]; Kapitel ‚Referenzteil' 35
Abb. II.14 Auszug aus dem Referenzteil des Benutzer-Leitfadens [M17] 36
Abb. II.15 Auszug aus dem Trainingsteil des Benutzer-Leitfadens [M17], zur Schrittfolge für „Relative Null von ‚Tilt' aufheben" 37
Abb. II.16 Auszug aus dem Datenteil des Pflichtenheftes [M16] 39
Abb. II.17 Auszug aus dem Funktionsteil des Pflichtenheftes [M16] 39
Abb. II.18 Auszug aus dem Funktionsteil des Pflichtenheftes [M16] 40
Abb. II.19 UML-Klassendiagramm für das OOA-Modell der neuen ‚Manuellen Justage', gegliedert nach der Erstellungsreihenfolge ihrer Elemente 41
Abb. II.20 Gegenüberstellung der von den Autoren vorgeschlagenen Definitionsphase (links) zur Allgemeinen (rechts) 42

Abb. II.21 Allgemeines Implementierungsschema für eine `Set`-Methode, mit `CanSet` 46
Abb. II.22 UML-Klassendiagramm für die Funktionskomponente der neuen ‚Manuelle Justage', gegliedert nach der Bearbeitungsreihenfolge ihrer Elemente .. 47
Abb. II.23 UML-Klassendiagramm für die Dialogfensterklasse `TManJustageDlg` gegliedert, nach der Erstellungsreihenfolge ihrer Elemente 49
Abb. II.24 UML-Klassendiagramm für die Dialogfensterklassen `TMotorOffsetDlg` und `TPsdOffsetDlg`, gegliedert nach der Erstellungsreihenfolge ihrer Elemente 50
Abb. II.25 UML-Klassendiagramm zum allgemeinen Aufbau des Observer-Musters (nach [8]) 52
Abb. II.26 UML-Klassendiagramm für Observer-Muster in der neuen ‚Manuellen Justage' (vereinfacht) 55
Abb. II.27 Interaktionsdiagramm für Observer-Muster in der neuen ‚Manuellen Justage' während der Antriebsbewegung 56
Abb. II.28 UML-Klassendiagramm für Polling in der neuen ‚Manuellen Justage' (vereinfacht) 58
Abb. II.29 Interaktionsdiagramm für Polling in der neuen ‚Manuellen Justage' während der Antriebsbewegung 59
Abb. II.30 Bsp. für hohe Komplexität des Observer-Musters bei 1 : 3 60
Abb. II.31 Bsp. für das einfache Design von Polling bei 1 : 3 60
Abb. II.32 Bsp. für Observer-Muster bei 3 : 1 61
Abb. II.33 Bsp. für Observer-Muster bei 2 : 2 62
Abb. II.34 Verallgemeinerte Einträge für Typen, Attribute, Methoden und Variablen (vgl. oben), nach Auszügen aus [M20] 63
Abb. II.35 Auszug für einen Methoden-Eintrag aus [M20] 64
Abb. II.36 UML-Klassendiagramm für das OOD-Modell der neuen ‚Manuellen Justage' mit Polling-Variante und Übersicht der benutzten Subsysteme 66
Abb. II.37 UML-Klassendiagramm für das OOD-Modell der neuen ‚Manuellen Justage' mit Observer-Muster und Übersicht der benutzten Subsysteme 67
Abb. II.38 Auszug für einen Methoden-Eintrag aus dem Designdokument [M20] 69
Abb. II.39 Auszug aus dem Funktionsteil des Pflichtenheftes [M16] 69
Abb. II.40 Implementierung der Bedingung für den Direktbetrieb, Methode `CanDoDirect` 70
Abb. II.41 Auszug für einen Methoden-Eintrag aus dem Designdokument [M28] des Subsystems ‚Motorsteuerung' 70
Abb. II.42 Fortsetzung der Implementierung des Direktbetriebs, Methode `DoDirect` – unter Benutzung von `CanDoDirect` und Methoden des Subsystems ‚Motorsteuerung' ... 70
Abb. II.43 Absicherung von Lese- und Schreibzugriffen und zur Überprüfung von Parameterübergaben in `TManJustage` 71
Abb. II.44 Überprüfung, ob der übergebene Parameter gültig ist 71
Abb. II.45 Nutzung der Mutator-Methode `SetOffset` anstatt eines Direkt-Zugriffs in der Funktionskomponente `TManJustage` 71
Abb. II.46 UML-Klassendiagramm für die Funktionskomponente der neuen ‚Manuelle Justage' (am Ende der Implementierungsphase) 72
Abb. II.47 Stark vereinfachtes Konzept zur Geschwindigkeitsoptimierung 73
Abb. II.48 Nutzung von Methoden zur Verwaltung der Teilbereiche 75
Abb. II.49 Realisierung der oberflächenspezifischen Funktionen / B 40 / und / B 50 / 76
Abb. II.50 Auszug aus dem Kapitel ‚Benutzeroberfläche' des Pflichtenheftes [M16], der neuen ‚Manuellen Justage' 76
Abb. II.51 Seit dem OOD unverändert: das UML-Klassendiagramm für die Dialogfensterklassen `TMotorOffsetDlg` und `TPsdOffsetDlg` (Implementierungsphase) 78
Abb. II.52 Seit dem OOD leicht verändert: das UML-Klassendiagramm für die Dialogfensterklasse `TManJustageDlg` (Implementierungsphase) 79
Abb. II.53 Auszug aus dem Dateiteil des Pflichtenheftes [M16] 84
Abb. II.54 Auszug aus dem Funktionsteil des Pflichtenheftes [M16] 85
Abb. II.55 Öffentliche Schnittstelle von `TManJustage`, zerlegt nach der Komplexität ihrer Methoden 86
Abb. II.56 Dialog ‚Manuelle Justage Funk-Test für `TManJustageTestDlg` 87
Abb. II.57 UML-Klassendiagramm der ‚Manuellen Justage' mit Testoberflächenklasse `TManJustageTestDlg` 88
Abb. II.58 Ausschnitt aus dem CTE-Diagramm zum Test der Nutzeroberfläche der neuen ‚Manuellen Justage' 91
Abb. II.59 Ausschnitt aus dem Testdokument [M25], Testfall 1 93

Abb. III.1 Vorteile des Reengineering gegenüber dem Forward Engineering (z.T. zusammengetragen aus [4]) 97
Abb. III.2 Dialoge ‚Topographie-Durchführung' (links) und ‚Topographie-Einstellungen' (rechts) 101
Abb. III.3 Konstruiertes Beispiel zur Verschmelzung zweier O-Blöcke (der entstandene F-Block kann als Ganzes als `Do`-Methode ausgelagert werden) 107
Abb. III.4 Konstruiertes Beispiel mit nicht verschiebbaren O-Blöcken (hier werden auch die untergeordneten, bedingten Anweisungsfolgen untersucht) 108
Abb. III.5 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine `Do`-Methode (Typ1) 108
Abb. III.6 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine `Do`-Methode (Typ2) 109
Abb. III.7 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine `Do`-Methode (Typ3) 110
Abb. III.8 Konstruiertes Beispiel für die Auslagerung einer `CanDo`-Methode 111
Abb. III.9 Konstruiertes Beispiel für die Auslagerung einer `CanDo`-Methode 112
Abb. III.10 UML-Klassendiagramm von `TAngleCtlDlg` (Nutzeroberfläche) nach der Dekomposition (mit Zuordnung der ausgelagerten Methoden) 116
Abb. III.11 UML-Klassendiagramm von `TAngleCtl` (Funktionskomponente) nach der Dekomposition 117
Abb. III.12 Beispiel zur Auslagerung einer `CanDo`-Methode 118
Abb. III.13 Beispiel zur Gewinnung zusätzlicher Bedingungen für `CanDo`-Methoden 119
Abb. III.14 UML-Klassendiagramm der getrennten ‚Manuellen Justage' (Polling-Variante) 121
Abb. III.15 UML-Klassendiagramm der Funktionskomponente `TTopographyOld` 122
Abb. III.16 Implementierung des Singleton-Musters für die Klasse `TTopography` 123
Abb. III.17 Zusammenfassen einer `Set`-Methode mit einem großen F-Block 125
Abb. III.18 Durch Dekomposition entstandene `Do`-Methoden mit langen Parameterlisten 125
Abb. III.19 Erweiterung einer `Set`-Methode durch zusätzliche Bedingungen 126
Abb. III.20 Zusammenfassen gleichgesinnter `Do`-Methoden durch zusätzlichen Parameter 127
Abb. III.21 UML-Klassendiagramm der getrennten ‚Topographie' 129
Abb. III.22 UML-Klassendiagramm für die getrennte ‚Manuellen Justage' (Observer-Muster mit Übersicht der benutzten Subsysteme) 134

ABKÜRZUNGSVERZEICHNIS

A

Abb. .... Abbildung
ADO .... **A**bstract **D**esign **O**bject
ADV .... je nach Kontext **A**bstract **D**esign **V**iew bzw. **A**bstract **D**esign **V**iewer
akad. .... akademisch(e|er|es)
API .... **A**pplication **P**rogramming **I**nterface

B

bspw. .... beispielsweise
bzw. .... beziehungsweise

C

ca .... circa
CASE .... **C**omputer **A**ided **S**oftware **E**ngineering
COM .... **C**omponent **O**bject **M**odell
CTE .... **C**lassification **T**ree **E**ditor (proprietäres Testwerkzeug)

D

d.h................ das heißt
DLL .... **D**ynamic **L**ink **L**ibrary

E

etc. .... et cetera

F

f .... folgende
ff .... fortfolgende

G

ggf. .... gegebenenfalls
GUI .... **G**raphical **U**ser **I**nterface

H

Hrsg. .... Herausgeber

I

i.d.R .... in der Regel
insbes. .... insbesondere
ISO .... **I**nternational **O**rganisation for **S**tandardization

M

M. .... **M**ethode(n)
MFC .... **M**icrosoft **F**oundation **C**lass (proprietäre Bibliothek für MVC)
MVC .... **M**icrosoft **V**isual **C**++ (proprietäre Programmiersprache)

O

OLE .... **O**bject **L**inking and **E**mbedding
OOA .... **O**bject-**o**riented **A**nalysis (Objektorientierte Analyse)
OOD .... **O**bject-**o**riented **D**esign (Objektorientiertes Design)
OOP .... **O**bject-**o**riented **P**rogramming (Objektorientierte Programmierung)

S

S. .... Seite(n)

T

Tab. .... Tabelle

U

u.a. .... unter anderem
u.U. .... unter Umständen

V

Verl. .... Verlag
vgl. .... vergleiche

X

XCtl .... steht für **X**-Ray **C**on**t**ro**l** (Steuerung von Röntgenstrahlung zur Untersuchung von Kristallstrukturen an Halbleitern)

Z

z.B. .... zum Beispiel
z.T. .... zum Teil

Kapitel I

EINFÜHRUNG

I.1 Zusammenfassung

Die vorliegende Diplomarbeit entstand im Rahmen des Projektseminars ‚Software-Sanierung' am Lehrstuhl für Softwaretechnik und Theorie der Programmierung des Institutes für Informatik der Humboldt-Universität zu Berlin.

Dieses Projekt wurde im Wintersemester 1998/99 ins Leben gerufen und steht unter der Leitung von Prof. Dr. Klaus Bothe. Gegenstand der Veranstaltung ist die Sanierung und Erweiterung eines Programmes mit der Bezeichnung ‚XCtl', das im Fachbereich ‚Röntgenbeugung an dünnen Schichten', am Physikalischen Institut der Humboldt-Universität entwickelt wurde, um die dortigen Messgeräte zur Halbleiter-Strukturanalyse zu steuern und die Ergebnisse graphisch aufzubereiten.

Während der mehrjährigen Arbeit an diesem Softwareprojekt haben sich dabei zwei Hauptaufgaben herausgebildet. Der erste Schwerpunkt ist das Reengineering einzelner Programmelemente. Dies umfasst das Reverse Engineering, Restructuring und Refactoring. Der zweite Punkt ist die Entwicklung neuer Programmfunktionen im Forward Engineering, mit der Dokumentation der Softwareentwicklungsphasen sowie die Erstellung von Benutzerleitfäden und Online-Hilfen. Dazu werden einzelne Teile dieses Softwareprojektes von ein bis zwei Studenten eigenständig unter bestimmten Gesichtspunkten bearbeitet.

Alle diese Arbeiten entstehen in enger Zusammenarbeit mit den Mitarbeitern des Physikalischen Institutes, die als Fachwissenschaftler bei physikalischen und technischen Problemen zu Rate gezogen werden können. Neben dieser Anwenderklientel arbeiten auch Studenten im Rahmen eines Praktikums an den Messplätzen mit dem Programm und bilden so eine zweite Nutzergruppe.

Die Wartung des ‚XCtl' Projektes bildet die Grundlage für Studien- oder Diplomarbeiten.

Thema dieser Diplomarbeit ist die Trennung (**Dekomposition**) der Nutzeroberfläche von der dazugehörigen **Funktionskomponente** – d.h. Programmfunktionalität. Dazu zählen der Zugriff auf andere Subsysteme/ Hardware, jede von der Oberfläche unabhängige Datenhaltung/-verwaltung sowie das Festhalten des Systemzustands, Berechnungen und Simulationen. Dieses Vorgehen wird sowohl im Forward- als auch im Reengineering untersucht.

Dazu wurde ein im ‚XCtl'-Steuerprogramm vorhandenes Subsystem (‚Manuelle Justage') grundlegend neu analysiert, entworfen, implementiert und getestet. Die theoretischen Ansätze zur Formalisierung des gesamten Softwareentwicklungsprozesses sind daraus abgeleitet. Die gewonnenen Erkenntnisse und zusammengetragenen Fakten werden am klassischen Wasserfall-Modell, d.h. von Analyse und Definition über Design, Implementierung bis hin zum Test, phasenspezifisch angewendet und am Anwendungsfall erläutert.

Im Reengineering wurden die beiden Anwendungsfälle ‚Topographie' und ‚Manuelle Justage' durch Arbeiten am Programmcode schrittweise getrennt, um eine allgemeine Herangehensweise für die Dekomposition zu entwickeln. Als Ergebnis entstand ein generalisiertes Regelwerk, was eine zeitlich effiziente Dekomposition des Programmcodes erlaubt.

In jeder Softwareentwicklungsphase wird dabei auf die jeweiligen Besonderheiten beim Forward- und beim Reengineering eingegangen. Dieses Modell bildet aber nur eine exemplarische Grundlage, da die Inhalte dieser Diplomarbeit auch in alle anderen phasenorientierten Prozess-Modellen, wie Spiral- oder V-Modell, angewendet werden können. Die Projektarbeit diente dabei zur Verifikation und Bewertung der Praxistauglichkeit der hier vorgestellten Ausführungen.

Die Voraussetzungen zum Verständnis dieser Arbeit sind Kenntnisse in der objektorientierten Programmierung sowie das Wissen über Struktur und Arbeitsabläufe bei der Softwareentwicklung nach Prozess- oder Vorgehensmodellen, insbesondere dem Wasserfall-Modell. Die Dekomposition wird im Wesentlichen an einzelnen Subsystemen dargestellt. Diese Ausführungen sind aber so allgemein gehalten, dass die Anwendung der Ausführungen auf ein komplettes Softwareprojekt problemlos möglich sein sollte.

I.1.1 Aufgabenteilung

Die vorliegende Diplomarbeit ist vollständig als Gemeinschaftswerk zu betrachten. Der gesamte Inhalt wurde von beiden Autoren gemeinsam erstellt und bearbeitet. Dies trifft auch auf alle von den Autoren erstellen Dokumente zu. Eine wiederholte abwechselnde Bearbeitung sollte dabei zu gut verständlichen und gleichzeitig hochgradig formalisierten Dokumenten führen (Review).

Um die parallele Entwicklung von Funktionskomponente und Nutzeroberfläche während der Implementierungs- und Testphase zu beweisen, wurde hier eine Aufgabenteilung vorgenommen. Beim Forward Engineering übernahm dazu der Autor Thomas Kullmann die Arbeiten an der Funktionskomponente der neuen ‚Manuellen Justage', während Günther Reinecker die Nutzeroberfläche entwickelte.

Um verallgemeinerbare Erfahrungen über die Trennung im Reengineering zu sammeln, wurden zwei eigenständige Subsysteme des ‚XCtl'-Steuerprogrammes ausgewählt und unabhängig voneinander von je einem Autor dekomponiert. Thomas Kullmann bearbeitete dazu das Subsystem ‚Topographie' (siehe **I.3.3**) und Günther Reinecker die ursprüngliche ‚Manuelle Justage' (siehe **I.3.2**). Die dabei gewonnenen Erkenntnisse wurden analysiert, formalisiert und in einer allgemeingültigen Schrittfolge zusammengefasst.

I.2 Kapitelübersicht

Kapitel I soll eine allgemeine Einführung in den Gegenstandsbereich und das zugrunde liegende Softwareprojekt geben. Dazu werden in Kapitel **I.3 Fallbeispiel: das ‚XCtl'-System** das Softwaresystem als Ganzes und die beiden Anwendungsfälle ‚Topographie' und ‚Manuelle Justage' grob skizziert. Anschließend zeigt Kapitel **I.4 Motivation** eine allgemeine Übersicht über Gründe, Richtlinien und Besonderheiten für die Dekomposition von Nutzeroberfläche und Funktionskomponente. Die untersuchten Aspekte werden auf fünf Abschnitte verteilt behandelt.

Kapitel II ist nach den Softwareentwicklungsphasen gegliedert und beschäftigt sich mit der Dekomposition bei der Entwicklung neuer Subsysteme oder ganzer Projekte im **Forward Engineering**. Am Anfang eines jeden Abschnittes werden die allgemeinen Grundlagen vorgestellt und diskutiert. Anschließend wird ausführlich auf die Besonderheiten und Erfahrungen bei der Neuentwicklung der neuen ‚Manuellen Justage' eingegangen.

Kapitel **II.1 Analysephase** beschäftigt sich mit der Erstellung des Lastenheftes und beschreibt die Einbeziehung existierender Software bei einem Neuentwurf. Kapitel **II.2 Definitionsphase** betrachtet die Dekomposition bei der Erstellung eines Pflichtenheftes bis hin zum Entwurf eines objektorientierten Analyse-Modells. Das folgende Kapitel **II.3 Designphase** bildet den ersten Schwerpunkt dieser Arbeit. Hier werden die Grundlagen zur Dekomposition auf der Designebene vorgestellt und unter Gesichtspunkten wie Zugriffsschutz, Robustheit und Komplexität, allgemein und am Beispiel der neuen ‚Manuelle Justage' bewertet. Durch den starken handwerklichen Charakter der in Kapitel **II.4 Implementierungsphase** beschriebenen wird dort nur kurz auf die Probleme bei der Implementierung von Nutzeroberfläche und Funktionskomponente aus dem Anwendungsfall neue ‚Manuelle Justage' eingegangen.

Im Kapitel **II.5 Testphase** werden, aus dem Anwendungsfall abgeleitete, Verfahren zum unabhängigen Testen von Funktionskomponente und Nutzeroberfläche vorgestellt und bewertet.

Da ein großer Teil des heutigen Software Engineering darin besteht, Alt-Systeme instand zuhalten und zu sanieren, wird in **Kapitel III** die Möglichkeit einer nachträglichen Dekomposition bei existierenden Softwareprojekten, im Reengineering, untersucht. Um möglichst viele Aspekte bei dieser Art der Dekomposition herauszuarbeiten, wird die Trennung der beiden voneinander unabhängigen Subsysteme ‚Topographie' und ‚Manuelle Justage' vorgestellt. Dazu erfolgt in Kapitel **III.1** eine **Ist-Analyse**, speziell für die ‚Topographie'. Danach wird in Kapitel **III.2 Restructuring** eine allgemeine Schrittfolge für die Dekomposition in einem Reengineering-Prozess ausführlich erläutert. Sie bildet den zweiten Schwerpunkt dieser Arbeit und wird exemplarisch jeweils an beiden Subsystemen schrittweise abgearbeitet. So kann der Umgang mit einer Vielzahl von Besonderheiten im Programmcode der Anwendungsfälle bei der Dekomposition erklärt werden.

Im Anhang befinden sich ein **Literaturverzeichnis** mit weiterführenden Nachschlagewerken, welche die Grundlage dieser Arbeit bilden. Die unter **verwendete Materialien** aufgeführten Dokumente wurden für die Arbeiten mit dem ‚XCtl'-Projekt benötigt oder von den Autoren neu erstellt. Im **Stichwortverzeichnis** befindet sich eine alphabetische Liste der verwendeten Terminologie mit entsprechenden Seitenzahlen, wo diese Begriffe eingeführt/ definiert oder erläutert werden.

I.3 Fallbeispiel: das ‚XCtl'-System

I.3.1 Allgemeine Beschreibung

Die Basis der praktischen Arbeit bildet eine Software mit der Bezeichnung ‚XCtl', die zur Steuerung von Messgeräten zur Halbleiter-Strukturanalyse benutzt wird. Dort werden Halbleiterkristalle mit Licht im Röntgenspektrum bestrahlt. Dabei wird ein Teil der Strahlung an der Probe reflektiert und von Detektoren erfasst. Die ‚XCtl'-Software übernimmt die Messwerte, wertet sie aus und stellt sie grafisch dar. So können Aussagen über Struktur und Qualität der Probe getroffen werden.

Das ‚XCtl'-Programm wird bei den Untersuchungsverfahren Röntgendiffraktometrie, Reflektometrie und Topographie (siehe [M1]) eingesetzt, um Defekte und Unregelmäßigkeiten in den Halbleiterkristall-Proben zu untersuchen. Zum Zeitpunkt der Entstehung dieser Arbeit existierten vier unabhängige Messplätze, die jeweils über einen separaten PC gesteuert wurden. Auf jedem der Rechner lief eine eigenständige Version des ‚XCtl'-Steuerprogrammes.

Der folgende Abschnitt beschreibt den Aufbau eines Messplatzes nach **Abb. I.1**. Die von einer Röntgenquelle (**RQ**) ausgehende primäre Röntgenstrahlung (**PS**) wird durch einen oder mehrere Kollimatorkristall(e) (**K**) parallelisiert und monochromatisiert – wahlweise auch verbreitert (aufgefächert). Wenn dieser kollimierte Strahl (**KS**) auf die kristalline Halbleiterprobe (**P**) trifft, wird er entweder an den atomaren Netzebenen gebeugt (Diffraktometrie, Topographie) oder an der Oberfläche (oder Grenzfläche) spiegelnd reflektiert (Reflektometrie). Das jeweilige Verhalten wird durch den Einfallswinkel zwischen Probe und einfallendem Röntgenstrahl bestimmt. Dazu kann die Probe im dreidimensionalen Raum positioniert werden, weil sie auf einem beweglich gelagerten Probenteller (**PH**) liegt.

Die gebeugte bzw. reflektierte Röntgenstrahlung (**RS**) wird bei der Topographie fotografisch (**F**) und bei der Diffraktometrie/ Reflektometrie durch Röntgendetektoren ((**K**)**D**) erfasst. Bei den Detektoren werden die Messdaten digital aufgezeichnet. Das ermöglicht spätere Bildtransformation und die Auswertung am Bildschirm. Wahlweise können die Daten auch in einer Datei abgelegt werden. Die Krümmung des Kollimators und die Antriebe zur Positionierung von Detektor und Probenteller besitzen zahlreiche Stelleinheiten und können mit minimalen Schrittweiten von ca. 1/180.000 Grad bewegt werden.

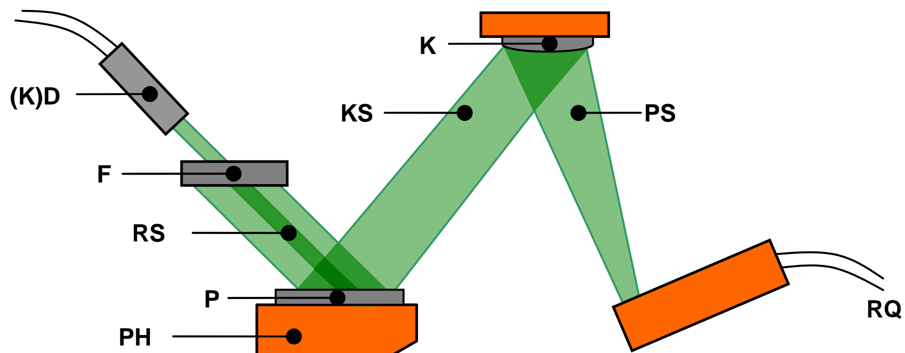


Abb. I.1 Schematischer Aufbau eines Topographiearbeitsplatzes (Quelle: nach [M2])

Ursprünglich handelte es sich bei der ‚XCtl'-Software um eine 16bit-Windows Anwendung, die in der Einsatzumgebung unter dem Betriebssystem Windows 3.11 arbeitete. Die Implementierungssprache für das Projekt ist C++, die Entwicklungsumgebung war Borland, Version 4.5. Die Ausgangsversion des ‚XCtl'-Programmes wurde 1998 durch den Physiker Heiko Damerow, einem damaligen Mitarbeiter im Fachbereich

‚Röntgenbeugung an dünnen Schichten', erstellt und bestand aus vier Subsystemen mit 27.000 LOC. Da diese Version Mängel und Fehler in Design und Implementierung aufwies, der Quellcode kaum kommentiert war und wenige System- und Benutzerdokumentationen existierten, wandte sich der Physikalische Fachbereich an Prof. Dr. Klaus Bothe, mit der Bitte, das Programm zu warten und zu erweitern.

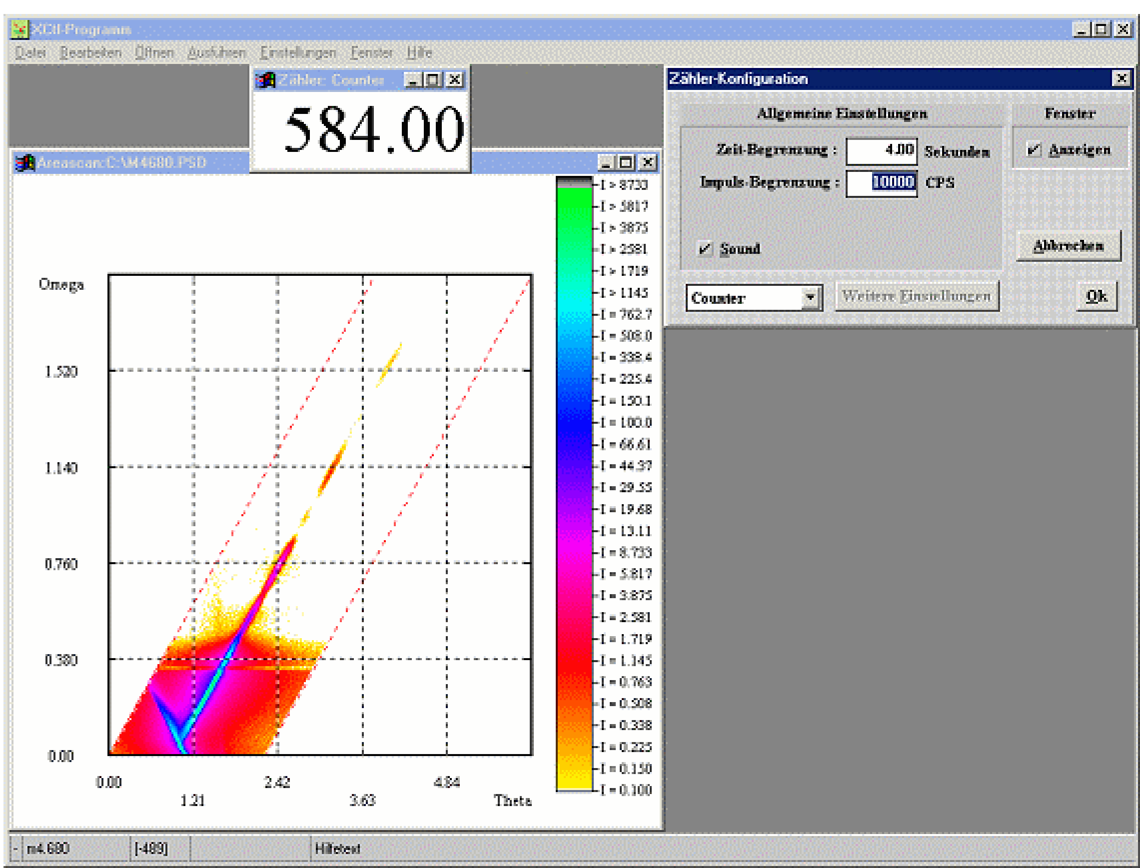


Abb. I.2 ‚XCtl'-Programm mit Zähler- und AreaScan-Fenster und Dialog zur Konfiguration der Detektoren (Quelle: ‚XCtl'-Programm)

Seitdem beschäftigt sich das Projektseminar ‚Software-Sanierung' mit der Analyse, der kontinuierlichen Umstrukturierung und Erweiterung des ‚XCtl'-Programmes. Dazu werden die existierenden Dokumente, der frühen Phasen der Software-Entwicklung, im Reverse Engineering analysiert und ergänzt bzw. angepasst. Fehlende Dokumente werden neu erstellt. Darauf aufbauend wird das Projekt, in Zusammenarbeit mit den Mitarbeitern des Fachbereichs ‚Röntgenbeugung an dünnen Schichten' im Forward Engineering, um neue Funktionalität erweitert. Beim Reengineering der einzelnen Subsysteme erfolgt eine Restrukturierung der Systemarchitektur sowie die Analyse und Verbesserung des Programmcodes und der Wartbarkeit. Der wesentlichste Punkt ist dabei die Fehlerminimierung.

Die aktuell auf den Messplätzen installierte Version (Stand 25.04.2003) umfasst 14 Subsysteme mit 38.786 LOC und 208 Klassen in 128 Dateien. Pro Subsystem sind durchschnittlich drei Dokumente für jede Softwareentwicklungsphase vorhanden. Alle Dokumente sind im Web-**Repository** des ‚XCtl'-Projektes phasenspezifisch für die einzelnen Subsysteme aufgelistet und für jeden Projektteilnehmer frei verfügbar.

Zusätzlich existieren eine Vielzahl Projekt begleitender Dokumente wie Beschreibungen von Arbeitsabläufen, Erläuterungen der genutzten physikalischen Prozesse, Benutzerdokumentationen. **Abb. I.3** zeigt eine Übersicht der Struktur des ‚XCtl'-Programmes von 2002 als Use-Case-Diagramm.

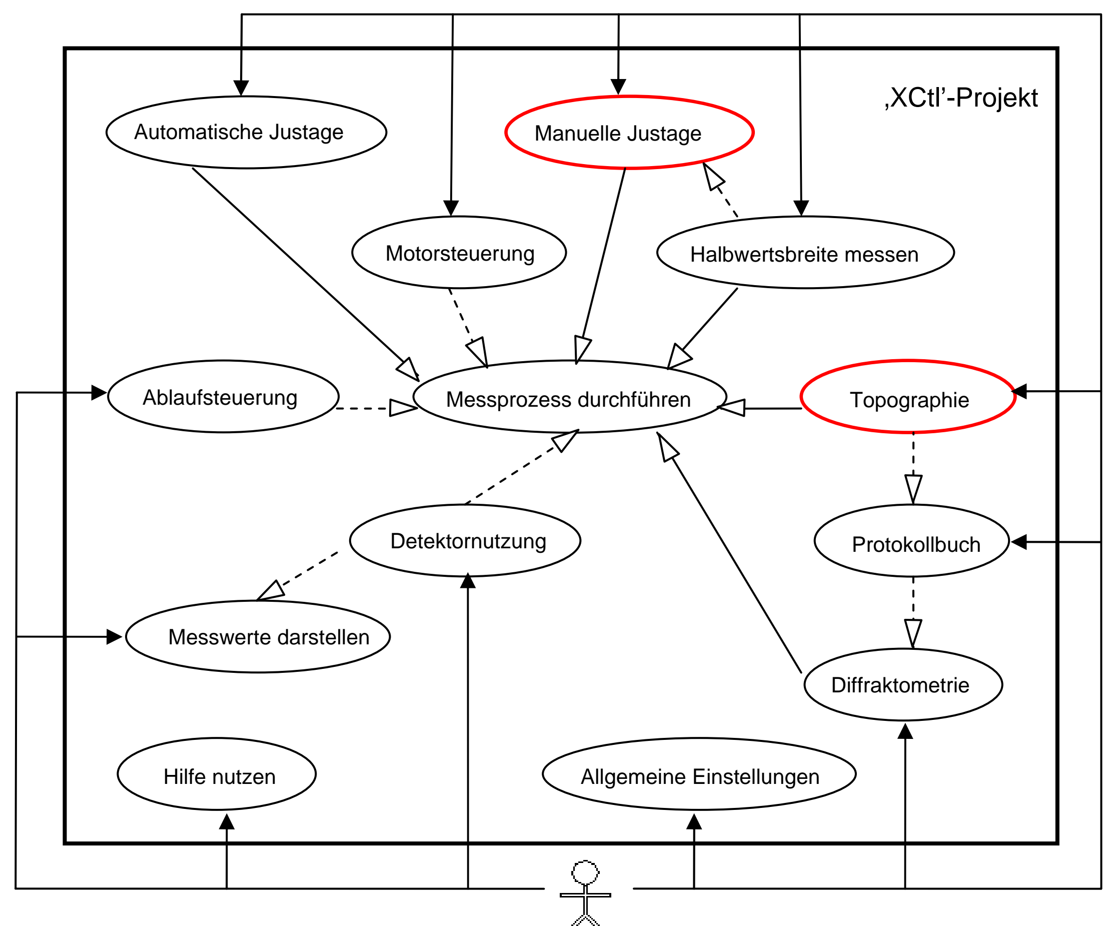


Abb. I.3 Use-Case-Diagramm des ‚XCtl'-Gesamtsystems,
rot = in dieser Arbeit bearbeitete Subsysteme (Quelle: nach **[M1]**)

Das Programm soll eine möglichst durchgängige, objektorientierte Schichtenarchitektur, eine umfassende Kommentierung des Quellcodes und eine vollständige Dokumentation aller Software-Entwicklungsphasen aufweisen. Dadurch kann es von den Physikern selbst weiterentwickelt bzw. gewartet werden. Die grafische Nutzeroberfläche soll zweisprachig, d.h. wahlweise deutsch oder englisch, zur Verfügung stehen. Ein wesentliches Ziel des Projektseminars ist bereits seit Juni 2003 realisiert. Dazu wurde das ‚XCtl'-Programmes in eine 32bit-Anwendung für das Betriebssystem Windows 2000 mit Microsoft Visual C++ als Entwicklungsumgebung portiert.

I.3.2 Subsystem ‚Manuelle Justage'

Der Anwendungsbereich **‚Manuelle Justage'** umfasst die Steuerung aller an einem Messplatz angeschlossenen Antriebe, um die Probe eines Halbleiterkristalls im dreidimensionalen Raum zu positionieren (siehe **[M15]**). Je nach Typ des Messplatzes (Topographie,
Diffraktometrie oder Reflektometrie) liegt die Anzahl der angeschlossenen Antriebe zwischen vier und sechs. **Abb. I.4** zeigt einen Topographie-Messplatz, wo die Halbleiterprobe – auf dem Probenteller (**PT**) und -halter (**PH**) positioniert – sehr genau in drei Freiheitsgraden bewegt werden kann. Dafür wird jeweils einer der Antriebe mit der Bezeichnung: azimutale Rotation (**AR**), Verkippung (**TL** – engl. Tilt) und Beugung Grob/Fein (**DC**, **DF** – engl. Diffraction coarse/fine) benutzt.

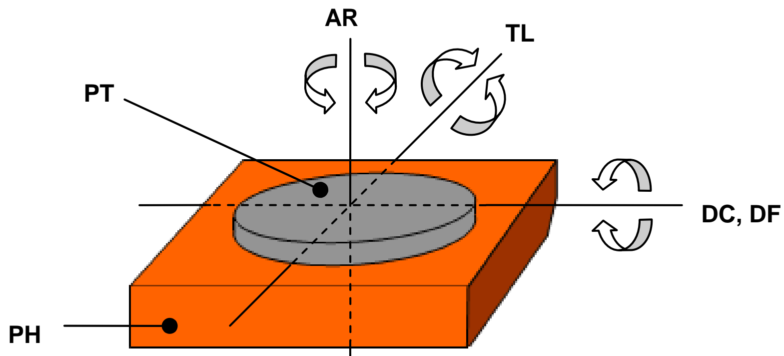


Abb. I.4 Freiheitsgrade der Probe bei der Topographie (Quelle: nach **[M1]**)

Jeder Antrieb kann separat und manuell gesteuert werden. Dies erfolgt durch Eingabe verschiedener Parameter wie einer Zielposition, einer Bewegungsrichtung oder der Bewegungsgeschwindigkeit. So wird die Positionierung der Probe, die Anpassung der Kollimator- an die Probenkrümmung und die Ausrichtung des Röntgendetektors (Diffraktometrie/ Reflektometrie) ermöglicht. Die ‚Manuelle Justage' dient meist der Vorbereitung eines physikalischen Experiments, z.B. der Topographie. Häufig wird sie aber auch nur dazu benutzt, sich einen Überblick über die aktuellen Antriebsparameter zu verschaffen oder die Halbwertsbreite zu messen.

Vollständige Funktionsbeschreibungen sind in **[M14]** (ursprüngliche ‚Manuelle Justage') bzw. **[M15]** (neue ‚Manuelle Justage') aufgeführt.

I.3.3 Subsystem ‚Topographie'

Das Ziel der **Topographie** ist die Darstellung der Kristallstruktur einer Messprobe von einem Halbleiterkristall (siehe **[M29]**). Dazu wird in einer Röntgenröhre ein Strahlenbündel erzeugt, das durch einen Kollimatorkristall aufgefächert und parallelisiert wird (siehe **Abb. I.1**). Dieses Röntgenlicht trifft in einem bestimmten Winkel nahezu monochromatisch auf die Messprobe und wird reflektiert. Zur Erfassung des Abbildes dient eine Fotoplatte, ein Film oder ein 2D-Detektor, die/der in den reflektierten Strahl gebracht wird. Das zu erfassende Kristall-Raumsegment hat eine Fläche von etwa 10 mm x 10 mm und eine Tiefe von max. 20 Mikrometern.

Die Belichtungszeit für eine Probe kann viele Stunden bis Tage betragen. Während dieser Zeit kann die Messanlage durch thermische Einflüsse geringfügig deformiert werden, was zur Abweichung von eingestellten Ausgangsbedingungen führt. Aus diesem Grund wird durch einen Kontrolldetektor ständig die von der Probe reflektierte Strahlung gemessen. Bei Abweichungen über einen vorgegebenen Intensitätsbereich hinaus muss die Stellung der Probe nachreguliert werden. Diese Korrektur erfolgt über den Motor Beugung Fein (siehe **Abb. I.4**, **DC**), automatisch.

Es gibt zwei Arten von Messungen: die Einfachbelichtung und die Mehrfachbelichtung. Im ersten Fall wird nur ein kleiner Ausschnitt der Probe einmal belichtet. Im zweiten Fall werden nacheinander mehrere Belichtungen durchgeführt. Nach Ablauf der Belichtungszeit wird die Probe hier auf ein neues Segment gestellt und erneut belichtet.

Um die Probe vor einem Topographie-Messvorgang in die richtige Ausgangsposition zu bringen, wird die ‚Manuelle Justage' benutzt. **Abb. I.5** zeigt das Photo eines realen Arbeitsplatzes.

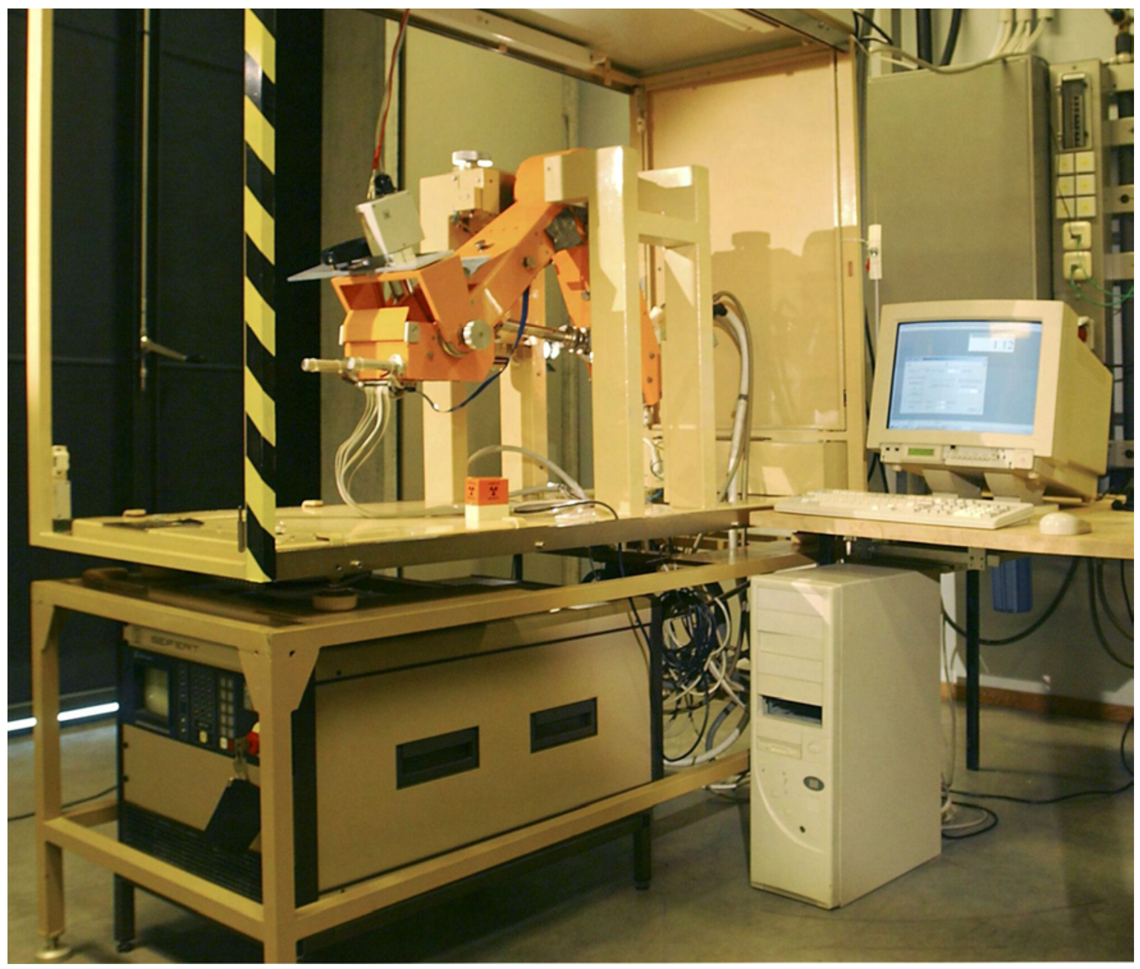


Abb. I.5 Arbeitsplatz RTK14 am Physikalischen Institut

I.4 Motivation

In den folgenden Abschnitten werden die Gründe und die Motivation für die design- und programmiertechnische Trennung der Funktionskomponente von der Nutzeroberfläche erläutert. Im nachfolgenden werden Funktionskomponenten oder Teile von diesen mit gekennzeichnet. Bei Nutzeroberflächen oder deren Programmcodefragmente erfolgt die Hervorhebung mittels . Es wird gezeigt, wann es sinnvoll ist oder nicht und wann unumgänglich, die Dekomposition bei einem kompletten Softwareprojekt oder einzelnen Subsystemen anzuwenden.

Der folgende Abschnitt zeigt, wie sie zur Bewältigung der Komplexität eines Softwareprojektes beiträgt. Schwerpunkt ist hier die Bedeutung für das Design der Software und die Bewertung anhand von Qualitätsmerkmalen. Die beiden nachfolgenden Abschnitte zeigen, nach Funktionskomponente und Nutzeroberfläche unterschieden, Vorteile für die Anpassungsfähigkeit und Wartbarkeit eines dekomponierten Softwareprojektes. Der vierte Abschnitt erläutert, warum es zweckmäßig sein kann, mehrere Programmiersprachen in einem Softwareprojekt zu verwenden und welche Rolle dabei die Dekomposition spielt. Im letzten Abschnitt wird die entscheidende Bedeutung bei der Realisierung einer Makrosteuerung dargestellt.

Die Bearbeitung aller Themen erfolgt nur unter dem Gesichtspunkt der objektorientierten Programmierung, weil sie ein essentieller Bestandteil der hier vorgestellten Dekomposition ist.

I.4.1 Bewältigung der Programmkomplexität

Nachdem die Kosten der Softwareentwicklung in der Mitte der 60er Jahre erstmals die Hardwarekosten überschritten haben (zur Softwarekrise siehe [2], S. 26ff), hat man wissenschaftliche Prinzipien zum Management der Komplexität eines Softwareprojektes entwickelt. Die Kernpunkte sind im Wesentlichen: Modularisierung, Abstraktion und Hierarchiebildung. Sie sollen nun im Hinblick auf die Dekomposition betrachtet und erläutert werden.

Eine der grundlegendsten Managementstrategien ist die Modularisierung (siehe [3], S. 571ff). Dieser, auch als Separation of Concerns bezeichnete Mechanismus, bedeutet die Zerlegung eines Programmes in einzelne Teilsysteme, die möglichst unabhängig voneinander bearbeitet werden können. Dazu müssen deutlich abgegrenzte Modul-Schnittstellen geschaffen werden. Im Hinblick auf die Dekomposition werden vorhandene Teil- oder Subsysteme in die beiden Komponenten Funktionalität und Nutzeroberfläche (engl.: **Graphical User Interface** = GUI) noch weiter aufgeteilt. Modularisierung kann sogar die Aufteilung eines Subsystems auf einzelne Dateien beinhalten, um auch den Programmcode voneinander zu separieren. Bei der Dekomposition entstehen dann mindestens zwei Dateien (Funktionskomponente und Nutzeroberfläche getrennt) für jedes Subsystem.

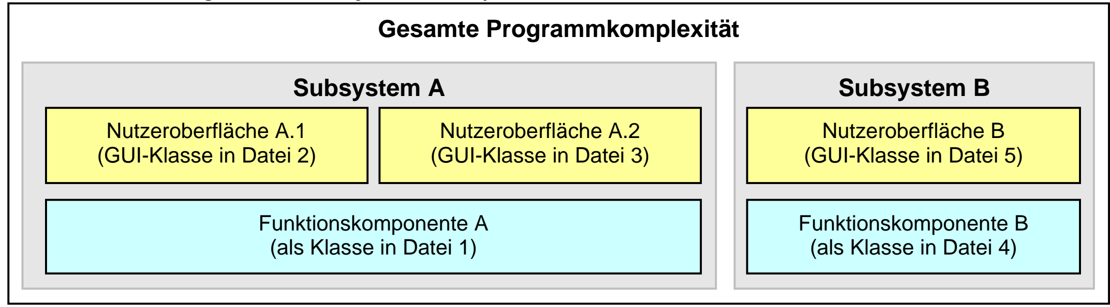


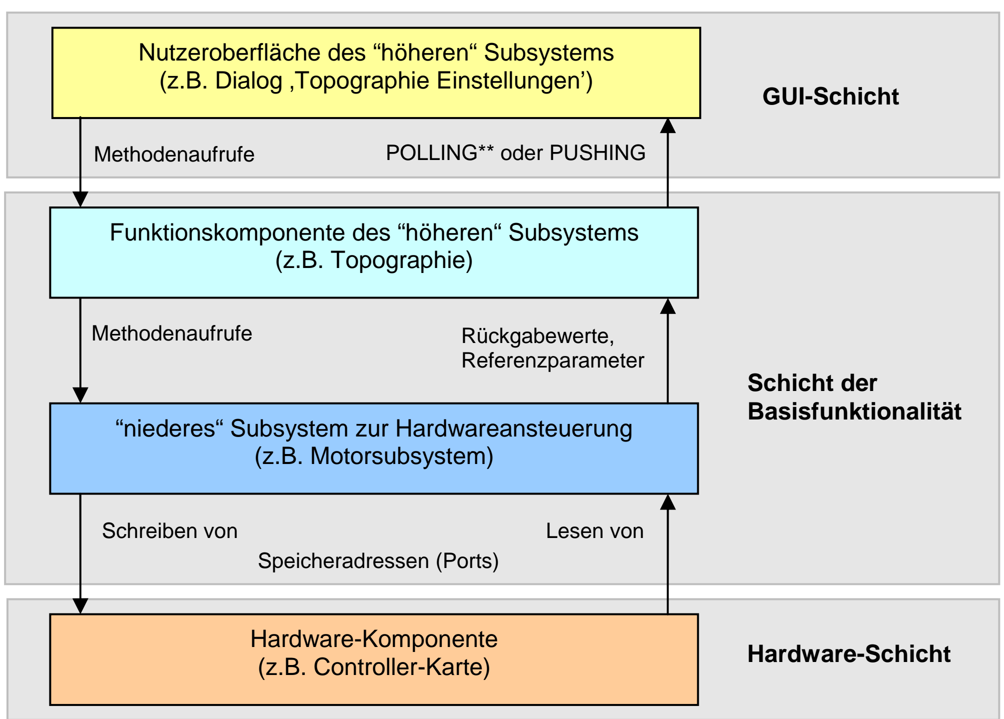
Abb. I.6 Bewältigung der Programmkomplexität durch Zerlegung in zwei Subsysteme, Modularisierung und Dekomposition in GUI- und Funktionskomponentenklasse

Die **objektorientierte Programmierung** = OOP unterstützt dies zusätzlich durch die Abstraktion des Klassen-Konstrukts. Hierbei ist die räumliche Trennung von Programmcode

mit dem Schutz von Daten und Funktionen vor Zugriffen durch andere Programmteile übersichtlich kombiniert. Erfolgt die Dekomposition von Funktionalität und Nutzeroberfläche in separate Klassen, lassen sich alle Vorteile der Objektorientierung ausnutzen.

So ermöglicht OOP die Integration von zwei Kernpunkten der Softwareentwicklung in einem Projektentwurf. Der erste Punkt ist das Information hiding (deutsch Datenkapselung). Dabei wird der Zugriff auf einzelne Komponenten, d.h. Attribute, Klassen oder Subsysteme, nur über eine wohl definierte Schnittstelle ermöglicht. Bei der Dekomposition von Funktionskomponente und Nutzeroberfläche wird Information hiding durch separate Klassen mit geschützten Attributen, d.h. meist privatisiert, und einer festen Anzahl an öffentlichen Methoden erreicht. Damit können Datenaustausch und -manipulation (zwischen der Nutzeroberfläche, den Funktionskomponenten und der Hardwareansteuerung) übersichtlich auf das Notwendigste beschränkt werden.

Der zweite Punkt ist das Environment hiding, d.h. die Minimierung von Umgebungswissen innerhalb einer Komponente. Treiberprogramme zur Hardwareansteuerung z.B. dürfen nichts über die sie benutzenden Anwendungen wissen. Bezogen auf die Dekomposition darf eine Funktionskomponente nichts über Anzahl und Zustände der assoziierten Nutzeroberflächen wissen. Überhaupt muss deren Schnittstelle (aus Sicht der Funktionskomponente) möglichst gering gehalten werden, um **Austausch, Flexibilität und Wartung der Nutzeroberfläche** einfach zu gestalten. Insgesamt lässt sich das Environment hiding jedoch schwierig umsetzen, da hierfür keine allgemeingültigen Designmuster existieren (nachzulesen bei **[1]**). Lösungsansätze werden ab Kapitel **II.3** ausführlich diskutiert.



"höheres" Subsystem – besteht aus Nutzeroberfläche und Funktionskomponente
"niederes" Subsystem – Basisfunktionalität (unabhängig von der Nutzeroberfläche)
→ zeigt den Datenfluss
** Ergebnis wird über einen Methodenaufruf zurückgegeben (siehe)

Abb. I.7 Schichten-Architektur am Beispiel der getrennten ‚Topographie'

Unter Anwendung dieser Richtlinien wird durch Dekomposition, neben der Modulzerlegung, auch die **hierarchische Architektur** in Softwareprojekte eingeführt oder weiter verfeinert. Es

entsteht eine Schichtenstruktur, in der jede Komponente eindeutig einer Schicht zugeordnet werden kann. Die Dekomposition wurde exemplarisch an drei Subsystemen durchgeführt. Dabei wurde das bereits im ‚XCtl'-Programm vorhandenen Schichtenmodell um die Schicht der Funktionskomponente erweitert. Diese Struktur zeigt **Abb. I.7**.

Aufgrund der Erfahrungen am Fallbeispiel ‚XCtl' erfolgt nun eine verallgemeinerte Auswertung und Bewertung anhand der inneren und äußeren Qualitätsmerkmale.

Durch die konsequente Anwendung der oben geschilderten Richtlinien zur Dekomposition, wird der Programmcode strukturierter. Kleinere, übersichtliche Fragmente machen den Code zudem erfass- und lesbarer – die Übersichtlichkeit steigt (siehe **II.6** und **III.3**). Das führt im Allgemeinen auch zu einer Fehlerminimierung und sollte Portierungs- und Wartungsarbeiten erleichtern. Im Fallbeispiel ‚XCtl' zeigten sich deutliche Zeitvorteile während der Produktpflege, d.h. Suche und Korrektur von Fehlern. Gleiches gilt auch für die Erweiterbarkeit, also Anpassung der Programmfunktionalität an Spezifikationsänderungen. Fehlersuche oder Änderungen konnten bereits im Vorhinein, entweder auf Funktionskomponente oder Nutzeroberfläche, eingegrenzt werden.

Durch die objektorientierten Mechanismen, d.h. hoher Zugriffsschutz und einfaches Schnittstellendesign der Funktionskomponente, können diese und die Nutzeroberfläche unabhängig voneinander und nahezu zeitgleich[1] getestet werden, siehe **II.5**.

Die Geschwindigkeit der Projektentwicklung sollte sich durch die Möglichkeiten der Arbeitsteilung generell erhöhen, weil auch die Implementierung von Funktionskomponente und Nutzeroberfläche grundsätzlich parallel möglich ist, siehe **II.4**. Unter dem gestiegenen Umfang an zu testenden Programmfunktionen, vorrangig durch die zusätzliche Schnittstelle der Funktionskomponente, leidet vorerst die Entwicklungsgeschwindigkeit. Dafür werden aber **Verifikation** (fehlerfreies Arbeiten, siehe **[3]**, S. 101) und **Validation** (erfüllt die geforderte Spezifikation, siehe **[3]**, S. 101) in der Testphase erleichtert. Durch die Dekomposition werden mehr Fehler bereits während der Entwicklung der Funktionskomponente bzw. Nutzeroberfläche bemerkt und korrigiert. Dadurch treten während der Modulintegration in den Anwendungskontext weniger Fehler auf, die analysiert und behoben werden müssen – das führt letztendlich zur Fehlerminimierung.

Auch während und nach der Anpassung der Software an andere Betriebssysteme (Portierung), sollte die Fehlerwahrscheinlichkeit deutlich sinken, weil Funktionskomponente und Nutzeroberfläche, jeweils durch Spezialisten, getrennt bearbeitet und getestet werden können. Die Kapitel **I.4.2** und **I.4.3** zeigen auch die verbesserte Wiederverwendbarkeit, also die Eigenschaft Elemente eines Programmes partiell oder vollständig für andere Zwecke einzusetzen; z.B. "Recycling" der Funktionskomponente beim Austausch der Nutzeroberfläche.

Neben den Vorteilen der Dekomposition, muss besonders bei paralleler Bearbeitung bedacht werden, dass ein erhöhter Kommunikationsaufwand zwischen den Entwicklern – in der Design-, Implementierungs- und Testphase sowie bei der Dokumentenerstellung – entsteht. Zu beachten ist auch das umfangreichere Design, weil ohne die Dekomposition keine Funktionskomponente entstehen würde. Darunter leidet auch die Effizienz, weil die zusätzliche Kommunikation der Nutzeroberfläche mit der Funktionskomponente die Ausführungsgeschwindigkeit erheblich vermindern kann. Dies wird insbesondere durch die Verwendung des im Kapitel **II.2.5** vorgestellte Verfahrens (`Can`-Methoden der Funktionskomponente) hervorgerufen.

Für den Anwender erhöht dies jedoch deutlich die Zuverlässigkeit, die sich in Verbindung mit separatem Testen als insgesamt robusteres Software-System ausdrückt. Damit sollte die Software auch unter "außergewöhnlichen" Bedingungen sinnvoll funktionieren, weil unzu-

[1] Test der Nutzeroberfläche ist erst nach Abschluss der Funktionskomponente möglich, ausgenommen Funktionsprototypen

lässige Eingaben nicht durch die Funktionskomponente verarbeitet werden und zu Fehler führen können, falls die Nutzeroberfläche diese Situationen nicht bereits ordnungsgemäß abdeckt.

I.4.2 Austausch, Flexibilität und Wartung der Nutzeroberfläche

Die Möglichkeiten der Oberflächengestaltung sind heutzutage sehr vielfältig. Der Entwurf ist im Wesentlichen vom Betriebssystem und der verwendeten Programmiersprache abhängig. Die meisten objektorientierten Sprachen bieten hierfür individuelle GUI-Bibliotheken an. Java z.B. hat mit seinem **Application Programming Interface** = API (deutsch: allgemeine Programmierschnittstelle) den großen Vorteil der Plattformunabhängigkeit. Das trifft für viele Sprachen, insbesondere für das von uns verwendete C++, leider nicht zu. Zudem sind fast alle Sprachen proprietär, d.h. herstellerabhängig.

Soll ein Wechsel der Entwicklungsumgebung oder Zielplattform stattfinden, ist besonders bei C++ als Programmiersprache die Nutzeroberfläche oft neu zu implementieren oder grundlegend zu überarbeiten. Dann ist bei Anwendungen, die derzeit auf mehreren Betriebssystemen arbeiten sollen, für jede Plattform eine eigene Version zu erstellen. Dies ist mit hohem programmiertechnischen Aufwand verbunden. Erfolgt zudem keine Trennung in Funktionskomponente und Nutzeroberfläche, wie in der Praxis oft üblich, erhöht sich dieser Aufwand erheblich. Erst die hier vorgestellte Dekomposition macht den Portierungs- bzw. Entwicklungsprozess ökonomischer und fehlerresistenter. Hier müssen nur die Klassen der Nutzeroberfläche ersetzt werden, welche die bereits vorhandenen Schnittstellen der Funktionskomponenten neu einbinden. Daher ist die Dekomposition auch beim Oberflächen-Prototyping sehr empfehlenswert, weil die Funktionskomponente dort lange Zeit unverändert bleibt und “nur“ verschiedene, provisorische Nutzeroberflächen, ohne die echte Funktionalität, erstellt und diskutiert werden. Die Programmfunktionen (Funktionskomponente) müssen nur selten angepasst werden, z.B. bei einem Plattformwechsel von 16 nach 32-Bit.

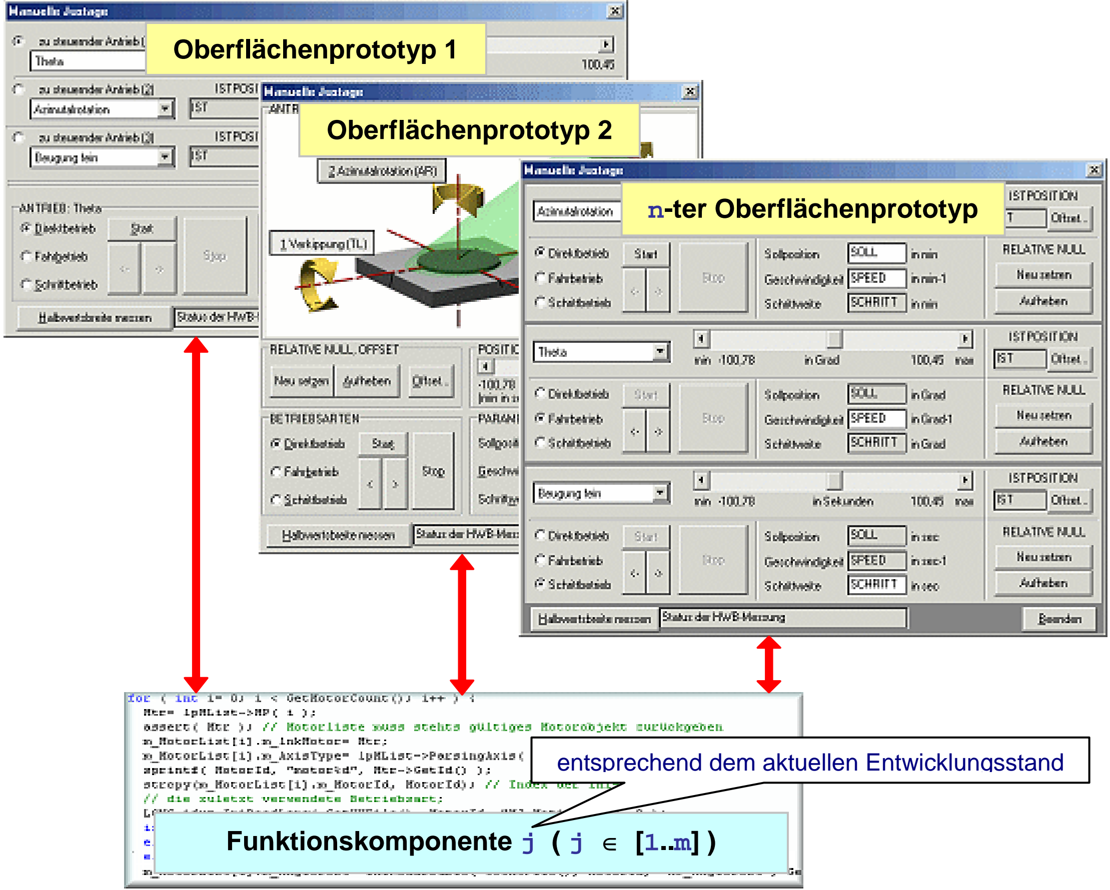


Abb. I.8 Evolution der Nutzeroberfläche – Oberflächen-Prototyping

Ein weiterer Grund für die Dekomposition sind multilinguale Anwendungen. Dabei stehen, mehrere Nutzeroberflächen in verschiedenen Sprachen zur Steuerung der gleichen Programmfunktion zur Verfügung. Nur die Fenster unterscheiden sich in der Beschriftung der Steuerelemente. Aufgrund der sprachspezifischen Eigenheiten, z.B. sehr große Wortlänge, können sie auch in der Gestaltung variieren. Die programmiersprachliche Einbindung mehrerer Nutzeroberflächen ist erforderlich, wenn die Oberflächenelemente nicht dynamisch eingebunden werden können, was einige GUI-Bibliotheken heute bereits unterstützen (z.B. das Ressourcen-Konstrukt in C++). Aber auch dann müssen Meldungsausgaben oder dynamische Beschriftungen, oft verstreut im gesamten Quelltext, gesucht und übersetzt werden. Die Übertragung der Nutzeroberfläche in andere Sprachen ist einfacher, wenn alle GUI-spezifischen Teile zusammengefasst sind. Im Falle der Dekomposition sind dies eigene Klassen. Die Funktionskomponente darf keine Ausgabe erzeugen und kann somit unverändert bleiben.

Ähnlich verhält es sich, wenn Oberflächen an mehrere Benutzergruppen angepasst werden müssen. Den verschiedenen Nutzern müssen oder dürfen, je nach zugewiesenen Arbeitsaufgaben und Rechten, nur ausgewählte Programmfunktionen zur Verfügung gestellt werden. Um nicht für jede Nutzergruppe eine eigene Oberfläche mit integrierter Funktionalität bereitzustellen, sollte nur eine Funktionskomponente entwickelt werden, die alle Produktfunktionen enthält. Die Realisierung der Zugriffsrechte erfolgt ausschließlich über neue Nutzeroberflächen, die jeweils nur einen Teil der Schnittstelle der einmal erstellten Funktionskomponente benutzen. So wird Redundanz vermieden, was Fehler minimiert und die Entwicklungszeit verkürzt.

Gleichzeitig erleichtert die Dekomposition die individualisierbare Oberflächengestaltung, bei der die Nutzeroberfläche leicht an die Bedürfnisse der Anwender angepasst werden kann. Dies ist z.B. bei Änderung der Reihenfolge und Verfügbarkeit von Funktionen in Symbolleisten (siehe **Abb. I.9**) der Fall.

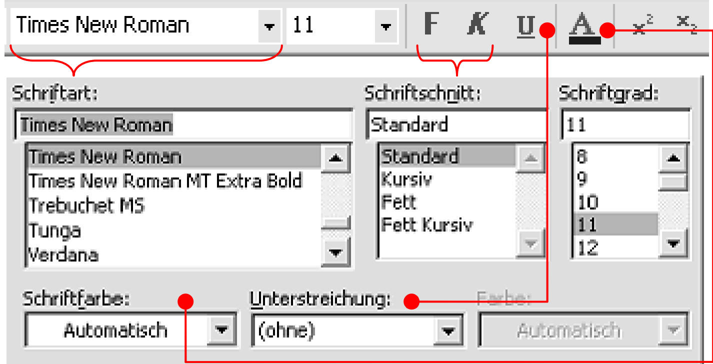


Abb. I.9 Beispiel der Schriftformatierung über eine individualisierbare Symbolleiste (oben) im Vergleich zum Ausschnitt des Dialogfensters ‚Zeichen' (unten) in Microsoft Word

Bei einer ausgelagerten Funktionskomponente werden die grundlegenden Programmfunktionen nicht tangiert, wenn die Nutzeroberfläche individualisier gestaltet werden soll. GUI-Klassen beinhalten dann die Sicherung und Wiederherstellung der gewünschten Anordnung sowie die Verknüpfung der Steuerelemente mit den entsprechenden Methoden der Funktionskomponente.

Wenn die gewünschte Funktionalität über mehrere Steuerelementtypen (siehe **Abb. I.9**, Auswahl der Schriftart über ein Kombinations- oder Gruppenfeld) realisiert werden kann oder in mehreren Fenstern angeboten werden soll, muss die zugehörige Funktionalität in eine Funktionskomponente ausgelagert werden. Wenn diese Dekomposition nicht erfolgen würde, müsste der Programmcode mehrfach vorhanden sein. Das birgt, neben

dem erhöhten Arbeitsaufwand und Fehlerrisiko, auch Gefahren bei der Wartung, weil Inkonsistenzen entstehen könnten.

I.4.3 Austausch und Wartung der Funktionskomponente

Neben dem Wechsel der Nutzeroberfläche besteht auch die Möglichkeit, die Funktionskomponente auszutauschen. Ein Szenario wäre die Verwendung von simulierten Reaktionen in einer Anwendung. Durch die Modularisierung könnten einzelne Programmteile mit einer Pseudo-Funktionalität hinterlegt werden, welche die eigentlichen Funktionen nur simulieren. Beispielsweise wird anstelle des Ergebnisses einer komplexen Berechung nur ein konstanter Wert zurückgegeben. So kann das Zusammenwirken mit anderen Programmteilen oder das Gesamtverhalten vorgeführt werden, obwohl die eigentliche Implementierung unvollständig ist.

Diese Vorgehensweise wird besonders beim **Prototypen**-Modell (siehe [3], S. 114ff) eingesetzt. Beim weiteren Fortschreiten des Entwicklungsprozesses wird diese Simulation Schritt für Schritt durch die eigentliche Funktionalität ersetzt. So erfolgt ein allmählicher Übergang vom Prototyp zum realen Produkt, ohne dass die Entwicklung der Nutzeroberfläche davon betroffen ist (siehe **Abb. I.10**).

Die Möglichkeit zum Austausch der Funktionskomponente kann besonders beim Testen der Verknüpfungen zwischen Nutzeroberfläche und Funktionskomponente eingesetzt werden. Hier können die eigentlich vollständig getesteten Funktionskomponenten durch Platzhalter (engl.: stubs) ersetzt werden, um mögliche Verknüpfungsfehler in der Nutzeroberfläche zu identifizieren (siehe **II.5.2**).

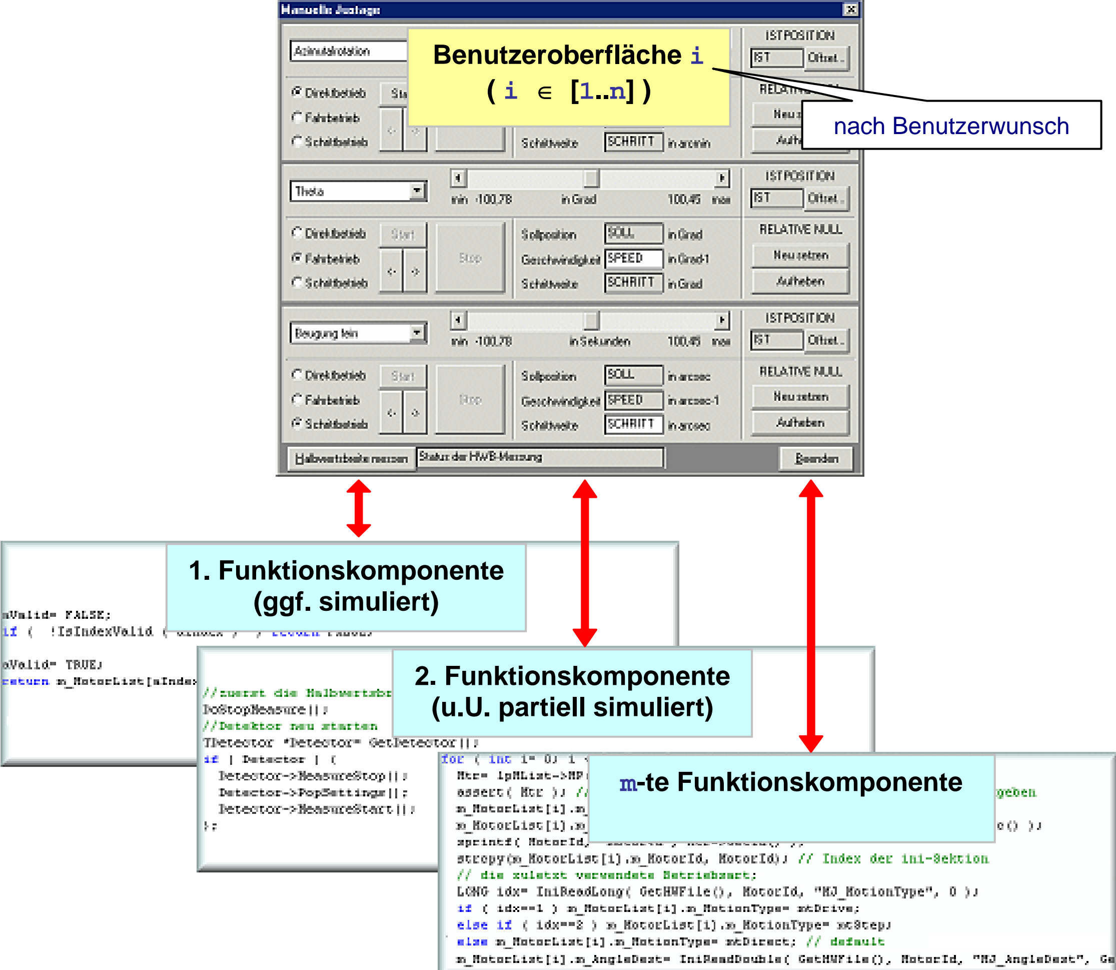


Abb. I.10 Übergang von der simulierten Funktionskomponente, über die Wartung, zum Endprodukt

Ein wichtiger Vorteil ist auch die unabhängige Wartbarkeit der Funktionskomponente. Bei konsequenter Dekomposition sind alle Algorithmen und Daten in der Funktionskomponente zusammengefasst. Es besteht so keine Gefahr, dass durch Bearbeitung des Programmcodes der Nutzeroberfläche ungewollt Produktfunktionen geändert werden. Andererseits ziehen Änderungen an der Schnittstelle der Funktionskomponente fast immer Anpassungen an der Implementierung der Nutzeroberfläche nach sich.

In Client-Server orientierten Anwendungen oder Subsystemen werden Teile der Funktionalität zentral durch einen Server bereitgestellt oder verwaltet. Die zur Client-Server-Kommunikation notwenigen Arbeitsabläufe[2] sind unabhängig von der Nutzeroberfläche und können durch die Dekomposition in die Funktionskomponente ausgelagert werden, obwohl die steigende Leistungsfähigkeit heutiger Rechnersysteme diesen Anwendungsfall zunehmend auf zentrale Daten- und Ressourcenverwaltung beschränkt.

I.4.4 Kombinierbarkeit von Programmiersprachen

Durch die Dekomposition ergibt sich prinzipiell die Möglichkeit Funktionalität und Nutzeroberfläche in verschiedenen Programmiersprachen zu implementieren.

Durch die Dekomposition könnte man die Funktionskomponente in einer für den Anwendungszweck besser geeigneten Programmiersprache entwickeln und eine Dynamic Link Library = DLL erstellen, die von der Nutzeroberfläche eingebunden wird. Linux und Microsoft Windows bieten damit ein Konstrukt für kompilierte Modulbibliotheken mit einer definierten Schnittstelle, die sich zur Laufzeit durch andere Programmiersprachen mit DLL-Schnittstelle einbinden lassen. Dort befindet sich dann nur ein Verweis auf die entsprechende Methode einer DLL. Heute bieten fast alle Programmiersprachen Möglichkeiten DLLs zu im- und exportieren.

In C/C++ sind bspw. das Pointerkonstrukt und die schnelle Ausführungsgeschwindigkeit für Port-basierten low-level Hardwarezugriff in 16Bit-Windows Anwendungen nutzbar. Eine java-basierte Nutzeroberfläche ist dafür frei portabel. Der Umstieg auf ein anderes Betriebssystem wäre dann durch Anpassung und Austausch der Funktions-DLL problemlos möglich. Die Oberfläche bliebe unverändert.

ActiveX (vor 1996 Object Linking and Embedding = OLE) ist Teil von Microsofts Component Object Model = COM, das die DLL-Philosophie für die objektorientierte Programmierung erweitert. ActiveX ermöglicht die Zusammenarbeit von verschiedenartigen Anwendungen und Komponenten auf Basis einer einheitlichen Schnittstelle.

ActiveX-Steuerelemente und -dokumente können umfassende Funktionen anbieten und in die eigene Nutzeroberfläche integriert werden. Will man solche Elemente einbinden oder selbst Fenster anbieten, muss die Programmiersprache der Nutzeroberfläche den ActiveX-Standard unterstützen.

ActiveX-Code-Komponenten erlauben das Starten und die Interprozesskommunikation mit anderen Programmen im ActiveX-Standard aus der eigenen Anwendung heraus. Wenn die Anwendung durch andere Programme kontrollierbar gestaltet werden soll, ist die Dekomposition der Funktionskomponente von der Nutzeroberfläche unerlässlich, weil die Steuerung unabhängig von der Nutzeroberfläche erfolgen muss. Neuere Versionen des zu kontrollierenden Programmes wären mit Änderung der Nutzeroberfläche sonst nicht mehr abwärtskompatibel steuerbar.

---

[2] z.B. durch Benutzung von Common Object Request Broker Architecture = CORBA oder Microsofts Distributed Component Object Modell = DCOM

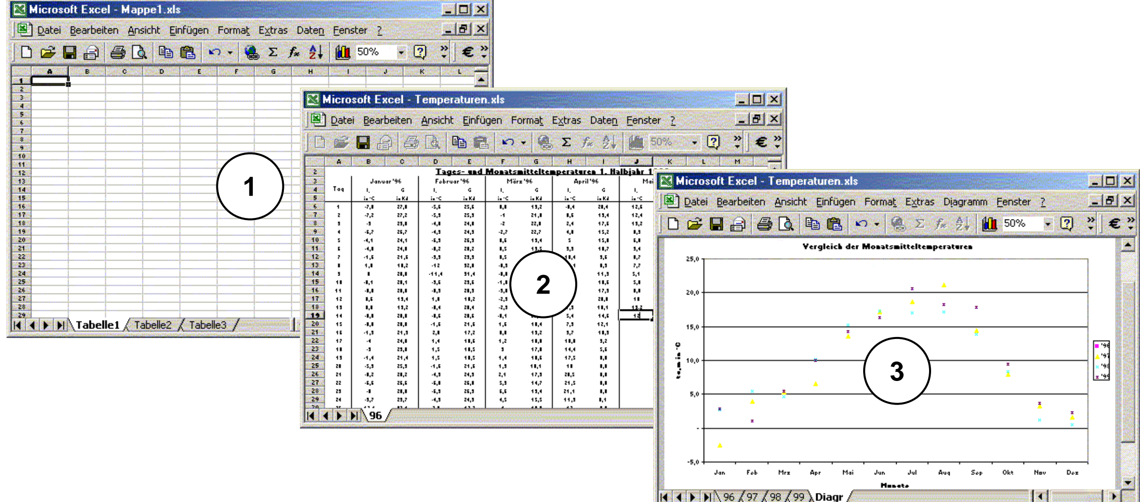


Abb. I.11 OLE-Automation mit Microsoft Excel aus einem anderen Programm heraus:
1 – starten; 2 – Datenerfassung; 3 – Diagrammerstellung

Über ActiveX kann z.B. die Funktionalität des kompletten Microsoft Office Pakets in eigenen Anwendungen verwendet werden. Einen guten Einstieg ins Thema COM bietet [11].

I.4.5 Makrosteuerung

Ähnlich ActiveX (siehe I.4.4) gestattet die Makrosteuerung die Automatisierung und Zusammenfassung von Arbeitsabläufen. Dazu werden Makros vom Programm interpretiert und abgearbeitet. Diese entsprechen Anweisungsfolgen die ausgewählte Produktfunktionen realisieren. Komplexe Arbeitsabläufe sind so problemlos reproduzierbar. Insbesondere wiederkehrende Aktionen, die im eigentlichen Programm nur über viele Menüaufrufe, Tastendrücke oder Mausklicks realisiert werden können. Heutzutage beinhalten viele Programme bereits eine solche Makroverarbeitung, um die Arbeit mit dem immer umfangreicheren Funktionsangebot effizient zu ermöglichen. Zudem besteht manchmal die Möglichkeit einen Makrorekorder zu implementieren, der die Aktionen des Benutzers aufzeichnet und selbständig in ein Makro umwandelt.

Um Makrosteuerung und -rekorder in die eigene Anwendung zu integrieren, empfiehlt sich jeweils die Erstellung eines eigenen Subsystems. Makros sollten ausschließlich auf der Ebene der Funktionskomponente arbeiten. Jede einzelne Anweisung in einem Makro kann so auf den Aufruf von Methoden einer Funktionskomponente beschränkt werden. Es ist wichtig die Makros von der Nutzeroberfläche zu trennen! Sonst würden nur die Oberflächenfenster ferngesteuert. Dadurch wäre die Programmierung umfangreicher[3] und während der Abarbeitung des Makros würden die benötigten Fenster angezeigt, gesteuert und geschlossen werden. Bei Änderungen an der Nutzeroberfläche wären "alte" Makros u.U. nicht mehr gültig. Besonders problematisch ist dies bei individualisierbaren Nutzeroberflächen.

Nur bei grundlegenden Änderungen an der Funktionskomponente sind u.U. Anpassungen an Makrosteuerung und -rekorder erforderlich, um Abwärtskompatibilität für alte Makros zu gewährleisten. Bei Integration oder späteren Änderungen der Makrosteuerung oder des -rekorders ist die Nutzeroberfläche davon nicht betroffen.

[3] Anstatt einem Methodenaufruf zum Ändern der Schriftgröße, müssten Menüaufruf und Auswahl der Größe im Dialogfenster erfolgen. Anschließend müsste das Fenster geschlossen werden.

Kapitel II

DEKOMPOSITION BEIM FORWARD ENGINEERING

In diesem Kapitel wird die Dekomposition von Funktionskomponente und Nutzeroberfläche beim Forward Engineering beschrieben. Dies wird exemplarisch am Neuentwurf des Subsystems ‚Manuelle Justage' vorgestellt. Neben dem verallgemeinerten Grundgerüst werden so gewonnene Erkenntnisse und praktische Lösungsstrategien für ausgewählte Probleme dargestellt. Schwerpunkt bildet die Designphase mit der Erläuterung von Design-Mustern für den Datenaustausch zwischen Nutzeroberfläche und Funktionskomponente.

Im Folgenden wird das vorhandene Subsystem, das bereits Teil des Originalprojektes von 1998 war, als **ursprüngliche ‚Manuelle Justage'** bezeichnet. Das im Forward Engineering erstellte Subsystem erhält die Bezeichnung **neue ‚Manuelle Justage'**. Für eine kurze Einführung in den Gegenstandsbereich dieses Systems, siehe **I.3.2**.

II.1 Analysephase

Am Beginn des Forward Engineering steht die Zusammenfassung der zu realisierenden **Basisanforderungen**. Das sind nur die grundlegendsten Eigenschaften, die das Produkt aus Sicht des Auftraggebers erfüllen soll. Sie werden ohne nähere Details im **Lastenheft** gesammelt und bilden die Grundlage für die Durchführbarkeitsstudie und den Projektplan (ausführlich in [2], S. 62ff).

Normalerweise entsteht das Lastenheft aus Gesprächen zwischen Auftraggeber und den Vertretern, die für die Softwareentwicklung zuständig sind. Existieren bereits ein Produkt oder ein Subsystem zur Lösung derselben Problemstellung (**Referenzprodukt**), so können die dort gewonnen Erkenntnisse, Dokumentationen und Änderungswünsche für die Neuentwicklung im Forward Engineering sehr hilfreich sein. Das gilt auch für gescheiterte Projekte, weil die Gründe dafür analysiert und im Neuentwurf berücksichtigt werden können. Um die anwender- und entwicklerseitigen Erfahrungen zu extrahieren, wird auf diesem Referenzprodukt eine Ist-Analyse durchgeführt. Die Einarbeitung in den Gegenstandsbereich und die Erstellung eines neuen Pflichtenheftes wird erleichtert und die entwicklerseitige Grundlage für Vertragsverhandlungen verbessert sich.

Diese Analyse muss nicht zwingend zu Beginn des Forward Engineering abgeschlossen sein. Beim Neuentwurf eines umfangreichen Projektes können Erkenntnisse während der Entwicklung wieder verloren gehen, wenn die Ist-Analyse vollständig zu Beginn erfolgte. Die vorhandenen Teile des Referenzproduktes sollten deshalb in den entsprechenden Phasen des Forward Engineering unter den jeweiligen Gesichtspunkten betrachtet werden.

Im günstigsten Fall stehen neben dem Programmcode alle phasenspezifischen Dokumente (Verhaltensspezifikation bzw. Pflichtenheft, Design-, Testdokumentation, Fehlerliste) des Vorgängerproduktes zur Verfügung. Dazu kann der Referenzentwurf während der neuen Designphase mit **Computer Aided Software Engineering** = CASE- Werkzeugen extrahiert und analysiert werden. Eine Untersuchung des bestehenden Programmcodes sollte hingegen erst bei der Neuimplementierung erfolgen.

Anwendungsfall: ‚Manuelle Justage'

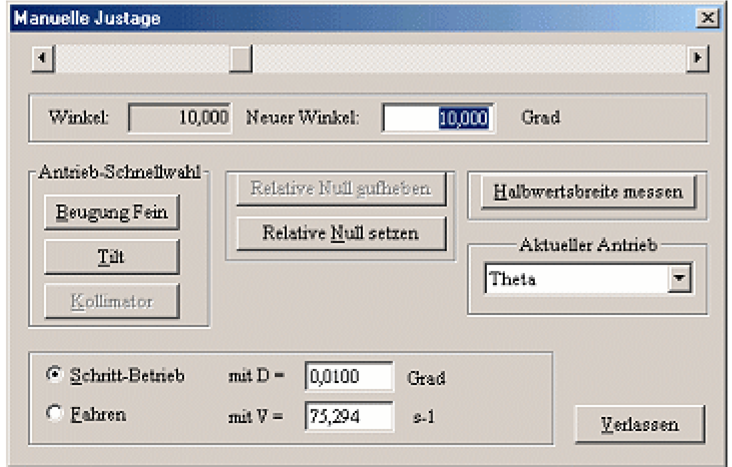
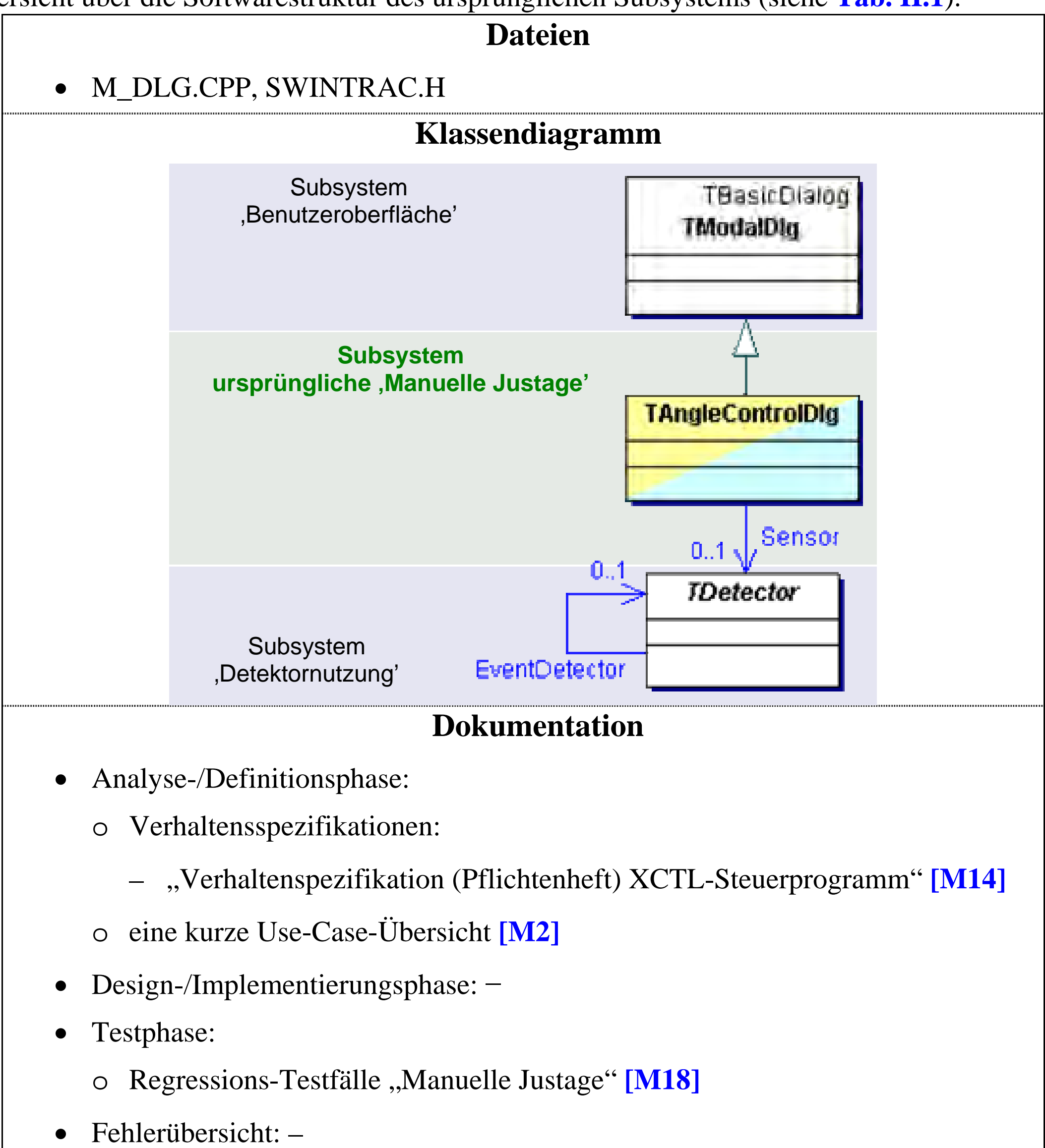


Abb. II.1 Dialogressource zur ursprünglichen ‚Manuelle Justage'

Der erste Kontakt mit der ‚Manuellen Justage' erfolgte bereits im Vorbereitungsseminar ‚Projekt Softwaresanierung', als die Verhaltensspezifikation [M13] einem Review unterzogen wurde. Dieses Dokument besteht aus der funktionalen Beschreibung der verwendeten Nutzeroberfläche (**Abb. II.1**) und beschreibt die Gültigkeitsbereiche für die Eingabefelder und

den Inhalt des Kombinationsfeldes. Die konkreten Werte stammen aus der Konfigurationsdatei eines Topographiearbeitsplatzes und haben damit nur exemplarischen Charakter. Zudem sind Änderungswünsche und offene Fragen aufgeführt.

Zu Beginn der Diplomarbeit erfolgten eine Bestandsaufnahme der Dokumente und eine Übersicht über die Softwarestruktur des ursprünglichen Subsystems (siehe **Tab. II.1**).



Dateien

- M_DLG.CPP, SWINTRAC.H

Klassendiagramm

Dokumentation

- Analyse-/Definitionsphase:
  - Verhaltensspezifikationen:
    - „Verhaltenspezifikation (Pflichtenheft) XCTL-Steuerprogramm" **[M14]**
  - eine kurze Use-Case-Übersicht **[M2]**
- Design-/Implementierungsphase: –
- Testphase:
  - Regressions-Testfälle „Manuelle Justage" **[M18]**
- Fehlerübersicht: –

Tab. II.1 Übersicht zur ursprünglichen ‚Manuellen Justage'

Im Hinblick auf eine prognostizierte Entwicklungszeit von fast einem Jahr wurde keine vollständige Ist-Analyse durchgeführt. Die vorhandenen Dokumente und der Programmcode wurden immer wieder im Verlauf des Softwareentwicklungsprozesses ausgewertet. An den entsprechenden Stellen von **Kapitel II** wird darauf verwiesen. Zur Einarbeitung in den Gegenstandsbereich war bereits die aus dem o.g. Review resultierende, überarbeitete Version der Verhaltensspezifikation **[M14]** verfügbar.

Bei der intensiven Beschäftigung mit dem existierenden Dialogfenster konnten die Basisanforderungen (sowohl Funktionen und Daten) vollständig und ohne Kontakt zum Auftraggeber abgeleitet werden. Da Entwicklungs- und Einsatzumgebung bereits vorgegeben und die Basisanforderungen überschaubar waren, wurde auf die Erstellung von Lastenheft, Aufwandsschätzung und Projektplan für die neue ‚Manuelle Justage' verzichtet.

Die Basisanforderungen wurden vielmehr stichpunktartig in die bei **II.2.1** vorgestellte Gliederungsstruktur des Pflichtenheftes eingeordnet. Die zu realisierenden Funktionen wurden im Kapitel „Funktionale Beschreibung“ und die anzuzeigenden und zu verwaltenden Datenelemente unter „Daten“ aufgeführt. Weil das Referenzprodukt des zu realisierenden Subsystems bereits existierte, konnte die Durchführbarkeitsstudie entfallen.

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
`TAngleControlDlg`	457	11	8	1	6	55	0	45	75	4	1	25

Tab. II.2 Ausgewählte Metriken für die Klassen der ursprünglichen ‚Manuellen Justage’

Die in **Tab. II.2** verwendeten Metriken wurden dem CASE-Werkzeug Togethersoft Together 5.5 entnommen. Sie wurden ausgewählt, um Programmcodestruktur, -zugriffsschutz, Design und Umfang der in dieser Arbeit vorgestellten Subsysteme effektiv zu bewerten. Dabei werden der Zustand vor und nach der Dekomposition und die verschiedenen Designmuster verglichen. Alle nachfolgenden Metrikübersichten enthalten die gleichen Elemente.

- **LOC** (**L**ines **Of** **C**ode) ist die Anzahl der Codezeilen ohne Leer- und Kommentarzeilen.
- **NOA** (**N**umber **Of** **A**ttributes) entspricht der Anzahl an Attributen einer Klasse.
- **NOO** (**N**umber **Of** **O**perations) ist die Anzahl der Methoden und Operatoren einer Klasse ohne Kon- und Destruktor(en).
- **NOCON** (**N**umber **Of** **Con**structors) umfasst alle Konstruktoren einer Klasse. Destruktoren werden nicht erfasst, weil es pro Klasse immer genau einen gibt.
- **NOOM** (**N**umber **Of** **O**verridden **M**ethods) ist die Anzahl aller überschriebenen Methoden und Operatoren der eingeerbten Basisklasse(n).
- **PPrivM** (**P**ercentage of **Priv**ate **M**embers) zeigt den prozentualen Anteil an `private` Attributen/ Methoden/ Operanden/ Kon-/ Destruktoren.
- **PProtM** (**P**ercentage of **Prot**ected **M**embers) ist der prozentuale Anteil an `protected` Attributen/ Methoden/ Operatoren/ Kon-/ Destruktoren.
- **PPubM** (**P**ercentage of **Pub**lic **M**embers) ist der prozentuale Anteil an `public` Attributen/ Methoden/ Operatoren/ Kon-/ Destruktoren. PPrivM + PProtM + PPubM ≡ 100 %
- **AC** (**A**ttribute **C**omplexity) ist die Summe aus der Gewichtung aller Attribute. Dazu ist jedem Attribut ein Wert zugeordnet. Beginnend bei eins für Standardtypen erhöht sich dieser mit steigendem Abstraktionsgrad, z.B. fünf für `void` und neun für nutzerdefinierte Typen.
- **MNOP** (**M**aximum **N**umber **Of** **P**arameters) errechnet sich aus der Anzahl der Parameter in der längsten Parameterliste aller Methoden und Operatoren.
- **NOC** (**N**umber **Of** **C**lasses) zählt die Anzahl der enthaltenen Unterklassen und Strukturen einer Klasse. Sind keine enthalten, ist der Wert eins. Für jedes weitere Element wird die Anzahl um eins inkrementiert. Alle anderen Metriken beziehen sich nur auf die direkt enthaltenen Attribute/ Methoden/ Operatoren/ Kon- und Destruktoren. Elemente aus den enthaltenen Klassen/ Strukturen werden nicht beachtet!
- **TCR** (**T**rue **C**omment **R**atio) stellt die Leer- und Kommentarzeilen in Relation zu LOC in Prozent.

II.2 Definitionsphase

Nach [2] (S. 98) besteht die wichtigste Tätigkeit der Softwareentwicklung in der Erstellung der Produkt-Definition. Dazu werden zuerst die Produktanforderungen ermittelt und im **Pflichtenheft** gesammelt beschrieben. Sie bilden die qualitativen und quantitativen Eigenschaften des Produktes aus der Sicht des Nutzers. Ein vorhandenes Lastenheft kann mit den darin enthaltenen Basisanforderungen als Grundlage verwendet und verfeinert werden. Bei Individualsoftware entstehen Pflichtenhefte meist in enger Zusammenarbeit mit dem Auftraggeber und den späteren Anwendern des Produktes.

Die verbalen Ausführungen der Produktanforderungen lassen jedoch großen Interpretationsspielraum und sollten daher in einem vollständigen, konsistenten **Produkt-Modell** formalisiert werden[4]. Die Erstellung eines solchen Modells wird vereinfacht, wenn die Produktanforderungen selbst bereits in einem hohen Formalisierungsgrad organisiert und beschrieben sind (siehe **II.2.1**).

Nach [2] (S.99) können dann aus Pflichtenheft und Produkt-Modell erste Oberflächen-Prototypen erstellt und mit Auftraggeber und Anwender abgestimmt werden. Ist dieser Prozess abgeschlossen, kann eine Vorabversion des Benutzerhandbuches erstellt werden. Dadurch sind die Entwickler gezwungen, sich in die Rolle des späteren Anwenders zu versetzen.

Alle genannten Dokumente, auch **Artefakte** genannt, ergeben die **Produkt-Definition**. Da diese die Vertragsgrundlage zwischen Auftraggeber und Auftragnehmer[5] bilden kann, sind deren Vollständigkeit, Eindeutigkeit und Widerspruchsfreiheit von großer Bedeutung. Durch weitere Formalisierung der Produkt-Definition entsteht in der Designphase das Produktdesign.

II.2.1 Aufbau von Pflichtenheften

In vielen Fällen ist das Pflichtenheft das alleinige Anforderungsdokument, weil in der Praxis oft auf die Erstellung eines Lastenheftes verzichtet wird. Im Hinblick auf eine gute Produkt-Definition und damit auf einen effizienten und erfolgreichen Entwicklungsprozess, ist dem Pflichtenheft besondere Aufmerksamkeit zu schenken. Zudem bildet es die Grundlage für alle weiteren Aktefakte der Produkt-Definition.

Während des gesamten Softwareentwicklungsprozesses birgt der geringe Formalisierungsgrad von verbalen Ausführungen[6] jedoch stets die Gefahr des Kommunikationsproblems (siehe [7], S. 174ff) durch Missverständnisse zwischen Auftraggeber und -nehmer. Darum sollten Beschreibungen so früh wie möglich formalisiert werden.

Größere Projekte sollten im Hinblick auf Übersichtlichkeit, Wartung und Arbeitsteilung anstatt in der Designphase bereits hier in mehrere Teile (spätere **Subsysteme**) aufgeteilt werden. Im Anschluss an die Erfassung aller Produktanforderungen muss dazu zunächst eine Modularisierung erfolgen, nach der dann die Zuordnung auf das jeweilige Subsystem erfolgt. So entsteht pro System ein separates Pflichtenheft. Für die aufgeteilten Produktanforderungen sind dann wie gewohnt Daten und Funktionen zu modellieren. Zudem müssen dort die zeitlichen Anforderungen (Zugriffszeiten) für den Datenzugriff, Berechnungen und Steuerungsvorgänge sowie die Einsatzumgebung definiert werden.

Für das Gesamtsystem (Steuerprogramm bzw. Programmskelett) sind grundlegende Ansprüche an die Nutzeroberfläche festzuhalten. Dazu können u.a. Menüstruktur sowie Anzahl und Zweck von Dialogfenstern und deren Zuordnung zu den einzelnen Subsystemen zählen.

[4] bei der objektorientierten Softwareentwicklung ist dies i.d.R. ein OOA-Modell
[5] dem/n Softwareentwickler(n)
[6] die bei Gesprächen zwischen Auftraggeber und -nehmer oder bei informalen texturellen Beschreibungen eingesetzt werden, siehe (S. 110)

Für jedes Pflichtenheft benötigt man somit die folgenden Gliederungspunkte:

- Funktionen
- Daten
- Leistungsanforderungen (Dynamik von Daten und Funktionen)
- Nutzungsoberfläche (Ansprüche an diese)
- Einsatzumgebung, Qualitätsmerkmale

Je nach Anwendungszweck spielen natürlich auch andere Punkte eine wichtige Rolle, auf die hier jedoch nicht eingegangen werden kann. Ein universelles Gliederungsmuster wird bei **[2]** (S. 115ff) vorgestellt.

Ein einfaches Mittel für einen semiformalen Ansatz ist die fortlaufende Nummerierung der Elemente aus dem Funktions-, Daten- und Leistungsanforderungsteil. Nach **[2]** (S. 1107ff) ist die Kategorisierung mit **/ F** `<ff>` **/** für Funktionen, **/ D** `<dd>` **/** für Daten und **/ L** `<ll>` **/** für Leistungsanforderungen eine Möglichkeit.

Durch den Einsatz deutlich unterscheidbarer Formatierungen, zahlreicher Abbildungen und Tabellen kann die Eindeutigkeit der verbalen Ausführungen weiter verbessert werden. Dazu zählt auch die Verwendung von Querverweisen innerhalb eines Dokumentes oder Verknüpfungen zu anderen Dateien bzw. Literatur.

Um Auftraggeber und Anwender die Arbeit mit dem Pflichtenheft zu erleichtern, sollte stets ihre Fachterminologie angewendet werden. Um das Verständnis des Entwicklerteams für diese Begriffe zu sichern, sollten sie entweder in einem Glossar gesammelt erläutert oder an den entsprechenden Stellen im Fließtext erklärt, hervorgehoben und in einem Stichwortverzeichnis zusammengefasst werden.

Die verbale Beschreibung der Produktanforderungen sollte stets kurz, knapp und so eindeutig wie möglich erfolgen. Um Beginn und Ende von umfangreichen Erläuterungen deutlich zu kennzeichnen, haben die Autoren dem Text eine vertikale Linie (siehe **Abb. II.4**) vorangestellt. Bedingungen, Hinweise und Warnungen werden zudem besonders gekennzeichnet.

Datenelemente lassen sich gut in einer Tabelle formalisieren. Dabei sind die folgenden grundlegenden Informationen zu integrieren (siehe z.B. **Abb. II.5**):

- Datentyp (logisch, aufgezählt, ganzzahlig, reellwertig oder Zeichenkette)
- Zugriffsart („nur lesen", „nur schreiben" oder „lesen und schreiben")
- Speichertyp (Arbeitsspeicher, Textdatei mit zu nennender Zeichenkodierung, sequentielle Dateistruktur, Datenbank und -typ)
- Speicherort (Position in einem Feld bzw. Dateiname und Hierarchie innerhalb einer Datei)
- Minimal-/ Maximalwert und Nachkommastellen (nur bei ganzen bzw. reellwertigen Zahlen); in einigen Anwendungen kann die Nennung der physikalischen Einheit für diese Werte erforderlich sein
- Anzahl der Zeichen (nur bei Zeichenketten)

Diese Informationen werden in fast allen späteren Entwicklungsschritten benötigt – Oberflächen-Prototyping (**II.2.2**), Erstellung von Hilfe-System (**II.2.3**) und Benutzerhandbuch (**II.2.4**), OOA (**II.2.5 a)** und **c)**), OOD (**II.3.1 a)**, **c)** und **d)** bzw. **II.3.2**), Implementierung (**II.4.1** und u.U. **II.4.2**) und Test (**II.5.1** und **II.5.2**). Sie können so in die Vertragsverhandlungen aufgenommen werden, wenn sie bereits im Pflichtenheft vollständig zusammengetragen wurden.

Um InkonsistenzenT[7]T bei Datenelementen, die in mehreren Subsystemen eines Projektes verwendet werden, zu vermeiden, sind diese nur einmal tabellarisch zu erfassen! Alle weiteren Subsysteme verweisen auf das Pflichtenheft, wo das benutzte Datenelement definiert ist.

Stichwort-, Tabellen- und Abbildungsverzeichnis können zum schnellen Nachschlagen der gewünschten Information beitragen. Auch eine Übersicht über die Produktfunktionen, Leistungsanforderungen, Datenelemente und später der Funktionen der Nutzeroberfläche können hinzugefügt werden.

Weil ein solches Pflichtenheft noch keine konkreten Oberflächenentwürfe enthält – diese entstehen erst während des Human-Interface-Prototypings; siehe **II.2.2** – ist die Dekomposition von Nutzeroberfläche und Funktionskomponente hier bereits voll umgesetzt.

Der Oberflächenentwurf ermöglicht Auftraggeber und Anwender einen realistischen Einblick in das spätere Produkt, weil die Produktfunktionen und -daten visualisiert werden. Um den Abstraktionsgrad des Pflichtenheftes zu senken, können die mit dem Auftraggeber abgestimmten Oberflächenfenster nachträglich in den Gliederungspunkt ‚Nutzeroberfläche' aufgenommen werden. Dort erfolgt die Erläuterung der Oberflächen, indem die Bedien- oder Steuerelemente beschrieben und durch Querverweise mit dem Daten- bzw. Funktionsteil verknüpft werden. Für eine konkrete Anwendung siehe z.B. **Abb. II.9**. Die Dekomposition bleibt dabei bestehen, weil Funktionen und Nutzeroberfläche weiterhin deutlich von einander getrennt sind.

Besonders empfehlenswert finden die Autoren die Erweiterung der oben vorgestellten Nummerierung auf die Elemente des neuen Gliederungspunktes. Dazu sind die Beschreibungen von Bedien- und Steuerelementen mit **/ B** `<bb>` **/** eindeutig zu adressieren (siehe z.B. **Abb. II.9**).

- Dialogfenster betreten
  - anzeigen der physikalischen Einheit für alle Ein- und Ausgabewerte
  - Verweis auf „aktuelle Istposition anzeigen" und „aktuelles Offset anzeigen"
  - beschriften der Titelleiste mit der Bezeichnung des Antriebs
- ‚aktuelle Istposition' anzeigen
- ‚aktuelles Offset' anzeigen
- Offset neu definieren
  - ‚entspricht Winkel'
  - ‚Offset eingeben'
- Dialogfenster verlassen
  - ‚OK' bzw. ‚Abbruch'

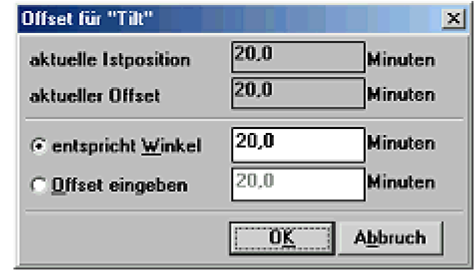


Abb. II.2 Realisierung von Funktionen und Daten als Bedien- und Steuerelemente eines Oberflächenfensters am Beispiel des Dialogfensters ‚Offset für <Antrieb>'

Die Integration des Oberflächenentwurfs stellt so die Vollständigkeit der Produkt-Definition und der Nutzeroberfläche sicher, weil:

- einerseits Produktanforderungen im Oberflächenentwurf unterschlagen worden sein könnten. Sie sind dann mit keinem Bedien- oder Steuerelement verknüpft.
- Andererseits sollte jedes Bedienelement mit dem Daten- oder Funktionsteil verknüpft sein. Wenn keine Querverweise enthalten sind, ist zu prüfen, ob das Pflichtenheft ergänzt werden muss. Es könnte sich aber auch um Funktionen handeln, die nur diese spezielle Nutzeroberfläche betreffen (z.B. das Anzeigen/ Verstecken anderer Oberflächenfenster)!

[7] insbesondere nach Änderungswünschen oder Wartungsarbeiten

So werden Mängel frühzeitig sichtbar gemacht, die sonst u.U. erst beim Einsatz des Produktes zu Tage getreten wären. Ein, um die Nutzeroberfläche erweitertes, Pflichtenheft erleichtert zudem die Erstellung des Hilfe-Systems und des Benutzer-Handbuchs (siehe **II.2.3** bzw. **II.2.4**).

Anwendungsfall: ‚Manuelle Justage'

Wie bereits erwähnt, besteht das ‚XCtl'-Steuerprogramm aus mehreren Subsystemen (siehe **Abb. I.3**). Für den Anwendungsfall ‚Manuellen Justage' konnte deshalb ein separates Pflichtenheft (**[M15]**) entstehen. Um die oben beschriebenen stilistischen und methodischen Hilfsmittel nutzen zu können, wurde das Pflichtenheft mit dem Textverarbeitungsprogramm Microsoft Word erstellt. Dies ermöglichte den Einsatz von Formatierungen und die Navigation innerhalb eines Dokumentes, ähnlich wie Hyperlinks in html-Dokumenten.

Im ersten Teil des Pflichtenheftes erfolgte die Zielbestimmung, welche aus den Basisanforderungen (**II.1**) abgeleitet wurden. Des Weiteren wurden Einsatz- und Entwicklungsumgebung, Qualitätsanforderungen und die Zielgruppe beschrieben.

Abgeschlossen wurde dieser Abschnitt mit einer Kapitelübersicht und einem Verweis auf die Layoutkonventionen, wo u.a. die Nomenklatur von Steuerelementen erläutert wird. Der letzte Teil wurde ausgelagert, weil er die Grundlage für alle von den Autoren erstellten Artekfakte bildet und somit dokumentübergreifend angewendet wird.

Im Kapitel ‚funktionale Beschreibung' (entspricht ‚Funktionen' aus o.g. Gliederung) werden die gewünschten Produktfunktionen ausführlich erläutert (siehe **Abb. II.3**).

II.1 Wiederherstellungen beim Start
II.2 Antriebsauswahl
II.3 Istposition
II.4 Offset für <Antrieb>/ Relative Null
II.5 Betriebsarten
 II.5.1 Direktbetrieb
 II.5.2 Fahrbetrieb
 II.5.3 Schrittbetrieb
II.6 Bewegungsparameter
 II.6.1 Sollposition
 II.6.2 Bewegungsgeschwindigkeit
 II.6.3 Schrittweite
II.7 Halbwertsbreitenmessung
II.8 <PSD>-Offset

Abb. II.3 Auszug aus der Gliederung des Pflichtenheftes **[M15]**;
Kapitel ‚funktionale Beschreibung'

Dazu gehören die benötigten physikalischen Vorgänge und Begrifflichkeiten sowie der Zweck der Datenelemente. Um Missverständnisse zwischen Entwicklern und Auftraggebern zu vermeiden, wurden alle Fachbegriffe nur einmalig im Text erläutert und fett hervorgehoben. Dazu gibt es ein Stichwortverzeichnis (siehe **[M15]**, Anhang B), in dem die Seitenzahlen der Begriffsdefinitionen bzw. Verweise auf andere Begriffe zu finden sind. Im Stichwortverzeichnis aufgeführte Fachtermini sind bei jeder Verwendung im Text kursiv formatiert ( ).

/ F 70 / – „Starten der Bewegung im Direktbetrieb"
Nach der Beschleunigungsphase bewegt sich der Antrieb mit der angegebenen *Geschwindigkeit*, um kurz vor dem Ziel abzubremsen und an der *Sollposition* zu halten. Wenn die *Sollposition* größer (bzw. kleiner) ist als die *Istposition*, bewegt sich der Antrieb vorwärts (bzw. rückwärts). Die maximale Abweichung zwischen der angefahrenen und der gewünschten Sollposition darf maximal DeathBand (siehe **Tabelle 14**) betragen. Wenn *Ist-* und *Sollposition* übereinstimmen, findet keine Bewegung statt. Die Genauigkeit beträgt hier wiederum Deathband.

Bedingung
Der Antrieb darf sich nicht bewegen. Wenn eine *Sollposition* eingegeben wird, die außerhalb des zulässigen Wertebereichs liegt, so wird die Positionsangabe auf die minimal oder maximal zulässige Position korrigiert, je nachdem, ob der zulässige Wertebereich unterschritten oder überschritten wurde.

Kennzeichnung für den Gültigkeitsbereich/ Umfang dieser Produktfunktion

Abb. II.4 Auszug aus dem Funktionsteil des Pflichtenheftes **[M15]**

Kernpunkt dieses Kapitels ist die Beschreibung der insgesamt 15 Produktanforderungen. Wie oben erwähnt, sind diese mit **/ F** `<ff>` **/** nummeriert und besonders formatiert (siehe **Abb. II.4**). Eine Zusammenstellung aller Produktfunktionen ist im Anhang E des Pflichtenheftes zu finden. Ist die Ausführung einer Produktfunktion an Bedingungen geknüpft (), sind diese gesondert nach der Beschreibung angegeben. Weil Funktionen oft mit vielen Datenelementen verknüpft sind, wurden keine Querverweise auf die Tabellen dieser Daten eingefügt. Allein in **/ F 70 /** wären 21 Verweise notwendig gewesen!

Da sich die Benennung der Gliederungspunkte im Kapitel ‚Daten' grundsätzlich nach den kursiv geschriebenen Fachbegriffen des Funktionsteils richtet, ist das schnelle Auffinden der entsprechenden Tabelle unproblematisch. Nur für selten verwendete Begriffe () wurden Querverweise eingefügt.

Das dritte Kapitel beschäftigt sich mit den zu verwaltenden Daten, welche für die Produktfunktionen benötigt werden. Neben diesen 20 Einträgen musste bei der späteren Integration der Oberflächen-Prototypen hier ein zusätzlicher Eintrag hinzugefügt werden. Dieses Datenelement wird zum Auswählen bzw. Anzeigen des zuletzt benutzten Antriebs in einem Steuerelement benötigt und wäre ohne den Nachtrag der Oberflächenfenster unterschlagen worden.

Insgesamt werden somit 21 Datenelemente zur Realisierung der neue ‚Manuellen Justage' benötigt. Darunter befinden sich sieben Elemente, die durch den Anwender manipuliert werden können. Da solche Datenelemente direkt an eine Produktfunktion gekoppelt sind, konnte hier auf eine ausführliche Erläuterung verzichtet werden. Diese Datenelemente sind als Fachtermini im Stichwortverzeichnis aufgeführt.

Die übrigen 14 Datenelemente werden durch die neue ‚Manuelle Justage' nur gelesen und stellen i.d.R. weiterführende Informationen (wie Minimal- oder Maximalwert, Nachkommastellengenauigkeit) für die manipulierbaren Datenelemente dar. Wie **Abb. II.5** () zeigt, finden sich diese Tabellen als Querverweis in den manipulierbaren Datenelementen wieder.

/ D 110 / – „eingegebene Sollposition"
Die Sollposition ist – wie die *Istposition* – abhängig vom aktuellen *Offset* des Antriebs. Die Minimal- und Maximalwerte können demnach **Tabelle 7** entnommen werden; ebenso die Nachkommastellengenauigkeit. Die Sollposition ist bei jeder Änderung zu speichern!

Bezeichnung	eingegebene Sollposition
Typ | Zugriff | Eintrag	ℝ | lesen und schreiben | NEU
Speichertyp	als Eintrag in einer ini-Datei (ASCII-Text)
Speicherort	**HARDWARE.INI –> [MOTOR<M>] –> MJ_AnlgeDest**
Minimalwert	siehe **Tabelle 8**
Maximalwert	siehe **Tabelle 9**
Standardwert	aktuelle Istposition
Nachkommastellen	siehe **Tabelle 10**

Tabelle 13 „Eigenschaften der Sollposition"

Abb. II.5 Auszug aus dem Datenteil des Pflichtenheftes **[M15]**

Zum besseren Verständnis wurde eine allgemein bekannte Symbolik (**Abb. II.5**, z.B. für reelle Zahlen) gewählt. Eine genaue Typfestlegung (wie `float`) wäre hier nicht möglich, weil die Programmiersprache bei neuen Projekten zu diesem Zeitpunkt noch nicht feststeht. Zudem wären solche Angaben für Auftraggeber und Anwender schwer verständlich.

Existiert bereits ein Referenzprodukt, können die daraus übernommenen Datenelemente speziell gekennzeichnet werden (siehe **Abb. II.5**, ‚bereits vorhanden' anstatt ‚NEU'). Obwohl fast alle Elemente der neuen ‚Manuellen Justage' aus anderen Subsystemen stammen,

konnten keine Verknüpfungen zu deren Dokumentationen hergestellt werden, da die Elemente nirgends detailliert beschrieben werden. Nur die Verhaltensspezifikation [M14] half in Ansätzen, indem sie exemplarische Werte einer Konfigurationsdatei zur Verfügung stellte. Die genauen Eigenschaften wurden durch ein problembezogenes Programmcode-Review der existierenden ‚Manuelle Justage' und der benutzten Subsysteme ermittelt.

Die neuen Datenelemente wurden wie beim Neuentwurf üblich mit dem Anwender abgestimmt. Insgesamt werden so 15 vorhandene, vier neue (mit den Eigenschaften bereits vorhandener) und zwei völlig neue Datenelemente beschrieben (siehe Abb. II.6).

/ D 10 / – „Name des ausgewählten Antriebs für einen Teilbereich"
/ D 20 / – „PSD-Röntgendetektor vorhanden"
/ D 30 / – „Antriebsliste"
/ D 40 / – „physikalische Einheit"
/ D 50 / – „Eigenschaften der Istposition"
/ D 60 / – „minimale Istposition"
/ D 70 / – „maximale Istposition"
/ D 80 / – „Nachkommastellen"
/ D 90 / – „Offset"
/ D 100 / – „Betriebsart"
/ D 110 / – „eingegebene Sollposition"
/ D 120 / – „eingegebene Bewegungsgeschwindigkeit"
/ D 130 / – „minimale Bewegungsgeschwindigkeit"
/ D 140 / – „maximale Bewegungsgeschwindigkeit"
/ D 150 / – „eingegebene Schrittweite"
/ D 160 / – „minimale Schrittweite"
/ D 170 / – „maximale Schrittweite"
/ D 180 / – „eingegebener PSD-Kanal"
/ D 190 / – „erster Kanal"
/ D 200 / – „letzter Kanal"
/ D 210 / – „Winkelwert/Kanal"

Abb. II.6 Auszug aus dem Anhang des Pflichtenheftes [M15] (Übersicht aller Datenelemente); Hervorhebung: schwarz die vorhandenen; **blau** die abgeleiteten und **rot** die neuen Elemente

Die Gliederung des Datenteils ähnelt der von Kapitel ‚funktionale Beschreibung' (vgl. Abb. II.3 und Abb. II.7). Weil nicht jede Produktfunktion Daten verwaltet, konnten im Kapitel ‚Daten' einige Gliederungspunkte entfallen. Um das Nachschlagen der verwendeten Datenelemente zu erleichtern, sind die Punkte sind in beiden Kapiteln identisch benannt und geordnet.

III.1 Wiederherstellungen beim Start	III.6 Bewegungsparameter
III.2 Antriebsauswahl	III.6.1 Sollposition
III.3 Istposition	III.6.2 Bewegungsgeschwindigkeit
III.4 Offset für <Antrieb>/ Relative Null	III.6.3 Schrittweite
III.5 Betriebsarten	III.7 <PSD>-Offset

Abb. II.7 Auszug aus der Gliederung des Pflichtenheftes [M15]; Kapitel ‚Daten'

Auf den Gliederungspunkt <u>Leistungsanforderungen</u> wurde verzichtet. Nach der Analyse der ursprünglichen ‚Manuelle Justage' schien die Ausführung der Produktfunktionen und der Zugriff auf die benötigten Daten problemlos möglich zu sein (einfache Berechnungen oder Speicherzugriffe). Durch das robuste Design der Funktionskomponente, das bei der Dekomposition entstand, traten durch die Hardwareansteuerung an den Messplätzen der Physik widererwartend Geschwindigkeitsprobleme auf, die nachgebessert werden mussten (siehe dazu II.4).

Im anschließenden Kapitel <u>‚Benutzeroberfläche'</u> (entspricht ‚Nutzeroberfläche' aus o.g. Gliederungsschema) wurden die Wünsche der Mitarbeiter des Physikalischen Institutes an die neue Nutzeroberfläche festgehalten. Dazu erfolgte auch eine Analyse und Bewertung des ursprünglichen Dialogfensters (siehe II.2.2). Die gewonnenen Erfahrungen wurden zunächst

stichpunktartig in diesem Kapitel festgehalten und bildeten das Grundgerüst für die Entwicklung der Oberflächen-Prototypen (siehe **II.2.2**).

Nachdem die Prototypen erstellt und mit den Mitarbeitern des Physikalischen Institutes abgestimmt waren, wurden diese Notizen wieder entfernt. Der von den Anwendern favorisierte Oberflächenentwurf wurde nachträglich in dieses Kapitel aufgenommen und beschrieben. Weil der Entwurf aus drei Dialogfenstern besteht, bilden diese die erste Gliederungsebene des Kapitels. Dort ist jedes Fenster mit exemplarischen Werten für die Bedien- und Steuerelemente abgebildet. (**Abb. II.2**, rechte Seite).

In der nächsten Gliederungsebene befinden sich jeweils die Punkte aus dem Kapitel ‚funktionale Beschreibung' (siehe **Abb. II.3**). Es müssen alle Produktfunktionen genannt werden! Zusätzlich wurden auch oberflächenspezifische Funktionen – z.B. zum Betreten und Verlassen des Dialogfensters – aufgenommen, die im Funktionsteil nicht beschrieben werden konnten.

Den Produktfunktionen (z.B. **Abb. II.8**,) wurden die betroffenen Steuerelemente untergliedert. Wie im Beispiel der Istposition können das auch mehrere sein (sowohl Bildlaufleiste als auch Eingabefeld zur Anzeige der Antriebsposition).

Die Gliederungspunkte sind entweder die Beschriftungen der Steuerelemente in Hochkommata (**Abb. II.8**,) oder eine abstrakten Aufgabenbeschreibung (**Abb. II.8**,) bei unbeschrifteten Steuerelementen. Variable Anteile in der Beschriftung der Steuerelemente sind durch (in Spitzklammern eingeschlossene) Platzhalter realisiert (z.B. <Antrieb> als Platzhalter für die Bezeichnung des Antriebs, der diese Funktion anbietet).

IV.1 Hauptdialogfenster ‚Manuelle Justage'
- IV.1.1 Dialogfenster betreten
- IV.1.2 Antriebsauswahl
- IV.1.3 Istposition
  - IV.1.3.a absoluter Wert
  - IV.1.3.b relative Positionierung
- IV.1.4 Offset für <Antrieb>/ Relative Null
  - IV.1.4.a Relative Null ‚setzen'
  - IV.1.4.b Relative Null ‚aufheben'
  - IV.1.4.c ‚Offset…'
  - IV.1.4.d Offset für <Antrieb> anzeigen
- IV.1.5 Betriebsarten
  - IV.1.5.a ‚Direktbetrieb'
  - IV.1.5.b ‚Fahrbetrieb'
  - IV.1.5.c ‚Schrittbetrieb'
- IV.1.6 Bewegungsparameter
  - IV.1.6.a ‚Sollposition'
  - IV.1.6.b ‚Geschwindigkeit'
  - IV.1.6.c ‚Schrittweite'
- IV.1.7 Halbwertsbreitemessung
- IV.1.8 ‚PSD-Offset…'
- IV.1.9 Hilfe
- IV.1.10 Dialogfenster verlassen

IV.2 Dialogfenster ‚Offset für <Antrieb>'
- IV.2.1 Dialogfenster betreten
- IV.2.2 Ausgangsdaten
  - IV.2.2.a ‚aktuelle Istposition'
  - IV.2.2.b ‚aktueller Offset'
- IV.2.3 Offset für <Antrieb> definieren
  - IV.2.3.a ‚entspricht Winkel'
  - IV.2.3.b ‚Offset angeben'
- IV.2.4 Dialogfenster verlassen
  - IV.2.4.a ‚OK'
  - IV.2.4.b ‚Abbruch'

IV.3 Dialogfenster ‚Offset für <PSD>'
- IV.3.1 Dialogfenster betreten
- IV.3.2 Ausgangsdaten
  - IV.3.2.a ‚Istposition von Theta'
  - IV.3.2.b ‚Winkelwert pro Kanal'
- IV.3.3 PSD-Kanaloffset definieren
  - IV.3.3.a ‚neu zugeordneter Kanal'
- IV.3.4 Ausgabedaten
  - IV.3.4.a ‚PSD-Kanaloffset'
  - IV.3.4.b ‚entspricht Winkel (Kanal 0)'
- IV.3.5 Dialogfenster verlassen
  - IV.3.5.a ‚OK'
  - IV.3.5.b ‚Abbruch'

Abb. II.8 Auszug aus der Gliederung des Pflichtenheftes **[M15]**; Kapitel ‚Benutzeroberfläche'

Die Beschreibung des Steuerelementverhaltens ist prinzipiell genauso aufgebaut wie die einer Produktfunktion. Die Nummerierung erfolgt mit / **B** `<bb>` / (für Benutzeroberfläche). Jeder Beschreibung ist ein Ausschnitt aus dem Dialogfenster vorangestellt, der das entsprechende Steuerelement zeigt (**Abb. II.9**,). So ist der Bezug zum Dialogfenster auch bei unbeschrifteten Steuerelementen garantiert.

Im Kern steht oft der direkte Verweis auf die / **F** `<ff>` /-Funktion (siehe **Abb. II.9**,). Änderungen oder zusätzliche Bedingungen, speziell für die Oberfläche, sind separiert (siehe **Abb. II.9**,) – z.B. Zustände der anderen Teilbereiche, die beachtet werden müssen.

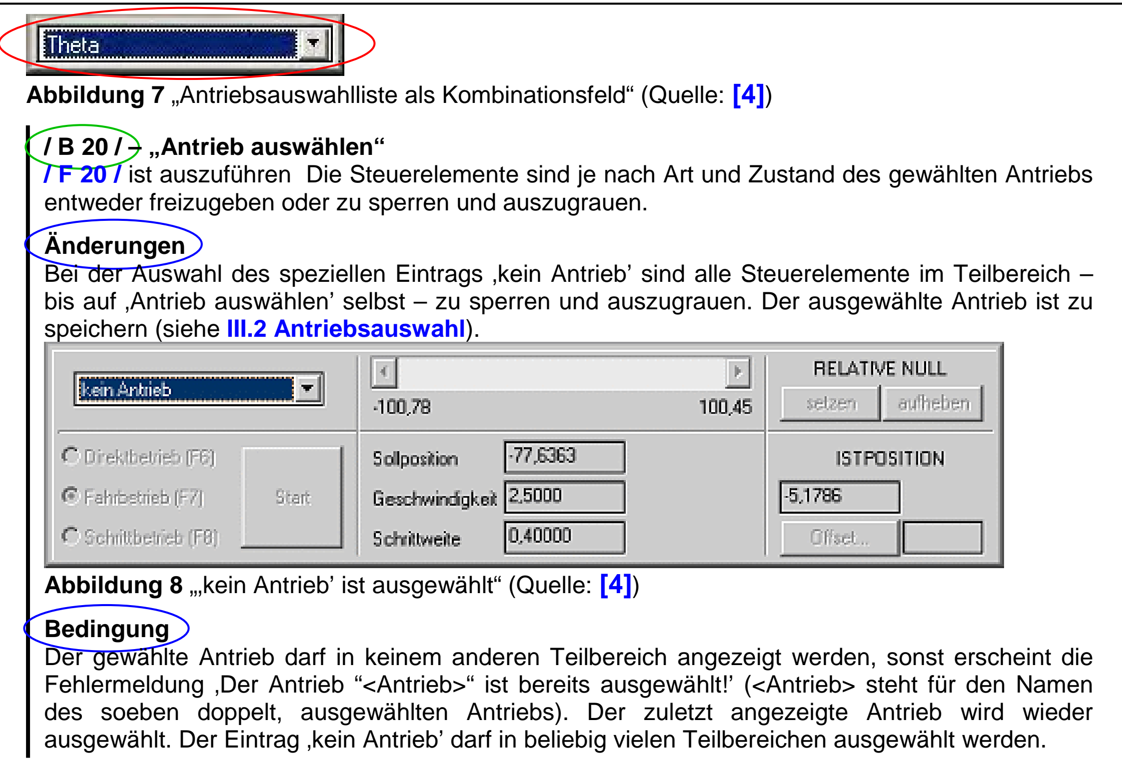


Abbildung 7 „Antriebsauswahlliste als Kombinationsfeld“ (Quelle: [4])

/ B 20 / – „Antrieb auswählen“
/ F 20 / ist auszuführen Die Steuerelemente sind je nach Art und Zustand des gewählten Antriebs entweder freizugeben oder zu sperren und auszugrauen.

Änderungen
Bei der Auswahl des speziellen Eintrags ‚kein Antrieb’ sind alle Steuerelemente im Teilbereich – bis auf ‚Antrieb auswählen’ selbst – zu sperren und auszugrauen. Der ausgewählte Antrieb ist zu speichern (siehe **III.2 Antriebsauswahl**).

Abbildung 8 „‚kein Antrieb’ ist ausgewählt“ (Quelle: [4])

Bedingung
Der gewählte Antrieb darf in keinem anderen Teilbereich angezeigt werden, sonst erscheint die Fehlermeldung ‚Der Antrieb “<Antrieb>“ ist bereits ausgewählt!’ (<Antrieb> steht für den Namen des soeben doppelt, ausgewählten Antriebs). Der zuletzt angezeigte Antrieb wird wieder ausgewählt. Der Eintrag ‚kein Antrieb’ darf in beliebig vielen Teilbereichen ausgewählt werden.

Abb. II.9 Auszug aus dem Teil Benutzeroberfläche des Pflichtenheftes **[M15]**

II.2.2 Oberflächen-Prototyping

Nach [2] (S. 100) ist ein **Oberflächen-Prototyp** (auch Oberflächenentwurf oder Human-Interface Prototyp) „ein provisorisches, ablauffähiges Software-System“, bei dem die Bedienoberfläche „ohne die dahinter liegenden fachlichen Funktionen“ realisiert ist. Ziel ist es, dem Auftraggeber und den späteren Anwendern einen realistischen Eindruck der Nutzeroberfläche zu geben. Die Oberflächen-Prototypen entsprechen stets dem aktuellen Erkenntnisstand des Entwicklerteams!

Diese sollen aus dem Produkt-Modell entstehen (siehe [2], S. 99) im Falle der Objektorientierung also erst nach der OOA. Durch den starken Formalisierungsgrad, die deutliche Gliederung und die dargelegten Informationen in einem Pflichtenheft, das nach **II.2.1** entstanden ist, empfehlen die Autoren das Oberflächen-Prototyping direkt auf dem Pflichtenheft aufzubauen.

Grundanliegen ist, dass so wieder früh Missverständnisse zwischen den beiden Parteien aufgedeckt und geklärt werden können. Dadurch müssen nur Pflichtenheft und Oberflächenentwürfe geändert werden – nicht aber das OOA-Modell, weil dieses nun erst nach dem Prototyping entsteht (ausführlich diskutiert bei **II.2.6**).

Zur Ableitung der Oberflächenfenster existieren allgemeine Regelwerke. Ein so gewonnener Oberflächenentwurf bildet aber nur die erste Grundlage. Für die Anordnung der Steuerelemente ist zudem das Verständnis über das System und die damit realisierten Aufgaben unerlässlich! Die Ansprüche der Anwender an die zukünftige Nutzeroberfläche müssen ebenfalls Beachtung finden. Dazu sind die Notizen im Kapitel ‚Benutzeroberfläche’ des Pflichtenheftes zu verwenden.

Nach Abstimmung der Oberflächen-Prototypen mit dem Auftraggeber/ Anwender steht die zukünftige Nutzeroberfläche im Wesentlichen fest. Nun kann nach **II.2.1** die Ersetzung der Notizen durch die Beschreibung des gewählten Oberflächenentwurfs erfolgen.

Anwendungsfall: ‚Manuelle Justage'

Zu Beginn musste die existierende Nutzeroberfläche unter dem Gesichtspunkt der **Software-Ergonomie** analysiert und kritisch bewertet werden. Der erste Kontakt mit dem ursprünglichen Dialogfenster erfolgte bereits im Vorbereitungsseminar ‚Projekt Softwaresanierung', beim Review der Verhaltenspezifikation **[M1]**. Dabei fiel sofort die unvorteilhafte Anordnung der Steuerelemente auf, welche den Arbeitsfluss behindert. Häufig benutzte Steuerelemente sind zu weit voneinander entfernt. Zudem sind die angebotenen Tastenkombinationen nicht konsequent zugewiesen und für einige grundlegende Funktionen sind gar keine definiert.

Bei der eigentlichen Analyse wurde intensiv mit dem existierenden Dialogfenster gearbeitet. Die Mängel und Fehler wurden vorerst nur in einer Fehlerliste **[M19]** dokumentiert, da sie aufgrund der fehlenden Einarbeitung in die Implementierung noch nicht behoben werden konnten.

Als Resultat dieser Arbeit entstand eine ausführliche Bewertung (**[M16]**) die der ursprünglichen Nutzeroberfläche eine für den Anwendungszweck unzweckmäßige Struktur bescheinigt. Daraus ergab sich auch eine Liste von Zielen für den Neuentwurf. Diese wurde in die Vorabversion des Pflichtenheftes Kapitel ‚Benutzeroberfläche' als Notizen übernommen (siehe **II.2.1**). Dazu zählten u.a., dass die im aktuellen Programmzustand verfügbaren Steuerelemente visuell hervorgehoben werden, indem alle übrigen Elemente gesperrt und ausgegraut sind. Dieses Vorgehen ist besonders wichtig, weil die beiden Anwendergruppen z.T. unterschiedliche Produktanforderungen stellen. Nur für den Bereich ‚Topographie' ist z.B. die Halbwertsbreitenmessung erforderlich. Der Bereich Diffraktometrie/ Reflektometrie benötigt den PSD-Offset, der bei der ‚Topographie' nicht verfügbar sein darf. Die im ursprünglichen Dialogfenster definierten Tastenkombinationen wurden für wichtige Steuerelemente wieder übernommen. Inhaltlich ähnliche Steuerelemente oder Elemente, die im Arbeitsprozess in ständigem Wechsel benutzt werden, wurden gruppiert. Größen und Abstände wurden vereinheitlicht, Freiflächen vermieden.

Die Mitarbeiter des Physikalischen Institutes akzeptierten die entwicklerseitigen Vorschläge und wünschten zusätzlich die Aufteilung des Hauptdialogs, damit die wichtigsten Antriebsparameter von drei Antrieben gleichzeitig sichtbar sind. Die vorhandene horizontale Bildlaufleiste, zur Anzeige der aktuellen Antriebsposition bzgl. der minimal und maximal möglichen Positionen war beizubehalten.

Diese Treffen hatten auch eine Erweiterung der Produktfunktionen zur Folge. Es sollten der Zustand und Inhalt der Steuerelemente bei jeder Änderung gesichert werden. So ist der letzte gültige Zustand nach einem Programmabsturz wiederherstellbar.

Als Ergebnis entstanden sechs Oberflächen-Prototypen, welche die ermittelten Unzulänglichkeiten auf unterschiedliche Weise beheben (siehe **[M16]**, Vorschlag 1 bis 6). Es resultierten drei sehr komplexe Oberflächenfenster aus der gleichzeitigen Anzeige von drei Antrieben (siehe z.B. **Abb. I.10**, oben). Mit mehr als 90 Steuerelementen (allein 29 pro Antrieb) schienen diese zu komplex. Deshalb entstanden zwei Entwürfe, wo nur die Daten eines ausgewählten Antriebs angezeigt werden (siehe **Abb. I.8**, mittlerer Entwurf). Die Anzahl der Steuerelemente konnte so auf ⅓ reduziert werden. Der letzte Vorschlag versuchte diese beiden Konzepte zu kombinieren (siehe **Abb. I.8**, Entwurf ganz rechts), indem nur die wichtigsten Parameter der drei Antriebe und die ausgewählte Statusinformationen des aktuell ausgewählten Antriebs angezeigt wurden.

Aufgrund der komfortablen und vielschichtigen Gestaltungsmöglichkeiten wurde die Borland Delphi-Entwicklungsumgebung zur Erstellung der Oberflächenentwürfe benutzt. So entstanden schnell und einfach Realisierungen, die mit einer vom ‚XCtl'-Projekt unabhängigen Pseudofunktionalität belegt waren.

Den Mitarbeitern des Physikalischen Institutes wurde schließlich ein komplexer und der letztgenannte Dialog vorgestellt, wobei überraschend der komplexere Vorschlag favorisiert

wurde. Der entscheidende Vorteil war die Sichtbarkeit aller Antriebsparameter. Neben minimalen Layoutveränderungen (vgl. **Abb. I.10**, oben mit **Abb. II.10**, oben) wurde zudem der Wunsch einer weiteren Produktanforderung geäußert, der als PSD-Offset nachträglich in das Pflichtenheft aufgenommen werden musste.

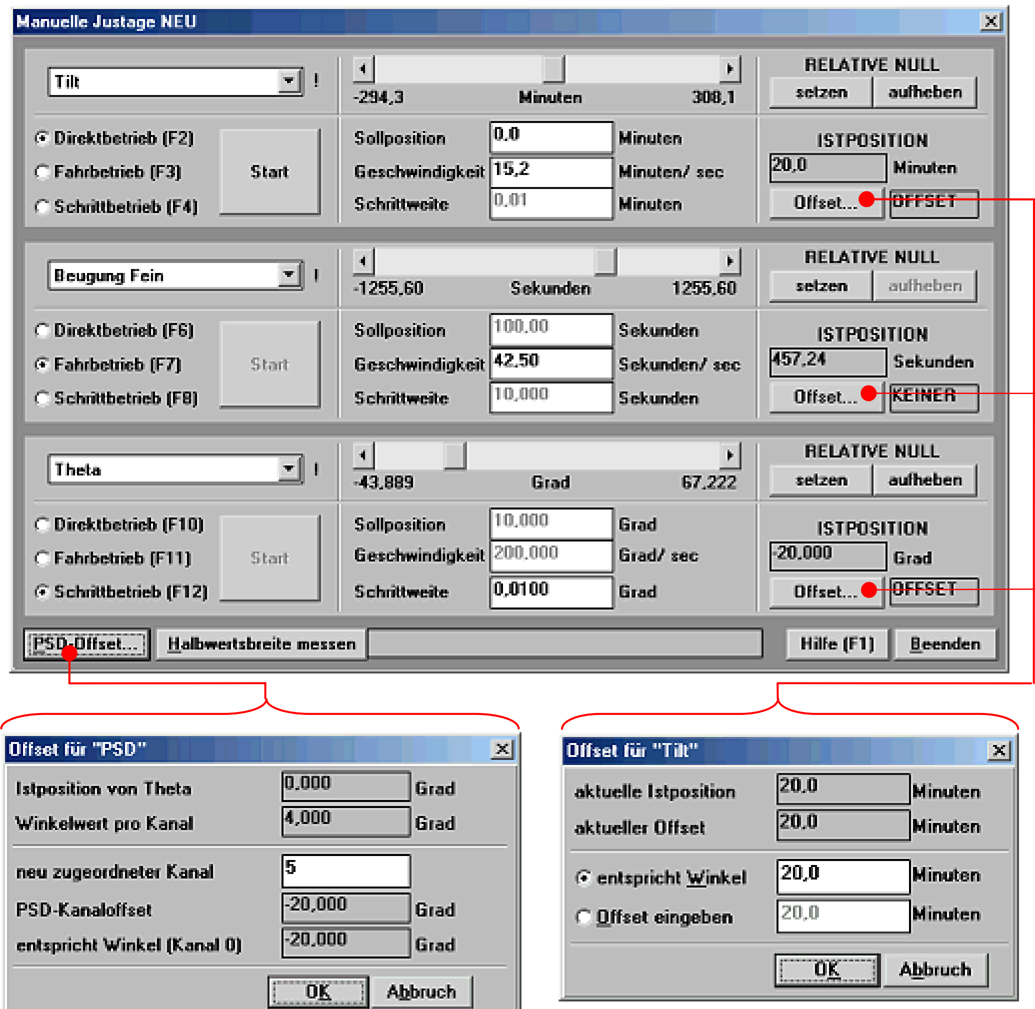


Abb. II.10 Neuentwürfe des Hauptdialogs ‚Manuellen Justage NEU' (oben); Dialog ‚Offset für <PSD>' (links) und Dialog ‚Offset für <Antrieb>' (rechts)

Durch die Produktanforderungen an die beiden Offsets waren zwei zusätzliche Dialogfenster erforderlich. Aufgrund ihrer Einfachheit waren hier nicht mehrere Entwürfe erforderlich. Die beiden Offset-Dialoge sind über Schaltflächen (siehe **Abb. II.10**) in den Human-Interface-Prototyp integriert und per E-Mail als ausführbare exe-Datei verschickt, getestet und akzeptiert.

Wie unter **II.2.1** beschrieben, wurde das Pflichtenheft daraufhin um die favorisierten Oberflächenentwürfe erweitert und dabei mit dem Funktions- und Datenteil verknüpft. Bei der Erstellung der Prototypen und Belegung mit einer Pseudofunktionalität wurde die Notwendigkeit von zusätzlichen, oberflächenspezifischen Funktionen deutlich, die im Teil ‚Nutzeroberfläche' des Pflichtenheftes ausführlich erläutert werden mussten. Bei der neuen ‚Manuellen Justage' waren dies hauptsächlich Wechselwirkungen zwischen den drei Teilbereichszuständen. Zur konfliktfreien Bedienung wurde mit den Mitarbeitern des Physikalischen Institutes z.B. festgelegt, dass ein Antrieb nicht in mehreren Bereichen des Hauptdialogs gleichzeitig angezeigt werden darf. Mit der Beschreibung solcher Eigenheiten im Pflichtenheft können spätere Probleme/ Diskussionen vermieden werden.

II.2.3 Hilfe-System

Bei den umfangreicher werdenden Softwaresystemen benötigt der Anwender zunehmend Unterstützung. Dies bezieht sich sowohl auf die Bedienung der Nutzeroberfläche, als auch auf den damit angebotenen Funktionsumfang. Je nach Produkt können die Anforderungen an ein Hilfe-System dabei sehr unterschiedlich sein. Dementsprechend gibt es zahlreiche Klassifizierungen.

Im Gegensatz zu Hilfsmitteln, die dem Anwender in gedruckter Form vorliegen (z.B. Benutzerhandbuch), erlaubt ein elektronisches Hilfe-System eine gewisse Anpassung an die aktuellen Bedürfnisse des Anwenders. So wird bei **dynamischen Hilfe-Systemen** bspw. der aktuelle Programmkontext verwendet, um ein entsprechendes Hilfe-Thema vorzuschlagen[8]. **Statische Hilfe-Systeme** hingegen liefern, unabhängig vom aktuellen Programmzustand, immer denselben Hilfe-Text.

Zudem variieren die Anforderungen an ein Hilfe-System auch stark durch die unterschiedlichen Fähigkeiten der jeweiligen Anwender. Beim Suchen/ Vorschlagen eines Hilfe-Themas berücksichtigt ein **individuelles Hilfe-System** daher das bisherige Nutzungsverhalten des Anwenders. Dies erfordert jedoch die Erstellung eines umfangreichen Benutzermodells (siehe **[2]**, S 670ff), für das Informationen gesammelt und aufbereitet werden müssen. Dazu zählt in erster Linie wann, wie und wie oft eine Produktfunktion benutzt wird. Weil die Erstellung einer solchen Hilfe und die Integration ins Softwaresystem extrem zeitaufwendig sind, bilden heute (noch) **uniforme Hilfe-Systeme** den Großteil der Hilfe-Systeme. Sie liefern unabhängig vom aktuellen Anwender immer dieselben Informationen.

In Zukunft werden **aktive Hilfe-Systeme** das Benutzungsverhalten der Anwender beobachten und bei der Verwendung von ausgewählten Funktionen selbständig Hilfe anbieten. Dieses Verhalten wird insbesondere erforderlich, weil die durch den Anwender vermuteten Konzepte mit zunehmender Softwarekomplexität immer stärker von der tatsächlichen Funktionalität abweichen. Solche Systeme werden aber nie in ihrer Reinform auftreten, sondern stets durch **passive Hilfe-Systeme** ergänzt werden, die warten, bis der Anwender selbst aktiv wird und eine Hilfeleistung anfordert.

<u>Anwendungsfall: ‚Manuelle Justage’</u>

Zu Beginn dieser Diplomarbeit war das Hilfe-System des ‚XCtl’-Projektes nahezu im gleichen Zustand wie es der Erstentwickler Heiko Damerow 1998 (siehe **I.3**) hinterlassen hatte. Die Hilfe bestand aus einer rtf-Datei, die mit dem Microsoft Hilfe-Compiler in eine windowskompatible hlp-Datei konvertiert werden konnte. Anfangs existierte nur ein Hilfe-Thema, bestehend aus drei Seiten. Während der Analyse- und frühen Definitionsphase für die neue ‚Manuelle Justage’, haben Sebastian Freund und Derrick Hepp diese um eine Einleitung in das von ihnen erstellte Subsystem der ‚Automatischen Justage’ ergänzt.

Für die Subsysteme ‚Diffraktometrie/ Reflektometrie’ und die ‚Graphische Darstellung der Messwerte’ folgten wenig später durch Stephan Berndt, Jens Ullrich und Bernhard Buss zwei Hilfemodule im **CVS**[9]. Durch die schlechte Wartbarkeit der rtf-Dateien, aufgrund der Syntax, die der Hilfe-Compiler benötigt, entschieden sich die drei Autoren für das Werkzeug HelpScribble von Just Great Software. Diese Anwendung ermöglicht eine komfortable Bearbeitung/ Wartung der rtf-Dateien. Im Hinblick auf die begrenzten Speicherressourcen wurden für beide Subsysteme getrennte Hilfedateien erstellt, um nur die benötigten Dateien

---

[8] Im einfachsten Fall ruft ein Druck auf die Taste F1 die Hilfe zum aktuell geöffneten Fenster auf.
[9] Concurrent Versions System, Anwendung zur Verwaltung verschiedener Programmversionen

installieren zu können (**Abb. II.11**). Für jede Version existieren jeweils zwei CVS-Module, einmal mit und einmal ohne Bilder.

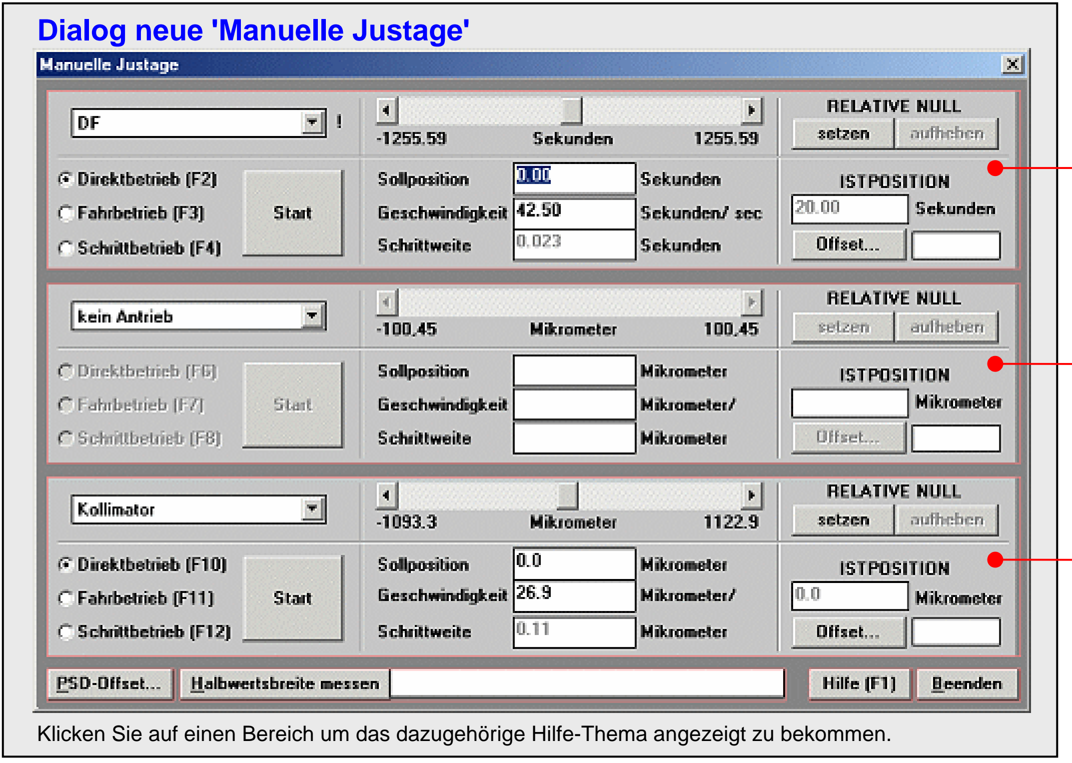
Abb. II.11 Entwicklung der ‚XCtl'-Hilfe

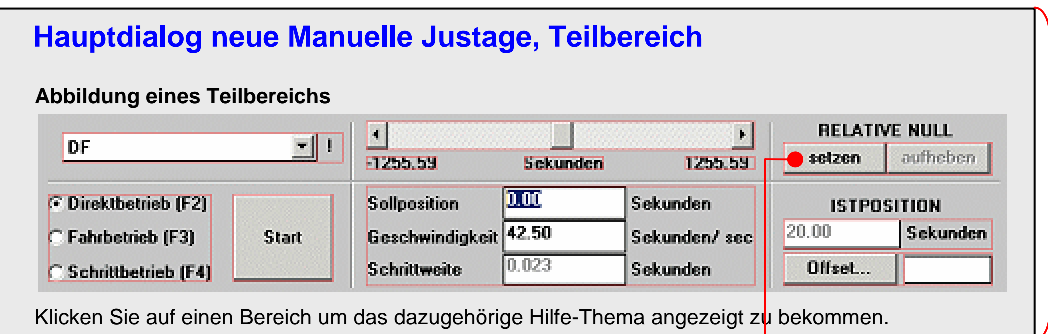
Anfangs wurde nach weiteren Möglichkeiten gesucht, um die einfache Erstellung und Wartung der Hilfetexte sowie ein komfortables Hilfe-System anzubieten, ohne dabei viel Festplattenspeicherplatz zu belegen. Da die Hilfe noch unter Microsoft Windows 3.1x nutzbar sein sollte, beschränkte sich das Angebot auf 16-Bit hlp-Dateien. Somit wurde HelpScribble als beste und komfortabelste Lösung weiter genutzt.

Problematisch blieben jedoch die Dateigrößen der bebilderten Versionen. Durch eine ausgiebige Recherche der Autoren zum Microsoft Hilfe-Compiler, bezüglich unterstützter Bildformate und Kompressionsmöglichkeiten, konnte die Gesamtgröße in mehreren Build-Test-Zyklen auf etwa 400 kB reduziert werden. Bei der höchsten Kompression für Microsoft Windows 3.1x reduzierte sich die Größe der bebilderten Hilfedateien auf etwa ½. Zudem ermöglichte der neueste Hilfe-Compiler die Verringerung der Farbtiefe von 24 auf 8 Bit für die Bilddateien und eine Dateigröße von ¼, ausgehend von der ursprünglichen Hilfeversion.

Durch diesen Erfolg konnten die einzelnen Dateien in einem Hilfeprojekt, d.h. einer hlp-Datei, zusammengefasst werden. Nur so waren ein universelles Inhaltsverzeichnis und die Verlinkung der einzelnen Hilfe-Themen untereinander möglich. Das Resultat dieser Arbeit zeigt **Abb. II.11**, Mitte.

Für die neue ‚Manuelle Justage' wurde nun ein wartungsfreundliches Konzept gesucht, das sowohl der Komplexität des Hauptdialogfensters als auch dem geringen Festplattenspeicherplatz auf den Arbeitsplätzen der Physik gerecht wird. Im Stil der vorhandenen Hilfe-Themen wurden pro Dialogfenster separate Seiten erstellt, die eine allgemeine Funktionsbeschreibung, eine Abbildung und die Tastenkombinationen enthält. Die Beschreibung der einzelnen Steuerelemente konnte nicht übersichtlich in diese Seite integriert werden. Daher wurden jeweils einzelne Themen erstellt, die mit dem entsprechenden Ausschnitt aus der Abbildung des Dialogfensters verknüpft wurden; d.h. der Klick auf ein Steuerelement ruft das entsprechende Hilfe-Thema auf (siehe **Abb. II.12**).





Hauptdialog neue Manuelle Justage, Relative Null - 'setzen'

Dies setzt die **Relative Null**, d.h die aktuelle Winkelposition (die Istposition) wird damit auf 0 gesetzt. Das Setzen bedeutet eine Darstellung aller Ausgaben in relativen Winkelpositionen, was durch OFFSET angezeigt wird. Die Minimal- und Maximalposition werden angepasst.
Dies wird zur leicheren Justierung einer Probe benutzt, indem z.B. eine häufig angefahrenen Winkelposition oder die Position von der aus eine Halbwertsbreitenmessung gestartet wird, als *relative Null* gesetzt wird. (siehe auch Offset)

Bedingungen
- 'kein Antrieb' darf nicht ausgewählt sein.
- Der Antrieb muss sich im Stillstand befinden.
- Die Halbwertsbreite darf nicht gemessen werden.

Hinweise
- Das aktuelle Offset für <Antrieb> wird neu berechnet.
- Wenn zuvor ein PSD-Kanaloffset definiert war, wird es durch diese Aktion aufgehoben; d.h. der zugeordnete Kanal ist wieder 0.

Abb. II.12 Verlinkung des Hauptdialogs mit den Hilfe-Seiten für die jeweiligen Steuerelemente am Beispiel der Funktion: Relative Null setzen; ein Rahmen entspricht einer Hilfe-Seite

Nach der oben vorgestellten Klassifikation handelt es sich demnach um eine passive, uniforme Hilfe mit dynamischen Ansätzen; teil-dynamisch, weil abhängig vom aktuell angezeigten Dialogfenster ein entsprechendes Hilfe-Thema angezeigt wird.

Am Anfang jedes Hilfe-Themas wird das behandelte Dialogfenster genannt (siehe **Abb. II.12**,). Bei der Beschreibung von Steuerelementen wird zudem die Benennung aus dem Kapitel ‚Benutzeroberfläche' des Pflichtenheftes verwendet. Wenn das Element unbeschriftet ist, wird eine abstrakte Aufgabenbeschreibung verwendet, die mit einer kursiven Schriftart hervorgehoben wird (siehe Beschreibung von **Abb. II.8**).

Die eigentliche Funktionsbeschreibung konnte leicht modifiziert aus dem gleichen Kapitel des Pflichtenheftes übernommen werden. Abstrakte Begriffe werden dabei in grüner Schrift und unterstrichen hervorgehoben (siehe **Abb. II.12**,) und sind zudem als Querverweis auf eine weiterführende Hilfe-Seite realisiert. Insgesamt entstanden so 38 Hilfe-Seiten, die sich mit dem Hauptdialog und den beiden Offset-Dialogen beschäftigen. Die Beschreibung ist insgesamt nicht so ausführlich wie im Referenzteil des Benutzerleitfadens (siehe **II.2.4**). Die Onlinehilfe verzichtet hier z.B. auf die Angabe von Wertebereichen. Es wird stichpunktartig erläutert, wann das Steuerelement bedienbar bzw. die Produktfunktion ausführbar ist (siehe **Abb. II.12**,). Dies entspricht genau den Bedingungen für die Produktfunktionen aus dem Funktionsteil des Pflichtenheftes. Abschließend wird unter ‚Hinweise' ggf. erläutert, welche Auswirkung die Produktfunktion auf die Nutzeroberfläche hat.

Die Integration der Benutzeroberfläche in das Pflichtenheft bei gleichzeitiger klarer Abgrenzung von Funktionen und Daten haben die Erstellung des Hilfe-Systems wesentlich erleichtert.

II.2.4 Benutzerhandbuch

„Obwohl integrierte Hilfe-Systeme heute bei Software-Produkten zum Standard-Funktionsumfang gehören, ist ein Benutzer-Handbuch dennoch ein unverzichtbarer Bestandteil eines guten Software-Produktes. Es ist – wie auch die Nutzeroberfläche – die Visitenkarte eines Software-Produktes. In vielen Fällen ist es die einzige Unterstützung, die der Benutzer für die Einarbeitung in das System erhält." (aus **[2]**, S. 640). Die Informationen aus Pflichtenheft, Produktmodell und Oberflächenprototyp gehen didaktisch-methodisch aufbereitet und unter Auswahl geeigneter Beispiele in das Benutzerhandbuch ein. Empfohlen wird die Erstellung eines solchen bereits in der Definitionsphase (siehe **[2]**, S. 100). Dadurch sind die Entwickler gezwungen sich so früh wie möglich in die Rolle des Anwenders zu versetzen.

Je nach Anwendergruppe werden diese in Form eines Trainings- (mit Kurs- oder Trainingsprogramm für Anfänger) oder Referenz-Handbuch (als Nachschlagewerk für Experten) erstellt. Das **Trainings-Handbuch** beschreibt die nötigen Schritte zum Erreichen von häufig auftretenden Arbeitszielen bei der Bedienung des Softwaresystems. Dies wird durch eine leicht nachvollziehbare Arbeitsanleitung erreicht. Damit ergibt sich eine **aufgabenorientierte Gliederung**, wobei die Aufgaben – sortiert nach Komplexität und mit der einfachsten Schrittfolge beginnend – strukturiert dargelegt sind.

Der Schwerpunkt des **Referenz-Handbuch**s liegt auf einer deutlichen Gliederung, um die benötigten Informationen schnell nachzuschlagen – eine **produktorientierte Gliederung**. Es werden alle Anzeigen und Produktfunktionen in der Reihenfolge beschrieben, wie sie sich aus der Struktur des Programmes ergeben. Im Gegensatz zum Trainings-Handbuch können und müssen alle Informationen im Referenz-Handbuch deutlicher gegliedert werden. Durch die Verwendung von Querverweisen können unbekannte Begriffe schnell nachgeschlagen werden.

Trainings- und Referenz-Handbuch können auch in einem **Benutzer-Leitfaden** kombiniert werden, wenn die Anwendergruppen in ihrem Wissen stark differenzieren. Wissen bezieht sich

hier sowohl auf den Umgang mit Softwaresystemen im Allgemeinen als auch auf das fachliche Wissen über die Aufgaben, die durch das Programm unterstützt werden.

Anwendungsfall: ‚Manuelle Justage'

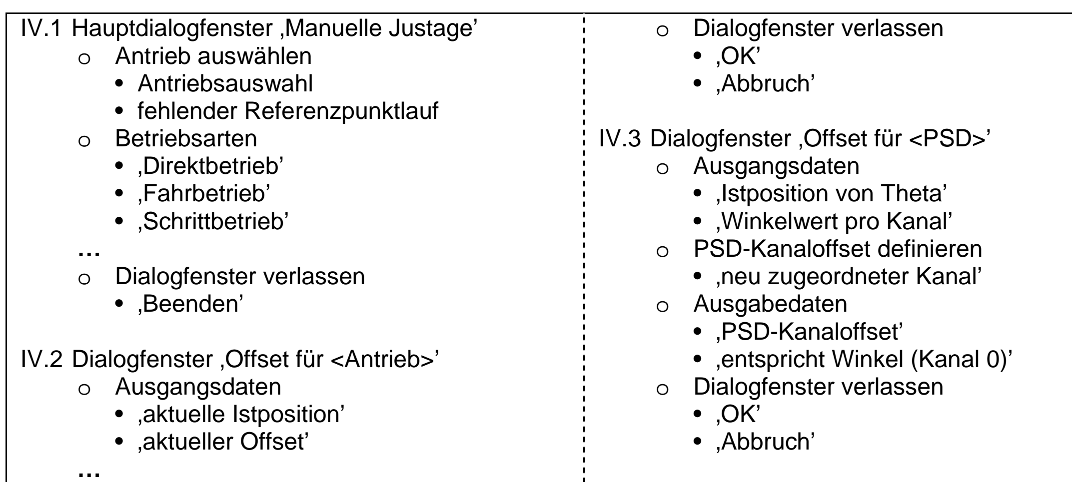
Weil die ‚Manuellen Justage' zum einen regelmäßig durch die Mitarbeiter des Physikalischen Institutes (als Experten) und zum anderen durch Studenten (einmalig im Rahmen ihres Praktikums) benutzt wird, musste ein Benutzer-Leitfaden erstellt werden mit einem Trainingsteil für die Studenten und einem Referenzteil für die Mitarbeiter des Physikalischen Institutes.

Begonnen wurde mit der Einleitung und einer kuren Einführung in den Gegenstandsbereich der ‚Manuellen Justage'. Anschließend wird eine Kapitelübersicht gefolgt von einer Produktstruktur gegeben. Um einen Überblick über die vom ‚XCtl'-Programm verwendeten Ressourcen zu erhalten, wird die angesteuerte Hardware zusammen mit den benutzten Einträgen der Konfigurationsdateien, den betreffenden Oberflächenfenstern und den Möglichkeiten zu deren Aufruf stichpunktartig in einer Abbildung zusammengefasst.

Die Dialogfenster der neuen ‚Manuellen Justage' sind ausklappbar abgebildet, um beim Studium der Referenz- und Trainingsteile verfügbar zu sein. Dann folgt eine allgemeine Nomenklatur für Steuerelemente, wo deren Funktion und Bedienung bebildert erläutert werden.

Entgegen [2] (S. 647f) folgt nun der Referenz- vor dem Trainingsteil. So sind die in den Schrittfolgen des Trainingsteils verwendeten Produktfunktionen bereits aus dem Referenzteil bekannt. Abgesehen von internen Produktfunktionen – wie das Betreten eines Dialogfensters – (die nur im Pflichtenheft beschrieben werden müssen), konnte der Inhalt aus dem Kapitel ‚Benutzeroberfläche' des Pflichtenheftes übernommen werden. Dementsprechend ähnelt die Gliederung des Referenzteils wieder der des Pflichtenheftes (vgl. **Abb. II.8**).

Um das Nachschlagen zu beschleunigen, ist der Referenzteil jedoch feiner gegliedert. Jedes Steuerelement (Bezeichnung in Hochkommata) wird dazu im Inhaltsverzeichnis aufgeführt.



IV.1 Hauptdialogfenster ‚Manuelle Justage'
- o Antrieb auswählen
  - • Antriebsauswahl
  - • fehlender Referenzpunktlauf
- o Betriebsarten
  - • ‚Direktbetrieb'
  - • ‚Fahrbetrieb'
  - • ‚Schrittbetrieb'
- ...
- o Dialogfenster verlassen
  - • ‚Beenden'

IV.2 Dialogfenster ‚Offset für <Antrieb>'
- o Ausgangsdaten
  - • ‚aktuelle Istposition'
  - • ‚aktueller Offset'
- ...
- o Dialogfenster verlassen
  - • ‚OK'
  - • ‚Abbruch'

IV.3 Dialogfenster ‚Offset für <PSD>'
- o Ausgangsdaten
  - • ‚Istposition von Theta'
  - • ‚Winkelwert pro Kanal'
- o PSD-Kanaloffset definieren
  - • ‚neu zugeordneter Kanal'
- o Ausgabedaten
  - • ‚PSD-Kanaloffset'
  - • ‚entspricht Winkel (Kanal 0)'
- o Dialogfenster verlassen
  - • ‚OK'
  - • ‚Abbruch'

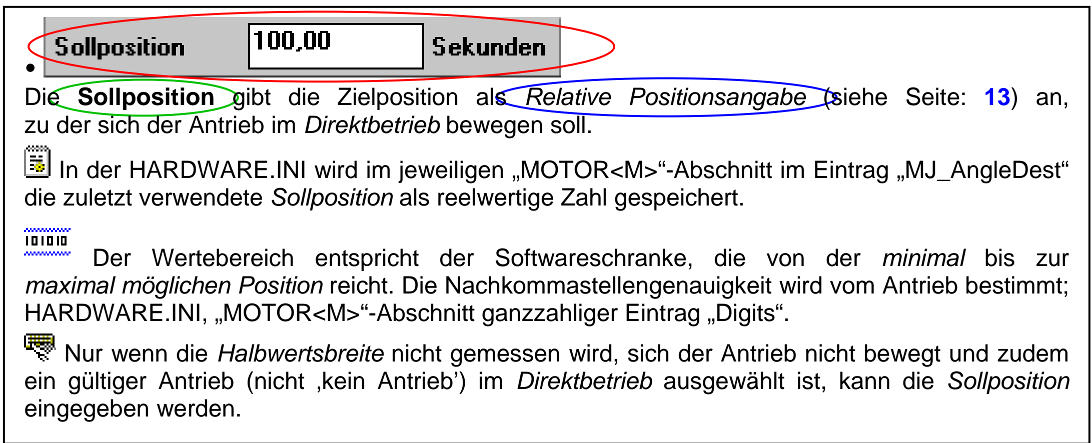
Abb. II.13 gekürzter Auszug aus dem Benutzer-Leitfaden [M17]; Kapitel ‚Referenzteil'

Die produktorientierte Gliederung erkennt man z.B. an den Unterpunkten von IV.3 (**Abb. II.13**), wo zuerst die gegebenen Ausgangsdaten, dann die möglichen Anwendereingaben und abschließend die daraus resultierenden Ausgabedaten dargestellt werden.

Vor der eigentlichen Funktionsbeschreibung ist das jeweilige Steuerelement (siehe **Abb. II.14**) abgebildet. So sind auch Steuerelemente nachzuschlagen, die im Dialogfenster nicht benannt sind (abstrakte Aufgabenbeschreibung). Nach der ausführlichen Beschreibung der Produktfunktion folgen drei Abschnitte mit Informationen aus dem Pflichtenheft-Kapitel ‚Daten':

- Im Absatz (, siehe **Abb. II.14**) werden kurz die benötigten Einträge aus den Konfigurationsdateien genannt. Ausführliche Informationen sind im Anhang nachzuschlagen.
- Die möglichen Anwendereingaben für freigegebene Eingabefelder sind im Absatz () beschrieben. Diese Angaben, z.B. Wertebereich und Nachkommastellengenauigkeit, sind besonders wichtig, weil sie dem Anwender im Dialogfenster nur indirekt[10] angezeigt werden. Diese Informationen wurden zusammen mit den Angaben über die Konfigurationsdateien aus dem Kapitel ‚Daten' des Pflichtenheftes entnommen.
- Im Absatz () werden die Bedingungen erläutert, unter denen die Produktfunktion benutzt werden kann. Nicht verfügbare Steuerelemente werden im aktuellen Programmzustand gesperrt und ausgegraut (siehe **II.2.2**, Erleichterung der Bedienung des komplexen Hauptdialogs). Diese Informationen entstammen dem Kapitel ‚Benutzeroberfläche' des Pflichtenheftes, Abschnitt ‚Bedingung' des jeweiligen Steuerelementes.

Wie im Pflichtenheft gibt es blau hervorgehobene Querverweise und Einträge im Stichwortverzeichnis (). Benutzte und im Stichwortverzeichnis nachzuschlagende Fachbegriffe sind wieder kursiv gekennzeichnet ().



Sollposition 100,00 Sekunden

Die **Sollposition** gibt die Zielposition als *Relative Positionsangabe* (siehe Seite: 13) an, zu der sich der Antrieb im *Direktbetrieb* bewegen soll.

In der HARDWARE.INI wird im jeweiligen „MOTOR<M>"-Abschnitt im Eintrag „MJ_AngleDest" die zuletzt verwendete *Sollposition* als reelwertige Zahl gespeichert.

Der Wertebereich entspricht der Softwareschranke, die von der *minimal* bis zur *maximal möglichen Position* reicht. Die Nachkommastellengenauigkeit wird vom Antrieb bestimmt; HARDWARE.INI, „MOTOR<M>"-Abschnitt ganzzahliger Eintrag „Digits".

Nur wenn die *Halbwertsbreite* nicht gemessen wird, sich der Antrieb nicht bewegt und zudem ein gültiger Antrieb (nicht ‚kein Antrieb') im *Direktbetrieb* ausgewählt ist, kann die *Sollposition* eingegeben werden.

Abb. II.14 Auszug aus dem Referenzteil des Benutzer-Leitfadens **[M17]**

Die im Trainingsteil behandelten Arbeitsabläufe wurden als Flussdiagramme umgesetzt. Um häufig auftretende Redundanz zu vermeiden, wurden mehrfach benötigte Schrittfolgen in gesonderte Flussdiagramme ausgelagert. Auf diese wurde dann nur verwiesen (siehe **Abb. II.15**). Durch die Aufteilung der Komplexität wurde auch die Lesbarkeit verbessert.

Bei der Formulierung der durchzuführenden Schritte wird der Leser direkt angesprochen (siehe **Abb. II.15**) und so zum Nachvollziehen angeregt. Mögliche Entscheidungssituationen sind als Fragen formuliert und binär als Ja-Nein-Zweige realisiert. Genauso wurden Problemsituationen umgesetzt, wenn z.B. die Ausführung einer Produktfunktion im aktuellen Zustand nicht möglich ist. Dann ist bei mindestens einem Entscheidungszweig – i.d.R. dem Problemfall – dargelegt, an welchen Merkmalen dieser im Dialogfenster zu erkennen ist (siehe **Abb. II.15**).

Im Hinblick auf gänzlich unerfahrene Anwender wurden die zu bedienenden Steuerelemente bei jedem Arbeitsschritt abgebildet. Besonders zu beachtende Beschriftungen werden zudem durch ein dickes, rotes Oval umrahmt.

[10] bei der Eingabe eines unzulässigen Wertes wird dieser automatisch in den Wertebereich eingepasst – siehe , Kapitel ‚Fehlertoleranz der Benutzeroberfläche'

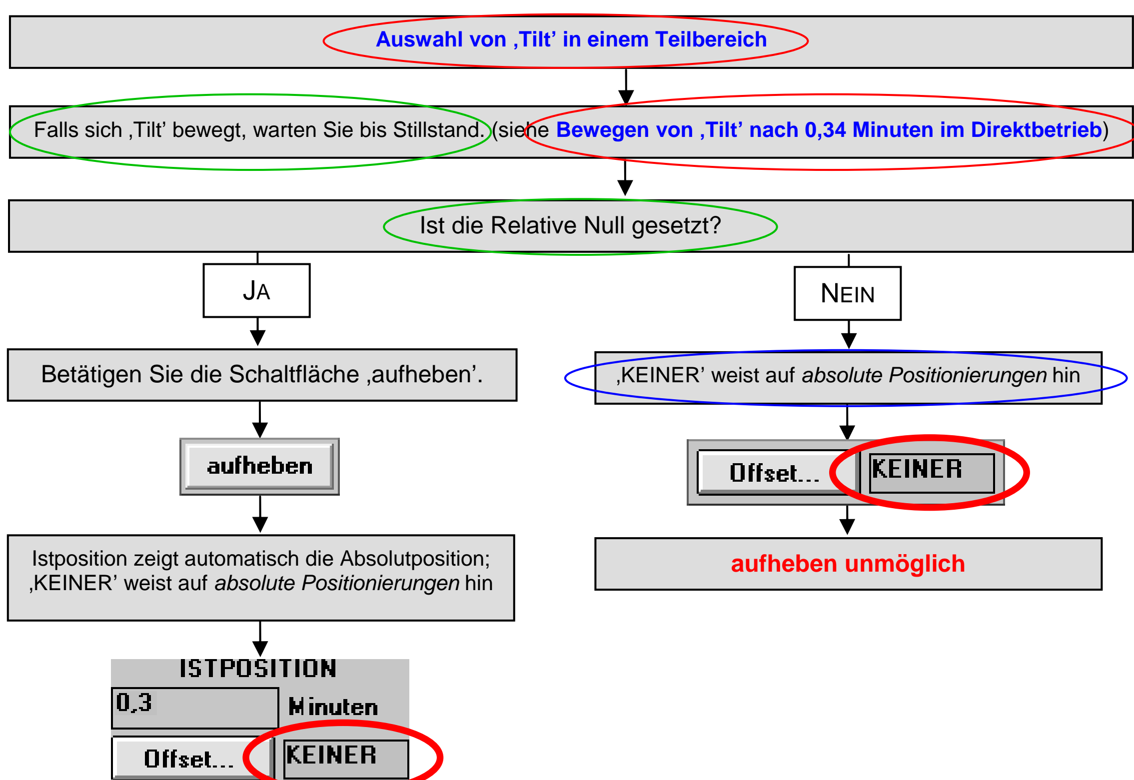


Abb. II.15 Auszug aus dem Trainingsteil des Benutzer-Leitfadens **[M17]**, zur Schrittfolge für „Relative Null von ‚Tilt' aufheben"

Im Anhang werden die benutzten Einträge aus den Konfigurationsdateien gesammelt erläutert. Dazu zählen auch die Nennung des Datentyps (logisch, aufgezählt, ganzzahlig, reellwertig oder Zeichenkette) und die Angabe gültiger Werte (z.B. Wertebereich und Nachkommastellengenauigkeit bei reellwertigen Einträgen).

Das ‚XCtl'-Programm ist auch als englischsprachige Version auf den Rechnern des Physikalischen Institutes verfügbar. Im Leitfaden gibt es dazu ein Kapitel englischsprachige Nutzeroberfläche, in dem die englischen Dialogfenster ebenfalls in einer ausklappbaren Übersicht dargestellt sind. Zusätzlich sind für jedes abgebildete Steuerelement die deutschen und englischen Beschriftungen in einer Tabelle gegenübergestellt.

Abschließend folgen Literatur- und Stichwortverzeichnis und eine Liste der enthaltenen Tabellen und Abbildungen.

Wie beim Hilfe-System wurde die Erstellung des Benutzerleitfadens durch die klare Strukturierung des Pflichtenheftes, besonders des Kapitels ‚Benutzeroberfläche', wesentlich erleichtert. Vor allem die Konsistenz zum Pflichtenheft und damit zum Programm ist sehr hoch, weil Bedingungen, Hinweise für jede der Produktfunktionen und jedes Steuerelement explizit formuliert sind. Auch ihre Beziehungen untereinander ließen sich gut aus dem Pflichtenheft extrahieren, so dass die Erstellung der Abläufe im Trainingsteil problemlos und einfach möglich war.

II.2.5 Objektorientierte Analyse

Die objektorientierten Grundkonzepte bilden zusammen mit den **Beziehungstypen**[11] die **objektorientierte Analyse** (siehe [2], S. 251f), welche die fachliche Lösung eines zu realisierenden Systems repräsentiert. Da diese Phase die Software-Architektur noch nicht berücksichtigt, beschäftigt sie sich, im Sinne der Dekomposition, nur mit der Modellierung der Funktionskomponente. Spätestens in diesem Abschnitt der Softwareentwicklung empfiehlt es sich große Projekte in mehrere Subsysteme (zu Paketen siehe [2], S. 205) zu zerlegen[12]. Dadurch kann pro Subsystem eine Funktionskomponente als Klasse erstellt werden, wobei der Klassenbezeichner den Charakter des Subsystems widerspiegeln sollte. Der wichtigste Teil der objektorientierten Analyse ist der Entwurf einer öffentlichen Schnittstelle für Funktionskomponenten. Um die Komplexität übersichtlich zu gestalten und gleichzeitig eine hohe Flexibilität/ Robustheit dieser Klasse zu garantieren, wird ein von den Autoren zusammengestelltes Muster vorgestellt. Dies beinhaltet im Wesentlichen eine Klassierung der Methodenbezeichner durch Präfixe:

- Nach [11] (S. 95ff) soll der direkte Zugriff auf Datenelemente über eine öffentliche Schnittstelle vermieden werden. Deshalb bedient man sich so genannter **Accessor**- und **Mutator-Methoden** zum Lesen und Schreiben eines gleichnamigen Attributes. Allgemein üblich ist die Benennung als `Get`-/ `Set`-Methoden, z.B. `GetValue()` und `SetValue(…)` für das Attribut `m_Value`.
- `Do`- Methoden initiieren die Steuerung der Programmfunktionalität oder Berechnungen, z.B. `DoDivide(…,…)`.
- Wenn die Ausführung von `Set`-/ `Do`- Methoden an Bedingungen geknüpft ist, können diese in `Can`-Methoden ausgelagert werden, z.B. `CanDoDivide(…,…)` und `CanSetValue(…)`. Die `CanDo`- und `CanSet`-Methoden sollen später das Ergebnis der Bedingung zurückgeben. So kann die Nutzeroberfläche den Zustand der Funktionskomponente erfragen und sich daran anpassen, z.B. durch Sperren und Ausgrauen eines Steuerelementes, wenn die Funktion nicht verfügbar ist.
- `On`-Methoden werden als Ergebnis von Zustandsänderungen aufgerufen und ermöglichen so die Reaktion auf solche Ereignisse. Zur Aktualisierung der Nutzeroberfläche könnte `OnDividing()` z.B. anzeigen, dass eine "langwierige" Berechnung von `DoDivide` gestartet wurde.

<u>Anwendungsfall: ‚Manuelle Justage'</u>

a) Gewinnung der Attribute

Für jeden Datenteil-Eintrag / **D** `<dd>` / entstand genau ein Attribut. So wurde beispielsweise aus **/ D 110 /** `m_AngleDest` abgeleitet. Weil dieses Attribut für jeden Antrieb zu verwaltet ist, wurde es mit allen anderen antriebsspezifischen Daten in der Struktur `TMotorData` zusammengefasst. Die Größe der Liste dieser Struktur hängt von der Anzahl der angeschlossenen Antriebe ab.

---

[11] Assoziationen, Kompositionen und Aggregation

[12] Nach Kapitel empfehlen die Autoren diese Modularisierung bereits bei der Erstellung des Pflichtenhefts durchzuführen.

/ D 110 / – „eingegebene Sollposition“
Die Sollposition ist – wie die Istposition – abhängig vom aktuellen Offset des Antriebs. Die Minimal- und Maximalwerte können dem Datenteil entnommen werden; ebenso die Nachkommastellengenauigkeit. Die Sollposition ist bei jeder Änderung zu speichern!

Bezeichnung	eingegebene Sollposition
Typ / Zugriff / Eintrag	R / lesen und schreiben / NEU
Speichertyp	als Eintrag in einer ini-Datei (ASCII-Text)
Speicherort	**HARDWARE.INI –> [MOTOR<M>] –> MJ_AngleDest**
Minimalwert	siehe **Tabelle 8**
Maximalwert	siehe **Tabelle 9**
Standardwert	aktuelle Istposition
Nachkommastellen	siehe **Tabelle 10**

Tabelle 15 Eigenschaften der Sollposition

Abb. II.16 Auszug aus dem Datenteil des Pflichtenheftes **[M15]**

b) Accessor- und Mutator-Methoden

Für jedes Attribut muss eine `Get`-Methode zum Auslesen erstellt werden, deren Rückgabewert später dem des Attributes entspricht. Aus der Zelle ‚Zugriff' im Datenteil des Pflichtenheftes kann abgeleitet werden, ob eine `Set`-Methode zum Setzen benötigt wird. Der aktuelle Inhalt des Attributes sollte, wenn möglich, von der `Set`-Methode zurückgegeben werden[13], weil der zu setzende Wert in ein Intervall eingepasst werden könnte oder weil das Setzen im aktuellen Zustand unzulässig war (der Inhalt des Attributes bleibt unverändert). Finden sich im Funktionsteil des Pflichtenheftes Bedingungen für das Setzen des Attributes, ist eine Methode mit dem Präfix `CanSet` zu erstellen. Die `Set`-Methode sollte dann auch das Ergebnis des `Can`-Methodenaufrufs zurückgeben, um zu zeigen, ob das Attribut gesetzt wurde.

Zu dem bei **a)** abgeleiteten Attribut `m_AngleDest` müssen laut **Abb. II.16** (‚Zugriff') zwei Methoden – zum Lesen und Setzen – entworfen werden. Diese Methoden heißen `GetAngleDest()`und `SetAngleDest()`. Dazu ist eine Methode `CanSetAngleDest()` notwendig, weil im Funktionsteil des Pflichtenheftes, eine Bedingung zum Setzen der Sollposition aufgeführt ist (siehe **Abb. II.17**).

Grundlagen
Die **Sollposition** ist nur im *Direktbetrieb* erforderlich. Dort gibt sie an, wohin sich der Antrieb bewegen soll. Die Sollposition besteht, wie die *Istposition* auch, aus der Absolutposition des Antriebs verschoben um einen *Offset*. Damit ist die Sollposition auch abhängig von dem *Offset für <Antrieb>* und der *Relativen Null*, weil diese intern über das *Offset* abgebildet werden.
Bedingung
Der Antrieb darf sich nicht bewegen und der *Direktbetrieb* muss ausgewählt sein.

Abb. II.17 Auszug aus dem Funktionsteil des Pflichtenheftes **[M15]**

c) Methoden zur Realisierung der Funktionalität

Jeder **/ F** `<ff>` /-Eintrag im Funktionsteil des Pflichtenheftes ist eine potentielle `Do`-Methode. Da dort jedoch die gesamte Programmfunktionalität beschrieben wird, muss sich nicht jeder Eintrag auf die Funktionskomponente beziehen. Es könnte sich auch um allgemeine Anforderungen an die Benutzeroberfläche handeln. Andererseits können mehrere Einträge in einer parametrisierten `Do`-Methode zusammengefasst werden. Die Signatur ist stets abhängig von der verbalen Spezifikation im Pflichtenheft und kann deshalb nicht allgemeingültig daraus konkretisiert werden.

Wenn die Ausführung solch einer Funktion an Bedingungen geknüpft ist, sollte eine `CanDo`-Methode erstellt werden. Diese kennzeichnet, ob die Funktion durchgeführt werden

[13] Eigentlich ist die Angabe der Methodensignatur in der OOA nicht üblich. Ein Teil ist hier jedoch bereits kanonisch im Pflichtheft ablesbar.

darf. Der Rückgabewert der `Do`-Methode sollte dann das Ergebnis des `Can`-Methodenaufrufs sein.

Die in **Abb. II.18** beschriebenen Funktionen weisen z.B. eine verwandte Funktionalität auf, so dass sie zu der Methode `DoDrive( EDirection )` zusammengefasst wurden. Dieser intuitiv notwendige Parameter zur Regelung der Bewegungsrichtung sollte daher schon hier berücksichtigt werden.

Grundlagen
Im **Fahrbetrieb** kann der Antrieb kontinuierlich (mit einer vorgegebenen *Bewegungsgeschwindigkeit*), entweder vorwärts oder rückwärts, bewegt werden.

/ F 90 / – „Vorwärtsbewegung im Fahrbetrieb“
Nach der Beschleunigungsphase bewegt sich der Antrieb, solange der Reiz – der zum Starten der Bewegung geführt hat – anliegt, kontinuierlich mit der angegebenen Geschwindigkeit vorwärts. Wenn die maximale *Istposition* erreicht wird, stoppt der Antrieb automatisch.

Bedingung
Der Antrieb darf sich nicht bewegen und die aktuelle *Istposition* muss kleiner der maximalen *Istposition* sein, damit die Bewegung startet.

/ F 100 / – „Rückwärtsbewegung im Fahrbetrieb“
Die Steuerung des Antriebs ist äquivalent zu **/ F 90 /**, die Bewegung erfolgt jedoch in entgegengesetzter Richtung, also rückwärts.

Bedingung
Der Antrieb darf sich nicht bewegen und die aktuelle Istposition muss größer der minimalen Istposition sein; damit die Bewegung startet.

Abb. II.18 Auszug aus dem Funktionsteil des Pflichtenheftes **[M15]**

Da sich die OOA eigentlich nicht auf die Präzisierung des Datentyps, der Parameterlisten und Zugriffsrechte konzentriert, kann nur ein geringer Teil der Metriken ausgewertet werden (siehe **Tab. II.3**). Im Mittelpunkt der neuen ‚Manuellen Justage’ steht die spätere Funktionskomponente in Form der Klasse `TManJustage`. Sie ist durch die oben vorgestellte Schrittfolge entstanden und repräsentiert die Produktfunktionen. Die Datenverwaltung wurde auf die beiden Hilfs-Strukturen `TMotorData` und `TPsdData` ausgelagert, die nach außen nicht sichtbar sind.

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
`TManJustage`	–	2	49	1	0	–	–	–	–	–	3	–
↳ `TMotorData`	–	15	0	0	0	–	–	–	–	–	1	–
↳ `TPsdData`	–	4	0	0	0	–	–	–	–	–	1	–

Tab. II.3 Ausgewählte Metriken für die neue ‚Manuelle Justage’
(nach der Analyse- und Definitionsphase)

Das entstandene OOA-Modell der neuen ‚Manuellen Justage’ zeigt **Abb. II.22**.

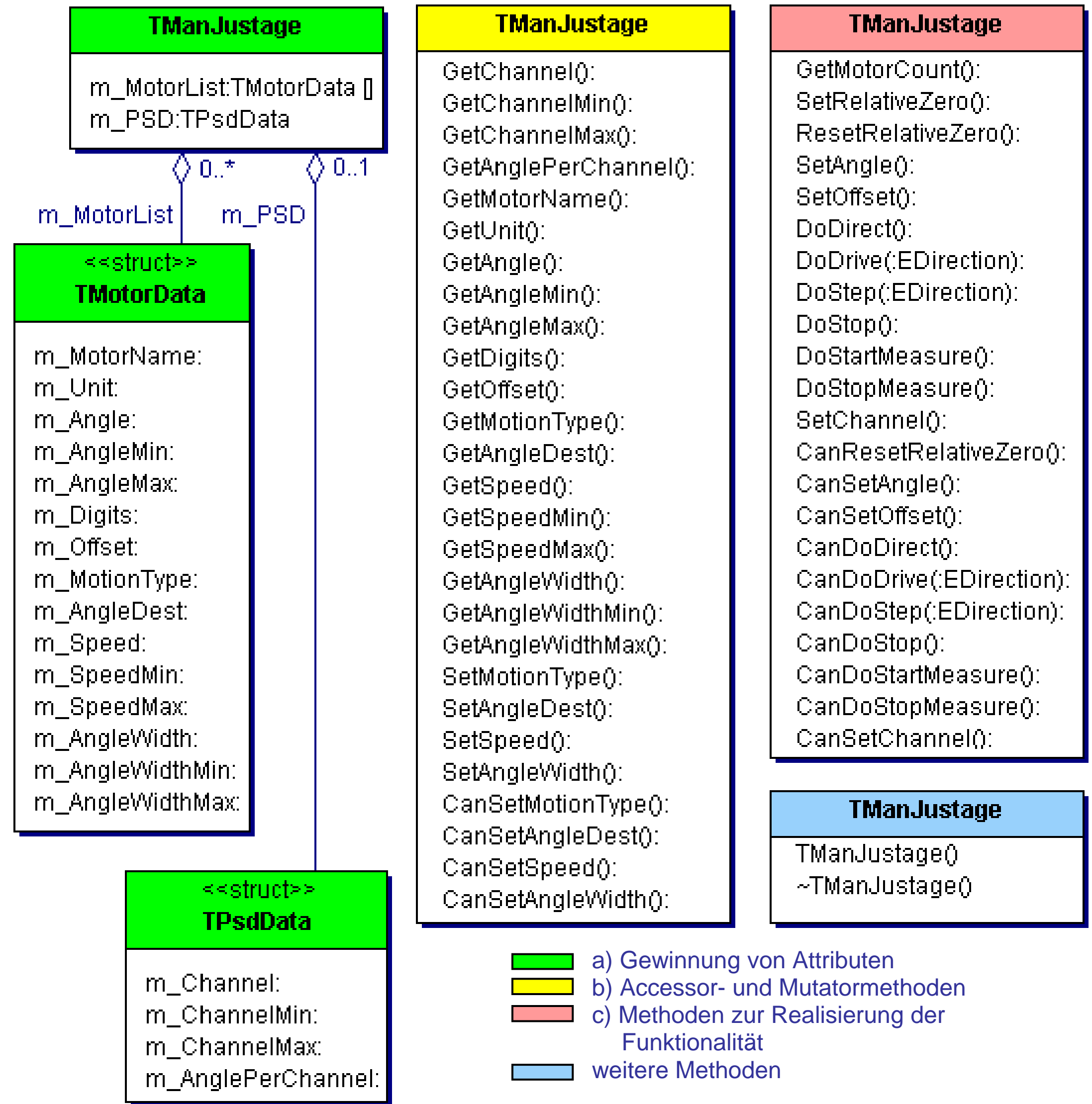


Abb. II.19 UML-Klassendiagramm für das OOA-Modell der neuen ‚Manuellen Justage', gegliedert nach der Erstellungsreihenfolge ihrer Elemente

II.2.6 Zusammenfassung

Die Dekomposition konnte in der Analyse- und Definitionsphase nur beim Pflichtenheft angewendet werden. Dazu wurden dieses nachträglich um den Oberflächenentwurf ergänzt (siehe **Abb. II.20**, linker Handlungsstrang).

Die Mitarbeiter des Physikalischen Institutes bewerteten das entstandene Pflichtenheft trotzdem als zu abstrakt, weil die Nutzeroberfläche getrennt von den eigentlichen Produktanforderungen beschrieben wurde. Da die Oberflächenentwürfe in der Praxis jedoch erst nach dem Pflichtenheft entstehen und nicht (wie von den Autoren praktiziert) nachgetragen werden, wären dort normalerweise gar keine Oberflächenentwürfe enthalten! Nach [2] (Abb. 2.1-2) beinhalten Pflichtenhefte nur die verbalen Produktanforderungen. Dies hätte aber den Abstraktionsgrad noch weiter gesteigert und das Verständnis weiter erschwert. Wahrscheinlich hatten die Mitarbeiter des Physikalischen Institutes meist nur mit den zahlreichen Verhaltenspezifikationen für das ‚XCtl'-Projektes gearbeitet. Dort werden die jeweiligen Subsysteme exemplarisch an einer ausgewählten Nutzeroberfläche erläutern. Nach Meinung der Autoren kann dies die abstrakte – weil rein verbale – Produktbeschreibung im Pflichtenheft für den Anwender bzw. Auftraggeber wesentlich verständlicher machen!

Nach **[2]** (S. 99) sollen die Prototypen nach der OOA erstellt werden (siehe **Abb. II.20**, rechte Vorgehensweise), um die verbalen Ausführungen des Pflichtenheftes zunächst stärker zu präzisieren. Um aber die Realisierung des dekomponierten Pflichtenheftes zu beschleunigen, wurden die Oberflächen-Prototypen vor dem OOA-Modell erstellt. Das ist möglich, weil OOA und Oberflächen-Prototyping weitgehend unabhängig voneinander sind.

Bei der Abstimmung des Pflichtenheftes können Inkonsistenzen, Redundanzen und Fehler frühzeitig in der Spezifikation aufgedeckt und Missverständnisse zwischen den Parteien geklärt werden. Dabei müssen nach dem linken Handlungsstrang nur Pflichtenheft und Oberflächenentwurf angepasst werden, nicht das OOA-Modell (**Abb. II.20**, vgl. dazu Beziehungen zwischen Pflichtenheft, Oberflächen und OOA zwischen der rechten und linken Seite).

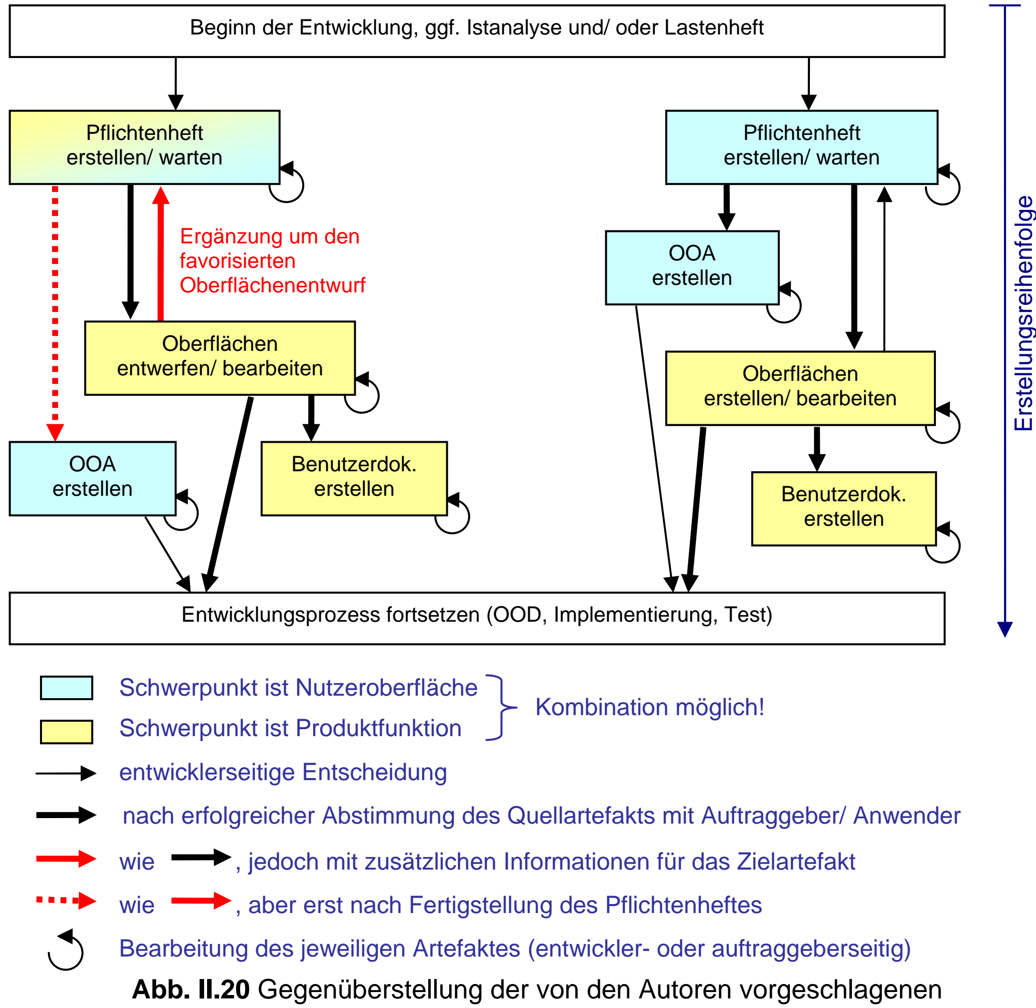


Abb. II.20 Gegenüberstellung der von den Autoren vorgeschlagenen Definitionsphase (links) zur Allgemeinen (rechts)

Damit wird das Oberflächen-Prototyping jedoch zu einem unerlässlicher Bestandteil der Definitionsphase, weil es zur Vervollständigung des Pflichtheftes benötigt wird! Heutzutage wird aus ökonomischen Gründen leider häufig auf das Prototyping verzichtet, obwohl neben den oben genannten Vorteilen auch der Bedienkomfort für den späteren Anwender steigt, da er auf eine unvorteilhafte oder komplizierte Oberflächengestaltung früh einwirken kann. Zudem kann es die Notwendigkeit zusätzlicher Produktfunktionen sichtbar machen. Im Falle der neuen ‚Manuellen Justage' waren dies die Offsets für die Antriebe und den PSD.

Auch wenn es zunächst zeitaufwendiger ist die abgestimmte Oberfläche im Pflichtenheft nachzutragen, so verbessert es doch deutlich die Vollständigkeit der Produktanforderungen. Bei

der neuen ‚Manuellen Justage' wurden so mehrere oberflächenspezifische Funktionen und ein zusätzliches Datenelement ermittelt und im Pflichtenheft erfasst.

Durch die starke Formalisierung, Gliederung und Informationsauswahl des nach **II.2.1** erstellten Pflichtenheftes lassen sich für die nachfolgenden Entwicklungsschritte eindeutigere und umfangreichere Informationen gewinnen. Des Weiteren kann die Integration der Oberflächenentwürfe und deren Beschreibung unvorteil- oder sogar fehlerhaftes Verhalten der Nutzeroberfläche schon in der Planungsphase eliminieren. Dies ist sonst erst beim Abnahmetest durch den Anwender möglich. Neben den Wartungsarbeiten am Programmcode müssten dann natürlich auch die Dokumente (bspw. Benutzerhandbuch, Online-Hilfe) überarbeitet werden.

Das Pflichtenheft der neuen ‚Manuellen Justage' hat die Erstellung von Benutzerdokumentation und Hilfe-System extrem vereinfacht, weil wesentliche Teile aus dem Kapitel ‚Nutzeroberfläche' des Pflichtenheftes, leicht abgewandelt, übernommen werden konnten. Das Pflichtenheft bildete die einzige Grundlage für diese Artefakte. Obwohl Hilfe-System und Benutzerleitfaden im Anwendungsfall der neuen ‚Manuelle Justage' nacheinander erstellt wurden, steht einer parallelen Bearbeitung nichts im Wege.

Für sich genommen haben die oberflächenspezifischen Elemente des Pflichtenheftes keinen Nutzen für die OOA. In seiner Gesamtheit hat es die Vollständigkeit und Korrektheit des OOA-Modells verbessert, weil Missverständnisse in der Spezifikation schnell aufgefunden und beseitigt wurden.

II.3 Designphase

In der Designphase ist der Architektur- oder Systementwurf durchzuführen (siehe [2], S. 986). Hierbei wird die Softwarearchitektur festgelegt. Sie ist wesentlich von der Art der Anwendung und der verwendeten Plattform abhängig. Ausgehend vom OOA-Modell der Definitionsphase entsteht durch Verfeinerung und Anbindung der Nutzeroberfläche das OOD-Modell.

Bei der Dekomposition spielt der Verbindung zwischen Nutzeroberfläche und Funktionskomponente eine wichtige Rolle. Dazu werden zwei Kommunikationsmuster, das Observer-Muster (**II.3.3–3**) und die Polling-Variante (**II.3.3–4**) parallel vorgestellt. Es erfolgt zudem eine allgemeine Bewertung. In Kapitel **II.3.5–1** wird abschließend auf die Erfahrungen bei der Umsetzung von Observer-Muster und Polling-Variante eingegangen.

II.3.1 Verfeinerung der Funktionskomponente

Die in **II.2.5** definierte Funktionskomponente muss nun weiter verfeinert werden (z.B. nach [2], S. 992). Dies beinhaltet u.a. den Zugriffsschutz von Attributen und Methoden und deren Präzisierung.

- Wenn die Programmiersprache dies ermöglicht, sollten **nicht-zustandsverändernde Methoden**, d.h. der Objektzustand bleibt unverändert, gekennzeichnet werden. In C++ const-member-functions (siehe [16]).
- Alle entstehenden Parameter sollten, wenn möglich, als konstant gekennzeichnet werden, um ihre Unveränderbarkeit sicherzustellen In C++ durch type specifier `const` für Parameter (siehe [16]).
- Zusätzlich zu den im OOA-Modell definierten Präfixen sollten nicht-zustandsverändernde Methoden – die durch die Festlegung der Signatur ausschließlich einen Wahrheitswert zurückgeben – mit den Präfixen `Is` oder `Has` gekennzeichnet werden. Je nachdem, was dem englischen Sprachgebrauch entspricht, z.B. `IsMoving` aber `HasPermition`.
- Da die Ausführung der entsprechenden `Do`- oder `Set`-Methode direkt an diese Bedingung geknüpft ist, sollten alle `CanDo`- und `CanSet`-Methoden das Ergebnis ihrer Bedingung als Wahrheitswert, d.h. per Rückgabewert, zurückgeben. So ist leicht feststellbar, ob die Funktion vollständig ausgeführt wurde.

Anwendungsfall: ‚Manuelle Justage'

Weil die neue ‚Manuelle Justage' prinzipiell die gleiche Funktionalität wie die ursprüngliche anbieten soll (Antriebsteuerung und Halbwertsbreitenmessung), wurde dort ein Programmcode-Review durchgeführt, das in **[M24]** dokumentiert wurde. Dabei wurden Beziehungen zu den Subsystemen des ‚XCtl'-Projektes herausgearbeitet, die bereits Teile dieser Funktionalität anbieten. Da im Repository des Projektes diese Subsysteme nicht dokumentiert waren, mussten sie einem Reverse Engineering unterzogen werden. Als Resultat entstanden die Designdokumente **[M26]** (Motorsteuerung) und **[M4]** (Ablaufsteuerung), einschließlich der Fehler, die während dieser Analyse gefunden wurden. Einzige Ausnahme war das Subsystem Detektornutzung, für das im Rahmen einer Studienarbeit bereits das Reverse Engineering Dokument **[M12]** erstellt wurde.

Um die fehlerfreie Anbindung an die Manuelle Justage zu gewährleisten, wurden die Subsysteme anschließend einem Reengineering unterzogen. Schwerpunkt war die Verbesserung

des Zugriffsschutzes für Attribute und die damit verbundene Erstellung und Verwendung entsprechender Accessor- und Mutator-Methoden. Oft verwendete `friend`-Relationen wurden so überflüssig. Außerdem erfolgte eine Neustrukturierung, im Wesentlichen eine Zusammenfassung nach Sinneinheiten, und eine ausführlichere Kommentierung des Programmcodes. Der bereits in **[M3]** identifizierte tote Programmcode wurde, mit Datum gekennzeichnet, auskommentiert. Die im Repository aufgeführte Fehler-Dokumentation der Subsysteme Motor- (**[M27]**), Ablaufsteuerung (**[M5]**) und ursprüngliche ‚Manuelle Justage' (**[M19]**) wurde aktualisiert, indem neue Fehler hinzugefügt und von den Autoren korrigierte gekennzeichnet wurden.

Um die neuen Systemzustände festzuhalten, erfolgte eine abschließende Dokumentation in den Designdokumenten **[M28]** und **[M6]**. Dort wurden schließlich die Methoden identifiziert, die in der neuen ‚Manuellen Justage' benutzt werden sollten. Zudem konnten einige Datenteil-Einträge im Pflichtenheft vervollständigt werden. Dies betraf die Angaben: ‚Typ', ‚Minimalwert', ‚Maximalwert', ‚Nachkommastellengenauigkeit' (nur bei reellwertigen Werten) und ‚Maximallänge' (nur Zeichenketten). Bei neuen Projekten besteht diese Möglichkeit nicht, weil weder Programmcode noch Designdokumente vorhanden sind. Alle Informationen sind in der Definitionsphase beim Benutzer zu erfragen. In jedem Fall sind nun die Attribute der Funktionskomponente zu typisieren und die Signaturen der Methoden zu vervollständigen.

a) Spezialisierung der Attribute

In den Designdokumenten konnten die in den anderen Subsystemen verwalteten Daten abgelesen werden. In Folge dessen wurden einige Attribute aus dem OOA-Modell entfernt, z.B. `m_Angle` für die Istposition. Der Typ für alle übrigen Attribute musste nicht aus dem Produktmodell abgeleitet werden. Aufgrund der Programmcodenähe der Reengineering-Dokumente war dieser dort direkt ablesbar. So wurde z.B. das Attribut `m_AngleDest` zu `double m_AngleDest` präzisiert.

Durch die im Pflichtenheft aufgeführten Beziehungen zwischen Istposition (/ **D 50** /), Sollposition (/ **D 110** /), Schrittweite (/ **D 150** /) und Offset (/ **D 90** /) wurde der gemeinsame Typ `typedef double TAngle` für alle Winkelangaben erkannt und definiert.

Um die Attribute besser vor direkter Manipulation von außen zu schützen, wurden alle `private` deklariert. Da diese Vorgehensweise Bestandsteil eines guten objektorientierten Designs sein sollte, war sie bereits im OOA-Modell geplant. Ausdruck dessen sind die dort eingeführten `Get`-/`Set`-Methoden, die nun weiter verfeinert werden mussten.

b) Signatur für Accessor- und Mutator-Methoden

Für alle unter **a)** konkretisierten Attribute ist der Rückgabewert der entsprechenden `Get`-Methode zu vervollständigen. Dafür wird der Typ des Attributes übernommen. Für `GetAngleDest` ergab sich `TAngle GetAngleDest( int, BOOL& )`. Der erste Parameter ist der Index des Antriebs im Motorsteuerungs-Subsystem, dessen Sollposition ausgelesen werden soll. Der Weg Antriebe zu adressieren, konnte erst im Programmcode-Review ermittelt werden. Der zweite Parameter zeigt die Gültigkeit des Rückgabewertes, die z.B. vom übergebenen Index abhängt.

Da die Methode `CanSetAngleDest` zurückgeben soll, ob `SetAngleDest` durchführbar ist, ergab sich `BOOL CanSetAngleDest( int )`. Weil diese `Can`-Methode existiert, ist der Rückgabewert für `SetAngleDest` nicht `void`, sondern von der `CanSet`-Methode übernommen: `BOOL SetAngleDest( int, TAngle& )`. Der erste Parameter ist wieder der Index des Antriebs und der zweite gibt die zu setzende Sollposition an. Er wurde als Referenz deklariert, um stets den wirklich in der Funktionskomponente gespeicherten Wert zurückzugeben. Ein vom Anwender eingegebenen Wert wird in der `Set`-Methode auf

Wertebereich und Nachkommastellengenauigkeit überprüft, ggf. korrigiert und an die aufrufende Nutzeroberfläche (per Referenz) zurückgegeben. Diese kann so jederzeit den aktuellen Systemzustand anzeigen. Die Gültigkeit der gesetzten Attribute muss damit nur einmal zentral in der Funktionskomponente verwaltet werden.

```
BOOL SetValue( Type &aValue ) {
  if ( CanSetValue(aValue)==FALSE ) {
    aValue= GetValue(); // vorher gesetzten Wert zurückgeben
    return FALSE;
  }
  aValue= m_Value= Manipuliere1(aValue);
  return TRUE;
}
```

Abb. II.21 Allgemeines Implementierungsschema für eine `Set`-Methode, mit `CanSet`

c) Signatur für Methoden zur Realisierung der Funktionalität

Die Parameterliste dieser Methoden ist völlig von der verbalen Spezifikation abhängig und daher nicht verallgemeinerbar. Nur der Rückgabewert kann, wie bei **b)**, vom Vorhandensein einer `CanSet`-Methode abgeleitet werden. Für die, bei **c)** in Kapitel **II.2.5**, abgeleiteten Methode des Fahrbetriebs resultiert aus der Existenz von `BOOL CanDoDrive( int, EDirection )` die Signatur `BOOL DoDrive( int, EDirection )`.

d) Kapselung fremder Funktionen und weitere Methoden

Nach **a)** liegen einige Attribute unter der Verwaltung anderer Subsysteme. Die Nutzeroberfläche darf auf diese Attribute und Methoden nicht direkt zugreifen, so dass in der Funktionskomponente einige zusätzliche Methoden erforderlich sind. Diese kapseln das benutzte Subsystem, indem sie Anfragen entsprechend weiterleiten. In der neuen ‚Manuellen Justage' entstand so z.B. die Methoden `BOOL IsMoving( int )`, welche im Motorsubsystem anfragt, ob sich der Antrieb bewegt. Die Benennung dieser Methoden sollte identisch zu den Methoden im Subsystem sein.

Bei der neuen ‚Manuellen Justage' waren vier zusätzliche Methoden erforderlich, welche die Offset Berechnungen durchführen, diese aber nicht den Objektzustand ändern. Sie sind deshalb nicht-zustandsverändernd und fallen in keine bislang genannte Kategorie. Aufgrund ihrer Berechnungsfunktion wurden sie mit dem Präfix `Calc` benannt.

Das Ergebnis der Anwendung der Schritte **a)** bis **d)** bei der neuen ‚Manuellen Justage' zeigt **Abb. II.22**. Nach dem OOA-Modell sollten alle benötigten Daten durch die neue ‚Manuelle Justage' verwaltet werden. Durch das Programmcode-Review der benutzten Subsysteme konnten dort einige Attribute und Methoden identifiziert werden, die benötigte Daten bereits verwalten. Diese konnten damit aus `TManJustage` entfernt werden. Das Element `TPsdData` wurde sogar komplett entfernt (vgl. **Abb. II.19** und **Abb. II.22**). Die für den Zugriff auf diese Attribute geplanten Methoden sind erhalten geblieben. Statt auf die Attribute direkt zuzugreifen (Accessor-Methoden), muss dort nun an das jeweilige Attribut bzw. die Accessor-Methode des entsprechenden Subsystems delegiert werden.

TManJustage

-m_MotorList:TMotorData *
-m_InkDetector:TOneDimDetector *
-m_Channel:int
-m_MotorCount:UINT

0..1
m_Motorlist

<<struct>>
TMotorData

-m_InkMotor:TMotor *
-m_AxisType:TAxisType
-m_MotorId:char [512]
-m_MotionType:EMotionType
-m_AngleDest:TAngle
-m_Speed:TSpeed
-m_AngleWidth:TAngle

TManJustage

+GetChannel():int
+GetAnglePerChannel():TAngle
+GetMotorName(:int,:BOOL &):LPCSTR
+GetUnit(:int,:BOOL &):LPCSTR
+GetAngle(:int,:BOOL &):TAngle
+GetAngleMin(:int,:BOOL &):TAngle
+GetAngleMax(:int,:BOOL &):TAngle
+GetDigits(:int,:BOOL &):UINT
+GetOffset(:int,:BOOL &):TAngle
+GetMotionType(:int,:BOOL &):EMotionType
+GetAngleDest(:int,:BOOL &):TAngle
+GetSpeed(:int,:BOOL &):TSpeed
+GetAngleWidth(:int,:BOOL &):TAngle
+GetDetector():TDetector *
+SetMotionType(:int,:EMotionType &):BOOL
+SetAngleDest(:int,:TAngle &):BOOL
+SetSpeed(:int,:TSpeed &):BOOL
+SetAngleWidth(:int,:TAngle &):BOOL
+CanSetMotionType(:int):BOOL
+CanSetAngleDest(:int):BOOL
+CanSetSpeed(:int):BOOL
+CanSetAngleWidth(:int):BOOL

Spezialisierung von Attributen
Signatur für Accessor- und Mutatormethoden
Signatur für Methoden zur Realisierung der Funktionalität
Kapselung fremder Funktionen
weitere Methoden

TManJustage

+GetMotorCount():UINT
+SetRelativeZero(:int):BOOL
+ResetRelativeZero(:int):BOOL
+SetOffset(:int,:TAngle &):BOOL
+DoDirect(:int):BOOL
+DoDrive(:int,:EDirection):BOOL
+DoStep(:int,:EDirection):BOOL
+DoStop(:int):BOOL
+DoStartMeasure(:HWND):BOOL
+DoStopMeasure():BOOL
+SetChannel(:int &):BOOL
+CanResetRelativeZero(:int):BOOL
+CanSetOffset(:int):BOOL
+CanDoDirect(:int):BOOL
+CanDoDrive(:int,:EDirection):BOOL
+CanDoStep(:int,:EDirection):BOOL
+CanDoStop(:int):BOOL
+CanDoStartMeasure():BOOL
+CanDoStopMeasure():BOOL

TManJustage

+IsMoving(:int,:BOOL &):BOOL
+IsCalibrated(:int,:BOOL &):BOOL
+ParsingAxis(:LPCSTR):UINT
+GetDetectorAxisIdx():int
+GetDetectorName(:BOOL &):LPCSTR
+HasDetectorAxis():BOOL
+IsMeasuring():BOOL
+IsMeasureReset():BOOL
+GetMeasureProgress(:int &,:LPSTR,:int &):BOOL
+GetMeasureHwb():double
+UpdateDetector():void

TManJustage

+TManJustage()
+~TManJustage()
+IsIndexValid(:int):BOOL
+HasOffset(:int,:BOOL &):BOOL
+GetMotorIdx(:LPCSTR):int
+GetMotorIdx(:TAxisType):int
+CalcOffsetFromAngle(:int,:TAngle &):TAngle
+CalcAngleFromOffset(:int,:TAngle &):TAngle
+CalcChannelOffset(:TChannel &):TAngle
+CalcChannelAngle(:TChannel &):TAngle

Abb. II.22 UML-Klassendiagramm für die Funktionskomponente der neuen ‚Manuelle Justage', gegliedert nach der Bearbeitungsreihenfolge ihrer Elemente

II.3.2 Ableitung der Oberflächenfenster aus dem Pflichtenheft

In einem objektorientierten Design wird jede Nutzeroberfläche, ob Fenster oder Dialog, durch eine separate GUI-Klasse realisiert. Die Anzahl und deren Typ kann aus dem Human-Interface-Prototypen übernommen werden. Daraus ergeben sich auch die zu verwendenden Fensterbasisklassen, welche wiederum die Vererbungshierarchie festlegen. Bis auf das Überschreiben von Methoden der Basisklassen, gibt es hier keine allgemeinen Vorgaben für Methoden.

Einige Datenteil-Einträge aus dem Pflichtenheft betreffen die Nutzeroberfläche. Daraus lassen sich wie bei **II.2.5 a)** und **II.3.1 a)** Attribute ableiten und spezialisieren. Methoden für den Zugriff auf die Attribute sind nicht erforderlich, weil die Nutzeroberfläche nicht durch andere Systeme benutzt wird.

Empfehlenswert ist die Erstellung separater Ereignisbehandlungsroutinen für Steuerelemente als Methoden. Diese werden bereits von vielen Entwicklungsumgebungen automatisch, in der Implementierungsphase, erzeugt. Sinnvoll ist auch hier wieder die Benennung mit dem Präfix `On`, was bspw. konform zu Microsoft Visual C++ (MVC) ist.

Für alle Methoden und Attribute kann ein strenger Zugriffsschutz gewählt werden, da ein Zugriff von außen nicht erforderlich ist. Nur das Anzeigen und Verstecken anderer Fenster ist erlaubt, um die horizontale Konsistenz zwischen den Nutzeroberflächen zu wahren. Ein Informationsaustausch darf nur über eine gemeinsame Funktionskomponente erfolgen.

Anwendungsfall: ‚Manuelle Justage'

Im ‚XCtl'-Projekt waren die Fensterbasisklassen nicht von Borland C++ übernommen, sondern mit Hilfe von API-Funktionen realisiert und im Subsystem ‚Benutzeroberfläche' zusammengefasst. Da eine entsprechende Dokumentation im Web-Repository fehlte, erfolgte ein Reverse Engineering des Programmcodes, durch die Autoren. Dabei wurden so viele Design- und schwerwiegende Laufzeitfehler identifiziert, dass eine komplette Umstrukturierung als beste Lösung erachtet wurde. Der ursprüngliche Zustand wurde in **[M7]** festgehalten.

Im Verlauf des Reengineering wurde eine völlig neue Vererbungshierarchie entwickelt, welche die spätere Win32-Portierung in ein MVC-Projekt berücksichtigt. Dazu wurden die GUI-Basisklassen des ‚XCTL'-Projektes soweit separiert, dass dieses Subsystem in einem solchen Projekt getestet werden konnte.

Zudem soll die neue ‚Manuelle Justage' auch über Tastenkombinationen steuerbar sein. Weil es in Win16-Anwendungen dafür keine praktikable Lösung gibt, unterstützen die Basisklassen (im Hinblick auf die Portierung) Tastenkombinationen, sobald das Projekt mit einem Win32-Compiler gelinkt wird (präprozessorgesteuert). So sind diese in der neuen Win32-Version der ‚Manuellen Justage' automatisch verfügbar. Zum neuen Zustand des Subsystems ‚Benutzeroberfläche' entstand das Designdokument **[M9]**.

Da für die neue ‚Manuelle Justage' nur modale Dialogfenster zu erstellen waren, erfolgte die Ableitung von `TModalDlg`. Diese Klasse definiert einen Pool von Behandlungsroutinen für grundlegende Ereignisse, die in den Ableitungen überschrieben und spezialisiert werden können.

Um die Nutzeroberfläche deutlich von der Funktionskomponente zu trennen, wurden die Dialogfensterklassen mit dem Suffix `Dlg` gekennzeichnet. So entstanden die Klassenbezeichner `TManJustageDlg` (für **Abb. II.10**, oben), `TMotorOffsetDlg` (**Abb. II.10**, rechts) und `TPsdOffsetDlg` (**Abb. II.10**, links). Dieses Vorgehen wurde sogar auf das gesamte ‚XCtl'-Projekt angewandt. Dabei wurde zusätzlich das Suffix `Wnd` eingeführt, um alle Fenster, die keine Dialoge darstellen, zu kennzeichnen.

Den Kernpunkt einer Dialogfensterklasse im ‚XCtl'-Projekt bildet die Methode `void Dlg_OnCommand( HWND, int, HWND, UINT )`. Dort können alle Steuerelementbotschaften empfangen, identifiziert, behandelt oder weitergeleitet werden. Durch die Komplexität der Benutzeroberfläche der neuen ‚Manuelle Justage' (71 Steuerelemente) und die mehrfach vorhandenen, identischen Teilbereiche, wurde die Auslagerung in einzelne Behandlungsroutinen (`On`-Methoden) für jedes Steuerelement nicht vorgenommen. Die Autoren entschieden sich spezielle Verwaltungsroutinen für die Teilbereiche zu erstellen, um deren Behandlung zusammenfassen zu können.

TManJustageDlg
-m_lnkManJustage:TManJustage *
-m_lnkTimer:TInterfaceTimer *
-m_Selection:UINT *
-m_Parallel:BOOL
-m_StateRefresh:BOOL
-m_DisplayNoActionMsg:BOOL

TManJustageDlg
#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL
#Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void
#Dlg_OnHScrollBar(:HWND,:HWND,:UINT,:int):void
#LeaveDialog():void
#OnTimer(:const TBasicTimer *):void

TManJustageDlg
-TB_Count():UINT
-TB_Next(:ETB &):BOOL
-TB_GetId(ETB:const):UINT
-TB_GetName(ETB:const,:LPSTR):void
-GetResId(ETB:const,ERessourceId:const):int
-GetCtrl(ETB:const,ERessourceId:const):HWND
-TB2Selection(ETB:const):UINT
-Selection2TB(UINT:const):ETB
-SetSelection(ETB:const,UINT:const):BOOL
-CanSetSelection(ETB:const,UINT:const):BOOL

TManJustageDlg
-TB_CheckLeft(ETB:const,:const int):BOOL
-TB_CheckMiddle(ETB:const,:const int):BOOL
-TB_CheckRight(ETB:const,:const int):BOOL
-TB_CheckExit(ETB:const,:const int):BOOL
-TB_StartStopClicked(ETB:const):void
-TB_AngleDestEntered(ETB:const,BOOL:const):void
-TB_SpeedEntered(ETB:const,BOOL:const):void
-TB_StepWidthEntered(ETB:const,BOOL:const):void
-TB_MotorChanged(ETB:const):void
-OnTB_Controls(ETB:const,BOOL:const):void
-OnTB_KindOfMove(ETB:const,BOOL:const):void
-OnTB_StartStop(ETB:const,BOOL:const):void
-OnTB_MotionParam(ETB:const,BOOL:const):void
-OnTB_Positions(ETB:const,BOOL:const):void
-OnTB_Offset(ETB:const,BOOL:const):void
-OnTB_Units(ETB:const):void
-OnTB_ControlsEnable(ETB:const):void
-OnTB_MotorListEnable(ETB:const):void
-OnTB_KindOfMoveEnable(ETB:const):void
-OnTB_StartStopEnable(ETB:const):void
-OnTB_MotionParamEnable(ETB:const):void
-OnTB_PositionsEnable(ETB:const):void
-OnTB_OffsetEnable(ETB:const):void

TManJustageDlg
+TManJustageDlg()
+~TManJustageDlg()
-OnMotionStarts(UINT:const):void
-OnMotionProgress(UINT:const):void
-OnMotionStops(UINT:const):void
-OnMeasureStarts():void
-OnMeasureProgress():void
-OnMeasureStops():void
-OnPsdOffset():void
-OnChooseTB():void
-OnNoAction():void
-OtherControls():void
-ScrollbarCalculate(TAngle:const,TAngle:const, TAngle:const, :int &,:int &,:int &):void

Ableitung und Spezialisierung von Attributen
Überschreiben von Methoden der Basisklasse(n)
Verwaltung der Teilbereiche
Behandlungsroutinen und Aktualisierung eines Teilbereichs
weitere Methoden

Abb. II.23 UML-Klassendiagramm für die Dialogfensterklasse `TManJustageDlg` gegliedert, nach der Erstellungsreihenfolge ihrer Elemente

TMotorOffsetDlg	TPsdOffsetDlg
-m_lnkManJustage:TManJustage * -m_Idx:UINT -m_Digits:UINT	-m_lnkManJustage:TManJustage * -m_Idx:UINT -m_Digits:UINT

TMotorOffsetDlg	TPsdOffsetDlg
#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL #Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void	#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL #Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void

TMotorOffsetDlg	TPsdOffsetDlg
+TMotorOffsetDlg(UINT:const,:TManJustage *) -CalcValue(:BOOL):BOOL -ChooseAngleMode(:BOOL):void -IsAngleMode():BOOL	+TMotorOffsetDlg(UINT:const,:TManJustage *) -CalcValue(:BOOL):BOOL

Ableitung und Spezialisierung von Attributen
Überschreiben von Methoden der Basisklasse(n)
weitere Methoden

Abb. II.24 UML-Klassendiagramm für die Dialogfensterklassen `TMotorOffsetDlg` und `TPsdOffsetDlg`, gegliedert nach der Erstellungsreihenfolge ihrer Elemente

II.3.3 Anbindung der Nutzeroberfläche an die Funktionskomponente

In den folgenden Kapiteln soll es darum gehen, welche Möglichkeiten es gibt, Nutzeroberfläche und Funktionskomponente miteinander zu verknüpfen. Diskutiert wird, welche Verfahren für den Daten- und Informationsaustausch existieren, so dass die Änderung des Zustands in einer Komponente automatisch in allen anderen beteiligten Komponenten nachvollzogen wird.

II.3.3–1 Ausgangspunkt: ADV-Modell

Die Dekomposition kann nicht nur als Hilfsmittel bei der Softwareentwicklung dienen, sondern auch grundlegender Mechanismus eines Entwicklungsmodells sein. Ein solches ist das **Abstract Design View-** = ADV-Modell. Dieses informale Modell für objektorientierte Softwaresysteme beschreibt die Struktur eines Softwaresystems auf der Designebene. Es wurde ursprünglich zur Kapselung der Nutzeroberfläche bei interaktiven Softwaresystemen benutzt. Die Basiselemente dieses Modells sind Programm- und Schnittstellenkomponenten.

Die Programmkomponenten werden im ADV-Modell als **Abstract Design Objects**, kurz ADOs bezeichnet. Sie sind die Träger der grundlegenden Funktionalität eines Programmes. Sie besitzen einen inneren Zustand und bieten eine öffentliche Schnittstelle zur Zustandsänderung. Eine Schnittstellenkomponente, im Modell als **Abstract Design Viewer** = ADV bezeichnet ist eine Programmkomponente mit Sonderfunktionalität. Dies kann sein: Timing-Funktionalität, Netzwerkkommunikation, Druckvorgänge oder Kommunikation mit dem Anwender – also die Nutzeroberfläche. Jedes ADV ist im Kern ein ADO und besitzt damit ebenso einen inneren Zustand und eine öffentliche Schnittstelle.

Die Beziehung zwischen einem ADO und einem ADV wird als **‚views-a'-Relation** bezeichnet. Es bedeutet, dass ein ADV ein ADO beobachtet. Besteht eine solche Beziehung, kennt ein ADV den Zustand des beobachteten ADOs. Wird ein ADV als Nutzeroberfläche eingesetzt, dann beobachtet diese ein ADO und zeigt dem Anwender dessen aktuellen Zustand. Zur Realisierung der ‚views-a'-Beziehung soll ein ADO durch eine **formale Assoziation** an ein ADV gebunden werden. Diese ist durch eine gemeinsame Namensgebung von ADO und ADV, der Konsistenz der Zustände, im Modell die **vertikale Konsistenz**, und der Möglichkeit zur Zustandsänderung des ADO durch das ADV gekennzeichnet. Ein ADV kann mit mehreren

ADOs assoziiert sein und kennt dann den Zustand all dieser Objekte. Ein ADO kann aber auch von mehreren ADVs gleichzeitig beobachtet werden. In diesem Fall muss für eine Konsistenz der Zustände aller ADVs gesorgt werden – im Modell die **horizontale Konsistenz**.

Ein ADV-Modell-konformer Softwareentwurf verlangt somit die Dekomposition in Funktionskomponente und Nutzeroberfläche. In den folgenden Kapiteln werden nun Design-Mechanismen zur Realisierung der Dekomposition nach den ADV-Modell-Richtlinien vorgestellt. Eine ausführliche Beschreibung des ADV-Modells ist bei [1] zu finden.

II.3.3–2 Anwendung bei der Dekomposition

Die Verknüpfung einer spezialisierten GUI-Klasse (ADV) mit einem ADO wird durch **Delegation** realisiert. Im einfachsten Fall erstellt die Nutzeroberfläche dazu selbst ein Objekt der Funktionskomponente und verwaltet dieses als Referenz. Beobachten mehrere ADVs dieselbe ADO, können Objekt und Referenz durch das **Singleton-Muster** (siehe [8], Creational Patterns –> Singleton) bereitgestellt werden. Damit ist nur die Erstellung eines einzigen Objektes möglich, um Zustandsinkonsistenzen der ADVs zu vermeiden. Haben mehrere GUI-Klassen Zugriff auf eine Funktionskomponenten-Klasse, dann hat jedes Fenster-Objekt eine eigene Referenz auf ein und dasselbe Objekt.

Die ‚views-a'-Relation ist damit durch das Designmittel Assoziation realisiert. Eingaben die der Anwender in der Nutzeroberfläche durchführt, erzeugen Zustandsänderung in der ADO. Dazu werden öffentlichen Methoden über die Referenz, auf die Funktionskomponente, gerufen. Die wohldefinierte Schnittstelle der Funktionskomponente garantiert Information Hiding gegenüber der Nutzeroberfläche und die unidirektionale Assoziation sorgt für Environment Hiding.

In der ADO können Zustandsänderungen durch Prozesse von außerhalb[14] ausgelöst werden. Geschieht dies durch ein ADV, hat nur dieses selbst Kenntnis davon. Alle anderen ADVs müssen vom ADO über Zustandswechsel informiert werden oder sich selbst um die Aktualisierung kümmern. Beides bedarf zusätzlichen Aufwands im Design. Weil das ADO nichts über die assoziierten ADVs wissen sollte (Environment Hiding), ist der Datenaustausch vom ADO zum ADV mit der o.g. Assoziation allein nicht realisierbar. Wenn die Synchronisation vom ADV initiiert wird, benötigt diese eine zusätzliche Komponente, um regelmäßig Zustandsaktualisierungen durchzuführen. Die dafür nötigen Verfahren werden in den folgenden Kapiteln ausführlich diskutiert.

II.3.3–3 Das Observer-Muster

Dieses Design-Muster (engl.: Pattern) beschreibt sprachunabhängige Designrichtlinien, welche Beziehungen von Komponenten in einem modularisierten Softwareprojekt aufbauen. Diese Richtlinien gewährleisten die Konsistenz der Zustände, das Information Hiding sowie die Wiederverwendbarkeit für die beteiligten Komponenten. Ein besonderer Punkt ist zudem die Wahrung des Environment Hiding. Eine ausführliche Beschreibung des Observer-Musters ist in [8] zu finden.

Die Basiselemente des Observer-Musters sind **Subjekte** (engl.: Subjects) und **Beobachter** (engl.: Observer). Zwischen diesen beiden besteht eine Beziehung, die als bekannt machen/ teilnehmen (engl.: **publish-subscribe**) bezeichnet wird. Sie ist dadurch charakterisiert, dass beliebig viele Beobachter mit einem Subjekt assoziiert sein können, ohne dass dieses Subjekt genaueres über seine Beobachter weiß. Ein Subjekt ist in der Lage, Nachrichten an assoziierte Objekte zu versenden, ohne wissen zu müssen, wer sie erhält. Zur Umsetzung dieser Beziehung

[14] z.B. Hardwareansteuerung oder langwierige Berechnungen

werden die vier Elemente: Subjekt, Beobachter, **Konkretes Subjekt** (engl.: Concrete Subject) und **Konkreter Beobachter** (engl.: Concrete Observer) mit folgenden Eigenschaften benötigt.

Subjekt

- definiert eine Schnittstelle, um Beobachter zu binden und wieder freizugeben
- speichert die assoziierten Beobachter in eine Liste.

Beobachter

- definiert eine Schnittstelle für Objekte, die bei Zustandsänderungen im Subjekt benachrichtigt werden sollen

Konkretes Subjekt

- hat einen inneren Zustand
- sendet eine Nachricht, wenn sich dieser Zustand geändert hat

Konkreter Beobachter

- hat einen inneren Zustand, der konsistent zum Subjekt-Zustand sein soll
- hat eine Referenz auf das konkrete Subjekt
- implementiert die Beobachter-Schnittstelle, um die Konsistenz herzustellen

Im objektorientierten Design werden diese vier Elemente durch Klassen oder Interfaces realisiert. Die Beziehungen werden durch Vererbung und Assoziationen beschrieben (siehe **Abb. II.25**). Die im Subjekt definierte Schnittstelle wird als abstrakte Klasse oder Interface entworfen. Das Konkrete Subjekt ist eine Klasse, die das Subjekt-Interface einerbt und für den jeweiligen Anwendungsbereich spezialisiert. Die im Beobachter definierte Schnittstelle ist wieder eine abstrakte Klasse oder ein Interface. Der, als Klasse erstellte, Konkrete Beobachter erbt dieses Beobachter-Interface ein und besitzt eine unidirektionale Assoziation auf das Konkrete Subjekt. Dieses hat über das Subjekt-Interface eine Liste von Referenzen auf Beobachter-Interfaces.

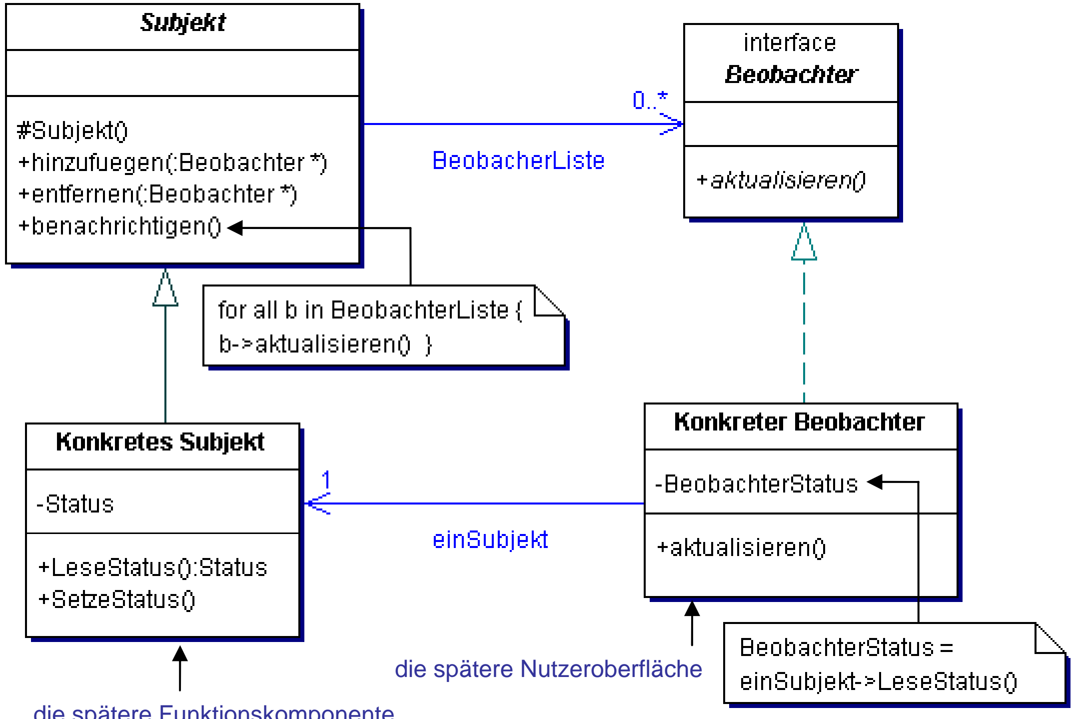


Abb. II.25 UML-Klassendiagramm zum allgemeinen Aufbau des Observer-Musters (nach **[8]**)

Bei der Dekomposition stellt die Funktionskomponente das Konkrete Subjekt dar, weil sie Änderungen ihres inneren Zustands an die Nutzeroberfläche senden soll (der Informationsfluss von der Oberfläche zur Funktionskomponente wurde bereits bei **II.3.3–2** erklärt). Wenn die Funktionskomponente mehrere Nutzeroberflächen gleichzeitig über einen Zustandsübergang informieren soll, ist ein Subjekt in Form einer abstrakten Klasse zu erstellen. Diese beinhaltet eine Beobachter-Liste und Methoden, die es gestatten, Beobachter-Interfaces dynamisch hinzuzufügen oder zu entfernen. Die Funktionskomponente erbt so den Verwaltungsmechanismus für Beobachter-Interfaces und hat somit Kenntnis über die assoziierten Nutzeroberflächen. Jede einzelne Nutzeroberfläche wird durch einen Konkreten Observer repräsentiert und ist von einer GUI-Basisklasse und einem Beobachter-Interface abgeleitet. Da Interface-Vererbung keine Mehrfachvererbung darstellt, ist das auch in Sprachen mit Einfachvererbung, wie z.B. Java, möglich.

Vorteile des Observer-Musters

- Es findet nur dann ein Informationsaustausch zwischen Subjekt und Beobachter statt, wenn wirklich ein Zustandsübergang stattgefunden hat.
- Bei einer Zustandsänderung im Subjekt werden sofort alle assoziierten Beobachter benachrichtigt.
- Beim Austausch des Konkreten Beobachters (z.B. Neuimplementierung) sind alle möglichen Zustandsänderungen durch das einzuerbende Beobachter-Interface vollständig definiert.

Neben den allgemeinen Vorteilen gibt es jedoch Aspekte, welche die Verwendung des Observer-Musters speziell bei der Dekomposition in Funktionskomponente und Nutzeroberfläche begünstigen.

- Das Observer-Muster bietet eine hohe Flexibilität, weil die Zustandsaktualisierungen in die Funktionskomponente ausgelagert werden. Dadurch ist die Nutzeroberfläche nicht so umfangreich und kann leichter ausgetauscht werden (z.B. beim Prototyping).

Nachteile des Observer-Musters

- Der Austausch des Konkreten Subjektes ist durch das Subjekt und das assoziierte Beobachter-Interface stark eingeschränkt. Bei der Dekomposition, wo das Konkrete Subjekt der Funktionskomponente entspricht, ist das aber eher selten (**I.4.3**: Ersetzung der simulierten Reaktion eines Programmes durch die wirkliche Programmfunktionalität).
- Ein Subjekt verwaltet eine Liste von Beobachtern. Nun kann es sinnvoll sein, dass nur ausgewählte Beobachter über bestimmte Ereignisse informiert werden sollen, um Ressourcen zu sparen. Dazu sind aber zusätzliche Informationen in der Beobachter-Liste des Subjektes nötig.

Anwendungsfall: ‚Manuelle Justage'

Bei der ‚Manuellen Justage' ist das Konkrete Subjekt die Klasse `TManJustage`. Da nur eine Oberfläche über Zustandsänderungen informiert werden soll, kann das Subjekt-Interface entfallen. Die Erzeugung von `TManJustage` findet im Konstruktor von `TManJustageDlg` statt. Um das Wissen der Funktionskomponente über die Nutzeroberfläche auf die Methoden des Observer-Interface zu beschränken, erhält der Konstruktor der Funktionskomponente nur

eine auf IManJustageDlg gecastete Referenz von TManJustgeDlg, die beim Erzeugen des Fensters (Konstruktor von TManJustageDlg) übergeben wird.

```
TManJustageDlg::TManJustageDlg() {
  m_lnkManJustage = new TManJustage( (IManJustageDlg*)this );
  … // Initialisierung der Attribute
}
```

IManJustageDlg ist hier das Beobachter-Interface. Es enthält die abstrakten Methoden, die bei Zustandsänderungen innerhalb der Funktionskomponente gerufen werden (Ereignisse).

Methode	Beschreibung
OnMotionStarts	Die Bewegung des, als Parameter übergebenen, Antriebs hat gestartet.
OnMotionProgress	Der Antrieb bewegt sich.
OnMotionStops	Der angegebene Antrieb hat gestoppt.
OnMeasureStarts	Die Halbwertsbreitenmessung hat gestartet.
OnMeasureProgress	Die Halbwertsbreite wird gemessen.
OnMeasureStops	Die Messung wurde gestoppt oder unterbrochen.

Tab. II.4 Abstrakte Methoden des Beobachters **TManJustage**

TManJustage ruft diese Methoden über die Referenz m_lnkManJustageDlg, die dem Konstruktor übergeben wurde.

Die Nutzeroberfläche der neuen ‚Manuellen Justage' TManJustageDlg ist ein modaler Dialog. Sie erbt TModalDlg und das Beobachter-Interface (IManJustageDlg) ein. Um eine Instanz von TManJustageDlg erstellen zu können, sind alle abstrakten Methoden des eingeerbten Beobachters (IManJustage) zu implementiert. So können bei Zustandsänderungen in der Funktionskomponente Anpassungen in der Oberfläche durchgeführt werden, um die Konsistenz der Zustände von Funktionskomponente und Nutzeroberfläche zu wahren.

Die Initiative für den Datenaustausch nach einem Zustandsübergang liegt bei der Funktionskomponente. Kann ein solcher Übergang von außerhalb ausgelöst werden, z.B. durch ein anderes Subsystem oder eine Hardware, und wird die Funktionskomponente darüber nicht automatisch informiert, ist eine Timer-Komponente zur Generierung von regelmäßigen Ereignissen nötig. Bei deren Eintreten kann der Zustand des ausgelagerten Prozesses ausgelesen werden.

Die Verwendung eines Timers für die neue ‚Manuelle Justage' resultiert aus der engen Hardwarekopplung über das Motorsteuerung-Subsystem und über die Halbwertsbreitenmessung des Subsystems Ablaufsteuerung. Da im gesamten ‚XCtl'-Projekt Timerereignisse nur über Fenster-Handle realisiert waren, mussten von den Autoren zuvor entsprechende Klassen für fensterunabhängige Timer implementiert werden. In Hinblick auf die Win32-Portierung entstanden dabei die Klassen TTimer16 und TTimer32. Je nach Plattform bildet die entsprechende Klasse die Basis für TInterfaceTimer (Observer-Muster) und TWindowTimer (fenster-handle-orientiert).

Die neue ‚Manuellen Justage' verwendet die Klasse TInterfaceTimer als Konkretes Subjekt. TManJustage ist hier Konkreter Beobachter und das Interface ITimer ist Beobachter-Schnittstelle. Die Funktionskomponente erstellt den Timer selbst und speichert dessen Referenz in m_lnkTimer, um diesen zu steuern.

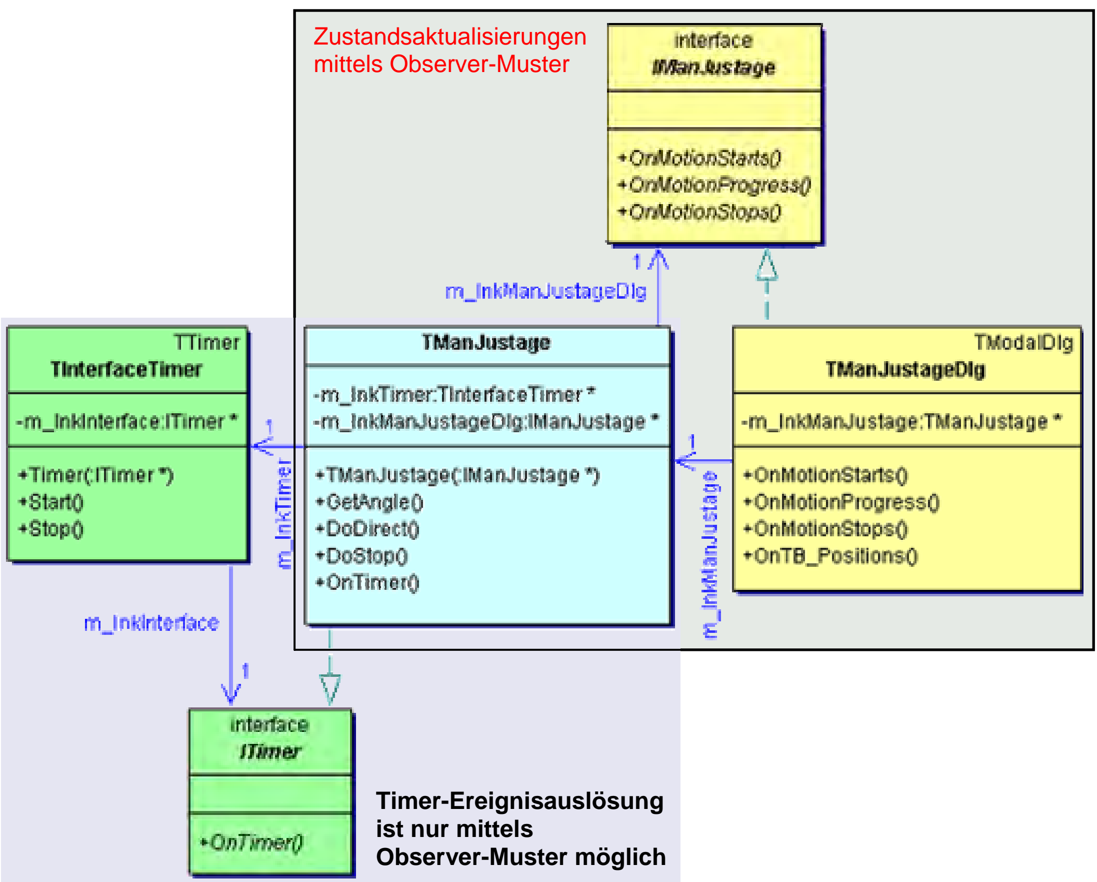


Abb. II.26 UML-Klassendiagramm für Observer-Muster in der neuen ‚Manuellen Justage' (vereinfacht)

Um eine Bewegung zu starten, wird in der Nutzeroberfläche z.B. die Methode `m_lnkTManJustage->DoDirect()` gerufen, woraufhin sich der Antrieb in Bewegung setzt und der Timer gestartet wird. Ist sein Intervall abgelaufen, ruft der Timer die Methode `OnTimer()` von `TManJustage`. Damit nun die Nutzeroberfläche aktualisiert wird, muss als Folge des Timer-Ereignisses eine der Methode aus **Tab. II.4** an `m_lnkTManJustage` delegiert werden. Das eigentliche Herstellen der Konsistenz erfolgt dann in der Nutzeroberfläche durch `OnTB_Positions` (Auslesen der aktuellen Antriebsposition mittels `m_lnkTManJustage->GetAngle()`) und `OnTB_ControlsEnable` (sperren bzw. freigegeben der Steuerelemente), siehe **Abb. II.27**:

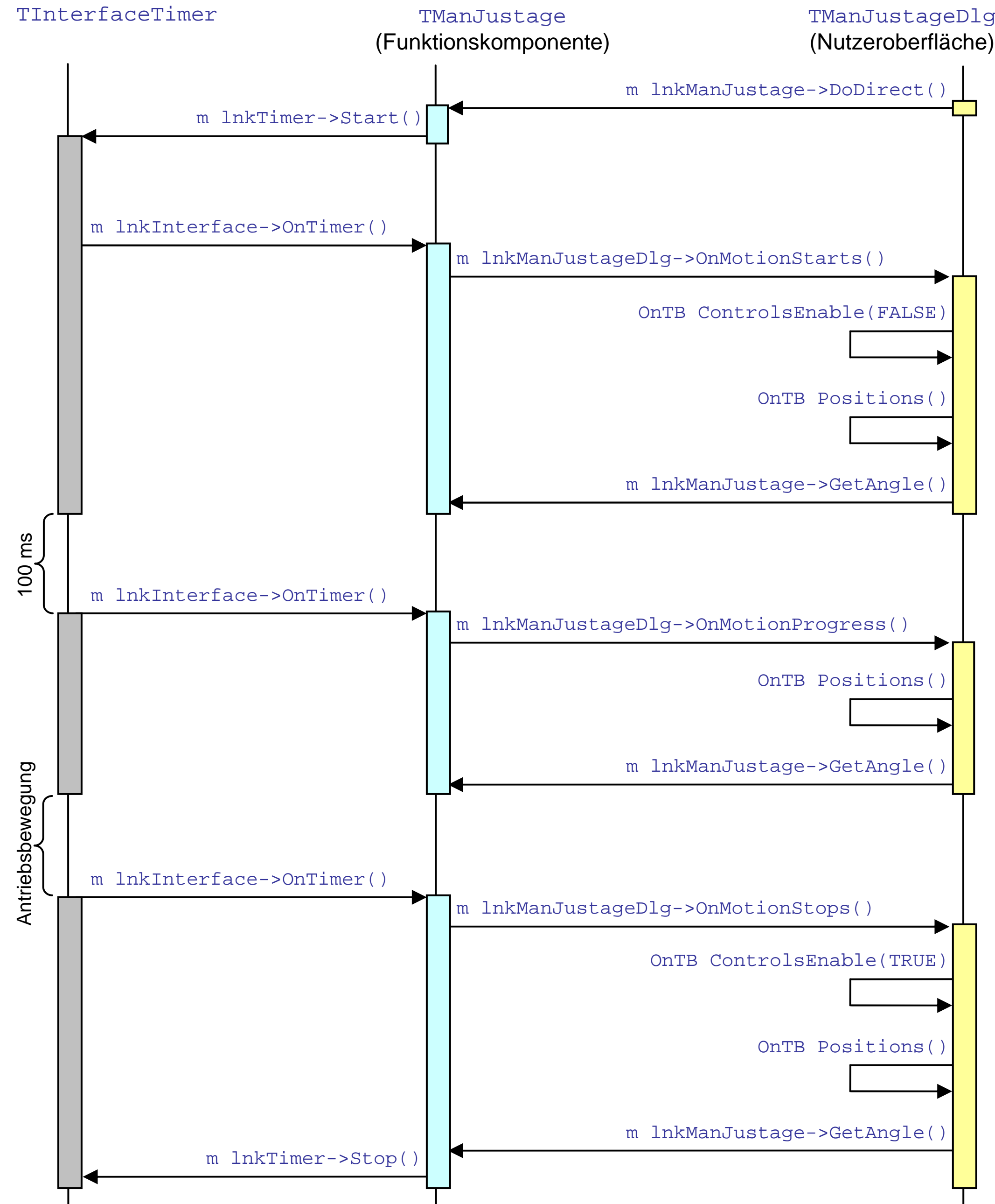


Abb. II.27 Interaktionsdiagramm für Observer-Muster
in der neuen ‚Manuellen Justage' während der Antriebsbewegung

II.3.3–4 Die Polling-Variante

Neben der, im vorigen Kapitel beschriebenen, eigentlichen Vorgehensweise für das Observer-Muster gibt es eine zweite Möglichkeit zur Umsetzung einer Subjekt-Beobachter-Beziehung. Diese Variante wird allgemein als Polling[15] bezeichnet. „Polling ist eine Programmiertechnik, bei der das Auftreten eines Ereignisses durch ständiges Abfragen ermittelt wird." (aus [10]) Angewendet auf die Dekomposition bedeutet dies, dass die Nutzeroberfläche (Konkreter Beobachter) das Auftreten eines Ereignisses innerhalb der Funktionskomponente (Konkreter Subjekt) durch ständiges Nachfragen ermittelt. Beim Polling entfallen sowohl das Subjekt- als auch das Beobachter-Interface, weil sie nicht benötigt werden.

[15] Die Methode heißt *Polling*, weil es wie bei den Bienen darum geht, mehrere Blüten immer wieder zu besuchen, um deren neue Pollen einzusammeln.

Erforderlich ist hier aber meistens ein Mechanismus (in der Regel ein Timer) der regelmäßig Ereignisse auslöst, die zum Aufruf von Methoden des Subjektes benutzt werden, um dessen Status zu erfragen. Diese Statusabfrage kann aber auch zu diskreten Zeitpunkten durch Anwendereingaben ausgelöst werden, dann wird kein Timer benötigt.

Vorteile der Polling-Variante

- Die Polling-Variante ist in allen objektorientierten Programmiersprachen weniger umfangreich[16] als das eigentliche Observer-Muster.
- Das Environment Hiding ist wesentlich besser, da das Konkrete Subjekt überhaupt keine Informationen über die assoziierten Konkreten Beobachter enthält. Deshalb werden weder die Beobachter noch die Subjekte Komponente benötigt.

Wie das eigentliche Observer-Muster hat auch die Polling-Variante Vorteile, die speziell bei der Dekomposition in Funktionskomponente und Nutzeroberfläche zum Tragen kommen.

- Das einfachere Design der Polling-Variante verbessert vor allem die Wiederverwendbarkeit der Funktionskomponente.
- Wenn die Aktualisierung zudem nur zu diskreten Zeitpunkten, wie Anwendereingaben, stattfinden soll, muss nicht jede Zustandsänderung behandelt werden. Beim Observer-Muster entsteht dort ein Overhead, weil stets jeder neue Zustand gemeldet wird.

Nachteile der Polling-Variante

- Bei der Verwendung eines Timers ist es sehr schwierig, den richtigen Zeitpunkt für die Abfragen zu finden. Werden diese in zu großen Zeitabständen gestellt, reagiert das Programm verzögert auf Ereignisse. Sind die Zeitabständen zwischen den Anfragen zu kurz, werden Rechenressourcen und Prozessorzeit verschwendet. Weil zur Gewährleistung der Aktualität eher öfter abgefragt wird, hat timergestütztes Polling fast immer Ressourcen-Verschwendung zur Folge.
- Da die Implementierung eines Konkreten Beobachters hier komplizierter ist als beim eigentlichen Observer-Muster, ist Polling nachteilig, wenn mehrere angeboten werden sollen.
- Muss zudem eine horizontale Konsistenz zwischen mehreren Konkreten Beobachtern gewährt werden, darf sie nicht durch zusätzliche Kommunikation zwischen diesen realisiert werden. Erforderlich wird dann eine Timer-Komponente für jeden Konkreten Beobachter. Eine zeitgleiche Aktualisierung ist damit trotzdem unmöglich, weil die Timer-Komponenten schlecht synchronisierbar sind

Anwendungsfall: ‚Manuelle Justage'

Um die Bewegung der Antriebe und die Halbwertsbreitenmesssung in der neuen ‚Manuellen Justage' zu verfolgen, ist man auch beim Polling auf die Verwendung der Timerkomponente angewiesen. Dazu ist wieder der oben erwähnte `TInterfaceTimer` nötig, der hier an die Nutzeroberfläche `TManJustageDlg` gebunden ist. Diese Anbindung erfolgt jedoch durch das eigentliche Observer-Muster. Dazu erbt `TManJustageDlg` neben `TModalDlg` auch das Timer-Interface `ITimer` ein. Das ist keine Mehrfachvererbung, weil `ITimer` nur ein Interface ist. `TManJustageDlg` erzeugt im Konstruktor dynamisch eine Referenz auf ein Timer-Objekt, das im Attribut `m_lnkTimer` gespeichert wird. Um die

[16] in Design und Implementation

Timer-Ereignisse zu behandeln, muss die Methode `OnTimer` implementiert werden. Für die Instanziierung der Oberfläche ist dies sogar zwingend, weil diese Methode in `ITimer` als pure-virtual function deklariert ist.

Nur bei der Anbindung der Timer-Komponente ist die Klasse `TInterfaceTimer` das Konkrete Subjekt, `TManJustageDlg` der Konkrete Beobachter und das Interface `ITimer` wieder die Beobachter-Schnittstelle.

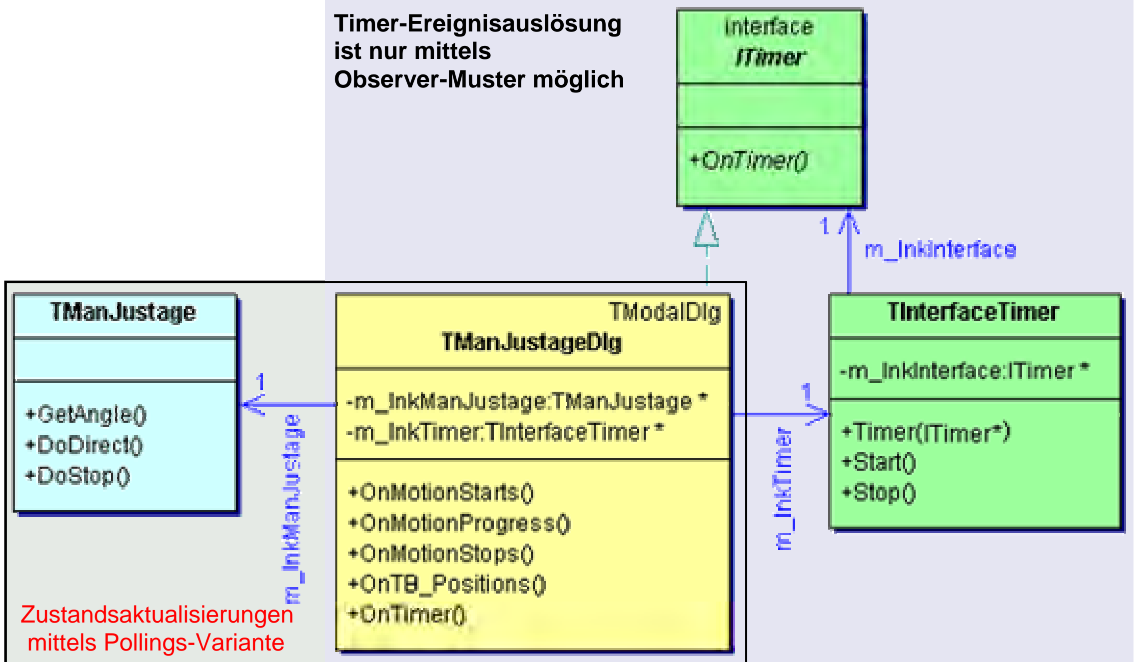


Abb. II.28 UML-Klassendiagramm für Polling in der neuen ‚Manuellen Justage' (vereinfacht)

Bei der Anbindung der Nutzeroberfläche an die Funktionskomponente mittels Polling ist `TManJustageDlg` der Konkrete Beobachter und `TManJustage` das Konkrete Subjekt. Alle Aktionen für den Datenaustausch werden von der Nutzeroberfläche gesteuert. Dazu wird hier mit `m_lnkManJustage->DoDirect()` die Antriebsbewegung gestartet. Auch die Verwaltung des Timers liegt bei der Nutzeroberfläche. Dieser wird mit `m_lnkTimer->Start()` gestartet. Ist das Timer-Intervall abgelaufen, wird die in `TManJustageDlg` implementierte Methode `OnTimer()` gerufen. Dort erfolgt die Aktualisierung der Nutzeroberfläche durch `OnTB_Positions()`, indem mittels `m_lnkManJustage->GetAngle()`, die aktuelle Antriebsposition ausgelesen und angezeigt wird (siehe **Abb. II.29**). Beim Starten und Stoppen eines Antriebs werden die Steuerelemente zusätzlich gesperrt bzw. freigegeben (`OnTB_ControlsEnable`).

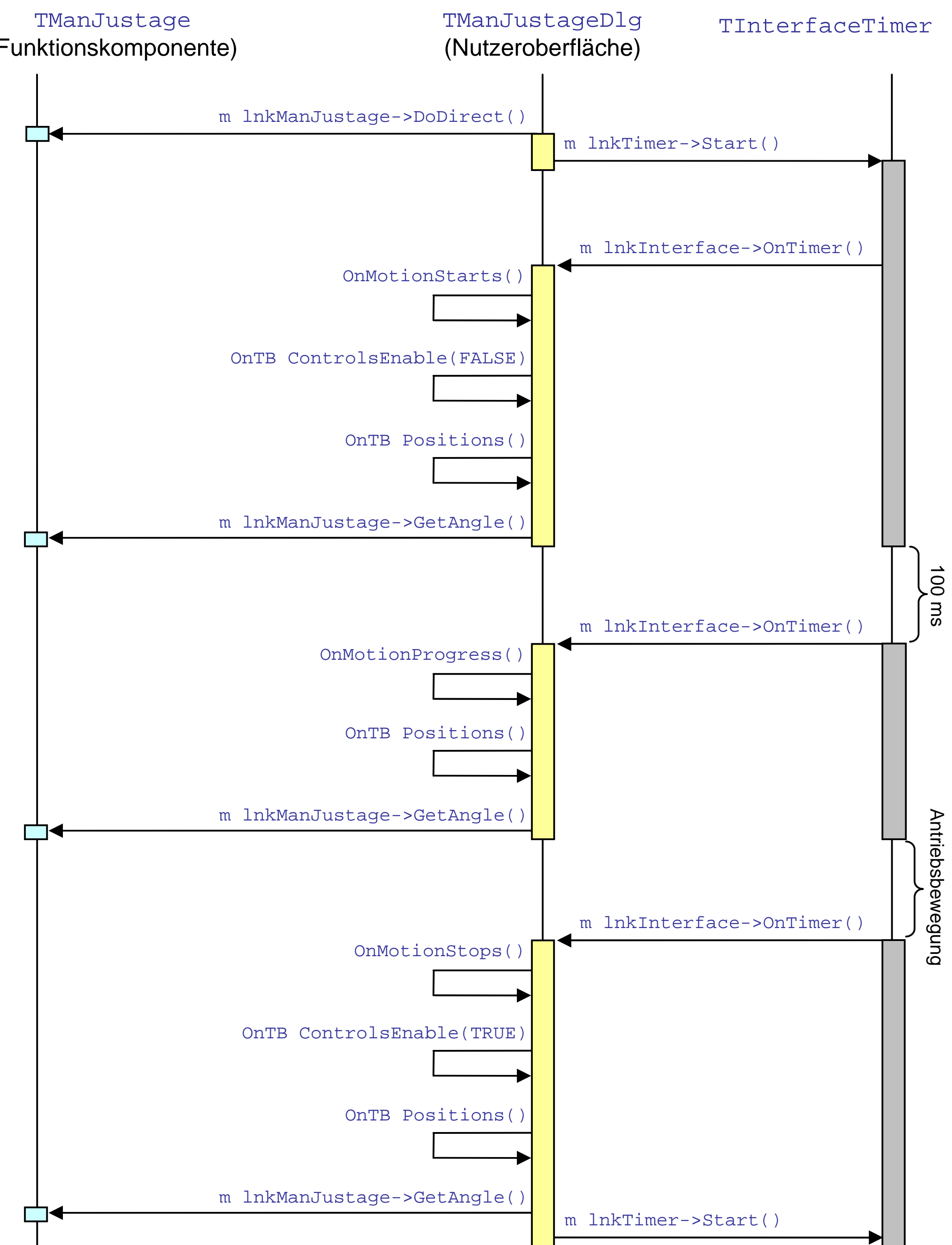


Abb. II.29 Interaktionsdiagramm für Polling
in der neuen ‚Manuellen Justage' während der Antriebsbewegung

II.3.3–5 Kardialitäten von Nutzeroberflächen und Funktionskomponenten

Sowohl das eigentliche Observer-Muster als auch die Polling-Variante gestatten die Verknüpfung einer Nutzeroberfläche mit einer oder mehreren Funktionskomponenten und umgekehrt. Um Fehleranfälligkeit bzw. Komplexität besser zu handhaben, sollte die Variante aber nicht beliebig ausgewählt werden.

Im Regelfall wird jede Nutzeroberfläche genau eine eigene Funktionskomponente besitzen (1 : 1). In einigen Fällen kann die Nutzeroberfläche aber auch auf mehrere Funktionskomponenten zugreifen (1 : n). Bei dem eigentlichen Observer-Muster würde für jede einzelne Funktionskomponente ein Beobachter-Interface benötigt (**Abb. II.30**).

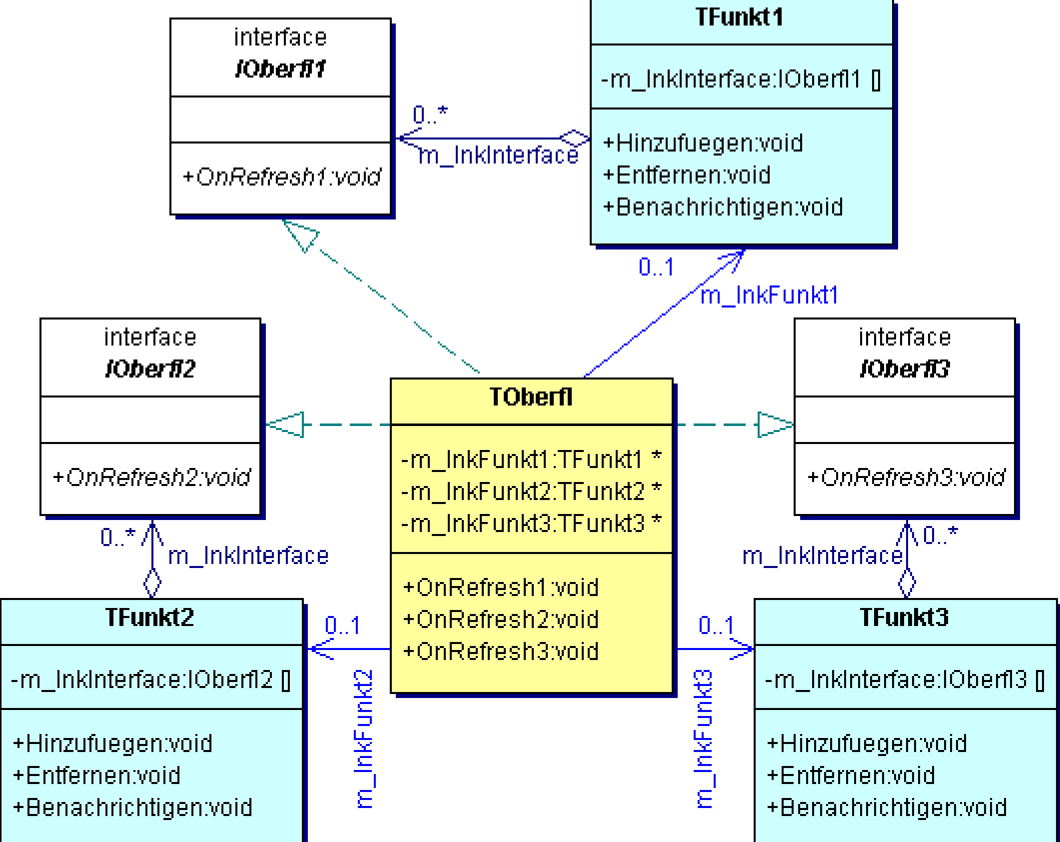


Abb. II.30 Bsp. für hohe Komplexität des Observer-Musters bei 1 : 3

Durch die Verwendung von Polling können Design und Programmcodeumfang bei diesen beiden Varianten problemlos reduziert werden, weil die Beobachter-Interfaces entfallen (**Abb. II.30**).

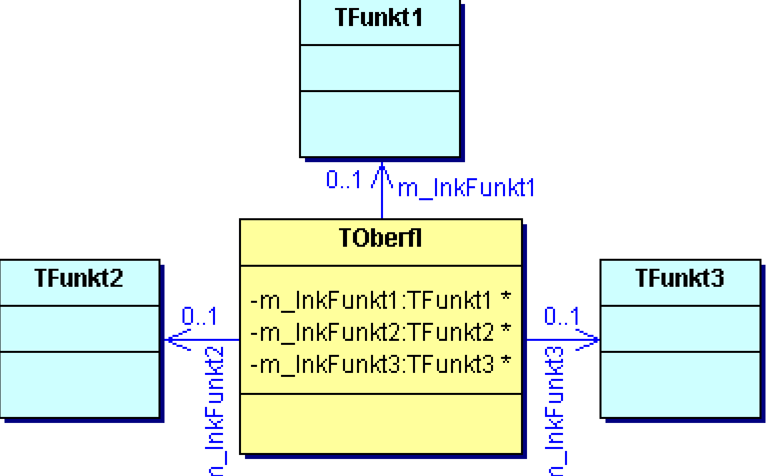


Abb. II.31 Bsp. für das einfache Design von Polling bei 1 : 3

Wenn mehrere Nutzeroberflächen auf genau eine Funktionskomponente zugreifen (m : 1), muss durch das Design immer sichergestellt werden, dass maximal eine Instanz der Funktionskomponente erstellt werden kann, z.B. durch das bereits erwähnte Singleton-Muster. Da jede Nutzeroberfläche bei Zustandsänderungen benachrichtigt werden muss, kann nur die Verwendung des eigentlichen Observer-Musters die horizontale Konsistenz zwischen den Nutzeroberflächen gewährleisten (**Abb. II.32**). Wenn eine Nutzeroberfläche eine Zustandsänderung in der Funktionskomponente auslöst und die Aktualisierung über

die Polling-Variante realisiert wird, müssten entweder ständig Timer-Komponenten für die Aktualisierung der Zustände sorgen oder die Nutzeroberflächen müssten miteinander kommunizieren. Da sich der Informationsaustausch zwischen den Nutzeroberflächen aber nur auf das Anzeigen/ Verstecken der Fenster beschränken sollte, blieben allein die Timer-Komponenten mit einer zu großen Ressourcenverschwendung.

Sobald sich mehrere, u.U. verschiedene Oberflächen an einer Funktionskomponente anmelden können, müssen diese ein einheitliches Beobachter-Interface besitzen. Wenn Ereignisse nur durch bestimmte Oberflächen behandelt werden sollen, dann muss die Art des Ergebnisses in der Methode des Beobachter-Interfaces durch zusätzliche Parameter übertragen werden. Das Oberflächenfenster kann nach Auswertung dieser Parameter entscheiden, ob das Ereignis zu behandeln ist (siehe **[8]**, Creational Patterns –> Observer –> push model). Damit wäre für die Funktionskomponente nur ein Subjekt-Interface erforderlich. Der dort implementierte Verwaltungsmechanismus kann auch direkt in das Konkrete Subjekt integriert werden, ein Subjekt-Interface wird dann nicht benötigt.

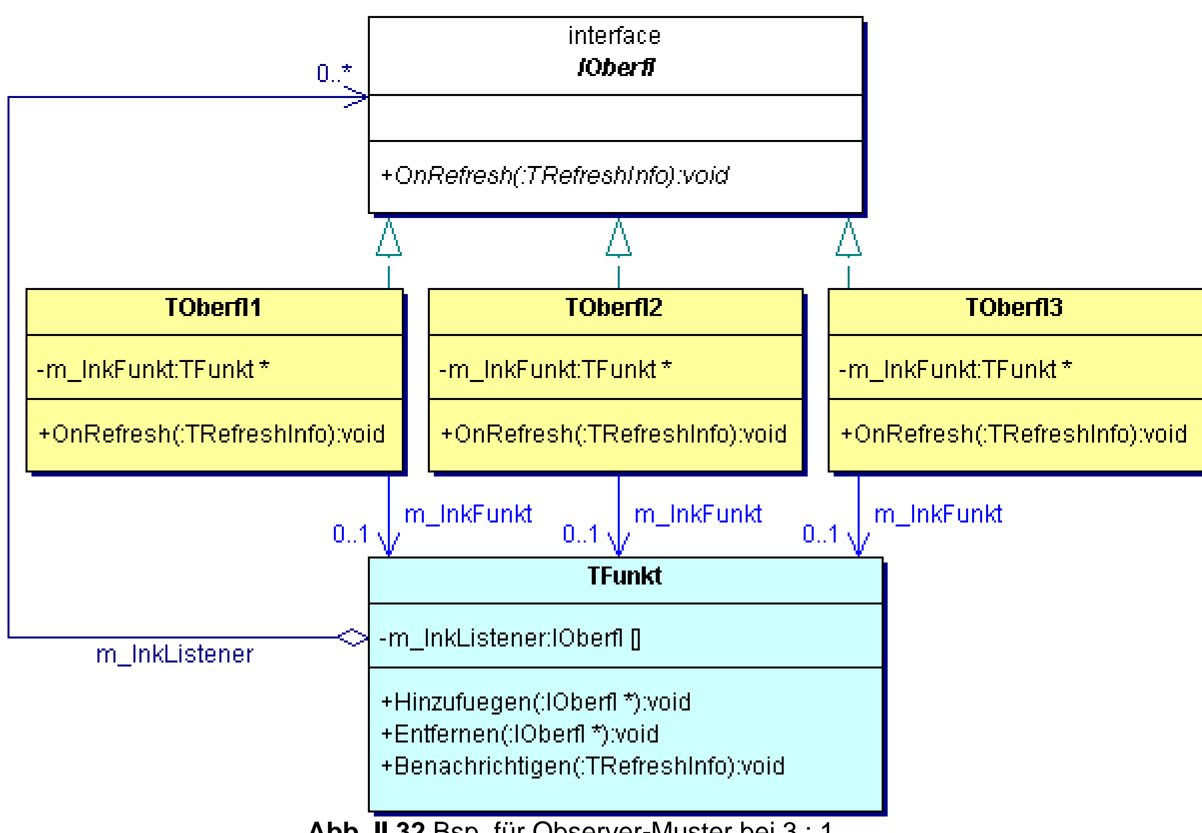


Abb. II.32 Bsp. für Observer-Muster bei 3 : 1

Im komplexesten Fall greifen mehrere Nutzeroberflächen derart auf mehrere Funktionskomponenten zu, dass die beiden vorher beschriebenen Varianten verknüpft zur Anwendung kommen (m : n). Auch hier ist das eigentliche Observer-Muster zu verwenden, da die Herstellung der horizontalen Konsistenz der Nutzeroberflächen höhere Priorität hat als die Verringerung des Umfangs des Programmcodes. Dies bedeutet, dass für jede einzelne Verknüpfung einer Funktionskomponente mit einer Nutzeroberfläche ein Beobachter-Interface in das Design zu integrieren ist (**Abb. II.33**).

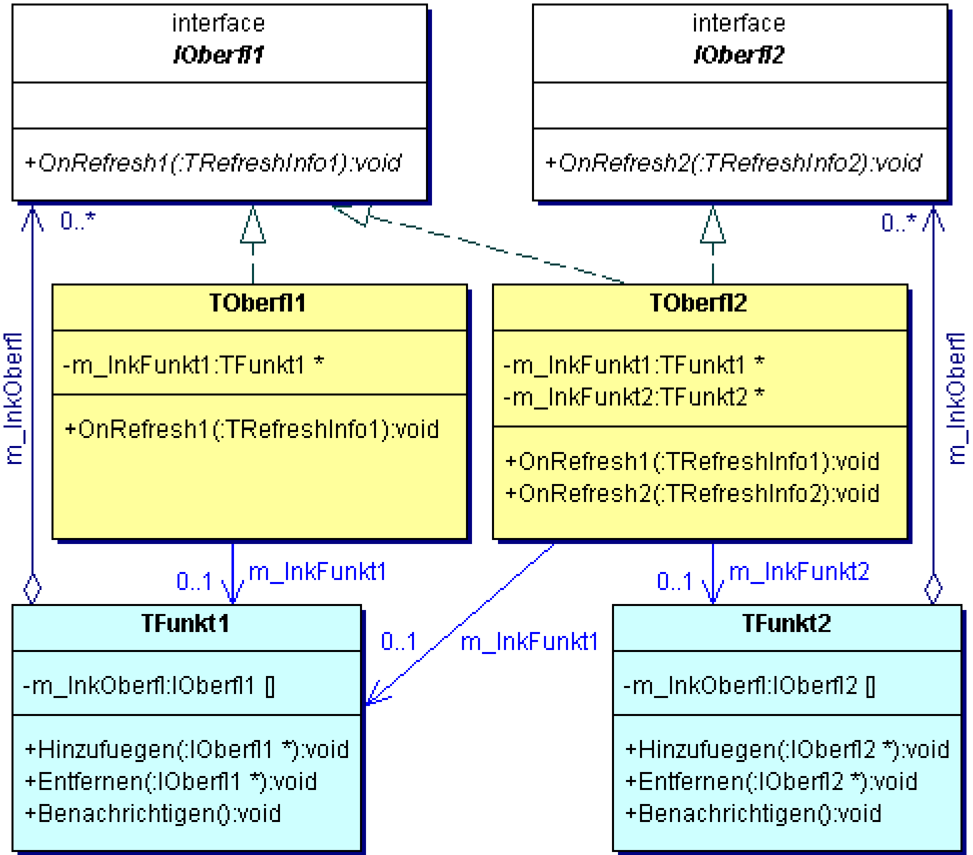


Abb. II.33 Bsp. für Observer-Muster bei 2 : 2

II.3.4 Aufbau von Designdokumenten

Um Missverständnisse zwischen dem OOD-Entwickler und dem Programmierer zu verringern, empfiehlt sich die Erstellung eines Designdokumentes. Dieses sollte alle, während der Designphase entstandenen, Diagramme (z.B. Klassen- oder Interaktionsdiagramme) und deren verbalen Beschreibungen zusammenfassen. So werden die Vorstellungen des OOD-Entwicklers über die Funktionen und Beziehungen von Klasse, Attributen und Methoden untereinander schriftlich festgehalten. Zudem führt die Erstellung eines solchen Dokumentes nochmals zu einer intensiven Beschäftigung mit der Materie, was Fehler und Unvollständigkeiten vor der Implementierungsphase aufdeckt. So könnte auch das bei einer späteren Wartungsphase anfallende, Reverse Engineering erheblich vereinfacht werden, weil neben der Programmcodekommentierung eine ausführliche Beschreibung des Gesamtsystems und seiner einzelnen Teile vorhanden ist.

Anwendungsfall: ‚Manuelle Justage'

Die Autoren haben für alle selbst erstellten Designdokumente ein einheitliches, stark formalisiertes Dokumentmuster entwickelt. Im ‚XCtl'-Projekt erfolgt die Erstellung getrennt nach Subsystemen. So entstanden für die ‚Ablaufsteuerung' **[M4]** und **[M6]**, für die ‚Motorsteuerung' **[M26]** und **[M28]**, für die ‚Benutzeroberfläche' **[M7]** und **[M9]**, für die neue ‚Topographie' **[M35]**, für die ursprüngliche **[M24]** und neue ‚Manuelle Justage' **[M21]**, jeweils nach dem gleichen Muster.

Die Gliederung eines Designdokumentes beginnt mit einer allgemeinen verbalen Einführung in den Gegenstandsbereich. Anschließend werden die verwendeten globalen Typen aufgelistet und erläutert. In C++ beinhaltet dies `struct`s, `enum`s, `union`s und `typedef`s – sofern

vorhanden. Darauf folgen Konstantendefinitionen, weil diese die oben erläuterten Typen benutzen könnten. In C++ sollte sich dies nur auf `const`-Konstrukte beziehen; `#define`-Direktiven sollten bekanntlich vermieden werden. Der Hauptteil des Dokumentes beinhaltet die Klassen. Dabei ist wichtig, die Beziehungen zwischen den Klassen zu erläutern, indem ein Klassendiagramm mit der kompletten Vererbungshierarchie und allen Beziehungstypen, aber ohne Methoden und Attribute, gezeigt wird. Liegt ein dekomponiertes Subsystem vor, werden zuerst die Klassen der Funktionskomponente behandelt und anschließend die der Nutzeroberfläche. Bei den letzteren befindet sich jeweils eine Übersicht über die Steuerelemente in den Dialogfenstern (in C++ die Gegenüberstellung der Ressourcen-Bezeichner und der Beschriftung). Abschließend folgen die globalen Variablen und Methoden.

Alle Typen, Attribute, Methoden und Variablen sind nach dem folgenden Schema formalisiert dargestellt (siehe **Abb. II.34**):

▶ `typedef enum { tb1, tb2, tb3 } ETB` `GLOBAL` `<N>`
Aufzählungstyp für die Adressierung der Teilbereiche
(zu deren Verwaltung siehe Methode `TB_Next`, `TB_Count`, `TB_GetId`, `TB_GetName`)

▶ `UINT m_MotorCount` `<ZSA>` `<N>`
Anzahl der Motoren in `m_MotorList`

▶ `UINT GetMotorCount( void )` `<ZSM>` `<N>`
gibt die aktuelle Anzahl der Antriebe in der Antriebsliste zurück (siehe `m_MotorCount`)

▶ `TSteering Steering` `<SCS>` `GLOBAL` `<N>`
Objekt für den Zugriff auf die Makrosteuerung

Legende:
`<N>` ≡ kennzeichnet neu eingefügte Programmcodeelemente beim Reengineering:
„`NEU`” | „”
`<ZSA>` ≡ Zugriffsschutz für Attribute:
„`private`“ | „`protected`“ | „`public`“
`<ZSM>` ≡ Zugriffsschutz für Methoden:
„`GLOBAL`“ | `<ZSA>`
`<SCS>` ≡ storage-class-specifier:
„`auto`“ | „`register`“ | „`static`“ | „`extern`“ | „`volatile`“ | „`mutable`”

Abb. II.34 Verallgemeinerte Einträge für Typen, Attribute, Methoden und Variablen (vgl. oben), nach Auszügen aus **[M21]**

Jedem Eintrag (▶) ist eine kurze Beschreibung hinzugefügt. Neben einem allgemeinen Zweck sind bei den Methoden alle Parameter erläutert und bei den Typendeklarationen alle Elemente. Jeder `Do`-Methode ist, sofern möglich, ein **/ F** `<ff>` **/** -Eintrag aus dem Pflichtenheft zugeordnet. Um zu überprüfen, ob alle Einträge im Design berücksichtigt werden, ist bei jeder Methode der entsprechende Eintrag genannt (siehe **Abb. II.35**).

Diese Zuordnung ist auch für Attribute und Datenteil-Einträge sowie zwischen Botschaftenbehandlungsroutinen und den Funktionen der Benutzeroberfläche vorgenommen worden.

▶ `BOOL DoStop( int )` PUBLIC
stoppt die Antriebsbewegung, unabhängig von der Betriebsart, der Parameter ist der Index des Antriebs in der Antriebsliste
(siehe [M8], Funktion: **/ F 80 /**)

Abb. II.35 Auszug für einen Methoden-Eintrag aus **[M21]**

II.3.5 Zusammenfassung

Obwohl im Falle der neuen ‚Manuellen Justage' mehrere Dialogfenster auf die Funktionskomponente zugreifen, konnte dennoch die Polling-Variante gewählt werden. Dies war möglich, weil alle Oberflächenklassen modale Dialogfenster repräsentieren und damit nicht gleichzeitig auf die Funktionskomponente zugreifen können. Die horizontale Konsistenz muss so nur beim Verlassen eines Dialogfensters beachtet werden. Zum Vergleich wurden beide Varianten, Observer-Muster und Polling-Variante, entworfen und später implementiert.

Zu diesem Zeitpunkt ist nur ein marginaler Unterschied zwischen Observer-Muster und der Polling-Variante ersichtlich. Er beschränkt sich fast ausschließlich auf die Kommunikation zwischen Funktionskomponente und Nutzeroberfläche; vgl. **Abb. II.37** (Observer) und **Abb. II.36** (Polling).

Die Erweiterung des OOA-Modells zu einem OOD-Modell gestaltete sich unproblematisch. Hier zeigten sich klar die Vorzüge der Objektorientierung. Verursachte die Dekomposition bereits im OOA-Modell eine erhöhte Komplexität, so hat sich dieser Trend leider auch in der Designphase fortgesetzt. Bei einem Vergleich von **Abb. II.19** (OOA) und **Abb. II.22** (OOD) ist besonders auffällig, dass ein Großteil der Attribute entfernt wurde. Wie bei **II.3.1** erwähnt, konnte deren Verwaltung in den benutzten Subsystemen identifiziert werden. Die für den Zugriff geplanten Methoden sind erhalten geblieben; statt Accessor-Methoden, die direkt auf die Attribute zugreifen, wird nun an das jeweilige Subsystem delegiert.

Dadurch kann man am Beispiel der neuen ‚Manuellen Justage' aber schlecht erkennen, dass die Funktionskomponente die Datenverwaltung[17] übernimmt; jedenfalls kann man dazu nicht nur **NOA** heranziehen. Erst wenn man anstelle der ersten beiden Zeilen aus **Tab. II.5** bzw. (die ersten drei Zeilen aus **Tab. II.7**) die im OOA geplante Funktionskomponente (**Tab. II.3**) verwendet, liegt die Verwaltung der Daten wieder eindeutig bei der Funktionskomponente; insgesamt verwaltete Daten 21 (Funktionskomponente, Polling-Variante) zu 12 (nutzeroberflächenspezifisch).

Des Weiteren fehlen einige Methoden in **Abb. II.22**, die für das Auslesen des Wertebereichs zuständig sind (z.B. `GetChannelMin`). Da die `Set`-Methoden den ihnen übergebenen Wert selbständig berichtigen und den ggf. korrigierten Wert zurückgeben (siehe **II.3.1 b)**), werden solche Methoden nicht benötigt. Die Oberfläche hat keine Kenntnis über den Wertbereich.

In einer deutlich ausgeprägten Schichtenstruktur sollten alle Objekte einer Schicht nur mit der direkt darüber- oder darunter liegenden Schicht kommunizieren dürfen. In einem nicht dekomponierten Subsystem, das nur aus der Nutzeroberfläche besteht, tauscht die Oberfläche daher direkt mit dem "niederen" Subsystem zur Hardwareansteuerung (▭) Daten aus. Bei einem dekomponierten System (**Abb. I.7**) erfolgt die Kommunikation durch zwischen Nutzeroberfläche (▭) und "niederem" Subsystem (▭) jedoch über die Funktionskomponente (▭)! Deshalb entsteht hier ein gewisses Methoden-Overhead (siehe **Abb. II.22**, ▭), dass nur der Delegation von Funktionsaufrufen zwischen den beiden Subsystemen dient.

---

[17] diese Eigenschaft sollte i.d.R. ein gut dekomponiertes Subsystem kennzeichnen

Bei dem Vergleich der Metriken von Funktionskomponente und Nutzeroberfläche (Klassen mit Suffix `Dlg`) sticht sofort der hohe Anteil an öffentlichen Methoden (**PPubM**) bei der Funktionskomponente ins Auge. Nur so steht der Nutzeroberfläche die gesamte Funktionalität zur Verfügung. Zusammen mit der Menge an angebotener Funktionalität (**NOO**) garantiert dies aber die hohe Wiederverwendbarkeit der Funktionskomponente. Bei der neuen ‚Manuellen Justage' wird diese von drei Nutzeroberflächen gleichzeitig genutzt. Sehr wichtig ist hierbei die Einhaltung der in diesem Kapitel vorgestellten Prinzipien wie: zentrale Verwaltung von Wertebereichen und Nachkommastellengenauigkeit sowie die Verwendung von `Can`- in `Do`- und `Set`-Methoden. Damit gewinnt die Funktionskomponente eine hohe Robustheit.

Bei **II.3.2** wurden bereits die minimalen öffentlichen Schnittstellen der Oberflächenfenster erwähnt, die sich nun deutlich in den sehr niedrigen **PPubM** der Dialogfensterklassen zeigen. Dieses Vorgehen sichert die horizontale Konsistenz zwischen den Oberflächenfenstern und den Funktionskomponenten.

Aufgrund des mangelnden Platzangebotes kann die neue ‚Manuelle Justage' hier nicht vollständig mit Attributen und Methoden abgebildet werden. Das komplette Design kann jedoch leicht selbst durch die Abbildungen **Abb. II.22** (für `TManJustage` und `TMotorData`), **Abb. II.23** (für `TManJustageDlg`) und **Abb. II.24** (für `TMotorOffsetDlg` und `TPsdOffsetDlg`) vervollständigt werden.

II.3.5–1 Bewertung von Observer-Muster und Polling im Anwendungsfall

Beim Vergleich des Observer-Musters (**Abb. II.37**) mit der Polling-Variante (**Abb. II.36**) fällt sofort die observer-typische Kommunikationsschnittstelle `IManJustageDlg` ins Auge. Obwohl das UML-Klassendiagramm damit wesentlich umfangreicher ist, werden dafür insgesamt etwas weniger Attribute, **NOA** 24 (Observer) zu 26 (Polling) benötigt. Die Methodenanzahl bleibt gleich (**NOO** 123). Der Grund ist, dass bei der Polling-Variante Zustandsänderungen durch die Nutzeroberfläche kompliziert ermittelt werden müssen. Dazu liest die Nutzeroberfläche den aktuellen Zustand von Attributen der Funktionskomponente aus und vergleicht diese mit zwischengespeicherten Werten. Dazu sind zusätzliche Attribute nötig.

Beim Observer-Muster informiert die Funktionskomponente als Auslöser der Zustandsänderungen sofort die Nutzeroberfläche.

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TManJustage	–	6	67	1	0	11	0	89	46	3	2	–
↳ TMotorData	–	8	0	0	0	0	0	100	66	0	1	–
TManJustageDlg	–	6	48	1	5	88	9	4	54	6	1	–
TMotorOffsetDlg	–	3	5	1	2	67	22	11	27	4	1	–
TPsdOffsetDlg	–	3	3	1	2	57	29	14	27	4	1	–

Tab. II.5 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Polling-Variante am Ende der Designphase)

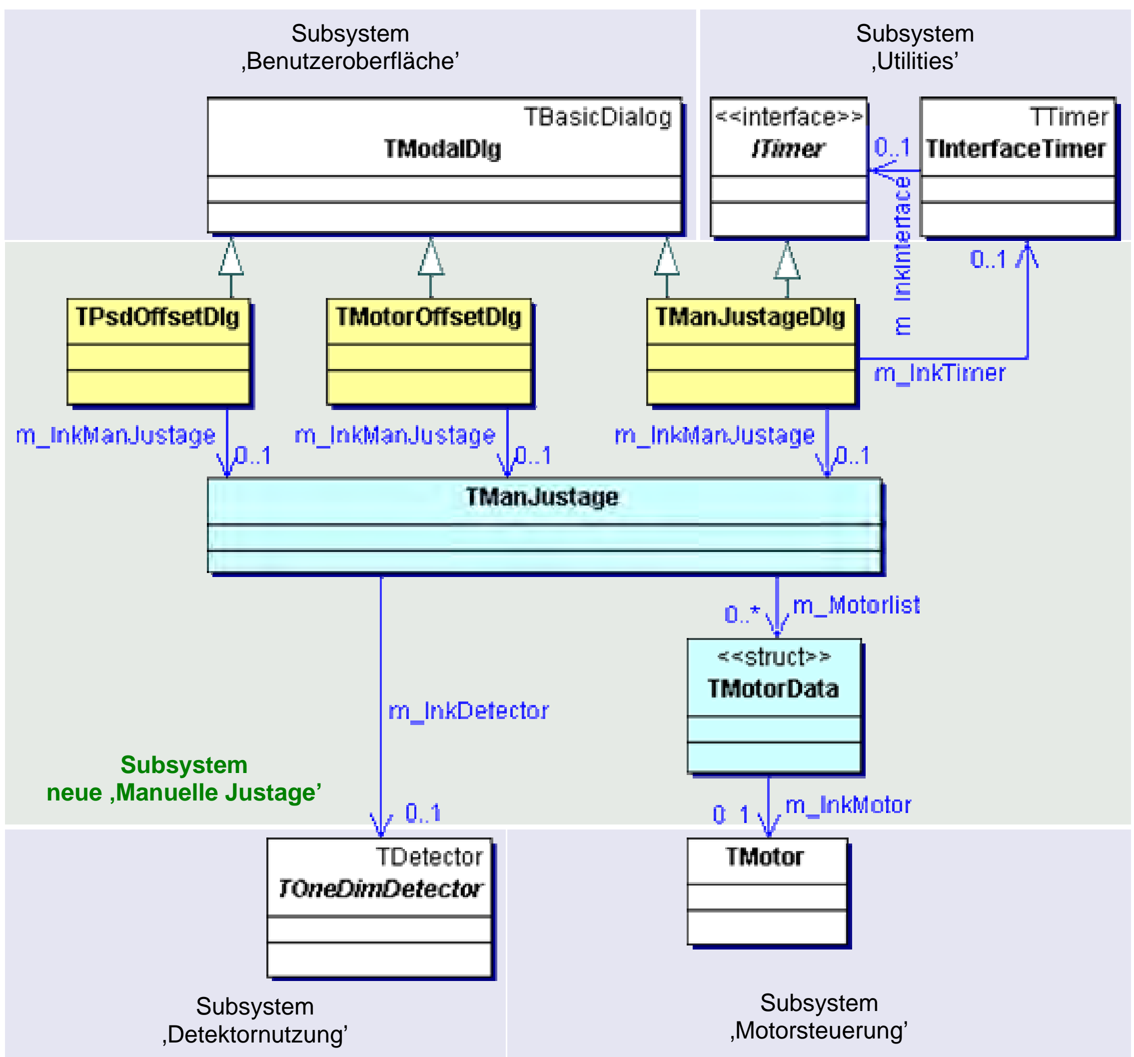


Abb. II.36 UML-Klassendiagramm für das OOD-Modell der neuen ‚Manuellen Justage' mit Polling-Variante und Übersicht der benutzten Subsysteme

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
IManJustageDlg	–	0	6	0	0	0	0	100	0	1	1	–
TManJustage	–	6	61	1	0	9	1	90	46	3	2	–
↳ TMotorData	–	7	0	0	0	0	0	100	57	0	1	–
TManJustageDlg	–	5	48	1	10	78	18	1	45	6	1	–
TMotorOffsetDlg	unverändert zu Tab. II.5											
TPsdOffsetDlg												

Tab. II.6 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Observer-Muster am Ende der Designphase)

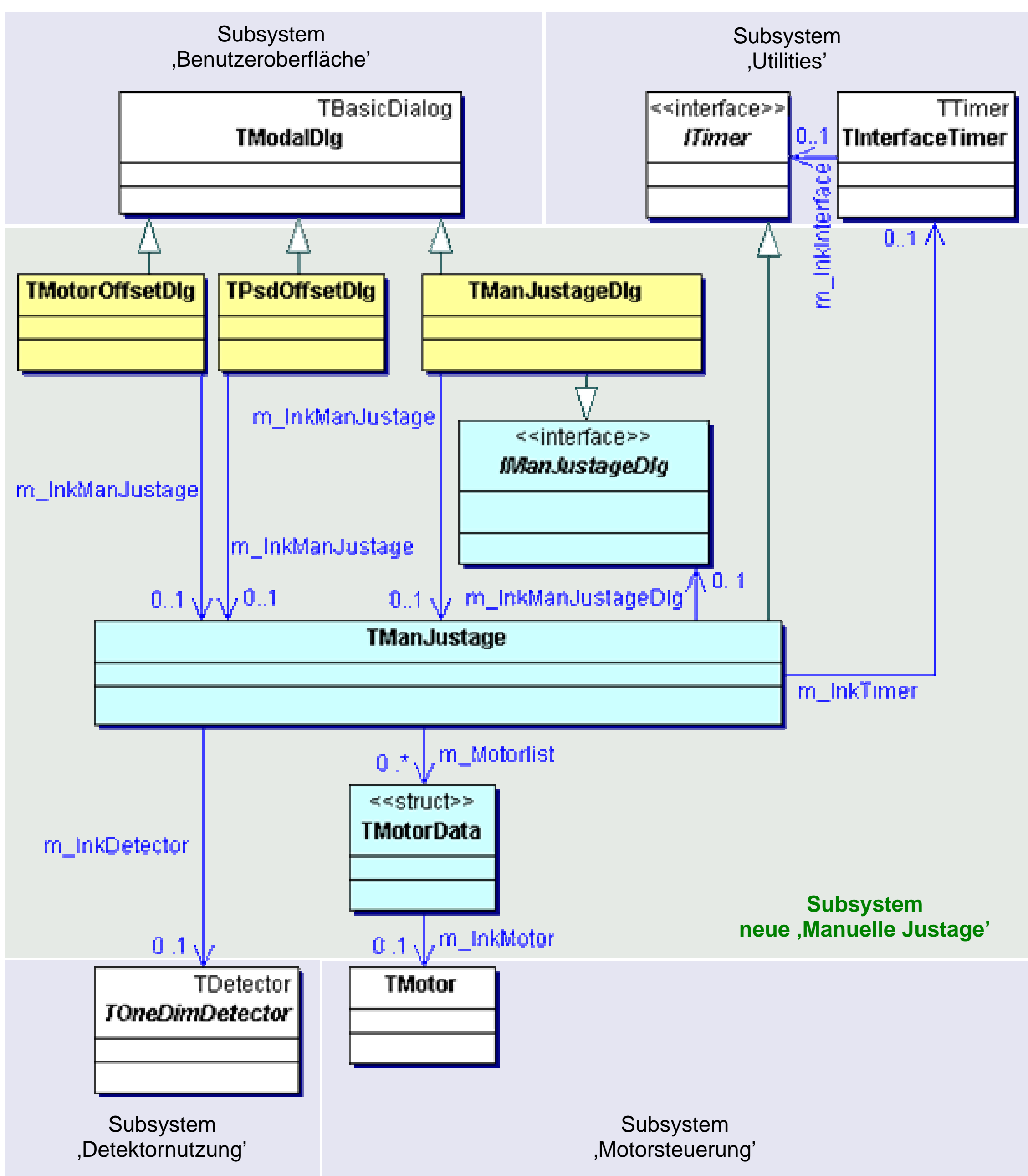


Abb. II.37 UML-Klassendiagramm für das OOD-Modell der neuen ‚Manuellen Justage' mit Observer-Muster und Übersicht der benutzten Subsysteme

II.4 Implementierungsphase

Nach der Erstellung des OOD-Modells sind die wichtigsten Aspekte der Dekomposition bereits abgearbeitet. Das OOD enthält nun die Richtlinien für die Implementierungsphase.

Bei der Implementierung sollte zuerst die Benennung der Dateien beachtet werden. Dabei ist sowohl die Zugehörigkeit zum Subsystem als auch die Unterscheidung in Funktionalität und Oberfläche zu kennzeichnen. In C++ sind deshalb für Funktionskomponente(n) und Nutzeroberfläche(n) je ein Dateipärchen, bestehend aus Header- und cpp-Datei, zu erstellen.

Die beiden aus der Designphase bekannten Kommunikationsmuster Observer-Muster und Polling-Variante wurden zum Vergleich auch in der Implementierungsphase weiterentwickelt. Da sich die anzuwendende Schrittfolge jedoch nicht unterscheidet, sind die Beispiele in diesem Kapitel aus der Polling-Variante. Die Unterschiede zwischen Observer-Muster und Polling-Variante werden abschließend in Kapitel **II.4.3–1** verglichen.

II.4.1 Schwerpunkte bei der Funktionskomponente

Die Grundlage für die Programmierung der Funktionskomponente bildet das in der Definitionsphase erstellte OOA- und in der Designphase verfeinerte OOD-Modell. Wurden das zugehörige Designdokument und das Pflichtenheft nach der in Kapitel beschriebenen Weise erstellt, wird die Implementierung der vollständigen Funktionalität erleichtert. Die Anforderungen an Methoden und Eigenschaften von Attributen sind bereits im Designdokument festgelegt. Durch zusätzliche Anmerkungen, z.B. über die in Methoden zu benutzenden Attribute, ist dem Programmierer damit vorgeschrieben wie und was zu implementieren ist. So wird verhindert, dass Abweichungen von der ursprünglichen Strategie/ Idee für die Funktionsweise einer Methode entstehen, zumal gerade bei umfangreichen Softwareprojekten einige Zeit zwischen dem Design und der eigentlichen Codierung einer Operation vergehen kann.

In der Funktionskomponente sollte beim Zugriff auf andere Subsysteme, besonders bei Hardwarezugriffen, deren Verfügbarkeit überprüft werden. Ziel ist im Fehlerfall einen gültigen Zustand zu hinterlassen, so dass die Aufrufumgebung über die öffentliche Schnittstelle informiert werden kann. Auf diese Weise kann die gesamte Fehlerbenachrichtigung in der Nutzeroberfläche implementiert werden. Diese gibt dann ggf. Fehlermeldungen, etc. aus, denn das darf nicht Aufgabe der Funktionskomponente sein! Durch die zusätzliche Kommunikation ist aber stets eine höhere Ressourcenbelastung, insbesondere der Ausführungsgeschwindigkeit, verbunden.

Auch funktionskomponenten-interne Zugriffe auf Attribute sollten zur Verbesserung der Robustheit über die dazugehörigen Accessor- und Mutator-Methoden durchgeführt werden. Die Fehlerüberprüfung bzw. -stabilisierung muss so nur dort implementiert werden. Dies gilt besonders für Schreibzugriffe, weil diese i.d.R. von Bedingungen abhängen. Eine unter diesen Prämissen erstellte Funktionskomponente sichert Robustheit, hohe Benutzerfreundlichkeit und somit Wiederverwendbarkeit des Subsystems.

Anwendungsfall: ‚Manuelle Justage'

Bei der Realisierung der neuen ‚Manuellen Justage' wurde die Funktionskomponente `TManJustage` in das Dateipärchen MJ_FUNK.H und MJ_FUNK.CPP implementiert. Durch das ausführliche Pflichtenheft und das Designdokument war für jede geplante Methode die vollständige Signatur und eine kurze Beschreibung vorhanden. Wo die Zuordnung der

Methoden und Attribute zu Produktanforderungen möglich war, wurde diese als weiterführende Informationsquelle im Designdokument angegeben (siehe **Abb. II.38**),).

Damit war z.B. eindeutig erkennbar, welche Aspekte der Pflichtenheftfunktion **/ F 70 /** in die Methode `DoDirect` zu integrieren waren (siehe **Abb. II.39**).

▸ `BOOL DoDirect( const int )` `PUBLIC`
löst Antriebsbewegung im Direktbetrieb aus; der erste Parameter ist der Index des Antriebs in der Antriebsliste, setzt `m_IsMoving`
(siehe **[M8]**, Funktion: **/ F 70 /**)

Abb. II.38 Auszug für einen Methoden-Eintrag aus dem Designdokument **[M21]**

/ F 70 / – „Starten der Bewegung im Direktbetrieb“
Nach der Beschleunigungsphase bewegt sich der Antrieb mit der angegebenen *Geschwindigkeit*, um kurz vor dem Ziel abzubremsen und an der *Sollposition* zu halten. Wenn die *Sollposition* größer (bzw. kleiner) ist als die *Istposition*, bewegt sich der Antrieb vorwärts (bzw. rückwärts). Die maximale Abweichung zwischen der angefahrenen und der gewünschten Sollposition darf maximal DeathBand[1] betragen. Wenn *Ist*- und *Sollposition* übereinstimmen, findet keine Bewegung statt. Die Genauigkeit beträgt hier wiederum DeathBand.

Bedingung
Der Antrieb darf sich nicht bewegen. Wenn eine *Sollposition* eingegeben wird, die außerhalb des zulässigen Wertebereichs liegt, so wird die Positionsangabe auf die minimal oder maximal zulässige Position korrigiert, je nachdem, ob der zulässige Wertebereich unterschritten oder überschritten wurde.

Abb. II.39 Auszug aus dem Funktionsteil des Pflichtenheftes **[M15]**

Durch die in **Abb. II.39** angegebene Bedingung war im OOA eine zugehörige `CanDo`-Methode geplant worden, die nun als `CanDoDirect` implementiert wurde (siehe **Abb. II.40**). Durch die Eindeutigkeit und Einfachheit der Bedingung konnte im Falle der neuen ‚Manuellen Justage’ auf Entscheidungstabellen verzichtet werden. Bei komplexeren Bedingungen sollte dieses Basiskonzept jedoch als Implementierungshilfe genutzt werden. Es muss dann auch als Teil des OOA-Modells in die Vertragsverhandlungen aufgenommen werden.

Nachdem die Ausführung der `Do`-Methode gesichert war (`CanDo`-M.), mussten die Methoden zur Realisierung der eigentlichen Funktionalität in den übrigen Subsystemen gesucht werden. In den Designdokumenten der Subsysteme ‚Motorsteuerung’, ‚Ablaufsteuerung’ und ‚Detektornutzung’ konnten die benötigten Methoden einfach identifiziert werden. Im Falle von **/ F 70 /** (`DoDirect`) waren dies nur Methoden der ‚Motorsteuerung’, im Wesentlichen zur Bewegung eines Antriebs zu einer neuen Antriebsposition (siehe **Abb. II.41**).

Es erfolgt nun die Implementierung der eigentlichen `Do`- und `Set`-Methode. Dazu wird anfangs die dazugehörige `Can`-Methode aufgerufen, die bei Nichterfüllung zur vorzeitigen Beendigung der `Do`- bzw. `Set`-Methode führt. Danach werden die identifizierten Methoden aus den Subsystemen aufgerufen, die u.U. auch zu einer vorzeitigen Beendigung der Methode führen können. Nur bei vollständiger und erfolgreicher Abarbeitung ist der Rückgabewert am Ende `TRUE`, sonst immer `FALSE` (siehe **Abb. II.42**).

```
// ist Direktbetrieb möglich ?
BOOL  TManJustage::CanDoDirect( const int aIndex ) const {
  BOOL bValid= FALSE;
  // TRUE <-> gültiger Antrieb, nicht in Bewegung (d.h. auch Halbwertsbreitenmessung ist inaktiv)
  if ( (IsMoving(aIndex, bValid)) || // inclusive IsIndexValid
       (!bValid) ) return FALSE;
  TMotorData *MtrData= &m_MotorList[aIndex];
  return ( MtrData->m_MotionType==mtDirect ); // TRUE <--> im Direktbetrieb
};
```

Abb. II.40 Implementierung der Bedingung für den Direktbetrieb, Methode `CanDoDirect`

- `BOOL  MoveToAngle( double )` PUBLIC

übersetzt den Parameter (*Absolutposition* in Nutzereinheiten) in eine *interne Antriebsposition* (Methode `Translate`) und fährt diese an (Methode `MoveToPosition`);

`mlMoveToDistance` und `mMoveToDistance`

Abb. II.41 Auszug für einen Methoden-Eintrag aus dem Designdokument **[M28]** des Subsystems ‚Motorsteuerung'

```
// Bewegung im Direktbetrieb
BOOL  TManJustage::DoDirect( const int aIndex ) {
  if ( !CanDoDirect(aIndex) ) return FALSE; // Direktbetrieb kann nicht durchgeführt werden

  TMotor *Mtr= m_MotorList[aIndex].m_lnkMotor; // Motor aus Motorenliste auswählen
  BOOL bValid= FALSE;
  Mtr->SetSpeed( GetSpeed(aIndex, bValid) ); // akt. Geschw. setzen
  Mtr->ActivateDrive();                      // Antrieb aktivieren
 // anfahren der Sollposition durch Funktion des Subsystems ‚Motorsteuerung'
  if ( (!bValid) || // Geschwindigkeit konnte nicht ermittelt  werden
       (!Mtr->MoveToAngle( GetAngleDest(aIndex, bValid))) ) return FALSE;

  return TRUE; // Bewegung erfolgreich gestartet
};
```

Abb. II.42 Fortsetzung der Implementierung des Direktbetriebs, Methode `DoDirect` – unter Benutzung von `CanDoDirect` und Methoden des Subsystems ‚Motorsteuerung'

Vor jedem Zugriff auf andere Subsysteme wird dessen Verfügbarkeit/ Zustand geprüft. Dazu wurden Methoden implementiert, die vorher den Status des gerufenen Systems (z.B. für das Vorhandensein eines Detektors `HasDetectorAxis`) oder die Gültigkeit eines (aus der Nutzeroberfläche übergebenen) Parameters überprüfen, z.B. `IsIndexValid` (siehe **Abb. II.43**).

Dazu wird bei `Is`-, `Has`-, `Do`- und `Set`-Methoden wie bei den `Can`-Methoden die Aufrufumgebung über den Fehlschlag (mit `FALSE`) bzw. über die erfolgreiche Durchführung einer Operation (mit `TRUE`) informiert. Da der Rückgabewert bei `Get`- und `Calc`-Methoden anderweitig verwendet wird, erfolgt die Rückgabe über einen zusätzlichen Referenzparameter (siehe **Abb. II.44** `aValid`). Im Fehlerfall erhalten diese Methoden einen Standardwert als Rückgabewert (hier `return ""`), falls die Aufrufumgebung das Auslesen des Gültigkeits-Parameters übergeht und den Rückgabewert wie gewohnt verwendet. Dieses Verfahren sichert, dass der Entwickler der Aufrufumgebung (bei der Dekomposition die Nutzeroberfläche) über eine Problemsituation informiert wird und entsprechend drauf reagieren kann, z.B. eine Fehlermeldung ausgibt.

```
// 1dimensionaler Detector (PSD) und dessen Antrieb vorhanden ?
BOOL  TManJustage::HasDetectorAxis () const {
  return ( (m_lnkDetector!=0) && (GetDetectorAxisIdx()!=-1) );
};

// Test ob Index im Intervall der Motorenliste
BOOL  TManJustage::IsIndexValid ( const int aIndex ) const {
  return ( (0 <= aIndex) && (aIndex < GetMotorCount()) );
};
```

Abb. II.43 Absicherung von Lese- und Schreibzugriffen und zur Überprüfung von Parameterübergaben in `TManJustage`

```
// Name des Antriebs für einen Index aus der Antriebsliste
LPCSTR  TManJustage::GetMotorName ( const int aIndex, BOOL &aValid ) const {
  aValid= FALSE;
  if ( !IsIndexValid(aIndex) ) return ""; // bei ungültigem Index: leere Zeichenkette

  aValid= TRUE;
  return m_MotorList[aIndex].m_lnkMotor->pCharacteristic(); // der Name
};
```

Abb. II.44 Überprüfung, ob der übergebene Parameter gültig ist

Ähnlich wie beim Zugriff auf andere Subsysteme unterliegt auch der Zugriff auf Attribute der Funktionskomponente selbst bestimmten Bedingungen. Dazu gehört bei der ‚Manuellen Justage', z.B. ob sich ein Antrieb gerade bewegt, welche Betriebsart gerade ausgewählt ist oder ob die aktuell angezeigte Antriebsposition durch ein nutzerdefiniertes Offset von der absoluten Position abweicht. Deshalb wurden, anstatt direkt auf die Attribute zuzugreifen, Accesor- und Mutatormethoden verwendet. In **Abb. II.45** wird vor dem Setzen des Kanal-Offsets geprüft, ob das Offset vorher erfolgreich ermittelt werden konnte (siehe `aValid`).

```
// setzen eines neuen aktuellen Kanals
BOOL  TManJustage::SetChannel ( int &aChannel ) {
  BOOL bValid;
  if ( !HasDetectorAxis() ) {
    m_Channel= 0;
    return FALSE;
  }

  int idx= GetDetectorAxisIdx();
  aChannel= max( m_lnkDetector->GetFirstChannel(),
             min( aChannel, m_lnkDetector->GetLastChannel() ) ); // Anpassung
  ResetChannelOffset(idx); // alten Kanaloffset zuerst entfernen
  TAngle NewOffset= GetOffset(idx, bValid) + CalcChannelOffset(aChannel);
  if ( (!bValid) || (!SetOffset(idx, NewOffset)) )
    return FALSE; // Offset konnte nicht gelesen oder gesetzt werden

  m_Channel= aChannel; // erst nach SetOffset, weil m_Channel dort auf FirstChannel gesetzt wird
  IniWriteLong( GetHWFile(), m_DetectorId, "MJ_Channel", m_Channel );
  return TRUE;
};
```

Abb. II.45 Nutzung der Mutator-Methode `SetOffset` anstatt eines Direkt-Zugriffs in der Funktionskomponente `TManJustage`

Wie bei **II.3.1** gefordert, wurden alle nicht-zustandsverändernden Methoden als const-member-functions deklariert. Bei der neuen ‚Manuellen Justage' sind dies alle `Is`-, `Has`-, `Can`-, `Calc`- und `Get`-Methoden des OOD-Modells. Sämtliche Parameter, die keine Referenz darstellen, wurden mit `const` gekennzeichnet.

TManJustage

-m_MotorList:TMotorData *
-m_InkDetector:TOneDimDetector *
-m_Channel:int
-m_MotorCount:UINT
-m_DetectorId:char [50]
-m_IsMeasuring:BOOL
-m_KzMeasuring:BOOL
-m_UpdateCount:UINT

0..1
m_Motorlist

<<struct>>
TMotorData

-m_InkMotor:TMotor *
-m_AxisType:TAxisType
-m_MotorId:char [512]
-m_MotionType:EMotionType
-m_AngleDest:TAngle
-m_Speed:TSpeed
-m_AngleWidth:TAngle
-m_KzMoving:BOOL
-m_SpAngle:TAngle
-m_HasAngle:BOOL
-m_SpMoving:BOOL
-m_HasMoving:BOOL

TManJustage

+GetChannel():int
+GetAnglePerChannel():TAngle
+GetMotorName(:int,:BOOL &):LPCSTR
+GetUnit(:int,:BOOL &):LPCSTR
+GetAngle(:int,:BOOL &):TAngle
+GetAngleMin(:int,:BOOL &):TAngle
+GetAngleMax(:int,:BOOL &):TAngle
+GetDigits(:int,:BOOL &):UINT
+GetOffset(:int,:BOOL &):TAngle
+GetMotionType(:int,:BOOL &):EMotionType
+GetAngleDest(:int,:BOOL &):TAngle
+GetSpeed(:int,:BOOL &):TSpeed
+GetAngleWidth(:int,:BOOL &):TAngle
+GetDetector():TDetector *
+SetMotionType(:int,:EMotionType &):BOOL
+SetAngleDest(:int,:TAngle &):BOOL
+SetSpeed(:int,:TSpeed &):BOOL
+SetAngleWidth(:int,:TAngle &):BOOL
+CanSetMotionType(:int):BOOL
+CanSetAngleDest(:int):BOOL
+CanSetSpeed(:int):BOOL
+CanSetAngleWidth(:int):BOOL

Attribute; **NEU: 4 (links) + 5 (rechts)**
Accessor- und Mutatormethoden
Methoden zur Realisierung der Funktionalität
Kapselung fremder Funktionen
weitere Methoden; **NEU: 10**

TManJustage

+IsMoving(:int,:BOOL &):BOOL
+IsCalibrated(:int,:BOOL &):BOOL
+ParsingAxis(:LPCSTR):UINT
+GetDetectorAxisIdx():int
+GetDetectorName(:BOOL &):LPCSTR
+HasDetectorAxis():BOOL
+IsMeasuring():BOOL
+IsMeasureReset():BOOL
+GetMeasureProgress(:int &,:LPSTR,:int &):BOOL
+GetMeasureHwb():double
+UpdateDetector():void

TManJustage

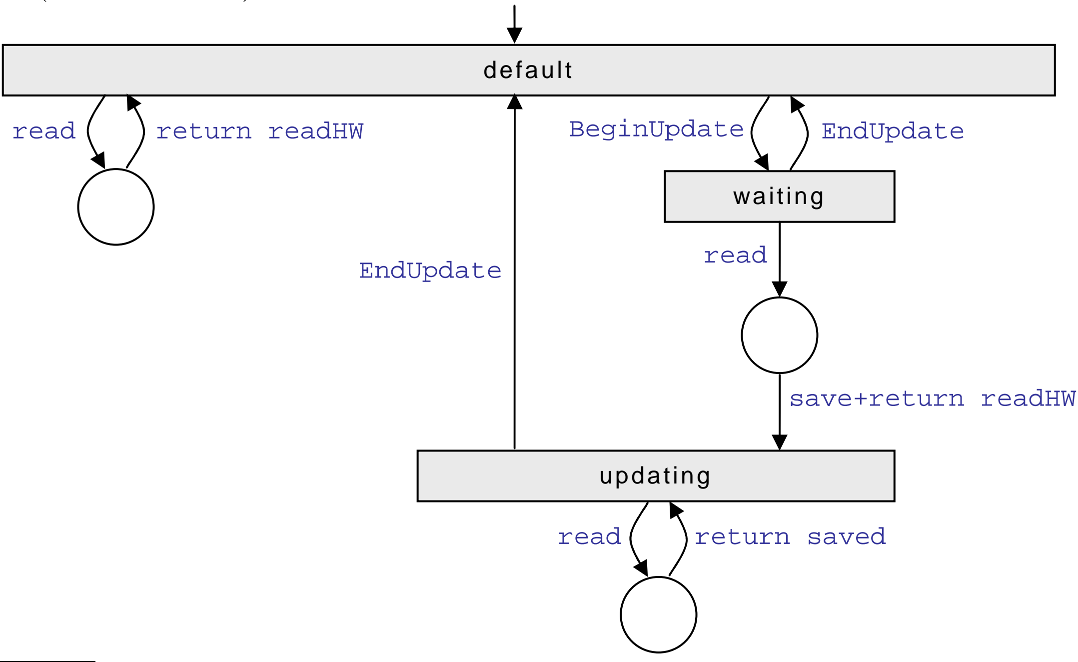
+GetMotorCount():UINT
+SetRelativeZero(:int):BOOL
+ResetRelativeZero(:int):BOOL
+SetOffset(:int,:TAngle &):BOOL
+DoDirect(:int):BOOL
+DoDrive(:int,:EDirection):BOOL
+DoStep(:int,:EDirection):BOOL
+DoStop(:int):BOOL
+DoStartMeasure(:HWND):BOOL
+DoStopMeasure():BOOL
+SetChannel(:int &):BOOL
+CanResetRelativeZero(:int):BOOL
+CanSetOffset(:int):BOOL
+CanDoDirect(:int):BOOL
+CanDoDrive(:int,:EDirection):BOOL
+CanDoStep(:int,:EDirection):BOOL
+CanDoStop(:int):BOOL
+CanDoStartMeasure():BOOL
+CanDoStopMeasure():BOOL

TManJustage

+TManJustage()
+~TManJustage()
+IsIndexValid(:int):BOOL
+HasOffset(:int,:BOOL &):BOOL
+GetMotorIdx(:LPCSTR):int
+GetMotorIdx(:TAxisType):int
+CalcOffsetFromAngle(:int,:TAngle &):TAngle
+CalcAngleFromOffset(:int,:TAngle &):TAngle
+CalcChannelOffset(:TChannel &):TAngle
+CalcChannelAngle(:TChannel &):TAngle
+DoStopEverything():void
+GetKzMoving(:const int,:BOOL &):BOOL
+SetKzMoving(:const int,BOOL:const):BOOL
+GetKzMeasuring():BOOL
+SetKzMeasuring(BOOL:const):void
+ResetChannelOffset(:const int):BOOL
-BeginUpdate():UINT
-EndUpdate():UINT
-DetermineAngle(:const int,:BOOL &):TAngle
-DetermineMoving(:const int,:BOOL &):BOOL

Abb. II.46 UML-Klassendiagramm für die Funktionskomponente der neuen ‚Manuelle Justage' (am Ende der Implementierungsphase)

Beim Abnahmetest an den Messplätzen mit realer Hardware traten Probleme mit der Ausführungsgeschwindigkeit auf. Schnell konnten diese auf den Zugriff der dort installierten Motorcontrollerkarten zurückgeführt werden. Weil sich die Nutzeroberfläche an jede Zustandsänderung anpassen sollte, wurden die Steuerelemente regelmäßig aktualisiert. Dazu wurde jeweils auf die Hardware zugegriffen, weil diese Zustand und Inhalt der Steuerelemente mitbestimmt. Der Schwerpunkt der Optimierung liegt somit bei der Nutzeroberfläche. Zu deren Unterstützung wurde ein zusätzlicher Mechanismus in die Funktionskomponente integriert, der durch die Nutzeroberfläche aktiviert (`BeginUpdate`) bzw. deaktiviert (`EndUpdate`) werden kann (siehe **Abb. II.47**).



default
bei **read** wird die Hardware ausgelesen und dieses Ergebnis zurückgegeben (Anfangszustand)

waiting
bei **read** wird die Hardware auslesen, das Ergebnis wird zwischengespeichert und zurückgegeben

updating
bei **read** wird der zwischengespeicherte Wert zurückgegeben, d.h. kein Hardwarezugriff !

Abb. II.47 Stark vereinfachtes Konzept zur Geschwindigkeitsoptimierung

So kann die Nutzeroberfläche durch `BeginUpdate` kennzeichnen, dass viele Hardwarezugriffe bevorstehen. Nur beim ersten Zugriff (`read`) wird wirklich auf die Hardware zugegriffen, alle übrigen Zugriffe geben einen zwischengespeicherten Wert zurück bis `EndUpdate` aufgerufen und in den Ausgangszustand zurückgekehrt wird. Bei der neuen ‚Manuellen Justage' werden die Antriebsposition und der Status der Bewegung[18] für jeden Antrieb zwischengespeichert. Außerdem sind die `BeginUpdate`-Aufrufe kaskadierbar, d.h. `EndUpdate` muss so oft gerufen werden, wie `BeginUpdate` zuvor gerufen wurde (in der Abbildung generalisiert).

Durch die Geschwindigkeitsoptimierungen wurden in der Funktionskomponente zusätzliche Methoden und Attribute erforderlich. Fast ausschließlich diese sind für die Erweiterung des OOD-Modells verantwortlich (vgl. **Tab. II.7** und **Tab. II.5**). Wie sich diese 19 Elemente verteilen, ist in **Abb. II.46** gekennzeichnet, sie befinden sich jeweils am Ende der Klassierung.

[18] entweder Antrieb bewegt sich oder er befindet sich im Stillstand

II.4.2 Schwerpunkte bei der Nutzeroberfläche

Die Hauptaufgabe des Oberflächen-Programmierers ist die Behandlung der Ereignisse, die durch die Nuteroberfläche ausgelöst werden. Im Kapitel ‚Benutzeroberfläche' des dekomponierten Pflichtenheftes sind dazu die Reaktion bzw. Funktion jedes Bedienelementes aufgeführt. Für oberflächenunabhängige Elemente sind dort Verknüpfungen für die zu realisierenden Produktanforderung angegeben, die in der OOA in Methoden der Funktionskomponente übersetzt wurden (auch der Zugriff auf Attribute wird über Methoden, Accessor- und Mutator-M. realisiert). Der Programmierer muss neben der Delegation an diese Methoden auch die oberflächenspezifischen Funktionen erfüllen/ implementieren, die im Kapitel ‚Benutzeroberfläche' verbal beschrieben sind.

Einer parallelen Implementierung von Nutzeroberfläche und Funktionskomponente steht zunächst nichts im Wege. Zum Test wird dann jedoch eine vollständig getestete Funktionskomponente benötigt.

<u>Anwendungsfall: ‚Manuelle Justage'</u>

Wie bereits aus dem Oberflächen-Prototyping und dem OOD-Modell bekannt, besteht die Nutzeroberfläche der neuen ‚Manuellen Justage' aus drei Dialogfenstern bzw. den Klassen `TManJustageDlg`, `TMotorOffsetDlg` und `TPsdOffsetDlg`. Dafür wurde das Dateipärchen MJ_GUI.H und MJ_GUI.CPP erstellt, ähnlich der Funktionskomponente. Aufgrund der geringen Komplexität der beiden Offset-Dialogklassen wurden dort die drei Klassen zusammengefasst.

Aus **II.3.2** ist bekannt, dass die Basisklasse für die Oberfläche der neuen ‚Manuellen Justage' `TModalDlg` ist. Sie stellt das Grundgerüst für modale Dialogfenster dar und bietet einen Pool an Methoden die in den abgeleiteten Klassen überschrieben und spezialisiert werden können. Den Kernpunkt bildet dabei die Methode `void Dlg_OnCommand( HWND, int, HWND, UINT )`, wo alle Steuerelementbotschaften empfangen, anhand der Parameter identifiziert und behandelt werden können. Durch die große Anzahl an Steuerelementen im Hauptdialog der neuen ‚Manuellen Justage' wäre diese Methode entweder zu komplex geworden oder sie hätte für jedes der 71 Steuerelemente zu einer separaten Behandlungsroutine verzweigen müssen. Beides hätte den Nachteil, dass die nahezu identisch zu behandelnden Teilbereiche separat zu implementieren wären. Daher wurden bereits im OOD mehrere kleine Methoden zur Verwaltung der Teilbereiche und der darin enthaltenen Steuerelemente geplant, die eine für alle Teilbereiche generalisierte Implementierung erlauben (siehe **Abb. II.48**).

```
void TManJustageDlg::Dlg_OnCommand( HWND aHWnd, int aId,
                                    HWND aCtrlWnd, UINT aCode )
{
  // prüfe zuerst die Steuerelemente in den Teilbereichen, diese wurden in drei Gruppen aufgeteilt:
  // LINKS: Auswahlliste Antrieb, Betriebsarten
  // MITTE: Start/ Stop und Betriebsarten, wenn [ENTER] gedrückt wurde
  // RECHTS: Relative Null setzen, aufheben; Offset setzen
  // wird das entsprechende Steuerelement zu aId gefunden, gibt die Check-Methode TRUE zurück
  ETB tb= (ETB)0;  // nur so gibt TB_Next beim ersten Aufruf tb1 zurück
  while ( TB_Next(tb) ) { // prüfe der Reihe nach die Teilbereiche
    if ( (TB_CheckLeft(tb, aId)) ||
         (TB_CheckMiddle(tb, aId)) ||
         (TB_CheckRight(tb, aId)) ||
         ((aCode==ONEXIT) && (TB_CheckExit(tb, aId))) ) return;
  }

  // prüfe dann die Steuerelemente außerhalb der Teilbereiche
  switch ( aId ) {
    …
    default:
      TModalDlg::Dlg_OnCommand( aHWnd, aId, aCtrlWnd, aCode );
  }
};
```

Abb. II.48 Nutzung von Methoden zur Verwaltung der Teilbereiche

Auf die Vorstellung einer oberflächenspezifischen Funktion wird hier verzichtet, ihre Beschreibung im Pflichtenheft beschränkt sich nur auf:

- die Umgestaltung der Nutzeroberfläche
- das Anzeigen von Meldungsfenstern
- die Ausgabe von Warntönen
- zum Anzeigen/ Verstecken anderer Oberflächenfenster
- Zustandsänderungen innerhalb der Oberflächenklasse

Zusätzlich sind bestimmte Steuerelemente mit einer Produktanforderung verknüpft, so dass die Anbindung an Methoden der Funktionskomponente nötig wird. Mit dem in **Abb. II.50** abgebildeten Steuerelement werden die beiden Funktionen **/ F 70 /** und **/ F 80 /** realisiert. Für **/ F 70 /** wurden die Implementierung der Methoden `DoDirect` und `CanDoDirect` bereits in **II.4.1** erläutert, für **/ F 80 /** wurden dazu `DoStop` und `CanDoStop` implementiert.

Bei Betätigung der ‚Start'- / ‚Stop'-Schaltfläche wird über den in **Abb. II.48** dargestellten Mechanismus in `Dlg_OnCommand` zuerst der entsprechende Teilbereich ermittelt. `TB_CheckMiddle` ruft dann `TB_StartStopClicked` (siehe **Abb. II.49**). Wenn sich der Antrieb nicht bewegt (`IsMoving`), wird die Bewegung mit `DoDirect` gestartet; ansonsten führt `DoStop` zum Stoppen des Antriebs. Im Erfolgsfall sichert ein Timer die in **Abb. II.50** geforderte Bildschirmaktualisierungen.

Im Fehlerfall wird `OnNoAction` aufgerufen, was in Abhängigkeit vom Eintrag "DisplayNoActionMsg" der Konfigurationsdatei zur Ausgabe der Fehlermeldung ‚Aktion ist im aktuellen Zustand nicht verfügbar!' führen kann.

```
void  TManJustageDlg::TB_StartStopClicked ( const ETB aTB ) {
  BOOL bValid;
  // Aufruf während der Halbwertsbreitenmessung ignorieren
  if ( m_lnkManJustage->IsMeasuring() ) return;

  int mSel = TB2Selection(aTB); // den in diesem Teilbereich ausgewählten Antrieb ermitteln
  // nicht in Bewegung und nicht ‚kein Antrieb' ausgewählt: Bewegung starten
  if ( (!m_lnkManJustage->IsMoving(mSel, bValid)) && (bValid) ) {
    // wenn Bewegung gestartet wurde: Timer für die Bildschirmaktualisierung wird gestartet (Polling)
    if ( m_lnkManJustage->DoDirect(mSel) ) m_lnkTimer->StartTimerIm();
    else OnNoAction(); // Bewegung kann nicht gestartet werden: Fehlermeldung ausgeben

  // wenn Bewegung gestoppt wurde: Timer für die Bildschirmaktualisierung wird direkt gerufen
  } else if ( m_lnkManJustage->DoStop(mSel) ) m_lnkTimer->Immediately();
  else OnNoAction(); // Bewegung kann nicht gestoppt werden: Fehlermeldung ausgeben
};
```

Abb. II.49 Realisierung der oberflächenspezifischen Funktionen **/ B 40 /** und **/ B 50 /**

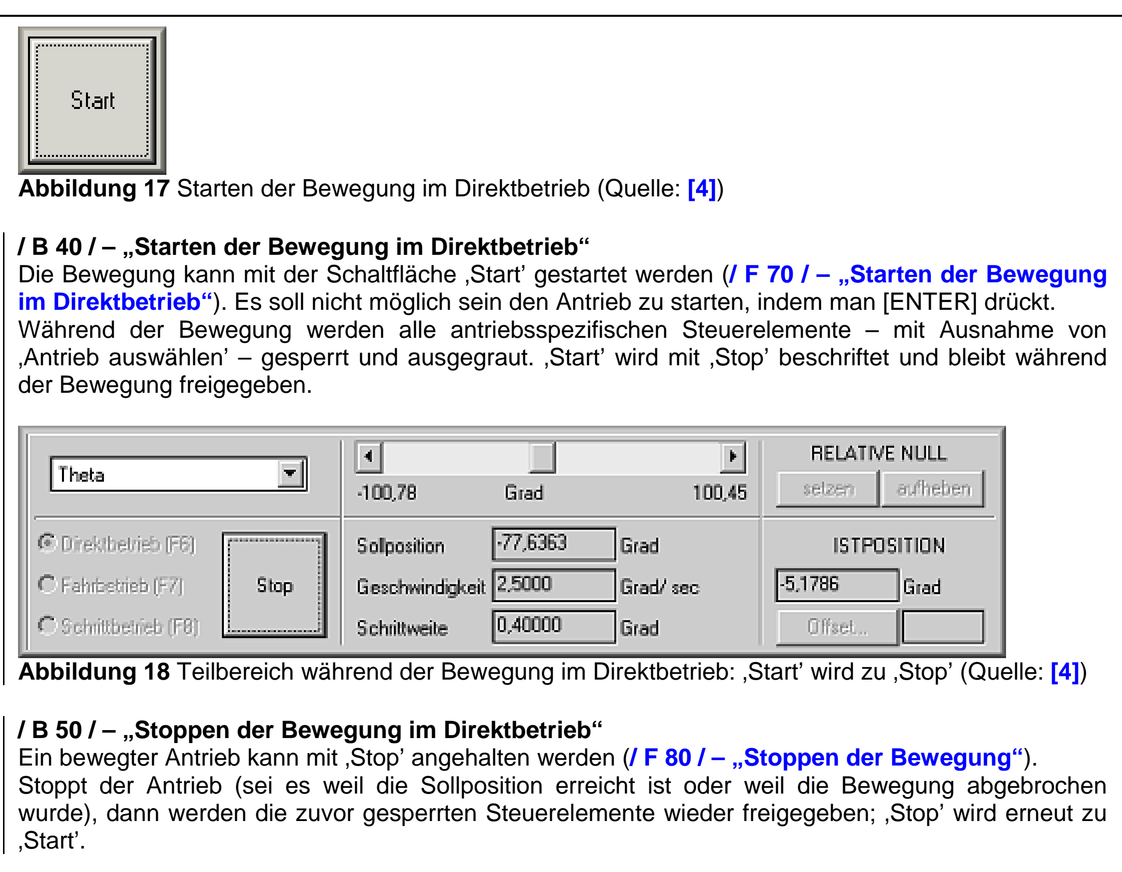


Abbildung 17 Starten der Bewegung im Direktbetrieb (Quelle: **[4]**)

/ B 40 / – „Starten der Bewegung im Direktbetrieb"
Die Bewegung kann mit der Schaltfläche ‚Start' gestartet werden (**/ F 70 / – „Starten der Bewegung im Direktbetrieb"**). Es soll nicht möglich sein den Antrieb zu starten, indem man [ENTER] drückt. Während der Bewegung werden alle antriebsspezifischen Steuerelemente – mit Ausnahme von ‚Antrieb auswählen' – gesperrt und ausgegraut. ‚Start' wird mit ‚Stop' beschriftet und bleibt während der Bewegung freigegeben.

Abbildung 18 Teilbereich während der Bewegung im Direktbetrieb: ‚Start' wird zu ‚Stop' (Quelle: **[4]**)

/ B 50 / – „Stoppen der Bewegung im Direktbetrieb"
Ein bewegter Antrieb kann mit ‚Stop' angehalten werden (**/ F 80 / – „Stoppen der Bewegung"**). Stoppt der Antrieb (sei es weil die Sollposition erreicht ist oder weil die Bewegung abgebrochen wurde), dann werden die zuvor gesperrten Steuerelemente wieder freigegeben; ‚Stop' wird erneut zu ‚Start'.

Abb. II.50 Auszug aus dem Kapitel ‚Benutzeroberfläche' des Pflichtenheftes **[M15]**, der neuen ‚Manuellen Justage'

Beim Programmcode-Review der ursprünglichen ‚Manuellen Justage' war negativ aufgefallen, dass die Aktualisierung eines Steuerelementes an vielen Stellen im Programmcode zu finden war. Diese Art der Implementierung ist wartungsunfreundlich und extrem fehleranfällig! Deshalb wurden im OOD einige Methoden bereitgestellt, die für die Aktualisierung von Inhalt und Zustand von kleinen, disjunkten Steuerelement-Gruppen zuständig sind. Nur dort wird die Aktualisierung der Steuerelemente (zentral) verwaltet! Wenn die verbale Funktionsbeschreibung des Pflichtenheftes eine Aktualisierung eines Steuerelementes fordert, wird stattdessen eine der soeben angesprochenen Methoden gerufen, die die Funktion übernimmt. Diese Vorgehensweise sicherte die unten vorgestellte Geschwindigkeitsoptimierung (Punkt eins und sechs).

Aus den bereits bei der Funktionskomponente angesprochen Gründen, traten während der Vorstellung der neuen ‚Manuellen Justage' widererwartend Probleme mit der Ausführungsgeschwindigkeit auf. Von Seiten der Oberfläche wurden folgende Änderungen durchgeführt:

- Der flüssige Arbeitsablauf wurde nach dem Starten/ Stoppen einer Antriebsbewegung zunächst durch die Oberflächenaktualisierung behindert. Der Aufruf der Methoden, die für die Anpassung der Steuerelemente zuständig sind, führte zu vollständiger Prozessorauslastung, was die Bedienung des Dialogfensters für einige Augenblicke behinderte. Zur Behebung wurden die Methoden `Notice` und `Dlg_OnEvent` eingeführt. Anstatt des direkten Methodenaufrufs wird nun (per `Notice`) eine entsprechende, benutzerdefinierte Windowsbotschaft an das Dialogfenster geschickt. Diese kommt in `Dlg_OnEvent` an, wo wieder die eigentliche Aktualisierungsmethode gerufen wird. Weil die Queue für benutzerdefinierte Botschaften neben der Kommandoqueue existiert, können die darin enthaltenen Steuerelement-Ereignisse parallel verarbeitet werden. Die Bedienung des Dialogfensters ist problemlos möglich. Für schnellere Rechnersysteme kann dieses Verhalten über den Eintrag „Parallel" der Konfigurationsdatei deaktiviert werden. Statt eine Windowsbotschaft zu verschicken, ruft `Notice` dazu direkt `Dlg_OnEvent`.

- Beim Aufruf des Timers zur Bildschirmaktualisierung (Methode `OnTimer`) wurde ursprünglich der Zustand aller Antriebe überwacht. Durch eine Änderung reduzierte sich diese Anzahl auf maximal drei, weil nur noch die in den drei Teilbereichen angezeigten Antriebe geprüft wurden (die Hardwarezugriffe wurden deutlich reduziert). Deshalb musste nun aber die Antriebsauswahl überwacht werden.

- Am ‚Topographie'-Arbeitsplatz konnte das Geschwindigkeitsproblem nur beobachtet werden, wenn die Antriebe ‚Beugung fein' und ‚Beugung grob' gleichzeitig angezeigt wurden. Dieses Verhalten basierte auf der im Pflichtenheft geforderten Bedingung, dass diese Antriebe nicht gleichzeitig bewegt werden dürfen, weil sie in dieselbe Bewegungsrichtung wirken. Wenn der Bewegungsstatus[19] von einem dieser Antriebe ausgelesen wurde, musste daher auch der Zustand des anderen Antriebs aus der Hardware ausgelesen und ausgewertet werden. Wurden beide Antriebe angezeigt, hatte dies viermal so viele Hardwarezugriffe zur Folge! Beseitigt wurde dieses Problem, indem die Nutzeroberfläche nun das zeitgleiche Anzeigen der Antriebe verhindert, damit konnte dann die gegenseitige Überprüfung der Antriebe in der Funktionskomponente entfernt werden.

- Anschließend wurde der in **Abb. II.47** vorgestellte Mechanismus zur Geschwindigkeitsoptimierung in die Nutzeroberfläche integriert. Bevor hier eine umfangreiche Bildschirmaktualisierung beginnt, wird die Funktionskomponente in den Zustand `waiting` bzw. `updating` überführt, damit die wirklichen Hardwarezugriffe reduziert werden. Am Ende der Aktualisierung wird die Funktionskomponente wieder auf `default` zurückgesetzt.

- Weil diese Verfahren allein jedoch nicht ausreichten, werden die Teilbereiche nun quasi-parallel aktualisiert. Dies erfolgt wieder durch den oben vorgestellten `Notice`-Mechanismus. Damit können die Teilbereiche gleichzeitig auf "Antworten" der Hardware warten.

- Als letztes besteht noch die Möglichkeit, die Zustandsaktualisierung (freigeben bzw. sperren und ausgrauen) der Steuerelemente über die Konfigurationsdatei vollständig zu deaktivieren. Über den Eintrag „StateRefresh" kann bestimmt werden, dass die

[19] entweder Antrieb bewegt sich oder er befindet sich im Stillstand

Zustandsaktualisierungsmethoden sofort verlassen werden. Die Steuerelemente bleiben stets freigegeben. Das hat jedoch keinen Einfluss auf die Aktualisierung des Inhaltes (z.B. der Beschriftung) der Steuerelemente. Bei der Benutzung von eigentlich gesperrten und ausgegrauten Steuerelementen erfolgt keine Reaktion. Die Funktionskomponente ist durch die Verwendung der `Can`-Methoden robust genug den "illegalen" Aufruf solcher Funktionen zu ignorieren. Zusätzlich wurde deshalb eine Rückmeldung programmiert, um den Anwender zu informieren.

Die Implementierung der wesentlich übersichtlicheren Offset-Dialoge gestaltete sich in jeder Hinsicht völlig problemlos. Hier mussten weder Methoden für die Verwaltung der Steuerelemente noch Geschwindigkeitsoptimierungen vorgenommen werden. Die Klassen `TMotorOffsetDlg` und `TPsdOffsetDlg` bestehen daher fast ausschließlich aus den überschriebenen Methoden der Basisklasse, damit auf die Benutzung der Bedienelemente reagiert werden kann.

TMotorOffsetDlg
-m_lnkManJustage:TManJustage *
-m_Idx:UINT
-m_Digits:UINT

TPsdOffsetDlg
-m_lnkManJustage:TManJustage *
-m_Idx:UINT
-m_Digits:UINT

TMotorOffsetDlg
#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL
#Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void

TPsdOffsetDlg
#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL
#Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void

TMotorOffsetDlg
+TMotorOffsetDlg(UINT:const,:TManJustage *)
-CalcValue(:BOOL):BOOL
-ChooseAngleMode(:BOOL):void
-IsAngleMode():BOOL

TPsdOffsetDlg
+TMotorOffsetDlg(UINT:const,:TManJustage *)
-CalcValue(:BOOL):BOOL

Attribute
Methoden der Basisklasse(n)
weitere Methoden

Abb. II.51 Seit dem OOD unverändert: das UML-Klassendiagramm für die Dialogfensterklassen **`TMotorOffsetDlg`** und **`TPsdOffsetDlg`** (Implementierungsphase)

Wenn man die Design-Änderungen seit dem OOD verfolgt (vgl. **Tab. II.7** und **Tab. II.5**), sind die beiden Offset-Dialogfensterklassen unverändert geblieben. Nur in `TManJustageDlg` wurden drei neue Methoden für die Geschwindigkeitsoptimierungen benötigt. In welche Kategorie diese eingeordnet werden müssen, ist in **Abb. II.52** gekennzeichnet (**NEU**). Die neuen Methoden befinden sich jeweils am Ende der hervorgehobenen Klassierung.

Die Kennzeichnung nicht-zustandsverändernder Methoden gestaltete sich in der Oberflächenimplementierung etwas umständlicher als bei der Funktionskomponente. Verallgemeinert werden kann nur, dass die Signatur von den Methoden der Basisklasse(n) nicht veränderlich ist. Die nachträgliche Kennzeichnung als const-member-function oder von `const`-Parametern ist nicht möglich. Ereignisse von Bedienelementen und Methoden zur Aktualisierung der Oberfläche stellen stets zustandsverändernde Methoden dar. Die zahlreiche Verwendung von `const`-Parametern ist ähnlich wie in der Funktionskomponente möglich, siehe **Abb. II.52**.

TManJustageDlg
-m_InkManJustage:TManJustage *
-m_InkTimer:TInterfaceTimer *
-m_Selection:UINT *
-m_Parallel:BOOL
-m_StateRefresh:BOOL
-m_DisplayNoActionMsg:BOOL

TManJustageDlg
#Dlg_OnInit(:HWND,:HWND,:LPARAM):BOOL
#Dlg_OnCommand(:HWND,:int,:HWND,:UINT):void
#Dlg_OnHScrollBar(:HWND,:HWND,:UINT,:int):void
#LeaveDialog():void
#OnTimer(:const TBasicTimer *):void
#OnEvent(:HWND,:UINT,:WPARAM,:LPARAM):BOOL

TManJustageDlg
-TB_Count():UINT
-TB_Next(:ETB &):BOOL
-TB_GetId(ETB:const):UINT
-TB_GetName(ETB:const,:LPSTR):void
-GetResId(ETB:const,ERessourceId:const):int
-GetCtrl(ETB:const,ERessourceId:const):HWND
-TB2Selection(ETB:const):UINT
-Selection2TB(UINT:const):ETB
-SetSelection(ETB:const,UINT:const):BOOL
-CanSetSelection(ETB:const,UINT:const):BOOL

TManJustageDlg
+TManJustageDlg()
+~TManJustageDlg()
-OnMotionStarts(UINT:const):void
-OnMotionProgress(UINT:const):void
-OnMotionStops(UINT:const):void
-OnMeasureStarts():void
-OnMeasureProgress():void
-OnMeasureStops():void
-OnPsdOffset():void
-OnChooseTB():void
-OnNoAction():void
-OtherControls():void
-ScrollbarCalculate(TAngle:const,TAngle:const, TAngle:const, :int &,:int &,:int &):void

TManJustageDlg
-TB_CheckLeft(ETB:const,:const int):BOOL
-TB_CheckMiddle(ETB:const,:const int):BOOL
-TB_CheckRight(ETB:const,:const int):BOOL
-TB_CheckExit(ETB:const,:const int):BOOL
-TB_StartStopClicked(ETB:const):void
-TB_AngleDestEntered(ETB:const,BOOL:const):void
-TB_SpeedEntered(ETB:const,BOOL:const):void
-TB_StepWidthEntered(ETB:const,BOOL:const):void
-TB_MotorChanged(ETB:const):void
-OnTB_Controls(ETB:const,BOOL:const):void
-OnTB_KindOfMove(ETB:const,BOOL:const):void
-OnTB_StartStop(ETB:const,BOOL:const):void
-OnTB_MotionParam(ETB:const,BOOL:const):void
-OnTB_Positions(ETB:const,BOOL:const):void
-OnTB_Offset(ETB:const,BOOL:const):void
-OnTB_Units(ETB:const):void
-OnTB_ControlsEnable(ETB:const):void
-OnTB_MotorListEnable(ETB:const):void
-OnTB_KindOfMoveEnable(ETB:const):void
-OnTB_StartStopEnable(ETB:const):void
-OnTB_MotionParamEnable(ETB:const):void
-OnTB_PositionsEnable(ETB:const):void
-OnTB_OffsetEnable(ETB:const):void
-OnTB_Supervise(ETB:const):void
-Notice(:const int,ETB:const,BOOL:const):void

- Attribute
- Methoden der Basisklasse(n); **NEU 1**
- Verwaltung der Teilbereiche
- Behandlungsroutinen und Aktualisierung eines Teilbereichs, **NEU 2**
- weitere Methoden

Abb. II.52 Seit dem OOD leicht verändert: das UML-Klassendiagramm für die Dialogfensterklasse `TManJustageDlg` (Implementierungsphase)

II.4.3 Zusammenfassung

Die Implementierung der Funktionskomponente gestaltete sich trotz ihres großen Umfangs (635 **LOC_F**) und der Vielzahl an Methoden (71 **NOO_F**) sehr übersichtlich. Der durchschnittliche Methodenumfang beträgt etwa 13 LOC. Durch die seit der OOA bekannte Klassifizierung der Methoden durch Präfixe konnte die große Datei deutlich gegliedert werden. Die stark formalisierte Designbeschreibung (incl. OOD) und das Pflichtenheft bildeten die alleinige Grundlage für die Erstellung der Funktionskomponente. Zu bemerken ist auch der hohe Kommentierungsgrad der Komponente (**TCR_F** 55), der die Wartung und das Verständnis der z.T. komplexen Bedingungen (`Can`-Methoden) erleichtern sollte.

Die Implementierung der Oberflächenklassen erfolgte parallel zur Funktionskomponente und durch einen anderen Bearbeiter als den der Funktionskomponente. Mit dem stark formalisierten und codenahen Designdokument und dem Pflichtenheft als Ausgangspunkt, entstanden die Oberflächeklassen. Die Implementierung des Hauptdialogfensters war durch die Geschwindigkeitsoptimierungen und die vielen Steuerelemente sehr aufwendig (1.103 $\mathbf{LOC_O}$) und besitzt mit etwa 19 LOC einen etwas komplexeren, durchschnittlichen Methodenumfang als die Funktionskomponente. Darin enthalten sind jedoch fast 50% Kommentierung ($\mathbf{TCR_O}$ 44) und zahlreiche Leerzeilen zur übersichtlicheren Gestaltung der Methoden.

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TManJustage	625	8	71	1	0	10	0	90	56	3	2	55
↳ TMotorData	10	10	0	0	0	0	0	100	68	0	1	100
TManJustageDlg	918	6	50	1	6	86	10	3	54	6	1	44
TMotorOffsetDlg	107	3	5	1	2	67	22	11	27	4	1	46
TPsdOffsetDlg	78	3	3	1	2	57	29	14	27	4	1	54

Tab. II.7 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage’ (Polling-Variante nach Abschluss der Implementierungsphase)

Bei der Integration der neuen ‚Manuellen Justage’ in das ‚XCtl’-Programm zeigte sich, dass wirklich nur die Nutzeroberfläche nach außen sichtbar sein muss. Bei der Verknüpfung des ‚XCtl’-Programmes mit der Oberfläche der neuen ‚Manuellen Justage’ musste deshalb nur MJ_GUI.H inkludiert werden.

Die während der Implementierungs-, Test- und Wartungsphase anfallenden Änderungen am Programmcode konnten fast immer im Vorhinein entweder auf die Funktionskomponente oder die Oberfläche eingegrenzt werden. Durch die deutliche Gliederung und die konsistente Benennung der Methoden der Funktionskomponente konnten Wartungsarbeiten sogar auf die Methode genau abgegrenzt werden!

II.4.3–1 Bewertung von Observer-Muster und Polling im Anwendungsfall

Auf die graphische Gegenüberstellung dieser beiden Kommunikationsmuster wird hier verzichtet, weil die Struktur seit der Designphase nahezu unverändert geblieben ist. Für die Geschwindigkeitsoptimierungen sind seitdem vier neue Attribute (NOA_F, **NEU 4**) und vier neue Methoden (NOO_F, **NEU: 4**) in der Funktionskomponente sowie zwei neue Methoden (NOO_N, **NEU: 2**) für die Nutzeroberfläche eingefügt worden. Diese Änderungen sind unabhängig vom gewählten Kommunikations-Muster, weil die Optimierungen in beiden Systemen identisch implementierbar waren.

Beim Vergleich des Implementierungsumfanges gibt es insgesamt einen kleinen Nachteil von 12 **LOC** für das Observer-Muster, 1.750 zu 1.738 (Polling-Variante). Dabei ist eine deutliche Umstrukturierung des Programmcodes zu erkennen. Weil die Nutzeroberfläche bei der Polling-Variante den Kommunikationsmechanismus initiiert, ist diese deutlich umfangreicher als beim Observer-Muster, LOC_O 1.103 zu 1054 (Observer-Muster). Dementsprechend muss die Situation der Funktionskomponente komplementär sein. Beim Observer-Muster stehen $\mathbf{LOC_F}$ 696 den 635 der Polling-Variante gegenüber. Ob die Umverteilung von etwa 50 LOC jedoch zu einer deutlich höheren Wiederverwendbarkeit der Observer-Funktionskomponente

führt und das komplexere Design rechtfertigt, ist bei diesem Anwendungsfall eher fraglich. In anderen Fällen könnten die äußeren Bedingungen jedoch den Einsatz der Polling-Variante verhindern oder sie unvorteilhaft machen. Abschließend favorisieren die Autoren hier die Polling-Variante. Im Allgemeinen sollte dem Observer-Muster nur Vorrang gewährt werden, wenn die Polling-Variante nicht anwendbar oder zu fehleranfällig ist (siehe **II.3.3–4**).

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
IManJustageDlg	13	0	6	0	0	0	0	100	0	1	1	52
TManJustage	675	8	**65**	1	0	12	1	87	65	3	2	55
↳ TMotorData	8	8	0	0	0	0	0	100	66	0	1	100
TManJustageDlg	869	5	50	1	11	77	19	4	45	**6**	1	46
TMotorOffsetDlg	unverändert zu **Tab. II.7**											
TPsdOffsetDlg												

Tab. II.8 Ausgewählte Metriken für die Klassen der neuen ‚Manuellen Justage' (Observer-Muster nach Abschluss der Implementierungsphase)

II.5 Testphase

Inhalt dieses Abschnittes ist die Vorgehensweise für das separate Testen von Funktionskomponente und Nutzeroberfläche bei einem dekomponierten Softwareprojekt. In dieser Phase der Entwicklung ist es unerheblich, ob die zu testende Anwendung ein Neuentwurf oder ein saniertes Projekt ist. Die Ausgangsbasis für alle Testaktivitäten ist stets eine fertig implementierte Anwendung bzw. ein Subsystem. So gelten die in diesem Kapitel vorgestellten Schrittfolgen für das Testen nicht nur für das Forward- sondern auch für das Reengineering.

Die Wahl des Kommunikationsmusters beeinflusst diese Schrittfolgen nur durch die unterschiedliche Fehleranfälligkeit (bedingt durch die Komplexität des Design und Umfang des Programmcodes) des zu testenden Systems. Dank der bei **II.4.3–1** vorgestellten Gemeinsamkeiten wird hier völlig auf eine Unterscheidung verzichtet. Zur Wahrung des roten Fadens wurden die Beispiele wieder der Polling-Variante entnommen.

II.5.1 Test einer Funktionskomponente

Beim Testen der Funktionskomponente ist zu überprüfen, ob alle im Pflichtenheft geforderten Produktspezifikationen vorhanden sind (Vollständigkeit), das Programm das Gewünschte leisten (Validation) und fehlerfrei arbeitet (Verifikation).

Die Funktionskomponente und ihre Anbindung an die benutzten Subsysteme sind vollständig objektorientiert. Hierbei besteht eine Datenkopplung zwischen den Operationen eines Objektes und den gemeinsam benutzten Objektattributen. Eine Methode kann durch Änderung eines Attributes einen Objektzustand hinterlassen, der das Verhalten des Folgezustands beeinflusst. Beim Testen muss daher die Aufrufreihenfolge der Objektoperationen beachtet werden. Nach **[3]** (S. 488ff) ist ein **Test objektorientierter Komponenten** durchzuführen!

Innerhalb dieses Tests wird zur Verifikation ein **kontrollflussorientierter Strukturtest** / White-Box-Testverfahren (siehe **[3]**, S. 400ff) verwendet. Ziel ist mit einer Anzahl von Testfällen und -daten alle vorhandenen Anweisungen und Verzweigungen des Programmcodes auszuführen und zu bewerten. Für die Validation und Sicherung der Vollständigkeit muss dieses jedoch um Elementen des **funktionalen Tests** / Black-Box-Testverfahren (siehe **[3]**, S. 426ff) ergänzt werden. Dies betrifft die Gewinnung von zusätzlichen Testdaten und -fällen aus der Spezifikation. Neben den direkt ableitbaren Testfällen können zusätzliche mittels Grenzwertanalyse und Äquivalenzklassenbildung (siehe **[3]**, S. 427ff bzw. 431ff) gewonnen werden.

Die Grundlage für den Test von Funktionskomponenten bilden ausgewählte Schritte eines Testverlaufs für objektorientierte Komponenten zusammengetragen aus **[3]**, Kapitel 5.13.1:

- „Durch Äquivalenzklassenbildung und Grenzwertanalyse werden aus den Parametern Testfälle abgeleitet. Das Objekt muss vorher in einen für diesen Testfall zulässigen Zustand versetzt werden." (aus **[3]**, Seite 489, Schritt 2a)
- Dann wird jede Operation einzeln getestet. Zuerst sollten nicht-zustandsverändernde Methoden überprüft werden (nach **[3]**, Seite 489, Schritt 2).
- „Anschließend werden die zustandsverändernden Operationen getestet." (nach **[3]**, Seite 489, Schritt 2) „Nach jeder Operationsausführung muss der neue Objektzustand geprüft und der oder die Ergebnisparameter mit den Sollwerten verglichen werden." (nach **[3]**, Seite 489, Schritt 2b)
- Danach folgt der Test jeder Folge voneinander abhängigen Operationen innerhalb der Klasse. „Alle potentiellen Verwendungen einer Operation sollte unter praktisch relevanten Bedingungen ausprobiert werden." (nach **[3]**, Seite 489, Schritt 3)

Aus dieser allgemeinen Schrittfolge und der klaren Strukturierung der öffentlichen Schnittstelle der Funktionskomponente (aus **II.2.5**) konnte ein genereller Testverlauf abgeleitet werden:

1) **Gewinnung von Testdaten**
2) **Generierung von Testfällen**
3) **Erstellung einer Testumgebung (Anpassung der existierenden Nutzeroberfläche oder Implementierung einer Testoberfläche)**
4) **Erzeugung von Objektinstanzen**
5) **Test nicht-zustandsverändernder Methoden**
6) **Test zustandsverändernder Methoden mit „geringer“ Komplexität**
7) **Test zustandsverändernder Methoden mit „hoher“ Komplexität**

Die starke Formalisierung von Pflichtenheft und Programmcode wird das Finden von Testdaten bei **1)** erleichtert. Durch die Tabellarisierung der Datenelemente im Pflichtenheft (mit konsequenter Zuordnung der Grenzwerte), können fast alle Testdaten allein aus diesem Dokument gewonnen werden. Für **2)** werden die aus dem Programmcode gewonnenen Testfälle durch die im Funktionsteil des Pflichtenheftes aufgeführte Spezifikation ergänzt.

Für den Test einer Klasse ist stets eine Testumgebung (siehe **[3]**, Seite 493) nötig, um den Objektzustand gezielt zu manipulieren und zu analysieren. Sie besteht aus:

- einem Testtreiber, um Operationen gezielt aufzurufen
- einem Botschaften-Generator zur Generierung von Eingaben
- einem Botschaften-Auswerter zur Bewertung der Ausgaben
- einem Objektzustands-Initialisator zur Schaffung eines bestimmten Vorzustandes und
- einem Testmonitor zur Protokollierung der Ausführungsfolge von Operationen .

Wenn Botschaften-Generator bzw. -Auswerter nicht automatisierbar sind, empfiehlt sich die Verwendung einer Testoberfläche, im Falle der Funktionskomponente i.d.R. die spätere Nutzeroberfläche. Es ist jedoch möglich, dass diese noch nicht in ihrer endgültigen Form vorliegt oder aufgrund ihrer Komplexität nicht für diesen Einsatz geeignet ist. Dann kann für **3)** eine separate Testoberfläche erstellt werden, wenn der dafür aufzubringende Mehraufwand abgewogen wird. Durch den Einsatz der Testoberfläche wird sichergestellt, dass eine vollständig getestete Funktionskomponente für den Test der eigentlichen Nutzeroberfläche bereit steht.

Zum Beginn des eigentlichen Testes (**4)**) müssen die erforderlichen Objekte (Systemkomponente, benutzte Subsysteme, Test-/ Nutzeroberfläche und Funktionskomponente) instanziiert und gezielt initialisiert sein.

Bezogen auf die Struktur der Funktionskomponente befasst sich Punkt **5)** ausschließlich mit `Get`-, `Is`-, `Has`-, `Can`-M., Punkt **6)** nur mit Mutator-Methoden (`Set`-M.) und Punkt **7)** mit `Do`-Methoden und dem übrigen, ungetesteten Teil der öffentlichen Schnittstelle.

Anwendungsfall: ‚Manuelle Justage’

Der Test erfolgte nicht an den Messplätzen im Physikalischen Institut, d.h. ohne angeschlossene Hardware. Die Reaktionen der Antriebe und Detektoren wurden durch Subsysteme des ‚XCtl’-Systems simuliert.

1) Gewinnung von Testdaten

Die Testdaten wurden aus dem Datenteil des Pflichtenheftes abgeleitet. Dies soll hier exemplarisch für das Datenelement ‚Sollposition' nachvollzogen werden. Der Datenteil-Eintrag **/ D 110 /** (siehe **Abb. II.53**, Tabelle 13) zeigt, dass es eine untere und eine obere Grenze für die Sollposition gibt. Dort ist erkennbar, dass die minimalen und maximalen Grenzen antriebsspezifisch, d.h. hardwareabhängig sind (siehe **Abb. II.53**, Tabelle 8 und 9). Die Testdaten sind den Einträgen ‚Speichertyp' und ‚Speicherort' nach aus der Konfigurationsdatei (für den betrachteten Antrieb) zu entnehmen. Zusätzlich ist die antriebsspezifische ‚Nachkommastellen'-Genauigkeit zu berücksichtigen (siehe **Abb. II.53**, Tabelle 10), die auch einer Konfigurationsdatei entnommen wird.

/ D 110 / – „eingegebene Sollposition"
Die Sollposition ist – wie die Istposition – abhängig vom aktuellen Offset des Antriebs. Die Minimal- und Maximalwerte können dem Datenteil entnommen werden; ebenso die Nachkommastellengenauigkeit. Die Sollposition ist bei jeder Änderung zu speichern!

Bezeichnung			eingegebene Sollposition		
Typ	Zugriff	Eintrag	ℝ	lesen und schreiben	NEU
Speichertyp			als Eintrag in einer ini-Datei (ASCII-Text)		
Speicherort			**HARDWARE.INI –> [MOTOR<M>] –> MJ_AnlgeDest**		
Minimalwert			siehe **Tabelle 8**		
Maximalwert			siehe **Tabelle 9**		
Standardwert			aktuelle Istposition		
Nachkommastellen			siehe **Tabelle 10**		

Tabelle 13 Eigenschaften der Sollposition

/ D 60 / - „minimale Istposition"

Bezeichnung			minimale Istposition		
Typ	Zugriff	Eintrag	ℝ	nur lesen	bereits vorhanden
Speichertyp			als Eintrag in einer ini-Datei (ASCII-Text)		
Speicherort			**HARDWARE.INI –> [MOTOR<M>] –> AngleMin**		

Tabelle 8 „minimale Istposition"

/ D 70 / - „maximale Istposition"

Bezeichnung			maximale Istposition		
Typ	Zugriff	Eintrag	ℝ	nur lesen	bereits vorhanden
Speichertyp			als Eintrag in einer ini-Datei (ASCII-Text)		
Speicherort			**HARDWARE.INI –> [MOTOR<M>] –> AngleMax**		

Tabelle 9 „maximale Istposition"

/ D 80 / - „Nachkommastellen"

Bezeichnung			Nachkommastellen		
Typ	Zugriff	Eintrag	ℕ	nur lesen	bereits vorhanden
Speichertyp			als Eintrag in einer ini-Datei (ASCII-Text)		
Speicherort			**HARDWARE.INI –> [MOTOR<M>] –> Digits**		
Minimalwert			**0**		
Maximalwert			**10**		
Standardwert			**3**		

Tabelle 10 „Nachkommastellen der Istposition"

Abb. II.53 Auszug aus dem Dateiteil des Pflichtenheftes **[M15]**

Mit den gewonnenen Testdaten lassen sich folgende Äquivalenzklassen ableiten:

Äquivalenzklasse - gültige Eingaben

- (->[MOTOR<M>]->AngleMin) ≤ Sollposition ≤ (->[MOTOR<M>]->AngleMax)

Äquivalenzklasse - ungültige Eingaben

- Sollposition < (->[MOTOR<M>]->AngleMin)
- Sollposition > (->[MOTOR<M>]->AngleMax)

Die Anzahl der durchzuführenden Testfälle hängt von den Bedingungen für das Setzen der Sollposition ab. Diese sind im Funktionsteil des Pflichtenheftes zu finden (siehe **Abb. II.54**). Damit sind im Zustand (‚Direktbetrieb' und Stillstand des Antriebs) alle drei Testdaten für die ‚Sollposition' auszuwerten. In allen anderen Programmzuständen ist einmal der Wert aus der ‚Äquivalenzklasse - gültige Eingaben' anzuwenden, um sicherzustellen, dass diese Eingabe ignoriert wird. Die Suche im Designdokument zeigte, dass `SetAngleDest` zur Manipulation der ‚Sollposition' auszuführen ist.

Grundlagen
Die **Sollposition** ist nur im *Direktbetrieb* erforderlich. Dort gibt sie an, wohin sich der Antrieb bewegen soll. Die Sollposition besteht, wie die *Istposition* auch, aus der Absolutposition des Antriebs verschoben um einen *Offset*. Damit ist die Sollposition auch abhängig von dem *Offset für <Antrieb>* und der *Relativen Null*, weil diese intern über das *Offset* abgebildet werden.

Bedingung
Der Antrieb darf sich nicht bewegen und der *Direktbetrieb* muss ausgewählt sein.

Abb. II.54 Auszug aus dem Funktionsteil des Pflichtenheftes **[M15]**

2) Generierung von Testfällen (CTE-unterstützt)

Die Generierung der Testfälle selbst wurde mit dem **Classification Tree Editor** (CTE) von Xerces vorgenommen (zu CTE siehe **[9]**, S. 138ff). Für den Test wurden vier CTE-Diagramme erstellt. Das erste Diagramm dient zur Auswertung aller nicht-zustandsverändernden Operationen, die einen Wahrheitswert zurückgeben (**logische Operationen**). Es waren 15 Testfälle mit jeweils 36 Testdaten (18 Methoden, Rückgabewert jeweils `true` und `false`) notwendig. Die Testfälle entsprechen den Programmzuständen, unter denen die Methoden ausgewertet werden sollen. Sie stellen eine Kombination aus den Betriebsarten, dem Antriebs-Offset, dem Bewegungsstatus eines Antriebs sowie der Halbwertsbreitenmessung dar. In jedem Programmzustand werden die Rückgabewerte aller 18 Methoden auf ihre Richtigkeit hin überprüft.

Das zweite Diagramm beinhaltet die Überprüfung der Verfügbarkeit eines ausgewählten Antriebs und Detektors und die Überwachung ihrer Daten in den einzelnen Zuständen. Die dafür zuständigen 11 Methoden werden mittels 22 Testdaten und 12 Testfällen überprüft. Während das erste Diagramm hauptsächlich zur Validation der Projektspezifikationen diente, wird hier die Kopplung der neuen ‚Manuellen Justage' mit den "niederen" Subsystemen (hauptsächlich also der Hardware) bewertet. Auch hier werden in jedem Programmzustand alle Methoden parallel beobachtet. Zustandsänderungen wurden durch die Steuerung der Hardware (z.B. Bewegen und Anhalten eines Antriebs) erreicht.

Die Testfälle für die übrigen nicht-zustandsverändernden Methoden (ausgenommen den logischen Operationen aus Diagramm eins) sind in einem dritten Diagramm mit den zustandsverändernden `Set`- und `Do`-Methoden zusammengefasst. Da diese Methoden Eingaben erfordern und sich die Testdaten teilweise überschneiden, wurden sie in einem CTE-Diagramm kombiniert. Es umfasst 4 `Calc`-, 7 `Do`-, 10 `Set`- sowie die zur Auswertung nötigen 8 `Get`-Methoden. Dieses CTE-Diagramm ist mit insgesamt 35 Testdaten, 72 Testfällen für die

Set-Methoden und 7 Testfällen für die Do-Methoden das größte. Der jeweilige Programmzustand wird wieder durch die Kombination aus den Betriebsarten, den verschiedenen Offsets, dem Bewegungsstatus eines Antriebs sowie der Halbwertsbreitenmessung bestimmt.

Gemeinsames Merkmal der drei beschriebenen CTE-Diagramme ist, dass alle Operationen nur für einen ausgewählten Antrieb getestet werden. Dieser eine getestete Antrieb kann aber als gute Referenz betrachtet werden, da sich die Steuerung der Antriebe nur in deren Adressierung unterscheidet. Um jedoch die gleichzeitige Bewegung mehrere Antriebe zu testen, war ein zusätzliches, viertes CTE-Diagramm mit 14 Testfällen und 10 Testdaten für 4 Antriebe nötig. Für den vollständigen Test der Funktionskomponente TManJustage wurden vier CTE-Diagramme mit insgesamt 120 Testfällen mit 103 Testdaten erstellt (siehe **[M22]**).

TManJustage

+GetMotionType:EMotionType
+GetAngleDest:TAngle
+GetSpeed:TSpeed
+GetAngleWidth:TAngle
+GetKzMoving:BOOL
+GetOffset:TAngle
+GetMotorCount:UINT
+GetMotorName:LPCSTR
+GetMotorIdx:int
+GetMotorIdx:int
+GetAngle:TAngle
+GetAngleMin:TAngle
+GetAngleMax:TAngle
+GetDigits:UINT
+GetUnit:LPCSTR
+GetDetectorAxisIdx:int
+GetDetectorName:LPCSTR
+GetChannel:int
+GetAnglePerChannel:TAngle
+GetKzMeasuring:BOOL
+GetMeasureProgress:BOOL
+GetMeasureHwb:double
+GetDetector:TDetector *

TManJustage

+IsIndexValid:BOOL
+HasOffset:BOOL
+IsMoving:BOOL
+IsCalibrated:BOOL
+HasDetectorAxis:BOOL
+IsMeasuring:BOOL
+IsMeasureReset:BOOL
+ParsingAxis:UINT
+CalcChannelOffset:TAngle
+CalcChannelAngle:TAngle
+CalcOffsetFromAngle:TAngle
+CalcAngleFromOffset:TAngle
+CanDoDirect:BOOL
+CanDoDrive:BOOL
+CanDoStep:BOOL
+CanDoStop:BOOL
+CanSetMotionType:BOOL
+CanSetAngleDest:BOOL
+CanSetSpeed:BOOL
+CanSetAngleWidth:BOOL
+CanResetRelativeZero:BOOL
+CanSetOffset:BOOL
+CanDoStartMeasure:BOOL
+CanDoStopMeasure:BOOL

TManJustage

+SetMotionType:BOOL
+SetAngleDest:BOOL
+SetSpeed:BOOL
+SetAngleWidth:BOOL
+SetKzMoving:BOOL
+SetOffset:BOOL
+SetKzMeasuring:void
+SetChannel:BOOL
+SetRelativeZero:BOOL
+ResetRelativeZero:BOOL

TManJustage

+DoDirect:BOOL
+DoDrive:BOOL
+DoStep:BOOL
+DoStop:BOOL
+DoStopEverything:void
+DoStartMeasure:BOOL
+DoStopMeasure:BOOL

nicht-zustandsverändernde Methoden (Get-, Has-, Is-, Can- und Calc-M.)
zustandsverändernd mit „geringer“ Komplexität (Set-M.)
zustandsverändernd mit „hoher“ Komplexität (Do-M.)

Abb. II.55 Öffentliche Schnittstelle von **TManJustage**, zerlegt nach der Komplexität ihrer Methoden

3) Erstellung einer Testumgebung (Anpassung der existierenden Nutzeroberfläche oder Implementierung einer Testoberfläche)

Die Nutzeroberfläche der ‚Manuellen Justage' ist sehr komplex (siehe **Abb. II.10**) und wurde parallel zur Funktionskomponente implementiert. Da die Oberfläche zu Beginn der Testaktivitäten für die Funktionskomponente noch nicht fertig war, wurde eine separate Testoberfläche erstellt. Diese ist dabei Testtreiber und Objektzustands-Initialisator zugleich. Die Aufgaben von Botschaften-Generator, -Auswerter und Testmonitor (siehe oben) waren hingegen nicht automatisierbar. Grund dafür ist die sehr große Anzahl von Testfällen und -daten, welche die Entwicklung eines automatischen Testverfahrens in angemessener Zeit nicht möglich machte. Zudem hätte die umfangreiche Implementierung selbst wieder getestet werden müssen.

Das Hauptmerkmal einer guten Testoberfläche ist die einfache Gestaltung und Implementierung. Aufgabe ist die Ansteuerung aller `public`-Methoden der Funktionskomponente und dabei jede zusätzliche Oberflächen-Funktionalität zu vermeiden.

Obwohl diese sehr komplex wirkt (**Abb. II.56**), hat sie im Vergleich zum Hauptdialog **Abb. II.10** (oben) deutlich weniger bedienbare Steuerelemente. So ist die Implementierung weniger komplex und fehleranfällig. Der Grund ist, dass im Hauptdialog fast alle Funktionen in den drei Teilbereichen (nahezu identisch) vorhanden sind. Zudem gibt es keine Adressierungsprobleme wie im Hauptdialog, d.h. Funktionen werden statt im ersten im dritten Teilbereich ausgelöst, die Werte werden im falschen Teilbereich aktualisiert.

Der Quellcode der Klasse `TManJustageTestDlg` umfasst mit 233 **LOC** nur ca. 16 % des Quellcodeumfangs der wirklichen Nutzeroberfläche. Auch die Verhältnisse von **NOO** 6 zu 60 und **NOA** 3 zu 12 (vgl. **Tab. II.9** und **Tab. II.7**) zeigen deutlich ihre Einfachheit.

Ein großer Vorteil der Testoberfläche ist, dass es für das Design der Oberfläche keine Vorgaben gibt. Damit konnte das Fenster auf die öffentliche Schnittstelle von `TManJustage` (vgl. **Abb. II.46**) zugeschnitten werden, was die parallele Überwachung fast aller nicht-zustandsverändernder Operationen in jedem Objektzustand ermöglicht. Nach jeder Nutzereingabe erfolgte dazu eine Aktualisierung.

Da die Offsetdialoge ‚Offset für <Antrieb>' und ‚Offset für PSD' (siehe **Abb. II.10**, unten) bereits implementiert waren, konnten die damit realisierten Produktfunktionen durch Verknüpfung dieser Dialoge mit der Testoberfläche ohne zusätzlichen Aufwand getestet werden.

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
`TManJustageTestDlg`	233	3	6	1	3	55	0	45	27	4	1	41

Tab. II.9 Ausgewählte Metriken der Testoberfläche

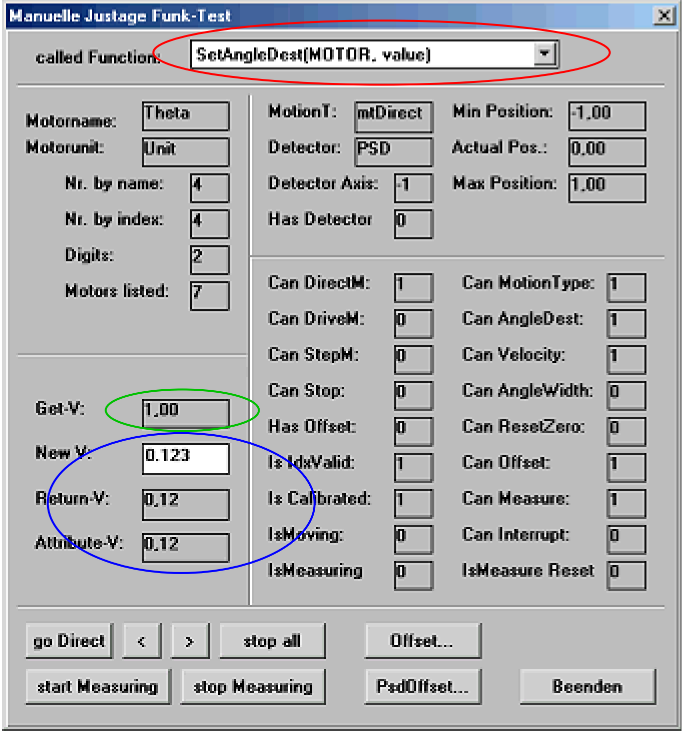


Abb. II.56 Dialog ‚Manuelle Justage Funk-Test für `TManJustageTestDlg`

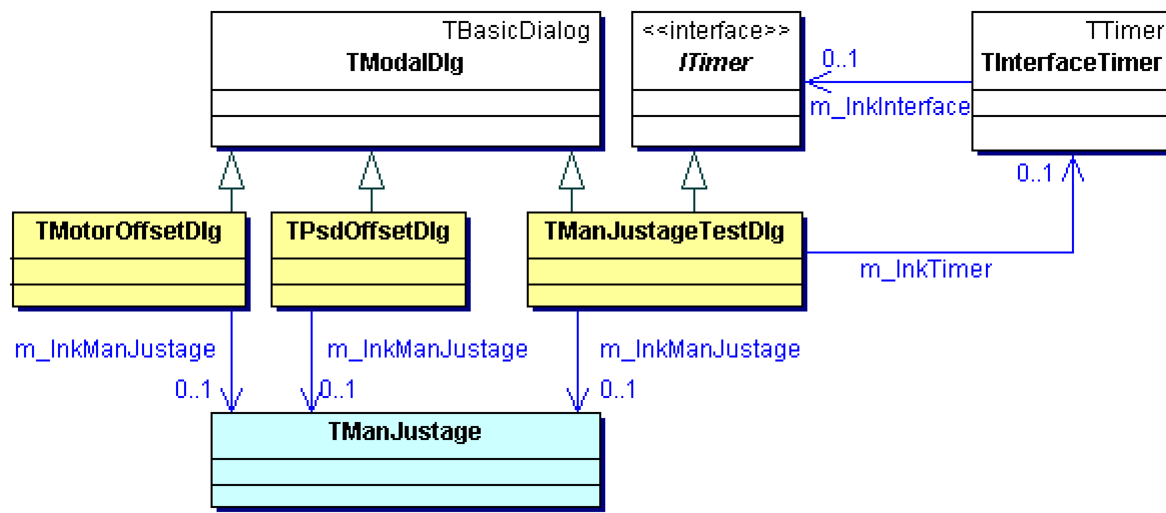


Abb. II.57 UML-Klassendiagramm der ‚Manuellen Justage' mit Testoberflächenklasse `TManJustageTestDlg`

4) Erzeugung von Objektinstanzen

Zuerst erfolgte die Erzeugung eines Objektes der Funktionskomponente `TManJustage` innerhalb der Testoberfläche `TManJustageTestDlg`. Da nur der Standardkonstruktor implementiert ist, wird immer auf die gleiche Art und Weise instanziiert. Da der Konstruktor aber Daten aus einer Konfigurationsdatei liest und Attribute mit Werten aus anderen Subsystemen initialisiert, kann der Ausgangszustand des Objektes doch variieren. Beim Programmstart wurden deshalb verschiedene Einstellungen der Betriebsarten, der Offsets und vom Antriebs- und Detektorstatus getestet. Es waren 7 Testfälle (verschiedene Versionen einer Konfigurationsdatei) nötig.

5) Test nicht-zustandsverändernder Methoden (`Get`-, `Is`-, `Has`-, `Can`-M.)

Hier sollten alle Methoden der Funktionskomponente `TManJustage` mit den Präfixen `Get`, `Can`, `Has`, `Is` und `Calc` getestet werden. Dies entspricht 66% der öffentlichen Schnittstelle bzw. 47 Methoden (siehe **Abb. II.55**, ▬). Die Testoberfläche überwacht davon ständig die Rückgabewerte von 31 Methoden (alle `Can`-, `Is`- und `Has`-M. sowie die Ausgaben über den Antriebs- und Detektorstatus). Neun von restlichen nicht-zustandsverändenden M. sind `Get`-Methode mit einer zugehörigen `Set`-M., so dass ihre Überprüfung erst bei **6)** erfolgt.

Die vier `Calc`-Methoden wurden mit Hilfe der Offsetdialoge ‚Offset für <Antrieb>' und ‚Offset für PSD' getestet. Sie führen nur Offset-Berechnungen durch, ohne den Objektzustand zu ändern. Sie sind damit nicht-zustandsverändernd.

Es sind noch 3 `Get`-Methoden übrig, deren Rückgabewert von `Do`-Methoden abhängig ist und somit bei **7)** getestet wird.

6) Test zustandsverändernder Methoden mit „geringer" Komplexität (`Set`-M.)

Das Kriterium „geringe" Komplexität bezieht sich auf `Set`-Methoden (siehe **Abb. II.55**, ▬). Dort werden Attribute der Funktionskomponente geschrieben und/ oder der Zustand "niederer" Subsysteme geändert. Die Auswahl einer `Set`-Methode erfolgte in der Testoberfläche durch das Kombinationsfeld (⬭), woraufhin zuerst die dazugehörige `Get`-Methode ausgeführt wird, um den aktuellen Zustand des zu verändernden Attributes festzuhalten
(⬭). Nach Eingabe des Parameters werden die Ergebnisse der `Set`-Methode (Rückgabewert oder Referenzparameter) und ein erneuter Aufruf der zugehörigen `Get`-Methode ausgegeben (⬭), um die beiden Werte zu vergleichen. So wurden nacheinander alle `Set`-Methoden ausgeführt.

7) Test zustandsverändernder Methoden mit „hoher" Komplexität (Do-M.)

Im letzten Testabschnitt wurden die „komplexen" Operationen, d.h. Do-Methoden, getestet. Diese bewirken im ‚XCtl'-System eine Zustandsänderung der angeschlossenen Hardware. Bei der neuen ‚Manuellen Justage' ist dies das Starten oder Stoppen einer Antriebsbewegung und die programmgesteuerte Durchführung einer Halbwertsbreitenmessung (siehe **Abb. II.55**,). In der Testoberfläche werden alle Do-Methoden durch Schaltflächen repräsentiert, da sie alle keine durch den Nutzer manipulierbaren Parameter besitzen. Die Bewegungsparameter werden ausschließlich aus den Objekt-Attributen gelesen, die zuvor mit Set-Methoden eingegeben wurden. Dazu wurden die Bewegungen erst an einen Antrieb und danach an mehreren Antrieben gleichzeitig durchgeführt.

II.5.2 Test einer Nutzeroberfläche

Die Nutzeroberfläche entspricht der öffentlichen Schnittstelle, über die der Anwender mit dem Programm kommuniziert. Bei den Testaktivitäten steht die Bedienung der Oberfläche im Mittelpunkt.

In **II.3.2** wurde bereits erläutert, dass die Eingaben des Anwenders (bei botschaftenbasierten Systemen) in Nachrichten an das Programm übersetzt werden. Beim ‚XCtl'-Projekt sind dies Windows-Botschaften an `Dlg_OnCommand` der entsprechenden Fensterklasse. Dort werden anhand der Parameter das benutzte Steuerelement und das eingetretene Ereignis identifiziert und verarbeitet. Die Botschaftenbehandlungsroutinen können somit nicht direkt aufgerufen und getestet werden! Es muss immer der "Umweg" über das Betriebssystem gewählt werden, um das jeweilige Fenster überhaupt adressieren zu können.

Weil damit jedoch nicht jeder Parameter der Nachrichtenübertragung von außen manipulierbar ist, kann nicht jeder mögliche Programmzustand durch Bedienung der Nutzeroberfläche erreicht werden. Es kann kein vollständig kontrollflussorientierter Test durchgeführt werden!

Da zudem alle vom Anwender sichtbaren und durchführbaren Aktionen der Nutzeroberfläche in der Programmspezifikation aufgeführt sind, sollte ein funktionaler Test/ Black-Box-Testverfahren angewandt werden. Alle Testfälle und -daten sind aus der Spezifikation ableitbar.

Die Anwendung wird dabei manuell bedient und das Verhalten vom Tester kontrolliert oder es wird ein **autarker Testtreiber** (siehe **[9]**, S. 19ff) verwendet. Dieser simuliert die Eingaben, steuert[20] das Programm fern und überprüft die ihm zugewiesenen Steuerelemente auf Inhalt und Zustand.

Für einen groben Vorabtest der Nutzeroberfläche muss die Funktionskomponente noch nicht vollständig getestet/ implementiert zur Verfügung stehen. In diesen Fällen kann eine "simulierte" Funktionskomponente (siehe **[3]**, S. 506, Platzhalter bzw. stubs) benutzt werden, bei der die einzelnen Methoden nur Standardwerte zurückgeben.

Unumgänglich bleibt der abschließende Feintest mit der wirklichen Funktionskomponente!

<u>Anwendungsfall: ‚Manuelle Justage'</u>

Zum Oberflächentest der neuen ‚Manuellen Justage' stand die Funktionskomponente bereits vollständig getestet zur Verfügung, so dass auf eine "simulierten" Komponente verzichtet wurde. Für den Test wurden die beiden oben angesprochenen Steuerungsarten angewandt. Zuerst wurde ein manueller Feintest durchgeführt, wobei die Testfälle mit CTE erfasst wurden. Für einen einfacheren, automatischen Regressionstest wurde anschließend der im Rahmen des

[20] zu Testreiber siehe , S. 414

‚XCtl'-Projekt entwickelte autarke Testtreiber – „Automatisches Testen oberflächenbasierter Systeme" = ATOS- Werkzeug (siehe [9], S. 30ff) – verwendet.

1) Ausführlicher Feintest

Vor dem Regressions- und Abnahmetest sollten Validation und Verifikation durchgeführt und die Vollständigkeit sichergestellt werden. Bei der Nutzeroberfläche bedeutet dies die korrekte, fehlerfreie und vollständige Verknüpfung der Steuerelemente nach den Spezifikationen des Pflichtenheft im Kapitels ‚Benutzeroberfläche'.

Dazu werden Testfälle definiert, die Handlungsanweisungen für den Tester vorgeben und ihn z.B. zu Eingaben auffordern. Durch deren sequentielle Abarbeitung sollen der Programmzustand kontrolliert geändert und Fehlersituationen provoziert werden.

Die Gesamtheit der Testdaten repräsentiert den aktuellen Programmzustand. Nach der Durchführung der Handlungsanweisungen eines Testfalls, muss der durch die Testdaten vorgeschriebene Zustand der Oberfläche vorliegen. Wenn das Testdatum bspw. vorschreibt, dass ein Steuerelement nach Abarbeitung des Testfalls gesperrt und ausgegraut sein soll, hat der Tester dies zu kontrollieren.

In Abhängigkeit von der Art des Steuerelementes wurden die Testdaten wie folgt ausgewählt:

1) Beschriftung von Steuerelementen

- Ist die Beschriftung eines Steuerelementes statisch, d.h. sie wird nicht durch den Programmcode geändert, sind dafür keine Testdaten nötig.
- Kann das Steuerelement nur eine abzählbare Menge an Beschriftungen anzeigen, sollte für jede einzelne Beschriftung ein Testdatum erstellt werden. Die Beschriftungen müssen durch die Analyse aller **/ B** <bb> /-Einträge im Kapitel ‚Benutzeroberfläche' des Pflichtenheftes ermittelt werden. Alle nutzeroberflächenspezifischen Funktionen sind zu überprüfen, weil jede das behandelte Steuerelement neu beschriften könnte. Bei Steuerelementen, wo die Beschriftungs- mit den Auswahlmöglichkeiten übereinstimmen (z.B. Kombinations- und Listenfelder), können die Testdaten aus dem zugeordneten ‚Daten'-Eintrag im Pflichtenheft abgelesen werden. Beispielsweise werden für das Kombinationsfeld *Antrieb auswählen* (**Abb. II.9**) die Antriebsbezeichnungen im zugehörigen Dateneintrag (**/D 30/**) beschrieben. Der oberflächenspezifische Eintrag ‚kein Antrieb' wird in **/B 10/** eingeführt. Für den Test wurden daraus drei Antriebe und ‚kein Antrieb' ausgewählt (**Abb. II.58**,). Möglich ist auch die Zusammenfassung von mehreren Testdaten zu einer Klasse, wenn laut Spezifikation keine funktionalen Unterschiede bestehen.
- Bei beliebigen Beschriftungsmöglichkeiten für ein Steuerelement (i.d.R. Eingabefelder) müssten für die Testdaten immer zwei Äquivalenzklassen (z.B. gültige/ ungültige Eingaben oder korrekte/ inkorrekte Ausgaben) definiert werden. Die Auswertung von Eingaben sollte jedoch nicht in der Nutzeroberfläche erfolgen, sondern in der Funktionskomponente. Diese prüft selbst, ob es sich um gültige Eingaben handelt, d.h. im gewünschten Wertebereich bzw. die Länge von Zeichenketten. Für die Behandlung ungültiger Zeichen in einer Eingabe (z.B. bei Zahlen) sollte die Basisklasse der Nutzeroberfläche Methoden bereits stellen. Damit entfällt die Auswertung dieser Steuerelemente fast komplett.

2) Systemzustände

- Zur Erleichterung der Testaktivitäten können Systemzustände mit in die Testdaten aufgenommen werden. Sie dienen dem Tester nur zur schnelleren Erfassung des aktuellen

Programmzustands. Weil im Hauptdialog der neuen ‚Manuellen Justage' z.B. viele Steuerelemente vom Antriebsstatus abhängen, kennzeichnet (**Abb. II.58**, ⬭) ob sich der Antrieb im aktuellen Zustand bewegen darf. Manchmal stehen diese Zustandsinformationen auch stellvertretend für die Beschriftung eines Steuerelementes. Bei **[M23]** steht ‚Halbwertsbreite' –> ‚wird gemessen' bspw. stellvertretend für die Beschriftung der Statuszeile, wo der Fortschritt der Messung angezeigt wird.

3) Zustand von Steuerelementen

- Zustandsunveränderliche Steuerelemente sind unabhängig vom aktuellen Systemzustand oder sollen sich nicht an diesen anpassen. Sie sind entweder immer freigegeben oder gesperrt und ausgegraut. Dafür sind keine Testdaten nötig.
- Zustandsveränderliche Steuerelemente sind im Kapitel ‚Benutzeroberfläche' des Pflichtenheft (**/B** <bb>/-Element) leicht am ‚Bedingung'-Absatz identifizierbar. Wenn das Steuerelement eine Produktanforderung realisiert, genügt auch ein ‚Bedingung'-Absatz bei der zugeordneten Produktanforderung (ein **/F** <ff>/-Element ist also vorhanden). Für solche Steuerelemente wurden die beiden Äquivalenzklassen:
  - „Funktion"-„ja" (Steuerelement soll im aktuellen Programmzustand freigegeben sein)
  - „Funktion"-„nein" (Steuerelement muss gesperrt und ausgegraut sein)

  definiert. Weil z.B. für das Kombinationsfeld *Antrieb auswählen* eine Bedingung (siehe **Abb. II.9** ⬭) angegeben ist, wurden die Testdaten **Abb. II.58**, ⬭ erstellt.

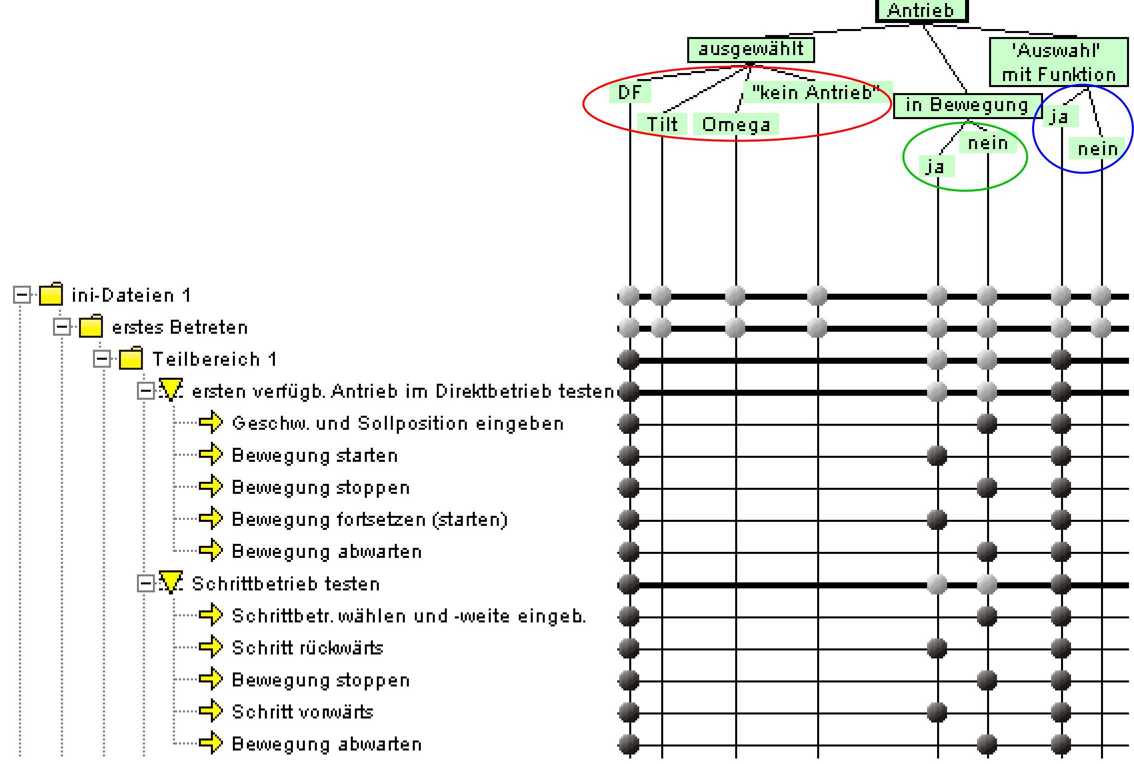


Abb. II.58 Ausschnitt aus dem CTE-Diagramm zum Test der Nutzeroberfläche der neuen ‚Manuellen Justage'

Die drei vorgestellten Kategorisierungen sind untereinander nicht disjunkt. Es ist demnach möglich, diese für ein Steuerelement zu kombinieren, um die Beschriftung, den Systemzustand und den Zustand des Elementes (freigegeben bzw. gesperrt und ausgegraut) gleichzeitig zu

beschreiben. In **Abb. II.58** ist das Steuerelement zur Auswahl eines Antriebs in Form eines Kombinationsfeldes dargestellt. Die Testdaten umfassen dort die aktuelle Beschriftung ( ), der Zustand ( ) und dem Zustand des Antriebs ( ).

Für den Hauptdialog wurden insgesamt 93 Testfällen mit jeweils 35 Testdaten definiert. Die ersten 57 Testfälle sichern, dass die Steuerelemente in den drei Teilbereichen vollständig, korrekt und fehlerfrei verknüpft sind. Es werden alle mit einer Funktion belegten Steuerelemente in allen drei Teilbereichen in verschiedener Reihenfolge getestet. Danach folgen 21 Testfälle, die Wechselwirkungen zwischen den Teilbereichen (siehe **[M23]**, z.B. ‚Antriebe in anderer Reihenfolge auswählen') und die Fehlertoleranz (**[M23]**, z.B. ‚versuchen Antrieb doppelt auszuwählen') testen. Dazu werden nacheinander die drei Teilbereiche bedient.

Dann wird das Dialogfenster verlassen und wieder betreten, um das Wiederherstellen der zuletzt benutzten Einstellungen (**/ B 10/**) zu überprüfen (12 Testfälle). Darin enthalten ist auch das Testen der Verfügbarkeit des PSD-Detektors.

Abschließend wird das Programm komplett beendet und mit einer anderen Konfigurationsdatei neu gestartet. Dabei wird das Wiederherstellen der zuletzt benutzten Einstellungen nach einem Neustart geprüft. Die Funktionen ‚PSD-Offset' und ‚Halbwertsbreite' dürfen aufgrund der geänderten Hardwarekonfiguration nicht mehr verfügbar sein (3 Testfälle).

Der gesamte Testvorgang wurde mehrfach durchgeführt. Bei den ersten beiden Tests war die Zustandsaktualisierung der Steuerelemente abgestellt, d.h. die Nutzeroberfläche dürfte keine `Can`-Methode aufrufen, um sich an Zustandsänderungen anzupassen. Der erste Test sollte überprüfen, ob die Nutzeroberfläche richtig mit der Funktionskomponente verknüpft ist. Dazu wurden die laut CTE-Diagramm freigegebenen Steuerelemente („Funktion"-„ja") mit gültigen Eingaben getestet. Diese mussten von der Funktionskomponente akzeptiert werden.

Nach dem erfolgreichen Durchlauf wurde ein zweiter Test mit der gleichen Implementierung durchgeführt. Nun wurden die laut CTE-Diagramm gesperrten und ausgegrauten Steuerelemente („Funktion"-„nein") benutzt. Folgende Aktionen wurden durchgeführt:

- In Eingabefelder wurden gültige Werte eingegeben. Aufgrund des Programmzustands mussten diese Eingaben von der Funktionskomponente ignoriert und die zuvor angezeigten Werte mussten wiederhergestellt werden.
- Schaltflächen wurden angeklickt. Die Ausführung der Produktfunktion musste von der Funktionskomponente ebenfalls ignoriert werden.
- Betriebsarten wurden ausgewählt, um zu kontrollieren, ob die zuvor ausgewählte Betriebsart wiederhergestellt wurde.

Beim dritten Test wurde die Zustandsaktualisierung der Steuerelemente aktiviert und ein vollständiger Test durchgeführt. Wenn die Schrittfolge eine Zahleneingabe beinhaltete, wurden zuerst unzulässig kleine, dann unzulässig große Werte eingegeben, um die Einpassung in den Wertebereich zu überprüfen. Abschließend wurde der geforderte, gültige Wert mit einer zu großen Nachkommastellengenauigkeit eingegeben, um die Rundung des Wertes zu überprüfen. Damit wurde nochmals die Robustheit der Funktionskomponente bewiesen und getestet.

Nach den angesprochenen Geschwindigkeitsoptimierungen wurde jeweils nur der dritte Test wiederholt.

2) Regressions-/ Abnahmetest

Für den Abnahmetest der neuen ‚Manuellen Justage', der durch die Mitarbeiter des Physikalischen Institutes durchgeführt wurde, mussten typische Arbeitsarbeitsabläufe ausgewählt werden. Weil das grundlegende Verhalten im Vergleich zum ursprünglichen Subsystem unverändert bleiben sollte, konnte auf die vorhandenen Regressionstestfälle (**[M18]**) zurückgegriffen werden. Diese deckten bereits ein repräsentatives Funktionsangebot ab und konnten nach der Erstellung von Testskripten für ATOS automatisch abgearbeitet werden. Ein integrierter Assistent ermöglichte eine einfache Erstellung der Skripte. Zur Adressierung der Steuerelemente wurden die Dialogressourcen verwendet. Trotz dieses Komforts war die Erstellung extrem zeitaufwendig, weil die Testfälle sehr umfangreich waren und die Reaktionszeiten auf verschiedenen Rechnern zu Timingproblemen führten. Vor den Geschwindigkeitsoptimierungen reagierte die Nutzeroberfläche nach Eingaben für Sekundenbruchteile nicht (siehe **II.4.2**). Dazu mussten Zeitverzögerungen, empirisch ermittelt, in die Testskripte integriert und getestet werden. Nach der ersten Geschwindigkeitsoptimierung wurde die Bildschirmaktualisierung parallel zu den Steuerelementen durchgeführt. Danach variierten die Wartezeiten von Test zu Test minimal, weshalb die Testskripte gewartet werden mussten, damit der Regressionstest nicht scheitert.

Die vier vorhandenen Testfälle wurden an das erweiterte Funktionsangebot (z.B. relative Positionspositionierung) und die Zustandsüberprüfung der Steuerelemente angepasst. Zudem mussten die Testskripte so auf die drei Teilbereiche verteilt werden, dass mit wenigen Testschritten nahezu und repräsentativ das vollständige Verhalten der Dialogfenster getestet werden kann. Um die Wechselwirkungen zwischen den Teilbereichen zu überprüfen, wurden zwei zusätzliche Testskripte eingefügt.

Diese sechs Testskripte entstanden am Ende der Analyse- und Definitionsphase. Sie wurden in dem separaten Dokument **[M18]** beschrieben, wo die vom Anwender abzuarbeitenden Schritte und die vom Programm durchgeführten Bildschirmaktualisierungen beschrieben werden (siehe **Abb. II.59**). In ‚Aktion' sind für jeden ‚Schritt' die Handlungsanweisungen angegeben. Die durch den Anwender zu kontrollierenden Bildschirmaktualisierungen sind in ‚Ereignisse und Ausgaben' fett hervorgehoben. In normaler Schrift werden Hinweise (z.B. Zustandsänderungen) gegeben.

Schritt	Aktion	Ereignisse und Ausgaben
0.	i. Auswahl der Antriebe ‚Azimutal Rot', ‚Beugung Grob' und ‚DF' im Kombinationsfeld *Antrieb auswählen* ii. Auswahl von ‚Direktbetrieb', ‚Fahrbetrieb' und ‚Schrittbetrieb' in den drei Teilbereichen iii. erneute Auswahl von ‚Beugung Grob' im Kombinationsfeld *Antrieb auswählen* des zweiten Teilbereichs	das Dialogfenster wird für die Testfälle vorbereitet
1.	Auswahl von ‚Tilt' im Kombinationsfeld *Antrieb auswählen*	i. Parameter von *Tilt* werden angezeigt ii. **‚Istposition' steht auf „0,0“ [arcmin]** iii. **‚aufheben' ist ausgegraut und gesperrt** iv. **‚Halbwertsbreite messen…' ist freigegeben**

Abb. II.59 Ausschnitt aus dem Testdokument **[M18]**, Testfall 1

Der automatisch ablaufende Regressionstest wird in unregelmäßigen Abständen für das gesamte System oder nach Wartungsarbeiten angewandt.

II.5.3 Zusammenfassung

Durch die strikte Aufgabenteilung zwischen Funktionskomponente und Nutzeroberfläche ist ein übersichtlicher und strukturierter Programmcode entstanden (siehe **II.4.3**), der die Fehlersuche und -korrektur erleichtern sollte.

Das bei der Funktionskomponente der ‚Manuellen Justage' angewendete Testverfahren ist zwar kontrollflussorientiert, bleibt aber selbst für einen C_0-Anweisungsüberdeckungstest ungenügend. Der Grund ist, dass die Implementierung der Funktionskomponente auch mögliche Probleme bei der Kommunikation mit der Hardware berücksichtigt. Da die verschiedenen Fehlerzustände nur durch die reale Hardware (z.B. durch Unterbrechung der Verbindung mit der Controllerkarte im richtigen Zeitpunkt) ausgelöst werden, kann der Test diese Fehlerbehandlung nicht abdecken! So können nicht einmal alle Knoten des Kontrollflussgraphen erreicht werden.

Trotzdem hat dieses Testverfahren, nach den Erfahrungen der Autoren, eine sehr stabile Funktionskomponente zum Ergebnis. Zudem sind Hardwareausfälle sehr seltene Fehler, bei denen der gesamte Messplatz untersucht und anschließend neu konfiguriert werden muss.

Beim Abnahmetest wird allgemein gegen die Spezifikation getestet, d.h. die eigentlichen Produktfunktionen werden verifiziert, validiert und auf Vollständigkeit getestet. Es stellt sich also die Frage nach der Notwendigkeit für den aufwendigen separaten Test der Funktionskomponente. Bei der neuen ‚Manuellen Justage', besteht die Notwendigkeit aufgrund der hohen Komplexität des Subsystems bei gleichzeitig hoher Robustheit gegenüber den verschiedenen Anwendergruppen. Auch die enge Hardwarekopplung bedingt diesen Test, weil die Fehlersuche beim Abnahmetest nicht nur in der Funktionskomponente und der Nutzeroberfläche, sondern auch in Teilen der Hardware stattfinden müsste.

Allgemein lässt sich feststellen, dass eine Funktionskomponente die eine hohe Komplexität, eine hohe Robustheit oder eine hohe Wahrscheinlichkeit der Wiederverwendbarkeit aufweist, stets separat zu testen ist. Weniger umfangreiche, aber trotzdem dekomponierte Subsysteme können durch einen Abnahmetest, wie hier vorgestellt, hinreichend verifiziert und validiert werden.

Der Feintest der Oberfläche ist zum einen abhängig von der Anzahl der Steuerelemente, die Zustandsänderungen auslösen (diese bilden Testfälle), und zum Zweiten von der Anzahl der Steuerelemente mit dynamischer Beschriftung (ergibt Testdaten). Pro zustandsveränderlichem Steuerelement werden zwei zusätzliche Testdaten benötigt. Der Mindestumfang des Feintests kann als Produkt aus Testfällen und Testdaten ganz grob abgeschätzt werden.

Der oben vorgestellte Feintest wäre damit viel zu umfangreich für einen automatischen Regressionstest. Allein 5.301 LOC ATOS-Testskript wären nach der Anzahl an Testfällen und Testdaten erforderlich! Aus Komplexitätsgründen ist die Überprüfung des Inhaltes der Steuerelemente dabei in der Hand des Testers. Pro dynamisch beschriftetem Steuerelement müsste also ein weiteres Testdatum integriert werden. Zudem besteht ein Testfall aus mindestens einer Handlungsanweisung, die in etwa mit LOC gleichgesetzt werden könnte. Die Wartung dieser Testskripte wäre damit nahezu unmöglich, weil auch die Wartezeiten ermittelt und aufgenommen werden müssten.

Der Regressionstest musste sich daher auf eine repräsentative Auswahl an Testfällen beschränken. Trotz des relativ geringen Umfangs von 931 LOC ATOS-Testskript konnte die Stabilität und Fehlerresistenz der sehr komplexen Nutzeroberfläche dennoch sichergestellt werden.

II.6 Fazit und Ausblick

In der Analysephase werden nur grundlegende Basisanforderungen an das zukünftige Produkt festgehalten. Damit ist diese Phase unabhängig von der Nutzeroberfläche. Die Dekomposition beginnt erst in der Definitionsphase. Nach der Sammlung der Produktanforderungen kann die durch die Dekomposition ermöglichte Arbeitsteilung beginnen. Während die Entwicklung der Oberflächen-Prototypen stattfindet, könnte sich der Anwendungsspezialist bereits mit den oberflächenunabhängigen Teilen des Produkt-Modells beschäftigen (im Wesentlichen die Erstellung des OOA-Modells). Erst hiernach kann das Pflichtenheft um die Nutzeroberfläche erweitert werden und Hilfe-Systeme sowie Benutzerhandbücher können entstehen.

Diese Arbeitsteilung wird in der Designphase fortgesetzt, indem das OOA-Modell zur späteren Funktionskomponente verfeinert wird und parallel die Oberflächenklassen entstehen. Zur Erstellung des OOD muss gemeinsam ein Kommunikationsmodell ausgewählt und an das jeweilige Design angepasst werden. Hier findet die eigentliche Dekomposition statt! Die Dokumentation kann dann wieder parallel erfolgen, weil jeder Entwickler nur seine Klassen beschreiben muss.

In der Implementierungsphase liegt der Schwerpunkt auf der Lösung des eigentlichen Anwendungsproblems. Die Kodierung von Funktionskomponente und Oberflächenklassen kann völlig unabhängig stattfinden. Die Programmierer müssen sich dazu nur an die im Design festgelegten Richtlinien halten. Während dieser Phase können fehlende Operationen der Funktionskomponente sichtbar werden, die der Nutzeroberflächenprogrammierer benötigt. Als Folge dessen stellt er Änderungswünsche an den Programmierer der Funktionskomponente, der diese Operationen bereitstellen und dokumentieren muss.

Der abschließende Test der Nutzeroberfläche ist nur mit einer lauffähigen Funktionskomponente möglich. Das bedeutet, dass diese vollständig implementiert und getestet sein muss. Steht diese noch nicht zur Verfügung, kann sicher der Tester der Nutzeroberfläche vorerst mir einer vereinfachten Funktionskomponente mit simulierten Eigenschaften (siehe **Abb. I.10**) begnügen oder abwarten. Prinzipiell besteht auch die Möglichkeit des Nutzeroberflächentests, obwohl die Funktionskomponente noch ungetestet ist. Voraussetzung hierfür ist jedoch der Abschluss der Implementierungsarbeiten. Diese Methode hat aber den Nachteil, dass auftretende Fehler nicht eindeutig der Nutzeroberfläche zuzuordnen sind. Die Dekomposition bietet zudem die Möglichkeit, Teile eines Sub- oder des Gesamtsystems strukturorientiert zu testen. Dies ist nur den Funktionskomponenten vorbehalten, denn für Nutzeroberflächen kann der Großteil von Methoden nicht einzeln aufgerufen und getestet werden! Somit lassen sind nicht dekomponierte Systeme nur auszugsweise kontrollflussorientiert testen!

Die vorgestellte Herangehensweise zur Dekomposition ist im Kern eine seit längerem bekannte und angewandte Technik. Dies beschränkt sich jedoch nur auf die Designphase. Problematisch ist hier, dass es unterschiedliche Auffassungen über die zu verwendenden Kommunikationsmuster gibt und dass versucht wird, eine Pauschalisierung für alle Anwendungszwecke vorzunehmen. Wie **II.3.3–5** zeigt, ist dies zwar möglich, aber nicht sinnvoll.

Die Autoren haben die Dekomposition auf den gesamten Softwareentwicklungsprozess erweitert und diesen ausgewertet. Um die eigene Arbeit und die nachfolgender ‚XCtl'-Entwickler zu erleichtern, wurden die entstandenen Dokumente stark formalisiert. Dies bietet die Möglichkeit, Teile solcher automatisch durch Werkzeuge, aus anderen Dokumenten oder aus Programmcode, erzeugen oder aktualisieren zu lassen oder umgekehrt Programmcode aus Dokumenten zu erzeugen. Die Einbeziehung solcher Funktionen in existierende Entwicklungswerkzeuge wird in Zukunft sicherlich an Bedeutung gewinnen.

Kapitel III

DEKOMPOSITION BEIM REENGINEERING

Mit steigender Komplexität der heutigen Softwaresysteme rückt der Einsatz des Reengineerings zunehmend auch ins Interesse der Projektmanager, weil Neuentwicklungen sehr zeitaufwendig und damit kostenintensiv sind. Zudem bleibt das System stets einsatzbereit, was Einsatzausfälle auf der Seite der Anwender weitgehend vermindert. **Abb. III.1** beleuchtet diese Aspekte auch aus der Sicht der universitären Lehre.

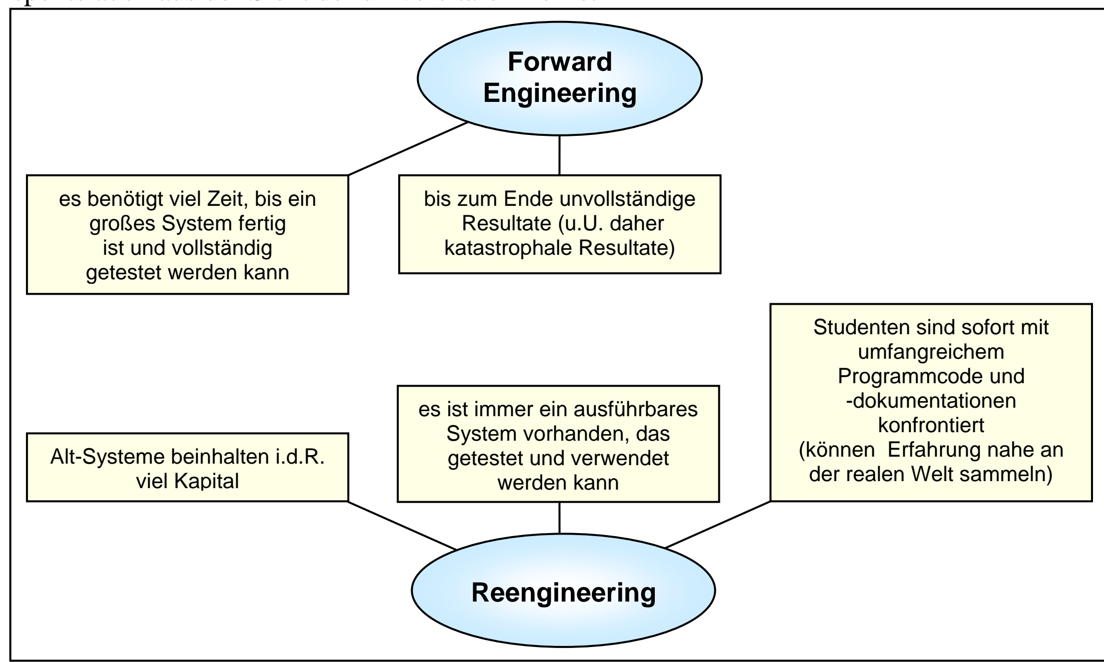


Abb. III.1 Vorteile des Reengineering gegenüber dem Forward Engineering
(z.T. zusammengetragen aus [4])

In diesem Kapitel geht es daher um die Erläuterung einer allgemeinen Schrittfolge für die Dekomposition von Funktionskomponente und Nutzeroberfläche eines objektorientierten

Subsystems beim Reengineering. Der Schwerpunkt liegt dabei auf den Änderungen am Programmcode. Diese werden zum leichteren Verständnis am Beispiel der Programmiersprache C++ erklärt, die auch im ‚XCtl'-Projekt eingesetzt wird. Aufgrund der Allgemeingültigkeit der verwendeten Sprachkonstrukte in der objektorientierten Welt und der weiten Verbreitung der Sprache sollte eine Übertragung, bspw. auf Java, keine Probleme bereiten. Zudem werden einige als optional gekennzeichnete Punkte ausgewiesen, die zur Dekomposition nicht zwingend erforderlich sind. Sie dienen nur zur Verbesserung der Struktur, Robustheit und Fehleranfälligkeit.

Diese Schrittfolge wurde aus der Dekomposition der Subsysteme ‚Manuelle Justage' und ‚Topographie' abgeleitet und verallgemeinert. Zur Veranschaulichung wird die Bearbeitung der Altsysteme in Anwendungsfällen dargestellt Dazu werden im Folgenden die vorhandenen Entwürfe als **ursprüngliche ‚Manuelle Justage'** und **ursprüngliche ‚Topographie'** und die im Reengineering dekomponierten als **getrennte ‚Manuelle Justage'** und **getrennte ‚Topographie'** bezeichnet.

III.1 Ist-Analyse

Zu Beginn des Reengineering muss das Altsystem analysiert werden. Falls keine entsprechenden Dokumente existieren, müssen diese in einem Reverse Engineering erstellt werden. Vorhandene Dokumente sind zu analysieren und u.U. zu aktualisieren.

Dies dient der Einarbeitung in den Gegenstandsbereich. Der Schwerpunkt sollte hierbei auf der Verhaltensspezifikation liegen, weil sie am besten den aktuellen Zustand des Systems widerspiegelt. Es können auch vorhandene Pflichtenhefte miteinbezogen werden, obwohl diese in der Praxis leider oft veraltet sind, weil ihre Wartung sehr aufwendig ist oder als überflüssig angesehen wird.

Zur Erfassung von Vererbungshierarchie und Beziehungstypen in einem OOD-Modell sollte jede Dokumentation der Designphase (z.B. UML-Diagramme) gesammelt und ausgewertet werden. Deshalb ist die Erstellung eines unter **II.3.4** beschriebenen Designdokumentes ratsam, weil neben den benötigten Informationen auch die Intentionen des damaligen OOD-Entwicklers enthalten sind.

Zur Bewertung von Design und Programmcode ist der Einsatz von Metriken denkbar. Dies kann helfen, Designfehler zu identifizieren und in der dekomponierten Version zu vermeiden.

Anwendungsfall: ‚Manuelle Justage'

Eine Ist-Analyse der ursprünglichen ‚Manuelle Justage' wurde bereits im Forward Engineering für die Erstellung der neuen ‚Manuelle Justage' vorgenommen (siehe **II.1**). Dies geschah phasenspezifisch, um die Effektivität zu steigern und die Komplexität besser bewältigen zu können. Hier lag der Schwerpunkt auf der Erstellung neuer und dem Studium der vorhandenen Artefakte der benutzten Subsysteme, weil die Interaktion mit diesen (Schnittstellen und Funktionsweise) im Mittelpunkt stand.

Am wichtigsten bei der Ist-Analyse im Reengineering ist das Verständnis des Programmcodes des zu dekomponierenden Subsystems selbst. Die Implementierung beschränkt sich auf das Dialogfenster ‚Manuelle Justage' (siehe **Abb. II.1**). `TAngleControlDlg` besteht aus 457 **LOC** (siehe **Tab. II.2**) mit sechs Nachrichtenbehandlungsroutinen, die aus der Basisklasse für Dialogfenster stammen und überschrieben wurden (**NOOM** 6). Die übrigen beiden Methoden (**NOO** 8) sind intern benutzte Methoden für Berechnung der Bildlaufleiste. Der Programmcode ist mit einer durchschnittlichen Methodengröße von 57 LOC entsprechend unübersichtlich, zumal die Steuerelement-basierten Nachrichten in der Methode `Dlg_OnCommand` auf 330 LOC behandelt werden.

Das Subsystem ursprüngliche ‚Manuelle Justage' ist abgeschlossen, weil kein anderes System auf ihren Programmcode zugreift. Damit ist die öffentliche Kennzeichnung aller Methoden mit public (**PPubM** 45) weder notwendig noch sinnvoll. Die elf Attribute (**NOA** 11) sind jedoch vollständig geschützt (**PPrivM** 55).

Zur Ansteuerung der Antriebe und Detektoren und zur Messung der Halbwertsbreite werden andere Subsysteme verwendet. Die dazu benutzten Methodenaufrufe dienen der Realisierung der Funktionalität und bilden etwa 50 % des Programmcodes. Trotz des guten Kommentierungsgrades (**TCR** 25) sind diese Aufrufe kaum beschrieben. Die Aktualisierung der Oberfläche hingegen ist fast "zu ausführlich" und der Zugriff auf Attribute und Methoden innerhalb der ‚Manuellen Justage' ist teilweise sogar falsch kommentiert.

Da der Programmcode durch die Schrittfolge in mehreren Pässen intensiv analysiert und schließlich modifiziert wird, ist genaueres Wissen hier noch nicht sinnvoll. Nur ein allgemeiner Überblick über das aktuelle Design (siehe **[M24]**) und, wie beim Forward Engineering auch, die

Anbindung der benutzten Subsysteme – ‚Motorsteuerung' ([M28]), ‚Ablaufsteuerung' ([M6]) ‚Detektornutzung' ([M12]) – ist zu diesem Zeitpunkt erforderlich.

Anwendungsfall: ‚Topographie'

Auch das Subsystem ‚Topographie' wurde einem Reengineering zur Dekomposition von Nutzeroberfläche und Funktionskomponente unterzogen. Die ursprüngliche ‚Topographie' umfasst zwei Dialogfenster (siehe **Abb. III.2**). Der Dialog ‚Topographie-Einstellungen' dient zur Eingabe verschiedener Parameter für den Detektor und den Antrieb, am Anfang und während einer Messung. Im Dialog ‚Topographie-Durchführung' wird der eigentliche Messvorgang durchgeführt und überwacht.

Zu Beginn der Ist-Analyse wurde ein Review der im Repository des ‚XCtl'-Projektes vorhandenen Dokumente (siehe **Tab. III.2**) zur Topographie durchgeführt. Das Hauptdokument der Analysephase ist **[M29]** („Topographie Gesamtvorgang"). Hier werden die physikalischen Grundlagen und die theoretischen Abläufe der einzelnen verfügbaren Topographie-Messvorgänge, zum Teil formalisiert in einer Pseudocodenotation, beschrieben. Die Verhaltensspezifikationen **[M31]** („Einstellen der Parameter für die Topographie") und **[M32]** („Start und Kontrolle der Topographie") erläutern die Durchführung der Messungen anhand der Nutzeroberflächen (siehe **Abb. III.2**, rechts bzw. links). Zur Protokollierung einer Messung zeigt **[M33]** („Topographie-Meßprotokoll_Buch") die allgemeine Verfahrensweise.

Für die Designphase existiert das Dokument **[M29]** („Software-Struktur"). Es beinhaltet die zum Subsystem gehörigen Dateien und ein Klassendiagramm zum Oberflächendesign, mit einer kurzen Erklärung der Aufgaben der einzelnen Nutzeroberflächen. Da die ‚Topographie' auf die Subsysteme ‚Detektorsteuerung', ‚Motorsteuerung' und ‚Ablaufsteuerung' zurückgreift, wird die Nutzung dieser dort ebenfalls beschrieben. Dazu werden ein Klassendiagramm des Subsystems ‚Ablaufsteuerung', zur automatischen Durchführung einer Messung, und die Anbindung des Subsystems ‚Motorsteuerung' mittels einer (nicht-objektorientierten) C-Schnittstelle, erläutert.

Das Subsystem ‚Topographie' besteht aus der Dialogklasse `TTopographyExecuteDlg` für das modale Dialogfenster ‚Topographie-Durchführung', der Klasse `TTopographySetParamDlg` für das ebenfalls modale Dialogfenster ‚Topographie-Einstellungen' (siehe **Abb. III.2**) und dem Ansatz einer Funktionskomponente in Form der Klasse `TTopographyOld`. Alle drei Klassen sind in der Header-Datei TOPOGRFY.H deklariert und in der Datei M_TOPO.CPP implementiert.

Eine Besonderheit ist, dass der Dialog ‚Topographie-Einstellungen' sowohl über einen Menüpunkt im ‚XCtl'-Hauptmenü (‚Einstellungen' → ‚Topographie…') als auch über die Schaltfläche ‚Einstellungen…' im Dialog ‚Topographie-Durchführung' aufgerufen werden kann. Geschieht dies auf die letztgenannte Weise, sind ein Teil der Eingabefelder, die zur Eingabe von Parametern vor einer Messung dienen, gesperrt und ausgegraut. Die übrigen gestatten, die Einstellungen für den Detektor während eines Messvorganges zu ändern.

Die Implementierung der ursprünglichen Topographie umfasst insgesamt 738 LOC ohne Leer- und Kommentarzeilen. Keines der anderen ‚XCtl'-Subsysteme greift auf den Programmcode dieser Komponente zu, d.h. das System ist abgeschlossen.

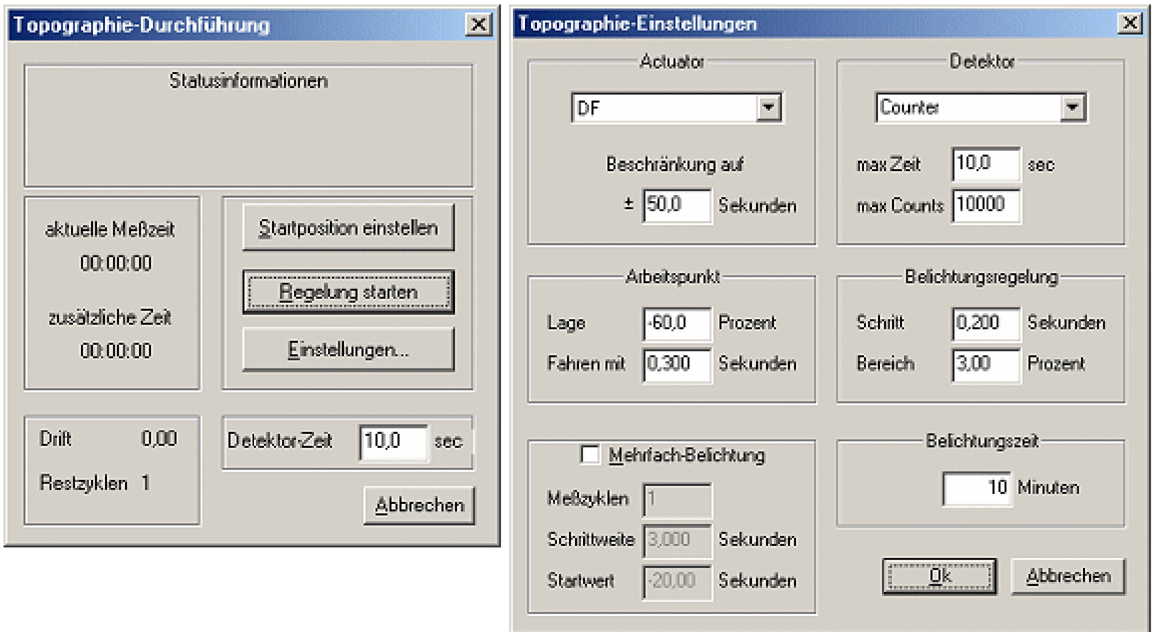


Abb. III.2 Dialoge ‚Topographie-Durchführung' (links) und ‚Topographie-Einstellungen' (rechts)

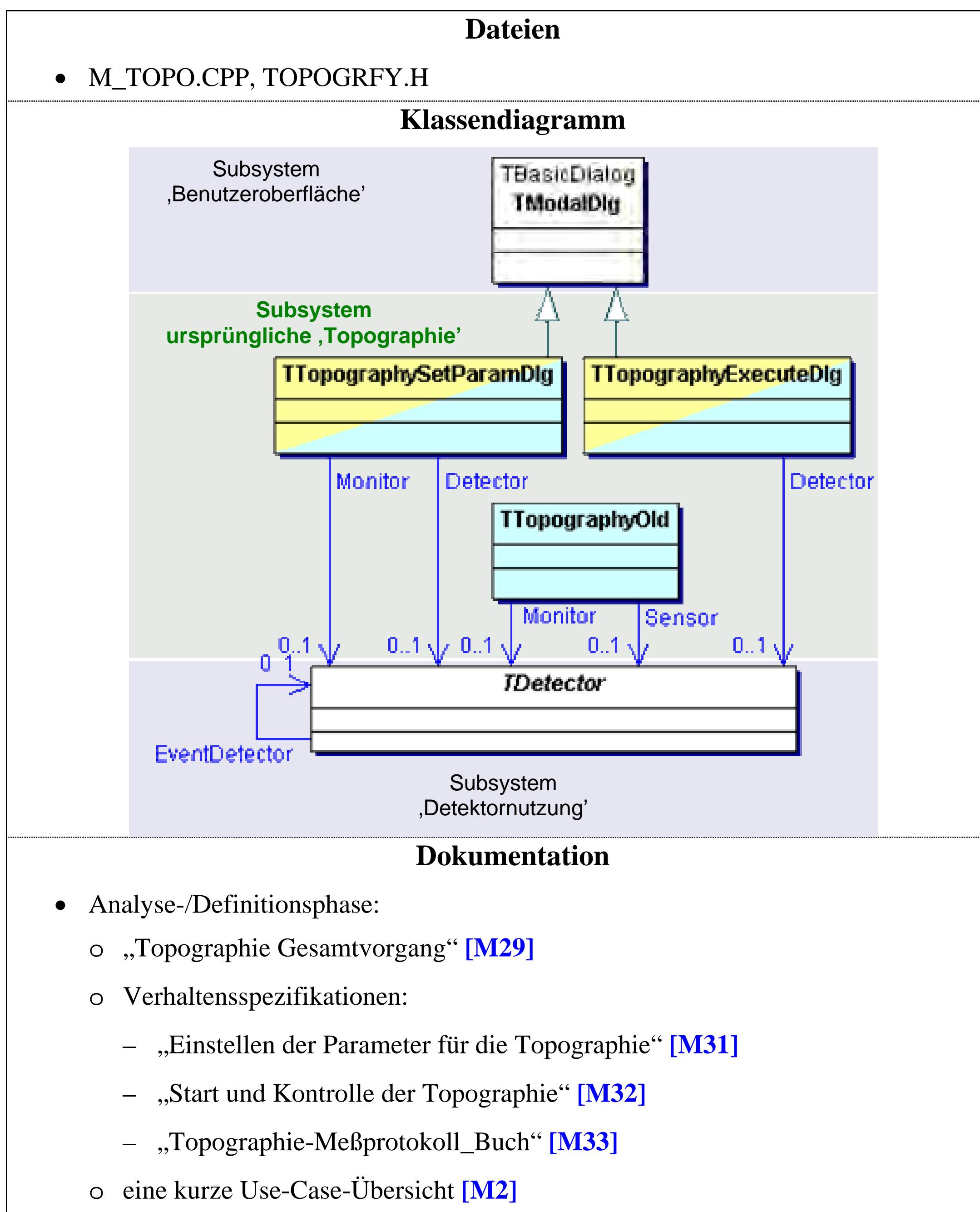
Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TTopographyOld	83	18	1	1	0	0	**90**	10	83	0	1	48
TTopographyExecuteDlg	418	13	4	1	4	72	0	28	101	4	1	35
TTopographySetParamDlg	237	7	4	1	4	50	0	50	41	4	1	46

Tab. III.1 Ausgewählte Metriken für die Klassen der ursprünglichen ‚Topographie'

Dateien

- M_TOPO.CPP, TOPOGRFY.H

Klassendiagramm



Dokumentation

- Analyse-/Definitionsphase:
  - „Topographie Gesamtvorgang" **[M29]**
  - Verhaltensspezifikationen:
    - „Einstellen der Parameter für die Topographie" **[M31]**
    - „Start und Kontrolle der Topographie" **[M32]**
    - „Topographie-Meßprotokoll_Buch" **[M33]**
  - eine kurze Use-Case-Übersicht **[M2]**
- Design-/Implementierungsphase:
  - „Software-Struktur" **[M29]**
- Testphase:
  - Regressions-Testfälle „Topographie" **[M36]**
- Fehlerübersicht:
  - tabellarische Übersicht aller gefundenen Fehler seit 22.6.2000 **[M34]**

Tab. III.2 Übersicht zur ursprünglichen ‚Topographie'

III.2 Restructuring

Bei der Dekomposition eines vorhandenen Systems ist eine phasenorientierte Bearbeitung nicht von Vorteil bzw. nicht möglich. Die Analyse und Definition werden bei dieser Dekomposition nur zur Einarbeitung in den Gegenstandsbereich benötigt. Die Anforderungen bleiben während der Dekomposition unverändert, Änderungen müssen nach dem eigentlichen Prozess erfolgen!

Die Designphase ist nicht explizit durchführbar, weil die Dekomponierbarkeit erst nach Änderungen am Programmcode festgestellt werden kann. Dies umfasst die Bestimmung und ggf. Veränderung der Kardialität von Funktionskomponenten und Nutzeroberflächen (siehe **III.2.1**, **2)**) sowie Änderungen an den Beziehungstypen und der Vererbungsstruktur. Zudem ergeben sich Anzahl, Funktion und Benennung von Methoden erst am Ende der Dekomposition aus der resultierenden Implementierung. Ursprüngliche Zuordnungen der Attribute zu den Klassen können sich dann geändert haben. Das Design entwickelt sich also indirekt während der Anwendung der Schrittfolge.

Die Implementierungsphase bildet den Kern der Dekomposition. Sie beinhaltet Aspekte des Designs und ständige Tests, um den korrekten Verlauf der Dekomposition sicherzustellen. Die Schrittfolge ist so gestaltet, dass sich Testaktivitäten und Implementierung ständig abwechseln. Jeder Implementierungsschritt wird durch eine Testphase abgeschlossen, um so Folgefehler zu vermeiden, neue Fehler früher zu erkennen und zu beheben.

III.2.1 Allgemeine Schrittfolge für die Dekomposition

1) Erstellung neuer Dateien

Wie beim Forward Engineering sind auch hier Implementierungs- und Header-Dateien für Funktionskomponente und Nutzeroberfläche getrennt zu erstellen. Die Benennung dieser vier Dateien sollte die Zugehörigkeit zu einem Subsystem widerspiegeln und anzeigen, ob es sich um Dateien für die Funktionskomponente bzw. Nutzeroberfläche handelt.

Die in der Ist-Analyse für das zu dekomponierende Subsystem identifizierten Klassen werden entsprechend ihrer Funktion in die verantwortliche Datei kopiert. Genauso ist mit den `include`s zu verfahren. Um die ursprünglichen Klassen zu erhalten, werden diese vorerst mit dem Suffix `OLD` gekennzeichnet (umbenannt) und in den alten Dateien belassen. Damit das gesamte Projekt nun stets die zukünftig getrennte Variante benutzt, müssen die `include`s anstatt auf die alten Dateien nun auf die neuen Dateien zeigen. Zudem muss die Datei mit der Deklaration der Funktionskomponente in der Header-Datei der Nutzeroberfläche inkludiert werden. Die alten Dateien werden nun nicht mehr benutzt, sichern jedoch weiterhin die ursprüngliche Implementierung.

✓ Nachdem die neuen Dateien auch in die Projekt-/ Makedatei aufgenommen wurden, kann das Gesamtsystem kompiliert und gelinkt werden. Noch nicht bearbeitete `include`s werden dabei immer erkannt, weil die Klassen im alten Subsystem nicht mehr gefunden wurden.

Jetzt müssen die Klassen in den neuen Dateien neu benannt werden. Empfohlen wird hier wieder die Kennzeichnung der Oberflächenklassen mit den Suffixen `Dlg` oder `Wnd` (siehe **II.3.2**).

✓ Beim Kompilieren und Linken wird jede Stelle angezeigt, wo Klassen noch mit der alten Benennung benutzt wird. Überall dort muss der alte Bezeichner durch den neuen ersetzt werden.

Zur Wahrung der Konsistenz von ursprünglichem Programmcode und existierender Dokumentation werden nun die mit `OLD` gekennzeichneten Klassen zurück benannt. Da diese

Klassen aber nicht benutzt werden, ist ein abschließendes Kompilieren und Linken damit nicht erforderlich, aber allgemein stets sinnvoll.

2) Behandlung von Ansätzen vorhandener Funktionskomponenten

Existieren mehrere von der Nutzeroberfläche unabhängige Klassen, stellen diese **Ansätze von Funktionskomponenten** dar. Ist noch kein solcher Ansatz vorhanden, muss in der entsprechenden Header-Datei ein leerer Klassenrumpf für die Funktionskomponente deklariert werden. Ansonsten ist eine Dekomposition nur dann problemlos möglich, wenn genau eine Funktionskomponente existiert.

Wenn mehrere vorhanden sind, gilt es diese zu einer zu verschmelzen. Sonst wird die spätere Auslagerung von Programmteilen in eine der Funktionskomponenten u.U. durch die Verteilung der Attribute verhindert. Die Verschmelzung ist jedoch nicht möglich, wenn mehrere Oberflächenfenster des zu dekomponierenden Subsystems gleichzeitig angezeigt werden können und jedes Fenster dabei eine eigene Instanz einer Funktionskomponente benutzt. Wenn Bezeichner (Attribute und Methoden) mehrfach vorhanden sind, müssen diese vor der Verschmelzung in einer Klasse umbenannt werden.

✓ Dies wird durch den Compiler unterstützt, der jede Stelle anzeigt, wo die Umbenennung von Attributen und Methoden unterschlagen wurde.

Anschließend können die Funktionskomponenten problemlos in einer Klasse vereinigt werden. Überall im System müssen die alten Klassenbezeichner durch den der neuen Klasse ersetzt werden. Die verschmolzene Funktionskomponente muss genau zum gleichen Zeitpunkt wie die ursprünglichen Komponenten im System initialisiert werden!

✓ Beim Finden der alten Klassenbezeichner hilft wieder der Compiler. Ein Regressionstest aller **benutzenden Subsysteme** verifiziert, dass durch die Verschmelzung keine schwerwiegenden Fehler entstanden sind. Der Test muss alle Anwendungsfälle des zu dekomponierenden Subsystems und aller benutzenden Systeme beinhalten!

3) Solldesign festlegen

Zu Beginn der eigentlichen Dekomposition muss das zukünftige Basisdesign durch die Wahl eines Kommunikationsmusters festgelegt werden (siehe **II.3.3**). Die Entscheidung sollte nach den in Kapitel **II.3.3–5** vorgestellten Kriterien getroffen werden. Dies zieht i.d.R. die Deklaration und Initialisierung neuer Attribute, die Einführung neuer Methoden und u.U. die Einerbung eines Interfaces in den Klassen der Nutzeroberfläche nach sich. Wenn bereits ein Ansatz einer Funktionskomponente existierte, empfiehlt es sich das existierende Kommunikationsmuster beizubehalten. So werden später Änderungen am Programmcode gering gehalten, damit keine zusätzlichen Fehlerquellen entstehen.

Zudem sollte die Einführung des Singleton-Musters (**II.3.3–2**) forciert werden, wenn sicherzustellen ist, dass zu jedem Zeitpunkt nur maximal eine Instanz der Funktionskomponente existieren darf. Dabei dürfen die unter **2)** genannten Kriterien über die Initialisierung der Komponente nicht verletzt werden.

✓ Nach dem Kompilieren und Linken ist ein Regressionstest auch der benutzenden Subsysteme durchzuführen.

4) Programmcodelayout verbessern (optional)

Zur Verbesserung der Lesbarkeit sollte der Programmcode formatiert, kommentiert und potentielle Fehlerquellen markiert werden. Toter Code ist auszukommentieren und mit dem aktuellen Datum zu kennzeichnen. Sicherzustellen ist ferner, dass keine uninitialisierten Attribute, lokale oder globale Variablen verwendet werden.

✓ Kompilieren, Linken und ein Regressionstest der benutzenden Subsysteme stellt sicher, dass keine Fehler unterlaufen sind.

Um Übersichtlichkeit und Lesbarkeit der Nutzeroberfläche weiter zu verbessern, ist die Behandlung von Steuerelementbotschaften in einzelne Methode auszulagern. Dazu muss der Programmcode durch den Aufruf einer neuen `On`-Methode ersetzt werden. Der ersetzte Code ist dann in diese auszulagern. Der Name der Methode sollte sich aus dem Ressourcen-Bezeichner der Botschaft und dem Präfix `On` zusammensetzen. Alle benötigten lokalen Variablen und Parameter sollten von der Behandlungsroutine[21] per Referenz an die `On`-Methode übergeben werden. Bei reinem Lesezugriff kann die Referenz durch `const` ersetzt werden. Wenn die Variable nur in dieser Methode benutzt wird, können Deklaration und Initialisierung dorthin verschoben werden. Eine Ausnahme bildet das Fenster-Handle, das nicht an die `On`-Methode weiterzugeben ist. Stattdessen sollte eine Methode der Fensterbasisklasse verwendet werden, die das Fenster-Handle zurückgibt – ggf. ist eine solche zu erstellen. Erfolgt die Behandlung aller Botschaften in einer `switch`-Abfrage, ist in jedem ausgelagerten `case`-Zweig das `break` durch `return` zu ersetzen.

✓ Die genannten Anpassungen können durch Kompilieren und Linken verifiziert werden. Ein erfolgreicher Regressionstest des zu dekomponierenden Subsystems und der benutzenden Systeme bildet den Abschluss dieses Schritts.

5) Neuordnung der Attribute

Um den Zustand des zu dekomponierenden Subsystems ausschließlich in der Funktionskomponente zu verwalten, müssen alle Attribute, die keine echten Eigenschaften der Oberfläche sind, von der Nutzeroberfläche in die Funktionskomponente verschoben werden. Die Initialisierung aller Attribute sowie die Erzeugung aller dynamischen haben dann im Konstruktor der Funktionskomponente zu erfolgen. Die Nutzeroberfläche darf nur Attribute enthalten, welche Eigenschaften wie Größe und Layout – allgemein den Zustand von Fenstern und Dialogen – kennzeichnen. Auch Referenzen auf Fenster oder Fenster-Handles verbleiben in der Oberfläche. Alle Attribute sind mit dem höchsten Zugriffsschutz zu versehen, egal ob in der Funktionskomponente oder der Nutzeroberfläche.

Daraufhin sind in der Funktionalität Mutator-Methoden für den direkten Schreib-Zugriff und Accessor-Methoden für den Lese-Zugriff zu implementieren. Sie bilden einen Teil der öffentlichen Schnittstelle der Funktionskomponente. Ihre Benennung sollte nach Kapitel **II.2.5 b)** erfolgen. Falls oberflächenspezifische Attribute in einer bereits vorhandenen Funktionskomponente enthalten sein sollten, sind diese in die entsprechende(n) Nutzeroberfläche(n) zu verschieben, zusammen mit deren Initialisierung. Dort sind sie mit dem höchsten Zugriffschutz zu versehen. Keine Methode der Funktionskomponente darf auf solche Attribute zugreifen, ggf. sind solche Methoden in die Nutzeroberfläche zu verschieben.

✓ Diese Umsortierungen werden durch den Compiler und Linker unterstützt. Zur Sicherheit sollte sich ein Regressionstest der oben benannten Subsysteme anschließen.

6) Zusammenfassen von Funktionalitätsaufrufen

Nur der bestehende Programmcode der Nutzeroberfläche ist noch in funktions- und oberflächenspezifische Anweisungsfolgen zu unterteilen. Alle von der Gestaltung der Oberfläche (z.B. von der Wahl der Art der verwendeten Steuerelemente) unabhängigen Anweisungsfolgen müssen in die Funktionskomponente ausgelagert werden. Dies sind:

[21] alle Methoden wo Botschaften für die Nutzeroberfläche ankommen und ausgewertet werden (z.B. `OnCommand`)

- Lesezugriffe auf Attribute der Funktionskomponente nur, wenn sie nicht der Aktualisierung der Nutzeroberfläche dienen; Schreibzugriffe nur, wenn sie nicht der Datenübernahme von Eingaben dienen
- Steuerung von Hardware oder Benutzung anderer Subsysteme
- Berechnungen und Simulationen
- Zustands- und Datenverwaltung (sofern unabhängig von der Nutzeroberfläche)

Alle übrigen Anweisungen sind eindeutig oberflächenspezifisch:

- Lese- und Schreibzugriffe auf Attribute der Nutzeroberfläche
- alle Anweisungen die ein Fenster-Handle als Parameter benötigen
- Meldungsfenster und sonstige texturelle Ausgaben
- alle Aufrufe von Methoden aus Oberflächenklassen
- Anweisungen zur Steuerung des Mauszeigers (z.B. Sanduhr)
- Akustische Ausgaben (z.B. Klänge, Warntöne)

Der Beginn einer funktionsspezifischen Anweisungsfolge (**F-Block**) ist nun mit dem Kommentar „`/* F */`" und der einer oberflächenspezifischen (**O-Block**) mit dem Kommentar „`/* O */`" zu kennzeichnen. Es gilt, die so entstandenen F-Blöcke im Programmcode zusammenzufassen, um die Anzahl der späteren `Do`-Methoden in der Funktionskomponente zu reduzieren. Dies wird durch die Verschiebung der O-Blöcke, vor oder hinter einen F-Block, erreicht.

a) Das Verschieben darf nie in eine höhere oder tiefere Ebene eines Anweisungsblocks erfolgen.

b) Der zu überspringende F-Block darf weder direkt noch indirekt[22] auf dieselben Attribute, Parameter, lokale und globale Variablen wie der O-Block zugreifen, es sei denn, beide greifen nur lesend zu.

c) Das Verschieben ist auch verboten, wenn der zu verschiebende oder der zu überspringende Block zum Verlassen eines Anweisungsblocks führt.

d) O-Blöcke, die erst nach der erfolgreichen Ausführung eines F-Blockes durchgeführt werden dürfen – z.B. Benennung von Steuerelementen, nachdem eine Aktion ausgelöst wurde – können nicht verschoben werden.

Ziel ist es, so viele O-Blöcke wie möglich, an nicht-verschiebbaren O-Blöcken oder am Anfang oder Ende des Anweisungsblocks zusammenzufassen. F-Blöcke, die durch die Verschiebung eines dazwischen liegenden O-Blocks direkt untereinander stehen, bilden eine neue Einheit. Für zusammenfallende O-Blöcke gilt das Gleiche.

Abb. III.3 und **Abb. III.4** sollen die o.g. Regeln verdeutlichen, die eine Umordnung der O-Blöcke verhindern. Dazu wird jeweils geprüft, ob der O-Block vor den davor liegenden oder nach den darauf folgenden F-Block verschoben werden kann. Bei einem roten Pfeil (→) ist dies nicht möglich. Dann ist der Hinderungsgrund (**a** bis **d**) angeben und das verantwortliche Programmcodefragment fett hervorgehoben. In **Abb. III.3** ist ein Beispiel enthalten, wo die Anzahl der F- und O-Blöcke reduziert werden kann (→), indem die Anweisung zum Ändern des Mauszeigers in eine Sanduhr umsortiert wird.

[22] Sicherzustellen ist, dass KEINE Anweisungsfolge auf das Attribut schreibend zugreift.

```
void TThrowSimDlg::OnStartThrow() { // Simulation zum schrägen Wurf
    /* F */
    if ( m_ThrowModule->IsBusy() ) return; // Simulation ist aktiv

    /* O */
    bool valid= false;
    double angle= m_AngleEditfield.GetDoubleValue( &valid );
    if ( valid==false ) {
      ShowErrDlg( „Eingabe 'Abwurfwinkel' prüfen!" );
      return;
    }
    double speed= m_SpeedEditfield.GetDoubleValue( &valid );
    if ( valid==false ) {
      ShowErrDlg( „Eingabe 'Anfangsgeschwindigkeit' prüfen!" );
      return;
    }

    /* F */
    m_lnkFunk->SetStartAngle( angle );
    m_lnkFunk->SetStartSpeed( speed );

    /* O */
    ::SetCursor( ::LoadCursor(GetInstance(), IDC_WAIT) ); // Sanduhr

    /* F */
    … // Berechnungen für schrägen Wurf
}
```



```
void TThrowSimDlg::OnStartThrow() { // Simulation zum schrägen Wurf
    /* F */
    if ( m_ThrowModule->IsBusy() ) return; // Simulation ist aktiv

    /* O */
    bool valid= false;
    double angle= m_AngleEditfield.GetDoubleValue( &valid );
    if ( valid==false ) {
      ShowErrDlg( „Eingabe 'Abwurfwinkel' prüfen!" );
      return;
    }
    double speed= m_SpeedEditfield.GetDoubleValue( &valid );
    if ( valid==false ) {
      ShowErrDlg( „Eingabe 'Anfangsgeschwindigkeit' prüfen!" );
      return;
    }
    ::SetCursor( ::LoadCursor(GetInstance(), IDC_WAIT) ); // Sanduhr

    /* F */
    m_lnkFunk->SetStartAngle( angle );
    m_lnkFunk->SetStartSpeed( speed );
    … // Berechnungen für schrägen Wurf
}
```


Abb. III.3 Konstruiertes Beispiel zur Verschmelzung zweier O-Blöcke
(der entstandene F-Block kann als Ganzes als `Do`-Methode ausgelagert werden)

```
void TInkrementalCalcDlg::OnStartCalc() { // laufzeitabhängige Ergebnisgenauigkeit
  /* O */
  bool valid= false;
  int repeatCount= m_RepeatEditfield.GetIntValue( &valid );
  if ( valid==false ) {
    ShowErrDlg( „Eingabe prüfen!“ );
    return;
  }

  /* F */
  if ( (repeatCount<0) || (repeatCount>199) )==false ) {        a)
    /* O */                                                      b)
    ShowErrDlg( „Wertebereich prüfen!“ );                        c)   b)
    return;                                                           c)
  }
  double result= m_lnkFunk->DoStartCalc( repeatCount );         a)

  /* O */
  m_ResultEditfield.SetDoubleValue( result );
}
```

Abb. III.4 Konstruiertes Beispiel mit nicht verschiebbaren O-Blöcken
(hier werden auch die untergeordneten, bedingten Anweisungsfolgen untersucht)

✓ Die Verifizierung erfolgt durch Kompilieren, Linken und einen abschließenden Regressionstest der benutzenden Subsysteme.

7) Erzeugen von Do- Methoden

Jeder F-Block ist in eine neue öffentliche Do-Methode (Name ist aussagekräftig zu wählen) der Funktionskomponente zu verschieben. Der verschobene F-Block ist in der Nutzeroberfläche durch einen Aufruf der entstandenen Do-Methode zu ersetzen. Je nach Typ werden die folgenden Signaturen notwendig:

Typ 1: Die Do-Methode wird nicht frühzeitig verlassen und hat keinen Rückgabewert.

```
void TThrowSimDlg::OnStartThrow() {
  … // angle, speed deklarieren und mit Eingaben aus Nutzeroberfläche initialisieren, Sanduhr anzeigen
  /* F */
  m_lnkFunk->SetStartAngle( angle );
  m_lnkFunk->SetStartSpeed( speed );
  … // Berechnungen für schrägen Wurf
}
```

AUSGANGSSITUATION

```
void TThrowSimDlg::OnStartThrow() {
  … // angle, speed deklarieren und mit Eingaben aus Nutzeroberfläche initialisieren, Sanduhr anzeigen
  m_lnkFunk->DoStartThrow( angle, speed );
}

void TThrowSim::DoStartThrow(double angle, double speed ) {
  SetStartAngle( angle ); // “m_lnkFunk->“ entfält innerhalb der Funktionskomponente
  SetStartSpeed( speed );
  … // Berechnungen für schrägen Wurf
} // ausgelagert aus TThrowSimDlg::OnStartThrow()
```

ERGEBNIS

Abb. III.5 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine Do-Methode (Typ1)

Typ 2: Wird die ausgelagerte Methode frühzeitig ohne Rückgabewert verlassen, ist `false` und am Ende des Methodenrumpfes `true` zurückzugeben. Die aufrufende Methode darf nur bei `true` fortgesetzt werden und ist bei `false` zu verlassen.

```
void TDialupWizardDlg::OnDialup() {
  … // number, user, password deklarieren und mit Eingaben aus Nutzeroberfläche initialisieren
  /* F */
  if ( RasIsDeviceBusy() ) return; // Leitung ist belegt
  CRasDialupPacket packet( number );
  packet.UserName= user;
  packet.Password= password;
  RasSendPacket( packet );
  … // Wählfortschritt im Fenster anzeigen
}
```

AUSGANGSSITUATION

```
void TDialupWizardDlg::OnDialup() {
  … // number, user, password deklarieren und mit Eingaben aus Nutzeroberfläche initialisieren
  if ( m_lnkFunk->DoDialup( number, user, password )==false ) return;
  … // Wählfortschritt im Fenster anzeigen
}

bool TDialup::DoDialUp( String number, String user, String password ) {
  if ( RasIsDeviceBusy() ) return false; // Leitung ist belegt
  CRasDialupPacket packet( number );
  packet.UserName= user;
  packet.Password= password;
  RasSendPacket( packet );
  return true;
} // ausgelagert aus TDialupWizardDlg::OnDialup()
```

ERGEBNIS

Abb. III.6 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine `Do`-Methode (Typ2)

Typ 3: Sonst wird die neue `Do`-Methode mit einem Rückgabewert verlassen. Dann muss ein neuer Referenzparameter (siehe **Abb. III.7**, `aValid`) zur `Do`-M. hinzugefügt werden. Dieser wird am Anfang mit `false` initialisiert. Er wird nur am Methodenende auf `true` gesetzt und kennzeichnet damit, dass die Methode bis zum Ende ausgeführt wurde. Die rufende Methode muss diesen Parameter auswerten und sich bei `false` selbst mit dem Ergebnis der gerufenen Methode (siehe **Abb. III.7**, `result`) verlassen. Nur bei `true` darf fortgesetzt werden.

```
EStatus TMotorSteeringDlg::StartMove( double speed ) {
  /* F */
  if ( !m_Motor->IsMoving() ) { // Motor ist nicht in Bewegung
    if ( !m_Motor->SetSpeed(speed) ) return SpeedErr; // unzulässige Geschwindigkeit
  } else return MoveErr; // ist in Bewegung
  m_Motor->StartMove(); // Bewegung starten
  /* O */
  m_Statusfield.SetText(„Motor wurde gestartet…“);
  return NoErr; // Fehlerstatus zurück an aufrufendes Fenster
}
```

AUSGANGSSITUATION

```
EStatus TMotorSteeringDlg::StartMove( double speed ) {
  bool valid;
  EStatus result= m_Motor->DoStartMove( speed, &valid );
  if ( !valid ) return result; // die Do-Methode wurde vorzeitig verlassen
  m_Statusfield.SetText( „Motor wurde gestartet…“ );
  return NoErr; // Fehlerstatus zurück an aufrufendes Fenster
}

EStatus TMotor::DoStartMove( double speed, bool *aValid ) {
  (*aValid)= false; // Initialisierung des Ausgabeparameters
  if ( !m_Motor->IsMoving() ) { // Motor ist nicht in Bewegung
    if ( !m_Motor->SetSpeed(speed) ) return SpeedErr; // unzulässige Geschwindigkeit
  } else return MoveErr; // ist in Bewegung
  m_Motor->StartMove(); // Bewegung starten
  (*aValid)= true; // Methode wurde bis zum Ende ausgeführt
  return NoErr;
} // ausgelagert aus TMotorSteeringDlg::StartMove()
```

ERGEBNIS

Abb. III.7 Konstruiertes Beispiel für die Auslagerung eines F-Blocks in eine `Do`-Methode (Typ3)

Die eigentliche Dekomposition ist mit diesem Schritt erfolgt! Alle Programmfunktionen sind damit von den oberflächenspezifischen Programmcodeelementen getrennt. Jede weitere Arbeit am Programmcode ist optional, kann aber den Austausch der Nutzeroberfläche erleichtern. Es entstehen die aus **II.2.5** bekannten `Can`-Methoden, zum einfachen Auslesen des Programmzustandes und zur Anpassung der Nutzeroberfläche an diesen.

✓ Abschließendes Kompilieren, Linken und der Regressionstest sichern, dass korrekt gearbeitet wurde.

8) Erzeugung von `CanDo`- Methoden (optional)

Alle Anweisungen, vom Anfang eines `Do`-Methodenrumpfes bis vor die erste **zustandsverändernde Anweisung** (d.h. Aufruf von zustandsverändernden Methoden, Änderung von Attributen, Parametern sowie lokalen oder globalen Variablen) sind potentielle Anweisungsfolgen für eine `CanDo`-Methode. Die Initialisierung von lokalen Variablen und **Ausgabeparametern** ist nicht als zustandsverändernde Anweisung zu betrachten! Ausgabeparameter sind hierbei Referenzparameter, deren übergebener Wert in jedem Fall überschrieben wird und die nur der Rückgabe von Werten dienen (z.B. siehe **Abb. III.7**, `aValid`-Parameter).

Jede Anweisung aus dem genannten Bereich von `Do`-Methoden, die als Bedingung zur Ausführung aller zustandsverändernden Anweisungen fungieren, können in eine neue `CanDo`-Methode verschoben werden. Dies ist insbesondere der Fall, wenn eine Bedingung zum frühzeitigen Verlassen der `Do`-Methode führt. Alle Parameter der `Do`-Methode, die in den

ausgelagerten Anweisungen verwendet werden, müssen an die CanDo-M. weitergereicht werden. Deklaration und Initialisierung von lokalen Variablen der Do-Methode, die in der CanDo-M. benutzt werden, sind in diese zu kopieren. Attribute der Klasse werden nicht geändert, da nur nicht-zustandsverändernde Anweisungen ausgelagert wurden! Wenn die Programmiersprache dies erlaubt, sind alle CanDo-Methoden als nicht-zustandsverändernd zu kennzeichnen. Die CanDo-Methode darf nur true zurückgeben, wenn alle Bedingungen die Ausführung der zustandsverändernden Anweisungen (in der Do.M.) erlauben.

Am Anfang der Do-Methode ist die CanDo-Methode aufzurufen und der Rückgabewert auszuwerten. Danach erfolgt die bisherige Deklaration und Initialisierung der lokalen Variablen und Ausgabeparameter, von denen u.U. einige entfernt werden können (von den meisten Compilern als Warnung angezeigt).

```
EStatus TMotor::DoStartMove( double speed, bool *aValid ) {
  (*aValid)= false; // Initialisierung des Ausgabeparameters
  if ( !m_Motor->IsMoving() ) { // Motor ist nicht in Bewegung
    if ( !m_Motor->SetSpeed(speed) ) return SpeedErr; // unzulässige Geschwindigkeit
  } else return MoveErr; // ist in Bewegung
  m_Motor->StartMove(); // Bewegung starten = Zustandsveränderung!!!
  (*aValid)= true;
  return NoErr;
}
```

AUSGANGSZUSTAND

```
EStatus TMotor::DoStartMove( double speed, bool *aValid ) {
  (*aValid)= false; // Initialisierung des Ausgabeparameters
  EStatus result;
  if ( !m_Motor->CanDoStartMove(speed, &result) ) return result;
  m_Motor->StartMove(); // Bewegung starten = Zustandsveränderung!!!
  (*aValid)= true;
  return NoErr;
}

// Bedingungen für den Start einer Bewegung
bool TMotor::CanDoStartMove( double speed, EStatus *result ) {
  if ( !m_Motor->IsMoving() ) { // Motor ist nicht in Bewegung
    if ( !m_Motor->SetSpeed(speed) ) { // unzulässige Geschwindigkeit
      (*result)= SpeedErr;
      return false; // zustandsverändernde Anweisung nicht ausführen!!!
    }
  } else { // ist in Bewegung
    (*result)= MoveErr;
    return false; // zustandsverändernde Anweisung nicht ausführen!!!
  }
  return true;
}
```

ERGEBNIS

Abb. III.8 Konstruiertes Beispiel für die Auslagerung einer CanDo-Methode

✓ Das Projekt ist zu kompilieren, zu linken. Ein Regressionstest der benutzenden Subsysteme sollte durchgeführt werden.

9) Erstellung von CanSet- und Gewinnung zusätzlicher Bedingungen für CanDo-Methoden (optional)

Es gibt Fälle, wo Do- und Set-Methoden nur unter immer gleichen Bedingungen aufgerufen werden. Alle nutzeroberflächen-unabhängigen Teile können dann zusätzlich in die entsprechende Can-Methode aufgenommen werden[23], wenn sie nicht-zustandsverändernde Anweisungen darstellen und bei JEDEM Aufruf der Do- bzw. Set-Methode auf die gleiche Weise überprüft werden. Wenn noch keine Can-Methode vorhanden ist, sollte eine erstellt werden. Die Do- bzw. Set-Methode bleiben unverändert (siehe **Abb. III.8**, **Zwischenstand**).

[23] Sie muss false zurückgeben, wenn die Do- bzw. Set-M. nicht ausgeführt werden darf.

Nur wenn die Can-Methode vor der Auslagerung der Bedingung(en) noch nicht existierte, dürfen diese durch Aufruf und Auswertung der Can-M. ersetzt werden (siehe **Abb. III.8**, **Ergebnis**). Wenn bereits eine Can-M. vorhanden war, darf nicht ersetzt werden! Grund sind die bereits in der alten Can-M. aufgeführten Bedingungen. Diese führen zu zusätzlichen Einschränkungen, die u.U. zu Fehlern führen könnten.

```
// Fenster für Druckerwartung ************************************************************************************
void TPrinterServiceDlg::OnPrintTestPage() {  // Versuch Testseite zu drucken
  if ( m_Printer->InService() || m_Printer->IsPrinting() ) return;
  if ( (m_Printer->IsCartridgeEmpty()) ||
       (!m_Printer->HasPaper()) ) return;
  m_Statusfield.SetText( „Testseite wird gedruckt…“ );
  DoPrint( m_Printer->GetTestPage() );
}

// Klasse für Druckeransteuerung ************************************************************************************
void TPrinter::DoSpoolerJob() {  // Versuch nächsten Druckauftrag zu verarbeiten
  if ( InService() || IsPrinting() ) return;
  if ( IsCartridgeEmpty() || !HasPaper() ) return;
  DoPrint( Spooler.GetJob() );
}

void TPrinter::DoPrint( … ) {
  … // aktuellen Auftrag abarbeiten
}
```
AUSGANGSZUSTAND

```
// Fenster für Druckerwartung ************************************************************************************
void TPrinterServiceDlg::OnPrintTestPage() {…}  // seit Ausgangszustand unverändert

// Klasse für Druckeransteuerung ************************************************************************************
void TPrinter::DoSpoolerJob() {…}  // seit Ausgangszustand unverändert

bool TPrinter::CanDoPrint() {  // NEU
  if ( InService() || IsPrinting() ) return false;
  if ( IsCartridgeEmpty() || !HasPaper() ) return false;
  return true;
}

void TPrinter::DoPrint( … ) {
  if ( !CanDoPrint() ) return;  // NEU, die immergleichen Bedingungen besser hier prüfen
  … // aktuellen Auftrag abarbeiten
}
```
ZWISCHENSTAND

```
// Fenster für Druckerwartung ************************************************************************************
void TPrinterServiceDlg::OnPrintTestPage() {  // Versuch Testseite zu drucken
  if ( !m_Printer->CanDoPrint() ) return;
  m_Statusfield.SetText( „Testseite wird gedruckt…“ );
  DoPrint(m_Printer->GetTestPage() );
}

// Klasse für Druckeransteuerung ************************************************************************************
void TPrinter::DoSpoolerJob() {  // Versuch nächsten Druckauftrag zu verarbeiten
  DoPrint( Spooler.GetJob() );  // CanDoPrint hier nicht erforderlich, weil Teil von DoPrint
}

bool TPrinter::CanDoPrint() {…}  // seit Zwischenstand unverändert

void TPrinter::DoPrint( … ) {…}  // seit Zwischenstand unverändert
```
ERGEBNIS

Abb. III.9 Konstruiertes Beispiel für die Auslagerung einer CanDo-Methode

✓ Diese Vorgehensweise steigert die Robustheit bei der Neuimplementierung einer Nutzeroberfläche. Am Ende ist wieder zu kompilieren, zu linken und ein Regressionstest der benutzenden Subsysteme ist durchzuführen.

10) Entfernen gleicher `Do`-/ `CanDo`- Methoden

Abschließend sind jeweils die `Do`-M. und die `CanDo`-Methoden untereinander zu vergleichen. Bei zwei identischen Methodenrümpfen kann eine Methode entfernt werden. Bei `Do`-M. müssen auch die gerufenen `CanDo`-Methoden identisch sein, um ein `Do`-/`CanDo`-Methodenpaar entfernen zu können. Jede Benutung der entfernten Methode ist durch den Aufruf der verbliebenen, identischen Methode zu ersetzen.

Wenn eine `Do`-Methode nur aus der Auswertung einer `CanDo`-Methode und dem abschließenden `return true` besteht, sind `Do`- und `CanDo`-Methode identisch. In diesem Fall ist die `Do`-Methode zu entfernen und ihre Aufrufe sind durch die der `CanDo`-Methode zu ersetzen.

✓ Abschließend muss das Projekt kompiliert und gelinkt werden, und ein Regressionstest muss sich anschließen.

11) Kennzeichnung nicht-zustandsverändernder Methoden und konstanter Parameter (optional)

Um ein robustes Design wie beim Forward Engineering zu entwickeln, sollten nicht-zustandsverändernde Methoden entsprechend gekennzeichnet werden (in C++ const-member-functions; siehe **[16]**). Gleiches gilt für unveränderliche Parameter (in C++ durch den type specifier `const` siehe **[16]**). Dies ist jedoch nicht in allen Programmiersprachen möglich.

✓ Zur Verifikation muss das Projekt kompiliert und gelinkt werden. Ein Regressionstest ist nicht erforderlich.

12) Suche und Korrektur von Fehlern (optional)

An das Ende dieser Schrittfolge sollten sich Wartungsvorgänge anschließen. Weil durch die Dekomposition ein gut strukturierter Programmcode entstanden ist und sich der Entwickler intensiv mit diesem beschäftigt hat, sollten Fehlersuche und -korrektur zu diesem Zeitpunkt sehr effektiv sein. Dazu sind die in der Verhaltensspezifikation oder in separaten Fehlerdokumenten aufgeführten Änderungswünsche und die im Programmcode markierten Fehler zu beheben. Zudem wurden während der Dekomposition wahrscheinlich zusätzliche Fehler gefunden, die nicht sofort behoben werden sollten. Vielmehr sollten sie im Programmcode gekennzeichnet und beschrieben werden, um in diesem Schritt nach einer nochmaligen Prüfung korrigiert zu werden.

✓ Zum Abschluss dieser Schrittfolge müssen die nach dem Kompilieren und Linken angezeigten Hinweise und Warnungen korrigiert werden. Ein Regressionstest sollte sich anschließen.

! Die wichtigste Grundregel bei dieser Schrittfolge ist, so wenig wie möglich am Programmcode zu ändern! Das verhindert versehentliche Änderungen am eigentlichen Programmverhalten.

13) Abschluss durch Fein- und Abnahmetest

Nach Abschluss aller Implementierungsarbeiten sollte ein Black-Box-Testverfahren der gesamten über die Oberflächenfenster steuerbaren Funktionalität des dekomponierten Subsystems durchgeführt werden. Damit wird sichergestellt, dass das Programmverhalten auch nach der Dekomposition weiterhin der Spezifikation entspricht.

14) Dokumentation des entstandenen Designs

Erst während der Schrittfolge kristallisieren sich Attribute und Methoden der Funktionskomponente und Nutzeroberfläche heraus. Mit Abschluss dieser Arbeiten kann durch Reverse Engineering aus dem Programmcode ein Designdokument erstellt werden. Dort hält der dekomponierende Entwickler die endgültige Schnittstelle, mit eigenen Anmerkungen versehen, fest. Diese Dokumentation ist für den späteren Austausch der Nutzeroberfläche sehr wichtig.

Anwendungsfall: ‚Manuelle Justage'

Um die Dekomposition auch im Reengineering durchzuführen und gleichzeitig einen Vergleich zu einem Neuentwurf herzustellen, wurde die ursprünglich vorhandene Implementierung der ‚Manuellen Justage' ebenfalls in Nutzeroberfläche und Funktionskomponente getrennt. Die oben vorgestellte Schrittfolge diente dabei als Grundlage, weshalb hier nur auf Eigenheiten bei der getrennten ‚Manuellen Justage' hingewiesen wird. Ergänzt werden diese Ausführungen im Anschluss durch den Anwendungsfall ‚Topographie', wo ebenfalls die Dekomposition im Reengineering vollzogen wurde.

Zur Gegenüberstellung von Observer-Muster und Polling-Variante wurden für die getrennte ‚Manuelle Justage' wieder beide Kommunikationsmuster implementiert. Durch die minimalen Unterschiede zwischen diesen Mustern sind die Beispiele wieder der Polling-Variante entnommen. Auf Besonderheiten des Observer-Musters wird in **III.2.2–1** eingegangen.

1) Erstellung neuer Dateien

Wie beim Forward Engineering wurden auch hier zwei Dateipärchen, für Funktionskomponente und Oberfläche, erstellt. Für die Deklaration entstanden MJ_OFUNK.H und MJ_OGUI.H und für die Implementierung MJ_OFUNK.CPP und MJ_OGUI.CPP. Die ursprüngliche Oberflächenklasse `TAngleControl` wurde in die MJ_-Dateifamilie aufgenommen, indem sie in die Dateien MJ_OLD.H und MJ_OLD.CPP verschoben wurde. Zuvor war diese Klasse mit Oberflächenfenstern anderer Subsysteme zusammen in der Datei SWINTRAC.H bzw. M_DLG.CPP enthalten. Für die zu dekomponierende Nutzeroberfläche wurde `TAngleCtlDlg` (anstatt `TAngleControl`) gewählt. Beide verwenden dieselbe Dialogressource (**Abb. II.1**), weil das äußere Erscheinungsbild und Bedienverhalten – die Spezifikation im Allgemeinen – unverändert bleiben sollte.

2) Behandlung von Ansätzen vorhandener Funktionskomponenten

Weil im Falle der ursprünglichen ‚Manuelle Justage' keine Funktionskomponente existierte, wurde eine leere Klasse mit dem Bezeichner `TAngleCtl` erstellt. In der Nutzeroberfläche musste MJ_FUNK.H inkludiert werden, um im Konstruktor ein Objekt von `TAngleCtl` erstellen zu können. Neben dem neuen Attribut `m_lnkAngleCtl` musste auch der Destruktor geändert werden, um das dynamisch erzeugte Objekt wieder freizugeben.

3) Solldesign festlegen

Durch die fehlende Funktionskomponente im ursprünglichen Subsystem war hier kein Kommunikationsmuster vorgeschrieben. Aufgrund der Richtlinien aus **II.3.3–5** wurde festgestellt, dass hier sowohl Observer-Muster als auch Polling-Variante anwendbar sind. Um diese auch im Reengineering vergleichen zu können, wurden beide Muster parallel implementiert.

Die Implementierung des Singleton-Musters wäre hier nicht sinnvoll gewesen, weil ein Teil der Attribute von `TAngleControl` bei jedem Aufruf des Dialogfensters neu mit den aktuellen Werten aus den benutzten Subsystemen initialisiert werden muss. Beim Singleton hätte eine

separate Initialisierungsmethode für diese Attribute erstellt werden müssen, die dann von der Nutzeroberfläche gerufen werden müsste. Diese Vorgehensweise ist zudem nicht erforderlich, weil nur `TAngleCtlDlg` auf die Funktionskomponente zugreift.

Die im ursprünglichen Subsystem eingesetzten Windows API-Methoden, die zur Verwaltung der Timer-Komponente und zur Bildschirmaktualisierung benutzt wurden, konnten durch die Klasse `TInterfaceTimer` ersetzt werden. Dadurch wurde die Verwaltung wesentlich komfortabler, portabler und fehlerresistenter (siehe auch Anwendungsfall ‚Topographie' – **3)**). In der Polling-Variante muss sich die Oberfläche um die Aktualisierung der Nutzeroberfläche kümmern, weil hier die Beziehung zu der Timer-Komponente besteht (siehe Klassendiagramm in **Tab. III.3**). Im Design des Observer-Musters liegt die Initiative bei der Funktionskomponente, welche die Oberfläche über Zustandsänderungen informiert. Deshalb bedient sich hier die Funktionskomponente der Timer-Komponente (vgl. **Abb. III.22**).

4) Programmcodelayout verbessern (optional)

Zur Verbesserung der Lesbarkeit wurde der Programmcode formatiert, kommentiert und potentielle Fehlerquellen markiert. Letzteres betraf fast alle Attribute. Ein Teil von ihnen wurde nicht im Konstruktor initialisiert, ein anderer zwar geschrieben, aber nur einmal und nicht als `const` gekennzeichnet. Anstatt den aktuellen Systemzustand aus der Nutzeroberfläche oder beim benutzten Subsystem direkt auszulesen, wurden zwischengespeicherten Werte aus Attributen verwendet. Damit war die Nutzeroberfläche selbst für die Aktualisierung dieser Werte zuständig, was der Entwickler z.B. bei `bMoveActive` und der Auswahl eines anderen Antriebs vergessen hatte!

Insgesamt sind dabei neun für den Anwender offensichtliche Fehler aufgefallen, die vorerst nur in **[M19]** dokumentiert werden konnten. Zu bemängeln war auch, dass der aktuelle Systemzustand (über Attribute) manipuliert wurde, um in einer anschließend gerufenen Methode ein bestimmtes Verhalten auszulösen. Zur Korrektur erfolgte die Übergabe von Parametern an die jeweilige Methode.

Zudem sind zahlreiche nutzerdefinierte Botschaften an das Fenster gesendet worden, um eine Methode auszuführen. Die Lesbarkeit des Programmcodes war erheblich erschwert, weil Botschaften- und Methodenbezeichner in keinem Zusammenhang standen. Problematisch war, dass die Botschaften mit `PostMessage` versandt wurden, was eine Verarbeitung im Hintergrund erlaubt (siehe **II.4.2**, Geschwindigkeitsoptimierungen). Deshalb konnten die `PostMessage`-Aufrufe nicht einfach durch den Aufruf der entsprechenden Methode ersetzt werden. Bei der getrennten ‚Manuellen Justage' waren die Unterschiede zwischen den Ausführungszeiten aber so minimal, dass sie nicht "spürbar" waren. Deshalb konnten die Methoden dort doch direkt aufgerufen werden, was Verständlichkeit, Übersichtlichkeit und Programmcodeumfang deutlich verbesserte.

Die Kernfunktionalität lag zu diesem Zeitpunkt bei der Nutzeroberfläche in Methode `Dlg_OnCommand`, wo auf 327 LOC (mehr als 70% des gesamten Systems) jede Steuerelementbotschaft behandelt wurde. Im Zuge der Layoutverbesserung erfolgte die Auslagerung des Programmcodes für jedes Steuerelement in getrennte Methoden (Zuordnung siehe **Abb. III.10**). Ein 31 zeiliges Makro war toter Code. Da es seit der Ausgangsversion von 1998 ungenutzt blieb, konnte es entfernt werden.

5) Neuordnung der Attribute

Diejenigen Attribute, die keine oberflächenspezifischen Eigenschaften widerspiegelten, wurden in die Funktionskomponente verschoben und mit dem Präfix `m_` gekennzeichnet (siehe **Abb. III.11**,). Trotz des hohen Zugriffsschutzes von Seiten der Funktionskomponente mussten nur Accessor- und Mutator-Methoden für das dynamische Feld

m_Speed implementiert werden. Die übrigen Attribute werden nur intern von den Methoden der Funktionskomponente benutzt. Die Initialisierung wurde, soweit vorhanden, aus der Oberflächenklasse entnommen bzw. mit Standardwerten belegt. Von ursprünglich 11 Attributen in TAngleCtlDlg verbliebenen 4 Attribute. Sie wurden mit m_ gekennzeichnet und mit aussagekräftigen Bezeichnern versehen (siehe **Abb. III.10**,).

Zusätzlich werden noch die beiden Referenzen auf die Funktions- und Timerkomponente (letztere wegen der Polling-Variante) benötigt. Die bei **4)** erwähnten Attribute, die den Zustand anderer Subsysteme zwischenspeichern, wurden entfernt. Stattdessen wurden in der Funktionskomponente neue Methoden zum Auslesen des aktuellen Zustands implementiert. Auch Eigenschaften der Nutzeroberfläche werden nun direkt ausgelesen, was bereits zwei Fehler eliminierte (siehe **[M19]**).

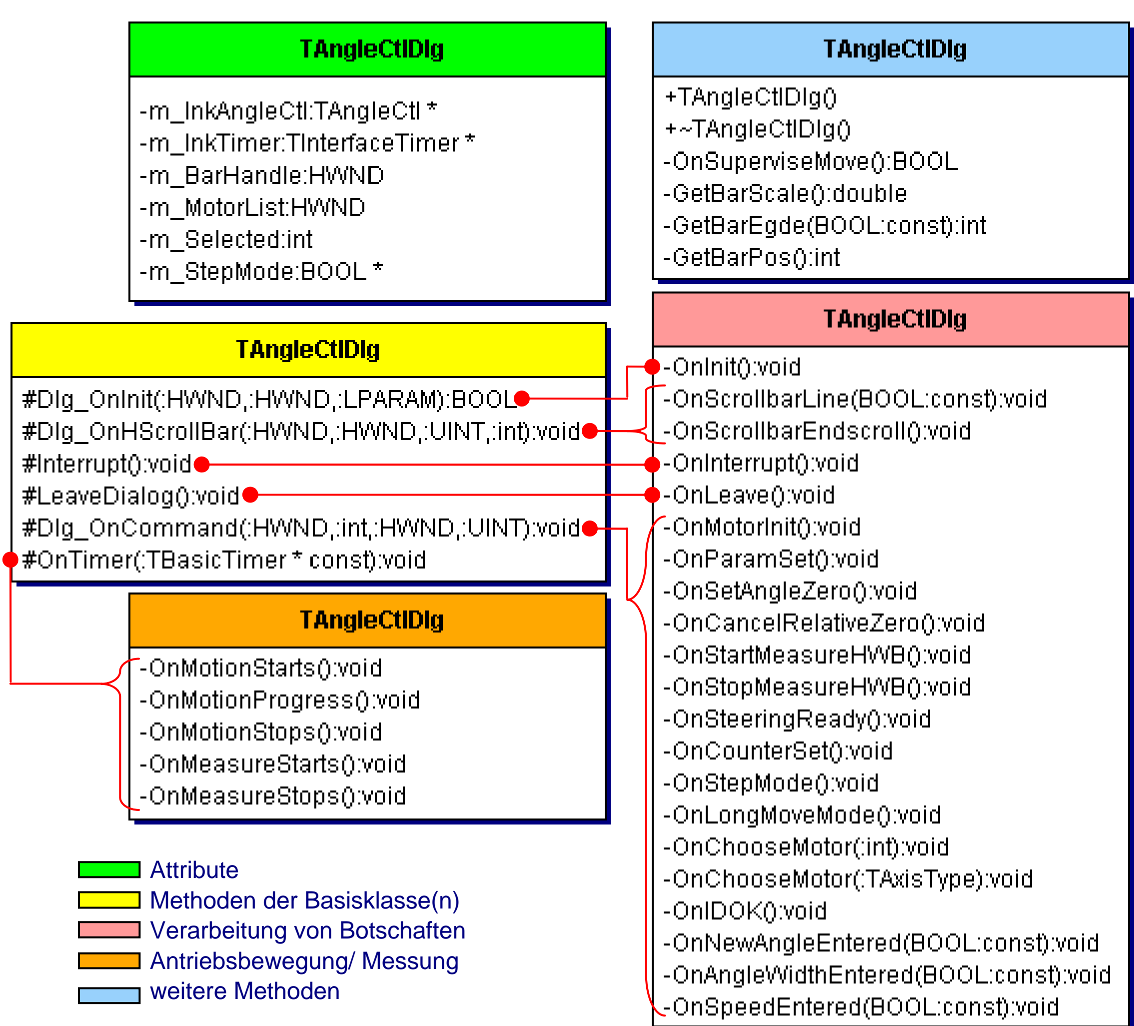


Abb. III.10 UML-Klassendiagramm von TAngleCtlDlg (Nutzeroberfläche) nach der Dekomposition (mit Zuordnung der ausgelagerten Methoden)

6) Zusammenfassen von Funktionalitätsaufrufen

Die Identifikation von F- und O-Blöcken gestaltete sich in den neuen On-Methoden von TAngleCtlDlg problemlos. Etwa die Hälfte aller Anweisungen konnte anhand des Handle-Parameters leicht für das Auslesen bzw. die Aktualisierung der Nutzeroberfläche identifiziert werden. Diese Methoden wurden fast komplett durch die entsprechende Variante der Fensterbasisklasse ersetzt, um einen höheren Bedienkomfort zu garantieren. Damit wurde bspw. das Problem mit der Konvertierung von Zahlenwerten gelöst (Wunschkriterium in **[M19]**). Zudem wurde dadurch der Programmcodeumfang reduziert. Bei den zuvor benutzen

API-Funktionen mussten die Zahlen zuerst in einer Zeichenkette formatiert und dann in die Steuerelemente eingetragen werden.

Die übrigen Methoden beschränken sich entweder auf den Zugriff anderer Subsysteme (F-Blöcke), die Ausgabe von Meldungsfenstern oder die Anpassung des Mauszeigers (O-Blöcke). Auf die zeitaufwendige Analyse der F- und O-Blöcke konnte verzichtet werden, weil pro `On`-Methode selten mehr als ein F-Block existierte.

7) Erzeugen von `Do`- Methoden

Neben 9 `Do`-Methoden (alle vom Typ 2 der oben vorgestellten Klassifikation, siehe **III.2.1**, **7)**) dienen 4 `Set`-, 1 `Reset`- und eine übrige Methode zur Steuerung der Subsysteme ‚Motorsteuerung' und ‚Detektornutzung' sowie ‚Ablaufsteuerung' oder zu deren Parametrisierung (siehe **Abb. III.11**,).

TAngleCtl

-m_Sensor:TDetector *
-m_MotorCount:UINT
-m_MoveSpeed:float *
-m_IsMeasuring:BOOL

TAngleCtl

+UpdateDetector():void
+SetSpeed(:const int,:double &):BOOL
+SetAngleWidth(:const int,:double &):BOOL
+SetCorrectionState(:const int):BOOL
+SetRelativeZero(:const int):BOOL
+ResetRelativeZero(:const int):BOOL
+DoInit(:const int):BOOL
+DoInitMotor(:const int,BOOL:const):BOOL
+DoDirect(:const int,:const double):BOOL
+DoStep(:const int,BOOL:const):BOOL
+DoDrive(:const int,BOOL:const):BOOL
+DoStop(:const int):BOOL
+DoStopEverything(:const int):void
+DoStartMeasure(HWND:const):BOOL
+DoStopMeasure(:const int):BOOL

TAngleCtl

+TAngleCtl()
+~TAngleCtl()
-DoSelectMotor(:const int):BOOL
-Moves(:const int,:BOOL &):BOOL

TAngleCtl

+GetAngle(:const int,:BOOL &):double
+GetAngleMin(:const int,:BOOL &):double
+GetAngleMax(:const int,:BOOL &):double
+GetSpeed(:const int,:BOOL &):double
+GetAngleWidth(:const int,:BOOL &):double
+GetDigits(:const int,:BOOL &):UINT
+GetUnit(:const int,:BOOL &):LPCSTR
+GetMotorIdx(TAxisType:const):int
+GetMotorName(:const int,:BOOL &):LPCSTR
+GetMotorCount():int
+GetActualMotor():int
+IsIndexValid(:const int):BOOL
+HasOffset(:const int,:BOOL &):BOOL
+IsMoving(:const int,:BOOL &):BOOL
+IsCalibrated(:const int,:BOOL &):BOOL
+IsMeasuring():BOOL
+IsMeasureReset():BOOL
+GetMeasureHwb():double
+MeasureStopped(:const int):BOOL

TAngleCtl

+CanSetSpeed(:const int):BOOL
+CanSetAngleWidth(:const int):BOOL
+CanSetOffset(:const int):BOOL
+CanResetRelativeZero(:const int):BOOL
+CanDoDirect(:const int):BOOL
+CanDoStep(:const int,BOOL:const):BOOL
+CanDoDrive(:const int,BOOL:const):BOOL
+CanDoStop(:const int):BOOL
+CanDoStartMeasure():BOOL
+CanDoStopMeasure():BOOL

Attribute
Auslesen von Zuständen der benutzten Subsysteme
Zugriff auf Attribute und Steuerung der benutzten Subsysteme
`Can`-M. für zuvor genannte Methoden
weitere Methoden

Abb. III.11 UML-Klassendiagramm von **`TAngleCtl`** (Funktionskomponente) nach der Dekomposition

Zum Auslesen der benutzten Subsysteme wurden 12 `Get`-, 5 `Is`-, 1 `Has`- und eine übrige Methode erstellt (siehe **Abb. III.11**,), die deren Zustand nicht ändern.

8) Erzeugung von CanDo- Methoden (optional)

Da die Ausführung der o.g. zustandsverändernden Methoden z.T. von Bedingungen abhängig ist, wurden dafür Can-Methoden (siehe **Abb. III.11**,) abgeleitet. Davon werden jedoch nur CanDoStartMeasure und CanResetRelativeZero direkt in der Nutzeroberfläche verwendet. Weil sich die existierende Nutzeroberfläche sonst nicht an Zustandsänderungen anpasst, ist die Verwendung der CanDo- in den Do- bzw. der CanSet- in den Set-Methoden der Funktionskomponente besonders wichtig. Ansonsten könnte die Nutzeroberfläche Methoden aufrufen, ohne die Aufrufbedingungen ausgewertet zu haben. Im Gegensatz zur ‚Topographie' wurden die Can-Methoden hier separat implementiert, um einen repräsentativen Vergleich zwischen Forward und Reengineering führen zu können. Zudem soll das damit verbundene Potential deutlich gemacht werden, dass es der Nutzeroberfläche nach der Dekomposition gestattet, sich an Zustandsänderungen anzupassen.

```
BOOL  TAngleCtl::DoDirect(int aIndex, double aPos) {
  BOOL bValid;
  if ( (IsMoving(aIndex, bValid)) || // Bedingungen für den Direktbetrieb
       (!bValid) || (m_IsMeasuring) ) return FALSE;

  DoSelectMotor(aIndex);
  mActivateDrive(); // Motor aktivieren (sonst bewegt sich der Antrieb nicht, wenn er gestartet wird)
  mMoveToDistance(aPos);
  BOOL bValid;
  return IsMoving(aIndex, bValid); // wurde die Bewegung gestartet?
}
```
AUSGANGSZUSTAND

```
BOOL  TAngleCtl::DoDirect(int aIndex, double aPos) {
  if ( !CanDoDirect(aIndex) ) return FALSE;
  DoSelectMotor(aIndex);
  mActivateDrive(); // Motor aktivieren (sonst bewegt sich der Antrieb nicht, wenn er gestartet wird)
  mMoveToDistance(aPos);
  BOOL bValid;
  return IsMoving(aIndex, bValid); // wurde die Bewegung gestartet?
}
// Bedingungen für den Direktbetrieb
BOOL  TAngleCtl::CanDoDirect(int aIndex) {
  BOOL bValid;
  if ( (IsMoving(aIndex, bValid)) ||
       (!bValid) || (m_IsMeasuring) ) return FALSE;
  return TRUE;
}
```
ERGEBNIS

Abb. III.12 Beispiel zur Auslagerung einer CanDo-Methode

9) Erstellung von CanSet- und Gewinnung zusätzlicher Bedingungen für CanDo-Methoden (optional)

Abb. III.13 zeigt ein Beispiel, wo durch die Analyse der Aufrufbedingungen für Do- und Set-Methoden innerhalb der Nutzeroberfläche zusätzliche Bedingungen bzw. neue Can-Methoden gewonnen werden konnten. Damit wurde die Robustheit der Funktionskomponente weiter gesteigert, weil die Ausführung von zustandsverändernden Methoden an weitere Bedingungen geknüpft wurde. Zudem verbessert sich auch die Wiederverwendbarkeit der Funktionskomponente, weil die in diesem Schritt gewonnen zusätzlichen Bedingungen dort nicht mehr nach dem Austausch der Nutzeroberflächen beachtet werden müssen.

Um den Umfang des Programmcodes zu reduzieren, wurde hier auf den Aufruf der CanDo-Methode in der Nutzeroberfläche verzichtet. Weil die Do-Methode denselben Rückgabewert wie die Can-M. besitzt und die Do-M. bei Nichterfüllung sofort verlassen wird, war der direkte Aufruf möglich.

```
void TAngleCtlDlg::Dlg_OnCommand(HWND, int aId, HWND, UINT) {
  switch (aId) {
    case cm_MeasureHWB:  // Halbwertsbreite messen/ abbrechen
      if ( m_lnkAngleCtl->IsMeasuring() ) {  // Messung ist aktiv: abbrechen
        m_lnkAngleCtl->DoStopMeasure();
        … // Aktualisieren der Nutzeroberfläche
        break;
      }
      // Messung beginnen; Makro "Inquire Hwb" ausführen
      if ( m_lnkAngleCtl->IsMoving() ) break;  // Antrieb ist in Bewegung: Abbruch
      m_lnkAngleCtl->DoStartMeasure(GetHandle());
      CtrlSetText(cm_MeasureHWB, "&Messung abbrechen");
      break;
    … // andere Botschaften behandeln
}

void TAngleCtl::DoStartMeasure(HWND aWnd) {
  … // Delegieren durch Aufruf von Methoden des Subsystems ‚Ablaufsteuerung'
}

void TAngleCtl::DoStopMeasure() {
  Steering.Reset();
}
```

AUSGANGSZUSTAND

```
void TAngleCtlDlg::Dlg_OnCommand(HWND, int aId, HWND, UINT) {
  switch (aId) {
    case cm_MeasureHWB:  // Halbwertsbreite messen/ abbrechen
      if ( m_lnkAngleCtl->DoStopMeasure() ) {  // Messung ist aktiv: abbrechen
        … // Aktualisieren der Nutzeroberfläche
      } else if ( m_lnkAngleCtl->DoStartMeasure() ) {  // Messung hat begonnen
        CtrlSetText(cm_MeasureHWB, "&Messung abbrechen");
      }
      break;
    … // andere Botschaften behandeln
}

BOOL TAngleCtl::CanStartMeasure() {
  return ( (!m_lnkAngleCtl->IsMeasuring()) && (!IsMoving()) );
} // gewonnen aus der Nutzeroberfläche

BOOL TAngleCtl::DoStartMeasure(HWND aWnd) {
  if ( !CanStartMeasure() ) return FALSE;
  … // Delegieren durch Aufruf von Methoden des Subsystems ‚Ablaufsteuerung'
  return TRUE;
}

BOOL TAngleCtl::CanStopMeasure() {
  return ( m_lnkAngleCtl->IsMeasuring() );
} // gewonnen aus der Nutzeroberfläche
BOOL TAngleCtl::DoStopMeasure() {
  if ( !CanStopMeasure() ) return FALSE;
  Steering.Reset();
  return TRUE;
}
```

ERGEBNIS

Abb. III.13 Beispiel zur Gewinnung zusätzlicher Bedingungen für `CanDo`-Methoden

10) Entfernen gleicher `Do`-/ `CanDo`- Methode

Durch den relativ geringen Funktionsumfang der getrennten ‚Manuellen Justage' entstanden hier keine Methoden doppelt. Bei der Analyse der entstandenen Funktionskomponente sind jedoch wiederkehrende Anweisungsfolgen aufgefallen, die in zwei private Hilfsmethoden ausgelagert werden konnten. Dies sind `MeasureStopped` zum Testen, ob die Halbwertsbreitenmessung beendet ist und `Moves`, um zu ermitteln, ob sich der Antrieb bewegt.

11) Kennzeichnung nicht-zustandsverändernder Methoden und konstanter Parameter (optional)

Analog zum Forward Engineering wurden nicht-zustandsverändernde Methoden (und aus **Abb. III.11**) als const-member-function gekennzeichnet. Bei sämtlichen Methoden konnten – bis auf Referenzparameter – alle Parameter als `const` deklariert werden; z.B. `double GetAngleWidth(const int, BOOL&) const`.

12) Suche und Korrektur von Fehlern (optional)

Während dieser Schrittfolge wurde bereits an mehreren Stellen darauf hingewiesen, wo Fehler vermutet und offensichtliche Fehler bereits korrigiert wurden. Ein Teil der im Fehlerdokument aufgeführten Fehler wurde so schon während der Schrittfolge behoben. Dies betraf auch den schlechten Bedienkomfort bei Zahleneingabefeldern vor der Dekomposition, der durch die Verwendung der neuen Fensterbasisklasse automatisch behoben wurde. Die unter **4)** benannten Fehler wurden im Zuge von **5)** behoben, indem zwischengespeicherte Werte z.B. durch die Methode `IsMoving` ersetzt wurden. Drei weitere Fehler wurden durch Verwenden der `Can`-Methoden innerhalb der Funktionskomponente automatisch korrigiert, weil die unzulässigen Operationen nicht mehr durchgeführt werden können.

Die übrigen fünf Fehler mussten durch klassische Wartungsarbeiten am Programmcode korrigiert werden.

13) Abschluss durch Fein- und Abnahmetest

Zunächst wurde hier ein Regressionstest durchgeführt, um zu prüfen, ob das Gesamtverhalten durch die Dekomposition nicht beeinflusst wurde. Analog zum Forward Engineering wurde anschließend ein Feintest – anhand des CTE-Diagrammes **[M19]** – durchgeführt, um auch das genaue Verhalten zu testen und sicherzustellen, dass durch die Dekomposition keine Fehler entstanden sind.

Bis auf die Fehlerkorrektur ist das Verhalten dieses Subsystems nach der Dekomposition völlig gleich geblieben. Interessant ist dabei die Tatsache, dass die Mitarbeiter des Physikalischen Institutes den Austausch der ursprünglichen durch die getrennte ‚Manuelle Justage' überhaupt nicht bemerkten.

14) Dokumentation des entstandenen Designs

Für die getrennte ‚Manuelle Justage' war kein Designdokument erforderlich, weil dieses Subsystem bereits durch die neue ‚Manuelle Justage' repräsentiert wird (siehe **Kapitel II**). Die Erstellung der getrennten ‚Manuellen Justage' hatte nur rein akademischen Charakter. Sie dient dem Vergleich der Dekomposition im Forward und Reengineering.

Zudem sind hier Polling-Variante und Observer-Muster zu Vergleichszwecken parallel entstanden. Zur Auswertung siehe **III.2.2–1**.

TAngleCtl

-m_Sensor:TDetector *
-m_MotorCount:UINT
-m_MoveSpeed:float *
-m_IsMeasuring:BOOL

+TAngleCtl
+~TAngleCtl
+DoInit:BOOL
+DoInitMotor:BOOL
+DoSelectMotor:BOOL
+DoDirect:BOOL
+DoStep:BOOL
+DoDrive:BOOL
+DoStop:BOOL
+DoStopEverything:void
+IsIndexValid:BOOL
+HasOffset:BOOL
+IsMoving:BOOL
+IsCalibrated:BOOL
+GetAngle:double
+GetAngleMin:double
+GetAngleMax:double
+GetSpeed:double
+GetAngleWidth:double
+GetDigits:UINT
+GetUnit:LPCSTR
+GetMotorIdx:int
+GetMotorName:LPCSTR
+GetMotorCount:int
+GetActualMotor:int
+SetSpeed:BOOL
+SetAngleWidth:BOOL
+SetCorrectionState:BOOL
+SetRelativeZero:BOOL
+ResetRelativeZero:BOOL
+DoStartMeasure:BOOL
+DoStopMeasure:BOOL
+IsMeasuring:BOOL
+IsMeasureReset:BOOL
+GetMeasureHwb:double
+MeasureStopped:BOOL
+UpdateDetector:void
+CanDoDirect:BOOL
+CanDoStep:BOOL
+CanDoDrive:BOOL
+CanDoStop:BOOL
+CanSetSpeed:BOOL
+CanSetAngleWidth:BOOL
+CanSetOffset:BOOL
+CanResetRelativeZero:BOOL
+CanDoStartMeasure:BOOL
+CanDoStopMeasure:BOOL
-Moves:BOOL

0..1
m_InkAngleCtl

TModalDlg
ITimer
TAngleCtlDlg

-m_InkAngleCtl:TAngleCtl *
-m_InkTimer:TTimer *
-m_BarHandle:HWND
-m_MotorList:HWND
-m_Selected:int
-m_StepMode:BOOL *

+TAngleCtlDlg
+~TAngleCtlDlg
#Dlg_OnInit:BOOL
#Dlg_OnCommand:void
#Dlg_OnHScrollBar:void
#Interrupt:void
#LeaveDialog:void
#OnTimer:void
-OnNewAngleEntered:void
-OnAngleWidthEntered:void
-OnSpeedEntered:void
-OnInit:void
-OnSuperviseMove:BOOL
-OnMotorInit:void
-OnParamSet:void
-OnSetAngleZero:void
-OnCancelRelativeZero:void
-OnStartMeasureHWB:void
-OnStopMeasureHWB:void
-OnSteeringReady:void
-OnCounterSet:void
-OnStepMode:void
-OnLongMoveMode:void
-OnChooseMotor:void
-OnChooseMotor:void
-OnIDOK:void
-OnScrollbarLine:void
-OnScrollbarEndscroll:void
-OnInterrupt:void
-OnLeave:void
-IsDrivemove:BOOL
-GetBarScale:double
-GetBarEgde:int
-GetBarPos:int
-OnMotionStarts:void
-OnMotionProgress:void
-OnMotionStops:void
-OnMeasureStarts:void
-OnMeasureStops:void

Abb. III.14 UML-Klassendiagramm der getrennten ‚Manuellen Justage' (Polling-Variante)

Anwendungsfall: ‚Topographie'

Die Bearbeitung des Subsystems ‚Topographie' erfolgte wie beim Anwendungsfall ‚Manuelle Justage' nach der allgemeinen Schrittfolge für die Dekomposition. Im Folgenden werden ausgesuchte Schwerpunkte, besondere Schwierigkeiten und Abweichungen zum vorherigen Anwendungsfall erläutert.

1) Erstellung neuer Dateien

Für die getrennte Topographie wurden die Dateipärchen TP_GUI.H, TP_GUI.CPP für die Klassen der Nutzeroberfläche und TP_FUNK.H, TP_FUNK.CPP für die Funktionskomponente erstellt. Weil die Bezeichner der ursprünglichen Klassen die Zugehörigkeit zum Subsystem Topographie gut erkennen ließen, wurden für die getrennte Topographie diese Bezeichner übernommen. Dazu mussten aber die vorhandenen Klassen neu bezeichnet werden.

2) Behandlung von Ansätzen vorhandener Funktionskomponenten

In diesem Subsystem existierte bereits der Ansatz einer Funktionskomponente in Form der Klasse `TTopographyOld`. Diese diente ausschließlich zur Initialisierung und Sicherung von Attributen, die durch beide Dialoge gemeinsam genutzt werden.

Im Konstruktor werden 12 Attribute mit festen Zahlwerten und in der einzigen Methode `Initialize` werden weitere 6 Attribute mit Werten aus einer Konfigurationsdatei initialisiert. Der Zugriff erfolgt für beide Dialoge durch die in `TTopographyOld` deklarierte `friend`-Relationen.

Diese Klasse wurde in die TP_FUNK.H und TP_FUNK.CPP kopiert und bildet dort als `TTopography` die neue Funktionskomponente. Die ursprünglichen Oberflächenklassen wurden in `TTopographyExecuteDlg` bzw. `TTopographySetParamDlg` umbenannt. Die zu dekomponierenden Versionen heißen `TTopographyExecDlg` und `TTopographyAdjustDlg` und befinden sich in TP_GUI.H und TP_GUI.CPP.

TTopographyOld
#bSmallAngleSide:BOOL #nNumberCycle:WORD #fWorkPoint:float #fControlRange:float #fControlStep:float #fMoveStep:float #fMaxAngleEscape:float #lMeasurementTime:long #nMotor:int #Sensor:TDetector * #fFailure:float #fMaxTime:float #dwMaxCounts:DWORD #bMonitorUsed:BOOL #Monitor:TDetector * #bMultibleShot:BOOL #dStartAngle:double #fAngleBetweenShots:float
+TTopographyOld +Initialize:void

Abb. III.15 UML-Klassendiagramm der Funktionskomponente `TTopographyOld`

3) Solldesign festlegen

Durch das Vorhandensein der Funktionskomponente `TTopographyOld` musste in der ursprünglichen ‚Topographie' bereits ein Kommunikationsmuster existieren. Dieses wurde als Polling-Variante identifiziert, weil die Initiative für die Herstellung der vertikalen Konsistenz bei der Nutzeroberfläche liegt. Obwohl beide Dialoge gleichzeitig geöffnet sein können, muss die Nutzeroberfläche während einer Messung nur im Dialogfenster `TTopographyExecDlg` aktualisiert werden. Daher konnte Polling auch bei der getrennten Topographie beibehalten werden. Weil nach der allgemeinen Schrittfolge zudem das vorhandene Kommunikationsmuster beibehalten werden sollte, konnte auf die Implementierung des Observer-Musters verzichtet werden. Zudem wäre der Unterschied nur minimal und hätte zu den gleichen Ergebnisse wie bei der getrennten ‚Manuellen Justage' geführt (siehe **III.2.2–1**).

Die ursprüngliche Nutzeroberfläche verwendete zur Aktualisierung während eines Messvorganges Windows API-Operationen. Da diese jedoch über ein Fenster-Handle kommunizieren, wurde eine robuste und portablere (ohne Fenster-Handle) Timer-Komponenten verwendet. Sie wird durch die Klasse `TInterfaceTimer` verwaltet. Zur Behandlung der Timerereignisse muss die Dialogklasse `TTopographyExecDlg` das Interface `ITimer` einerben und die virtuelle Methode `OnTimer(TBasicTimer* const)` überschreiben. Der Parameter dient zur Identifizierung des zum Ereignis zugehörigen Timers, da für eine Messung zwei Timer gleichzeitig benötigt werden.

Die in der Funktionskomponente vorhandenen `friend`-Relationen wurden entfernt und die Oberflächenklassen erhielten je ein privates Attribut `TTopography *m_lnkTopography` für die Delegation an die Funktionskomponente. Von der ursprünglichen Funktionskomponente `TTopographyOld` dürfte es im gesamten ‚XCtl'-Programm nur maximal eine Instanz geben. Deshalb erfolgte eine globale Instanziierung, nicht beim Aufruf eines Dialogs sondern einmal beim Start des ‚XCtl'-Programmes. Es gab aber keinen Mechanismus, der die Erzeugung weiterer Instanzen verhinderte. Um die einmalige Instanziierung in der getrennten ‚Topographie' sicherzustellen, wurde ein Singleton-Muster für die Klasse `TTopography` integriert.

```
class TTopographyOld {
public:
  TTopographyOld();
  …
};

TTopographyOld  TopographyOld;  // globale Instanz
```

AUSGANGSZUSTAND

```
class TTopography {
private:
  TTopography();   // kein Standard-Konstruktor verfügbar
  TTopography( const TTopography& ) {};   // keinen Copy-Konstruktor
  TTopography& operator=( const TTopography& ) {}; // keinen Zuweisungsoperator

  static  TTopography  *m_Instance;   // ist die EINZIGE Instanz

public:
  static  TTopography  *GetInstance();   // ist die einzige Möglichkeit zur Instanziierung
  …
};

TTopography*  TTopography::m_Instance= 0;   // Initialisierung des Singleton

// Erzeugung einer Singleton-Instanz
TTopography* TTopography::GetInstance() {
  if ( m_Instance==0 ) m_Instance= new TTopography;
  return m_Instance;  // wenn noch keine Instanz existierte, wurde diese zuvor erzeugt
}
```

ERGEBNIS

Abb. III.16 Implementierung des Singleton-Musters für die Klasse **`TTopography`**

Die Nutzeroberflächen, in Form der Dialoge `TTopographyExecDlg` und `TTopographyAdjustDlg`, erben die Basisklasse für modalen Dialogfenster `TModalDlg` ein (siehe Klassendiagramm in **Tab. III.9**) und sind damit in ihrer Grundstruktur festgelegt (siehe **Abb. III.21**).

4) Programmcodelayout verbessern (optional)

Die Strukturierung des Programmcodes erfolgte nach den gleichen Kriterien wie bei der getrennten ‚Manuellen Justage'. Die Dialogklasse `TTopographyExecDlg` besitzt zur

Überwachung einer Messung zwei Timer-Komponenten. Zusätzlich war noch ein weiterer Timer vorhanden, der nie gestartet, nur gestoppt wurde. Dieser wurde als Toter Code eingestuft und auskommentiert. Auch die Klasse `TTopographyAdjustDlg` wies Toten Code auf, wie das Feld `SensorList[][0]`, das zwar deklariert, aber nirgends benutzt wurde.

Die Nachrichtenbehandlung in `TTopographyExecDlg` erfolgte in der Methode `Dlg_OnCommand` innerhalb einer großen `switch`-Fallunterscheidung. Dadurch war fast die gesamte Funktionalität der Klasse dort konzentriert Von insgesamt 422 **LOC** entfallen allein 314 (ca. ¾) auf diese Methode. Jede Behandlung eines `case`-Falls wurde in eine neue private `On`-Methode ausgelagert. Um konsistent zur existierenden Benennung zu sein, wurden diese mit dem Präfix `Dlg_On` versehen. Bei der Klasse `TTopographyAdjustDlg` wurde ebenso verfahren (siehe **Abb. III.21**).

5) Neuordnung der Attribute

Jedes Attribut des Subsystems, das direkt oder indirekt für die Berechungen und Messvorgängen in der Topographie relevant ist, wurde in die Funktionskomponente verschoben. Zur eindeutigen Kennzeichnung wurde den Attributen dabei das Präfix `m_` vorangestellt.

Das Attribut `Detector` war in beiden Oberflächenklassen deklariert. Insgesamt wird aber nur eine Detektorreferenz benötigt, die nun durch die Funktionskomponente verwaltet wird. In allen drei Klassen wurde der Zugriffsschutz `private` für alle Attribute gewählt.

In der Dialogklasse `TTopographyExecDlg` verblieben nur die neuen Referenzen auf zwei Timer- und eine Funktionskomponente. In `TTopographyAdjustDlg` werden nur zwei Handle auf Kombinationsfelder und die neue Referenz auf die Funktionskomponente benötigt. Ein vorhandenes Flag `nRestrictions` – für den Aufrufstatus des Dialogfensters ‚Topographie-Einstellungen' – wurde durch den Wahrheitswert `m_bCtrlStatus` ersetzt. Nach dem Verschieben der funktionsspezifischen Attribute entstandenen für deren Zugriff jeweils `Get`- und `Set`-Methoden in `TTopography`. Da keine Bedingungen für den Schreibzugriff vorhanden waren, werden keine `CanSet`-Methoden benötigt; die `Get`- und `Set`-Methoden wurden `inline` deklariert.

6) Zusammenfassen von Funktionalitätsaufrufen

Das Finden und Zusammenfassen der F-Blöcke gestaltete sich unproblematisch, da die funktionsspezifischen Anweisungen zum großen Teil bereits zusammengefasst waren. Nur zu Beginn oder am Ende einer Methode einer Oberflächenklasse wurden oberflächenspezifische Aktionen ausgeführt. Das Zusammenfassen der F-Blöcke beschränkte sich meist auf die vielen verstreuten `Set`-Methoden, die zur Parametrisierung der Messvorgänge benötigten werden (siehe **Abb. III.17**).

```
void TTopographyExecDlg::Dlg_OnGotoWorkPoint( void ) {
  …
/* O */
  CtrlSetText( id_Text, _GoToStartpoint);     // Beschriftung ändern
/* F */
  SetAngleWidth( GetMoveStep() );             // Motorschrittweite setzen
/* O */
  CtrlSetEnabled(cm_SwitchControl, FALSE ); // 'Startposition anfahren' und
  CtrlSetEnabled(cm_GotoWorkPoint, FALSE ); // 'Regelung starten'  ausgrauen
/* F */
   // Anfahren des um dStartAngle vom aktuellen Stand entfernten Punktes
  Steering.Reset();
  … // weitere funktionsspezifische Anweisungsfolgen
};
                                                          AUSGANGSZUSTAND
```

```
void TTopographyExecDlg::Dlg_OnGotoWorkPoint( void ) {
  …
/* O */
  CtrlSetEnabled(cm_SwitchControl, FALSE ); // 'Startposition anfahren' und
  CtrlSetEnabled(cm_GotoWorkPoint, FALSE ); // 'Regelung starten'  ausgrauen
  CtrlSetText( id_Text, _GoToStartpoint);     // Beschriftung ändern
/* F */
  SetAngleWidth( GetMoveStep() );             // Motorschrittweite setzen
   // Anfahren des um dStartAngle vom aktuellen Stand entfernten Punktes
  Steering.Reset();
  … // weitere funktionsspezifischen Anweisungsfolge
};
                                                          ERGEBNIS
```

Abb. III.17 Zusammenfassen einer `Set`-Methode mit einem großen F-Block

7) Erzeugen von `Do`- Methoden

Für die Klasse `TTopography` entstanden 19 `Do`-Methoden. Die identifizierten F-Blöcke waren soweit zusammengefasst, dass innerhalb einer Methode der Oberflächenklasse höchstens eine `Do`-Methode gerufen werden muss. Die dabei entstandenen Methoden haben meistens sehr viele Parameter, weil die aus den benutzten Subsystemen gerufenen Methoden meist selbst viele Parameter erfordern. Die neu entstandene Methode `DoStartMeasure` ruft bspw. `StartCmdExecution` (Subsystem ‚Ablaufsteuerung') mit fünf Parametern! Durch die Dekomposition muss die Funktionskomponente diesen Methodenaufruf in einer `Do`-M. kapseln und benötigt damit auch mindestens fünf Parameter, wenn diese Eigenschaften nicht global in der Funktionskomponente gespeichert werden können.

Bei einer anderen Methode `DoResetTimes` resultieren die Parameter aus der Vielzahl an `Set`-Methoden, die in einem F-Block zusammengefasst sind. Die Folge ist ein unschönes Gesamtdesign (siehe **Tab. III.10**, **MNOP** 5). Alternativ hätten die Parameter auch in einer Struktur gesammelt übergeben werden können, obwohl dies wieder die Komplexität erhöht hätte.

```
// Durchführung einer Messung mittels Kommandoverarbeitung
BOOL  TTopography::DoStartMeasure (TCmdId aCmdId, int p1, int p2,
                                   LPSTR aStr, HWND aControlWnd) {
  …
  return Steering.StartCmdExecution(aCmdId, p1, p2, aStr, aControlWnd);
};

// Herstellen des Ausgangszustandes
void TTopography::DoResetTimes (DWORD aStart, DWORD aCurrent,
                                BOOL aRun, BOOL aAdd, BOOL aFinish) {
  SetStartTime  ( aStart );
  SetCurrentTime( aCurrent );
  SetTimeRunning( aRun );
  SetAdditionalTime( aAdd );
  SetTimeFinish ( aFinish );
};
```

Abb. III.18 Durch Dekomposition entstandene `Do`-Methoden mit langen Parameterlisten

8) Erzeugung von CanDo- Methoden (optional)

Eine Besonderheit der ‚Topographie' ist, dass keine CanDo-Methode benötigt werden, da die sich die Nutzeroberfläche nicht an Zustandsänderungen der Funktionskomponente anpassen muss. Die Bedingungen in den Do-Methoden prüfen nicht die Durchführbarkeit selbst, sondern beschränken sich auf die Auswahl eines bestimmten Messvorganges.

9) Erstellung von CanSet- und Gewinnung zusätzlicher Bedingungen für CanDo-Methoden (optional)

Werden im Dialog ‚Topographie-Einstellungen' unzulässige Werte eingegeben, erfolgt weder eine Korrektur dieser noch die Übernahme in die Funktionskomponente. Laut Verhaltensspezifikation soll nur eine Fehlermeldung ausgegeben werden, die den Anwender auffordert, den Wert selbst zu korrigieren. Weil die Übernahme der Eingabe nicht vom aktuellen Systemzustand abhängt, sondern nur vom zu setzenden Wert, konnte auf die Erstellung von CanSet-M. verzichtet werden. Die Bedingungen wurden direkt in die Set-Methoden implementiert. Die Nutzeroberfläche ruft die Set-Methode und gibt beim Fehlschlag eine Fehlermeldung aus.

Dabei zeigte sich, dass die Wertebereiche für die Eingaben zum Teil erheblich von der Spezifikation abwichen, weil die Genauigkeit statisch anstatt antriebsabhängig war. Teilweise fehlte die Grenzwertüberprüfung völlig. So mussten alle Bedingungen neu bewertet und implementiert werden (siehe **Abb. III.19**).

```
// Schrittweite zum Anfahren des Arbeitspunktes setzen
GetDlgItemText(GetHandle(), id_MoveStep, buf, 10);
valueF = atof(buf);
if ( valueF >= 0.0001 ) TopographyOld.fMoveStep = valueF; // Grenze
else return FALSE;
```
AUSGANGSZUSTAND

```
// Schrittweite zum Anfahren des Arbeitspunktes setzen
BOOL TTopography::SetMoveStep( float aStep ) {
  // Minimalwert in Abhängigkeit von der Nachkommastellengenauigkeit des Antriebs bestimmen
  float min= 1 / pow((float)10, (float)GetMotorDigits(eSF));
  if ( (aStep < min) || (aStep > 99999.0) ) return FALSE; // Grenzwerte

  m_fMoveStep= aStep;
  return TRUE;
};
```
ERGEBNIS

Abb. III.19 Erweiterung einer Set-Methode durch zusätzliche Bedingungen

Auch in der Nutzeroberfläche waren keine Bedingungen zur Ausführung von Do-Methoden vorhanden. In Zusammenhang mit dem vorhergehenden Schritt **8)** der Dekomposition, sind in der getrennten ‚Topographie' überhaupt keine CanDo-Methoden enthalten.

10) Entfernen gleicher Do-/ CanDo- Methode

In der Dialogklasse TTopographyExecDlg konnten die Einstellungen für den Detektor, auch während der Messung geändert werden. Ein ggf. laufender Messvorgang wird dazu unterbrochen, die Detektorparameter geändert und die Messung dann fortgesetzt. Dieses Verhalten war an verschiedenen Stellen implementiert, jedoch mit unterschiedlichen Aufrufbedingungen. Bei Schritt **7)** der Dekomposition entstanden so die Methoden DoSetDetecParams_Start, DoSetDetecParams_ReStart, und DoSetDetecParams_NoStart. Obwohl diese nicht völlig identisch waren, konnten sie in einer Methode DoSetDetectorParams zusammengefasst werden, weil die Kernfunktionalität, das Ändern der Parameter mittels

`Detector->SetExposureSettings`, die gleiche war. Indem der zusätzliche Parameter `BOOL Start` hinzugefügt wurde, konnten jede einzelne Möglichkeit der Benutzung nachgebildet werden (siehe **Abb. III.20**). Diese Art der Vereinfachung kann jedoch nur nach eingehender Analyse des Programmcodes durchgeführt werden; sie ist nicht verallgemeinerbar!

```
// Parameter ändern - Messung immer neu starten
void TTopography::DoSetDetecParams_Start(float Time, DWORD Count) {
  Detector->MeasureStop();   // Messung stoppen
  Detector->SetExposureSettings( TExposureSettings(Time, Counts));
  Detector->MeasureStart();   // Messung fortsetzen
}

// Parameter ändern - Messung nur starten wenn schon gemessen wurde
void TTopography::DoSetDetecParams_ReStart(float Time, DWORD Count) {
  // es wird gerade gemessen
  if ( Detector->IsActive() ) {
    Detector->MeasureStop();   // Messung stoppen
    Detector->SetExposureSettings(TExposureSettings(Time, Counts));
    Detector->MeasureStart();   // Messung fortsetzen
  }
  else  // es wird NICHT gemessen
    Detector->SetExposureSettings( TExposureSettings(Time, Counts) );
}

//  Parameter ändern - Messung nie starten
void TTopography::DoSetDetecParams_NoStart(float Time, DWORD Count) {
  Detector->SetExposureSettings(TExposureSettings(Time, Counts));
}
```

AUSGANGSZUSTAND

```
// Detektorparameter ändern
void TTopography::DoSetDetectorParams(float Time, DWORD Count, BOOL Start)
{
  // es wird gerade gemessen
  if ( (m_lnkDetector->IsActive()) || (Start) ) {
    m_lnkDetector->MeasureStop();   // Messung stoppen
    m_lnkDetector->SetExposureSettings(TExposureSettings(Time, Count));
    m_lnkDetector->MeasureStart(); // Messung fortsetzen
  }
  // es wurde NICHT gemessen oder es soll NICHT gestartet werden
  else m_lnkDetector->SetExposureSettings(TExposureSettings(Time, Count));
};
```

ERGEBNIS

Abb. III.20 Zusammenfassen gleichgesinnter `Do`-Methoden durch zusätzlichen Parameter

11) Kennzeichnung nicht-zustandsverändernder Methoden und konstanter Parameter (optional)

In der Funktionskomponente `TTopography` konnten alle 42 nicht-zustandsverändernden Methoden (betrifft alle `Get`-, `Is`- und `Has`-M.) als const-member-functions deklariert werden. Dies war auch bei 7 `Do`-Methoden möglich, weil dort keine Attribute aus `TTopography` geschrieben, sondern nur Funktionen anderer Subsysteme ausgeführt werden. In jeder vorkommenden Methode konnten alle Parameter als konstant gekennzeichnet werden, wie z.B. bei
`void DoSetTimes (const DWORD, const DWORD, const BOOL)`. Für die Methoden `SetDetector` und `SetMonitor` beschränkt sich das allerdings auf die Deklaration ihrer Parameter als konstante Zeiger, es entstanden so: `SetDetector(TDetector *const)` und `SetMonitor(TDetector *const)`.

12) Suche und Korrektur von Fehlern (optional)

Die verbesserte Lesbarkeit des Programmcodes hatte deutliche Auswirkungen auf die Fehlerbeseitigung. Es wurden 21 neue Fehler entdeckt und behoben. Die meisten davon

waren fehlerhafte Grenzwerte für die Eingabeparameter (siehe **9)**). Von den im Fehlerdokument (siehe **[M34]**) bereits vorhandenen 15 Fehlern konnten weitere 12 beseitigt werden. Dies umfasst das Entfernen von Totem Code, die Beseitigung redundanter Operationen und die Anpassung von Variablentypen. Um die Software-Ergonomie zu verbessern, wurden die Ressourcen der beiden Fenster überarbeitet. Dies beinhaltet die Minimierung von Freiflächen, die Ausrichtung an Fluchtlinien und die Gruppierung von Steuerelementen.

Die Verhaltensspezifikationen der Analysephase konnten unverändert bleiben, da sie nur allgemeine Verfahren zur Topographie beinhalten. Eine Anpassung wird erst dann nötig, wenn die Nutzeroberfläche völlig neu gestaltet wird.

Aus Sicherheitsgründen ist die ursprüngliche ‚Topographie' unverändert im ‚XCtl'-System verblieben. Die Fehler wurden nur in der dekomponierten Version entfernt. Die korrigierten Fehler sind im Fehlerdokument (siehe **[M34]**) nicht entfernt, sondern besonders gekennzeichnet.

13) Abschluss durch Fein- und Abnahmetest

Aufgrund der vielen Korrekturmaßnahmen am Programmcode und der geänderten Funktionalität bzgl. des Eingabeverhaltens wurde vor dem eigentlichen Abnahmetest ein Feintest durchgeführt. Das Subsystem ‚Topographie' wurde funktional gegen die Spezifikation (Black-Box-Testverfahren) getestet.

Zur Überprüfung des Verhaltens bei Eingaben innerhalb und außerhalb der jeweiligen Wertebereiche entstand ein CTE-Diagramm mit 35 Testfälle und 39 Testdaten (siehe **[M37]**). Die jeweiligen Grenzwerte wurden Verhaltensspezifikation **[M31]** entnommen.

Der Test der Oberflächenfunktionalität umfasst das Verhalten der Eingabefelder des Dialogs ‚Topographie-Einstellungen' während der beiden Messungen. Für die 15 Felder waren 30 Testdaten (jeweils: freigegeben und gesperrt) und vier Testfälle, zwei für die Art des Aufrufs und zwei für die Art der Messung, nötig (siehe **[M38]**). Für den Dialog ‚Topographie-Durchführung' wurde ein CTE-Diagramm mit 10 Testfälle und 23 Testdaten erstellt, um Reaktionen auf ungültige Zahlenformate im Eingabefeld und den Status der Schaltflächen während und nach den Messvorgängen zu testen.

Für den abschließenden Regressionstest konnten die Testfälle für die ursprüngliche ‚Topographie' (siehe **[M36]**) genutzt werden. Diese führen komplette Messungen mit bestimmten Parametern durch und vergleichen die Ergebnisse mit gespeicherten Referenzwerten.

14) Dokumentation des entstandenen Designs

Das Dokument „Software-Struktur" (**[M29]**) war nach der Dekomposition veraltet und musste durch eine neue Dokumentation ergänzt werden. Dazu wurde am Ende der Dekomposition, entsprechend der Vorgaben aus **II.3.4**, ein neues Designdokument für die getrennte ‚Topographie' erstellt (siehe **[M35]**).

In der alten Dokumentation waren neben dem Subsystem ‚Topographie' auch die benutzten Subsysteme erläutert. Die Erläuterungen zur Anbindung der ‚Motorsteuerung' konnten entfallen, da sie bereits in den selbst erstellten Artekfakten **[M26]** bzw. **[M28]** vorhanden sind. Für die Ausführungen zur ‚Ablaufsteuerung' gilt das gleiche, da für dieses Subsystem eigene Designdokumente mit ausführlichen Klassenbeschreibungen vorhanden sind (siehe **[M4]** bzw. **[M6]**).

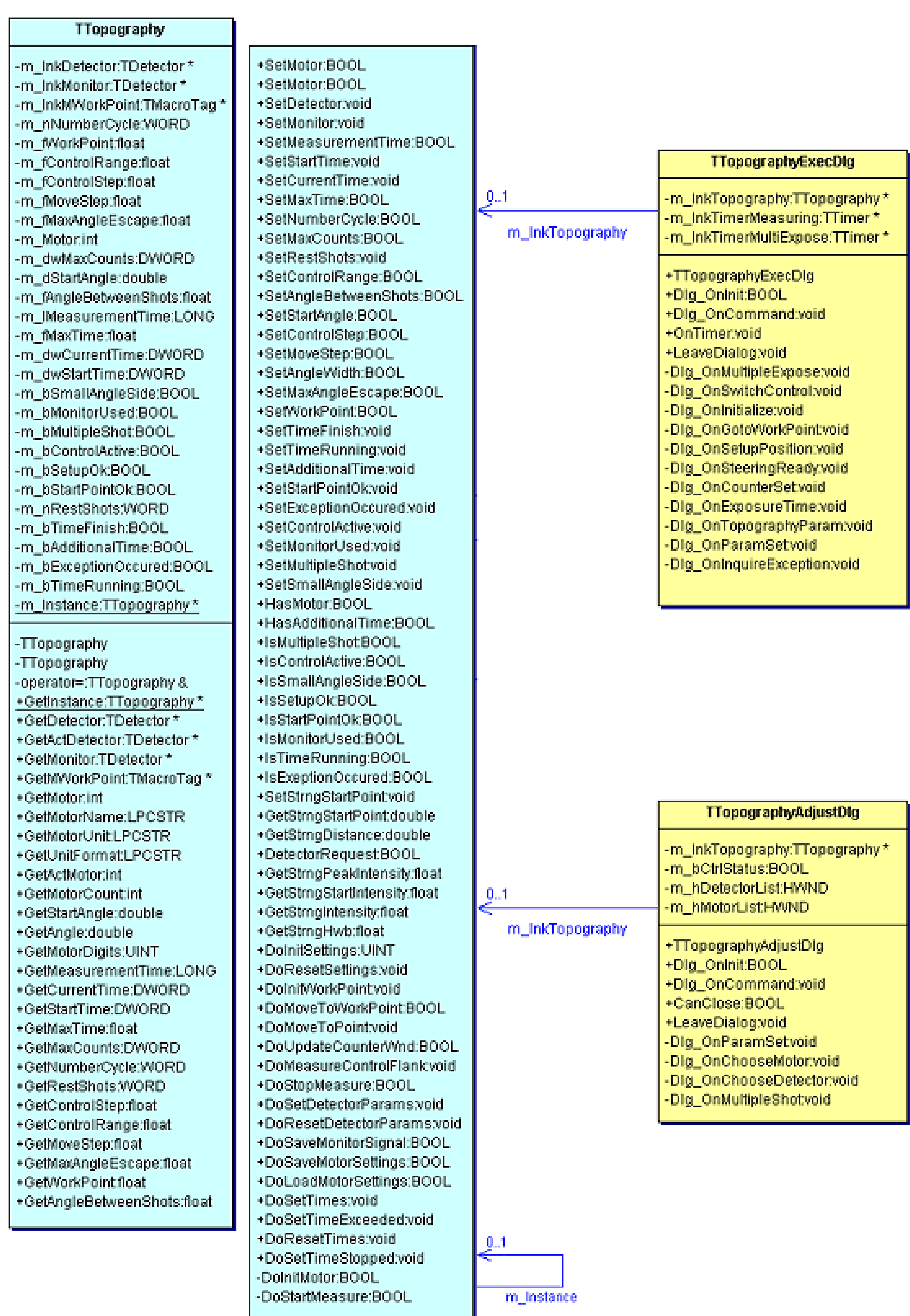


Abb. III.21 UML-Klassendiagramm der getrennten ‚Topographie'

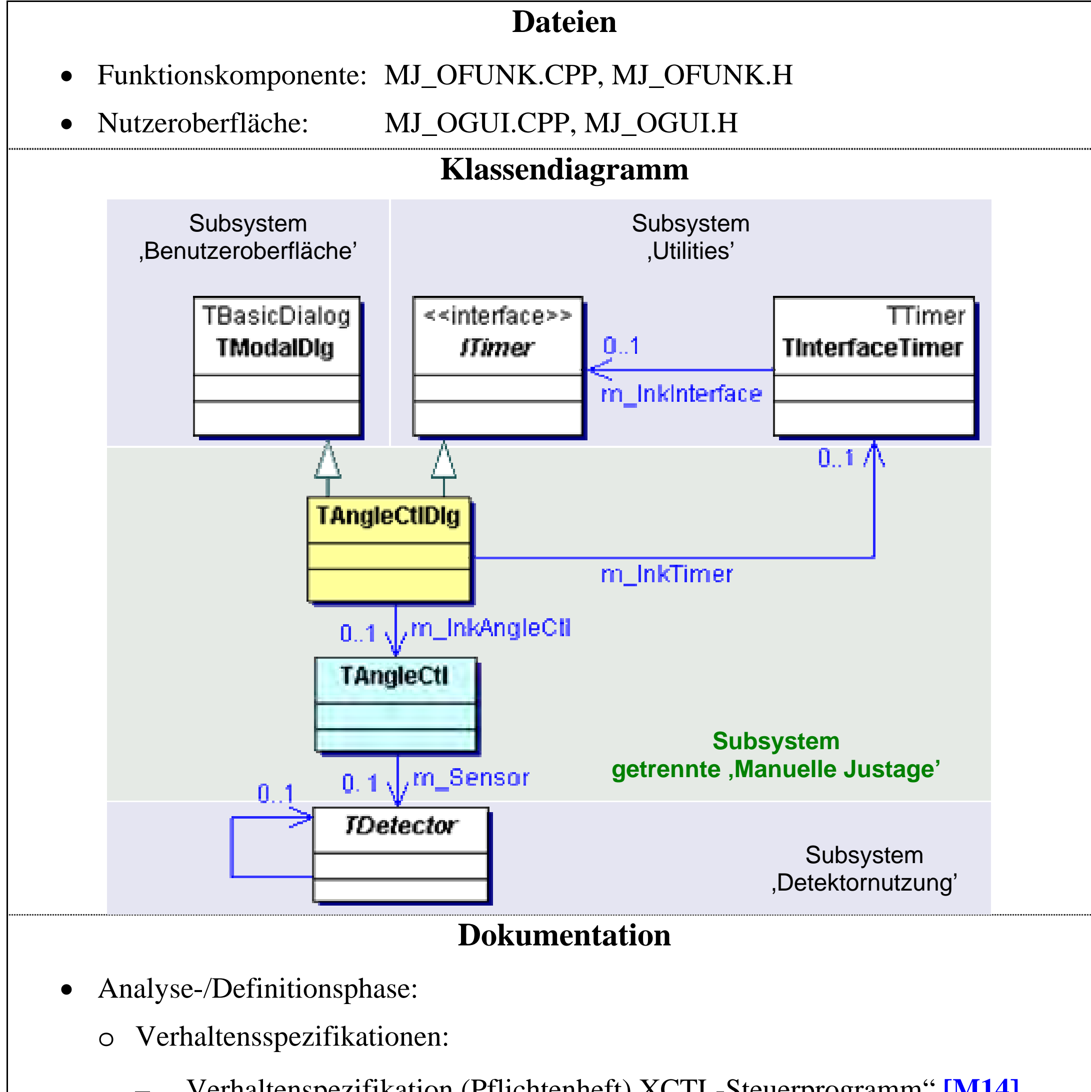
III.2.2 Zusammenfassung

Anwendungsfall ‚Manuelle Justage'

Dateien

- Funktionskomponente: MJ_OFUNK.CPP, MJ_OFUNK.H
- Nutzeroberfläche: MJ_OGUI.CPP, MJ_OGUI.H

Klassendiagramm



Dokumentation

- Analyse-/Definitionsphase:
  - Verhaltensspezifikationen:
    - „Verhaltenspezifikation (Pflichtenheft) XCTL-Steuerprogramm" **[M14]**
  - eine kurze Use-Case-Beschreibung
- Design-/Implementierungsphase: –
- Testphase:
  - Regressions-Testfälle „Manuelle Justage", Version 2.2 von Jens Hanisch **[M21]**
- Fehlerübersicht:
  - Fehler in der alten Manuellen Justage (**NEU**) **[M19]**

Tab. III.3 Übersicht zur getrennten ‚Manuellen Justage' (Polling-Variante)

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TAngleCtl	370	4	46	1	0	10	0	90	36	2	1	44
TAngleCtlDlg	438	6	37	1	6	82	13	4	46	4	1	38

Tab. III.4 Ausgewählte Metriken für die Klassen bei der getrennten ‚Manuellen Justage' (Polling-Variante)

Am anschaulichsten kann der Vergleich zwischen dem Ausgangszustand und dem dekomponierten System anhand der Klassendiagramme von Tab. II.1 und Tab. III.3 (Polling-Variante) bzw. Abb. III.22 (Observer-Muster) geführt werden. Augenscheinlich ist bei der Dekomposition zuerst das komplexere Design. Hierfür ist insbesondere die Timer-Komponente verantwortlich, die bei der Polling-Variante mit der Nutzeroberfläche und beim Observer-Muster mit der Funktionskomponente in Beziehung steht (ausführlich siehe 3)). Trotz der komplexen Anbindung werden zur Steuerung des Timers nur 6 LOC benötigt.

Deutlich sichtbar wird beim Vergleich mit dem ursprünglichen System auch die Trennung der benutzten Subsysteme von der Nutzeroberfläche durch die Funktionskomponente. Die meist nicht-objektorientierte Anbindung dieser Systeme an die ursprünglichen ‚Manuellen Justage' wurde auch in die getrennten Variante übernommen. Damit ist die deutliche Schichten-Architektur – wie bei der Neuentwicklung (vgl. Abb. II.37) – im Klassendiagramm nicht erkennbar.

Während der Dekomposition konnte die vorhandene Fehlerliste [M19] von 1 auf 10 Fehler erweitert werden – dies allein mit Fehlern, die für den Anwender sichtbar sind. Davon konnten 50 % bereits während der Schrittfolge, einige sogar allein durch Anwendung dieser, korrigiert werden (siehe 12)). Die intensive Beschäftigung mit dem Programmcode im Laufe der Dekomposition erlaubte es, die übrigen Fehler am Ende der Dekomposition zu entfernen.

Die Dekomposition macht sich jedoch negativ im Programmcodeumfang bemerkbar. Von ursprünglich 457 **LOC** ist die Anzahl auf 808 **LOC** gestiegen (Zuwachs von 85 %) – vgl. Tab. II.2 und Tab. III.4. Abzüglich der deutlich gestiegenen Kommentierung relativiert sich der Zuwachs auf nur noch 40 % – von 342 LOC auf 479 LOC. Verantwortlich ist dafür fast ausschließlich die in 4) vollzogene Auslagerung der Behandlung von Steuerelementbotschaften aus `Dlg_OnCommand` in separate Methoden. Dadurch waren allein für die mit Funktionalität belegen Steuerelemente mindestens 9 zusätzliche LOC notwendig. Für die 20 Steuerelemente ergibt dies allein 180 LOC.

Durch die Verteilung des Programmcodes verringert sich der durchschnittliche Methodenumfang für die Nutzeroberfläche wesentlich. Aus ursprünglich 57 LOC (8 $\mathbf{NOO_O}$) wurden 11 LOC (37 $\mathbf{NOO_O}$). Die Anzahl der Attribute sank von 11 $\mathbf{NOA_O}$ auf 6 $\mathbf{NOA_O}$. Im Vergleich dazu besitzt die Funktionskomponente einen durchschnittlichen Umfang von 8 LOC (46 $\mathbf{NOO_F}$) und 4 $\mathbf{NOA_F}$ Attribute[24].

Der Vergleich von Forward und Reengineering ist am Beispiel der ‚Manuellen Justage' nicht einfach. Erstens hat die neue ‚Manuellen Justage' einen wesentlich höheren Funktionsumfang, dies sowohl in der Funktionskomponente als auch der Nutzeroberfläche. Zweitens muss hier auch der von den Mitarbeitern des Physikalischen Institutes gewählte Oberflächenentwurf mit in die Bewertungskriterien einfließen[25]. Allein dies führt zu einem wesentlichen höheren LOC/ NOA/ NOO-Aufkommen auf Seiten der Nutzeroberfläche der neuen ‚Manuellen Justage'.

Drittens haben sich die Autoren dieser Arbeit für die objektorientierte Anbindung bei der neuen ‚Manuellen Justage' entschieden, was zusätzliche LOC bei der Funktionskomponente verursacht. Der Grund ist, dass die nicht-objektorientierte Schnittstelle des Subsystems ‚Motorsteuerung' keine Möglichkeit einer sicheren Anbindung bietet. Im getrennten Subsystem musste die größtenteils nicht-objektorientierten Anbindung durch die Schrittfolge (aus dem ursprünglichen System) übernommen werden.

[24] für Gründe/ Möglichkeiten/ Vorteile zur Reduzierung der Attributanzahl siehe
[25] die drei Teilbereiche im Hauptdialogfenster und die zusätzlichen Offset-Dialoge tragen dem Rechnung

Weil ein Vergleich der Metriken daher kaum Sinn macht, wird an dieser Stelle die Wiederverwendbarkeit der beiden Funktionskomponenten betrachtet. Nach Ansicht der Autoren ist der Austausch der Funktionskomponenten untereinander einfach möglich. Der Einsatz von TAngleCtl in der neue ‚Manuellen Justage' hieße jedoch ein Verzicht auf einen Teil der zusätzlichen Funktionalität von TManJustage (siehe **Tab. III.6**). Die drei Teilbereiche im Hauptdialogfenster können trotzdem problemlos angesteuert werden. Die in **Tab. III.5** aufgeführten Methoden realisieren in beiden Funktionskomponenten dieselbe Funktionalität – abgesehen von der Art der Ansteuerung der benutzten Subsysteme und der Robustheit.

Die Methodenbezeichner wurden während der Dekomposition im Reengineering vom Forward Engineering übernommen. Wo die Übernahme nicht möglich war, zeigt **Tab. III.7**. Grundlage dieses Vergleichs sind **Abb. II.46** bzw. **Abb. III.14**. und die Analyse des Programmcodes.

DoDirect, CanDoDirect, DoDrive, CanDoDrive, DoStep, CanDoStep, DoStop, CanDoStop, DoStopEverything,
GetMotorCount, GetMotorIdx, IsIndexValid, HasOffset, IsMoving, IsCalibrated, GetAngle, GetAngleMin, GetAngleMax, GetDigits, GetSpeed, SetSpeed, CanSetSpeed, GetAngleWidth, SetAngleWidth, CanSetAngleWidth, GetUnit, GetMotorName, SetRelativeZero, CanSetOffset, ResetRelativeZero, CanResetRelativeZero, UpdateDetector,
DoStartMeasure, CanDoStartMeasure, DoStopMeasure, CanDoStopMeasure, IsMeasuring, IsMeasureReset,
GetMeasureHwb

Tab. III.5 Identisches Funktionsangebot von **TManJustage** und **TAngleCtl**,
Benennung wurde von **TManJustage** übernommen = 39 Methoden

TManJustage	realisierte Funktionalität (die **TAngleCtl** nicht leisten kann)
GetMotionType, SetMotionType, CanSetMotionType, GetAngleDest, SetAngleDest, CanSetAngleDest	Wiederherstellen aller zuletzt benutzten Antriebsparameter
GetOffset, SetOffset, CalcOffsetFromAngle, CalcAngleFromOffset	Offset für Antrieb
GetDetector, GetDetectorName, HasDetectorAxis, GetChannel, SetChannel, ResetChannelOffset, GetAnglePerChannel, GetDetectorAxisIdx, CalcChannelOffset, CalcChannelAngle	Offset für Detektor
GetMeasureProgress, ParsingAxis	zusätzliche Informationen, wie Fortschritt der Messung, in der Nutzeroberfläche
GetKzMoving, SetKzMoving, GetKzMeasuring, SetKzMeasuring, BeginUpdate, EndUpdate, DetermineMoving, DetermineAngle	Geschwindigkeitsoptimierungen

Tab. III.6 Zusätzlich geforderte Funktionalität für **TManJustage** = 30 Methoden

TAngleCtl	realisierte Funktionalität (kein entsprechendes Pendant in **TManJustage**)
DoInit	Initialisierung beim Aufruf des Dialogfensters (in TManJustage im Konstruktor beinhaltet)
DoInitMotor, SetCorrectionState	muss vor dem Starten der Antriebsbewegung aufgerufen werden (in TManJustage in DoDirect, DoStep, DoDrive beinhaltet)
Moves, MeasureStopped	private Hilfsmethoden (TAngleCtl intern)

Tab. III.7 Durch die Dekomposition entstandene Methoden von **TAngleCtl**,
die beim Einsatz zu beachten sind = 5 Methoden

III.2.2–1 Bewertung von Observer-Muster und Polling im Anwendungsfall

Im Gegensatz zur neuen ‚Manuellen Justage' wurden Polling-Variante und Observer-Muster für die getrennte Version parallel erstellt. Hier gab es keine Unterschiede für die Schrittfolge, so dass sie robust genug ist, die beiden Kommunikationsmuster zuzulassen. Das resultierende Design zeigen **Abb. III.22** (Observer-Muster) und das Klassendiagramm aus **Tab. III.3** (Polling-Variante).

Wie beim Forward Engineering zeigt der Vergleich des Implementierungsumfanges wieder einen kleinen Nachteil von 8 **LOC** für das Observer-Muster, 816 zu 808 (Polling-Variante) – vgl. **Tab. III.8** und **Tab. III.4**.

Bei der Nutzeroberfläche ist die Polling-Variante mit 438 $\mathbf{LOC_O}$ etwas umfangreicher als beim Observer-Muster 414 $\mathbf{LOC_O}$. Entsprechend ist der Umfang der Funktionskomponente komplementär, d.h. das Observer-Muster hat 402 $\mathbf{LOC_F}$ und die Polling-Variante 370 $\mathbf{LOC_F}$. Der Unterschied zwischen den beiden Kommunikationsmustern ist wie beim Forward Engineering minimal. Es entsteht mit der Auswahl eines Musters kaum zusätzlicher Aufwand für die Realisierung der Kommunikation. Vielmehr besteht der Unterschied in einer Umverteilung der Verantwortlichkeiten! Die Anzahl von verschiedenen Arten der auszutauschenden Nachrichten schlägt sich im Interface des Observer-Musters nieder. Damit wächst die Differenz zwischen den beiden Mustern mit steigender Komplexität der Kommunikation. Dies darf jedoch nicht auf die Anzahl der ausgetauschten Nachrichten reduziert werden. Wenn nur ein einzelner Nachrichtentyp immer wieder versendet wird, macht dies keinen Unterschied zwischen Observer und Polling!

Abschließend wird noch einmal darauf hingewiesen, dass die Wahl des Kommunikationsmusters im Reengineering zusätzlich vom vorhandenen Muster im ursprünglichen System abhängt. In allen anderen Fällen müssen wieder die in **II.3.3–4** genannten Kriterien herangezogen werden. Verallgemeinert sollte das Observer-Muster, aufgrund des umfangreicheren Designs, nur eingesetzt werden, wenn die Polling-Variante nicht anwendbar oder zu fehleranfällig ist.

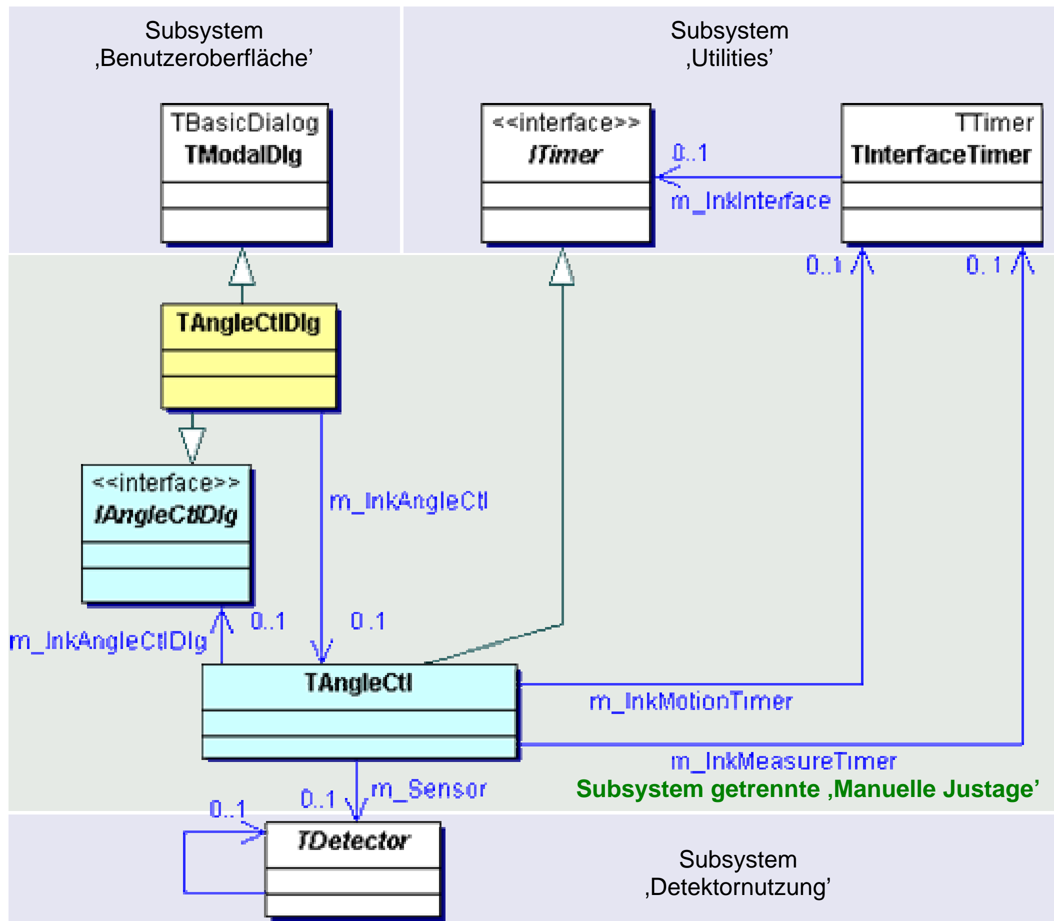
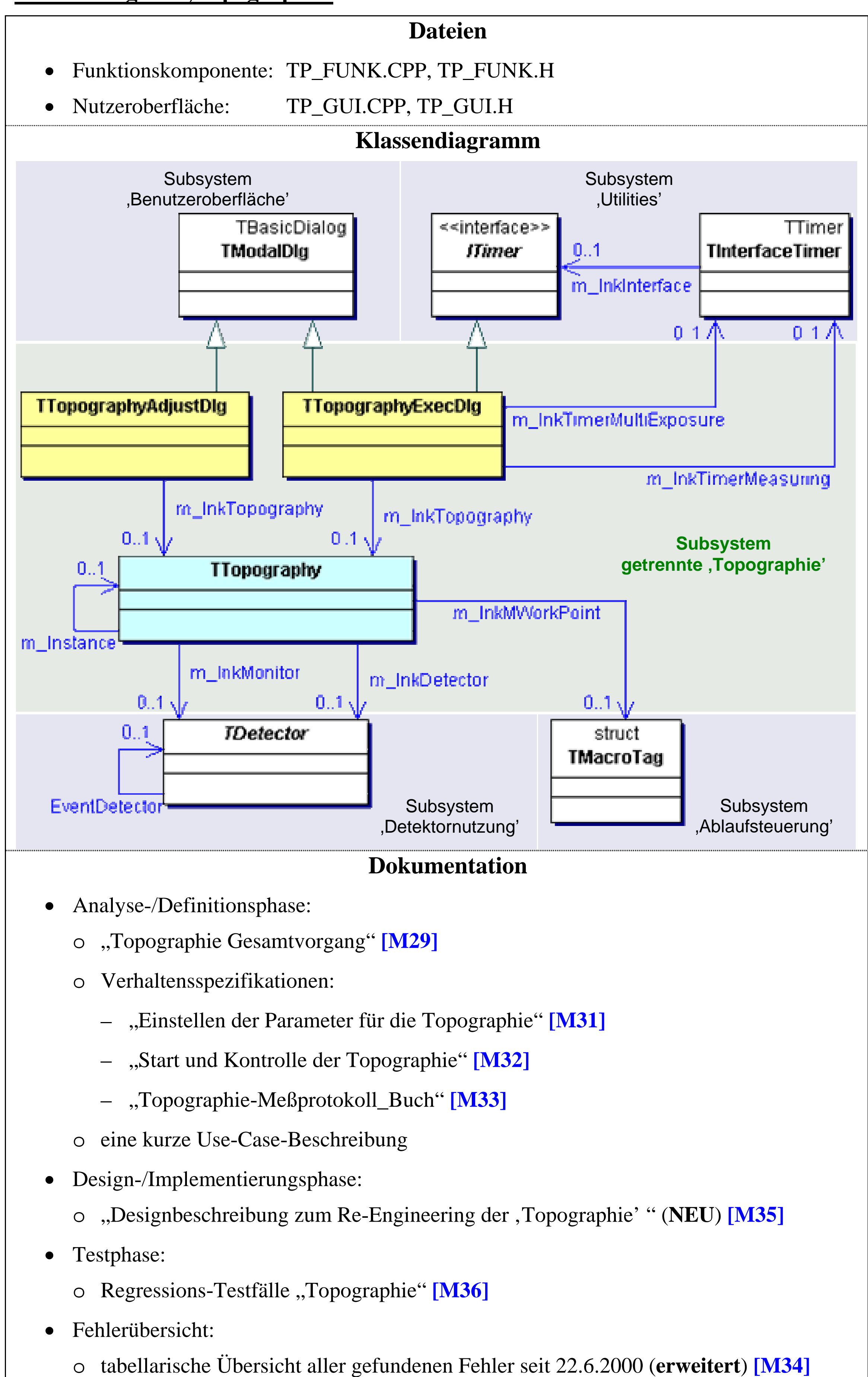


Abb. III.22 UML-Klassendiagramm für die getrennte ‚Manuellen Justage' (Observer-Muster mit Übersicht der benutzten Subsysteme)

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TAngleCtl	402	7	47	1	0	16	2	82	63	2	1	41
TAngleCtlDlg	414	5	35	1	10	71	24	5	37	4	1	39

Tab. III.8 Ausgewählte Metriken für die Klassen bei der getrennten ‚Manuellen Justage' (Observer-Muster)

Anwendungsfall ‚Topographie'



Dateien

- Funktionskomponente: TP_FUNK.CPP, TP_FUNK.H
- Nutzeroberfläche: TP_GUI.CPP, TP_GUI.H

Klassendiagramm

Dokumentation

- Analyse-/Definitionsphase:
  - „Topographie Gesamtvorgang" **[M29]**
  - Verhaltensspezifikationen:
    - „Einstellen der Parameter für die Topographie" **[M31]**
    - „Start und Kontrolle der Topographie" **[M32]**
    - „Topographie-Meßprotokoll_Buch" **[M33]**
  - eine kurze Use-Case-Beschreibung
- Design-/Implementierungsphase:
  - „Designbeschreibung zum Re-Engineering der ‚Topographie' " (**NEU**) **[M35]**
- Testphase:
  - Regressions-Testfälle „Topographie" **[M36]**
- Fehlerübersicht:
  - tabellarische Übersicht aller gefundenen Fehler seit 22.6.2000 (**erweitert**) **[M34]**

Tab. III.9 Übersicht zur getrennten ‚Topographie'

Klasse	LOC (0, 1000)	NOA (0, 30)	NOO (0, 50)	NOCON (0, 5)	NOOM (0, 10)	PPrivM	PProtM (0, 10)	PPubM	AC	MNOP (0, 4)	NOC	TCR (5, 100)
TTopography	472	29	93	2	0	27	0	73	197	5	1	44
TTopographyExecDlg	398	3	15	1	4	74	0	26	27	4	1	38
TTopographyAdjustDlg	195	4	8	1	4	62	0	38	36	4	1	48

Tab. III.10 Ausgewählte Metriken für die Klassen der getrennten ‚Topographie'

Trotz des unzweckmäßigen Designs mit zwei sich gegenseitig rufenden Nutzeroberflächen gestaltete sich die Dekomposition unkompliziert. Eine zukünftige Zusammenlegung der Dialoge, wie bereits in der Projektgruppe geplant, ist problemlos möglich. Dann sollten sich Vorzüge der Dekomposition deutlich zeugen und eine effiziente Neuimplementierung ist möglich, d.h. schnell und fehlerresistent.

Durch die Zerlegung der Komplexität bei der Dekomposition konnte eine Vielzahl der vorhandenen Fehler beseitigt werden. Einige grundlegende Probleme sind auch nach der Dekomposition erhalten geblieben, die jedoch erst nach Änderung der Spezifikation und Absprache mit den Mitarbeitern des Physikalischen Institutes gelöst werden können. Das betrifft hauptsächlich den Zeitpunkt und die Art[26] der Auswertung von Nutzereingaben.

Durch die verbesserte Übersichtlichkeit und Kenntnisse der Spezifikation konnte in Schritt **10)** die Robustheit gesteigert werden, da mehrfach genutzte Funktionalität nur an einer Stelle implementiert ist.

Durch die Dekomposition ist ein umfangreicher Codezuwachs bei alle Klassen festzustellen, insgesamt 738 **LOC** (ursprüngliche ‚Topographie') zu 1.065 (getrennte ‚Topographie') – vgl. **Tab. III.1** und **Tab. III.10**. Am größten ist der Zuwachs bei der Funktionskomponente, mit ca. 468 %! Die Komplexität der Nutzeroberflächen hat sich dabei nur geringfügig geändert.

Die Anzahl der Methoden hat in der Nutzeroberfläche von $\mathbf{NOO_O}$ 8 auf 23 deutlich zugenommen. Wie bei der getrennten ‚Manuellen Justage' ist erkennbar, dass die Anzahl der `On`-Methoden direkt proportional zur Anzahl der Steuerelemente ist. Durch die Auslagerung von Programmcode aus `Dlg_OnCommand` konnte diese Methode von LOC 314 auf 61 reduziert werden. Jede neue Methode benötigt jedoch zusätzliche 6 LOC. Trotzdem konnte der Umfang der Nutzeroberfläche von $\mathbf{LOC_O}$ 655 auf 593 reduziert werden, bei deutlich gesteigerter Übersichtlichkeit.

Die Funktionskomponente ist von $\mathbf{NOO_F}$ 1 auf 93 angewachsen. Das liegt zum großen Teil am neu hinzugefügten Zugriffsschutz, der sich in fast 60 Accessor- und Mutator-Methoden ausdrückt. Einen erheblichen Anteil an LOC benötigen diese für die Überprüfung der Gültigkeitsbereiche, die oftmals durch die dynamischen Grenzen sehr komplex sind. Die ausgelagerte Funktionalität beansprucht den restlichen Teil NOO. Nachteilig ist dort die Entstehung langer Parameterlisten, was aber stark vom Design der benutzten Subsysteme (Schritt **7)**) abhängig ist (**MNOP** von 0 auf 5). Die $\mathbf{LOC_F}$ ist zwar von 83 auf 472 gestiegen, dafür sind fast alle Fehler korrigiert und die Robustheit verbessert worden.

---

[26] die Aufforderung an den Anwender seine Eingabe selbst zu korrigieren, ohne einen Hinweis auf den Gültigkeitsbereich

III.3 Fazit und Ausblick

Im Gegensatz zum Forward Engineering ist eine deutliche, phasenorientierte Dekomposition beim Reengineering nicht möglich. Nur während des Reverse Engineering, zur Ermittlung des Ist-Zustands, ist die Bearbeitung – in der Reihenfolge Test, Implementierung, Design, Analyse und Definition – durchführbar. Bei der eigentlichen Dekomposition liegt der Schwerpunkt auf der Arbeit am Programmcode. Die Analyse- und Definitionsphase entfällt während dieser Arbeit komplett und wird durch das Reverse Engineering ersetzt. Das Design entsteht indirekt aus der vorgestellten Schrittfolge. Die Testphase wird wiederholt am Ende jedes Schrittes durchgeführt, um die Arbeit des dekomponierenden Entwicklers zu verifizieren.

1) **Erstellung neuer Dateien**
2) **Behandlung von Ansätzen vorhandener Funktionskomponenten**
3) **Solldesign festlegen**
4) **Programmcodelayout verbessern (optional)**
5) **Neuordnung der Attribute**
6) **Zusammenfassen von Funktionalitätsaufrufen**
7) **Erzeugen von `Do-` Methoden**
8) **Erzeugung von `CanDo-` Methoden (optional)**
9) **Erstellung von `CanSet-` und Gewinnung zusätzlicher Bedingungen für `CanDo-` Methoden (optional)**
10) **Entfernen gleicher `Do-/ CanDo-` Methoden**
11) **Kennzeichnung nicht-zustandsverändernder Methoden und konstanter Parameter (optional)**
12) **Suche und Korrektur von Fehlern (optional)**
13) **Abschluss durch Fein- und Abnahmetest**
14) **Dokumentation des entstandenen Designs**

Der dadurch entstehende Programmcode ist im dekomponierten Subsystem fast immer wesentlich umfangreicher als der des ursprünglichen Subsystems. Das resultiert aber nur aus der Aufteilung der vorher zusammenhängenden Programmfunktionalität in viele kleine, übersichtlichere Methoden. Dies beinhaltet hauptsächlich die Schritte **5)** bis **9)**, wo die geforderte Robustheit und Wiederverwendbarkeit der Funktionskomponente sichergestellt werden. Zudem verbessern diese Methoden die Lesbarkeit und Verständlichkeit des Programmcodes. Damit werden Wartung und Fehlersuche erleichtert. Zudem können Fehler meist sofort der Nutzeroberfläche oder der Funktionskomponente zugeordnet werden.

Obwohl die Schrittfolge algorithmisiert ist, sind leider nicht alle Teile automatisierbar; dies betrifft in jedem Fall **2)**, **3)**, **6)**, **9)**, **12)** und **13)**. Da diese Schrittfolge nur an zwei Anwendungsfällen und in einer Programmiersprache exemplarisch durchgeführt werden konnte, ist die statistische Aussagefähigkeit sehr eingeschränkt. Erforderlich wäre hierzu die Anwendung in einer Vielzahl von unterschiedlichen Projekten auf verschiedenen Plattformen und Programmiersprachen. Vielleicht lässt sich für jede Programmiersprache eine angepasste Schrittfolge erstellen, um den Grad der Algorithmisierbarkeit weiter zu erhöhen.

Wünschenswert ist die Erstellung von programmcode-analysierenden Werkzeugen, die den Programmierer bei seinen Entscheidungen zur Dekomposition unterstützen. Sinnvoll wäre z.B. ein Werkzeug zum Finden von nicht-zustandsverändernden Anweisungen.

Interessant wird zudem die Bewertung der Dekomposition durch andere Entwickler im ‚XCtl'-Projekt beim Austausch der Nutzeroberfläche oder bei Wartungs- und Portierungsarbeiten.

Kapitel IV

GESAMTFAZIT

Da im Forward Engineering nur ein Anwendungsfall und im Reengineering nur zwei Anwendungsfälle ein und desselben Projektes betrachtet werden konnten, besteht nur eine beschränkte Aussagefähigkeit der vorgestellten Erkenntnisse. Die Verwendung nur einer Programmiersprache lässt zunächst eine zusätzliche Beschränkung vermuten. Da es sich dabei aber um C++ handelt, kann diese Sprache dennoch als repräsentativ für alle objektorientierten Programmiersprachen angesehen werden. Die Übertragbarkeit wird gesichert, weil nur Elemente der Schnittmenge von Sprachelementen[27] aller objektorientierten Programmiersprachen verwendet werden.

Sowohl beim Forward als auch beim Reengineering liegt der Schwerpunkt der Dekomposition auf der Wahl eines Musters für die Kommunikation zwischen Funktionskomponente und Nutzeroberfläche. Diese wird durch die in **II.3.3–5** vorgestellten Richtlinien eindeutig vorgeschrieben. Bei gleichen Spezifikationen entsteht, unabhängig von Forward oder Reengineering, das gleiche Design. Auch bei der Strukturierung der öffentlichen Schnittstelle der Funktionskomponente zeigt sich derselbe Effekt. Unter Verwendung der unter **II.2.5** und **II.3.1** (Forward) bzw. **III.2.1** (Reengineering) vorgestellten Richtlinien entsteht das gleiche Ergebnis, das sich zudem durch eine einheitliche und hohe Robustheit auszeichnet. Dabei lässt sich die Basisfunktionalität anhand des `Do`-Methoden-Präfixes leicht lokalisieren.

Diese Richtlinien führen beim Forward und Reengineering zwangsläufig zu einer besseren Verteilung der Komplexität auf mehrere Methoden und zu einer daraus resultierenden hohen Übersichtlichkeit. Der Unterschied besteht nur in der Herangehensweise. Beim Neuentwurf entsteht dies schrittweise und phasenorientiert aus der Produktspezifikation, wo hingegeben beim Reengineering die vorhandene Komplexität restrukturiert und verteilt wird.

[27] Klassenkonstrukt mit Methoden und Attributen; Referenzparameter
(verwendetet, aber optional: Zugriffsschutz für Attribute und Methode; Methoden mit Rückgabewerten; Kennzeichnung nicht-zustandsverändernder Methoden und konstanter Parameter; Klassenvererbung; automatische Ereignisgenerierung (Timer) für Kommunikationsmuster Polling); Interfacevererbung mit abstrakten Methodendeklarationen für Kommunikationsmuster)

Eine deutliche phasenorientierte Bearbeitung ist hier unmöglich (siehe **III.2**), weil sich die Schrittfolge vorrangig auf die Implementierung bezieht. Gleiches gilt auch für die Arbeitsteilung, die nur bei der Dekomposition von Subsystemen beim Forward Engineering anwendbar ist.

Die wesentlichen Nachteile der Dekomposition sind in beiden Softwareentwicklungsprozessen die verschlechterten Laufzeiteigenschaften und ein erhöhter Entwicklungsaufwand, der sich in einem deutlich umfangreicheren Programmcode zeigt. Letzteres wirkt sich aber nicht negativ auf die Wartbarkeit aus. Im Gegensatz zu einem nicht-dekomponierten System werden Erweiterbarkeit, Fehlersuche und -korrektur sogar deutlich verbessert. Die Ursache für die schlechten Laufzeiteigenschaften bildet die hohe Robustheit der Funktionskomponente, die jeweils überprüft, ob die gewünschte Aktion im aktuellen Programmzustand zulässig ist. Wenn man auf den Aufruf von `Can`-Methoden innerhalb der Funktionskomponente verzichtet, kann die Robustheit gegen Laufzeit ausgetauscht werden. Dies ist von der Spezifikation der Qualitätsmerkmale im Produktmodell abhängig und somit von Anwendungsfall zu Anwendungsfall zu entscheiden. Zudem besteht die Funktionskomponente zum Großteil aus öffentlichen Methoden, die sich in einer hohen PubM-Metrik widerspiegeln. Die umfangreiche Schnittstelle ist jedoch erforderlich, damit die Nutzeroberfläche auf die angebotene Basisfunktionalität zugreifen kann.

Zu den Vorteilen der Dekomposition zählt die Möglichkeit, dass Funktionskomponente und Nutzeroberfläche getrennt getestet werden können. Neben dem sonst üblichen Test der Nutzeroberfläche kann die Funktionskomponente zusätzlich einem White-Box-Testverfahren unterzogen werden. Sie benötigt hierzu aber eine Testumgebung, welche z.B. ein Debugger oder eine Testoberfläche sein kann. Die Nutzeroberfläche behindert dieses durch die schwer anzusteuernde Ereignisgenerierung von Nutzerinteraktionen.

Den Anstoß für die Dekomposition von Anwendungen bildet die Wiederverwendbarkeit der Funktionskomponente mit dem daraus entstehenden, wesentlichen Vorteil, die Nutzeroberfläche einfach austauschen zu können. Jede neu entwickelte Klasse der Nutzeroberfläche beschränkt sich nur auf die Verknüpfung der Nutzerinteraktionen mit der, von der Funktionskomponente bereitgestellten, Funktionalität. Da diese unverändert bleibt, kann die Basisfunktionalität als Fehlerquelle ausgeschlossen werden. Damit beschränkt sich die Fehlersuche bei einer neuen Nutzeroberfläche wirklich nur auf die Nutzeroberfläche.

Es hat sich gezeigt, dass die in Kapitel **I.4** erläuterte allgemeine Motivation für die Dekomposition sich vollständig bestätigt hat. Sowohl beim Forward als auch beim Reengineering konnten alle dargelegten Vorteile nachgewiesen und, wie erwartet, umgesetzt werden. Aus Sicht der Autoren stellt die Dekomposition stets eine Verbesserung der Softwareentwicklung dar, da sie sich sowohl auf den Entwicklungsprozess als auch die Eigenschaften des Produktes positiv auswirkt. Dekomposition hat stets verbesserte Strukturierung des Programmcodes zur Folge. Dies gilt auch für die Artefakte der Softwareentwicklung. Wie in dieser Arbeit gezeigt, lässt sich auch die Struktur dieser Dokumente verbessern. Wird zusätzlich eine starke Formalisierung innerhalb der Artefakte umgesetzt, könnte der Softwareentwicklungsprozess teilautomatisiert werden. Als mögliches Ziel steht dann die automatische Gewinnung von Programmcode aus den Entwicklerdokumenten und umgekehrt.

Anhang A – LITERATURVERZEICHNIS

[1] P.S.C. Alencar, D.D. Cowan, C.J.P. Lucena. *A Logic Theory of Interfaces and Objects*, IEEE Transactions on Software Engineering Vol. 28, No. 6, Juni 2002

[2] H. Balzert. *Lehrbuch der Software-Technik –Software-Entwicklung–*, 2000 Spektrum Heidelberg, Berlin: Spektrum Akad. Verl.

[3] H. Balzert. *Lehrbuch der Software-Technik –Software-Managment, Software-Qualitätssicherung, Unternehmensmodellierung–*, 1998 Spektrum Heidelberg, Berlin: Spektrum Akad. Verl.

[4] K. Bothe. *Praxisnähe durch Reverse Engineering-Projekte: Erfahrungen und Verallgemeinerungen*, Februar 2001, 7. Workshop SEUH, Zürich

[5] M. El-Ramly, P. Iglinski, E. Stroulia, P. Sorenson, B. Matichuk. *Modeling the System-User Dialog Using Interaction Traces*, 8th Working Conference on Reverse Engineering, 2.-5. Oktober 2001, Stuttgart

[6] G. Fischer, A. Lemke, T. Schwab. *Knowledge-based Help Systems*, CHI '85 Proceedings, April 1985

[7] K. Fuchs-Kittowski, H. Parthey, W. Umstätter, R. Wagner-Döbler (Hrsg.). *Wissenschaftstheoretische Positionen in Bezug auf die Gestaltung von Software*, Organisationsinformatik und Digitale Bibliothek in der Wissenschaft: Wissenschaftsforschung Jahrbuch 2000. Berlin: Gesellschaft für Wissenschaftsforschung 2001

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*, 1997 Addison Wesley

[9] J. Hanisch, J. Letzel *Automatisierung von Regressionstests eines Programms zur Halbleiter-Strukturanalyse*, Diplomarbeit November 2002

[10] H.B. Kief. *NC / CNC-Handbuch*, 1999 Carl Hanser Verlag München, Wien

[11] S. Meyers. *Effektiv C++ programmieren: 50 Möglichkeiten zur Verbessung Ihrer Programme*, Addison-Wesley 1995

[12] E. Rubiko. *COM, DCOM Objektmodell und OLE Architektur: Eigenschaften, Schnittstellen, Konzepte*, 11.1999 (http://www-mmt.inf.tu-dresden.de/Lehre/Wintersemester_98_99/Hauptseminar/COM_DCOM.pdf)

[13] T. K. Sarma. *Applying Observer Pattern in C++ Applications*, Januar 2001

[14] C. Szallies. *On Using the Observer Design Pattern*, August 1997

[15] K. Tucker, R.E. K. Stirewalt. *Model Based User-Interface Reengineering*, 6th Working Conference on Reverse Engineering, 6.-8. Oktober 1999 Atlanta, Georgia, USA

[16] ISO/IEC 14882-1998. *Information Technology – Programming Languages – C++*

Anhang B – VERWENDETE MATERIALIEN

‚XCTL' – GESAMTSYSTEM

[M1] U. Sacklowski. *Der Anwendungsbereich: Eine Einführung*, Version 2.4

[M2] K. Schützler. *Use-Case-Diagramm XCtl-Gesamtsystem*, April 2001

[M3] S. Lützkendorf. *Toter Code: Ergebnisse mit SNiFF und McCabe*, erstellt: 15.06.1999

SUBSYSTEM: ‚ABLAUFSTEUERUNG'

[M4] T. Kullmann, G. Reinecker. *Reverse Engineering des Subsystems Ablaufsteuerung*, Version 1.9

[M5] U. Sacklowski. *Ablaufsteuerung – Fehler*, erstellt: 15.09.2002

[M6] T. Kullmann, G. Reinecker. *Reengineering des Subsystems Ablaufsteuerung*, Version 1.1

SUBSYSTEM: ‚BENUTZOBERFLÄCHE'

[M7] T. Kullmann, G. Reinecker. *Reverse Engineering der Basisklassen für die Benutzeroberfläche*, Version 1.2

[M8] U. Sacklowski. *Benutzeroberfläche – Fehler*, erstellt: 06.03.2002

[M9] T. Kullmann, G. Reinecker. *Reengineering der Basisklassen für die Benutzeroberfläche*, Version 1.0

[M10]T. Kullmann, G. Reinecker. *Layoutkonventionen und Steuerelemente*, Version 1.0

SUBSYSTEM: ‚DETEKTORNUTZUNG'

[M11]U. Sacklowski. *Detektornutzung – Fehler*, erstellt: 19.06.2000

[M12]J. Picard, R. Harder, A. Paschold. *Reverse Engineering des Subsystems Detektoren des RTK-Steuerprogramms*, November 2000

SUBSYSTEM: ‚MANUELLE JUSTAGE'

[M13]K. Bothe. *Verhaltensspezifikation ‚RTK-Steuerprogramm'*, Version 1.1

[M14]K. Bothe. *Verhaltensspezifikation (Pflichtenheft) XCTL-Steuerprogramm*, Version 2.2

[M15]T. Kullmann, G. Reinecker. *Pflichtenheft neue ‚Manuelle Justage'*, Version 2.1

[M16]T. Kullmann, G. Reinecker. *Bewertung der Neuentwürfe des Oberflächenfensters zur ‚Manuelle Justage'*, Version 1.7

[M17]T. Kullmann, G. Reinecker. *XCTL-Steuerprogramm, Benutzer-Leitfaden der Basisanwendung: neue ‚Manuelle Justage'*, Version 1.6

[M18]J. Hanisch. *Regressions-Testfälle ‚Manuelle Justag*e', Version 2.2

[M19]J. Wegener. *Systemtest „Probe und Kollimator justieren"; CTE-32-Diagramm(e)*, erstellt: 10.02.2000

[M20]T. Kullmann, G. Reinecker. *Regressions-Testfälle neue „Manuelle Justage*, Version 1.5

[M21]T. Kullmann, G. Reinecker. *Designbeschreibung neue ‚Manuelle Justage'*, Version 2.0

[M22]T. Kullmann, G. Reinecker. *neue ‚Manuelle Justage' – Funktionskomponente; CTE-XL-Diagramm(e)*, erstellt: 25.04.2002

[M23]T. Kullmann, G. Reinecker. n*eue ‚Manuelle Justage' – Nutzeroberfläche; CTE-XL-Diagramm(e)*, erstellt: 07.02.2002

[M24]T. Kullmann, G. Reinecker. *Reverse Engineering des ursprünglichen Subsystems ‚Manuelle Justage'*, Version 1.6

[M25]T. Kullmann, G. Reinecker. *Fehler - alte ‚Manuelle Justage'*, Version 1.0

SUBSYSTEM: ‚MOTORSTEUERUNG'

[M26]T. Kullmann, G. Reinecker. *Reverse Engineering der objektorientierten Teile des Subsystems Motorsteuerung*, Version 1.5

[M27]U. Sacklowki. *Motorsteuerung – Fehler*, erstellt: 19.06.2000

[M28]T. Kullmann, G. Reinecker. *Reengineering der objektorientierten Teile des Subsystems Motorsteuerung*, Version 1.1

SUBSYSTEM: ‚TOPOGRAPHIE'

[M29]M. Gollnick. *Software-Struktur*, Version 1.0

[M30]U. Sacklowski. *Topographie – Gesamtvorgang*, Version 2.2

[M31]M. Gollnick, U. Sacklowski. *Einstellen der Parameter für die Topographie*, Version 3.0

[M32]M. Gollnick, U. Sacklowski. *Start und Kontrolle der Topographie*, Version 3.0

[M33]U. Sacklowski. *Topographie-Meßprotokoll_Buch*, Version 1.0

[M34]U. Sacklowski. *Topographie – Fehler*, erstellt: 19.06.2000

[M35]T. Kullmann, G. Reinecker. *Designbeschreibung zum Reengineering der ‚Topographie'*, Version 1.4

[M36]J. Hanisch. *Regressions-Testfälle Topographie*, Version 2.3

[M37]T. Kullmann, G. Reinecker. n*eue ‚Topographie' – Funktionskomponente; CTE-XL-Diagramm(e)*, erstellt: 17.07.2002

[M38]T. Kullmann, G. Reinecker. n*eue ‚Topographie' – Nutzeroberfläche; CTE-XL-Diagramm(e)*, erstellt: 17.07.2002

SUBSYSTEM: WINDOWS-RESSOURCEN

[M39]T. Kullmann, G. Reinecker. *Ressourcenübersicht*, erstellt: 01.06.2002

Anhang C – STICHWORTVERZEICHNIS

A

Abstract Design Objects ... 50
Abstract Design View ... 50
Abstract Design Viewer ... 50
AC (Codemetrik ‚Attribute Complexity') ... 20
Accessor-Methode ... 38
Application Programming Interface ... 12
Artefakte ... 21
Ausgabeparameter ... 110

B

Basisanforderungen ... 18
benutzenden Subsysteme ... 104
Benutzer-Leitfaden (Benutzerhandbuch) ... 34
Beobachter (Observer-Muster) ... 51
Beziehungstypen ... 38

C

Classification Tree Editor ... 85
Computer Aided Software Engineering ... 18
Concrete Observer ... *Siehe* Konkreter Beobachter
Concrete Subject ... *Siehe* Konkretes Subjekt
CTE ... *Siehe* Classification Tree Editor
CVS ... 31

D

Dekomposition ... 2
Delegation ... 51
Dialog
  ‚Offset für <Antrieb>' ... 30
  ‚Offset für <PSD>' ... 30
  ‚Topographie-Durchführung' ... 101
  ‚Topographie-Einstellungen' ... 101
  neue ‚Manuelle Justage' ... 30
  Test der neuen ‚Manuellen Justage' ... 87
  ursprüngliche ‚Manuelle Justage' ... 18

F

F-Block ... 106
formale Assoziation ... 50
Forward Engeering ... 3
Funktionskomponente ... 2
  Ansatz von ... 104

G

Gliederung
  aufgabenorientierte ... 34
  produktorientierte ... 34
Graphical User Interface ... 9

H

hierarchische Softwarearchitektur (Komplexitätsbewältigung) **10**
Hilfe-System
  aktives **31**
  dynamisches **31**
  individuelles **31**
  passives **31**
  statisches **31**
  uniformes **31**
Human-Interface Prototyp .... *Siehe* Oberflächenprototyp

K

Konkreter Beobachter **52**
Konkretes Subjekt **52**

L

Lastenheft **18**
LOC (Codemetrik ‚Lines Of Code') **20**
logische Operationen **85**

M

Manuelle Justage **6**
  neue **17**
  ursprüngliche **17**
MNOP (Codemetrik ‚Maximum Number Of Parameter') **20**
Mutator-Methode **38**

N

nicht-zustandsverändernde Methoden **44**
NOA (Codemetrik ‚Number Of Attributes') **20**
NOC (Codemetrik ‚Number Of Classes') **20**
NOCON (Codemetrik ‚Number Of Constructors') **20**
NOO (Codemetrik ‚Number Of Operations') **20**
NOOM (Codemetrik ‚Number Of Overriden Methods') **20**

O

Oberflächenentwurf .... *Siehe* Oberflächenprototyp
Oberflächen-Prototyp **28**
objektorientierte Analyse **38**
objektorientierte Programmierung **9**
O-Block **106**
Observer .... *Siehe* Beobachter
Observer-Muster **51**
OOA .... *Siehe* objektorientierte Analyse

P

Pflichtenheft **21**
PPrivM (Codemetrik ‚Percentage of Private Members') **20**
PProtM (Codemetrik 'Percentage of Protected Members') **20**
PPubM (Codemetrik ‚Percentage of Public Members') **20**
Produkt-Definition **21**
Produkt-Modell **21**
Prototyping (Komplexitätsbewältigung) **14**
publish-subscribe (Observer-Muster) **51**

R

Referenz-Handbuch (Benutzerhandbuch) **34**
Referenzprodukt **18**
Repository **5**

S

Singleton-Muster **51**

Software-Ergonomie **29**
Subjects .... *Siehe* Subjekte
Subjekte (Observer-Muster) **51**
Subsysteme **21**

T

TCR (Codemetrik ‚True Comment Ratio') **20**
Testtreiber
  autarker **89**
Testverfahren
  Black-Box .... *Siehe* funktionaler Test
  funktionales **82**
  objektorientiert **82**
  Strukturtest **82**
  White-Box .... *Siehe* Strukturtest
Topographie **7**
  getrennte **98**
  ursprüngliche **98**
Trainingshandbuch (Benutzerhandbuch) **34**

V

Validation **11**
Verifikation **11**
views-a-Relation 50

Z

zustandsverändernde Anweisung **110**