RE-ENGINEERING DES SUBSYSTEMS ABLAUFSTEUERUNG

Dokument zur Diplomarbeit
- Designphase -

Autoren	Thomas Kullmann, Günther Reinecker
Dokumentversion	1.1
Zustand	abgeschlossen
letzte Bearbeitung	15.08.02

Inhalt

I Überblick

Im Mittelpunkt bei der Erarbeitung dieses Dokuments stand die Vollständigkeit. Jeder Typ, jede Klasse, jeder Member, u.ä. wird kurz und präzise erläutert. Dazu wurde dieses Dokument stark formalisiert, die Layoutkonventionen sind unter [7] zu finden. Weiterführende Informationen zum Subsystem Antriebssteuerung findet man unter [1] und [2]. In [3] kann zum Thema Detektoren nachgeschlagen werden.

Um das Dokument übersichtlicher zu gestalten, wurden die behandelten Attribute und Methode nach "Sinneinheiten" geordnet.

► Dokumentation des Istzustands und Änderungen am Subsystem

Die Dokumentation bezieht sich auf den Quellcode der hier aufgelisteten Dateien (Stand: 05.08.2002). Nachfolgende Änderungen oder Neuimplementierungen können nicht berücksichtigt werden.

h-Dateien (Deklaration)	cpp-Dateien (Implementation)
• M_STEERG.H	• M_STEERG.CPP
• WORKFLOW.H	

Tabelle 1 „Auflistung der zum Subsystem zugehörigen Dateien" (Quelle: selbst)

Im Gegensatz zu Version 1.9 werden hier auch die Änderungen am Subsystem dokumentiert und erklärt. Änderungen wurde rot (nicht fett und damit im Gegensatz zu Änderungswünschen, Fehlerquellen, etc.) hervorgehoben. Neue Member sind durch **NEU** und geänderte Signaturen sind durch **GEÄ** in der letzten Spalte (nach dem Zugriffsschutz, z.B. **PRIVATE**) markiert.

Die deutlich gestiegene LOC-Anzahl ist auf die neue ausführliche Kommentierung und die zahlreich eingefügten Leerzeilen, die die Lesbarkeit in großen Methoden verbessern sollen, zurückzuführen. Kaum Einfluss haben die neuen Member, weil sie fast ausschließlich aus dem globalen Namensraum in `TSteering` verschoben wurden. Der Fährnis halber wurde der auskommentierte, tote Code entfernt, um die Metriken nicht zu verfälschen.

Die Ablaufsteuerung wird über das `Steering`-Objekt, das in Datei M_STEERG.CPP deklariert ist, realisiert. Sie besteht aus einer Reihe von **Makros**[1], die ihrerseits jeweils aus einer "nahezu beliebige" Kombination von vordefinierten **Kommandos**[2] bestehen können. Diese können wiederum durch die Angabe von **Kommandoparametern** individualisiert werden und besitzen eine gewisse Einzelfunktionalität. Aus der Reihenfolge/ dem Ablauf der Kommandos ergibt sich für das Makro eine Gesamtfunktionalität, die beim Ausführen des Makros (und somit dem Ablauf der einzelnen Kommandos im Makro) durchgeführt wird. Die Verarbeitung des Makros kann zwischen den Kommandos und den **Einzelschritten**, aus dem die Kommandos bestehen, gestoppt und zu einem späteren Zeitpunkt wieder fortgesetzt werden. Die Makroverarbeitung kann jedoch auch abgebrochen werden, so dass ein Fortsetzen unmöglich ist.

`TSteering` enthält in `aMacroList` eine Liste der geladenen Makros. Derzeit ist ihre Größe (und damit auch die maximale Anzahl von Makros) statisch auf `20` begrenzt, kann aber unproblematisch erhöht werden (siehe Anmerkungen bei `TMacroTag`). Mit der Methode `Initialize` können Makros aus STANDARD.MAK und mit `LoadMacroByUser` aus SCAN.MAK geladen werden. Die Syntax der mak-Dateien hat Kay Schützler unter **[4]** zusammengestellt, gültige Parameterbelegungen der verwendeten Kommandos sind beim jeweiligen Kommando selbst, beim Konstruktor aufgezählt – hier wird auch deren Funktion erklärt. Nur bei sehr komplexen Kommandos, die aus mehreren Einzelschritten bestehen, war es vorteilhaft, die Beschreibung der Funktionalität und der Parameterbelegung direkt bei der Klasse zu erklären, nicht im Konstruktor.

Jedes Makro wird durch eine Struktur vom Typ `TMacroTag` repräsentiert. Entsprechend dem Namen, in der mak-Datei, wird jedem Makro ein eineindeutiger Typ (`TMacroId`) zugeordnet, auch dieser behindert das Hinzufügen von Makros, weil nur maximal ein Makro eines Typs in `aMacroList` vorhanden sein kann (bereits vorhandene Makros werden ggf. überschrieben). Neben Typ, dem Namen des Makros und der Datei, aus der das Makro geladen wurde, enthält `TMacroTag` auch eine dynamische Liste von `TCmdTag`-Objekten, die die Kommandos repräsentieren – sie kann nahezu beliebig erweitert werden. Die Liste der möglichen Kommandos ist in **Tabelle 4** dargestellt.

Während der Abarbeitung eines Makros wird für das aktuelle Kommando meist ein Objekt, das von `TCmd` abgeleitet ist, erstellt, das die gewünschte Funktionalität durchführt. Nur für nicht mehr verwendete Kommandos und Kontrollfluss-Kommandos, die die Abarbeitung des Makros beeinflussen[3], wird kein Objekt erstellt – ihre Funktionalität wurde direkt in `TSteering`, Methode `StartCmdExecution` implementiert, wo das nächste auszuführende Kommando ermittelt wird.

[1] eine Art vereinfachtes Programm

[2] ausgewählten Befehlen

[3] bedingte und unbedingte Sprünge, sowie `Stop`

II Typen

► **`typedef enum { … } TMacroId`**
ordnet jedem Makro einen eineindeutigen Typ zu – im Gegensatz zur "beliebigen", texturellen Bezeichnung der mak-Datei (Bezeichnungen sind identisch zu den verwendeten, symbolischen Werten)

"beliebige" Bezeichnung	`TMacroId`	Verwendung
"SetupTopography"	`SetupTopography`	**Einstellen des Arbeitspunkt** STANDARD.MAK: 4 Kommandos bei `TTopographyExecute` benutzt
"SearchReflection"	`SearchReflection`	**Peak-Suche** STANDARD.MAK: 5 Kommandos bei `TAdjustmentExecute` benutzt
"InquireHwb"	`InquireHwb`	**Bestimmen der Halbwertsbreite** STANDARD.MAK: 16 Kommandos bei `TAdjustmentExecute` und `TAngleControl` benutzt
"AzimutalJustify"	`AzimutalJustify`	**Makro zur Azimutalen Justage** STANDARD.MAK: 22 Kommandos bei `TAdjustmentExecute` benutzt
"Test"	`Test`	**Test** STANDARD.MAK: 4 Kommandos bei `TAdjustmentExecute` benutzt
"ScanJob"	`ScanJob`	**Standard zur automatischen Ausführung von Scan's** SCAN.MAK: 30 Kommandos **nie ausgeführt**
"AreaScanJob"	`AreaScanJob`	**Standard zur automatischen Ausführung von AreaScan's** SCAN.MAK: 10 Kommandos **nie ausgeführt**

Tabelle 2 „Beziehungen zwischen Bezeichnung und eindeutigem Makrotyp – Groß-/ Kleinschreibung ist zu beachten!" (Quelle: nach STANDARD.MAK und SCAN.MAK)

Der Typ wird aus der "beliebigen", texturellen Bezeichnung (die im jeweils bei [COMMON] –> Name in der mak-Datei steht) ermittelt. Die Bezeichnung wird bei Methode `TSteering::ParsingMacroId` in den Typ übersetzt. Eine Umkehrfunktion ist nicht erforderlich.

► **`typedef struct { … } TMacroTag`**
repräsentiert ein Makro; besteht aus dessen Typ, Namen, mak-Datei und der Kommandoliste

Attribut in `TCmdTag`		Verwendung
► **`TMacroId Id`**	**PUBLIC**	eineindeutiger Typ des Makros
► **`BOOL bIsReady`**	**PUBLIC**	dieses Flag kennzeichnet, das das Makro erfolgreich eingelesen wurde **wird nie auf `FALSE` gesetzt; Redundant, weil fehlerhafte Makros nicht in `TSteering::aMacroList` aufgenommen werden**
► **`char Name [ _MAXLENMACRONAME+1 ]`**	**PUBLIC**	"beliebige", texturelle Bezeichnung des Makros, so wie diese in Dialogfenstern anzuzeigen ist; Für die korrekte Initialisierung von `Id` ist es erforderlich, dass nur Werte aus Tabelle 2 verwendet werden
► **`char FileName [ _MAXLENMAKFILENAME+1 ]`**	**PUBLIC**	enthält Dateinamen und Erweiterung der mak-Datei, aus der das Makro geladen wurde (derzeit nur STANDARD.MAK oder SCAN.MAK)
► **`TCmdTag* CmdList`**	**PUBLIC**	Zeiger auf die Kommandoliste
► **`int Length`**	**PUBLIC**	Anzahl der Kommandos in `CmdList`

Tabelle 3 „Attribute von `TMacroTag`" (Quelle: selbst)

► **`typedef enum { … } TCmdId`**
ordnet jedem Kommando einen eineindeutigen Typ zu – im Gegensatz zur "beliebigen", texturellen Bezeichnung in der mak-Datei (Bezeichnungen sind identisch zu den verwendeten, symbolischen Werten)

"beliebige" Bezeichnung	`TCmdId`	Klasse für die gewünschte Funktionalität
"ChooseAxis"	`ChooseAxis`	`TChooseAxisCmd`
"SetWidth"	`SetWidth`	`TSetWidthCmd`
"LoadPoint"	`LoadPoint`	`TLoadPointCmd`
"MoveToPoint"	`MoveToPoint`	`TMoveToPointCmd`
"ChooseDetector", "ChooseDevice"	`ChooseDetector`	`TChooseDetectorCmd`
"GotoIntensity"	`GotoIntensity`	`TGotoIntensityCmd`
"GotoPeak "	`GotoPeak`	`TGotoPeakCmd`
"ShowValue"	`ShowValue`	`TShowValueCmd`
"Calculate"	`Calculate`	`TCalculateCmd`
"ControlFlank"	`ControlFlank`	`TControlFlankCmd`
"SetupScan"	`SetupScan`	`TSetupScanCmd`
"SetFileName"	`SetFileName`	`TSetFileNameCmd`
"SaveData"	`SaveData`	`TSaveDataCmd`
"Scan"	`Scan`	`TScanCmd`
"AreaScan"	`AreaScan`	`TAreaScanCmd`
"Inquire"	`Inquire`	bedingter Sprung bei `InquireResult == TRUE` wird das nächste (sonst das übernächste) Kommando im Makro ausgeführt[4]
"GotoLine"	`GotoLine`	unbedingter Sprung es wird das (als erster *Kommandoparameter* angegebene) Kommando ausgeführt
"Stop"	`Stop`	kennzeichnet das Ende einer Kommandofolge in einem Makro; ggf. wird dieses Kommando beim Einlesen des Makros ergänzt

Tabelle 4 „Beziehungen zwischen Bezeichnung, eindeutigem Kommandotyp und Benutzung – Groß-/ Kleinschreibung ist zu beachten!" (Quelle: selbst)

Die Methode `TSteering::ParsingCmd` ist für die Abbildung von *Bezeichnung* auf `TCmdId` zuständig. Eine Umkehrfunktion ist nicht erforderlich.

[4] Wenn dabei das letzte Kommando erreicht wird oder eine Position außerhalb der Kommandoliste angesprungen werden würde, ist die Makrobearbeitung beendet.

► **`typedef enum { … } TCParam`**
ordnet "ausgewählten" *Kommandoparametern* einen eineindeutigen Typ zu – im Gegensatz zur "beliebigen", texturellen Bezeichnung in der mak-Datei (Bezeichnungen sind identisch zu den verwendeten, symbolischen Werten); Solche Parameter diesen zur näheren Präzisierung des Kommandos selbst oder sie geben an, wie nachfolgenden Parameter (z.B. Positionsangaben) zu interpretieren sind.

"beliebige" Bezeichnung	`TCParam`		"beliebige" Bezeichnung	`TCParam`
"AbsorberUsed"	`AbsorberUsed`		"MaximizeCollimator"	`MaximizeCollimator`
"AreaScanResult"	`AreaScanResult`		"MaximizeGradient"	`MaximizeGradient`
"Argument"	`Argument`		"MaximizeTilt"	`MaximizeTilt`
"Array"	`Array`		"Middle"	`Middle`
"BackMove"	`BackMove`		"Min"	`Min`
"DecreaseWidth"	`DecreaseWidth`		"Opposite"	`Opposite`
"Difference"	`Difference`		"Peak"	`Peak`
"DynamicWidth"	`DynamicWidth`		"Reflection"	`Reflection`
"Equidistant"	`Equidistant`		"Relative"	`Relative`
"ForAreaScan"	`ForAreaScan`		"Result"	`Result`
"ForScan"	`ForScan`		"ScanResult"	`ScanResult`
"Hwb"	`Hwb`		"SmallSide"	`SmallSide`
"IncreasePeak"	`IncreasePeak`		"Standard"	`Standard`
"Interpolation"	`Interpolation`		"Start"	`Start`
"LargeSide"	`LargeSide`		"StaticStepWidth"	`StaticStepWidth`
"LastGoal"	`LastGoal`		"ThisDFPos"	`ThisDFPos`
"List"	`List`		"ToLargerAngle"	`ToLargerAngle`
"Max"	`Max`		"ToSmallerAngle"	`ToSmallerAngle`

Tabelle 5 „Beziehungen zwischen Bezeichnung und eindeutigem *Kommandoparameter*-Typ – Groß-/ Kleinschreibung ist zu beachten!" (Quelle: selbst)

Die Methode `TSteering::ParsingCmd` ist (unter Anderem) für die Abbildung von *Bezeichnung* auf `TCmdId` zuständig. Eine Umkehrfunktion ist nicht erforderlich.

► **`typedef struct { … } TCmdTag`**
repräsentiert ein einzelnes Kommando in einem Makro; besteht aus dem Typ des Kommandos und den Parametern; im folgenden **Kommandoinformation** genannt

Attribut in `TCmdTag`	Verwendung
► **`TCmdId Id`** **PUBLIC**	eineindeutiger Typ des Kommandos, entsprechend diesem Typ wird ein Objekt erstellt, das die gewünschte Funktionalität durchführt
► **`TCParam P1`** **PUBLIC**	erster *Kommandoparameter*
► **`TCParam P2`** **PUBLIC**	zweiter *Kommandoparameter*
► **`char P3[_MAXLENP3+1]`** **PUBLIC**	dritter *Kommandoparameter*, der für die Übergabe von verschiedenen Werten (int, float, LPSTR und Kombinationen daraus) individuell pro Kommando interpretiert wird

Tabelle 6 „Attribute von `TCmdTag`" (Quelle: selbst)

► **`typedef enum { … } TCCode`**
gibt den Fortschritt der Kommandoverarbeitung an

TCCode	Verwendung
`CFirstStep`	weiter bei Methode `FirstStep` des Kommandos
`CControlStep`	weiter bei Methode `ControlStep` des Kommandos
`CReadyStep`	weiter bei Methode `ReadyStep` des Kommandos
`CReady`	weiter bei Methode `Ready` (**gibt nur `CReady` zurück**) des Kommandos kennzeichnet Kommandoverarbeitung erfolgreich
`CMeasure`	signalisiert `Steering`, dass die Intensitätsmessung des aktuellen Detektors neu gestartet werden soll
`CRecall`	Kommandoausführung kurz unterbrechen (Methode `TSteering::StartTimer` aufrufen)
`CStop`	**wird nur an einer Stelle zurückgegeben, aber nie ausgewertet**

Tabelle 7 „symbolische Werte von `TCCode`“ (Quelle: selbst)

III Konstanten

► **`#define id_Report 10300`** **GLOBAL**
Ressourcen-ID eines Listenfeldes, in das `TSteering::SendReport` seine Informationen ausgeben kann, deklariert in RC_DEF.H

► **`const UINT nMaxArg= 4`** **GLOBAL**
maximale Anzahl von berechenbaren Zwischenergebnissen – siehe `TSteering::dCalcArg`; kann nicht `static` deklariert werden

► **`const UINT _MAXLENMAKFILENAME= 20`** **GLOBAL**
maximale Länge von `TMacroTag::Filename`

► **`const UINT _MAXLENMACRONAME= 20`** **GLOBAL**
maximale Länge von `TMacroTag::Name`

► **`const UINT _MAXLENP3= 50`** **GLOBAL**
maximale Länge von `TCmdTag::P3`

IV Klassen

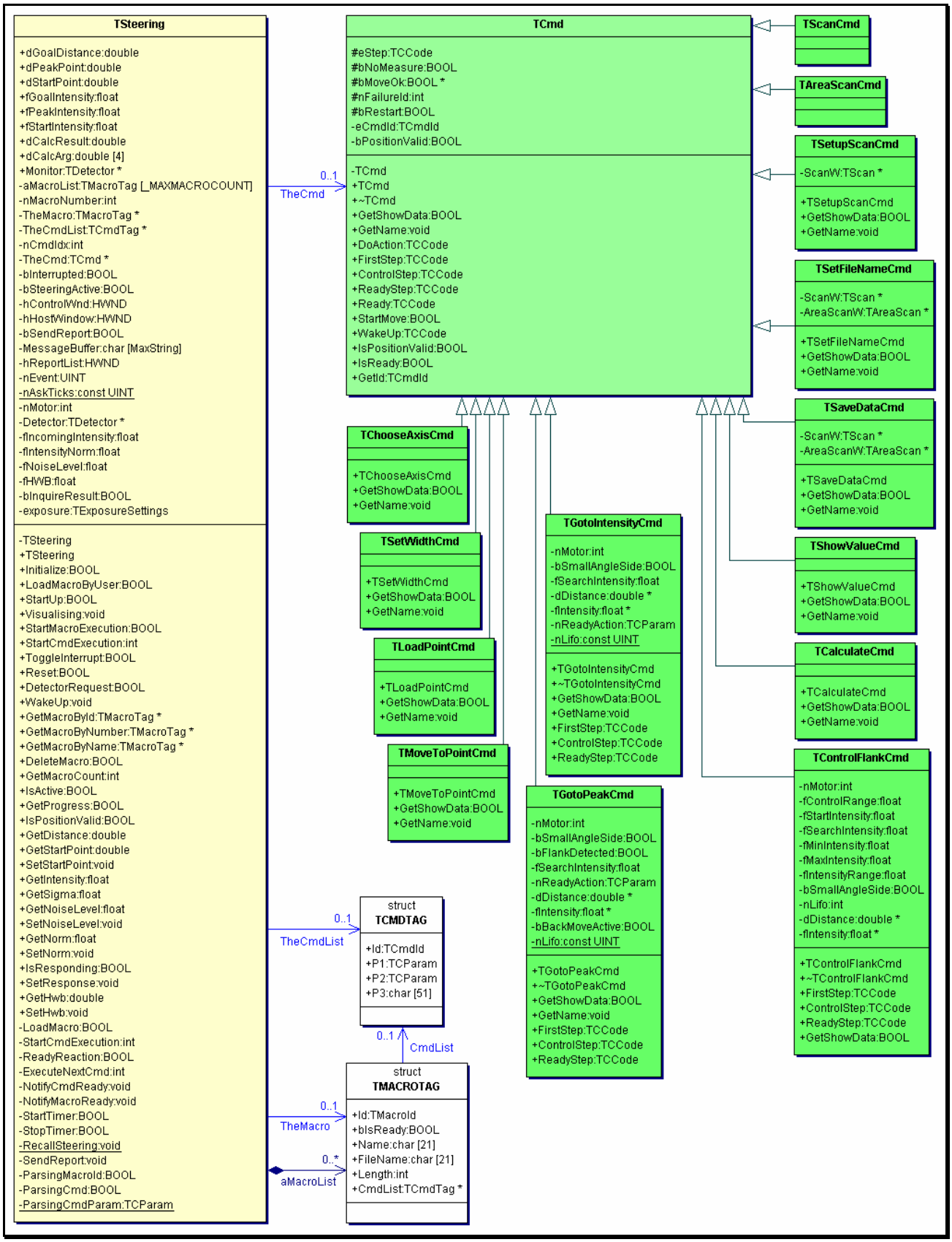

Abbildung 1 „UML-Klassendiagramm: Subsystem Ablaufsteuerung“ (Quelle: Together® ,Version 6.0)

IV.1 Klasse TSteering

Deklaration : WORKFLOW.H

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TSteerig` könnten erzeugt werden. Zur Benutzung der Ablaufsteuerung kann nur das `Steering`-Objekt verwendet werden, d.h. es brauchen keine eigenen Objekte von `TSteering` erstellt werden! Tut man dies doch, kommt es sogar zu Fehlern, weil jedes `TSteering`-Objekt mit dem Hauptfenster kommuniziert. Dieses sendet seine Nachrichten jedoch stets an das o.g. `Steering`-Objekt.

IV.1.1 Attribute

► **`TMacroTag aMacroList[_MAXMACROCOUNT]`** **PRIVATE**

Liste der (mit Methode `LoadMacro` geladenen) Makros; mit den Informationen über das Makro (Typ, etc.) und den auszuführenden Kommandos; Initialisierung nicht notwendig, da nur die Anzahl der Elemente (`nMacroNumber`) initialisiert werden muss

► **`int nMacroNumber`** **PRIVATE**

Anzahl der Makros in `aMacroList`, das letzte Makro befindet sich bei Index `aMacroNumber-1`; im Konstruktor mit `0` initialisiert

► **`TMacroTag* TheMacro`** **PRIVATE**

Zeiger auf das derzeit ausgeführte Makro; im Konstruktor mit `NULL` initialisiert

► **`TCmdTag* TheCmdList`** **PRIVATE**

Liste der Kommandos von `TheMacro`; vor der Makroausführung (Methode `StartMacroExecution`) initialisiert

► **`int nCmdIdx`** **PRIVATE**

Index in `TheCmdList` des derzeit ausgeführten Kommandos; Wertebereich `[0 … TheMacro::Length-1]`; im Konstruktor mit `-1` initialisiert

► **`TCmd* TheCmd`** **PRIVATE**

Entsprechend der Id des aktuellen Kommandos (`TheCmdList[ nCmdIndx ]->Id`) wird ein Objekt (abgeleitet von `TCmd`) erstellt, das die gewünschte Funktionalität des Kommandos durchführt. `TheCmd` ist ein Zeiger auf dieses Objekt. im Konstruktor mit `NULL` initialisiert.

► **`BOOL bSteeringActive`** **PRIVATE**

gibt an, ob derzeit ein Kommando ausgeführt wird; Bei `bInterrupted == TRUE`, ist `bSteeringActive == FALSE`, weil die Makroverarbeitung gestoppt wurde. im Konstruktor mit `FALSE` initialisiert

► **`BOOL bInterrupted`** **PRIVATE**

gibt an, dass die Makroverarbeitung gestoppt wurde; Stoppen und Fortsetzen der Makroausführung ist mit Methode `ToggleInterrupt` möglich. beim Start der Makroverarbeitung (Methode `StartCmdExecution`) mit `FALSE` initialisiert

► **BOOL bReset** **PRIVATE** **NEU**
gibt an, dass die Makroverarbeitung mit Methode `Reset` abgebrochen wurde; beim Start der Makroverarbeitung (Methode `StartCmdExecution`) mit `FALSE` initialisiert; bei Methode `Reset` auf `TRUE` gesetzt

► **HWND hControlWnd** **PRIVATE**
Handle auf das Dialogfenster, das ein Listenfeld für Informationsausgaben enthalten kann (siehe `hReportList`); Dieses Fenster empfängt die Botschaft `cm_CounterSet` vom (bei Methode `StartUp` gesetzten) Detektor. Daraufhin muss das Zählerfenster aktualisiert werden (Methode `TDetector::UpdateDisplay`) und `Steering` muss informiert werden, dass neue Messwerte vorliegen (`TSteering::DetectorRequest`) – **tut man dies nicht, hängt die Makroverarbeitung in einer Endlosschleife.**

► **HWND hHostWindow** **PRIVATE**
Handle zu dem Fenster, das über die Windowsbotschaft `cm_SteeringReady` benachrichtigt werden soll, wenn die Makroverarbeitung beendet ist

► **BOOL bSendReport** **PRIVATE**
bestimmt, ob bei Methode `SendReport` Informationen ausgegeben werden sollen (`TRUE`) oder nicht (`FALSE`); im Konstruktor mit `FALSE` initialisiert; bei Methode `Visualising` mit dem Wert des ersten Parameters überschrieben

► **char MessageBuffer[MaxString]** **PRIVATE**
Meldung die bei Methode `SendReport` – entweder in einem Listenfeld (siehe `hReportList`) oder in der Statuszeile des Hauptfensters ‚Steuerprogramm' – ausgegeben werden soll

► **HWND hReportList** **PRIVATE**
Handle zu einem Listenfeld, in das der Inhalt von `MessageBuffer` (bei Methode `SendReport`) ausgegeben werden soll; kann `NULL` sein wenn die Informationen in der Statuszeile des Hauptfensters ‚Steuerprogramm' ausgegeben werden sollen; im Konstruktor mit `NULL` initialisiert und bei Methode `Visualising` ggf. mit dem Handle des Steuerelements mit Ressourcen-ID `::id_Report` überschrieben

► **UINT nEvent** **PRIVATE**
Handle auf einen aktiven Timer in `TSteering` oder wenn `TTSteering`-Timer inaktiv ist `0`. Darf nicht `static` sein, sonst verwendet Methode `RecallSteering` ein anderes `nEvent` als `StartTimer` und `StopTimer` – **Compiler-/ Linkerfehler?!**

► **static const UINT nAskTicks= 30** **PRIVATE**
Diese Konstante gibt an, wie lange [in Millisekunden] die Kommandoverarbeitung dauern soll

► **int nMotor** **PRIVATE**
Index eines ausgewählten Antriebs in Objekt `lpMList`; nur in den von `TCmd` abgeleiteten Klassen benutzt, wird jeweils vor der Benutzung initialisiert

► **double dGoalDistance** **PUBLIC**
entspricht der Absolutposition des ausgewählten Antriebs (in **Nutzereinheiten**), die bei Kommando `TGotoIntensityCmd` oder `TGotoPeakCmd` errechnet wurde; nur in den von `TCmd` abgeleiteten Klassen benutzt

► **double dPeakPoint** **PUBLIC**

Mit `TLoadPointCmd` kann die aktuelle Absolutposition des gewählten Antriebs (in **Nutzereinheiten**) – unter Anderem in `dPeakPoint` – zwischengespeichert werden Mit anderen Kommandos können so gespeicherte Positionen auf dem Bildschirm ausgegeben oder wieder angefahren werden. nur in den von `TCmd` abgeleiteten Klassen benutzt

Dient derzeit nur zur Speicherung der Position des Intensitätsmaximums (siehe [5]: Abbildung 5).

► **double dStartPoint** **PUBLIC**

kann, wie `dPeakPoint`, dazu benutzt werden, die aktuelle Absolutposition zwischenzuspeichern; nur in den von `TCmd` abgeleiteten Klassen benutzt

► **TDevice* Device** **PRIVATE**

Zeiger auf den ausgewählten Detektor; bei Methode `StartUp` initialisiert und irgendwo anders geschrieben

► **TDetector* Monitor** **PUBLIC**

Verknüpfung auf einen Detektor für `TScanCmd`, `TAreaScanCmd` und `TGotoPeakCmd`

► **float fGoalIntensity** **PUBLIC**

entspricht der Intensität des ausgewählten Detektors am Zielpunkt der Bewegung; nur in den von `TCmd` abgeleiteten Klassen benutzt

► **float fPeakIntensity** **PUBLIC**

dient zur Speicherung des Intensitätsmaximums (siehe [5]: Abbildung 5) des ausgewählten Detektors; nur in den von `TCmd` abgeleiteten Klassen benutzt

► **float fStartIntensity** **PUBLIC**

wird nur bei Methode `TTopographyExecute::Dlg_OnCommand` gelesen, aber nie geschrieben

► **float fIncomingIntensity** **PRIVATE**

enthält die aktuelle Intensität[5] des ausgewählten Detektors

► **float fIntensityNorm** **PRIVATE**

ist ein Divisor, der zur Korrektur der aktuellen Intensität des ausgewählten Detektors (`fIncomingIntensity`) benutzt werden kann; wenn `fIntensityNorm != 0`, dann gilt `fIncomingIntensity := fIncomingIntensity/ fIntensityNorm`
(sonst bleibt `fIncomingIntensity` unverändert)

im Konstruktor mit `0` initialisiert

► **float fNoiseLevel** **PRIVATE**

definiert das Rausch-Level; Intensitäten unterhalb von `fNoiseLevel` werden bei `TGotoPeakCmd` nicht beachtet. im Konstruktor mit `300` initialisiert

[5] Einheit ist abhängig vom Detektor

► **double dCalcResult** **PUBLIC**

Beim Aufruf von Kommando `TCalculateCmd` wird `dCalcResult`, je nachdem welchen der Werte (`Difference`, `Opposite`, `Middle`) der Parameter annimmt, berechnet (Absolutpositionen in **Nutzereinheiten** des ausgewählten Antriebs). nur in den von `TCmd` abgeleiteten Klassen benutzt

► **double dCalcArg[nMaxArg]** **PUBLIC**

kann, wie `dPeakPoint`, dazu benutzt werden, die aktuelle Absolutposition zwischenzuspeichern; der zweite Parameter des `TLoadPointCmd`-Kommandos gibt an, an welchem Index die Position zu speichern ist; nur in den von `TCmd` abgeleiteten Klassen benutzt

► **float fHWB** **PRIVATE**

beim Aufruf von Methode `TCalculateCmd::TCalculateCmd(Hwb)` wird die Halbwertsbreite (siehe [5]: Abbildung 5) berechnet (in **Nutzereinheiten** des ausgewählten Antriebs); nur in den von `TCmd` abgeleiteten Klassen benutzt

► **BOOL bInquireResult** **PRIVATE**

nur ein Lesezugriff:

- Methode `ExecuteNextCmd`

► **TExposureSettings exposure** **PRIVATE**

speichert Detektorparameter vor Pausieren der Makroverarbeitung, um diese beim Fortsetzen wiederherstellen zu können

IV.1.2 Methoden

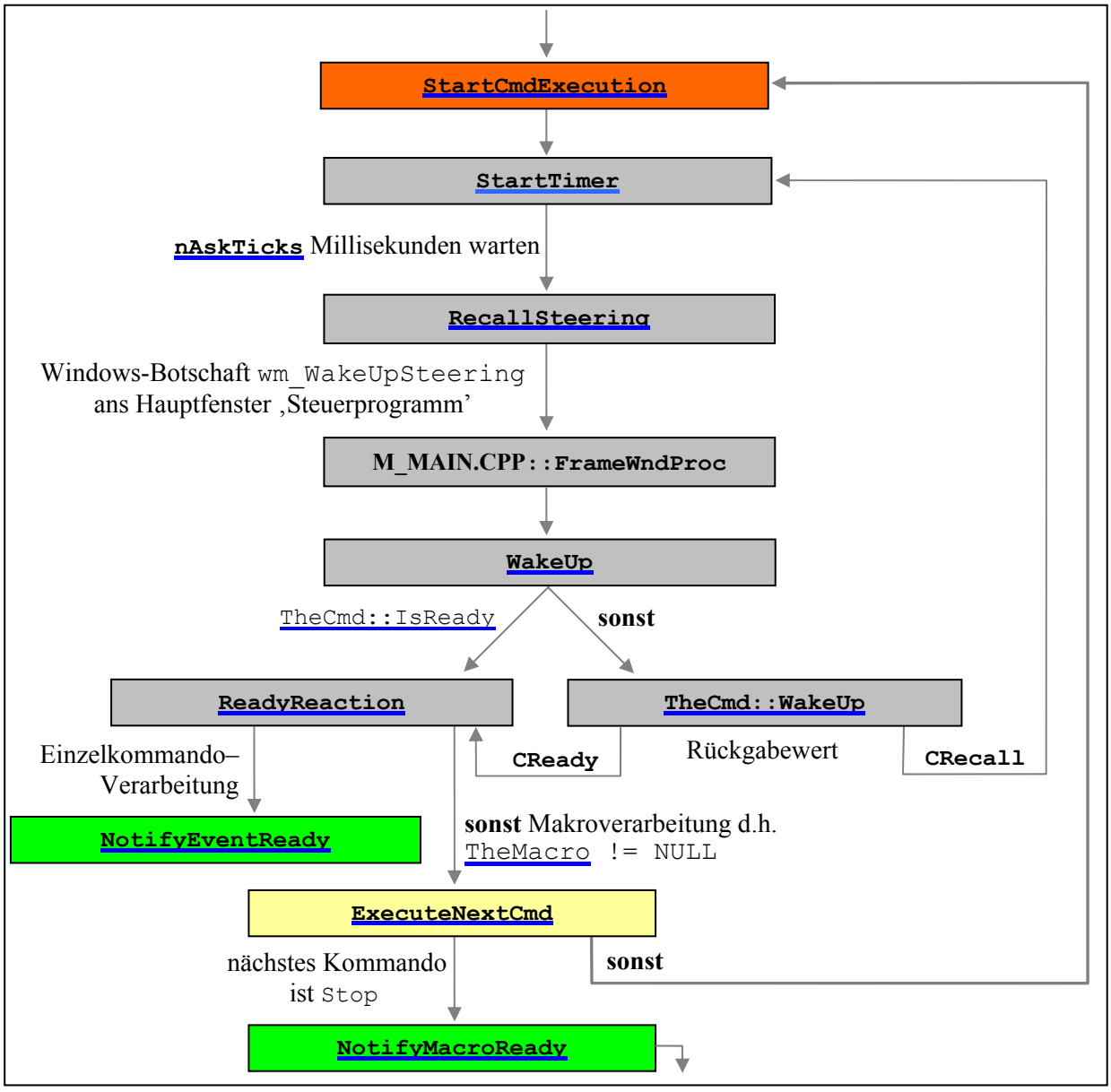

Abbildung 2 „Methodenaufrufe während der Einzelkommando- bzw. Makroverarbeitung" (Quelle: selbst)

Die grau hinterlegten Methoden ermöglichen eine Pausierung der Kommandoverarbeitung. Die blau unterstrichenen Member sind Teil des `Steering`-Objekts.

► **`TSteering( void )`** **PUBLIC**

Der Konstruktor initialisiert einen kleinen Teil der Attribute. Diese sind unter **IV.1.1 Attribute** besonders gekennzeichnet.

► **`TSteering( TSteering& )`** **PRIVATE**

leer implementiert, um das Kopieren von `TSteering`-Objekten zu verhindern

► **BOOL Initialize(const HWND, const int, const int)** **PUBLIC**

deaktiviert (durch den Aufruf von `Visualising(0, 0, 0, 0, 0, 0)`) Meldungsausgaben bei Methode `SendReport` und initialisiert die Makroverarbeitung, indem die drei Parameter an Methode `StartUp` weitergegeben werden. Anschließend werden die in **Tabelle 2** genannten Makros aus STANDARD.MAK geladen (Methode `LoadMacro`). Sobald das Laden eines Makros fehlschlägt, wird `FALSE` zurückgegeben, sonst `TRUE`.

Die aufgerufene Methode `Visualising` verwendet `hControlWnd`, obwohl dieses erst bei Methode `StartUp` initialisiert wird.

► **BOOL LoadMacroByUser(void)** **PUBLIC**

lädt die in **Tabelle 2** genannten Makros aus SCAN.MAK; Sobald das Laden eines Makros fehlschlägt, wird `FALSE` zurückgegeben, sonst `TRUE`.

► **BOOL StartUp(const HWND, const int, TDetector*)** **PUBLIC**

der erste Parameter wird `hControlWnd` zugewiesen; der zweite Parameter entspricht dem Index des Antriebs, der ausgewählt werden soll (siehe **[1]**: `mlSetAxis`); der dritte Parameter entspricht dem Index des auszuwählenden Detektors (siehe **[3]**: `TDList::DP(int)`)

Rückgabewert stets `TRUE`

► **void Visualising(BOOL, BOOL)** **PUBLIC**

Die Parameter regeln, wie die Meldungen bei Methode `SendReport` ausgegeben werden. Der erste Parameter gibt an, ob Meldungen ausgegeben werden sollen (wird `bSendReport` zugewiesen). Der zweite Parameter regelt, wo die Meldungen ausgegeben werden (siehe Methode `SendReport`):

- `TRUE` → `hReportList` wird mit dem Handle des Steuerelements mit Ressourcen-Id `id_Report` initialisiert, das sich im Fenster mit dem Handle `hControlWnd` befinden muss
- `FALSE` → wird `hReportList` mit `NULL` initialisiert

► **BOOL StartMacroExecution(TMacroTag*, HWND)** **PUBLIC**

prüft ob der erste Parameter verschieden von `NULL` ist, sonst wird `FALSE` zurückgegeben; Wenn bereits ein Makro ausgeführt wird, ist der Rückgabewert ebenfalls `FALSE`. Sonst wird der Detektor eingerichtet, das (als ersten Parameter) übergebene Makro wird ausgeführt, der zweite Parameter wird `hHostWindow` zugewiesen und `TRUE` zurückgegeben. siehe **Abbildung 2**

► **int StartCmdExecution(TCmdId, int, int, LPSTR, HWND)** **PUBLIC**

erzeugt ein neues Kommando: der erste Parameter ist der Typ (`TCmd::Id`), der zweite bis vierte Parameter sind der erste bis dritte *Kommandoparameter* (`TCmd::P1` bis `TCmd::P3`); der letzte Parameter wird `hControlWnd` zugewiesen; anschließend wird dieses Kommando ausgeführt und der Rückgabewert von `StartCmdExecution( TCmdTag )` zurückgegeben

Wertebereich vom zweiten und dritten Parameters sollte `TCParam` sein. Kann nicht korrigiert werden, weil an einigen Stellen `Omega2ThetaScan` (`typedef enum TxScanType`) angegeben wird (`TxScanType` und `TCParam` sind disjunkt)!

► **int StartCmdExecution(TCmdTag)** **PRIVATE**

Wenn das (als Parameter übergebene) Kommando NULL ist, wird FALSE zurückgegeben. sonst wird der Detektor eingerichtet und ein dem Kommandotyp entsprechendes Objekt erzeugt, das die gewünschte Funktionalität durchführt. Bei einem unbekannten Kommandotyp oder bei Fehlern während der Kommandoausführung wird FALSE zurückgegeben, sonst wird Methode StartTimer aufgerufen (um die Kommandoausführung für einige Augenblicke zu pausieren) und TRUE zurückgegeben. siehe **Abbildung 2**

► **int ExecuteNextCmd(void)** **PRIVATE**

Sobald das erste Kommando bei Methode StartMacroExecution erfolgreich durchgeführt und eine kleine Pause gemacht wurde (siehe **Abbildung 2**), wird das nächste Kommando ausgeführt (auch Sprünge werden unterstützt). Für das nächste Kommando wird wiederum Methode ExecuteNextCmd verwendet. Nachdem das nächste Kommando gestartet wurde, wird TRUE zurückgegeben; ist das Kommando vom Typ Stop, wird FALSE zurückgegeben.

► **BOOL StartTimer(BOOL)** **PRIVATE**

versucht einen Timer zu starten; Bei Erfolg enthält nEvent das Handle zum Timer, es wird TRUE zurückgegeben. Schlägt dies fehl, ist nEvent==0 und es wird FALSE zurückgegeben. Wenn der Timer aktiv ist, wird einmalig die Methode RecallSteering aufgerufen, danach stoppt der Timer – Parameter TIME_ONESHOT. siehe **Abbildung 2**

► **BOOL StopTimer(void)** **PRIVATE**

stoppt den Timer, falls dieser aktiv ist; setzt nEvent auf 0; Rückgabewert stets TRUE

► **static void CALLBACK RecallSteering (UINT, UINT, DWORD, DWORD, DWORD)** **PRIVATE**

wird beim Aufruf des Timers gerufen; setzt nEvent = 0 und kennzeichnet damit, dass der Timer nun inaktiv ist; Anschließend wird die Windows-Botschaft wm_WakeUpSteering ans Hauptfenster ‚Steuerprogramm' geschickt. Dort wird Methode Steering->WakeUp aufgerufen. siehe **Abbildung 2**

► **void WakeUp(void)** **PUBLIC**

siehe **Abbildung 2**; zusätzlich wird auch der Rückgabewert CMeasure von Methode TheCmd::WakeUp ausgewertet, dann wird die Detektormessung neu gestartet und Statusinformationen des bearbeiteten Kommandos werden ausgegeben (Methode SendReport)

► **BOOL DetectorRequest(void)** **PUBLIC**

ermittelt die aktuelle Intensität des ausgewählten Detektors und dividiert sie durch fIntensityNorm; das Ergebnis wird in fIncomingIntensity gespeichert. Anschließend wird die Funktionalität von Methode WakeUp durchgeführt. siehe auch hControlWnd

Rückgabewert ist stets TRUE

► **BOOL ReadyReaction(void)** **PRIVATE**

siehe **Abbildung 2**; zusätzlich werden Statusinformationen des bearbeiten Kommandos ausgegeben (Methode SendReport)

► **void NotifyMacroReady(void)** **PRIVATE**
stoppt den Detektor und benachrichtigt das Fenster mit dem Handle `hHostWindow` (über die Windowsbotschaft `cm_SteeringReady`), dass die Ausführung des Makros beendet ist; siehe **Abbildung 2**

► **void NotifyCmdReady(void)** **PRIVATE**
Quellcode ist völlig identisch zu Methode `NotifyMacroReady`; wird jedoch aufgerufen, um die Beendigung eines Einzelkommandos zu verkünden; siehe **Abbildung 2**

► **BOOL ToggleInterrupt(void)** **PUBLIC**
Bei `bInterrupted == FALSE` wird die Makroausführung gestoppt. Anschließend (`bInterrupted == TRUE`) kann die Ausführung ab diesem Punkt fortgesetzt werden, wenn Methode `ToggleInterrupt` erneut aufgerufen wird. gibt den neuen Wert von `bToggleInterrupt` zurück, d.h. Aktion erfolgreich?

► **BOOL Reset(void)** **PUBLIC**
bricht die Verarbeitung des aktuellen Makros ab; Makro kann später nicht fortgesetzt werden; Rückgabewert stets `TRUE`

► **TMacroTag* GetMacroById(TMacroId)** **PUBLIC**
sucht in `aMacroList` nach einen Makro mit dem (als Parameter übergeben) Typ; bei Erfolg wird ein Zeiger auf das gefundene Makro zurückgegeben, sonst `NULL`

► **TMacroTag* GetMacroByNumber(int)** **PUBLIC**
gibt einen Zeiger auf das Makro aus `aMacroList` zurück (den Index bestimmt der erste Parameter); Rückgabewert ist `NULL`, wenn der Parameter den Wertebereich `[0 … nMacroNumber-1]` verletzt

► **TMacroTag* GetMacroByName(LPSTR)** **PUBLIC**
wie Methode `GetMacroById`, hier wird jedoch ein Makro mit dem (als Parameter übergebenen) Namen gesucht

► **BOOL DeleteMacro(TMacroTag*)** **PUBLIC**
löscht das erste Makro aus `aMacroList`, das den Namen des (als Parameter übergebenen) Makros besitzt und verringert `nMacroNumber` um `1`; `FALSE` wird zurückgegeben, wenn der Parameter `NULL` ist, sonst `TRUE`

► **int GetMacroCount(void)** **PUBLIC**
gibt die Anzahl der Makros in `aMacroList` zurück (entspricht `nMacroNumber`)

► **BOOL IsActive(void)** **PUBLIC**
gibt zurück, ob die Makroverarbeitung aktiv ist (nicht pausiert, nicht abgebrochen, nicht fertig)

► **BOOL IsReset(void)** **PUBLIC**
gibt zurück, ob die Makroverarbeitung mit Methode `Reset` abgebrochen wurde

► **BOOL GetProgress (int&, LPSTR, int&)** **PUBLIC**
wenn die Makroverarbeitung nicht aktiv ist (`IsActive() == FALSE`), wird `FALSE` zurückgegeben; sonst wird im ersten Parameter der Index des aktuellen Schritts (`nCmdIdx`), im zweiten der Fortschritt des aktuellen Kommandos, im dritten die Anzahl der Schritt während der Makroverarbeitung zurückgeben, Rückgabewert ist `TRUE`

► **BOOL IsPositionValid(void)** **PUBLIC**
gibt an, ob die Positionierung des ausgewählten Antriebs erfolgreich war (entspricht `bPositionValid` des aktuellen Kommandos)

► **double GetDistance(void)** **PUBLIC**
gibt die aktuelle Absolutposition (in **Nutzereinheiten**) des ausgewählten Antriebs zurück

► **double GetStartPoint (void)** **PUBLIC**
gibt die `dStartPoint` zurück

► **void SetStartPoint (double)** **PUBLIC**
setzt `dStartPoint` auf den (als Parameter übergebenen) Wert

► **float GetIntensity(void)** **PUBLIC**
gibt die aktuelle Intensität[6] des ausgewählten Detektors zurück (entspricht `fIncomingIntensity`)

► **float GetSigma(void)** **PUBLIC**
gibt den Faktor `fSigma` des ausgewählten Detektors zurück (entspricht einem Wert, der aus der aktuellen Intensität berechnet wird, siehe Methode `TDevice::CalculateSigma`); Quelle: [3]

► **float GetNoiseLevel(void)** **PUBLIC**
gibt das Rausch-Level (`fNoiseLevel`) zurück

► **void SetNoiseLevel(float)** **PUBLIC**
setzt das Rausch-Level (`fNoiseLevel`) auf den (als Parameter übergebenen) Wert

► **float GetNorm(void)** **PUBLIC**
gibt den Divisor, der zur Korrektur der aktuellen Intensität verwendet werden kann zurück (entspricht `fIntensityNorm`)

► **void SetNorm(float)** **PUBLIC**
setzt den Divisor, der zur Korrektur der aktuellen Intensität verwendet werden kann zurück auf den (als Parameter übergebenen) Wert (entspricht `fIntensityNorm`)

► **BOOL IsResponding(void)** **PUBLIC**
gibt an, ob Methode `SendReport` Meldungen ausgeben soll (entspricht `bSendReport`)

---

[6] Einheit ist abhängig vom Detektor

► **void SetResponse(BOOL) PUBLIC**
weist den Parameter `bSendReport` zu (gibt an, ob Methode `SendReport` Meldungen ausgeben soll)

► **double GetHwb (void) PUBLIC**
gibt `fHwb` (bei `TCalculateCmd` berechnete Halbwertsbreite) zurück

► **void SetHwb (double) PUBLIC**
setzt `fHwb` auf den (als Parameter übergebenen) Wert

► **BOOL LoadMacro(LPSTR, LPSTR) PRIVATE**
lädt das Makro mit dem Namen des ersten Parameters aus der mak-Datei mit dem Namen des zweiten Parameters (der Dateiname darf keinen Pfad beinhalten und muss auf „.MAK“ enden; die Datei wird in dem Verzeichnis gesucht, wo die Anwendung gestartet wird). Bei Erfolg wird das Makro in `aMacroList` eingefügt – wenn bereits ein Makro dieses Typs (`TMacroId`) existiert, wird dieses gelöscht – anschließend wird `nMacroNumber` um eins erhöht.

Die Methode gibt `TRUE` zurück, wenn das Makro erfolgreich geladen wurde – `FALSE` in jeden erdenklichen Fehlerfall (Datei oder Makro nicht gefunden oder Syntaxfehler).

► **void SendReport(void) PRIVATE**
wenn `bSendReport == FALSE` oder `MessageBuffer == NULL` ist, wird keine Meldung ausgegeben; wenn `hReportList != NULL`, wird Meldung `MessageBuffer` am Ende des Listenfelds hinzugefügt, sonst wird die Meldung im Hauptfenster ‚Steuerprogramm’ ausgegeben

► **BOOL ParsingMacroId(TMacroTag&, LPSTR) PRIVATE**
übersetzt die, als Parameter übergebene, “beliebige“, texturelle Bezeichnung eines Makros in dessen Typ und speichert diese bei Erfolg im Makro (das als erster Parameter übergeben wird); Wenn ein Makro diesen Typs bereits in `aMacroList` vorhanden ist oder die Bezeichnung nicht übersetzt werden kann, wird `FALSE` zurückgegeben, sonst `TRUE`. siehe **Tabelle 2**

► **BOOL ParsingCmd(TCmdTag&, LPSTR, LPSTR, LPSTR, LPSTR) PRIVATE**
übersetzt die (als ersten Parameter übergebene) “beliebige“, texturelle Bezeichnung eines Kommandos in dessen Typ; die Parameter drei bis fünf werden (abhängig von diesem Typ) als *Kommandoparameter* interpretiert, die bei Erfolg (zusammen mit dem Typ) im ersten Parameter abgespeichert werden. Wenn die Bezeichnung nicht übersetzt werden konnte oder einer der Parameter fehlt oder ungültig ist, wird eine Meldung ausgegeben und `FALSE` zurückgegeben, sonst `TRUE`. siehe **Tabelle 4**

► **TCParam ParsingCmdParam(LPSTR param) PRIVATE**
übersetzt die (als ersten Parameter übergebene) “beliebige“, texturelle Bezeichnung von ausgewählten Parametern in dessen Typ und gibt ihn zurück (bei fehlerhaftem oder unbekanntem Parameter wird `0` zurückgegeben); siehe **Tabelle 5**

IV.1.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	**1.241**
	‚**LOC** of **I**mplementation'[7]	LOCI	998
	‚**LOC** of **D**eclaration'	LOCD	243
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	**32**
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	44
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	76
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	2
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	0
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	47
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	53
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	169
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	**5**
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	293
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	29

Tabelle 8 „ausgewählte Metriken der Klasse `TSteering`" (Quelle: Together® ‚Version 6.0)

[7] Diese Metriken sind nicht Bestandteil von Together, sondern wurden manuell ermittelt.

IV.2 Klasse TCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Obwohl es keine abstrakten Methoden gibt, werden nur Objekte von den von `TCmd` abgeleiteten Klassen (Spezialisierungen) erstellt.

IV.2.1 Friends

Klasse	Gründe für die Friend-Relation
`TSteering`	`nFailureId`, `bRestart`

Tabelle 9 „Friends der Klasse `TCmd`“ (Quelle: selbst)

IV.2.2 Attribute

► **`TCCode eStep`** **PROTECTED**

gibt an, welcher Schritt als nächstes (bei der Kommandoverarbeitung) ausgeführt werden soll; siehe **Tabelle 7**

► **`BOOL bNoMeasure`** **PROTECTED**

gibt an, dass keine Intensitätsmessung des ausgewählten Detektors stattfinden soll, wenn Methode `WakeUp` das nächste Mal aufgerufen wird; im Konstruktor mit `FALSE` initialisiert aber im Konstruktor fast aller abgeleiteten Klassen mit `TRUE` initialisiert

► **`BOOL* bMoveOk`** **PROTECTED**

Liste von `BOOL`-Werten, die Anzahl der Elemente entspricht der Anzahl der angeschlossenen Antriebe (siehe **[1]**: `mlGetAxisNumber`); Jedes Element gibt an, ob sich der Antrieb (der diesem Index entspricht) bewegt. Liste wird im Konstruktor dynamisch erzeugt und im Destruktor freigegeben

► **`int nFailureId`** **PROTECTED**

kennzeichnet verschiedene Fehlerzustände während der Kommandoverarbeitung – die Bedeutung variiert von Kommando zu Kommando (Die leeren Zellen geben an, dass der angegebene Wert bei dieser Klasse nicht verwendet wird.):

Wert von `nFailure`	0	1	11
`TSaveDataCmd`	es ist kein Fehler aufgetreten	speichern der Datendatei fehlgeschlagen	
`TChooseAxisCmd`		der als Parameter angegebene Antrieb konnte nicht ausgewählt werden	
`TChooseDeviceCmd`		der als Parameter angegebene Detektor konnte nicht ausgewählt werden	
`TGotoIntensityCmd`			die zu suchende Intensität (`fSearchIntensity`) ist (wider erwartend) `0`
sonst			

Tabelle 10 „Bedeutung der Werte in den Klassen, wo `nFailure` benutzt wird“ (Quelle: selbst)

die Makroverarbeitung in Methode `TSteering::StartCmdExecution` wird abgebrochen, wenn `nFailure` auf einen von `0` verschiedenen Wert setzt; im Konstruktor mit `0` initialisiert

► **BOOL bRestart** **PROTECTED**
gibt an, dass die Makroverarbeitung mit Methode `TSteering::ToggleInterrupt` unterbrochen und soeben wieder fortgesetzt wurde; wird nur bei `TScanCmd` und `TAreaScanCmd` verwendet, dort im Konstruktor jeweils mit `FALSE` initialisiert

► **TCmdId eCmdId** **PRIVATE**
speichert den Kommando-Typ; im Konstruktor mit dem Typ der (als Parameter übergebenen) *Kommandoinformationen* initialisiert

► **BOOL bPositionValid** **PRIVATE**
gibt an, ob die Positionierung des ausgewählten Antriebs erfolgreich war; im Konstruktor mit `TRUE` initialisiert

IV.2.3 Methoden

► **TCmd(TCmdTag)** **PUBLIC**
im Konstruktor wird `bMoveOk` dynamisch erzeugt und ein Großteil der unter **IV.2.2 Attribute** initialisiert
nun werden alle initialisiert

► **TCmd(TCmd&)** **PRIVATE**
leer implementiert, um das Kopieren von `TCmd` (und abgeleiteten) Objekten zu verhindern

► **virtual ~TCmd(void)** **PUBLIC**
gibt die dynamisch erzeugte Liste `bMoveOk` wieder frei

► **TCCode DoAction(void)** **PUBLIC**
setzt die Verarbeitung des Kommandos im nächsten Schritt fort – je nach `eStep` (`CFirstStep`, `CControlStep`, `CReadyStep` bzw. `CReady`); Der Rückgabewert einer aufgerufenen Methode (`FirstStep`, `ControlStep`, `ReadyStep` bzw. `Ready`) wird zurückgegeben.

► **virtual TCCode FirstStep(void)** **PUBLIC**
► **virtual TCCode ControlStep(void)** **PUBLIC**
► **virtual TCCode ReadyStep(void)** **PUBLIC**
► **virtual TCCode Ready(void)** **PUBLIC**
geben hier alle nur `CReady` (Kommandoverarbeitung erfolgreich beendet) zurück
Die abgeleiteten Klassen können hier den nächsten Schritt der Kommandoverarbeitung implementieren und zurückgeben welches der nächste, zu verarbeitenden Schritt ist.

► **virtual BOOL GetShowData(LPSTR)** **PUBLIC**
gibt hier nur `TRUE` zurück
In den abgeleiteten Klassen überschrieben, dort wird – je nach Fehlerzustand der in `nFailureId` abgelegt ist – eine Fehlermeldung im ersten Parameter platziert und stets TRUE zurückgegeben. Wird nur von `TSteering` aufgerufen, um den Grund für die fehlgeschlagene Kommandoausführung oder Statusinformationen ausgeben zu können.
In allen abgeleiteten Klassen wurden Statusinformationen in Deutsch bzw. Englisch (abhängig von `GermanVersion`) eingefügt oder ergänzt.

► **`virtual void GetName( LPSTR aName ) Kull-`** **PUBLIC**

gibt im ersten Parameter den Namen der Klasse zurück, hier nur „BASE“

In den abgeleiteten Klassen überschrieben, um dort den jeweiligen Namen zurückzugeben; ignoriert wird das führende „T“ und das anschließende „Cmd“; bei `TChooseAxisCmd` wird demnach „ChooseAxis“ zurückgegeben.

► **`BOOL StartMove( const int, double )`** **PUBLIC**

bewegt den Antrieb mit dem Index der als erster Parameter angegeben wird zur Absolutposition (in **Nutzereinheiten**) im zweiten Parameter; nach erfolgreicher Positionierung wird `TRUE` – sonst `FALSE` – zurückgegeben

► **`virtual TCCode WakeUp( void )`** **PUBLIC**

wird aufgerufen, nachdem die Kommandoausführung pausiert wurde; prüft, ob die durch das Kommando gestarteten Antriebe mittlerweile still stehen – ist dies nicht der Fall, wird `CRecall` zurückgegeben; Wenn die Antriebe stehen wird entweder der Rückgabewert von Methode `DoAction` (wenn `bNoMeasure == TRUE`) oder `CMeasure` zurückgegeben. siehe **Abbildung 2**

► **`BOOL IsPositionValid( void )`** **PUBLIC**

gibt `bPositionValid` zurück; gibt an, ob die Positionierung des ausgewählten Antriebs erfolgreich war

► **`BOOL IsReady( void )`** **PUBLIC**

gibt zurück, ob `eStep == CReady` ist (Kommandoverarbeitung erfolgreich beendet)

► **`TCmdId GetId( void )`** **PUBLIC**

gibt `eStep` (was ist der nächste Schritt der der Kommandoverarbeitung) zurück

IV.2.4 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	157
	‚**LOC** of **I**mplementation'	LOCI	96
	‚**LOC** of **D**eclaration'	LOCD	61
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	7
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	12
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	19
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	2
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	0
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	14
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	23
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	64
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	55
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	2
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	46
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	65

Tabelle 11 „ausgewählte Metriken der Klasse `TCmd`" (Quelle: Together® ‚Version 6.0)

IV.3 Klasse TChooseAxisCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TChooseAxisCmd` können erzeugt werden.

IV.3.1 Methoden

► **`TChooseAxisCmd( TCmdTag )`** **PUBLIC**

wählt die Antriebsachse (die als erster *Kommandoparameter* in den -*informationen* übergebenen wird) aus (gültige Werte: `TAxisType`); Bei Misserfolg wird `nFailureId 1` gesetzt. Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Fehlerinformationen (`nFailureId != 0`) oder Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.3.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	39
	‚**LOC** of **I**mplementation'	LOCI	29
	‚**LOC** of **D**eclaration'	LOCD	10
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	7
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	18

Tabelle 12 „ausgewählte Metriken der Klasse `TChooseAxisCmd`" (Quelle: Together® ‚Version 6.0)

IV.4 Klasse TSetWidthCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TSetWidthCmd` können erzeugt werden.

IV.4.1 Methoden

► **`TSetWidthCmd( TCmdTag )`** **PUBLIC**

setzt die Schrittweite (die als dritter *Kommandoparameter* in den -*informationen* übergebenen wird) für den ausgewählten Antrieb (gültige Werte: `float`); Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.4.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	31
	‚**LOC** of **I**mplementation'	LOCI	21
	‚**LOC** of **D**eclaration'	LOCD	10
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	7
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	23

Tabelle 13 „ausgewählte Metriken der Klasse `TSetWidthCmd`" (Quelle: Together® ‚Version 6.0)

IV.5 Klasse TLoadPointCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TLoadPointCmd` können erzeugt werden.

IV.5.1 Methoden

► **`TLoadPointCmd( TCmdTag )`** **PUBLIC**

Abhängig vom ersten *Kommandoparameter* in den, als Parameter übergebenen, *-informationen* wird `TSteering::dGoalDistance` in einem anderen Attribut von `TSteering` gespeichert.

erster Kommandoparameter	`TSteering::dGoalDistance` gespeichert in
`Argument`	`TSteering::dCalcArg[`<zweiter *Kommandoparameter*>`]`
`Start`	`TSteering::dStartPoint`
`Peak`	`TSteering::dPeakPoint`

Tabelle 14 „Möglichkeiten bei der Arbeitsweise von `TLoadPointCmd`" (Quelle: selbst)

Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**
überschrieben um Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.5.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	43
	‚**LOC** of **I**mplementation'	LOCI	33
	‚**LOC** of **D**eclaration'	LOCD	10
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	9
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	21

Tabelle 15 „ausgewählte Metriken der Klasse `TLoadPointCmd`" (Quelle: Together® ‚Version 6.0)

IV.6 Klasse TMoveToPointCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TMoveToPointCmd` können erzeugt werden.

IV.6.1 Methoden

► **`TMoveToPointCmd( TCmdTag )`** **PUBLIC**
Abhängig vom ersten *Kommandoparameter* in den, als Parameter übergebenen, *-informationen* wird eine Absolutposition des ausgewählten Antriebs angefahren.

erster Kommandoparameter	anzufahrende Absolutposition	
`Peak`	`TSteering::dPeakPoint`	siehe `TLoadPointCmd`
`Start`	`TSteering::dStartPoint`	
`LastGoal`	`TSteering::dGoalDistance`	spart `TLoadPointCmd` ein
`Result`	`TSteering::dCalcResult`	siehe `TCalculateCmd`
`Relative`	<aktuelle Antriebsposition> + <dritter *Kommandoparameter*>	
sonst	<dritter *Kommandoparameter*>	

Tabelle 16 „Möglichkeiten bei der Positionierung von `TMoveToPointCmd`“ (Quelle: selbst)

Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

► **`virtual BOOL GetShowData( LPSTR )`** **`PUBLIC`**

überschrieben um Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.6.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode’ inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	62
	‚**LOC** of **I**mplementation’	LOCI	52
	‚**LOC** of **D**eclaration’	LOCD	10
	‚**N**umber **O**f **A**ttributes’	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations’	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers’ Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors’	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods’	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers’	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers’	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers’	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass’	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity’	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters’	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity’	CC	
Kommentare			
	‚**N**umber **O**f **C**omments’	NOC	12
	‚**T**rue **C**omment **R**atio’	TCR (5, 400)	19

Tabelle 17 „ausgewählte Metriken der Klasse `TMoveToPointCmd`“ (Quelle: Together® ‚Version 6.0)

IV.7 Klasse TChooseDetectorCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TChooseDeviceCmd` können erzeugt werden.

IV.7.1 Methoden

► **`TChooseDetectorCmd( TCmdTag )`** **PUBLIC**

wählt den Detektor (der als erster *Kommandoparameter* in den -*informationen* übergebenen wird) aus (gültige Werte: Index in `lpDList`) – bei Misserfolg wird `nFailureId 1` gesetzt. Dann werden `fExposureTime`, `dwExposureCounts` und `fFailure` des ausgewählten Detektors gesetzt. Die Werte sind (durch je ein Leerzeichen voneinander getrennt, also `<float> <DWORD> <float>`) im dritten *Kommandoparameter* gespeichert. Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Fehlerinformationen (`nFailureId != 0`) oder Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.7.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	52
	‚**LOC** of **I**mplementation'	LOCI	42
	‚**LOC** of **D**eclaration'	LOCD	10
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	8
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	16

Tabelle 18 „ausgewählte Metriken der Klasse `TChooseDetectorCmd`" (Quelle: Together® ‚Version 6.0)

IV.8 Klasse TGotoIntensityCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TGotoIntensityCmd` können erzeugt werden.

Dieses Kommando sucht eine Intensität[8] des ausgewählten Detektors, die durch Bewegung des aktuellen Antriebs angefahren wird. Der erste *Kommandoparameter* regelt ob der Antrieb bevorzugt rückwärts (`SmallSide`) oder vorwärts (sonst) bewegt werden soll, solange die gesuchte Intensität nicht erreicht ist. Der zweite *Kommandoparameter* bestimmt wie die Antriebsposition zu berechnen und anzufahren ist, wenn an der gesuchten Intensität vorbeigefahren wurde (gültige Werte siehe `nReadyAction`).

IV.8.1 Attribute

► **`int nMotor`** **PRIVATE**

Index des, zu Beginn des Kommandos, ausgewählten Antriebs (siehe **[1]**: `mlGetAxis)`; im Konstruktor initialisiert

► **`BOOL bSmallAngleSide`** **PRIVATE**

gibt an, ob der erste *Kommandoparameter* `SmallSide` entspricht; bei `TRUE` wird der Antrieb in Methode `FirstStep` rückwärts (sonst vorwärts) bewegt; im Konstruktor initialisiert

Inkonsistenz: wird komplementär zu `TGotoPeakCmd` verwendet

► **`float fSearchIntensity`** **PRIVATE**

entspricht der anzusteuernden Intensität; im Konstruktor als Produkt aus `TSteering::dPeakIntensity` und dem dritten *Kommandoparameter* berechnet

► **`static const UINT nLifo= 3`** **PRIVATE**

bestimmt wie viele Elemente maximal in `dDistance` und `fIntensity` enthalten sein sollen; im Konstruktor mit `3` initialisiert (`2` würde völlig ausreichen)

► **`double* dDistance`** **PRIVATE**

ist eine im Konstruktor dynamisch erzeugte Liste mit `nLifo` Elementen; Sie dient als „History“ der zuletzt angefahrenen Absolutposition des ausgewählten Antriebs. bei Methode `ControlStep` wird der aktuelle Wert (`dDistance[0]`) mit dem letzten Wert verglichen (`dDistance[1]`)

► **`float* fIntensity`** **PRIVATE**

wie `dDistance`, jedoch für die Intensitätswerte des verwendeten Detektors

► **`TCParam nReadyAction`** **PRIVATE**

bestimmt, wie die endgültige Absolutposition des Antriebs in Methode `ControlStep` zu berechnen ist; im Konstruktor mit dem Wert des zweiten *Kommandoparameters* initialisiert; gültige Werte

- `BackMove` (anfahren der zuletzt angefahrenen Absolutposition) und
- `Interpolation` (Interpolation zwischen aktuellen und letzten Absolutposition und Intensitätswerten)

[8] Vielfaches aus Peak-Intensität und drittem Kommandoparameter (darf nur `float`-Werte kleiner gleich `1` annehmen, sonst kommt es zur Endlosschleife)

IV.8.2 Methoden

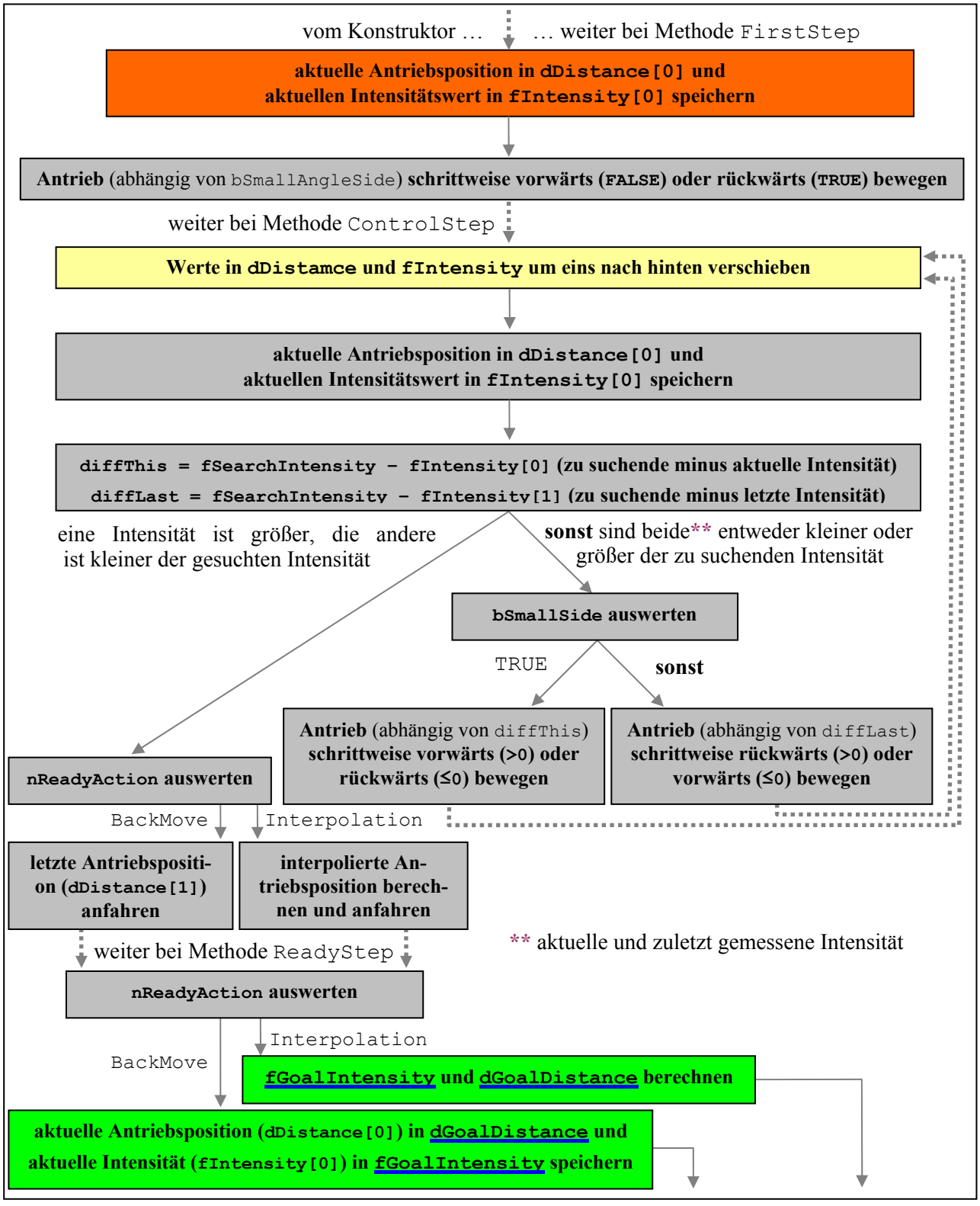

Abbildung 3 „Schematische Darstellung der Abläufe bei `TGotoIntensityCmd`“ (Quelle: selbst)
blau unterstrichen sind Attribute des `Steering`-Objekts; punktierte Pfeile bedeuten Kommando-Pausierung

► **`TGotoIntensityCmd( TCmdTag )`** **PUBLIC**

erzeugt `dDistance` und `fIntensity` dynamisch und initialisiert jedes, der unter **IV.8.1 Attribute** genannten, Attribute

► **`~TGotoIntensityCmd( void )`** **PUBLIC**

gibt die dynamisch erzeugten Listen `dDistance` und `fIntensity` wieder frei

► **`virtual TCCode FirstStep( void )`** **PUBLIC**

aktuelle Absolutposition und Intensitätswert in `dDistance[0]` und `fIntensity[0]` speichern; abhängig von `bSmallAngleSide` wird der Antrieb entweder um einen Schritt rückwärts (`TRUE`) oder vorwärts (`FALSE`) bewegt. nach einer Pausierung der Kommandoverarbeitung wird in Methode `ControlStep` fortgesetzt

► **`virtual TCCode ControlStep( void )`** **PUBLIC**

Wert in `dDistance` und `fIntensity` um eins nach hinten verschieben („History“) und die aktuelle Absolutposition und Intensitätswert in `dDistance[0]` und `fIntensity[0]` speichern;
`diffThis = fSearchIntensity - fIntensity[0]` und
`diffLast = fSearchIntensity - fIntensity[1]`

- `diffThis · diffLast < 0` (eine Intensität ist größer, die andere kleiner der gesuchten Intensität)
  In Abhängigkeit von `nReadyAction` wird eine Antriebsposition interpoliert und angefahren (`Interpolation`) oder die zuletzt verwendete angefahren (`BackMove`) berechnet und angefahren. nach einer Pausierung der Kommandoverarbeitung wird in Methode `ReadyStep` fortgesetzt.
  Sollte hier nicht auch der Fall `diffThis · diffLast == 0` (aktuelle oder letzte Intensität stimmt mit der gesuchten Intensität überein) mit eingeschlossen werden?
- sonst (entweder sind beide[9] kleiner oder größer der gesuchten Intensität)
  In beiden Fällen wird nach der Pausierung der Kommandoverarbeitung wieder Methode `ControlStep` aufgerufen.
  - `bSmallAngleSide == TRUE`
    wenn `diffThis > 0` (aktuelle ist kleiner der gesuchten Intensität), dann Antrieb einen Schritt vorwärts (sonst rückwärts bewegt)
  - sonst
    wenn `diffLast > 0` (letzte ist kleiner der gesuchten Intensität), dann Antrieb einen Schritt rückwärts (sonst vorwärts bewegen)
    Warum wird hier nicht `diffThis` ausgewertet? Diese Abfrage ist im Quelltext als vermutliche Fehlerquelle gekennzeichnet.

In jedem Fall wird `CRecall` zurückgegebenen

► **`virtual TCCode ReadyStep( void )`** **PUBLIC**

wenn `nReadyAction == Interpolation` ist, wird `TSteering::dGoalDistance` und `TSteering::fGoalIntensity` berechnet; bei `BackMove` wird die aktuelle Absolutposition in `TSteering::dGoalDistance` und die aktuelle Intensität in `TSteering::fGoalIntensity` gespeichert; abschließend kennzeichnet `eStep = CReady`, dass die Kommandoverarbeitung erfolgreich beendet wurde; gibt `CReady` zurück

[9] die aktuelle und letzte Intensität

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**
überschrieben um Fehlerinformationen (`nFailureId != 0`) oder Statusinformationen zurückzugeben:

eStep	Rückgabe im ersten Parameter
`CReady`	"GotoIntensity Ready"
`CFirstStep`	"Ziel-Intensität <i>" (wobei <i> = `fSearchIntensity` ist)
`CControlStep`	"D:<d> I:<i>" (wobei <d> = `dDistance[0]` und <i> = `fIntensity[0]` ist)
sonst	`0`

Tabelle 19 „Möglichkeiten der Textrückgabe bei `TGotoIntensityCmd::GetShowData`“ (Quelle: selbst)
Rückgabewert stets `TRUE`

IV.8.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode’ inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	70
	‚**LOC** of **I**mplementation’	LOCI	35
	‚**LOC** of **D**eclaration’	LOCD	35
	‚**N**umber **O**f **A**ttributes’	NOA (0, 30)	7
	‚**N**umber **O**f **O**perations’	NOO (0, 50)	5
	‚**N**umber **O**f **M**embers’ Attribute + Methoden	NOM = NOA + NOO	12
	‚**N**umber **O**f **C**onstructors’	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods’	NOOM (0, 10)	5
	‚**P**ercentage of **Priv**ate **M**embers’	PPrivM	50
	‚**P**ercentage of **Prot**ected **M**embers’	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers’	PPubM	50
	‚**W**eighted **M**ethods **P**er **C**lass’	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity’	AC	48
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters’	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity’	CC	
Kommentare			
	‚**N**umber **O**f **C**omments’	NOC	47
	‚**T**rue **C**omment **R**atio’	TCR (5, 400)	29

Tabelle 20 „ausgewählte Metriken der Klasse `TGotoIntensityCmd`“ (Quelle: Together® ‚Version 6.0)

IV.9 Klasse TGotoPeakCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TGotoPeakCmd` können erzeugt werden.

IV.9.1 Attribute

► **int nMotor** **PRIVATE**
Index des ausgewählten Antriebs (siehe [1]: `mlGetAxis`); im Konstruktor mit aktuell ausgewähltem Antrieb initialisiert

► **BOOL bSmallAngleSide** **PRIVATE**
gibt an, ob der erste *Kommandoparameter* `SmallSide` entspricht; bei `TRUE` wird der Antrieb in Methode `FirstStep` vorwärts (sonst nach rückwärts) bewegt; im Konstruktor initialisiert

► **BOOL bFlankDetected** **PRIVATE**
gibt an, dass eine Antriebsposition mit "hinreichender Genauigkeit zum Peak" erreicht wurde;Iist dies der Fall, wird die Bewegungsrichtung geändert und der ausgewählte Antrieb wird auf den Peak eingependelt. im Konstruktor mit `FALSE` initialisiert

► **TCParam nReadyAction** **PRIVATE**
bestimmt, wie `TSteering::dGoalDistance`, `TSteering::dPeakPoint` und `TSteering::fPeakIntensity` in Methode `ReadyStep` zu berechnen sind (`nReadyAction == Interpolation`) bzw. dass der Antrieb die letzte Position wieder anfahren soll (`BackMove`)

► **static UINT nLifo= 3** **PRIVATE**
bestimmt wie viele Elemente in `dDistance` und `fIntensity` maximal enthalten sein sollen; im Konstruktor mit `3` initialisiert

► **double* dDistance** **PRIVATE**
ist eine im Konstruktor dynamisch erzeugte Liste mit `nLifo` Elementen; Sie dient als „History" der zuletzt angefahrenen Absolutposition des ausgewählten Antriebs.

► **float* fIntensity** **PRIVATE**
wie `dDistance`, jedoch für die Intensitätswerte des verwendeten Detektors

► **BOOL bBackMoveActive** **PRIVATE**
gibt an, dass beim nächsten Aufruf von Methode `ReadyStep` Berechnungen durchzuführen sind, danach ist die Kommandoverarbeitung erfolgreich beendet
nicht initialisiert, wenn `bBackMove` uninitialisiert `TRUE` annimmt, wird trotz `nReadyAction == Interpolation` die Berechnung immer für `BackMove` durchgeführt, der Antrieb wurde aber nicht auf die für `BackMove` gewünschte, letzte Absolutposition gefahren.

► **float fSearchIntensity** **PRIVATE**
nur ein Lesezugriff zur Berechnung von `TSteering::dGoalDistance`; kein Schreibzugriff

IV.9.2 Methoden

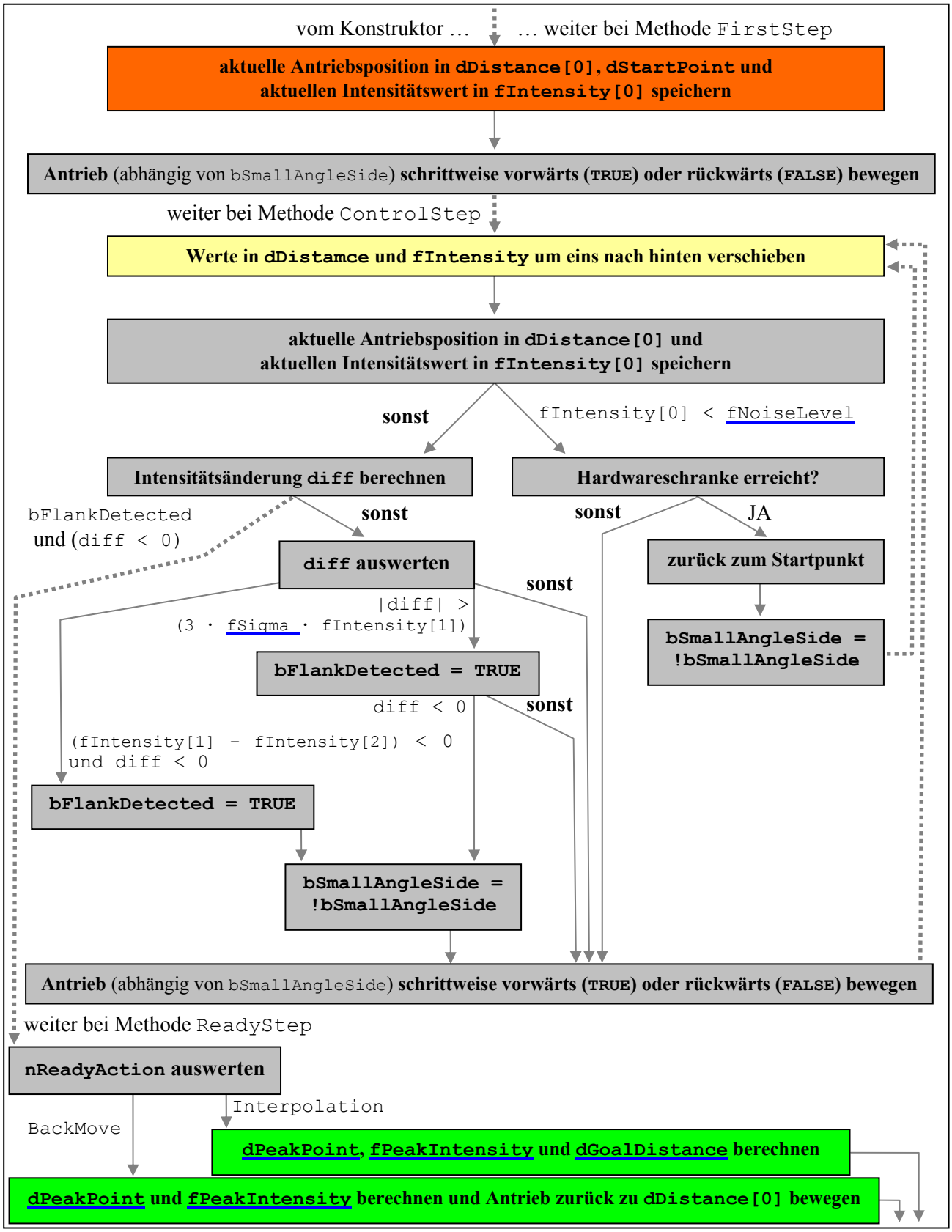

Abbildung 4 „Schematische Darstellung der Abläufe bei `TGotoPeakCmd`“ (Quelle: selbst)
blau unterstrichen sind Attribute des `Steering`-Objekts; punktierte Pfeile bedeuten Kommando-Pausierung

► **TGotoPeakCmd(TCmdTag)** **PUBLIC**

erzeugt `dDistance` und `fIntensity` dynamisch und initialisiert einige (unter **IV.9.1 Attribute** genannte) Attribute

► **~TGotoPeakCmd()** **PUBLIC**

gibt die dynamisch erzeugten Listen `dDistance` und `fIntensity` wieder frei

► **virtual TCCode FirstStep(void)** **PUBLIC**

aktuelle Absolutposition in `dDistance[0]` und `TSteering::dStartPoint` und aktuelle Intensitätswert in `fIntensity[0]` speichern; Abhängig von `bSmallAngleSide` wird der Antrieb entweder um einen Schritt vorwärts (`TRUE`) oder rückwärts (`FALSE`) bewegt. nach einer Pausierung der Kommandoverarbeitung wird in Methode `ControlStep` fortgesetzt

► **virtual TCCode ControlStep(void)** **PUBLIC**

Wert in `dDistance` und `fIntensity` um eins nach hinten verschieben („History“) und die aktuelle Absolutposition und Intensitätswert in `dDistance[0]` und `fIntensity[0]` speichern

`diff= fIntensity[0] – fIntensity[1]` Intensitätsänderung seit letzter Messung

- aktuelle Intensität ist unter dem Rausch-Level (`TSteerinmg::fNoiseLevel`)
  In beiden Fällen wird nach einer Pausierung der Kommandoverarbeitung wieder Methode `ControlStep` aufgerufen.
  - Hardwareschranke des ausgewählten Antriebs erreicht, dann bewege den Antrieb zurück zum Startpunkt (`TSteering::dStartPoint`), ändere die bevorzugte Bewegungsrichtung (`bSmallAngleSide`) und starte nach einer Pausierung wieder in Methode ControlStep
  - sonst wird der Antrieb wie in Methode `ReadStep` schrittweise auf den Peak zu bewegt
- `bFlankDetected == TRUE` und `diff < 0` Peak gefunden
  nach einer Pausierung der Kommandoausführung weiter bei Methode `ReadyStep`
- `| diff | > 3 · TSteering::fSigma · fIntensity[1]` große Intensitätsänderung
  `bFankDetected` `TRUE` setzen; wenn die aktuelle kleiner ist als zuletzt gemessene Intensität, dann ändere die bevorzugte Bewegungsrichtung (`bSmallAngleSide`); bewege den Antrieb schrittweise in Richtung Peak (wie Methode `FirstStep`) und starte nach einer Pausierung wieder bei Methode `ControlStep`
- `diff < 0` und `(fIntensity[1] – fIntensity[2]) < 0`
  (Intensität hat während der letzten beiden Messungen abgenommen)
  `bFankDetected` `TRUE` setzen; ändere die bevorzugte Bewegungsrichtung (`bSmallAngleSide`); bewege den Antrieb schrittweise in Richtung Peak (wie Methode `FirstStep`) und starte nach einer Pausierung wieder bei Methode `ControlStep`
- sonst bewege den Antrieb schrittweise in Richtung Peak (wie in Methode FirstStep) und starte nach einer Pausierung wieder bei Methode `ControlStep`

In jedem Fall wird `CRecall` zurückgegebenen

► **`virtual TCCode ReadyStep( void )`** **PUBLIC**

`TSteering::dPeakPoint` und `TSteering::fPeakIntensity` werden je nach `nReadyAction` unterschiedlich berechnet; wenn `nReadyAction == Interpolation` ist, wird auch `TSteering::dGoalDistance` berechnet, bei `BackMove` wird der Antrieb zur letzten Antriebsposition gefahren; abschließend kennzeichnet `eStep = CReady`, dass die Kommandoverarbeitung erfolgreich beendet wurde; gibt `CReady` zurück

Verwendung von zwei nicht initialisierten Attributen: `bBackMoveAction` und `fSearchIntensity`.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Fehlerinformationen (`nFailureId != 0`) oder Statusinformationen zurückzugeben:

eStep	Rückgabe im ersten Parameter
`CReady`	"GotoPeak Ready"
`CControlStep`	"D:<d> I:<i>" (wobei <d> = `dDistance[0]` und <i> = `fIntensity[0]` ist)
sonst	0

Tabelle 21 „Möglichkeiten der Textrückgabe bei `TGotoPeakCmd::GetShowData`“ (Quelle: selbst)

Rückgabewert stets `TRUE`

IV.9.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode’ inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	190
	‚**LOC** of **I**mplementation’	LOCI	153
	‚**LOC** of **D**eclaration’	LOCD	37
	‚**N**umber **O**f **A**ttributes’	NOA (0, 30)	9
	‚**N**umber **O**f **O**perations’	NOO (0, 50)	5
	‚**N**umber **O**f **M**embers’ Attribute + Methoden	NOM = NOA + NOO	14
	‚**N**umber **O**f **C**onstructors’	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods’	NOOM (0, 10)	5
	‚**P**ercentage of **Priv**ate **M**embers’	PPrivM	56
	‚**P**ercentage of **Prot**ected **M**embers’	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers’	PPubM	44
	‚**W**eighted **M**ethods **P**er **C**lass’	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity’	AC	66
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters’	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity’	CC	
Kommentare			
	‚**N**umber **O**f **C**omments’	NOC	59
	‚**T**rue **C**omment **R**atio’	TCR (5, 400)	40

Tabelle 22 „ausgewählte Metriken der Klasse `TGotoPeakCmd`“ (Quelle: Together® ‚Version 6.0)

IV.10 Klasse TShowValueCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TShowValueCmd` können erzeugt werden.

IV.10.1 Methoden

► **`TShowValueCmd( TCmdTag )`** **PUBLIC**

Abhängig vom ersten *Kommandoparameter* in den, als Parameter übergebenen, *-informationen* wird ein Meldungsfenster angezeigt. Anschließend ist das Kommando beendet, `eStep` wird `CReady` gesetzt.

erster Kommandoparameter	angezeigter Text
`Hwb`	**"Halbwertsbreite : <h> arcsec"**, wobei <h> = `TSteering::fHwb`
`Peak`	**"Peak (D,I) : (<d>,<i>)"**, wobei <d> = `TSteering::dPeakPoint` und <i> = `TSteering::fPeakIntensity`
`Start`	**"Startposition : <s>"**, wobei <s> = `TSteering::dStartPoint`

Tabelle 23 „Möglichkeiten bei der Positionierung von `TShowValueCmd`“ (Quelle: selbst)

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um `0` zurückzugeben; Rückgabewert stets `TRUE`

IV.10.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode’ inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	41
	‚**LOC** of **I**mplementation’	LOCI	31
	‚**LOC** of **D**eclaration’	LOCD	10
	‚**N**umber **O**f **A**ttributes’	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations’	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers’ Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors’	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods’	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers’	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers’	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers’	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass’	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity’	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters’	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity’	CC	
Kommentare			
	‚**N**umber **O**f **C**omments’	NOC	7
	‚**T**rue **C**omment **R**atio’	TCR (5, 400)	18

Tabelle 24 „ausgewählte Metriken der Klasse `TShowValueCmd`“ (Quelle: Together® ‚Version 6.0)

IV.11 Klasse TCalculateCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TCalculateCmd` können erzeugt werden.

IV.11.1 Methoden

► **`TCalculateCmd( TCmdTag )`** **PUBLIC**

Abhängig vom ersten *Kommandoparameter* in den, als Parameter übergebenen, *-informationen* werden verschiedene Werte berechnet und in `TSteering::dCalcResult` oder `TSteering::fHwb` gespeichert.

erster Kommandoparameter	welches Attribut wird geschrieben (Attribute von `TSteering`)
`Difference`	`dCalcResult = dCalcArg[2] – dCalcArg[1]`
`Opposite`	`dCalcResult = 2 · dPeakPoint – dCalcArg[1]`
`Hwb`	`fHWB = dCalcArg[2] – dCalcArg[1]` wird entsprechend der Einheit (des ausgewählten Antriebs) weiter umgerechnet
`Middle`	`dCalcResult = ( dCalcArg[1] + dCalcArg[2] ) / 2`

Tabelle 25 „Möglichkeiten bei der Berechnung von `TCalculateCmd`“ (Quelle: selbst)

Anschließend ist das Kommando beendet, `eStep` wird `CReady` zugewiesen.

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.11.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode’ inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	50
	‚**LOC** of **I**mplementation’	LOCI	40
	‚**LOC** of **D**eclaration’	LOCD	10
	‚**N**umber **O**f **A**ttributes’	NOA (0, 30)	0
	‚**N**umber **O**f **O**perations’	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers’ Attribute + Methoden	NOM = NOA + NOO	2
	‚**N**umber **O**f **C**onstructors’	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods’	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers’	PPrivM	0
	‚**P**ercentage of **Prot**ected **M**embers’	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers’	PPubM	100
	‚**W**eighted **M**ethods **P**er **C**lass’	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity’	AC	0
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters’	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity’	CC	
Kommentare			
	‚**N**umber **O**f **C**omments’	NOC	7
	‚**T**rue **C**omment **R**atio’	TCR (5, 400)	14

Tabelle 26 „ausgewählte Metriken der Klasse `TCalculateCmd`“ (Quelle: Together® ‚Version 6.0)

IV.12 Klasse TControlFlankCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TControlFlankCmd` können erzeugt werden.

IV.12.1 Attribute

- **`int nMotor`** **PRIVATE**
- **`float fControlRange`** **PRIVATE**
- **`float fStartIntensity`** **PRIVATE**
- **`float fSearchIntensity`** **PRIVATE**
- **`float fMinIntensity`** **PRIVATE**
- **`float fMaxIntensity`** **PRIVATE**
- **`float fIntensityRange`** **PRIVATE**
- **`BOOL bSmallAngleSide`** **PRIVATE**
- **`static const UINT nLifo= 4`** **PRIVATE**

bestimmt wie viele Elemente maximal in `dDistance` und `fIntensity` enthalten sein sollen; im Konstruktor mit 4 initialisiert (`2` würde völlig ausreichen)

- **`double* dDistance`** **PRIVATE**
- **`float* fIntensity`** **PRIVATE**

IV.12.2 Methoden

- **`TControlFlankCmd( TCmdTag )`** **PUBLIC**
- **`~TControlFlankCmd()`** **PUBLIC**
- **`virtual TCCode FirstStep( void )`** **PUBLIC**
- **`virtual TCCode ControlStep( void )`** **PUBLIC**
- **`virtual TCCode ReadyStep( void )`** **PUBLIC**
- **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**

überschrieben um Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.12.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	198
	‚**LOC** of **I**mplementation'	LOCI	162
	‚**LOC** of **D**eclaration'	LOCD	36
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	11
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	5
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	16
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	5
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	61
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	39
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	49
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	84
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	65

Tabelle 27 „ausgewählte Metriken der Klasse `TControlFlankCmd`" (Quelle: Together® ,Version 6.0)

IV.13 Klasse TSetupScanCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TSetupScanCmd` können erzeugt werden.

IV.13.1 Attribute

► **`TScan* ScanW`** **PRIVATE**

IV.13.2 Methoden

► **`TSetupScanCmd( TCmdTag )`** **PUBLIC**

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**
überschrieben um Fehlerinformationen (`nFailureId == 1`) oder Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.13.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	61
	‚**LOC** of **I**mplementation'	LOCI	48
	‚**LOC** of **D**eclaration'	LOCD	13
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	1
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	3
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	25
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	75
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	9
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	8
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	14

Tabelle 28 „ausgewählte Metriken der Klasse `TSetupScanCmd`" (Quelle: Together® ‚Version 6.0)

IV.14 Klasse TSetFileNameCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TSetFileNameCmd` können erzeugt werden.

IV.14.1 Attribute

► **`TScan* ScanW`** **PRIVATE**

► **`TAreaScan* AreaScanW`** **PRIVATE**

IV.14.2 Methoden

► **`TSetFileNameCmd( TCmdTag )`** **PUBLIC**

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**
überschrieben um Fehlerinformationen (`nFailureId == 1`) oder Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.14.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	53
	‚**LOC** of **I**mplementation'	LOCI	39
	‚**LOC** of **D**eclaration'	LOCD	14
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	2
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	4
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	40
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	60
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	18
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	7
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	13

Tabelle 29 „ausgewählte Metriken der Klasse `TSetFileNameCmd`" (Quelle: Together® ‚Version 6.0)

IV.15 Klasse TSaveDataCmd

Deklaration : M_STEERH.CPP
Implementation : M_STEERG.H
Es gibt keine abstrakten Methoden, d.h. Objekte von `TSaveDataCmd` können erzeugt werden.

IV.15.1 Attribute

► **`TScan* ScanW`** **PRIVATE**

► **`TAreaScan* AreaScanW`** **PRIVATE**

IV.15.2 Methoden

► **`TSaveDataCmd( TCmdTag )`** **PUBLIC**

► **`virtual BOOL GetShowData( LPSTR )`** **PUBLIC**
überschrieben um Fehlerinformationen (`nFailureId != 0`) oder Statusinformationen zurückzugeben; Rückgabewert stets `TRUE`

IV.15.3 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	55
	‚**LOC** of **I**mplementation'	LOCI	41
	‚**LOC** of **D**eclaration'	LOCD	14
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	2
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	2
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	4
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	2
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	40
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	60
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	18
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	7
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	13

Tabelle 30 „ausgewählte Metriken der Klasse `TSaveDataCmd`" (Quelle: Together® ‚Version 6.0)

IV.16 Klasse TScanCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TScanCmd` können erzeugt werden.

IV.16.1 Attribute, Methoden

Die für die Diffraktometrie relevanten Kommandos wurde bereits einem Reverse-Engineering Prozess unterzogen – Informationen findet man unter [3].

► **`DWORD dwStartTimeTicks`** **`PRIVATE`**

Anfangszeit des Scans; benutzt um Restzeit zu berechnen

IV.16.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	503
	‚**LOC** of **I**mplementation'	LOCI	355
	‚**LOC** of **D**eclaration'	LOCD	148
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	**32**
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	5
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	37
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	5
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	84
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	16
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	142
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	179
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	53

Tabelle 31 „ausgewählte Metriken der Klasse `TScanCmd`" (Quelle: Together® ‚Version 6.0)

IV.17 Klasse TAreaScanCmd

Deklaration : M_STEERH.CPP

Implementation : M_STEERG.H

Es gibt keine abstrakten Methoden, d.h. Objekte von `TAreaScanCmd` können erzeugt werden.

IV.17.1 Attribute, Methoden

Die für die Diffraktometrie relevanten Kommandos wurde bereits einem Reverse-Engineering Prozess unterzogen – Informationen findet man unter [3].

IV.17.2 Bewertung

Metrik		Kennung (min,max)	Wert
Klasse			
	‚**L**ines **O**f **C**ode' inkl. Leer- und Kommentarzeilen	LOC (0, 1000)	226
	‚**LOC** of **I**mplementation'	LOCI	157
	‚**LOC** of **D**eclaration'	LOCD	69
	‚**N**umber **O**f **A**ttributes'	NOA (0, 30)	18
	‚**N**umber **O**f **O**perations'	NOO (0, 50)	5
	‚**N**umber **O**f **M**embers' Attribute + Methoden	NOM = NOA + NOO	23
	‚**N**umber **O**f **C**onstructors'	NOCON (0, 5)	1
	‚**N**umber **O**f **O**verridden Methods'	NOOM (0, 10)	5
	‚**P**ercentage of **Priv**ate **M**embers'	PPrivM	75
	‚**P**ercentage of **Prot**ected **M**embers'	PProtM (0, 10)	0
	‚**P**ercentage of **Pub**lic **M**embers'	PPubM	25
	‚**W**eighted **M**ethods **P**er **C**lass'	WMPC1 (0, 30)	
Attribute			
	‚**A**ttribute **C**omplexity'	AC	37
Methoden			
	‚**M**aximum **N**umber **O**f **P**arameters'	MNOP (0, 4)	1
	‚**C**yclomatic **C**omplexity'	CC	
Kommentare			
	‚**N**umber **O**f **C**omments'	NOC	81
	‚**T**rue **C**omment **R**atio'	TCR (5, 400)	52

Tabelle 32 „ausgewählte Metriken der Klasse `TAreaScanCmd`" (Quelle: Together® ‚Version 6.0)

V Attribute

► **`TSteering Steering`** **GLOBAL**

Dieses Objekt kann nur zur Makroverarbeitung benutzt werden. Es dürfen keine weiteren Objekte von `TSteering` erstellt werden (siehe Konstruktor von `TSteering`).

► **`static const UINT _MAXMACROCOUNT= 20`** **GLOBAL**

definiert die maximale Anzahl von Makros, kann kein `static-const` Member in `TSteering` sein

► **`TAdjustmentParameter AdjustmentParameter`** **GLOBAL**

Justageparameter; wird in `TSetAdjustmentParam` mit den eingegebenen Werten initialisiert; in `TAdjustmentExecute` wird nur das Element `fNoiseLevel` verwendet

VI Anhang

VI.1 Verwandte Dokumente

[1] „Beschreibung einer Schnittstelle zur Motorenansteuerung: Das C-Interface des RTK-Steuerungsprogramms“, Studienarbeit von Sebastian Freund und Derrick Hepp

[2] „Reverse-Engineering der objektorientierten Teile des Subsystems Motorsteuerung“, Version 1.5 von Thomas Kullmann und Günther Reinecker

[3] „Reverse-Engineering des Subsystems Detektoren des RTK-Steuerprogramms“, November 2000 von Jan Picard, René Harder und Alexander Paschold

[4] „Der Aufbau von Makros im XRay-Programm“, am 09.09.1999 von Kay Schützler erstellt

[5] „Pflichtenheft ‚Manuelle Justage’“, Version 2.1 von Thomas Kullmann und Günther Reinecker

[6] „Toter Code: Ergebnisse mit SNiFF und McCabe“, am 15.09.1999 von Stefan Lützkendorf erstellt

[7] „Layoutkonventionen und Steuerelemente“, Version 1.0 von Thomas Kullmann und Günther Reinecker

VI.2 Index

VI.3 Tabellen

VI.4 Abbildungen