Wiedergewinnung von Subsystemen durch Use-Case-Analyse und Dateirestrukturierung am Beispiel des XCTL-Systems

Kay Schützler

30. April 2001

Zusammenfassung

In der vorliegenden Arbeit geht es darum, die aktuelle Subsystemstruktur eines existierenden komplexen und umfangreichen Softwaresystems zu ermitteln und zu dokumentieren. Die dazu verwendeten Methoden werden ebenso beschrieben wie die damit gefundenen Subsysteme. Neben der Dokumentation der Struktur entstand eine Version des Systems, bei der dessen einzelne Quelldateien auf Unterverzeichnisse verteilt wurden, die den gefundenen Subsystemen entsprechen. In dieser Arbeit wurde im Sinne des Reverse Engineering die ursprüngliche Architektur nur herausgearbeitet und übersichtlicher dargestellt, nicht jedoch verändert.

Abbildungsverzeichnis

Tabellenverzeichnis

Inhaltsverzeichnis

1 Einleitung

1.1 Begriffsbestimmungen

Software Engineering

Software Engineering umfasst sowohl den Prozess der Neuentwicklung von Software als auch die Wartung, die oftmals einen umgekehrten Prozess beinhaltet, da beispielsweise die Dokumente aus den früheren Entwicklungsphasen unvollständig sein können oder gar fehlen.

Forward und Reverse Engineering. Häufig möchte man jedoch exakt bezeichnen können, welchen dieser Prozesse man betrachtet. Daher wurden die Begriffe Forward und Reverse Engineering geschaffen. Dabei bezeichnet Forward Engineering den klassischen Prozess des Fortschreitens von höheren Abstraktionsniveaus zur tatsächlichen Implementation des Softwaresystems. Das Reverse Engineering stellt den umgekehrten Prozess dar. Das Ziel ist dabei die Wiedergewinnung der frühen Dokumente der Softwareentstehung.

Reengineering und Restructuring. Es soll hier explizit festgestellt werden, dass eine Veränderung des vorhandenen Systems bezüglich seines Funktionsumfangs und seines externen Verhaltens nicht Gegenstand des Reverse Engineering ist. Diese Aufgaben werden unter dem Begriff Reengineering zusammengefasst.

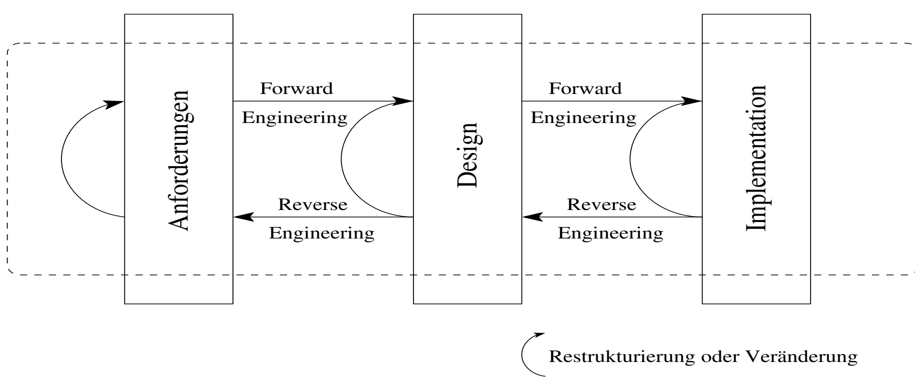

Abbildung 1: Die Beziehungen zwischen den Begriffen des Software Engineering

Mit Restructuring bzw. Restrukturierung schließlich bezeichnet man einen Prozess, der zwar das externe Verhalten (funktional und semantisch) des Softwaresystems unverändert lässt, jedoch seiner Darstellung auf einem bestimmten Abstraktionsniveau eine neue Form gibt. Als Beispiel ist hier die Umstellung der Implementation auf eine neue Programmiersprache oder auf eine neue Compiler-Version denkbar.

Subsysteme

Der Begriff Subsystem stammt aus dem Bereich der so genannten Software-Architektur, wobei Architektur hier im weiter gefassten Sinn des englischen Begriffs Architecture zu verstehen ist, der wohl besser mit Aufbau übertragen werden sollte.

Große Softwaresysteme sollten in Komponenten zerlegbar sein, damit sie überhaupt beherrschbar bleiben. Solche statischen Komponenten können je nach Feinheit der Zerlegung mit Subsystem oder Modul bezeichnet werden, wobei ein Subsystem eine Gruppierung von Modulen (oder niedrigeren Subsystemen) darstellt.

Die Subsystembildung ist entscheidend für die Wartbarkeit komplexerer Softwaresysteme, da sonst ab einer bestimmten Größe eines Systems jegliche Übersicht verloren gehen würde.

Use Cases

Allgemein ausgedrückt stellen Use Cases eine verbale operationale Beschreibung der Nutzung eines Systems dar (Vgl. [1], S. 48ff). Die Kernaufgabe von Use Cases ist somit, die funktionalen Anforderungen auch dynamisch allgemein verständlich und klar festzuhalten. Die dynamische Beschreibung von Anforderungen ermöglicht größere Detailtiefe und Genauigkeit mit nur geringem Mehraufwand.

Einen weiteren wichtigen Punkt bei der Nutzung von Use Cases bildet die Möglichkeit, dass Use Cases sich aufeinander beziehen können. Dafür stehen verschiedene Relationen zur Verfügung (Vgl. [2], S. 166ff).

Eine wichtige Relation zwischen zwei Use Cases stellt die Generalisierung bzw. Verallgemeinerung dar. Durch die Generalisierung entsteht ein abstrakter Use Case, der die gemeinsame Funktionalität mehrerer konkreter Use Cases beinhaltet. Die konkreten Use Cases erben somit das Verhalten des abstrakten Use Cases, wodurch unter Anderem eine vereinfachte und übersichtlichere Darstellung ermöglicht wird.

Die zweite im Rahmen dieser Arbeit verwendete Relation ist die *extend*-Relation bzw. die Erweiterung. Mit Hilfe der *extend*-Relation werden potenzielle Erweiterungen der Operationen eines Use Cases modelliert. Der erweiterte Use Case kann an bestimmten so genannten Erweiterungspunkten um die Aktionen des erweiternden Use Cases verlängert werden.

1.2 Das XCTL-Programm

Das XCTL-Programm stellt ein am Institut für Physik entwickeltes Softwaresystem zur Steuerung der so genannten Roentgen-Topographie-Kamera dar, die hauptsächlich zur Untersuchung der Reinheit und Struktur von Siliziumproben eingesetzt wird. Dabei wird eine Probe gerichteten Roentgen-Strahlen ausgesetzt. Die Intensitätsverteilung der reflektierten Roentgen-Strahlung ist von der Regelmäßigkeit der bestrahlten Kristallgitterstruktur abhängig.

Das Aufgabengebiet des XCTL-Programms umfasst dabei die folgenden Punkte:

- Interaktive und automatische Steuerung mehrerer Stellmotoren zur Justage der Probe
- Erfassung der Roentgen-Messwerte durch die Abfrage verschiedener externer Roentgen-Detektoren
- Grafische Darstellung und Speicherung der erfassten Werte
- Unterstützung und Durchführung verschiedener Messprozesse

Leider ist das XCTL-Programm aufgrund seiner Komplexität teilweise fehlerhaft und unvollständig. Zusätzlich dazu verließ der ursprüngliche Entwickler das Institut für Physik. Daher entstand auf Anfrage der Physik eine Projektgruppe am Institut für Informatik, die sich im Rahmen eines Seminars mit der Wartung des XCTL-Programms befasst. Detailliertere Ausführungen zum Gegenstandsbereich des XCTL-Systems und des Seminars finden sich in [3] und [4].

1.3 Ziel der Arbeit

Das Projekt "Softwaresanierung" besteht hauptsächlich aus einem jedes Semester neu angebotenen Seminar, in dem die Studenten eine Reihe von Methoden des Reverse Engineering an einem Beispiel aus der Praxis anwenden können. Das XCTL-Programm bietet dazu reichlich Möglichkeiten, da seine Quelltexte auf den ersten Blick nahezu chaotisch anmuten. Allein der Umfang der Quellen[1] macht einen schnellen Zugang unmöglich.

Zunächst konzentrierten wir uns auf ein allgemeines Verständnis des Programms und seiner Anwendung sowie auf die Beseitigung offensichtlicher Fehler. Zusätzlich dazu entstand im Rahmen einer Diplomarbeit eine weitere Komponente, die bereits eine Erweiterung der Funktionalität des ursprünglichen Programms darstellt.

Im Laufe mehrerer Semester wurde immer klarer, dass eine allgemeine Übersicht und Modularisierung des Programms von Vorteil wäre. Zum einen können für ein Software-System aus klar abgegrenzten Komponenten leichter Teilaufgaben verteilt werden, zum anderen ist es für einen Bearbeiter auch leichter, einen Einstieg in die Materie zu finden, wenn er zunächst ein abgegrenztes Teilsystem

[1] ca. 31000 Quelltextzeilen in Borland C++

behandeln kann. Dieser Punkt ist auch deswegen von großer Wichtigkeit, da in einem Seminar naturgemäß die teilnehmenden Studenten relativ häufig wechseln, somit eine schnelle Einarbeitung für neue Teilnehmer gewährleistet sein muss.

Daher ist es also das Ziel dieser Arbeit, eine Zerlegung des XCTL-Systems in disjunkte Subsysteme zu finden und zu dokumentieren. Die so entstehende Quelltextversion soll die bisherigen Quellen bei anschließenden Arbeiten im Rahmen des Projekts "Softwaresanierung" ersetzen.

2 Allgemeine Beschreibung und Charakterisierung des Vorgehens

2.1 Zur Einordnung der Methodik

Die Ermittlung von Subsystemen in existierenden Systemen wird derzeit durch unterschiedliche Methoden bzw. Tools unterstützt (Vgl. [5] und [6]). Die dabei eingesetzten Techniken unterscheiden sich in ihrer Zielstellung, wobei insbesondere auch unterschiedliche Rahmenbedingungen der Ausgangssoftware zu beachten sind. Zu diesen Rahmenbedingungen zählt beim Bauhaus Projekt ([5]) ein zu untersuchendes System, das in der Sprache C implementiert ist und für das Vorschläge für objektorientierte Komponenten gemacht werden. Für das URCA-System ([6]) dagegen wird in Visual C++ erstellte Software vorausgesetzt, die zudem bereits weitestgehend objektorientiert strukturiert sein soll.

Somit konnten beide Systeme nicht auf unser in Borland C++ vorliegendes XCTL-Programm angewendet werden, zum einen auf Grund der verschiedenen Implementationssprachen und zum anderen auf Grund der fehlenden sauberen Software-Architektur. Die vorliegende Architektur ist vorrangig durch eine Mischung von imperativem und objektorientiertem Stil geprägt, zusätzlich dazu verfügt das System über eine chaotische Schnittstellenstruktur, die kaum allein durch Tool-Anwendung bereinigt werden kann.

Vielmehr ist in einem ersten Schritt eine manuelle Bereinigung der Ausgangssituation gefragt, die inhaltlich und semantisch orientiert sein muss. Diese Bereinigung umfasst zwei Teilprozesse:

- die Quelltextabgrenzung, also die semantische Zuordnung von Quellcode zu Teilaufgaben bzw. Anwendungsfällen des Systems und
- die "Reparatur" defekter Schnittstellenbeziehungen.

Der erste Teilprozess befasst sich mit der konkreten Zuordnung von Quellcodeteilen zu bestimmten Teilaufgaben. Der vorgefundene Code selbst wird nicht in Frage gestellt, sondern lediglich von anderen Bestandteilen abgegrenzt. Damit bleibt die ursprüngliche Software-Architektur erhalten, sie wird lediglich herausgearbeitet bzw. hervorgehoben. Letztlich stellt genau das auch die Bedeutung des Begriffs "Architecture Recovery" dar, der von anderen Autoren (s. [5]) in gewisser Weise unberechtigt als "Verbesserung der Architektur" beispielsweise in Form von Vorschlägen für den Übergang existierender imperativer in objektorientierte Architekturen ausgelegt wird.

Nach diesem ersten Schritt, der Quellcodeabgrenzung unter Beibehaltung der ursprünglichen Software-Architektur, treten als offensichtliche Mängel dieser Architektur unzulängliche Schnittstellenbeziehungen unterschiedlichster Ausprägung klarer zu Tage. Dazu zählen in C/C++-Systemen:

- eine schlechte bzw. fehlende Konzeption für die Strukturierung der Header- und Implementationsdateien,
- Direktzugriff auf Variablen eines anderen Subsystems,
- Verletzung des Prinzips der Trennung von Benutzeroberfläche und inhaltlichen Komponenten u.a.

Vorgefundene mangelhafte Schnittstellenbeziehungen müssen ebenfalls manuell und semantisch orientiert bereinigt oder mit Hilfe von so genannten Wrapper-Komponenten (s. a. [7] S. 669ff) versteckt werden.

Anschließend an diese manuell auszuführenden Schritte kann eine Restrukturierung durchgeführt werden, für die dann unterstützende Tools einsetzbar sind. Hierbei ist unter Anderem die Überführung imperativer in objektorientierte Software-Architekturen von Interesse.

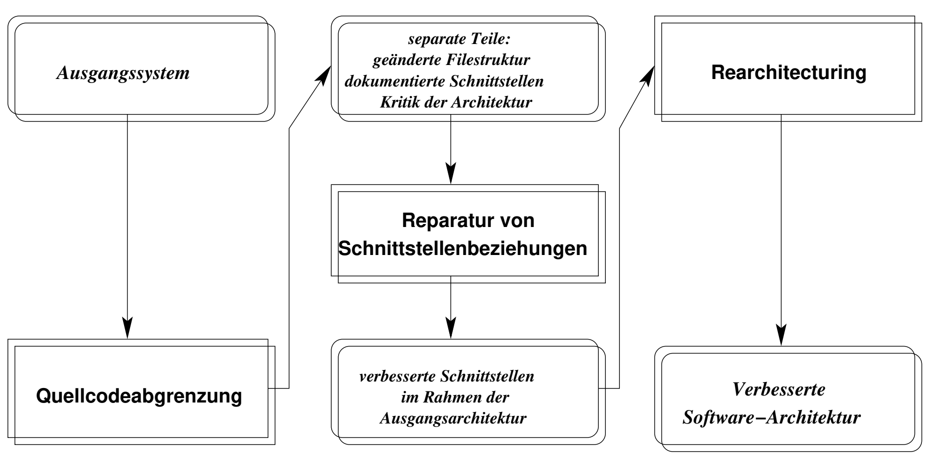

Abbildung 2: Der Prozess der Architekturverbesserung in der grafischen Übersicht

Der Gesamtprozess der Architekturverbesserung kann Abbildung 2 entnommen werden. Die vorliegende Arbeit hat den ersten Teilprozess, die Quellcodeabgrenzung, zum Gegenstand, die methodisch ausgearbeitet und auf das XCTL-System als Beispiel angewendet wird.

2.2 Die Methodik im Überblick

Die Wiedergewinnung von Subsystemen erfolgte in sieben Schritten, wie auch Abbildung 3 verdeutlicht. Es handelt sich um eine manuelle Schrittfolge, deren einzelne Charakteristika in Abschnitt 2.4 behandelt werden.

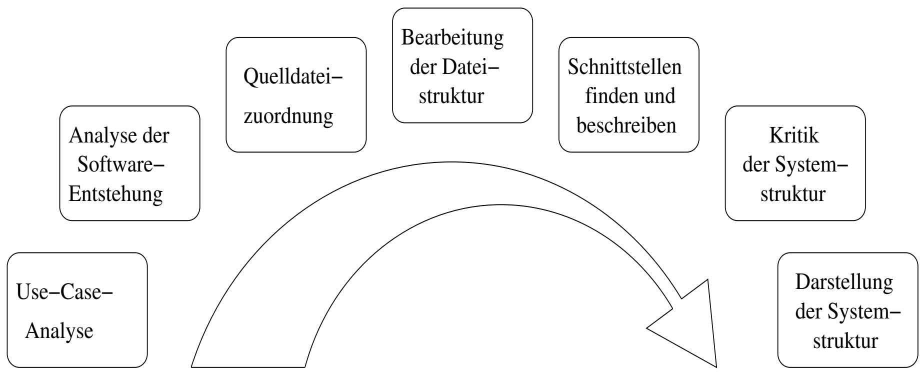

Abbildung 3: Das Vorgehensmodell in der grafischen Veranschaulichung

Das Vorgehensmodell beschreibt einen Reverse-Engineering-Prozess, bei dem in einem existierenden Softwaresystem anhand von Use Cases die vorhandenen Subsysteme identifiziert werden sollen. Am Ende des Prozesses steht eine Version des Systems, die inhaltlich und funktional identisch mit dem Ausgangssystem ist, in der jedoch die Subsysteme durch Verteilung der Quelldateien auf verschiedene Unterverzeichnisse gut erkennbar herausgearbeitet sind. Dadurch, durch die entstehende grafische Darstellung des Systems und durch die Dokumentation der Kritikpunkte entsteht eine gute Ausgangssituation für weitere Arbeiten mit dem System.

2.3 Allgemeine Beschreibung der einzelnen Schritte

Use-Case-Analyse. Zuerst werden unter Einbeziehung der Anwender des Systems die wichtigsten Anwendungsfälle bzw. Use Cases erfasst. Die Use Cases bieten erste Anhaltspunkte für die Aufteilung des Softwaresystems nach Einsatzbereichen. Hinweise für Teilsysteme ergeben sich zum Beispiel aus Funktionalität, die einer Reihe von Use Cases gemeinsam ist, oder aus Fachbegriffen, die mit bestimmten Use Cases assoziiert werden und die auch in den Quellen auffindbar sind.

Analyse der Software-Entstehung. Die anschließende Analyse der Softwareentstehung hat vor allem den Zweck, Aussagen über die zu erwartende Qualität der aufzufindenden Subsystemzerlegung zu treffen. Je stärker Subsysteme in der Entwicklung des Softwaresystems eine Rolle gespielt haben, desto leichter werden sie im Quelltext identifiziert werden können. In diesen Schritt können prinzipiell alle zur Softwareentstehung verfügbaren Informationen eingehen, oft reicht aber auch schon die Untersuchung eines speziellen Aspekts wie zum Beispiel die Anzahl der Entwickler oder die Zuordnung bestimmter Entwickler zu einzelnen Teilaufgaben. Weitere derartige Informationen können beispielsweise die Kenntnis, ob möglicherweise ein standardisiertes Entwicklungsmodell angewendet wurde, die Art und Weise der Aufgabenspezifikation u. ä. darstellen.

Quelldateizuordnung. Die dritte Phase befasst sich mit der Zuordnung der Quelldateien zu den einzelnen aus den Use Cases abgeleiteten Subsystemen. Hierbei fließen Informationen aus den ersten beiden Schritten in die jeweiligen Entscheidungen über die Zugehörigkeit einer Datei zu einem bestimmten Systemteil ein. Die Entscheidung, zu welchem Subsystem eine Datei gehört, wird auf Grund ihres Inhalts getroffen. Dazu können Namen von Klassen, Funktionen, Variablen etc. sowie eventuell im Quelltext vorhandene Kommentare genutzt werden. Die Analyseergebnisse des zweiten Schritts helfen bei der Bewertung der Sicherheit einer Zuordnung.

Bearbeitung der Dateistruktur. In vielen Fällen werden die Quelldateien nicht vollständig und überschneidungsfrei auf die einzelnen vermuteten Subsysteme zu verteilen sein. Dies ist insbesondere zu erwarten, wenn bei der Entwicklung der Software nur wenig Wert auf Subsysteme gelegt wurde. Insbesondere Headerdateien sind anfällig für solche Fehler, wenn ihnen kein durchdachtes und konsistent angewendetes System zu Grunde liegt. In solchen Fällen kann eine vorsichtige Umstrukturierung die Zerlegung erheblich fördern. Dabei sollten Deklarationen und Definitionen möglichst genau den Subsystemen zugeordnet werden, zu denen sie höchstwahrscheinlich konzeptionell gehören. Hier sind die gleichen Methoden verwendbar, die auch bei der Quelldateizuordnung hilfreich waren. Eine analoge Veränderung von Implementationsdateien ist mit der Gefahr verbunden, dass die ursprüngliche Architektur versehentlich verändert wird. Daher sollten an den Implementationsdateien grundsätzlich keine Änderungen vorgenommen werden.

Schnittstellen finden und beschreiben. In diesem Schritt dominiert derzeit ein eher pragmatisches Vorgehen. Da man ja zuvor die Implementations- und Headerdateien den einzelnen Subsystemen zugeordnet hat, kann man zur Annäherung an die Schnittstellen zunächst die Include-Direktiven in jeder Implementationsdatei darauf untersuchen, inwieweit Headerdateien aus anderen Subsystemen verwendet werden. Die Menge der Headerdateien eines Subsystems, die in Implementationsdateien eines anderen Subsystems inkludiert werden, enthält dann die Schnittstelle des ersten Subsystems. Die in diesen Headerdateien vorgefundenen global zugänglichen Funktionen, Typen, Klassen etc. müssen zunächst als Teile der Schnittstelle angesehen werden. Es kann dann noch überprüft werden, welche konkreten Inhalte aus diesen Headerdateien auch tatsächlich in den Implementationsdateien anderer Subsysteme anzutreffen sind, mehr als eine syntaktische Beschreibung der Schnittstellen kann jedoch ohne genauere Analyse der einzelnen Subsysteme nicht gewonnen werden. Daher beschränkt sich die Beschreibung der Schnittstellen im Rahmen der Subsystemfindung auf eine überblicksartige Darstellung der wichtigsten Schnittstellenelemente, die vorrangig von den öffentlich zugänglichen Klassen gebildet werden.

Kritik der Systemstruktur. In den vorausgehenden Abschnitten werden mit Sicherheit Design-Probleme und -Fehler des Ausgangssystems aufgefallen sein. Allerdings sind diese Probleme von unterschiedlicher Natur und auch an sehr verschiedenen Stellen zu erwarten. Daher sollte in dieser Analysephase vor der direkten Kritik der Subsystemaufteilung und Schnittstellenbildung eine Klassifizierung und Beschreibung aller Design-Probleme und -Fehler erfolgen. Diese Klassifizierung kann dann die Grundlage für eine bewertende Kritik der Systemstruktur bilden, bei der Punkte wie die Klarheit der Trennung der einzelnen Subsysteme, die allgemeine Qualität und Plausibilität der Strukturierung oder die Güte der Schnittstellen behandelt werden.

Darstellung der Systemstruktur. Im letzten Schritt entstehen schließlich mehrere grafische Veranschaulichungen der ermittelten Systemstruktur. Da die einfache Aufführung der Subsysteme und ihrer Beziehungen untereinander besonders bei niedriger Qualität der Subsystemunterteilung schnell unübersichtlich wird, sollten hierarchisierte Grafiken erstellt werden, in denen die Beziehungen stärker strukturiert werden können. Zum allgemeinen Verständnis des Systems tragen auch Übersichten über die zu jedem Subsystem gehörenden Klassen und die Beziehungen zwischen den wichtigsten Klassen bei. Dazu zählen insbesondere die Klassen, über die zwischen den Subsystemen kommuniziert wird. Diese Übersichten können somit auch als Darstellungen der Subsystemschnittstellen verstanden werden.

2.4 Merkmale des Vorgehensmodells

Fachliche Voraussetzungen zur Methodenanwendung. Da zu Beginn des Vorgehens die Untersuchung von Use Cases des Softwaresystems steht, ist zur erfolgreichen Anwendung der Methode ein grundlegendes Verständnis des Gegenstandsbereichs der Software unabdingbar. Eine weitere Voraussetzung stellt die Vertrautheit mit den Strukturen der jeweiligen Implementationssprache dar. Der weitere Verlauf der Methode verlangt nach einer kreativen Zusammenführung von Gegenstandsbereich und Programmstrukturen.

“Sanfte” Restrukturierung. Das in dieser Arbeit verwendete Vorgehensmodell stellt eine sehr sanfte Methode der Restrukturierung dar.

Die Umstellungen im Quelltext sind vergleichsweise gering, da in den Implementationsdateien nur einzelne Definitionen umgelagert werden, wenn sie offensichtlich einem anderen Subsystem zuzuordnen sind und sich bei der Umordnung die Abhängigkeiten zwischen den einzelnen Dateien verringern.

Eine weitere zulässige Veränderung stellt die Zerlegung von Headerdateien dar, wenn in diesen Headerdateien klar abgrenzbare Teile verschiedener Subsysteme vorzufinden sind. Dadurch verändern sich natürlich auch die Include-Direktiven in den jeweiligen Implementationsdateien, was allerdings einen beherrschbaren Aufwand zur Folge hat.

Permanente Funktionsfähigkeit. Ein sehr wichtiges Kennzeichen der als Schrittfolge realisierten Methode ist die Tatsache, dass in jedem Schritt eine vollständig funktionsfähige Variante des Systems existiert, da die Veränderungen sich nur auf die Anordnung des Quelltextes, nicht jedoch auf seinen Inhalt beziehen.

Parallelisierbarkeit einzelner Schritte. Es ist ebenfalls möglich, einige Schritte zeitgleich und teilweise mit verschiedenen Gruppen zu bearbeiten. So sind die Use-Case-Analyse und die Analyse der Softwareentstehung problemlos parallelisierbar. Ähnliches gilt für die letzten beiden Schritte.

Parallele Bearbeitung der Quellen. Da die Umstrukturierung derart sanft erfolgt, können prinzipiell andere Entwickler parallel dazu am System weiterarbeiten, wenn eine gut organisierte zentrale Verwaltung der Quellen, zum Beispiel mit dem *Concurrent Version System* (*CVS*), erfolgt.

Ziel des Vorgehens: Disjunkte Subsysteme. Da bei dieser Methode das Hauptaugenmerk auf der Verdeutlichung der vorhandenen Subsysteme liegt, wird auch hauptsächlich Wert auf die Identifizierung disjunkter Teile des Systems gelegt.

3 Vorgehensbeschreibung am Fallbeispiel

3.1 Analyse der Use Cases

3.1.1 Die Use Cases im XCTL-Projekt

Da ein Use Case neben einer Situationsbeschreibung auch ein erwartetes Systemverhalten umfasst, liegt die Annahme nahe, dass aus einer Reihe von Use Cases auch auf Systemteile bzw. Subsysteme geschlossen werden kann.

Daher wurde im April 2000 schließlich auch im Projekt "Softwaresanierung" der Wunsch formuliert, das XCTL-Programm mit Hilfe einer solchen Use-Case-Analyse in einzelne Subsysteme zu zerlegen, zumal die Aufteilung der Aufgaben für die einzelnen Projektteilnehmer bisher auch schon anhand einer gewissen Use-Case-Sicht vorgenommen worden war.

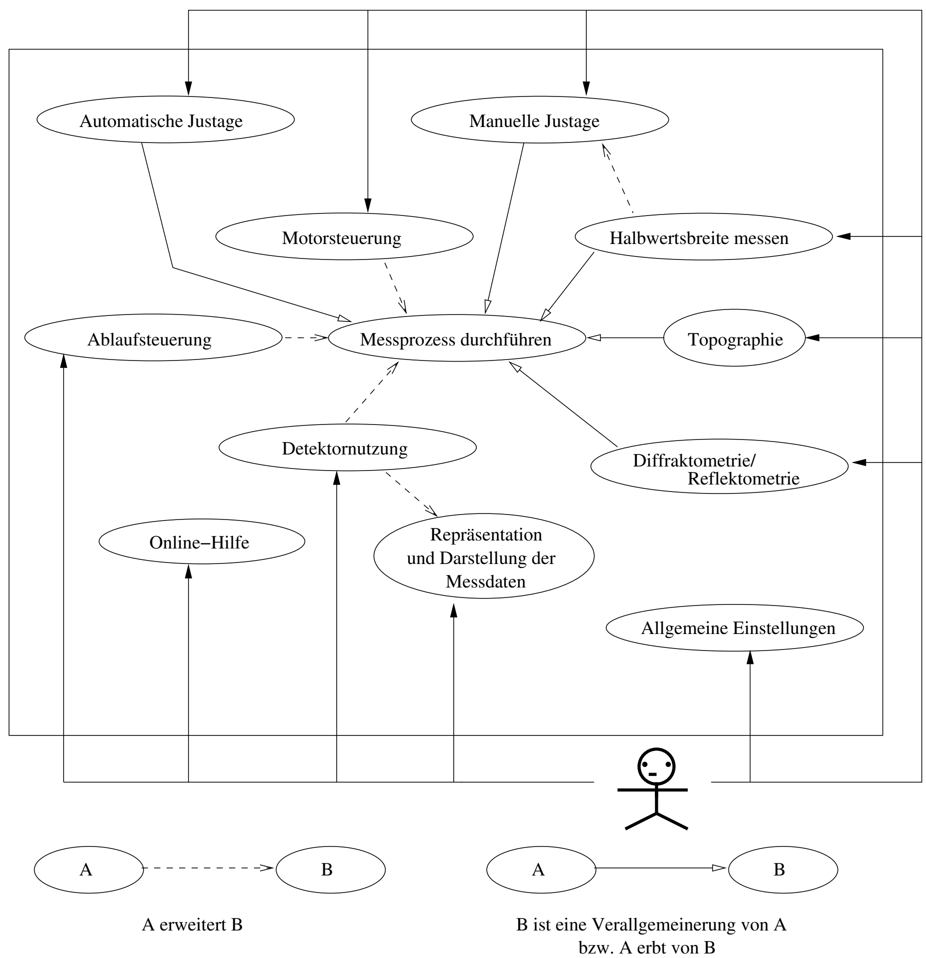

Abbildung 4: Use Cases im XCTL-Projekt

Aus diesen Anwendungsfällen entstand schließlich auch eine Liste mit einer Reihe von Use Cases, die für die Subsystembildung in Frage kamen. Die für das XCTL-Projekt wichtigen Use Cases sind in Abbildung 4 dargestellt.

3.1.2 Beschreibung der Use Cases

Motorsteuerung. Die Ansteuerung der Motoren stellt einen sehr umfangreichen Anwendungsfall dar. Die einzelnen Teile umfassen

- die Erkennung und Einrichtung angeschlossener Motoren,
- die Verwaltung von Statusinformationen für die angeschlossenen Motoren und
- die Ansteuerung der Motoren durch die Kommunikation mit den Controller-Karten.

Die Erkennung der Motoren beschränkt sich hauptsächlich auf die Überprüfung, ob ein in der Initialisierungsdatei aufgeführter Motor auch tatsächlich angeschlossen ist. Die Einrichtung der Motoren bietet zum Beispiel die Möglichkeit, einen so genannten Referenzpunktlauf durchzuführen, der die Konsistenz zwischen der tatsächlichen und der vom Programm intern gespeicherten Motorposition sicherstellen soll.

Zu den Statusinformationen zählt beispielsweise die aktuelle Position eines Motors. Auch die Information, ob ein bestimmter Motor in Bewegung ist oder nicht, wird vom Programm erfasst. Zusätzlich werden noch die in der Initialisierungsdatei angegebenen Maximal- und Minimalwerte bezüglich Position, Geschwindigkeit und Schrittweite der einzelnen Motoren verwaltet.

Die Ansteuerung der Motoren schließlich befasst sich mit der Übersetzung von Positionsangaben für einen Motor in Steuerbefehle für die jeweilige Controller-Karte und bearbeitet auch sonstige Kommunikationsaufgaben, die in der Interaktion von XCTL-Programm und Controller-Karten anfallen.

Weiterhin besteht die Möglichkeit, dass XCTL-Programm für Testzwecke auch ohne angeschlossene Motoren zu benutzen. Das Verhalten der Motoren wird dann von der Software simuliert.

Detektornutzung. Auch die Detektornutzung umfasst zahlreiche einzelne Aufgaben, darunter

- die Erkennung und Einrichtung angeschlossener Detektoren,
- die Verwaltung von Statusinformationen für die angeschlossenen Detektoren und
- die Abfrage der gemessenen Detektorwerte.

Man erkennt sofort eine Reihe von Parallelen zur Motorsteuerung. So ist auch hier die Erkennung der Detektoren auf die Überprüfung, ob ein angegebener

Detektor tatsächlich angeschlossen ist, beschränkt. Die Einrichtung befasst sich bei den Detektoren mit der Anpassung einiger Detektorkennwerte an die vorliegende Menge der Roentgenstrahlung.

Bei den Detektoren sind weniger Statusinformationen zu verwalten, darunter die Information, ob sich ein Detektor gerade in einer Messphase befindet oder ob bereits wieder valide Daten vorliegen und ob die Messung mit oder ohne Ton erfolgen soll.

Auch bei den Detektoren ist es möglich, das Programm eingeschränkt ohne angeschlossene Detektoren zu benutzen. Die Einschränkung besteht darin, dass nicht alle der nachfolgend beschriebenen Detektoren simuliert werden können.

Die Detektoren werden anhand der Anzahl der erfassten Dimensionen klassifiziert. Die einfachste Form stellt ein Zählrohr dar, das nur die Menge der eintreffenden Roentgen-Strahlen erfasst, jedoch die Richtung oder Position unbeachtet lässt. Daher wird diese Detektorenart auch als nulldimensionale Detektoren bezeichnet. Die nächste Klasse von Detektoren erfasst bereits in einer Dimension den Ursprung der auftreffende Roentgen-Strahlung. Diese Detektoren heißen auch PSDs, was für Position Sensitive Detector steht. Als drittes wurde am Institut für Physik eine Kamera erworben, die in der Lage ist, die auftreffende Roentgen-Strahlung als zweidimensionales Feld zu erfassen. Diese wurde allerdings bisher noch nicht in das XCTL-Programm integriert.

Ablaufsteuerung. Ein Benutzer des XCTL-Programms ist in der Lage, bestimmte Abläufe in den Anwendungsfällen über eine externe Makro-Datei individuell zu konfigurieren, indem er unter Anderem Parameter wie Motor-Positionen oder Speicher-Dateien festlegt.

Messprozess durchführen. Dieser Use Case stellt das abstrakte Gerüst für die Hauptanwendungen des XCTL-Systems dar. In allen fünf davon abgeleiteten Use Cases werden Teile der drei oben beschriebenen Use Cases, die über die *extend*-Relation mit der Messprozessdurchführung verbunden sind, verwendet. Es benutzt also jeder der fünf nachfolgend beschriebenen Anwendungsfälle Elemente der Motorsteuerung und der Detektornutzung und jeder dieser Use Cases kann über die Ablaufsteuerung an nutzerspezifische Voraussetzungen angepasst werden.

Topographie. Die Topographie stellt einen typischen und auch den ursprünglichen Anwendungsfall des XCTL-Programms dar. Bei der Topographie werden mit den von Kristallproben reflektierten Roentgen-Strahlen Fotoplatten belichtet, deren spätere Auswertung Aussagen über die Kristallgüte ermöglicht. Hierbei wird zunächst die reflektierte Strahlung durch eine genaue Justage der Probe auf ein Maximum gebracht. Während des späteren langwierigen Belichtungsprozesses kommt es zu wärmebedingten Verformungen der Apparatur, die vom Programm automatisch ausgeglichen werden.

Diffraktometrie/Reflektometrie. Die Diffraktometrie und die Reflektometrie haben für die Nutzung des XCTL-Systems einen genau so hohen Stellenwert wie die Topographie. Anders als bei der Topographie wird bei Diffraktometrie und Reflektometrie in mehreren aufeinanderfolgenden Belichtungsphasen jeweils nur ein Teil der Probe bestrahlt. Dem Programm obliegt dabei wieder die automatische Positionierung der Probe bzw. des verwendeten Detektors an Hand zuvor gewählter Werte.

Manuelle Justage. Die Menge der von der Probe reflektierten Roentgenstrahlung ist von der Position der Probe relativ zur einfallenden Roentgenstrahlung abhängig. Daher muss die Probe vor der Durchführung eines Messprozesses zunächst mit Hilfe von Stellmotoren in die richtige Lage gebracht, also justiert werden. Diese Positionierung erfolgt durch manuelle Steuerung der Motoren und die parallele audiovisuelle Kontrolle der Roentgenmesswerte durch den Anwender. Zur Gütekontrolle steht als Hilfsmittel die weiter unten beschriebene Messung der Halbwertsbreite zur Verfügung.

Automatische Justage. Die Automatische Justage erfüllt die gleiche Aufgabe, wie die Manuelle Justage, allerdings ist hier die Suche der optimalen Position der Probe durch ein softwaregesteuertes Gradientenverfahren automatisiert.

Halbwertsbreite messen. Der Begriff Halbwertsbreite bezieht sich auf die grafische Darstellung der Abhängigkeit zwischen der Menge der von der Probe reflektierten Roentgenstrahlung und der Position eines speziellen Stellmotors.

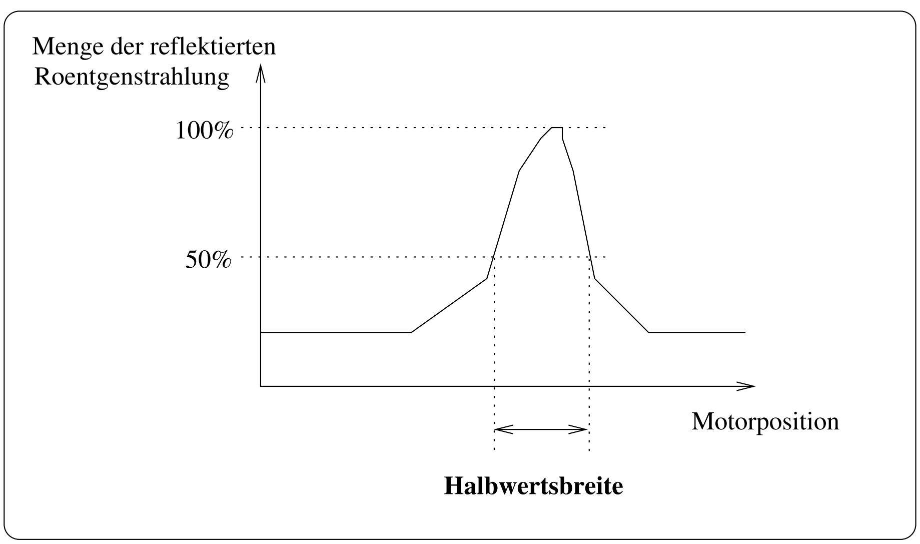

Abbildung 5: Grafische Veranschaulichung der Halbwertsbreite

Sie stellt ein Maß für die Güte der Positionierung dar: je kleiner die Halbwertsbreite, desto besser die Einstellung der Probe. Die Ermittlung der Halbwertsbreite erfolgt ebenfalls automatisiert durch die Software.

Repräsentation und Darstellung der Messdaten. Hierbei ist für den Benutzer von Interesse, in welcher Art und Weise er die Detektormessdaten dargeboten bekommt und in welcher Form diese Daten gesichert werden können. Die Bedürfnisse des Anwenders bezüglich Darstellung und Sicherung der Messdaten werden auch von der Art der eingesetzten Detektoren bestimmt.

Online-Hilfe. Der Sinn und die Aufgabe einer Online-Hilfefunktion sind unmittelbar einsichtig. Allerdings ist sie im XCTL-Programm nur ansatzweise implementiert, so dass sie vor allem wegen ihrer gravierenden Unvollständigkeit kaum nutzbar ist.

Allgemeine Einstellungen. Im Rahmen der allgemeinen Einstellungen kann der Nutzer des Programms beispielsweise seinen Namen, das Probenmaterial, den untersuchten Reflex und einige weitere Angaben in der Initialisierungsdatei vermerken lassen.

3.1.3 Analyse der Use Cases bezüglich ihrer Verwendbarkeit für die Subsystembildung

Da das Use-Case-Diagramm die Nutzersicht auf das System darstellt, können die Subsysteme nicht direkt aus den Use Cases abgeleitet werden. Stattdessen muss untersucht werden, inwieweit die Inhalte der Quelldateien für die Durchführung bestimmter Anwendungsfälle geeignet sind. Dabei ist man auf die Dateinamen, die Namen von Klassen, Variablen, Typen etc. sowie auf Kommentare des Entwicklers angewiesen.

So erhält man auch einen anfänglichen Überblick über das System. Im Ergebnis dieser ersten Analysen war festzustellen, dass einige Use Cases direkt ein Subsystem bilden können, einige Use Cases gar nicht als Subsysteme realisiert wurden und einige nur in erweiterter oder gekürzter Form. Im nachfolgenden werden die in Abbildung 4 aufgeführten Use Cases dahingehend klassifiziert, ob sie einem Subsystem im XCTL-Programm entsprechen.

Motorsteuerung. Die Motorsteuerung wurde auch vom Entwickler als eigenständiger Use Case angesehen und in einem eigenen Subsystem implementiert. Er stellte dem Anwender sogar eine extern als DLL (dynamic link library) nutzbare Bibliothek zur Verfügung, die der Abwicklung dieses Use Cases dient.

Detektornutzung. Auch bei der Nutzung der Detektoren wurde vom Entwickler eine DLL erstellt, so dass hier von einem eigenständigen Subsystem ausgegangen werden kann.

Ablaufsteuerung. Im Einsatzumfeld des XCTL-Programms existiert eine Reihe von wiederkehrenden Abläufen, die anhand ihrer Namen als Klassen in einer Quelldatei wiedergefunden werden konnten und somit die Grundlage für ein Subsystem bilden.

Messprozess durchführen. Dieser Use Case stellt als abstrakter Use Case nur eine logische Strukturierung der Anwendungsfälle zur Erhöhung der Übersicht dar. Er findet sich daher auch nicht in expliziter Form im XCTL-Programm.

Topographie. Die Topographie kann als Subsystem im XCTL-Programm vermutet werden, da einige Klassen existieren, die die Bezeichnung "Topography" im Namen tragen.

Diffraktometrie/Reflektometrie. Im Anwendungsbereich der Diffraktometrie und Reflektometrie existieren einige spezielle Fachbegriffe, die auch in verschiedenen Dateien des XCTL-Programms wieder aufzufinden waren. Dieser Use Case wurde daher in den Prozess der Subsystembestimmung aufgenommen.

Repräsentation und Darstellung der Messdaten. Die Darstellung der Detektormessdaten wird beim Einsatz des Programms auch mit dem Begriff Kurve in Verbindung gebracht. Da eine Klasse mit dem Namen "TCurve" existiert, die in einer eigenständigen Datei implementiert ist, wird auch dieser Use Case zur Subsystembildung hinzugezogen.

Online-Hilfe. Es existieren zwei Dateien, die die Bezeichnung "Help" in ihren Namen enthalten. Trotz ihres geringen Umfangs muss hier von einem potenziellen Subsystem ausgegangen werden.

Automatische Justage. Die Automatische Justage ist für die vorliegende Aufgabe der Wiedergewinnung von Subsystemen nicht von so großem Interesse, da sie erst im Projekt "Softwaresanierung" als nachträgliche Ergänzung entstand und nicht Bestandteil der Originalquellen ist.

Manuelle Justage. Die Manuelle Justage konnte dagegen bei näherer Betrachtung der Quellen leider nicht als ein vom Software-Entwickler geplantes echtes Subsystem identifiziert werden, da es sich im Wesentlichen nur um ein einzelnes Fenster der Oberfläche handelt und die Implementation in die allgemeine Ablaufsteuerung integriert wurde.

Halbwertsbreite messen. Für diesen Use Case gilt das gleiche wie für die Manuelle Justage. Er geht in der Implementation der Ablaufsteuerung auf.

Allgemeine Einstellungen. Auch dieser Use Case wird im Programm nicht als eigenes Subsystem realisiert, sondern findet sich teilweise verstreut in Dateien, die offensichtlich auch noch ganz andere Funktionen umfassen.

3.1.4 Die ermittelten Subsysteme

Aus den oben untersuchten Use Cases ergeben sich somit die folgenden sieben Subsysteme:

- Motorsteuerung,
- Detektornutzung,
- Repräsentation und Darstellung der Messdaten,
- Topographie,
- Diffraktometrie/Reflektometrie,
- Ablaufsteuerung und
- Online-Hilfe.

Dazu kommen noch drei weitere Subsysteme, die nicht direkt aus den Bedürfnissen des Anwenders abgeleitet werden können, jedoch aus Entwicklersicht sinnvoll erscheinen:

- Interaktion mit der Software,
- Interne Funktionalität und Allgemeine Definitionen und
- Windows-Ressourcen (Fenster-Definitionen).

Bei diesen Subsystemen zeigt sich, dass die Bildung von Teilsystemen nicht nur von Use Cases dominiert wird, sondern dass dafür auch andere Prinzipien, wie die Trennung von Benutzeroberfläche und inhaltlichen Komponenten zur Erhöhung der Portabilität, und betriebssystemspezifische Gegebenheiten relevant sind.

3.2 Analyse der Softwareentstehung

Die in Abschnitt 3.1.3 erwähnten Use Cases sollen möglichst direkt als Vorlagen für eine Subsystemzerlegung angewendet werden. Dafür muss jedoch zunächst geklärt werden, wie sauber eine solche Trennung erfolgen kann. Dabei spielen sicher eine Reihe von Faktoren eine Rolle, jedoch ist der Einfluss des Entwicklers bzw. der Entwickler unzweifelhaft von sehr großer Bedeutung. Insbesondere ist die Frage von Interesse, ob ein einzelner Entwickler oder eine Gruppe von Entwicklern mit der Erstellung des Systems beschäftigt war. Wenn ein Entwickler ein Projekt dieses Umfangs allein in Angriff nimmt, so muss man davon ausgehen, dass negative Auswirkungen auf die Qualität nahezu unvermeidlich sind.

Eine Gruppe von Entwicklern, die parallel an einem Projekt arbeiten, muss - am besten bereits im Vorfeld - klären, wer welche Aufgabe übernimmt. Dadurch ist die Gruppe gezwungen, das Projekt in abgegrenzte Teilaufgaben bzw. das Softwaresystem relativ sauber in Subsysteme zu zerlegen. Spätere Verletzungen dieser Unterteilung führen sicher zu heftigen Diskussionen, da ja dann ein Entwickler ungerechtfertigt in die Arbeit eines anderen eingreift. Wenn sich solche Vorfälle häufen, wird wohl idealerweise die Modulstruktur neu angepasst werden, so dass wieder eine klare Trennung erfolgen kann, oder es werden sich überschneidende Teilaufgaben gemeinsam bearbeitet, was wieder eher für die Qualität förderlich ist, da bekanntlich vier Augen mehr sehen als zwei. Dies entspricht auch den Erfahrungen, die mit dem so genannten Pair Programming im Rahmen des Extreme Programming (s. [12]) gemacht wurden.

Ein einzelner Entwickler dagegen muss auf eine Reihe dieser Dinge zunächst überhaupt nicht achten. Natürlich wird auch er sich eine Subsystemstruktur ausdenken, da er sonst von Anfang an zum Scheitern verurteilt ist.

Aber bereits bei der Frage der Namenskonventionen kann er seinen persönlichen Vorlieben freien Lauf lassen. Dies fördert erfahrungsgemäß die Wartbarkeit nicht immer, da in diesem Fall nur der ursprüngliche Entwickler um die Konvention weiß und er diese höchstwahrscheinlich nicht schriftlich formuliert hat, da er sich nie mit anderen Entwicklern um die konkrete Auslegung streiten musste.

Weiterhin wird sich ein einzelner Entwickler bei auftretenden Überschneidungen auch leichter über die vorgegebenen Modulgrenzen hinwegsetzen, da er sich (erneut als Einziger) in beiden Modulen auskennt und "weiß, was er tut". Auch die Schnittstellen zwischen den einzelnen Subsystemen werden bei einem Einzelentwickler tendenziell schlechter als bei einer Gruppe sein. Da bei Gruppenarbeit jeder für die Schnittstelle seines Subsystems verantwortlich ist, kann er sich bei allzu schlechter Implementation auch jede Menge Kritik seitens der anderen Entwickler einhandeln, so dass er naturgemäß mehr Arbeit in die Qualität der Schnittstelle stecken wird.

Ein einzelner Entwickler dagegen kann seine Schnittstellen jederzeit um eine beliebige Anzahl von Funktionen erhöhen, ohne dass ihm jemand Vorwürfe machen wird. Eventuell spart er sich sogar die Entwicklung separater Zugriffsfunktionen für interne Daten und macht einfach alle privaten Daten öffentlich zugreifbar, da er weiß, welche Variablen man von außen anfassen darf und welche nicht.

Das XCTL-Programm wurde überwiegend von einem einzelnen Entwickler erstellt. Leider sind auch viele der gerade beschriebenen Qualitätsmängel eingetreten, obwohl zum Beispiel die vorgefundene Namenskonvention durchaus plausibel ist und auch weitestgehend konsequent angewendet wurde. Eine Datei wurde von einem anderen Entwickler erstellt, der auch prompt alle von ihm verwendeten Variablen mit von dieser Konvention abweichenden Namen versehen hat.

3.3 Zuordnung der Quelldateien zu den Subsystemen

3.3.1 Übersicht über die ursprünglichen Quelldateien

In Tabelle 1 sind alle Quelldateien des ursprünglichen Systems mit ihrer Zeilenzahl in alphabetischer Reihenfolge und typsortiert aufgelistet.

Datei	LOC		Datei	LOC		Datei	LOC
Am9513.cpp	492		Kmpt1.c	686		M_steerg.h	547
Braunpsd.cpp	793		Am9513a.h	141		M_topo.h	77
C_layer.cpp	268		C_layer.h	33		M_xscan.h	336
Counters.cpp	2152		Comclass.h	241		Prkmpt1.h	111
Dlg_tpl.cpp	188		Comhead.h	224		Radicon.h	70
L_layer.cpp	526		Dfkisl.h	110		Rc_def.h	478
M_arscan.cpp	3105		Dlg_tpl.h	92		St_layer.h	39
M_curve.cpp	769		Help_def.h	8		Testdev.h	25
M_data.cpp	1874		Ieee.h	96		C_8x2.inc	57
M_device.cpp	386		L_layer.h	47		Pcl_830.inc	84
M_dlg.cpp	814		M_curve.h	132		Stoe_psd.inc	22
M_layer.cpp	547		M_data.h	148			
M_main.cpp	1950		M_devcom.h	264		Counters.rc	80
M_scan.cpp	1223		M_devhw.h	164		Main.rc	1450
M_steerg.cpp	3106		M_dlg.h	66		Motors.rc	214
M_topo.cpp	714		M_layer.h	309		Scanner.rc	86
Motors.cpp	3826		M_lscan.h	135		Splib.rc	52
St_layer.cpp	159		M_motcom.h	210		Sphelp.rtf	185
Testdev.cpp	145		M_mothw.h	313		Sphelp.hpj	30
Kisl1.c	634		M_psd.h	221			

Tabelle 1: Die Quelldateien des XCTL-Programms

In den nachfolgenden Abschnitten werden den einzelnen Subsystemen die entsprechenden Quelldateien zugeordnet. Dabei wird kurz auf die Dateiinhalte eingegangen, da diese ausschlaggebend für die Zuordnung waren. Es wird ebenfalls kurz auf die Probleme eingegangen, die einzelne Dateien bei ihrer Einordnung verursachten.

3.3.2 Motorsteuerung

Dem Subsystem Motorsteuerung sind zwei reine Implementationsdateien zuzuordnen:

- M_layer.cpp und
- Motors.cpp.

Das Motoren-Subsystem stellt die am klarsten abgegrenzte Komponente des XCTL-Programms dar. Die Zuordnung dieser beiden Dateien konnte eindeutig erfolgen, da sich bereits von Anfang an Studenten des Seminars mit diesem Teilsystem auseinandergesetzt haben, wodurch die dazu gehörenden Quelldateien sehr ausführlich und verständlich kommentiert und ihre Inhalte dokumentiert wurden.

Leider enthalten die beiden obigen Dateien nicht die komplette Funktionalität des Teilsystems. Der Entwickler hat im Übermaß von der Möglichkeit Gebrauch gemacht, dass auch in Headerdateien Implementationselemente auftreten dürfen. Daher müssen hier auch explizit die Headerdateien des Subsystems aufgeführt werden, die eigentlich dem Abschnitt über die Schnittstellen vorbehalten bleiben sollten. In folgenden Headerdateien finden sich Implementationsdetails des Motorsubsystems:

- Motcom.h und
- Mothw.h.

3.3.3 Detektornutzung

Die Detektornutzung beinhaltet die meisten Einzeldateien. Dazu gehören zum einen die beiden von externen Entwicklern implementierten Dateien

- Kisl1.c und
- Kmpt1.c,

dann die beiden allgemeineren Dateien

- C_layer.cpp und
- Counters.cpp

sowie die einzelnen Detektortypen zugeordneten Dateien

- Am9513.cpp,
- Braunpsd.cpp und
- Testdev.cpp.

Auch mit diesem Teilsystem sind schon seit längerer Zeit einige Studenten befasst, allerdings ist die Abgrenzung nicht mehr so klar vornehmbar, wie bei den Motoren. In der hier vorliegenden Variante wurde eine sehr enge Auswahl getroffen, indem in dieses Subsystem nur Dateien aufgenommen wurden, die ausschließlich der Nutzung der Detektoren dienen.

Es ist festzustellen, dass die Implementation des Detektorsubsystems im Vergleich zum Motorsubsystem mit einer feineren Unterteilung erfolgte. Im Motorsubsystem sind die einzelnen Motortypen alle zusammen in der Datei Motors.cpp realisiert, beim Detektorsubsystem dagegen erhielten die speziellen Detektoren eigene Implementationsdateien.

Auch bei diesem Teilsystem sind einige Implementationsdetails in den folgen Headerdateien versteckt:

- M_devcom.h,
- M_devhw.h und
- M_psd.h.

3.3.4 Repräsentation und Darstellung der Messdaten

Mit der internen und externen Datendarstellung befassen sich die folgenden Dateien:

- M_curve.cpp und
- M_data.cpp.

In M_curve.cpp wird mit zwei Klassen die grundlegende Funktionalität zur internen Repräsentation und Speicherung der Detektormessdaten implementiert. M_data.cpp schließlich befasst sich mit der Darstellung der Messwerte als so genannte Rocking-Kurven, in denen zu den verschiedenen Positionen eines Motors die entsprechenden Messwerte zugeordnet werden, bzw. als Bitmaps, in denen die Intensitätsverteilung mit verschiedenen Farbwerten dargestellt wird. Auch hier sind die Definitionen einiger kleinerer Funktionen in den Headerdateien

- M_curve.h und
- M_data.h

enthalten.

3.3.5 Topographie

Das Subsystem Topographie wurde vom Entwickler in einer einzelnen Datei implementiert:

- M_topo.cpp.

In den dort implementierten Klassen wird im Wesentlichen die gesamte Funktionalität der Topographie bereitgestellt. Diese wird noch in die Verwaltung der Parameter, die Ablaufsteuerung, die im wesentlichen mit Hilfe der später aufgeführten Funktionen aus M_steerg.cpp realisiert wird, und die Benutzerschnittstelle unterteilt.

3.3.6 Diffraktometrie/Reflektometrie

Die Diffraktometrie/Reflektometrie wurde in den folgenden Dateien realisiert:

- M_scan.cpp und
- M_arscan.cpp.

Dazu kommt noch die Datei

- St_layer.cpp,

die, wie aus ihrem fragmentarischen Inhalt ersichtlich wird, für die Implementation der Einbindung der CCD-Kamera vorgesehen ist.

M_scan.cpp enthält im Wesentlichen die Funktionalität zur Verwendung der nulldimensionalen Detektoren, in M_arscan.cpp wird die Nutzung der PSDs realisiert.

In der Headerdatei

- M_xscan.h

finden sich zusätzliche Implementationselemente.

3.3.7 Ablaufsteuerung

In der Datei

- M_steerg.cpp

findet sich die Implementation eines globalen Objekts, das die interne Ablaufsteuerung sicherstellt. Ebenso werden in M_steerg.cpp einzelne Abläufe, die in verschiedenen Anwendungssituationen wiederholt verwendet werden, in Form einer Klassenhierarchie implementiert.

3.3.8 Online-Hilfe

Die Ansätze für eine Online-Hilfe finden sich in den Dateien

- Sphelp.rtf und
- Sphelp.hpj,

deren Inhalte durch Formatvorgaben seitens des Betriebssystems bestimmt werden.

3.3.9 Interaktion mit der Software

Zu diesem Subsystem gehören drei Dateien:

- Dlg_tpl.cpp,
- M_dlg.cpp und
- M_device.cpp.

In der Datei Dlg_tpl.cpp wird eine allgemeinere Klasse zur Interaktion des Benutzers mit dem XCTL-Programm definiert, die Datei M_dlg.cpp implementiert einige speziellere Fenster. M_device.cpp realisiert das Fenster für die Messwertanzeige und Bedienung der Detektoren. Erneut enthalten die Headerdateien

- Dlg_tpl.h und
- M_dlg.h

einige kleinere Implementationsdetails.

3.3.10 Interne Funktionalität und Allgemeine Definitionen

Unter den restlichen Quelldateien sind die folgenden zwei für die Implementation von internen Funktionen zuständig:

- M_main.cpp und
- L_layer.cpp.

M_main.cpp beinhaltet einen Großteil der Funktionalität, die zur Ausführung des Programms unter Microsoft Windows nötig ist, wie zum Beispiel die allgemeine Verarbeitung der Fenster-Nachrichten. Außerdem befasst sich dieser Teil mit der initialen Einrichtung des Programms bei seinem Start.

L_layer.cpp stellt eine Reihe von Funktionen zur Verfügung, die von vielen Fenstern und Anwendungen des XCTL-Programms genutzt werden, darunter beispielsweise auch die Funktionen, die eine allgemein nutzbare Statusleiste am unteren Rand der Programmoberfläche implementieren.

3.3.11 Windows-Ressourcen (Fenster-Definitionen)

Dieses eher entwicklungstechnisch motivierte Subsystem bündelt die Beschreibungen des Erscheinungsbildes aller Fenster der XCTL-Oberfläche. Dazu gehören die Dateien:

- Motors.rc,
- Counters.rc,
- Main.rc,
- Splib.rc und
- Scanner.rc.

Die in diesen Dateien definierten Bezeichner, die zum Beispiel Fensterschaltknöpfe und ähnliches identifizieren, werden dem XCTL-Programm über die von der Entwicklungsumgebung automatisch erstellte Headerdatei

- Rc_def.h

bekannt gemacht.

Subsystem	Datei		Subsystem	Datei
Motor-	M_layer.cpp		Repr. u. Darst.	M_curve.cpp
steuerung	Motors.cpp		der Messdaten	M_data.cpp
	M_layer.h			M_curve.h
	M_motcom.h			M_data.h
	M_mothw.h		Topographie	M_topo.cpp
	Ieee.h			M_topo.h
	C_8x2.inc		Diffraktometrie/	M_arscan.cpp
Detektor-	Am9513.cpp		Reflektometrie	M_scan.cpp
nutzung	Braunpsd.cpp			St_layer.cpp
	C_layer.cpp			M_lscan.h
	Counters.cpp			M_xscan.h
	Testdev.cpp			St_layer.h
	Kisl1.c		Ablaufsteuerung	M_steerg.cpp
	Kmpt1.c			M_steerg.h
	Am9513a.h		Online-Hilfe	Sphelp.rtf
	C_layer.h			Sphelp.hpj
	Dfkisl.h			Help_def.h
	M_devcom.h		Interne Fktn.	L_layer.cpp
	M_devhw.h			M_main.cpp
	M_psd.h			L_layer.h
	Prkmpt1.h		Windows-Resrc.	Counters.rc
	Radicon.h			Main.rc
	Testdev.h			Motors.rc
	Pcl_830.inc			Scanner.rc
	Stoe_psd.inc			Splib.rc
Interaktion mit	Dlg_tpl.cpp			Rc_def.h
der Software	M_device.cpp			
	M_dlg.cpp			
	Dlg_tpl.h		Nicht zugeordnet	Comhead.h
	M_dlg.h			Comclass.h

Tabelle 2: Vorläufige Quelldateiaufteilung nach ihrer Zuordnung zu den Subsystemen

3.3.12 Darstellung der Zuordnung der Quellen zu den einzelnen Subsystemen

Tabelle 2 auf Seite 28 zeigt die ermittelte Zugehörigkeit der Quelldateien zu den Subsystemen. Auf die nicht zugeordneten Dateien wird im nachfolgenden Abschnitt eingegangen.

3.4 Bearbeitung der Dateistruktur

Im vorliegenden Fall konnten die Implementationsdateien sehr gut den einzelnen Subsystemen zugeordnet werden. Daher konnten diese auch auf entsprechend benannte Unterverzeichnisse verteilt werden. Der Versuch, die Headerdateien ebenfalls dergestalt zuzuordnen, wurde jedoch von der vorgefundenen Situation zunächst verhindert, da die Headerdateien ihre jeweilige Entstehung offenbar keinem Konzept sondern beliebigen Zufällen zu verdanken scheinen. Die Situation wurde dadurch verschärft, dass sich die Headerdateien zum Teil untereinander inkludierten, inhaltlich überschnitten oder Inhalte aus verschiedenen Teilsystemen enthielten.

Insbesondere das letzte Problem trat massiv in den Dateien

- Comhead.h
- Comclass.h und
- L_layer.h

auf. Comhead.h enthielt eine Reihe von Aufzählungstypen, die zwar strukturell bzw. syntaktisch ähnlich waren, jedoch inhaltlich verschiedenen Subsystemen zugeordnet werden mussten. Daher wurde die Datei Comhead.h aufgelöst und ihr Inhalt auf fünf andere Dateien verteilt. Die programmweit genutzten Makro-Definitionen wurden in der Datei

- Evrythng.h

gesammelt. Danach wurden die Aufzählungstypen den entsprechenden Subsystemen zugeordnet und auf die folgenden Dateien verteilt:

- Dettypes.h,
- Mottypes.h,
- Datsvtps.h und
- Scntypes.h.

Die Datei Comclass.h enthielt im Wesentlichen nur zwei allgemein benötigte Klassen, die keine eigene Datei rechtfertigten. Der Inhalt dieser Datei wurde nach Evrythng.h verlagert. In der Datei L_layer.h fanden sich sowohl etliche Prototypen von im gesamten Programm benutzten Funktionen, als auch die automatisch generierten Informationen aus der Datei Splib.rc. Diese wurden nach Rc_def.h verschoben, die Funktionsprototypen befinden sich jetzt in der Datei Evrythng.h.

Bei der Zerlegung dieser Headerdateien fiel auf, dass sich das XCTL-Programm nur übersetzen ließ, wenn die Include-Direktiven in einer ganz bestimmten Reihenfolge erfolgten. Es stellte sich heraus, dass in der Datei L_layer.h nicht nur Prototypen von in L_layer.cpp implementierten Funktionen auftraten, sondern auch Prototypen anzutreffen waren, die in ganz anderen Dateien sowohl definiert als auch benutzt wurden. Diese wurden an den Anfang der betroffenen Dateien gebracht. Schließlich war festzustellen, dass die Datei Pcl_830.inc überflüssig war, da sie von keiner anderen Datei inkludiert wurde. Diese Datei wurde daher aus den Quellen entfernt.

Nach dieser Zerlegung wurden etliche hierarchische Inklusionen entfernt, da sie in der vorliegenden Version nur selten inhaltlich motivierbar waren. Dadurch stieg zwar die Zahl der in die einzelnen Implementationsdateien inkludierten Headerdateien, doch stieg damit auch die Menge der verfügbaren Informationen über die Abhängigkeiten zwischen den Subsystemen und den jeweiligen Dateien. Dabei zeigte sich nur noch deutlicher, dass die ursprüngliche Headerdateientwicklung beinahe vollständig ohne Konzept erfolgt war. Daher sollten die aktuellen Headerdateien in nachfolgenden Arbeiten vollständig durch neue ersetzt werden, da sie ihre eigentliche Aufgabe, die Schnittstellen zwischen den Subsystemen zu beschreiben, nicht erfüllen.

Die Headerdateien wurden schließlich analog den Implementationsdateien in subsystemspezifische Unterverzeichnisse gebracht. Tabelle 3 auf Seite 31 zeigt die so entstandene Verteilung der einzelnen Dateien. In der Tabelle 4 auf Seite 32 ist die Zuordnung der Subsysteme zu den Bezeichnungen der Unterverzeichnisse aufgeführt.

3.5 Ermittlung und Beschreibung der Schnittstellen

3.5.1 Allgemeine Vorgehensweise bei der Bestimmung der Schnittstellen

Um die Schnittstellen eines Teilsystems zu bestimmen, müssen zunächst für jedes Subsystem die von außen zugänglichen Sprachelemente festgestellt werden, da prinzipiell alle Entitäten[2] einer Sprache Teil der Schnittstelle sein können.

[2] Im Falle von C/C++: Konstanten, Variablen, Typen, Funktionen, Klassen und ihre Memberfunktionen

Subsystem	Datei		Subsystem	Datei
Motor-	M_layer.cpp		Repr. u. Darst.	M_curve.cpp
steuerung	Motors.cpp		der Messdaten	M_data.cpp
	Ieee.h			Datsvtps.h
	M_layer.h			M_curve.h
	M_motcom.h			M_data.h
	M_mothw.h		Topographie	M_topo.cpp
	Mottypes.h			M_topo.h
	C_8x2.inc		Diffraktometrie/	M_arscan.cpp
Detektor-	Am9513.cpp		Reflektometrie	M_scan.cpp
nutzung	Braunpsd.cpp			St_layer.cpp
	C_layer.cpp			M_lscan.h
	Counters.cpp			M_xscan.h
	Testdev.cpp			St_layer.h
	Kisl1.c		Ablaufsteuerung	M_steerg.cpp
	Kmpt1.c			M_steerg.h
	Am9513a.h			Scntypes.h
	C_layer.h			
	Dettypes.h		Online-Hilfe	Sphelp.rtf
	Dfkisl.h			Sphelp.hpj
	M_devcom.h			Help_def.h
	M_devhw.h			
	M_psd.h		Interne Fktn.	L_layer.cpp
	Prkmpt1.h			M_main.cpp
	Radicon.h			Evrythng.h
	Testdev.h			
	Stoe_psd.inc		Windows-Resrc.	Counters.rc
Interaktion mit	Dlg_tpl.cpp			Main.rc
der Software	M_device.cpp			Motors.rc
	M_dlg.cpp			Scanner.rc
	Dlg_tpl.h			Splib.rc
	M_dlg.h			Rc_def.h

Tabelle 3: Abgeschlossene Dateiaufteilung nach Zerlegung einiger Header

Subsystem		Name des Unterverzeichnisses
Motorsteuerung	Implementation	neustrukt\motrstrg\
	Header	neustrukt\include\motrstrg\
Detektornutzung	Implementation	neustrukt\detecuse\
	Header	neustrukt\include\detecuse\
Interaktion mit der	Implementation	neustrukt\swintrac\
Software	Header	neustrukt\include\swintrac\
Repräsentation und	Implementation	neustrukt\datavisa\
Darstellung der Messdaten	Header	neustrukt\include\datavisa\
Topographie	Implementation	neustrukt\topogrfy\
	Header	neustrukt\include\topogrfy\
Diffraktometrie/	Implementation	neustrukt\difrkmty\
Reflektometrie	Header	neustrukt\include\difrkmty\
Ablaufsteuerung	Implementation	neustrukt\workflow\
	Header	neustrukt\include\workflow\
Online-	Implementation	neustrukt\help\
Benutzerdokumentation	Header	neustrukt\include\help\
Interne Funktionalität und	Implementation	neustrukt\internls\
Allgemeine Definitionen	Header	neustrukt\include\internls\
Windows-Ressourcen	Implementation	neustrukt\winresrc\
(Fenster-Definitionen)	Header	neustrukt\include\winresrc\

Tabelle 4: Zuordnung der Subsysteme zu den einzelnen Unterverzeichnissen

Im Idealfall sind dem Nutzer eines Subsystems nur solche Funktionen und Aufzählungstypen zugänglich, die für die Darbietung eines kompletten und zusammengehörigen Konzepts nötig sind und dabei keine Implementationsdetails verraten (s.a. [13]).

Die vorliegende Situation war jedoch weit vom Ideal entfernt, da viele der Klassenmethoden unnötig als public (also frei verfügbar) deklariert worden sind. Also blieb als Ausweg nur, für die in einer Datei auftretenden Funktionen und Klassen festzustellen, ob sie in anderen Subsystemen benutzt werden. Im vorliegenden Fall geschah dies mit dem spartanischen aber leistungsfähigen UNIX-Standard-Werkzeug *grep*. Die Headerdateien waren dabei nur geringfügig hilfreich, die meisten Informationen enthielten tatsächlich die Implementationsdateien, da ihre jeweilige Existenz auch weniger vom Zufall bestimmt schien, als die der Headerdateien.

Dabei stellte sich unter anderem heraus, dass der Entwickler des XCTL-Programms die durch die Use Cases erfolgte Trennung zwischen den Subsystemen, die verschiedene Anwendungen bearbeiten, und dem Subsystem, das sich mit der Oberfläche beschäftigt, nur ansatzweise selbst vollzogen hatte. Allerdings können aufgrund der Objektorientiertheit dieses Teilsystems die falsch zugeordneten Oberflächenelemente sicher lokalisiert werden, da alle Fenster Ableitungen einer der beiden Basisklassen aus dem Oberflächenteilsystem sind. Jedoch können diese nicht einfach umgelagert werden, da der Entwickler Oberflächenbedienung und echte Funktionalität in den gleichen Klassen implementiert hat.

3.5.2 Die konkreten Subsystem-Schnittstellen

Motorsteuerung. Das Subsystem für die Motorsteuerung verfügt über eine vergleichsweise gut entwickelte Schnittstelle. Allerdings existiert diese Schnittstelle sowohl in Form einer C++-Klasse, als auch als reines C-Interface, das als Wrapper um die C++-Implementation realisiert wurde. Dieses Interface wurde im Projekt "Software-Sanierung" bereits im Rahmen einer Studienarbeit ausführlich dokumentiert (s. [14]).

Die zusätzliche C-Schnittstelle wurde den spärlichen Aufzeichnungen des Entwicklers zufolge für entwickelnde Anwender, die kein C++ beherrschen, zur externen Nutzung des Motorenteilsystems eingeführt. Interessanterweise wird auch bei der Motorennutzung innerhalb des Programms das C-Interface bevorzugt, da es klarer aufgebaut und einsichtiger anzuwenden ist.

Detektornutzung. Auch für die Detektoren war eine solche C-Schnittstelle vorgesehen. Allerdings ist diese bei weitem nicht so ausgereift wie die Motorenschnittstelle. Daher erfolgt programmintern der Zugriff über die Klasse TDList, in der alle in der Initialisierungsdatei aufgeführten und angeschlossenen Detektoren verwaltet werden, sowie über die jeweilige Detektorenklasse, die die jeweilige spezielle Funktionalität realisiert.

Dies führt zu der eigenwilligen Situation, dass in Programmteilen, die die Detektoren und die Motoren verwenden, sowohl Zugriffe auf C++-Objekte als auch Aufrufe von C-Funktionen für die Kommunikation mit diesen beiden, eigentlich sehr ähnlich implementierten Subsystemen erfolgen.

Datenrepräsentation und -darstellung. In diesem Subsystem wurde die Schnittstelle ebenfalls in Form mehrerer Klassen realisiert, die für die anderen Subsysteme die nötige Funktionalität zur Bearbeitung und Verwaltung von Scan-Kurven (M_curve.cpp) bzw. Bitmaps (M_data.cpp) kapseln.

In der Datei M_data.cpp befindet sich auch eine Klasse, die ein Darstellungsfenster realisiert. Diese Klasse sollte eigentlich eher im Teilsystem Interaktion mit der Software zu finden sein.

Topographie. Die Topographie müsste eigentlich über eine Schnittstelle verfügen, durch die die Funktionalität von den Fensterklassen getrennt ist. Allerdings sind auch hier die Fenster nicht im entsprechenden Subsystem zu finden, sondern werden vor Ort realisiert und dann auch mit der Funktionalität komplett verbunden.

Diffraktometrie/Reflektometrie. Das gleiche Bild wie bei der Topographie bietet sich auch bei der Diffraktometrie/Reflektometrie. Alle hier implementierten Klassen sind von den in Dlg_tpl.cpp definierten Fensterklassen abgeleitet und vollständig von der eigentlichen Funktionalität untersetzt.

Ablaufsteuerung. Die Datei M_steerg.cpp, in der die allgemeine Ablaufsteuerung realisiert ist, enthält eine größere Sammlung von Steuerungsabläufen, die verschiedene wiederkehrende Aufgaben im Zusammenspiel von Detektoren und Motoren abdecken. Diese Sammlung ist in Form mehrerer C++-Klassen, die eine gemeinsame Basisklasse haben, realisiert und kann über die entsprechenden Klassenmethoden genutzt werden.

Online-Hilfe. Die Hilfsfunktion ist mit von Microsoft Windows angebotenen Möglichkeiten entwickelt und stellt daher genau genommen ein separates System dar. Daher und auch wegen des eher marginalen Umfangs der vorhandenen Inhalte existiert, abgesehen von einigen definierten Bezeichnern, die der betriebssystemspezifischen Funktion WinHelp als Parameter zu Verfügung gestellt werden, auch keine echte Schnittstelle, über die das XCTL-Programm auf die Hilfe zugreifen könnte.

Interaktion mit der Software. Dieser Teil des XCTL-Programms stellt zunächst in Dlg_tpl.cpp zwei Klassen zur Verfügung, von denen die jeweils benötigten Fensterklassen abgeleitet werden können. Einige solcher Fensterklassen werden in der Datei M_dlg.cpp definiert und können dann systemweit genutzt werden. Hier zeigt sich ein echtes Manko in der Dekomposition des Systems. Ursprünglich scheint der Entwickler vorgesehen zu haben, dass entweder alle Fensterklassen in der Datei M_dlg.cpp implementiert werden oder dass jedes Fenster eine eigene Datei erhält, wie es mit dem Detektorfenster in M_device.cpp geschehen ist. Diese Entscheidung wurde jedoch nicht getroffen. Stattdessen wurden die Implementationen der Fensterklassen auf die einzelnen Subsysteme verteilt, was zum Beispiel eine Überarbeitung der Oberfläche schwierig erscheinen lässt.

Interne Funktionalität und Allgemeine Definitionen. Dieses Subsystem enthält Dateien, die einer auf Use Cases basierenden Zuordnung nicht zugänglich waren. Daher haben diese Dateien auch relativ verschiedene Aufgaben. M_main.cpp stellt mit den dafür nötigen (und vorgegebenen) Funktionen die Verbindung zum Betriebssystem her. L_layer.cpp bietet eine Reihe von globalen Funktionen an, die im gesamten XCTL-Programm genutzt werden. Zusätzlich dazu implementiert L_layer.cpp das Fenster mit den allgemeinen Informationen zum Programm, das eigentlich auch in das Subsystem Interaktion mit der Software gehört.

Windows-Ressourcen (Fenster-Definitionen). Das Interface dieses Moduls erschöpft sich im Wesentlichen in der Datei Rc_def.h. Diese ist aufgrund der Tatsache, dass sie automatisch erstellt wird, sehr unübersichtlich. Zur Benutzung des Windows-Ressourcen-Subsystems ist daher die Verwendung des Ressourceneditors der jeweiligen Entwicklungsumgebung zu empfehlen.

3.6 Kritik der festgestellten Systemstruktur

3.6.1 Beschreibung und Klassifikation vorgefundener ungünstiger Designentscheidungen

Das vorliegende System ist von einigen Designentscheidungen geprägt, die sich für die Systemstruktur als ungünstig erwiesen haben. Darunter sind sowohl einige kleinere Design-Probleme festzustellen, die für Inkonsistenzen bei der Aufteilung der Subsysteme sorgen, als auch zum Teil schwer wiegende Design-Fehler anzutreffen, die für unnötige Abhängigkeiten und Überschneidungen zwischen den Subsystemen verantwortlich sind.

Implementationsdetails in Headerdateien. Zu den leichteren Problemen zählt die Tatsache, dass in nahezu allen Headerdateien Implementationsdetails anzutreffen sind. Dies führt allerdings nur zu einer geringfügig verschlechterten Lesbarkeit der Headerdateien, da die Klassendeklarationen durch die eingefügten Funktionskörper der so genannten Inline-Funktionen in die Länge gezogen werden.

Elemente verschiedener Teilsysteme in einer Headerdatei. Ein ähnliches Problem wie das eben beschriebene stellt die gemeinsame Aufführung aller Bezeichner von Oberflächenfenstern in einer einzigen Headerdatei dar, durch die eine schnelle Zuordnung von bestimmten oberflächenbezogenen Bezeichnern zu den Teilsystemen unnötig erschwert wird.

Fehlendes Konzept für die Headerdateien. Das eigentlich schwerwiegendste Designproblem der Headerdateien ist genau genommen die vollständig fehlende Strukturierung, also gewissermaßen die Abwesenheit eines Designs. Das führte auch zu sehr kritikwürdigen Aufteilungen von Inhalten auf die einzelnen Dateien, die teilweise nur nach syntaktischen Gesichtspunkten nicht jedoch gemäß den Subsystemen erfolgten.

Globale Veröffentlichung lokaler Elemente. Auch wurden in einer Headerdatei etliche Funktionsprototypen, die nur lokal von Bedeutung sind, global verfügbar gemacht, was zahlreiche unmotivierte Abhängigkeiten zwischen den einzelnen Headerdateien zur Folge hatte. Dies stellt somit bereits einen Design-Fehler dar, der zwar spürbare Auswirkungen hat, jedoch vergleichsweise leicht behebbar ist.

Unklare Zuordnungen in Implementationsdateien. Ein weiterer Designfehler ist die Implementation subsystemfremder Klassen in einigen Implementationsdateien. Daraus folgten im vorliegenden Fall nicht zwangsläufig unnötige Abhängigkeiten zwischen den Subsystemen, es ist allerdings für die Benennung aller Elemente eines Subsystems hinderlich, wenn einzelne Teile in den Dateien eines ganz anderen Subsystems aufzufinden sind.

Mangelhafte Trennung von Oberfläche und Funktionalität. Der im vorliegenden Softwaresystem gefundene schwerste Designfehler ist die mangelhafte, häufig sogar fehlende Abgrenzung der Benutzeroberfläche von der Funktionalität in verschiedenen Subsystemen. Dieser Designfehler führt einerseits zu Schwierigkeiten bei der Zuordnung der Dateien zu den Subsystemen und ist andererseits auch nur schwer behebbar.

3.6.2 Bewertung der Subsystemunterteilung und Schnittstellenbildung

Aus der Anzahl und Art der in Abschnitt 3.6.1 aufgeführten Probleme wird verständlich, dass weder die Subsystemunterteilung noch die Schnittstellenbildung des XCTL-Systems als gut bezeichnet werden kann.

Einen besonders unschönen Aspekt stellt die fehlende Abgrenzung des Oberflächensubsystems von den anderen Bereichen des Programms dar, da dadurch in fast allen Subsystemen die Kopplung erhöht und die Kohäsion verringert wird, was natürlich den allgemeinen Prinzipien einer vernünftigen Modularisierung zuwider läuft. Die Ursachen dieses Fehlers und auch der verschiedenen anderen kleineren Unzulänglichkeiten bei der Umsetzung der Systemstruktur sind wohl in dem nur kurzfristig effektiven Bestreben des Entwicklers nach Zeitersparnis und in der Tatsache, dass das Programm sich in ständiger Weiterenwicklung befand, zu suchen.

Es ist klar, dass unter der nur unvollständig ausgeführten Strukturierung des Programms in Subsysteme auch die Qualität der Schnittstellen zwischen den einzelnen Teilsystemen zu leiden hat. Neben dem fehlenden Gesamtkonzept für die Ausführung der Schnittstellen, das sich im katastrophalen Zustand der Headerdateien spiegelt, sind insbesondere die Inkonsistenzen bei der Unterteilung in Subsysteme für die unnötig umfangreichen Schnittstellen verantwortlich.

Man muss allerdings auch anerkennen, dass die logische Bildung der Subsysteme durchaus vernünftig motiviert ist, so dass die Systemstruktur nicht im Ganzen als schlecht bewertet werden muss. Wäre die in Ansätzen vorgefundene Subsystemunterteilung auch vollständig und konsistent vorgenommen worden, hätte in Verbindung mit einem klaren Schnittstellenkonzept ein gut strukturiertes System entstehen können.

Zusammenfassend lässt sich also feststellen, dass die geplante Aufteilung des XCTL-Programms in einzelne Subsysteme zwar prinzipiell als gut zu bewerten ist, diese Unterteilung in der Implementation jedoch unbefriedigend erfolgte und zu größtenteils schwer handhabbaren Schnittstellen führte.

3.7 Darstellung der herausgearbeiteten Systemstruktur

Auf den nachfolgenden Seiten werden verschiedene Abbildungen aufgeführt, die in ihrer Gesamtheit zahlreiche Informationen über das XCTL-System grafisch veranschaulichen und somit direkter zugänglich machen sollen.

Zunächst werden in Abbildung 6 auf Seite 38 alle Subsysteme und ihre Beziehungen zueinander in einer speziellen Übersicht dargeboten. Da die Subsysteme zahlreiche Verflechtungen aufweisen, wurde eine Darstellung gewählt, die mit Hilfe einer in der Grafik näher erläuterten Ebenenbildung eine erhöhte Übersichtlichkeit gegenüber einer "flachen" Darstellung erreicht.

Ab Seite 39 werden alle im XCTL-System vorgefundenen Klassen in Klassendiagrammen dargestellt. Es sind nur die Klassennamen, nicht jedoch die jeweiligen Methoden und Attribute aufgeführt, da es in diesen Darstellungen hauptsächlich um die Veranschaulichung der Klassenhierarchien ging. Im vorliegenden System ist häufig die Situation anzutreffen, dass Basisklassen und ihre abgeleiteten Klassen in verschiedenen Subsystemen implementiert wurden. Um die Klassenhierarchien trotzdem möglichst anschaulich und vollständig darstellen zu können, wurden die jeweils subsystemfremden Klassen mit gestrichelten Symbolen aufgeführt.

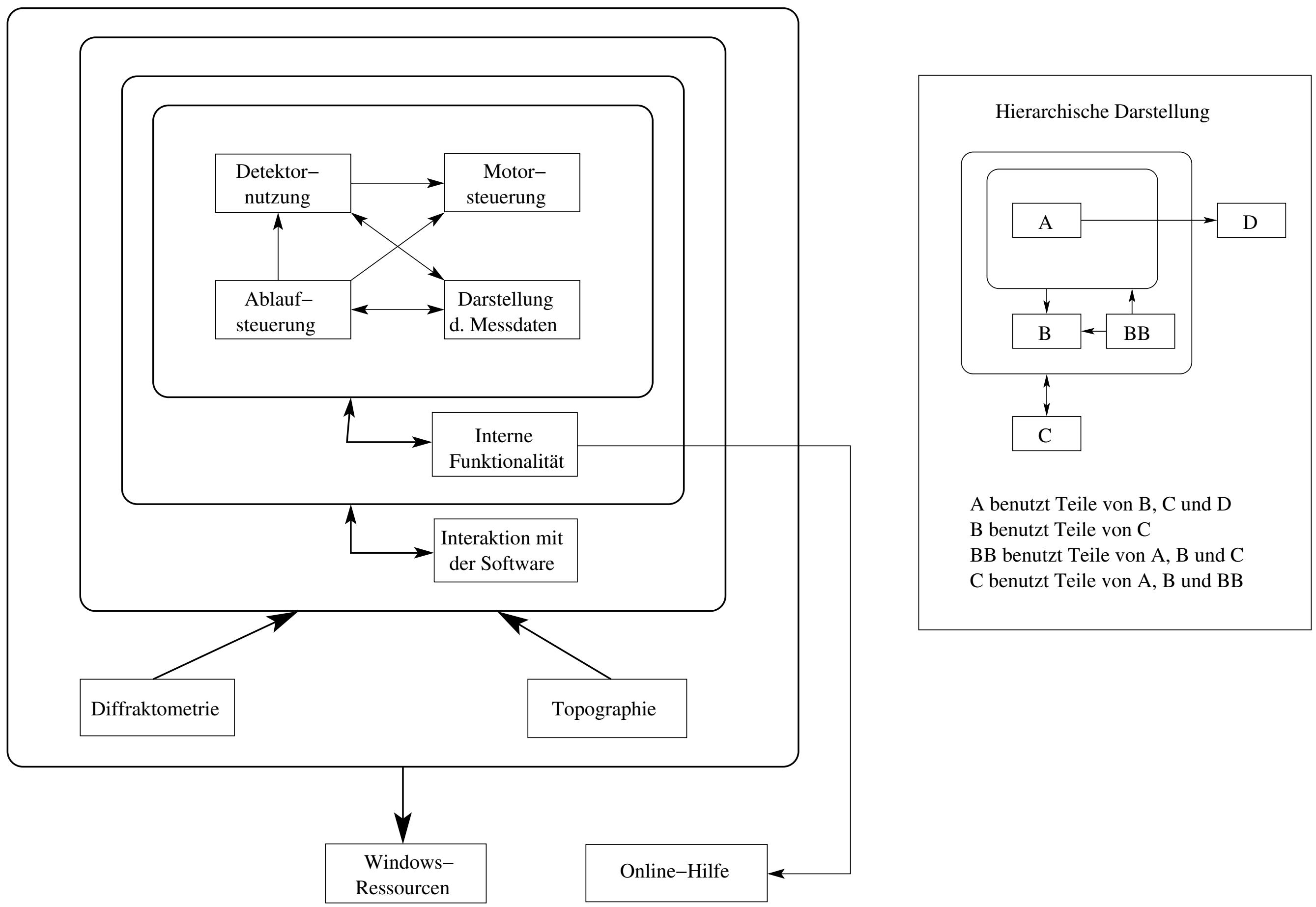

Abbildung 6: Die Subsysteme in der grafischen Veranschaulichung

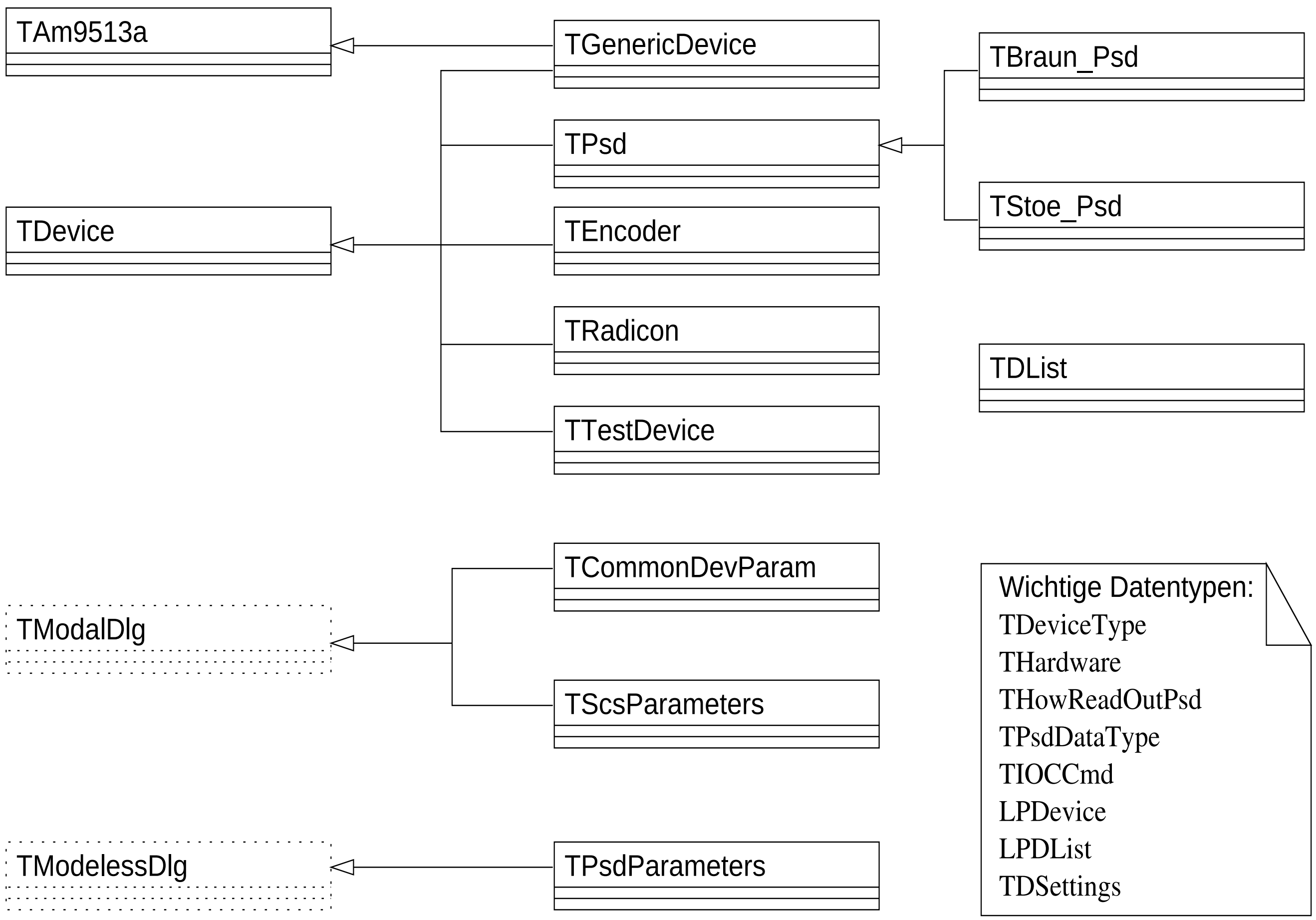

Abbildung 7: Darstellung der im Subsystem Detektornutzung vorgefundenen Klassen

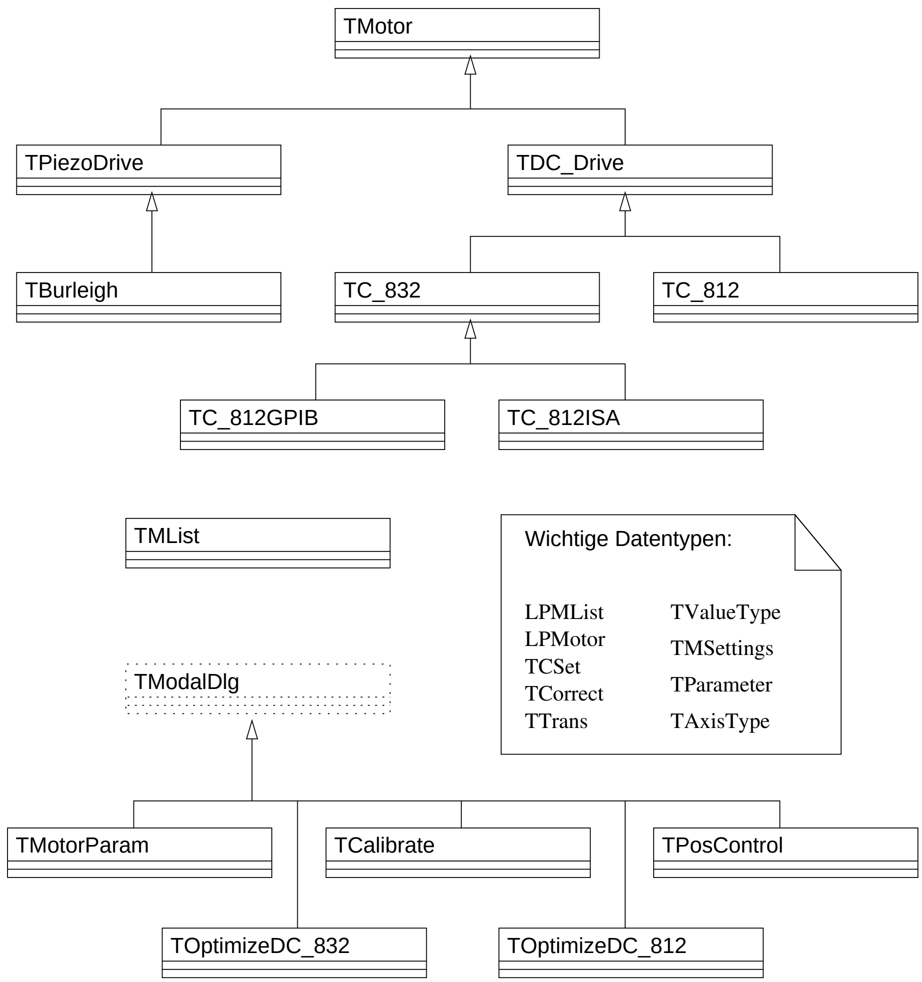

Abbildung 8: Darstellung der im Subsystem Motorsteuerung vorgefundenen Klassen

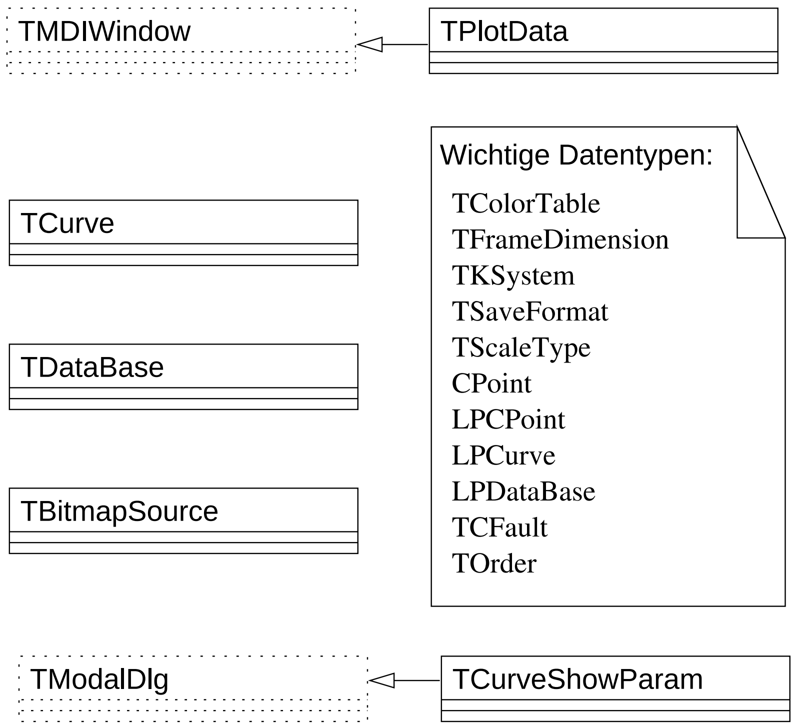

Abbildung 9: Im Subsystem Datendarstellung und -repräsentation vorgefundene Klassen

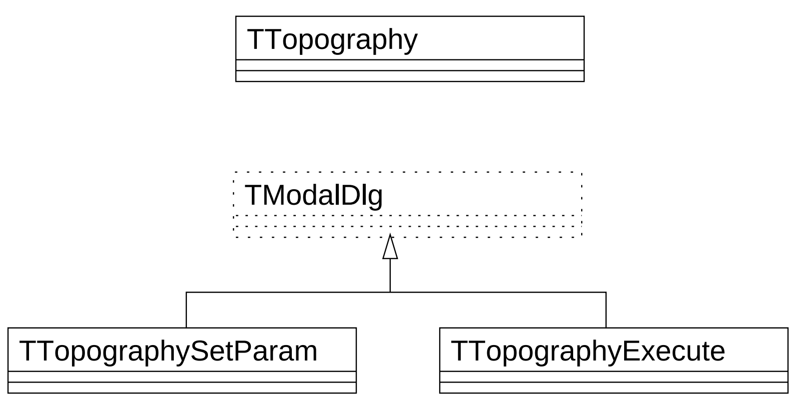

Abbildung 10: Darstellung der im Subsystem Topographie anzutreffenden Klassen

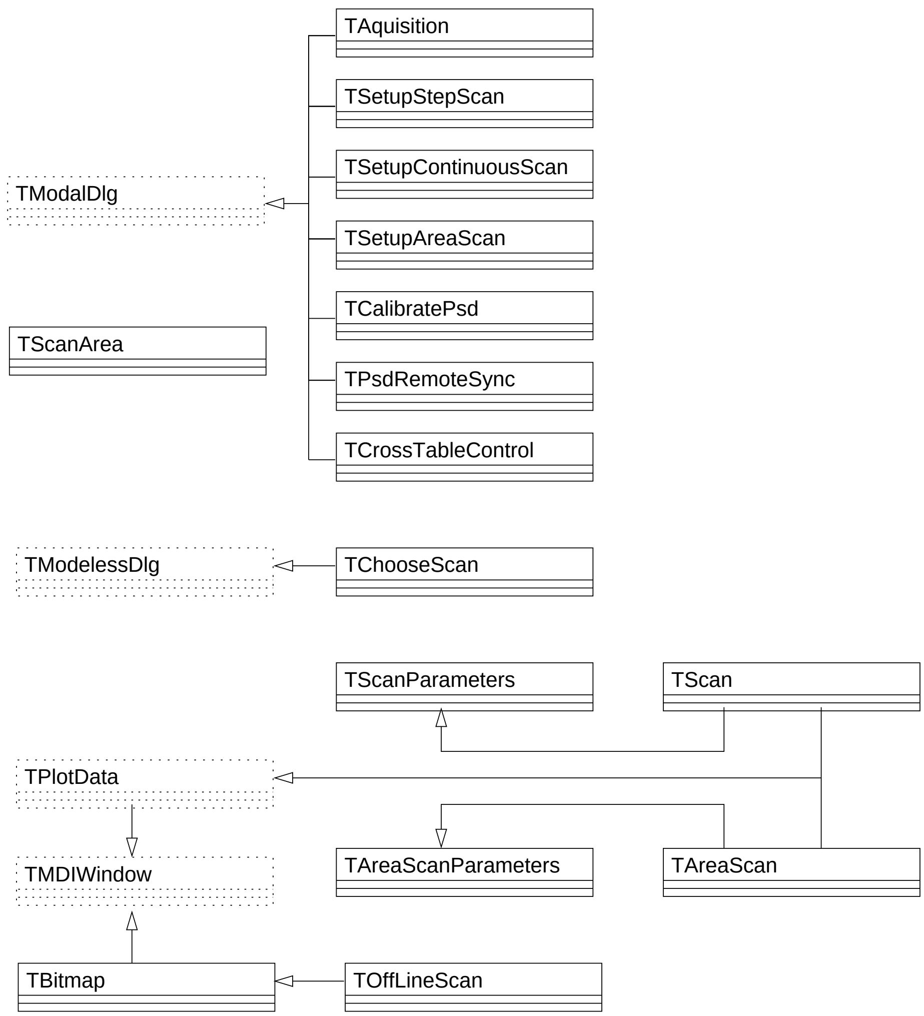

Abbildung 11: Darstellung der im Subsystem Diffraktometrie/Reflektometrie vorgefundenen Klassen

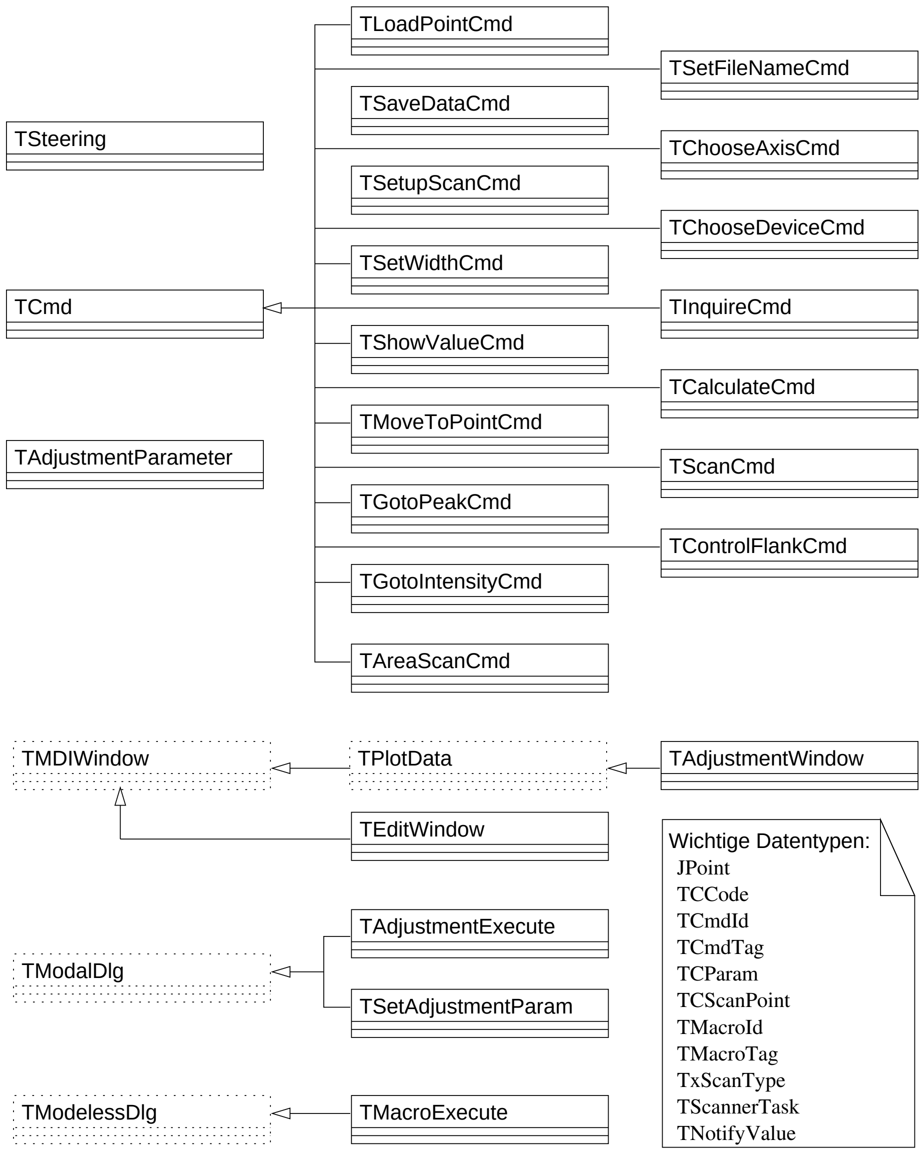

Abbildung 12: Darstellung der zur Ablaufsteuerung bereit stehenden Klassen

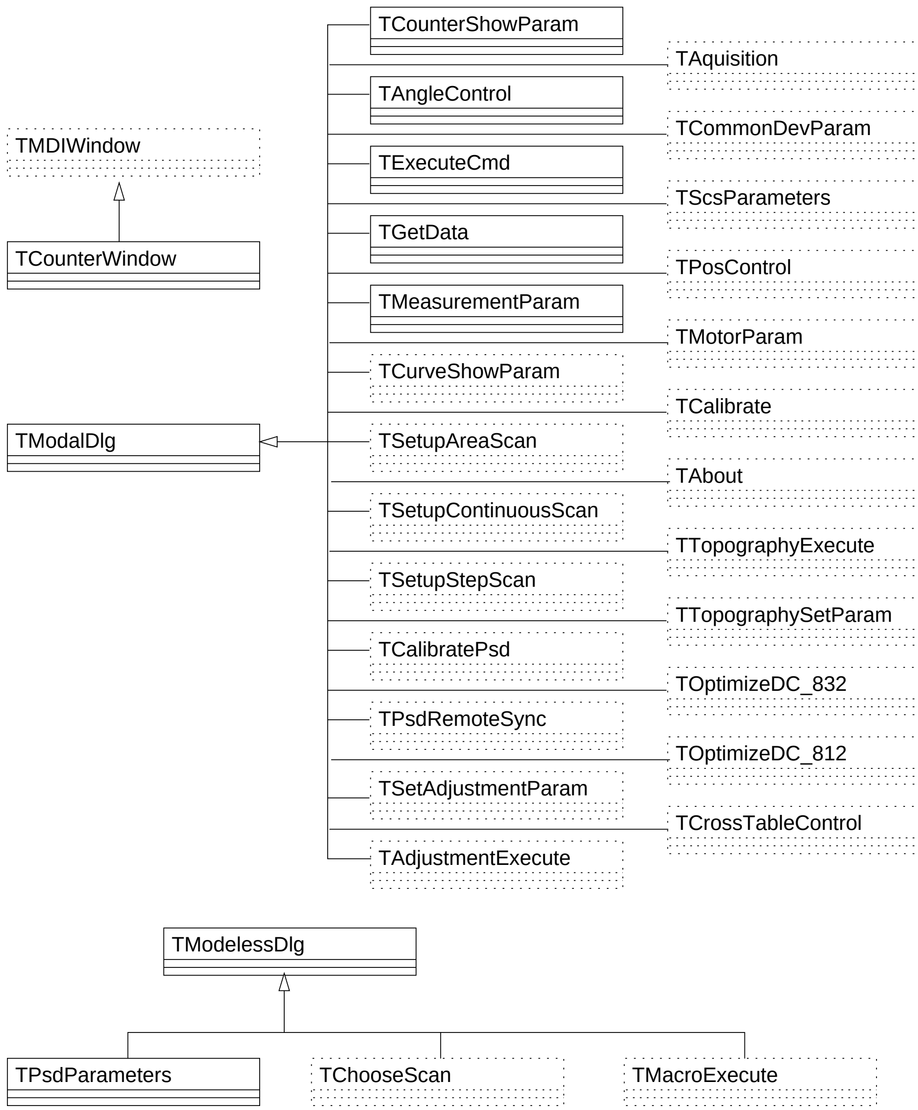

Abbildung 13: Darstellung der zur Interaktion mit der Software bereit gestellten Klassen

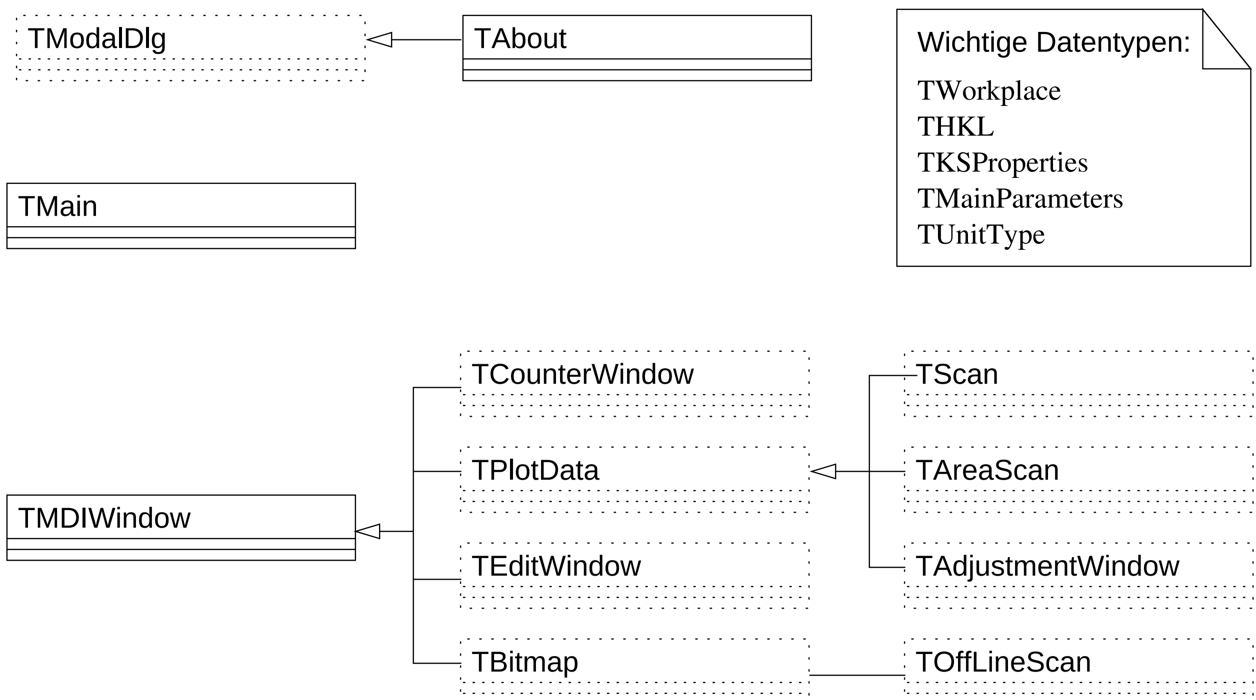

Abbildung 14: Darstellung der im Subsystem Interne Funktionalität vorgefundenen Klassen

4 Verwandte Arbeiten

4.1 Das Bauhaus-Projekt

Im Bauhaus-Projekt geht es den daran beteiligten Wissenschaftlern vorrangig um die Entwicklung einer mindestens halbautomatischen Methode zur Wiedergewinnung und Verbesserung der Architektur eines Softwaresystems auch auf der Ebene der so genannten atomaren Komponenten, die unter anderem von den Elementen der verwendeten Programmiersprache bestimmt werden (s. a. [8]).

Zur direkten Erkennung von Subsystemen wird im Bauhaus-Projekt ein funktionaler Ansatz verfolgt. Dabei werden aus Funktionsaufruf-Graphen Dominanzbäume gewonnen. Deren Teilbäume können jeweils als Teile von Subsystemen interpretiert werden, die dann durch Untersuchung der Quelltexte näher spezifiziert werden. In [9] findet sich eine ausführliche Beschreibung dieser Vorgehensweise.

In der vorliegenden Form allerdings beherrscht das Bauhaus-Tool nur in C implementierte Systeme. Da C++ über Konzepte wie Vererbung und Polymorphie verfügt, wäre eine Erstellung derartiger Aufrufgraphen für C++-basierte Systeme erheblich aufwändiger und voraussichtlich auch fehleranfälliger.

Für eine Anwendung des Bauhaus-Tools auf die vorliegende Software wäre ebenfalls zu klären, in welcher Form Quellen mit verschiedenen Einsprungspunkten verarbeitet werden können, da das XCTL-System als Programm für Microsoft Windows über eine Reihe von so genannten Callback-Funktionen verfügt, die nicht durch das Programm selbst, sondern durch die Betriebssystemumgebung - beispielsweise nach Mouse-Aktionen des Benutzers - aufgerufen werden.

4.2 Das URCA-System

Das URCA-System stellt ein an der Universität von Belgrad entwickeltes System zur Entdeckung von funktional zusammen gehörigen Systemteilen dar.

Dazu werden verschiedenen Use Cases die bei ihrer Durchführung aufgerufenen Funktionen zugeordnet. Dem Anwender obliegt es dabei, geeignete Use Cases auszuwählen. Die automatisch gewonnenen Beziehungen zwischen Use Cases und Quellcodeteilen können dann für Dekompositionsaufgaben bei der Entwicklung von UML-Darstellungen des Systems eingesetzt werden. In [6] findet sich eine genaue Darstellung dieser Schrittfolge.

Derzeit liegt das URCA-Tool jedoch nur als Prototyp vor. Daher ist es auf Software beschränkt, die mit Microsoft Visual C++ entwickelt wurde. Weiterhin sind die verschiedenen grafischen Darstellungen, die durch die vom Tool gewonnenen Informationen ermöglicht werden, derzeit nur mit dem Einsatz weiterer externer Tools, zum Beispiel Rational Rose für die UML-Ansichten, verfügbar.

4.3 Generic Extraction Technique

Dieser Zugang zur Architektur-Wiedergewinnung wird bei Philips Research verfolgt. Kernziel ist dabei die Entwicklung eines universalen Parser-Generators, der durch Parametrisierung an alle in einem großen Unternehmen verwendeten Programmiersprachen angepasst werden kann.

Damit entsteht eine Lösung für das größte Problem bei automatisierten Reverse-Engineering-Projekten: die Extraktion der architekturrelevanten Informationen direkt aus dem Quelltext.

In [10] werden bereits einige wichtige Parametrisierungen erläutert, zudem wird auf die Möglichkeiten, die derart automatisch extrahierte Informationen für weitere Aufgaben des Reverse Engineering eröffnen, kurz eingegangen.

4.4 Einsatz verschiedener Tools zur Architekturwiedergewinnung

In [11] werden die Erfahrungen dargestellt, die mit vier verschiedenen Tools bei verschiedenen Reverse-Engineering-Aufgaben für das in C implementierte UNIX-Programm Xfig gemacht wurden.

In der Bewertung zeigt sich, dass dem Einsatz fast aller dieser Tools zunächst eine manuelle Aufteilung der Quellen in Subsysteme bzw. Komponenten vorausgeht. Nur das von IBM entwickelte Tool Lemma verfügt zusätzlich dazu über die Möglichkeit, diese erste Aufteilung anhand funktionaler Abhängigkeiten vorzunehmen.

Allerdings wird auch in diesem Artikel festgestellt, dass ein Hauptproblem die Anpassung der Parserkomponente des jeweiligen Tools an die implementationsabhängigen Besonderheiten des untersuchten Programms darstellt.

5 Zusammenfassung und Ausblick

5.1 Gewonnene Erkenntnisse

Zunächst kann festgestellt werden, dass auch in größeren Softwaresystemen mit Hilfe eines methodischen Vorgehens die Subsysteme, die der ursprünglichen Entwicklung zu Grunde lagen, mit vertretbarem Aufwand wieder gewonnen werden können.

Dabei liegt es in der Natur der Sache, dass die Qualität der ermittelten Struktur nicht besser sein kann, als die der ursprünglich vom Entwickler des Systems erstellten. Allerdings fördert die Unterteilung eines vorliegenden Systems in Subsysteme auch besonders die kritikwürdigen Elemente des Ausgangsdesigns zu Tage.

Das Fallbeispiel hat gezeigt, dass die Quelltextabgrenzung nach Subsystemen, der erste Schritt für eine folgende Architekturverbesserung, tatsächlich so sehr von den speziellen Gegebenheiten der Implementation und des Gegenstandsbereichs der Software abhängen kann, dass nur eine überwiegend manuelle Bearbeitung zu wirklich zufriedenstellenden Ergebnissen führt.

Im vorliegenden XCTL-System sind beispielsweise verschiedene Programmierparadigmen - imperativ und objektorientiert - parallel eingesetzt worden. Zudem wird das XCTL-System in zwei verschiedenen Anwendungsbereichen eingesetzt, die sich jedoch vergleichsweise ähnlich sind, wodurch die Zuteilung von Quellcodeteilen erschwert wird.

Mit Hilfe des in dieser Arbeit vorgestellten Vorgehensmodells ist es trotz ungünstiger Voraussetzungen seitens des Untersuchungsgegenstands gelungen, eine Unterteilung in Subsysteme vorzunehmen, die wohl durchaus mit dem Ausgangsdesign des Entwicklers vergleichbar ist. Damit ist für weitere Arbeiten am XCTL-System eine erheblich verbesserte Ausgangssituation geschaffen worden.

So kann die schon bisher im Projekt "Softwaresanierung" praktizierte Aufteilung der notwendigen Aufgaben an Hand der Anwendungsfälle nun auch in Bezug auf die jeweils betroffenen Quelltextteile einfacher und sicherer erfolgen.

Weiterhin hat die für Subsystemermittlung nötige intensive Beschäftigung mit dem XCTL-System viel zum Verständnis von Programm und Gegenstandsbereich beigetragen, so dass Studenten, die zum Projekt neu hinzukommen, die Einarbeitung in das XCTL-System erleichtert werden kann. Hierbei sei speziell auf die im Laufe dieser Arbeit entstandene Präzisierung der Use Cases verwiesen, die einen ersten Einblick in das System ermöglicht.

Schließlich sind bei der Ermittlung der Subsysteme zahlreiche Kritikpunkte am untersuchten System festgestellt worden, deren Bearbeitung die XCTL-Software-Architektur zum Teil erheblich verbessern wird.

5.2 Notwendige Arbeiten zur Verbesserung der Systemstruktur

Wie bereits erwähnt, sollen die mit dieser Arbeit gewonnenen Erkenntnisse auch für essenzielle Verbesserungen des XCTL-Systems verwendet werden. Das Hauptaugenmerk liegt dabei natürlich zunächst bei der Beseitigung der zu Tage getretenen Designfehler.

Bildung einer sauberen Headerdateistruktur. Dabei sollte zunächst die Struktur der Headerdateien überarbeitet werden, da ihre Fehler und Ungereimtheiten schon immer einer schnellen Einarbeitung in die Inhalte der Quellen entgegen standen. Bei dieser Überarbeitung sollte zunächst ein von Grund auf neues Headerdateikonzept für das XCTL-System entworfen werden, da das bisherige nicht rekonstruierbar ist bzw. bisher kein echtes Konzept existiert.

Abgrenzung der Benutzeroberfläche. Ein weiterer Schwachpunkt des Systems, der das allgemeine Verständnis erschwert, ist die überwiegend fehlende Abgrenzung der Benutzeroberfläche von den inhaltlichen Komponenten. Dadurch sind notwendige Veränderungen der Oberfläche auf Grund von Anwenderwünschen nur sehr schwer umsetzbar.

Der Aufwand für die Aufteilung der betroffenen Klassen ist bei einem methodischen Vorgehen beherrschbar und wird durch die spätere bessere Handhabbarkeit des Gesamtsystems und des Subsystems, das für die Interaktion des Anwenders mit dem Programm zuständig ist, sicher wieder wettgemacht.

Überarbeitung der Schnittstellen. Weiterhin müssen dringend die Schnittstellenbeziehungen verbessert werden, da bei den vorliegenden Schnittstellen zahlreiche Ungereimtheiten existieren. Dazu müsste zunächst für jedes Subsystem eine inhaltlich motivierte Schnittstelle entworfen werden, die dann mittels Wrapping implementiert werden sollte. Dabei werden die von der Schnittstellenspezifikation geforderten neuen Funktionen durch geeignete Kombinationen von Aufrufen vorhandener Funktionen der alten Schnittstelle gewonnen.

Den komplexeren Teil der Schnittstellenbereinigung im vorliegenden System bildet die Umstellung der Programmteile, in denen auf die alte Schnittstelle zugegriffen wird, auf die Funktionalität der neuen Schnittstelle. Dies ist genau genommen nicht das eigentliche Einsatzfeld von Wrapper-Schnittstellen (Vgl. [7] S. 669ff), da diese vorrangig die Nachnutzung von vorhandenen Bibliotheken für neu zu erstellende Software ermöglichen sollen.

Daher sollte eine genauere Aufwandsabschätzung zunächst untersuchen, ob nicht ein alternatives Vorgehen, das zum Beispiel jede öffentlich verfügbare Funktion hinsichtlich ihres Sinns und ihrer Brauchbarkeit im Rahmen der Subsystemschnittstellen untersucht, schnellere Erfolge zeigen würde.

Verbesserung der Subsystembildung. Schließlich führte die Analyse der Use Cases bezüglich ihrer Umsetzung in Subsysteme des XCTL-Programms bei einigen Anwendungsfällen zu der Feststellung, dass diese vom Entwickler nicht explizit als Subsysteme realisiert wurden.

Daher sollte es auch ein Ziel weitere Verbesserungen der Systemstruktur sein, die Analogien zwischen den Use Cases und Subsystemen zu verstärken. Insbesondere die Manuelle Justage und die Allgemeinen Einstellungen sollten als eigenständige Subsysteme ausgeführt sein.

5.3 Mögliche Verbesserungen des Vorgehensmodells

Das in dieser Arbeit vorgestellte Vorgehensmodell hat sich an seinem ersten großen Fallbeispiel - dem XCTL-System - bewährt. Dennoch sind an einigen Punkten durchaus noch Verbesserungen denkbar.

So stellt die in dieser Arbeit vorgestellte Methode zur Ermittlung der Schnittstellen bereits einen vielversprechenden Ansatz dar, es wäre jedoch noch zu überprüfen, wie mit gängigen Tools zur Analyse von Quelltexten umfangreichere Informationen über die Schnittstellen gewonnen werden können. Dadurch könnte dann auch die anschließende Beschreibung der Schnittstellen über die zur Zeit gebotene Überblicksdarstellung hinaus gehen.

Die Anwendung des Vorgehensmodells auf das XCTL-System hat bei einigen Schritten Möglichkeiten zur Automatisierung von Abläufen aufgezeigt. So könnte die Zuordnung der Headerdateien zu den Subsystemen nach der Aufteilung der Implementationsdateien auch toolunterstützt erfolgen.

Dazu müsste nur analysiert werden, in welchen Implementationsdateien die von der Headerdatei deklarierten Entitäten implementiert werden. Die Zuordnung der Implementationsdatei zu einem bestimmten Subsystem bestimmt dann auch über die Zuordnung der zugehörigen Headerdatei.

Ein weiterer Effekt eines solchen noch zu entwickelnden Tools könnte die parallele Sammlung von Schnittstelleninformationen sein. Hierzu müsste nur protokolliert werden, welche Funktionen auch über Subsystemgrenzen hinweg verwendet werden.

Auch die derzeit angebotenen Darstellungen der Systemstruktur sind noch bezüglich ihrer Nutzbarkeit genauer zu untersuchen. Hier muss die Erfahrung im Projekt zeigen, ob und wie die Darstellungen zu erweitern oder zu ergänzen sind. Dabei ist aber zu beachten, dass diese Darstellungen genau genommen nur einen ersten Überblick über das System ermöglichen sollen. Eine genauere Darstellung der Subsysteminhalte sollte das Ziel von auf die Subsystemwiedergewinnung aufsetzenden Arbeiten sein.

Literatur

[1] Murray R. Cantor, "Object-Oriented Project Management with UML", John Wiley & Sons, 1998.

[2] Ivar Jacobson, Grady Booch, James Rumbaugh, "The Unified Software Development Process", Addison-Wesley, 1999.

[3] Klaus Bothe, "Reverse Engineering: the Challenge of Large-Scale Real-World Educational Projects", 14th Conference on Software Engineering Education and Training, Charlotte, USA, 2001.

[4] Klaus Bothe, Ulrich Sacklowski, "Praxisnähe durch Reverse Engineering-Projekte: Erfahrungen und Verallgemeinerungen", 7. Workshop SEUH, Zürich, Schweiz, 2001.

[5] T. Eisenbarth, R. Koschke, E. Plödereder, J.-F. Girard, M. Würthner "Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen", 1. Workshop Software Reengineering, Bad Honnef, 1999.

[6] Dragan Bojic, Dusan Velasevic, "A Use-Case Driven Method of Architecture Recovery for Program Understanding and Reuse Reengineering", in: Proceedings of the CSMR 2000, Zürich, Schweiz, 2000.

[7] Helmut Balzert, Lehrbuch der Software-Technik, Band 2, Spektrum Akademischer Verlag, 1998.

[8] Rainer Koschke, "Atomic Architectural Component Recovery for Program Understanding and Evolution", Dissertation am Institut für Informatik, Universität Stuttgart, 2000.

[9] J.-F. Girard, R. Koschke, "Finding Components in a Hierarchy of Modules: a Step towards Architectural Understanding", in: Proceedings of the ICSM '97, IEEE Computer Society, 1997.

[10] Tobias Rötschke, "Towards a Generic Extraction Technique", 2. Workshop Software Reengineering, Bad Honnef, 2000.

[11] Susan Elliott Sim, Margaret-Anne D. Storey, "A Structured Demonstration of Program Comprehension Tools", 7th Working Conference on Reverse Engineering, Brisbane, Australien, 2000.

[12] Andreas Bleul, "Programmier-Extremisten", in: c't Magazin für Computertechnik 3/2001, Verlag Heinz Heise GmbH & Co KG, 2001.

[13] Bjarne Stroustrup, "Die C++-Programmiersprache", Addison-Wesley-Longman, 1998.

[14] Sebastian Freund, Derrick Hepp, "Beschreibung einer Schnittstelle zur Motoransteuerung: Das C-Interface des RTK-Steuerprogramms", Studienarbeit am Institut für Informatik, Humboldt-Universität zu Berlin, 2000.