# DIPLOMARBEIT

Zur Erlangung des akademischen Titels
Diplom-Informatiker

---

# SOFTWARESANIERUNG DURCH REFACTORING

UNTERSUCHUNG VON METHODEN ZUR VERBESSERUNG DER SOFTWAREQUALITÄT
IN EINER KOMPONENTE EINES SOFTWARESYSTEMS
ZUR KRISTALLSTRUKTURANALYSE

Jan Picard

September 2003

1. Gutachter: Prof Dr. Klaus Bothe
2. Gutachter: Dipl-Inf. Kay Schützler

---

Institut für Informatik
Mathematisch-Naturwissenschaftliche Fakultät II
Lehrstuhl für Softwaretechnik
Humboldt-Universität zu Berlin

## Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig und nur unter Zuhilfenahme der angegebenen Quellen erstellt habe.

Jan Picard

Berlin, 15. November 2003

## Einverständniserklärung

Hiermit erkläre ich mein Einverständnis mit der öffentlichen Ausstellung meiner Diplomarbeit in der Bibliothek der Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin.

Jan Picard

Berlin, 15. November 2003

## Zusammenfassung

Die vorliegende Arbeit dokumentiert den Verlauf und die Ergebnisse eines Softwaresanierungs-Prozesses. Gegenstand der Untersuchung war ein Softwaresystem zur Halbleiter-Strukturanalyse, das am Institut für Physik der Humboldt-Universität zu Berlin eingesetzt wird. Mit ihm werden dünne Kristallstrukturen mit Hilfe mehrerer Motoren in einem Röntgenlichtstrahl positioniert und das reflektierte Licht mit verschiedenen Detektoren erfasst. Die Software steuert diesen Messvorgang und bietet Möglichkeiten zur Messwertverarbeitung. So können Unregelmäßigkeiten in Schichten und an Schichtübergängen untersucht und visualisiert werden. Gegenstand meiner Betrachtungen war das Subsystem „Detektornutzung“, das dem Gesamtsystem die Nutzung von Röntgenlichtdetektoren ermöglichen soll.

Die Software wurde in einem Zeitraum von fast zehn Jahren evolutionär entwickelt. Die Entwicklung verlief dabei als chaotischer Prozess. Erweiterungen sind häufig nach dem Grundsatz des geringstmöglichen Aufwandes hinzugefügt worden. Das Hinzufügen neuer Funktionalität ging daher mit einer Verschlechterung der Softwarequalität einher.

Das Ziel meiner Sanierungsbestrebungen sollte also sein, die Qualität der Software derart zu verbessern, dass die Wartung und Erweiterung der Software „wirtschaftlicher“ als eine Neuentwicklung sein würde. Zu diesem Ziel sollte mich insbesondere der Einsatz von Refaktorisierungstechniken bringen.

# Inhalt

# 1 Problemstellung

## 1.1 Das Projekt XCTL

Die Arbeitsgruppe „Röntgenbeugung an Schichtsystemen“ am Institut für Physik der Humboldt-Universität unter Professor Köhler untersucht Halbleiterstrukturen mit Röntgenlicht. An circa zehn Messplätzen werden nach verschiedenen Untersuchungsverfahren Eigenschaften von Halbleiterkristallproben erforscht.

An den Messplätzen werden Kristallproben auf einem Probenteller in Röntgenlicht positioniert. Abhängig vom Untersuchungsverfahren und damit vom Einfallswinkel des Röntgenstrahls wird das Licht an den Schichtgrenzen reflektiert (Reflektometrie) bzw. an den atomaren Netzgitterebenen gebeugt (Topographie, Diffraktometrie). Das reflektierte/gebeugte Licht wird mit Röntgendetektoren erfasst. Die erfassten Intensitäts- und Intensitätsverteilungsdaten werden dann am angeschlossenen Messplatz-PC von der XCTL-Software transformiert, visualisiert und gespeichert.

XCTL steht dabei für „X-Ray Control“-System. Die XCTL-Software ist als „Zero-Budget“-Projekt am Institut für Physik von Mitarbeitern entwickelt worden. Diese Mitarbeiter verfügten über das notwendige Expertenwissen in der Problemdomäne und haben ohne Kenntnisse im Software-Engineering ein lauffähiges Projekt von beachtlicher Größe entwickelt.

Später wurde das Projekt mehrfach erweitert: neue Untersuchungsverfahren wurden implementiert, neue Hardware eingebunden usw. Leider verlief dieser Erweiterungsprozess ohne Vorplanung; Analyse-, Entwurfs- und Implementationsphase waren praktisch eins. Die Dokumente, die nach den Lehren des Software-Engineerings als Ergebnis der einzelnen Entwicklungsphasen entstehen sollten, existieren zum Teil bis heute nicht. Beim Hinzufügen neuer Funktionalität wurde mehr Wert auf geringen Änderungsaufwand als auf gutes Design gelegt.

Im Sommer 1998 wandte sich Professor Köhler vom Institut für Physik an Professor Bothe vom Institut für Informatik mit der Frage, ob das Projekt nicht im Rahmen studentischer Arbeiten am Institut für Informatik weitergeführt werden könne. Für den Lehrstuhl „Softwaretechnik“ des Instituts für Informatik ergab sich hier ein interessantes neues Betätigungsfeld; die Projektgruppe „Softwaresanierung“ wurde gegründet.

Studenten konnten hier die erworbenen Kenntnisse über Sanierungstechniken in der Praxis anwenden. Die Anzahl der Personen, die gleichzeitig an dem Code arbeiteten, stieg gegenüber früher beträchtlich. Nach und nach wurden die verschiedenen Subsysteme mit Reengineering-Techniken bearbeitet; die nötige Dokumentation entstand. Verschiedene Studien- und Diplomarbeiten konnten über Projektinhalte verfasst werden.

## 1.2 Das Detektoren-Subsystem

Das Detektoren-Subsystem besteht aus verschiedenen, mehr oder weniger zusammenhängenden logischen Bausteinen, die den anderen Teilen der XCTL-Software die Nutzung der Detektoren ermöglichen soll. Es ist jetzt das letzte Subsystem, das mit historisch gewachsenem Design und ohne Dokumentation geblieben ist.

Der Code erweist sich bei näherer Untersuchung als ein Gemisch von Assembler, C und „C mit Klassen“. Da (fast) jeder C-Code auch gültiger C++-Code ist, wird das Projekt vom C++-Compiler anstandslos übersetzt. Jedoch nutzt der Code die Vorteile objektorientierter Programmierung mit C++ nur in sehr geringem Maße. Der Einsatz von Vererbung, Polymorphie, Zugriffsschutz, Datenkapselung und Wiederverwendung wirkt eher wie nachträglich implementiert. Die Vorteile dieser Prinzipien können so nicht zum Tragen kommen.

Wie schwierig die Situation ist, habe ich erst beim Arbeiten mit den Quellen realisiert. Das Codeverständnis wird vielfach unnötig erschwert. Dieser Zustand ist an sich schon veränderungswürdig, jedoch drängt vor allem das Voranschreiten der Zeit zum Handeln.

## 1.3 Motivation

Die Software wurde unter Borland C++ in den Versionen 3 bis 5 entwickelt. Den Borland C++-Compilern vor Version 5 fehlen zahlreiche wichtige Spracheigenschaften von C++, zum Beispiel Exceptions, die STL (Standard Template Library) oder Namensräume. Für die Hersteller aktueller CASE-Tools stellt die Integration in Borland C++ kein für Kunden attraktives Feature dar; Integration in Entwicklungsumgebungen für Windows wird in erster Linie für Microsoft Visual C++ bzw. für Borland C++ Builder angeboten.

Da das gesamte Entwicklungssystem voll auf die Borland C++-Compilerlinie zugeschnitten ist, bleibt der Projektgruppe die Anwendung von Tools versagt, die neuere Entwicklungsmethoden unterstützen. So ist zum Beispiel Round-Trip-Engineering mit Software-Entwicklungssystemen wie Rational Rose oder ObjectiF nicht möglich. Automatische Codetest-Systeme wie Cantata++ oder CppUnit können nicht genutzt werden, daher mussten und müssen Codetest-systeme selbst geschaffen werden.

An der bisher benutzten PC-Plattform Intel 80486/ Microsoft Windows 3.11 nagt der Zahn der Zeit sehr stark. Die PCs haben Mühe, das Datenaufkommen effizient zu verarbeiten. Das Entwickeln auf der Zielhardware ist nur mit viel Aufwand möglich.

Sobald am Institut für Physik entsprechende Hardware vorhanden ist, soll deshalb auf die Entwicklungsumgebung Microsoft Visual C++ umgestiegen werden, mit der eine 32bit-Version der XCTL-Software unter einem aktuellen Windows-Betriebssystem weiterentwickelt werden soll.

Mehrere Test-Portierungen scheiterten an der unzureichenden Qualität der Software: am schlechten Design und an der fehlenden Dokumentation.

Diese Situation bietet den Antrieb für mein Wirken am XCTL-Projekt in den Jahren 2000 bis 2003.

## 1.4 Ziele und Vorgehensweise

*"Ohne Ziel ist jeder Weg richtig,*
*aber nur wer ein Ziel hat, kommt an."*

*Chinesisches Sprichwort*

Software altert. Aus heutiger Sicht stellt das XCTL-Programm ein Software-Altsystem (legacy system) dar. Um die in das Projekt investierte Arbeit zu sichern, müssen Teile des Systems saniert werden. Dazu existieren zwei Möglichkeiten: die Teilsysteme zu modifizieren (Reengineering) oder zu ersetzen (Forward Engineering). Um hier den richtigen Weg zu finden, muss eine Wirtschaftlichkeitsanalyse durchgeführt werden.

Im Falle des XCTL-Systems kann dabei Wirtschaftlichkeit nicht mit ökonomischer Wirtschaftlichkeit gleichgesetzt werden. Hier muss in erster Linie die bisher investierte Arbeit und der benötigte Änderungsaufwand betrachtet werden. In zweiter Linie soll das im Code enthaltene Wissen bewahrt werden. Deshalb hat sich die Projektgruppe für den Weg des Reengineerings entschieden.

Meine Ziele bestehen nun darin, die Qualitätsmängel im Detektoren-Subsystem zu beseitigen:

- Redokumentation
  Durch Reverse-Engineering sollen Informationen über den vorliegenden Code zusammengestellt werden, um Modelle auf höherer Abstraktionsebene bilden zu können. Anhand dieser Modelle sollen Entwicklerdokumente neu entstehen bzw. verbessert werden. Aus vorhandenen Dokumenten, den Oberflächendialogen und vor allem durch eingehendes Studieren des Codes soll eine Wissensbasis entstehen, die sich in den entsprechenden Entwicklerdokumenten wiederspiegelt.
- Verbesserung der Softwarequalität
  Durch Refactoring, das Verbessern von Codeabschnitten unter Beibehaltung des von Außen beobachtbaren Verhaltens, soll die Qualität des Codes erhöht werden. Durch Einsatz von Refactoring-Techniken soll das Detektoren-Subsystem wieder ein „vernünftiges“ Design erhalten. Dazu muss sowohl die interne Struktur des Subsystems als auch sein Interface überarbeitet werden. Das neue Design soll dokumentiert werden, so dass die Softwarewartung in Zukunft erleichtert wird.

Der Weg dahin wird möglicherweise eher unüblich sein. Während die klassische Methodik ein vollständiges Reverse-Engineering mit anschließendem Forward-Engineering vorsieht, soll hier eine Softwaresanierung durch sukzessives und evolutionäres Modifizieren des Codes nach Refaktorisierungsregeln den Kern der Arbeit bilden.

Mit einer „divide-et-impera“-Strategie werde ich vom Problem „Detektoren-Subsystem“ problematische Teilgebiete abspalten, die einzeln bearbeitet werden. Man beachte, dass ich hier vorab keine vollständige Zerlegung in Teilprobleme anstrebe, sondern das sukzessive Abspalten von Teilen von einem komplexeren Rest.

Dieser Ansatz soll es ermöglichen, schnell Verbesserungen am Code umsetzen zu können. Dadurch wird es einfacher, den verbleibenden Rest, sozusagen das Kernproblem, zu verstehen, weil sich seine Komplexität schrittweise verringert. Anders ausgedrückt führe ich hier qualitative Verbesserungen am Code herbei, ohne vorab das Ziel der Transformation im Ganzen zu kennen.

Es wird also ein echter evolutionärer Prozess sein, der in vielen Schritten zum Ziel führt. Möglicherweise muss dabei bereits modifizierter Code noch einmal modifiziert werden, weil er in einem neuen Kontext die Grundlage für weitere Verbesserungen bildet. Insgesamt erwarte ich mir durch diese Vorgehensweise eine effizientere Problembearbeitung.

Die zu bearbeitenden Teile werden nach Möglichkeit so ausgewählt, dass sie innerhalb kurzer Zeit (angestrebt sind hier ein bis zwei Tage) fertig bearbeitet sind. Diese „Politik der kleinen Schritte“ wird sich unter anderem in wesentlich kürzeren Check-in-Zyklen bemerkbar machen, von denen auch die anderen Projektteam-Mitglieder profitieren werden.

## 1.5 Überblick über die vorhandenen Detektoren

Die Arbeitsgruppe „Röntgenbeugung“ setzt zwei Arten von Detektoren ein: nulldimensionale und eindimensional-ortsauflösende Detektoren.

Nulldimensionale Detektoren (im Volksmund „Geigerzähler“) zählen die einfallenden Röntgenphotonen, die durch das Detektorfenster in ein unter Hochspannung gesetztes Gasgemisch gelangen. Sie kollidieren mit Gasatomen und führen zu einer Stoßionisation, die elektrisch erfasst und somit gezählt werden kann. Durch eine Messung wird also ein Wertepaar (Anzahl der Photonen, gemessene Zeit) ermittelt.

Eindimensional-ortsauflösende Detektoren, kurz PSD (**p**osition-**s**ensitive **d**etector), erfassen außer der Tatsache, das ein Röntgenphoton detektiert worden ist, auch den Ort, wo das geschehen ist. Bei einem PSD ist dazu die Detektorfläche in einer Dimension in so genannte Kanäle aufgeteilt. Mehrere Kanäle lassen sich zu Kanalgruppen zusammenfassen, für die jeweils ein gemeinsamer Messwert ermittelt wird. Neben einem Tupel von Messwerten lässt sich per Integration auch ein skalarer Intensitätsmesswert über alle Kanäle ermitteln.

Die Detektoren sind direkt an die Messplatz-PCs angeschlossen. In der Regel dient dazu eine Interface-Steckkarte, von der aus ein Kabel zum Detektor führt. Aus Sicht des Detektorensubsystems ist an einem Detektor nur diese Schnittstellenkarte interessant, denn nur mit dieser kommuniziert das Programm.

Neben den real als Hardware existierenden Detektoren gibt es auch simulierte Detektoren, die nur als Software existieren. Diese simulierten Detektoren sollen für die Entwickler einen Teil der Testumgebung darstellen, mit der sie ihr System testen können, ohne dass reale Hardware vorhanden sein muss.

Die nulldimensionalen Detektoren:

- Der Szintillisationszähler Radicon SCSCS ( **s**cintillation **c**ounter **s**ingle **c**hannel **s**pectrometer ) der russischen Firma Radicon Ltd. aus St. Petersburg ist der meistbenutzte nulldimensionale Detektor.

- An die Interface-Karte „AXIOM AX5216“ lassen sich bis zu 5 Zählrohre anschließen, die keine eigene Interface-Karte haben. Es handelt sich also nicht um einen Detektor im eigentlichen Sinne, sondern nur um eine Karte zur Ansteuerung von Detektoren, aber aus Sicht des XCTL-Programms spielt das wie oben erwähnt keine Rolle. An diese Karte wird üblicherweise ein weiterer russischer SCSCS angeschlossen.
- Der Testdetektor „Simulant“ von Heiko Damerow. Er liefert aus den aktuellen Motorpositionen berechnete Werte, die das Verhalten eines realen Detektors simulieren sollen.
- Der Testdetektor „Test“ von Kay Schützler. Dieser Detektor ermittelt Werte aus der Datenbasis einer real vorgenommen Messung. Im Gegensatz zum vorgenannten Testdetektor wird hier die Position von drei Motoren berücksichtigt. Er liefert also realistischere Werte, die allerdings mit höherem Speicherverbrauch erkauft werden.

Die eindimensionalen Detektoren:

- Der „Braun PSD“ der deutsch/österreichischen Firma M. Braun.
  Durch den Detektor strömt während der Messung ein Gasgemisch (Argon /Methan bzw. Xenon/Methan). Eine externe Bargraph-Anzeige aus LEDs kann die aktuelle Intensität visualisieren. Die Detektor-Elektronik befindet sich auf einer 8-bit ISA-Karte mit der Bezeichnung ASA (Amplitude-Spectra-Analyser).
- Der „Stoe PSD“ der US-amerikanischen Firma Stoe.
  Dieser Detektor arbeitet ebenfalls mit einem Argon/Methan-Gasgemisch. Der Detektor, der seit 1980 produziert wird, wurde 2001 von Stoe aus dem Programm genommen.
- Der Testdetektor „PSD“ simuliert das Verhalten eines realen PSDs.

# 2 Softwaresanierung

## 2.1 Die Hypothesen von Lehman und Belady

Lehman und Belady untersuchten über 30 Jahre lang Softwaresysteme und erkannten dabei, dass reale Softwaresysteme einer Evolution unterliegen. Die Entwicklung eines Softwaresystems fängt mit seiner Auslieferung an den Nutzer gerade erst an. Reale Software, die für den Nutzer tatsächlich einen Nutzen darstellt, muss kontinuierlich gewartet und weiterentwickelt werden. Neue Wünsche und Anforderungen an das Softwaresystem entwickeln sich. Hardware und Betriebssysteme, auf denen die Software läuft, altern meist noch schneller als das Softwaresystem selbst. Jeder Softwareeinsatz stellt eine Investition dar, und diese Investition muss geschützt und erhalten werden.

Lehman und Belady entwickelten Hypothesen, die sie die „Gesetze der Software-Evolution“ nannten. Diese Hypothesen entstanden aus Untersuchungen an IBMs OS/360; sie wurden in den 70er, 80er und 90er Jahren des 20. Jahrhunderts durch weitere Untersuchungen gestützt[1] .

Lehman teilte Software in Klassen ein[2]:

Software vom **S**-Typ kann vollständig durch eine formale *S*pezifikation beschrieben werden. Erfüllt die Software ihre Spezifikation, ist die Entwicklung abgeschlossen. Beispiele für Software vom **S**-Typ sind Programme, die numerische Berechnungen durchführen.

Software vom **E**-Typ beschrieb Lehman ursprünglich als „Software, die eine menschliche oder soziale Aktivität mechanisiert“. Später wurde die Definition abgeändert zu „Software, die ein Problem oder eine Aktivität der realen Welt bearbeitet oder ausführt“. Die Entwicklung von Software vom E-Typ ist dann erfolgreich, wenn die Software vom Anwender akzeptiert wird. Schon 1980 erkannte Lehman, dass Software vom E-Typ durch ihre ureigene Natur ständig gewartet und weiterentwickelt werden muss, um ihren Anforderungen zu entsprechen. Software vom E-Typ ist dabei Teil seiner Umwelt(**e**mbedded): Die Umwelt verändert sich und die Verwendung der Software trägt zu dieser Veränderung bei. Mit dieser Veränderung der Umwelt beeinflussen Faktoren wie Qualität, Funktionalität, Performanz, Änderbarkeit oder Ergonomie den Grad der Akzeptanz der Software und damit ihren Erfolg.

---

[1] Vgl. [WS]

[2] Vgl. [FWS2], Seite 8ff

Ursprünglich wurde noch eine dritte Klasse von Software als Mittelweg unterschieden: Software vom **P**-Typ. Das waren Programme, die ein **P**roblem lösten, ihre Entwicklung war erfolgreich, wenn das Problem gelöst war. Diese dritte Klasse hat aber heute keine Bedeutung mehr, da sie entweder der S- oder der E-Klasse zugeordnet werden kann.

### 2.1.1 Gesetze der Software-Evolution[3]

**Das Gesetz der kontinuierlichen Veränderung**

(1. Gesetz, 1974)

Ein Programm unterliegt während seiner gesamten Lebenszeit ständiger Veränderung oder es verliert nach und nach seinen Nutzen. Dieser Prozess schreitet fort, bis es kostengünstiger wird, das Programm durch ein neuerstelltes Nachfolgerprogramm zu ersetzen.

**Das Gesetz der zunehmenden Komplexität**

(2. Gesetz, 1974)

Die Komplexität eines Programms nimmt kontinuierlich zu, es sei denn, man ergreift explizit Gegenmaßnahmen, um sie zu erhalten oder zu reduzieren.

**Gesetz der Bewahrung der organisatorischen Stabilität**

(4. Gesetz, 1978)

Der durchschnittliche Aufwand, der für die Pflege einer Software vom E-Typ erbracht wird, ist während der gesamten Lebenszeit des Softwaresystems nahezu konstant.

**Gesetz des kontinuierlichen Wachstums**

(6. Gesetz, 1991)

Der Funktionsumfang eines Softwaresystems vom E-Typ muss kontinuierlich wachsen, um die Nutzerzufriedenheit zu erhalten.

**Gesetz der abnehmenden Qualität**

(7. Gesetz, 1996)

Die Qualität von Systemen vom E-Typ  wird als abnehmend wahrgenommen, es sei denn sie werden rigoros gewartet und an sich ändernde Benutzungsbedingungen angepasst.

[3] Vgl. [LEH96]

## 2.2 Konzepte der Software-Sanierung

Aus den Gesetzen der Software-Evolution ergibt sich, dass Softwaresysteme im Laufe ihres Lebens wachsen, altern, komplexer werden, und an Qualität verlieren. Da Softwaresysteme in der Regel eine größere Lebensdauer haben als die zum Betrieb genutzte Hardware bzw. das Betriebssystem, ist regelmäßige Wartung absolut notwendig, damit die Software unabhängig von der Einsatzbasis weiter genutzt werden kann. Insbesondere Techniken, die sich auf spezifische Eigenschaften der eingesetzten Hardware bzw. des eingesetzten Betriebssystems verlassen, können mit zunehmendem Alter der eingesetzten Systeme ein großes Problem darstellen[4].

Die Softwarewartung stellt ein eigenes Gebiet der Softwaretechnik dar. Der Aufwand für die Wartung ist dabei um ein mehrfaches größer als der für die (Erst-)Entwicklung der Software. Defizite im Wartungsprozess führen dazu, dass der Wartungsaufwand in unwirtschaftliche Dimensionen steigt. Dann ist es an der Zeit zu entscheiden, ob der Lebenszyklus des Softwaresystems beendet und an seiner Stelle ein neues System eingesetzt werden soll, oder ob eine Software-Sanierung durchgeführt werden soll.

Der Begriff der Software-Sanierung umfasst die Konzepte und Verfahren, die ein Softwaresystem dahingehend verändern, dass es wieder wirtschaftlich und leistungsfähig wird. Die Grenze zur Software-Wartung ist dabei unscharf; der Begriff der Software-Sanierung beschreibt eher die Schritte, die zu unternehmen sind, wenn die Wartung vorher nicht im notwendigen Maße durchgeführt worden ist.

### 2.2.1 Forward Engineering

Forward Engineering entspricht dem Entwicklungsablauf bei der Neuentwicklung von Software. Nach einer Analyse der Anforderungen an die Software-Komponente wird ein entsprechendes Software-Design entwickelt und implementiert. Auch bei partieller Neuentwicklung findet Forward Engineering statt.

[4] [SNE91], Seite 62: „Weinberg beschreibt, wie sich Programme durch Testen und Benutzen immer mehr ihrer Hardware-Basis und ihren Eingabedaten anpassen. Falls entweder die Daten oder die Hardware geändert wird, bricht die bisher reibungslos laufende Software zusammen. Falls sie durch genormte Schnittstellen von den Daten und der Hardware nicht ausreichend entkoppelt ist, wird sie auch nie wieder so gut funktionieren wie vorher. Sie bleibt nur als Krüppel mit verminderter Leistung am Leben.

### 2.2.2 Reverse Engineering

Durch Reverse Engineering wird versucht, aus den Artefakten einer Software-Entwicklungsphase die Modelle und Dokumente vorhergehender Entwicklungsphasen wiederherzustellen.

Durch Rekonstruieren (design recovery) versucht man, aus den Quelltexten das Design der Software wiederzugewinnen. Dieser Prozess kann zwar durch entsprechende Hilfsmittel unterstützt werden, letzten Endes bleibt es jedoch eine kreative Tätigkeit, die dem Entwickler von keiner Maschine abgenommen werden kann.

Durch das Redefinieren versucht man, aus dem beobachtbaren Programmverhalten seine ursprüngliche Produkt-Definition abzuleiten.

Durch Reverse Engineering kann wertvolles Wissen zurückgewonnen und bewahrt werden. Dieses Wissen über die tatsächlich vorhandenen Artefakte, ihre Struktur, Semantik und Schwächen, ist Grundvoraussetzung für jede Form von Wartung bzw. Weiterentwicklung.

### 2.2.3 Redokumentation

Redokumentation umschreibt die Tätigkeit, bei der fehlende Entwicklerdokumente zu einer Software-Komponente neu erstellt bzw. bereits existierende Dokumente überarbeitet werden. Redokumentation ist auf allen Entwicklungsstufen möglich: Definitions-, Design- und Implementationsdokumente können überarbeitet bzw. neu erstellt werden.

Oft spiegelt die vorhandene Dokumentation nur einen früheren Entwicklungsstand wieder, die Konsistenz zwischen real Vorhandenem und seiner Beschreibung muss also wieder hergestellt werden.

Fehler, Unklarheiten und Mehrdeutigkeiten erschweren das Verständnis und können Ursache für neue Fehler, Unklarheiten und Mehrdeutigkeiten sein. Falsche Informationen können dabei ebenso fatal sein wie fehlende. Manchmal hilft es, das Vorhandene einfach unter einem anderen Gesichtspunkt zu beschreiben, um anderen (und sich selbst) zu einer tieferen Einsicht zu verhelfen.

### 2.2.4 Reengineering

Reengineering ist ein systematischer Prozess, durch den ein Altsystem in einer neuen Form reimplementiert wird. Er kann alle vorgenannten Aktivitäten umfassen.

## 2.3 Verstehen von Altsystemen

Um ein Reengineering erfolgreich durchführen zu können, muss man das Altsystem verstehen.

Verstehen ist dabei eine Grundbedingung für die menschliche Komponente im Softwareentwicklungsprozess. Für Computer und Entwicklungssysteme ist Verstehen ohne Bedeutung, sie kennen von sich aus keinerlei semantische Assoziationen. Für den Menschen hingegen ist Verstehen im Entwicklungsprozess essenziell.

Verstehen bedeutet in diesem Zusammenhang in erster Linie Modellbildung. Die Standardverfahren der Modellbildung, Abstraktion und Vereinfachung, werden auf das Softwaresystem angewandt, um ein für den menschlichen Geist möglichst vollständig erfassbares Modell zu entwickeln.

Dieser Eindruck, eine Software verstanden zu haben, ist daher immer subjektiv. Ob das selbsterstellte Modell, das im Geiste jedes Einzelnen individuell existiert, mit dem objektiven Modell des Softwaresubsystems übereinstimmt, ist daher von herausragender Bedeutung. Damit ein Entwickler einen Softwareentwicklungsprozess zum gewünschten Ergebnis führen kann, muss sein individuelles Modell zum objektiven Modell semantisch äquivalent sein.

Das objektive Modell muss in Zusammenarbeit von Kunden und den Spezialisten für Systemanalyse im Entwicklungsteam spezifiziert und in geeigneten Darstellungsformen ( z. B. in UML ) festgehalten werden. Erst dadurch ist eine Verifikation des Ergebnisses am Zielmodell möglich. Erst dann kann der Entwicklungsprozess effizient auf verschiedene Entwickler verteilt werden, die Aufgaben, Probleme, Lösungen und Ergebnisse im Entwicklungsprozess standardisiert kommuniziert werden.

Im XCTL-System fehlte dieses objektive Modell viele Jahre lang. Daher wurde Ziel und Richtung im Entwicklungsprozess von den individuellen Modellen der jeweiligen Entwickler bestimmt.

Entwickler haben verschiedene syntaktische Möglichkeiten, die semantischen Zielmodelleigenschaften zu realisieren. Phantasie, Können und Erfahrung bestimmen dabei die Wahl der Mittel, sofern sie nicht durch Projektrichtlinien eingeschränkt sind.

Die Untersuchung des Altsystems hinterlässt im Gedächtnis meist nur Informationsfragmente. Diese müssen durch wiederholtes Studieren des Codes in die richtige Relation zueinander gesetzt und ergänzt werden. Dadurch entsteht ein multidimensionales Gebilde von Fakten, die durch bestimmte Beziehungen untereinander verbunden sind. Werkzeuge können dabei helfen, gezielte Fragen zu beantworten, deren Antworten neue Verbindungen zwischen den Informationen herstellen oder alte modifizieren.

Für das Verstehen eines Altsystems existieren prinzipiell drei Möglichkeiten[5]:

- Verstehen als black box
- Verstehen als white box
- Mischformen daraus.

Das black-box-Verstehen beschränkt sich auf die Untersuchung extern beobachtbaren Verhaltens und der vorhandenen Dokumentation. Diese Methode ist gut geeignet, sich einen Überblick über ein System zu verschaffen. Insbesondere wenn nur das Verstehen einer Teilkomponente von Interesse ist, reicht black-box-Verstehen der anderen Komponenten meist aus, um deren Beziehung zur untersuchten Komponente einschätzen zu können.

White-Box-Verstehen ist um ein vielfaches aufwendiger. Dazu müssen die Interna der Implementation untersucht werden. Dem deutlich größeren Aufwand steht dabei ein wesentlich detailliertes Wissen über den Code gegenüber. Der verborgene Schatz an Informationen, die man Kommentarzeilen früherer Entwickler entnehmen kann, bleibt beim black-box-Verstehen einfach vorenthalten.

### 2.3.1 Semantische Konsistenz

Exakte Übereinstimmung zwischen dem Namen einer Funktion und dem, was sie tatsächlich tut, bezeichne ich hier als semantische Konsistenz. Das rein syntaktische Konstrukt des Bezeichners, das zugehörige semantische Modell und das tatsächliche Verhalten stimmen überein.

[5] Vgl. dazu [BAL98], Seite 671

Ich habe zum Verstehen des Codes des XCTL-Systems einen Zwischenweg zwischen den beiden extremen Verstehensmethodiken verwandt. Ich habe für alle Methoden untersucht, inwiefern semantische Konsistenz besteht. Der dazu nötige Kontrollaufwand ist lokal auf die jeweilige zu untersuchende Funktion beschränkt. In der zu untersuchenden Funktion aufgerufene Funktionen werden als semantisch konsistent angesehen. Hat man jede einzelne Funktion untersucht, ist auch das Gesamtsystem aus allen Funktionen semantisch konsistent. In allen Fällen, in denen die semantische Konsistenz nicht gegeben war, habe ich durch die Refaktorisierung „Methode umbenennen“ einen neuen Namen vergeben, der die Semantik der Funktion besser beschreibt.

Anschließend war das Verstehen deutlich einfacher, weil durch die sichergestellte semantische Konsistenz globale Zusammenhänge schneller erkannt werden konnten. Eine Funktion, die ihrerseits mehrere weitere Funktionen aufruft, ist leichter zu verstehen, wenn man nur ihren Funktionskörper untersuchen muss, nicht jedoch den der aufgerufenen Funktionen.

Semantische Konsistenz ist also eine wichtige Eigenschaft von Software. Sie beeinflusst direkt die Qualitätsmerkmale Verständlichkeit und Wartbarkeit. Ihre Erhöhung stellt in jedem Falle eine Verbesserung dar.

Werden Designinformationen über die Software extrahiert (Cross Referencer, grafische Darstellungen in UML-Diagrammen), so gehen die Detailinformationen verloren, die erkennen lassen könnten, dass der Name einer Funktion nur in schwachem Zusammenhang zu ihrer Funktionalität steht. Der Name führt zu falschen Vermutungen über das interne Verhalten und vermindert somit auch die Effizienz und den Produktivitätsgewinn durch SE-Tools.

# 3 Refactoring

Das folgende Kapitel soll in das Refactoring einführen, weil es den Kern meiner Arbeit am Detektoren-Subsystem darstellt.

Auf keinen Fall soll dieses Kapitel ein Ersatz für das Studium entsprechender Literatur sein, allem voran Martin Fowlers Buch „Refactoring – Wie sie das Design vorhandener Software verbessern“. In diesem Buch werden Gedanken und Verfahren zur Verbesserung vorhandener Software dargestellt, die für sich so simpel und einleuchtend sind, dass sie jedem Entwickler eigentlich von allein klar sein müssten. Zusammen mit dem vorliegenden Code wurde mir bewusst, dass es mit den Hilfsmitteln, die Fowler in dem Buch zur Verfügung stellt, viel zu verbessern gab.

Daher will ich hier nicht das Buch wiedergeben, sondern nur Grundkonzepte und Verfahren darzustellen, die für das Verständnis meiner Arbeit essentiell sind.

### 3.1.1 Was ist Refactoring?

Refactoring ist eine Transformationsmaßnahme, die auf der Design- und der Implementationsebene arbeitet. Das System wird in eine neue Darstellung überführt, die verschiedenen Anforderungen besser gerecht wird als die alte. Die meisten Refaktorisierungen zielen darauf ab, die Komplexität eines Codeabschnittes oder eines Designs zu reduzieren, was direkt zu einer Verbesserung der Verständlichkeit führt. Unnötige Abhängigkeiten können aufgelöst und Design-Muster eingearbeitet werden.

Gerade C++-Programmierer, die mit C „vorbelastet“ sind, neigen verstärkt zu einem prozeduralen Programmierstil. Der entstehende Code ist dann zwar korrektes C++, die neuen Möglichkeiten der Sprache (und damit meist auch die der Objektorientierung) werden aber nicht genutzt. Hier kann Refactoring wahre Wunder bewirken: der Code wird einfacher, portabler, weniger fehleranfällig, wiederverwendbarer, wartbarer – und vor allem verständlicher. Jede dieser Verbesserungen stellt für sich eine Qualitätssteigerung dar, alle zusammen einen Qualitätssprung.

Transformationsregeln versuchen dabei, die objektive Verbesserung vom subjektiven Eindruck einer Verbesserung unabhängig zu machen. Immer dann, wenn eine bestimmte Situation im Code vorgefunden wird, kann durch eine spezielle – festgeschriebene und damit reproduzierbare – Transformation eine objektive Verbesserung erreicht werden.

Durch Refactoring wird per definitionem das Externverhalten der Software nicht beeinflusst. Wohl aber können Effizienz, Zuverlässigkeit, Änderbarkeit und Übertragbarkeit des Softwaresystems deutlich gesteigert werden. Indirekt wird dadurch die Erweiterung der Software erheblich vereinfacht. Außerdem wird durch die Qualitätssteigerung das Auffinden und Beseitigen von vorhandenen Fehlern erheblich erleichtert und die Wahrscheinlichkeit für das Einbringen neuer Fehler verringert.

## 3.2 „Bad Smells“

Um zu erklären, wann Refactoring eingesetzt werden sollte, nutzt Martin Fowler das Konzept des „übel riechenden“ Codes. Statt eines formellen Systems nutzt er hier ein Konzept, dass eher auf emotionalem Wege die menschliche Intuition anspricht. Anhand konkreter Codemuster erklärt er, an welchen Stellen Refaktorisierungen zu besseren Lösungen führen.

Die meisten dieser „Gerüche“ findet man auch im XCTL-System. Folglich sollte sich das System durch Anwendung von Refaktorisierungen verbessern lassen.

### 3.2.1 Duplizierter Code

Den aufdringlichsten „Geruch“ verströmt duplizierter Code. Kommt dieselbe Codepassage an mehreren Stellen im Programm (zumindest in ähnlicher Form) vor, so kann der Code verbessert werden, wenn man einen Weg findet, diesen Codeabschnitt nur einmal vorkommen zu lassen, z. B. indem man ihn in eine eigene Funktion umwandelt, die dann jeweils aufgerufen wird. Wartung und Erweiterung ist dann nur an dieser einen Stelle notwendig.

### 3.2.2 Lange Methoden

Seit langem ist bekannt, dass eine Methode umso schwerer zu verstehen ist, je länger und komplexer sie ist.[6]

Kurze Methoden sichern dagegen eine höhere Wiederverwendbarkeit, da sie nur klar abgegrenzte Aufgaben erledigen. Wird für eine Methode ein guter Name genutzt, der exakt diese Aufgabe beschreibt, braucht man zum Codeverstehen den Methodenkörper nicht einmal lesen.

[6] Vgl. dazu die Halstead-Lesbarkeitsmetriken und die McCabe-Komplexitätsmetriken

Auch lange Methoden haben naturgemäß nur einen Namen. Semantische Konsistenz zwischen diesem Namen und dem Code lässt sich aber nur in den Codeabschnitten herstellen, welche die Hauptaufgabe der Methode erledigen. Die Codeabschnitte für die Unteraufgaben sind zwar zur Erfüllung der Hauptaufgabe notwendig, weisen aber nur wenig Übereinstimmung mit dem Namen der Methode auf. Damit ist ein eingehendes Studium diese Passagen nötig, um ihren Sinn zu verstehen.

Bei inhaltlicher Übereinstimmung von Methodenname und Methodenaufgabe kann der Programmierer und der Lesende überprüfen, ob der Code seinen Namen gerecht wird, d. h. ob er die dem Namen entnehmbare Aufgabe tatsächlich erfüllt. Ist diese Übereinstimmung für alle Unteraufgaben einer Methode erfüllt, so lässt sich auch die Überprüfung für die Hauptaufgabe leicht vollziehen.

Fowler schlägt hierfür vor, nach Kommentaren zu suchen, die die Aufgabe eines Codeabschnittes erläutern. Unter einem Namen, der dieser Aufgabe entspricht, extrahiert man dann den Codeabschnitt einfach in eine neue Methode. Man erhält so eine alte Methode, die einfacher zu lesen ist, und eine neue Methode, die man an anderer Stelle wieder verwenden kann.

### 3.2.3 Grosse Klassen

Grosse Klassen sind Klassen mit mehr als einer Aufgabe. In ihnen kann daher nur schwache Bindung bestehen, ein Hinweis auf ein mangelhaftes Design.

Gute Kandidaten dafür sind Klassen mit vielen Attributen. Sie sind unflexibel, schwer zu verändern, schwer zu erweitern, schwer wiederzuverwenden und schwer zu verstehen. Diese Liste von Mängeln sollte einem Programmierer ausreichen, um die Notwendigkeit zu erkennen, große Klassen in mehrere kleine aufzuspalten.

### 3.2.4 Lange Parameterlisten

In den Zeiten struktureller Programmierung war es üblich, einer Funktion alle Daten als Parameter zu übergeben, die sie für ihre Funktion brauchte. Bei objektorientierter Programmierung ist es sinnvoller, nur so viel an Parametern zu übergeben, dass die Methode sich alles selbst holen kann, was sie zur Erledigung ihrer Funktionalität braucht. Übergibt man als Parameter Objektreferenzen, so kann die Methode die erforderlichen Daten bei den Parameterobjekten selbst erfragen.

### 3.2.5 Neid

Neid tritt auf, wenn sich ein Objekt mehr für die Daten eines anderen Objektes interessiert als für seine eigenen. Sinnvollerweise müssen die entsprechenden Methoden dann in andere Klasse transferiert werden. Neid ist ein Ausdruck von unnötig hoher Kopplung; die zwei Klassen sind in einem höheren Maße voneinander abhängig, als notwendig. Refaktorisierungen, die Neid beseitigen, reduzieren somit die Komplexität der Software.

### 3.2.6 Datenklumpen

Datenelemente, die nur zusammen verwandt, als Parameter übergeben bzw. manipuliert werden, bilden Datenklumpen. Fasst man diese Datenelemente zu einem neuen Objekt zusammen, wird der innere Zusammenhang dieser Daten explizit gemacht. Die zusammengehörenden Daten können zusammen als ein Objekt übergeben werden, was den Code klarer werden lässt.

Manche der Datenelemente verkörpern nicht den Objektzustand; sie enthalten nur Informationen, die aus den anderen der zusammengehörenden Daten berechnet werden. Je nachdem, mit welcher Wahrscheinlichkeit diese Daten genutzt werden, können verschiedene Optimierungsstrategien verfolgt werden:

Bei hoher Wahrscheinlichkeit, dass die Daten verwendet werden, können Ergebnisse von Berechnungen vorab ermittelt und dann zwischengespeichert werden können, wodurch Berechnungen gespart werden[7].

Wenn die Wahrscheinlichkeit, dass Daten genutzt werden, eher gering ist - vielleicht weil eine Vorbedingung nur selten erfüllt wird und daher auch nur selten eine Verzeigung zum dem Codeabschnitt stattfindet, der die Daten verwenden würde – sollte man die Berechnung erst zum spätestmöglichen Zeitpunkt, nämlich wenn das Ergebnis tatsächlich benötigt wird, ausführen[8].

In jedem Fall ist es günstiger, eine Methode einzuführen, die den entsprechenden Wert liefert. Welche Strategie zur Ermittlung des Ergebnisses genutzt wird, bleibt dann hinter der Methodenschnittstelle verborgen.

[7] Vgl. [MEY97], Seite 109ff : „Richtlinie 18 : Spare die Kosten erwarteter Berechnungen“

[8] Vgl. [MEY97], Seite 100ff : „Richtlinie 17 : Erwäge Lazy Evaluation ( verzögerte Berechnung )“

### 3.2.7 Spekulative Allgemeinheit

„Spekulative Allgemeinheit“ umschreibt Codeabschnitte, die Dinge realisieren sollen, die in unbestimmter Zukunft eventuell benötigt werden. Das wichtigste Abgrenzungskriterium ist hierbei, dass im Moment nicht sicher ist, ob der Mechanismus überhaupt genutzt werden wird.

Einen nicht verwendeten Mechanismus zu verstehen ist schwierig genug - ist er zusätzlich nur rudimentär implementiert, ist jeder Versuch, ihn verstehen zu wollen, verschwendeter Aufwand. Solche Mechanismen sollten so schnell wie möglich entfernt werden, zu Gunsten der Wartbarkeit des funktionierenden Codes.

### 3.2.8 Kommentare

Code mit Kommentaren zu versehen ist gut. Die richtigen Stellen mit den richtigen Kommentaren zu versehen, ist allerdings nicht trivial. Zu oft werden Kommentare dazu genutzt, den Geruch von schlechtem Code zu überdecken.

Ist ein Codeabschnitt mit einem Kommentar versehen, der beschreibt, *was* getan wird, dann stimmt etwas nicht. Entweder ist der Kommentar schlicht überflüssig oder der Code ist so schlecht, dass er geändert werden muss. Was kann einen Codeabschnitt besser erklären als der Code selbst?

Insbesondere mit „kleinen“ Refaktorisierungen wie „Datenelement umbenennen“, „Methode umbenennen“ bzw. „Methode extrahieren“ können hier sehr schnell Verbesserungen herbeigeführt werden.

Die einzig wertvollen Kommentare sind die, die beschreiben, *warum* der Code etwas tut. Solche Kommentare stellen für die spätere Entwicklung eine unschätzbar wertvolle Quelle von Metainformationen zur Verfügung, die aus dem Code allein nur schwer bzw. gar nicht gewonnen werden können.

## 3.3 Refaktorisierungen

Im Folgenden sollen einige der von mir bei meiner Arbeit verwandten Refaktorisierungen dargestellt werden. Dadurch will ich dem Leser zeigen, durch welche verblüffend einfachen Transformationen Codeverbesserungen erreicht werden können.

### 3.3.1 Methode extrahieren

Methode extrahieren kann aus zwei Gründen angewandt werden: „Duplizierter Code“ soll an einer Stelle zusammengefasst werden oder aus einer „Langen Methode“ sollen Teile mit einer semantisch konsistenten Bezeichnung abgespalten werden.

#### 3.3.1.1 Zusammenfassen von dupliziertem Code

Im alten Code des Detektorensubsystems fand man häufig solche Codeabschnitte:

```
int TDevice::GetData( float &data )
{
…
  if( fIntensity == 0.0 )
    fSigma = 1.0;
  else
    fSigma = sqrt( fIntensity )  / ( fIntensity );
…
};
```

oder

```
int TDevice::PollDevice( void )
{
…
  if( fIntensity == 0.0 )
    fSigma = 1.0;
  else
    fSigma = sqrt( fIntensity )  * 0.9 / fIntensity;
…
};
```

Beide Abschnitte stellen ( bis auf einen Faktor ) duplizierten Code dar. Kombiniert man die Refaktorisierungen „Methode extrahieren“ und „Methode parametrisieren“, so kann man den üblen Geruch nach dupliziertem Code beseitigen, indem man eine neue Methode mit einem beschreibendem Namen einführt, die zusätzlich einen Faktor als Parameter übernehmen kann.

```
float TDetector::CalculateSigma(float intensity, float factor)
{
  if ( intensity == 0.0 )
    return 1.0;
  return factor * sqrt(intensity) / intensity;
}
```

Mit den Mitteln von C++ kann für den Parameter Faktor sogar ein Standardwert von “1” vorgegeben werden, so dass dieser Parameter nur bei Bedarf verwandt werden muss.

```
class _COUNTERCLASS TDetector
{
…
protected:

 //! Berechnet Sigma aus der ermittelten Intensitaet
 static float CalculateSigma(float intensity, float factor=1.0);
};
```

Anschließend können an allen Fundstellen statt des duplizierten Codes ein einfacher Methodenruf integriert werden:

```
int TZeroDimSimpleTestDetector::PollDetector( void )
{
  …
    fSigma = CalculateSigma( fIntensity, 0.9 );
  …
}
```

#### 3.3.1.2 Zerlegen von langen Methoden

Eine furchtbare Erscheinung im XCTL-Projektcode waren die Eventhandler-Methoden Dlg_OnCommand(). Diese Funktionen sollen je nach erhaltener Windows-Botschaft eine Aktion anstoßen.

Praktisch wurde das durch gigantische switch-Anweisungen realisiert, in deren case-Zweigen die vollständigen Aktionen deklariert waren, z. B. TSetupAreaScan::Dlg_OnCommand() mit 210 Zeilen.

```
void TCommonDevParam::Dlg_OnCommand( HWND hwnd,int id,HWND
hwndCtl,UINT codeNotify )
{
…
    case cm_SpecificParameters:
      // spezifische Parameter für einen Device setzen
      Device->SetSpezificParametersDlg( );
      FORWARD_WM_COMMAND( GetHandle(
),cm_ActivateChanges,0,0,PostMessage );
      break;
…
};
```

Hier habe ich durch mehrfaches Refaktorisieren mit „Methode extrahieren“ die Aktionen in den case-Zweigen in selbständige Methoden transferiert.

```
void TCommonDevParamDlg::Dlg OnCommand( HWND hwnd, int id, HWND
hwndCtl, UINT codeNotify )
{
  switch ( id )
  {
…
    case cm_SpecificParameters:
      OnSpecificParameters();
      break;
…
  }
}

void TCommonDevParamDlg::OnSpecificParameters()
{
  Detector->RunSpecificParametersDlg( );
  FORWARD_WM_COMMAND( GetHandle( ), cm_ActivateChanges, 0, 0, PostMes-
sage );
}
```

So entstehen aus einer Methode mit 100 Zeilen zehn Methoden mit je zehn Zeilen, die jede eine abgeschlossene Aufgabe unter einem eigenen Namen ausführen

### 3.3.2 Erklärende Variable einführen

Erklärende Variablen verknüpfen temporäre Werte mit Namen, die etwas über ihre Bedeutung aussagen. Diese Form der Zwischenspeicherung dient ausschließlich dem Menschen, der dieses Stück Code verstehen soll, jeder optimierende Compiler wird solche nicht notwendigen Zwischenspeicherungen im Zielcode entfernen.

Im folgenden Beispiel wird eine Methode aus dem alten Projektcode überarbeitet:

```
BOOL TCommonDevParam::CanClose( void )
{
…
  GetDlgItemText( GetHandle( ),id_ExposureCounts,( LPSTR ) buf,10 );
  fvalue = atof( buf );
  if( ( fvalue > 0 ) )
  {
    Device->dwExposureCounts = fvalue;
    if( abs(fvalue) > Device->dwUpperCountBound )
      Device->dwExposureCounts = Device->dwUpperCountBound;
    if( abs(fvalue) < Device->dwLowerCountBound )
      Device->dwExposureCounts = Device->dwLowerCountBound;
  }
  else return FALSE;

  GetDlgItemText( GetHandle( ),id_ExposureTime,( LPSTR ) buf,10 );
  fvalue = atof( buf );
  if( ( 0.0 != fvalue ) )
  {
    Device->fExposureTime = fvalue;
    Device->fExposureTime = min( Device->fExposureTime,Device-
>fUpperTimeBound );
    Device->fExposureTime = max( Device->fExposureTime,Device-
>fLowerTimeBound );
  }
…
};
```

An dieser Methode fallen mehrere Missstände auf: Die aus den Eingabefeldern des Dialogfensters ausgelesenen Werte werden in einer temporären Variable mit der nicht sehr aussagekräftigen Bezeichnung „fvalue“ gespeichert.

Der Abschnitt kann klarer werden, wenn man die neuen Werte für die maximale Belichtungsdauer und die maximale Impulsanzahl in je einer erläuternden Variable speichert. Im Beispiel wird in der Variable „fNewExposureTime“ der neue Wert für die Belichtungsdauer gespeichert.

```
BOOL TCommonDevParamDlg::CanClose( void )
{
…
  GetDlgItemText( GetHandle( ), id ExposureCounts, ( LPSTR ) buf, 10
);
  DWORD dwNewExposureCounts = atol(buf);

  GetDlgItemText( GetHandle( ), id_ExposureTime, ( LPSTR ) buf, 10 );
  float fNewExposureTime = atof(buf);

  Detector->SetExposureSettings( TExposureSettings( fNewExposureTime,
dwNewExposureCounts) );

…
}
```

### 3.3.3 Temporäre Variable zerlegen

Das vorhergehende Beispiel kann auch ein Bespiel für die Refaktorisierung „Temporäre Variable zerlegen“ angesehen werden.

Im alten Code werden nacheinander verschiedene Werte in dieselbe temporäre Variable „gezwängt“, obwohl deren Typen gar nicht korrespondieren. Diese unnötige Sparsamkeit trägt nur dazu bei, die Klarheit des Codes herabzusetzen.

Im transformierten Code werden die neuen Werte für die maximale Belichtungsdauer und die maximale Impulsanzahl in je einer eigenen temporären Variable gespeichert. Jede temporäre Variable für sich erhält so eine Bedeutung und eine Verantwortlichkeit.

### 3.3.4 Methode/Feld verschieben

Methode verschieben wird u. a. angewandt, um den Geruch des Neids zu bekämpfen. Eine Methode, die in erster Linie mit den Daten einer anderen Klasse arbeitet, wird in diese andere Klasse verschoben. So können unnötige Abhängigkeiten reduziert werden.

Analog dazu kann auch ein Attribut in eine andere Klasse verschoben werden, wenn es eher dazu passt und wenn es die Klarheit des Codes erhöht.

Beide Refaktorisierungen habe ich bei der Umwandlung der Radicon-C-Treiber in eine C++-Klasse angewandt. Die Methode zum Senden der Radicon-Firmwaredatei an den Controller ist eine Aufgabe der Hardwareklasse, daher wurde diese Methode von TRadicon nach TRadiconHW verschoben. Aus demselben Grund habe ich die Attribute, die die zu verwendenden Kommunikationsports speichern, in die Hardwareklasse verschoben.

### 3.3.5 Feld kapseln

„Feld kapseln“ ist eine starke Hilfe bei der Wiederherstellung von Interfaces. Generell sind öffentliche Attribute eines der größten Übel der Objektorientierten Programmierung[9]. Öffentliche Attribute können überall im Code modifiziert werden. Daher können Fehlerquellen auch überall im Projektcode verteilt sein, das Grundprinzip der Kapselung ist verletzt.

Um das Feld zu kapseln, wird für das Attribut der Zugriffsschutz verstärkt, mindestens also als geschützt(protected) deklariert. Anschließend können die benötigten Accessor-(Get-) und Mutator-(Set-)methoden mit dem nötigen Zugriffsschutz hinzugefügt werden.

Im Falle einer Fehlersuche kann man nun davon ausgehen, dass das Attribut nur durch den Aufruf der entsprechenden Mutatormethode verändert wurde. Sämtliche Aufrufe dieser Methode lassen sich mit einem Debugger oder mit entsprechender Sourcecode-Instrumentalisierung überwachen und Fehler somit schnell auffinden.

### 3.3.6 Redundante Bedingungsteile konsolidieren

Im folgenden ( später nochmals verwandten ) Beispiel kann der Leser einen Bedingungsausdruck erkennen, dessen Aktionsanweisung der unbedingten Anweisung davor stark ähnelt.

```
int TBraunPsd::GetBufferSize( void )
{
  uCBufferLength = ( ( 4096 >> uPositionScale ) + 18 );

  if ( uCBufferLength < ( ( 4096 >> uEnergyScale ) + 18 ) )
    uCBufferLength = ( ( 4096 >> uEnergyScale ) + 18 );

  return uCBufferLength + 1;
}
```

Indem man die gleich bleibenden Elemente ( hier das „+18“ ) aus dem bedingten Ausdruck herauszieht, wird der gesamte Abschnitt vereinfacht.

[9] Vgl. [MEY95], Seite 95ff : „Vermeiden Sie Datenelemente in der öffentlichen Schnittstelle“

```
int TBraunPsd::GetBufferSize( void ) const
{
   // Max(uPositionScale,uEnergyScale)
   int factor;
   if ( uPositionScale < uEnergyScale )
     factor = uEnergyScale;
   else
     factor = uPositionScale;

   return ( 4096 >> factor ) + 18 + 1;
}
```

Generell läßt sich diese Refaktorisierung für alle Codeabschnitte anwenden, bei denen in allen Zweigen eines Bedingungsausdrucks gleiche Operationen ausgeführt werden.

### 3.3.7 Bedingten Ausdruck durch Polymorphismus ersetzen

Switch-Anweisungen kommen in objektorientiertem Code relativ selten vor. Gerade Programmierer, die lange Zeit nur strukturiert programmiert haben, neigen dazu, auch in objektorientiertem Code Verzweigungen aufgrund von Typschlüsseln zu nutzen.

Beispiel

```
class Scan {
   enum { StepScan, Omega2ThetaScan, ContinuousScan } EScantype;
   EScantype m_scantype
   void DoScan();
}

void Scan::DoScan() {
      switch(scantype) {
            case StepScan:
                  ...
            case Omega2ThetaScan:
                  ...
            case ContinuousScan:
                  ...
        }
}
```

Das ist sicher (syntaktisch) korrektes C++, aber eines der wesentlichen Ideen der objektorientierten Programmierung wird hier nicht genutzt: Polymorphie.

Entitäten der Realität, die sich im Verhalten unterscheiden, sollten durch verschiedene Klassen modelliert werden, ihre Gemeinsamkeiten durch eine gemeinsame Oberklasse.

Zurück zum Beispiel: Nutzt man stattdessen mehrere Klassen, lässt sich der Unterschied im Verhalten durch virtuelle Funktionen ausdrücken.

```
class Scan {
      virtual void DoScan();
}

class StepScan : public Scan {
      void DoScan();
}

class Omega2ThetaScan : public Scan {
      void DoScan();
}

class ContinuousScan : public Scan {
      void DoScan();
}
```

Für solche Codetransformationen gibt es die Refaktorisierung „Typenschlüssel durch Unterklassen ersetzen“ aus dem Katalog von Fowler[10].

Im aktuellen XCTL-Projekt ist an vielen Stellen noch die erste Version implementiert. Im Ergebnis entstehen gigantische, nur mit großem Aufwand zerlegbare Methoden, die ich als den Hauptmangel des Projektcodes ansehe.

### 3.3.8 Methode/Feld umbenennen

Wenn es möglich ist, eine Methode oder ein Attribut durch einen treffenderen Namen zu beschreiben, und damit die Verständlichkeit zu erhöhen, dann sollte die Methode oder das Attribut umbenannt werden.

### 3.3.9 Felder/Methoden nach oben/unten verschieben

Diese Refaktorisierungen beziehen sich auf das Verlagern von Methoden oder Attributen in einer Vererbungshierarchie nach oben bzw. nach unten. Funktionalität oder Daten, die allen Subklassen gemein sind, können in die Superklasse verschoben werden. Analog dazu kann Verhalten oder Daten, die nicht allen Subklassen einer Klasse gemein sind, in der Hierarchie nach unten verlagert werden.

[10] [FOW00], Seite 227ff

Im der Detektoren-Hierarchie habe ich das Attribut bSound und die zugehörigen Methoden GetSound() und SetSound() aus der Basisklasse TDetector nach unten verschoben, weil nicht alle Subklassen von TDetector diese Daten bzw. dieses Verhalten haben sollen.

### 3.3.10 Template-Methoden bilden

Verwenden alle Subklassen einen gemeinsamen Algorithmus, der nur an einigen Stellen zwischen den Subklassen variiert, können Template-Methoden eingesetzt werden. Alle Subklassen erben das Verhalten ( den Algorithmus ) der Basisklasse, können aber die betreffenden Stellen mittels überschriebener virtueller Methoden modifizieren.

Durch den Einsatz von virtuellen Methoden in der Basisklasse, die nicht pur virtuell sind, kann ein modifizierbares Standardverhalten an die Subklassen vererbt werden.

Angewandt habe ich diesen Mechanismus z.B. in TDetector:

```
BOOL TDetector::SetParameters( void )
{
  // Wenn keine Messung lief, einfach Parameter setzen
  if ( bActive == FALSE )
          return _SetParameters();

  // Sonst Messung erst beenden, Parameter setzen und Messung wieder
starten
  MeasureStop();
  BOOL bRetCode = _SetParameters();
  MeasureStart();
  return bRetCode;
}
```

Subklassen haben mit _SetParameters() die Möglichkeit, innerhalb des von TDetector vorgegebenen Rahmenalgorithmus eigene Parameter zu setzen.

### 3.3.11 Vererbung durch Delegation ersetzen

Der Einsatz von Vererbung in der Objektorientierten Programmierung impliziert immer, dass die Subklasse zur Superklasse in einer „Ist ein“-Beziehung steht[11]. Es entsteht ein Interface, dass die Subklasse auch semantisch zu erfüllen hat.

[11] [MEY95], Seite 151ff : „Sorgen Sie dafür, dass öffentliche Vererbung „ist ein“ bedeutet“

Manchmal ist die Vererbung aber der Semantik nach nicht gerechtfertigt, weil die „Ist ein“-Beziehung gar nicht besteht. Solche Konstruktionen sieht man im XCTL-System leider zu oft. Klassen werden von anderen abgeleitet, um einfach auf deren Daten zuzugreifen:

TScan ( eine Meßverfahrensklasse ) ist von TScanParameters ( einer Oberflächenklasse ) abgeleitet, um einfach an die Einstellungen für die nächste Messung zu gelangen. TScan ist aber keine Oberflächenklasse!

TGenericDetector war von TAm9513 abgeleitet, um einfach das für die Hardwarekommunikation nötige Verhalten zu erben. TGenericDetector ist aber keine Spezialisierung von TAm9513, sondern eine Detektorklasse.

In beiden Fällen war die Vererbung ein mit geringem Aufwand verbundener Weg, das Implementationsziel schnell zu erreichen. Leider sind die langfristig damit erkauften Nachteile enorm.

Im Bereich des Detektoren-Subsystems habe ich daher die Mehrfachvererbung für TGenericDetector zugunsten einer Nutzungsbeziehung beseitigt. TGenericDetector nutzt jetzt ein TAm9513-Objekt für die Hardwarekommunikation[12].

[12] Im Sinne der Objektorientierung also eine „hat ein“-Beziehung

# 4 Refactoring im Detektoren-Subsystem

*Haben Sie schon einmal den gleichen dummen Fehler zweimal gemacht? Willkommen in der realen Welt. Haben Sie schon einmal den gleichen dummen Fehler hundertmal nacheinander gemacht? Willkommen in der Welt der Software-Entwicklung.*

*Tom DeMarco*[13]

Im Folgenden soll beschrieben werden, wie ich Teilprobleme des Detektoren-Subsystems mit Hilfe von Refaktorisierungen bearbeitet habe.

Die vorhandenen Probleme stellen dabei ein mehrdimensionales Gebilde dar. Einige Qualitätsmängel finden sich in der gesamten XCTL-Software bzw. im gesamten Detektoren-Subsystem. Wenn eine Änderung nur dann sinnvoll nutzbar ist, wenn sie am Gesamtsystem durchgeführt wird, dann habe ich das getan. Alle anderen Änderungen habe ich nur in den Codeabschnitten durchgeführt, die dem Detektoren-Subsystem zuzuordnen sind, damit Umfang und Komplexität der Arbeit in einem durch eine Einzelperson (durch mich) kontrollierbaren Rahmen bleibt. Ich werde in diesem Rahmen also Probleme beschreiben, die mehrere Teile des Detektoren-Subsystems und meistens auch andere Abschnitte des XCTL-Systems betreffen.

Auf der anderen Seite konnte das Gesamtproblem „Detektoren-Subsystem“ in mehrere problematische Subsystemteile zerlegt werden. Ich werde natürlich auch die Bearbeitung dieser Teilprobleme darstellen. Damit ergibt sich eine Überschneidung zwischen globalen Problemen und Problemen in den Subsystemteilen. Wenn ein Problem also nicht spezifisch für einen Subsystemteil ist, wird es als globales Problem dargestellt.

Globale Probleme sind in erster Linie Probleme im Umgang mit der Sprache C++ bzw. Probleme im Design und der Nutzung von Softwareschnittstellen. Lokale Probleme sind in erster Linie Probleme in der Implementation von Algorithmen und in der Verteilung von Zuständigkeiten (Designprobleme).

In den folgenden Abschnitten soll der Wert von Refaktorisierungen als Mittel zum schnellen Erreichen von qualitativen Codeverbesserungen gezeigt werden.

[13] [TDM97], Seite 147

Ich habe zu jedem Teilproblem eine Lösung gefunden, die mindestens eine der gravierenden Schwächen des Ausgangscodes nicht mehr aufweist und dabei die Ausprägung aller anderen Qualitätsmerkmale nicht verschlechtert. Durch die Refaktorisierung wird ein Code erzeugt, der zumindest besser als der Ausgangscode ist, das Ergebnis der Transformation ist also weder optimal noch die einzig mögliche Lösung.

## 4.1 Bezeichner

Hinter Bezeichnern (englisch identifier) verbirgt sich sowohl ein abstraktes als auch ein reales Konzept. Das reale Konzept ist der Code, für den der Bezeichner steht; das abstrakte Konzept ist die Semantik, die in dem Codeabschnitt steckt. Sowohl Datenelement- als auch Funktionsbezeichner sollen gegenüber Menschen und Compilern einen Sachverhalt beschreiben. Den Compilern ist dabei die Semantik des Bezeichners völlig gleich, er sollte aus einer gültigen Kombination von Zeichen bestehen, sich von allen anderen Bezeichnern unterscheiden und gleiche Sachverhalte sollten durch gleiche Bezeichner beschrieben werden.

Für Menschen haben Bezeichner einen anderen Wert. Natürlich sollten auch für den Menschen die Anforderungen des Compilers an Bezeichner erfüllt sein, da sonst für ihn der Übersetzungsprozess nur wenig zufrieden stellend enden würde. Die Semantik des Bezeichners ist dagegen von wesentlich größerer Bedeutung. Beschreibt der Bezeichner exakt das, wofür er steht, stellt er eine (korrekte !) Metainformation über den Code bereit. Für die Korrektheit ist allerdings die oben erläuterte semantische Konsistenz erforderlich.

Ist die semantische Konsistenz zwischen Bezeichner und Bezeichnetem nur unzureichend, stehen zwei relativ einfache Refaktorisierungen zur Verfügung: „Datenelement umbenennen“ und „Methode umbenennen“.

Im Kern sind diese Refaktorisierungen rein syntaktischer Art, die Semantik von Codeabschnitten wird dadurch nicht beeinflusst. Am besten ändert man die Deklaration des zu ändernden Bezeichners und lässt dann den C++-Compiler für sich arbeiten. Die strenge Typprüfung des Compilers wird jede nötige Umbenennung finden und auf ihre Notwendigkeit hinweisen.

Allerdings ist das eben ein syntaktisches Problem, eine rein lexikalische Ersetzung von allen zutreffenden Bezeichnern würde nur im Ausnahmefall zum richtigen Ergebnis führen. Ein Werkzeug, das solche Refaktorisierungen unterstützen soll, müsste mindestens über einen Parser für die Implementationssprache verfügen.

Die Umbenennungs-Refaktorisierungen habe ich häufig angewendet. Beispiel: Eine der umfangreichsten Umbenennungen war die Umbenennung von TDevice in TDetector. Ursprünglich hieß die Wurzelklasse aller Detektoren TDevice. Device heißt aber übersetzt „Vorrichtung, Gerät“; es würde also auch z. B. die Motoren beschreiben. TDetector hingegen beschreibt eindeutig einen Detektor.

## 4.2 Subsystem-Konsolidierung

Das Detektoren-Subsystem stellt eine historisch gewachsene Komponente dar, die vom Rest des XCTL-Systems genutzt werden kann. Kai Schützler hatte in seiner Diplomarbeit[14] die ehemals in einem gemeinsamen Verzeichnis liegenden Quelltextdateien des Projektes auf ihre Subsystemzugehörigkeit untersucht und sie in entsprechend benannte Unterverzeichnisse eingeordnet. So entstand u. a. das nach dem Anwendungsfall „Detektorennutzung“ benannte Verzeichnis „detecuse“, in dem sich alle Quelltextdateien befinden sollten, die zum Anwendungsfall „Detektorennutzung“ und damit zum Detektoren-Subsystem gehörten.

Im Rahmen meiner Studienarbeit musste ich aber feststellen, dass dadurch noch keine sauberen Subsystemgrenzen geschaffen waren. Code, der zum Detektoren-Subsystem zugerechnet werden musste, befand sich in anderen Quelltextdateien außerhalb der Verzeichnisses „detecuse“, während sich andererseits in den Dateien im Subsystemverzeichnis Code befand, der nicht zum Detektoren-Subsystem gehört.

Das Detektoren-Subsystem soll als Komponente einzeln übersetz- und nutzbar sein. Es soll theoretisch möglich sein, sie in mehreren Projekten wiederzuverwenden. Daher ist ihre Vollständigkeit von großer Bedeutung. Alle Funktionen, die zur Nutzung der Detektoren zur Verfügung stehen sollen, müssen sich innerhalb dieser Komponente befinden.

Andererseits wird die Softwarewartung erschwert, wenn sich in dieser Komponente Code befindet, der nicht zum Detektoren-Subsystem gehört. Änderungen an diesem nichtzugehörigen Code erfordern das Neuerstellen der Detektoren-Komponente; stellen also eine unnötige Erstellungsabhängigkeit dar. Solcher Code erhöht die Kopplung zwischen den Subsystemen.

Zum Subsystem gehörenden Code im Verzeichnis „detecuse“ zusammenzuführen und nicht zugehörigen Code in andere Subsysteme zu transferieren, würde die Softwarequalität erhöhen.

[14] https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98y/publikationen/diplomarbeiten/Subsysteme.ps

Für das Gesamtprojekt ändert sich dadurch das von außen beobachtbare Verhalten des Codes nicht. Die Änderungen wirken sich nur auf die Verbindungen zwischen den Systemkomponenten aus. Solche strukturverbessernden Maßnahmen wurden schon durchgeführt, bevor es dafür den Begriff „Refactoring“ gab.

Der Anwendungsfall „Nutzung der Detektoren“ bestimmt die inhaltlichen Anforderungen an das Detektoren-Subsystem. Zum Detektoren-Subsystem gehören alle die Codeabschnitte, die mindestens eine der folgenden Funktionalitäten realisieren:

- Verwalten der verfügbaren Detektoren
- Verwalten der Detektorkonfiguration
- Einstellen von Hardwareparametern
- Einstellen von Messungsparametern, die mit der Funktion eines Detektors zusammenhängen
- Kommunikation mit der Detektor-Hardware

Ich habe zwei Klassen gefunden, die entsprechend dieser Klassifizierung falsch platziert waren.

TCalibratePsd ist eine Oberflächenklasse, mit der ein PSD vor der Messung kalibriert werden kann. Sie ist zum Detektoren-Subsystem zuzurechnen, weil sie die Funktion „Einstellen von Messungsparametern“ erfüllt. Stattdessen war die Klasse Teil des Hauptprogramms. TCalibratePsd arbeitet hauptsächlich mit den Daten von Detektorobjekten, wodurch sich der Codegeruch „Neid“ über Subsystemgrenzen hinweg verbreitete.

Der Transfer in das Detektoren-Subsystem bedeutete die Integration der Klasse in die Dateien „detecgui.cpp/.h“ und das Verschieben der Dialogfenster-Definitionen in die Ressourcen-Dateien counters.rc. Allem Anschein nach war diese Klasse deshalb Teil des Hauptprogramms, weil sich dadurch das Dialogfenster mit weniger Aufwand aktivieren ließ. Den Aufwand, den der Entwickler damals auf Kosten der Struktur „gespart“ hatte, musste ich jetzt zum Bereinigen der Strukturschwäche erbringen. Ich habe innerhalb der neuen Utility-Klasse DetectorGUI eine Methode zum Erzeugen und Aktivieren eines TCalibratePsd-Fensters bereitgestellt und diesen neuen Teil der Subsystemschnittstelle an den entsprechenden Passagen im Hauptprogramm integriert.

Im Gegensatz dazu stellte sich bei der Klasse TCounterWindow heraus, dass sie der obigen Klassifizierung entsprechend nicht zum Subsystem gehört. TCounterWindow stellt die Messwerte eines aktiven Detektors dar. Es handelt sich also um eine Form der Messwert*verwendung*, analog zu den Scan-Klassen von Topographie und Diffraktometrie. Daher habe ich hier den umgekehrten Weg beschritten und die Klasse TCounterWindow nebst zugehörigen Ressourcen-Elemente in das Hauptprogramm verschoben.

## 4.3 Umgang mit totem Code

Wie schon im ersten Kapitel erwähnt, ist der Einsatz von fertigen Test-Tools im Augenblick nicht möglich. Mit Tools zum Bestimmen der Code-Überdeckung ließe sich feststellen, welche Codepassagen tatsächlich genutzt werden.

Wäre es (effizient) möglich, alle ungenutzten Codepassagen zu ermitteln und aus dem Code zu entfernen, würde der Code kürzer, prägnanter, lesbarer und wartbarer – kurz, die Qualität würde erhöht[15]. Bei Codeabschnitten kann es sich sowohl in syntaktischer als auch in semantischer Hinsicht um toten Code handeln.

Insbesondere beim Reengineering von schlecht dokumentiertem Altcode treffen hier zwei konträre Zielstellungen aufeinander: nicht verwendeter Code muss entfernt werden, da nicht vorhandenem Nutzen erheblicher Mehraufwand entgegensteht. Andererseits stecken in solchem Code möglicherweise wichtige Informationen, die durch das Entfernen nicht mehr zur Verfügung stehen.

Daher muss der semantische Wert aller derartigen Codepassagen per Codeinspektion durch den Entwickler überprüft werden. Möglicherweise hängen diese nichtgenutzten Codeabschnitte voneinander ab und müssen daher gemeinsam betrachtet werden. Möglicherweise wird der Abschnitt nur durch einen vergessenen Aufruf nicht aktiviert, obwohl er eigentlich ein wertvoller Abschnitt ist. Möglicherweise steckt in ihm zumindest eine gute Idee eines früheren Entwicklers.

Um Informationen aus solchen Abschnitten zu bewahren, gibt es je nach der Wichtigkeit der Informationen zumindest zwei grundlegende Maßnahmen: Wichtige Informationen müssen in der Entwicklerdokumentation festgehalten werden, weniger wichtige Informationen werden durch ein vernünftiges Quellcode-Managementsystem bewahrt.

[15] [SNE91], Seite 62: „Programme, die lange leben, werden durch die toten Funktionen und Daten der Vergangenheit immer stärker belastet. Deshalb muss Software in regelmäßigen Abständen bereinigt werden. Durch die Bereinigung gewinnt sie eine neue Gestalt und befreit sich von unnötigem Ballast. Die Software wird sozusagen neugeboren.“

Sind die Informationen konserviert, gilt eine weitere wichtige Regel: Kein spekulatives Design[16]. Wird eine Funktion eventuell in Zukunft benötigt, dann soll sie auch in Zukunft realisiert werden, wenn feststeht, dass sie auch tatsächlich benötigt wird. Durch Refactoring sind solche Veränderungen zu einem späteren Zeitpunkt einfach zu realisieren. Alle Eventualitäten zu berücksichtigen ist unmöglich; Ansatzpunkte für eventuelle Erweiterungen vorab zu integrieren teuer.

Derartige Codeabschnitte im Code zu behalten bedeutet, sie mitpflegen zu müssen. Deaktivierte/auskommentierte Codeblöcke unterliegen noch stärkerer Erosion als normaler Produktionscode, weil in diesen Abschnitten nicht einmal der Compiler sicherstellt, ob der Abschnitt nach der letzen Änderung noch übersetzbar bleibt.

Gefährlich daran ist eben, dass der Code an sich funktioniert; der Compiler übersetzt das Projekt anstandslos und das Programm tut in etwa, was es soll. Wird nach einem halben Jahr der bis dahin auskommentierte Block wieder aktiviert, so ist er höchstwahrscheinlich nicht mehr fehlerfrei übersetzbar. Der betreffende Entwickler erbt dann die dankbare Aufgabe, den Abschnitt mit seinen ohnehin enthaltenen Fehlern an den aktuellen Entwicklungsstand anzupassen. Es entsteht also Aufwand, der eigentlich einer früheren Entwicklungsperiode (und den damals verantwortlichen Entwicklern) zuzurechen wäre.

Daher sehe ich in dieser Situation nur zwei mögliche Auswege: Erstens: Der inaktive Code wird entfernt und bei eventuellem späteren Bedarf nach altem Vorbild neu erstellt oder zweitens, es gibt für jeden inaktiven Codeabschnitt einen Verantwortlichen ( äquivalent zum „Maintainer“ in Open-Source-Projekten ), der für die Wartung dieses Abschnittes verantwortlich ist.

Wenn die Projektleitung tatsächlich wünscht, dass toter Code mitgepflegt wird, muss sie sich des entsprechenden Mehraufwandes bewusst sein, insbesondere bei der Projekt-Aufwandsabschätzung. Metriktools, die den Projektleitern bei der Aufwandsabschätzung helfen sollen, lassen üblicherweise Code in Kommentaren außer Acht und lassen so ein falsches Bild vom tatsächlichen Aufwand entstehen.

Einige Arten von syntaktisch totem Code lassen sich relativ gut mit Compilern aufspüren, indem man das Warn-Niveau beim Erstellungsprozess erhöht. So meldet der Borland C++-Compiler Variablen, die zwar deklariert werden, denen aber nie ein Wert zugewiesen wird. Das sind sicher tote Codeelemente.

[16] spekulatives Design verströmt den Geruch von „Spekulativer Allgemeinheit“.

Innerhalb von Funktionen können Kontrollflusspfade existieren, die nie erreicht werden können. Einige davon, allerdings ausschließlich solche, die Aufgrund der Sprachsyntax nicht erreicht werden können, können vom Compiler entdeckt werden.

Beispiel:

```
int foo(int arg) {
      int i=7, j, k=8;

      if (i<5)
            return 12;
      return 10;

      i=28;
}
```

Diese Funktion enthält mehrere tote Code-Elemente. Der Borland-C++ Compiler würde zwei( rein syntaktische ) Probleme erkennen und melden:

- Die Variable j, die deklariert, jedoch nie verwandt wird
- Die Zuweisung i=28, zu der kein erreichbarer Kontrollflusspfad existiert

Durch die Nutzung von Defs/Uses-Verfahren könnte festgestellt werden, dass der Variablen k zwar ein Wert zugewiesen wird (def), dieser allerdings nie genutzt wird.

Durch einen logischen/semantischen Fehler im Code kann die Zeile „return 12“ nie ausgeführt werden. Der Kontrollflusspfad kann nicht durchlaufen werden, weil die Vorbedingung „i<5“ nie erfüllt sein wird. Solche Fehler können vom Compiler gar nicht gefunden werden. Dazu müsste man Codetestsysteme nutzen oder den Code mit offenen Augen manuell überprüfen.

Das Aufspüren von totem Code auf der Ebene der Funktionen und Methoden ist um einiges schwieriger.

Bei strukturierten Programmiersprachen wie C war die Frage nach der tatsächlichen Nutzung einer Funktion relativ einfach zu entscheiden. Wenn eine uses/used-by-Analyse ergab, dass eine Funktion nie aufgerufen wird, dann war sie definitiv toter Code. Gab es im Code jedoch Stellen, an denen eine Funktion aufgerufen wurde, so kann nur nach einem Codeüberdeckungstest eine verbindliche Aussage getroffen werden.

Während meiner Arbeit am Detektoren-Subsystem habe ich mit der Cross-Referenzer-Komponente von doxygen Funktionen gefunden, die nie verwendet wurden. Solche Funktionen habe ich aus dem Code entfernt.

Im Bereich der objektorientierten Programmierung, wie im XCTL-Projekt mit C++, ist das Aufspüren von toten Methoden schwieriger. Hier muss der statische und der dynamische Typ jedes Objektes berücksichtigt werden, für das eine Methode aufgerufen wird.

Einfacher waren die Fälle zu bearbeiten, wo bekannt war, dass eine Funktionalität vorhanden war, aber nicht mehr benötigt wurde. So war z.B. Code für die Unterstützung einer Matrox-Framegrabberkarte vorhanden, die nicht mehr verwandt wurde. Aus den oben angeführten Gründen habe ich die Unterstützung für diese nicht verwandte Karte entfernt.

## 4.4 „Algorithmus ersetzen“

Man betrachte folgende Methode:

```
int TBraunPsd::GetBufferSize( void )
{
  uCBufferLength = ( ( 4096 >> uPositionScale ) + 18 );

  if ( uCBufferLength < ( ( 4096 >> uEnergyScale ) + 18 ) )
    uCBufferLength = ( ( 4096 >> uEnergyScale ) + 18 );

  return uCBufferLength + 1;
}
```

Erst beim Versuch, duplizierten Code zu eliminieren, verstand ich, dass dieser Code auf unnötig komplizierte Weise etwas Einfaches tat: Ermittle das Maximum von uPositionScale und uEnergyScale und berechne damit eine Puffergröße.

```
int TBraunPsd::GetBufferSize( void ) const
{
   // Max(uPositionScale,uEnergyScale)
   int factor;
   if ( uPositionScale < uEnergyScale )
     factor = uEnergyScale;
   else
     factor = uPositionScale;

   return ( 4096 >> factor ) + 18 + 1;
}
```

Beide Algorithmen liefern bei gleichen Eingangsdaten dasselbe Ergebnis. Der zweite verwendet statt des Klassenattributs uCBufferLength, das ohnehin nur in dieser Methode verwandt wird, eine temporäre Variable, deren Name ihre Funktion erklärt. Der zweite Algorithmus verzichtet darauf, mehrfach dieselbe Berechnung durchzuführen, obwohl sie für das Ermitteln des Maximums nicht erforderlich war.

Ein anderes merkwürdiges Konstrukt fand ich in:

```
     for(;;)
           {
           if(Header.HighADCCounts < 6000l)
                 { nDelay += 1; break; }
           if(Header.HighADCCounts < 8000l)
                 { nDelay += 2; break; }
           if(Header.HighADCCounts < 12000l)
                 { nDelay += 3; break; }
           if(Header.HighADCCounts < 15000l)
                 { nDelay += 4; break; }
           nDelay = 5;
           break;
           }
```

Selbst in C ist das einfach schlechter Programmierstil. Eine mit break abgebrochene Endlosschleife, die ohnehin nur einmal durchlaufen wird, widerspricht allen Grundsätzen der strukturierten Programmierung. Meine Version lautet:

```
if ( Header.HighADCCounts < 6000 )
  nDelay += 1;
else
  if ( Header.HighADCCounts < 8000 )
    nDelay += 2;
  else
    if ( Header.HighADCCounts < 12000 )
      nDelay += 3;
    else
      if ( Header.HighADCCounts < 15000 )
        nDelay += 4;
      else
        nDelay = 5;
```

Das entspricht dem Schulbeispiel der Mehrfachverzweigung, die auch von durchschnittlich befähigten Programmierern wie mir sofort als solche erkannt wird.

Diese beiden Beispiele sollen folgendes zeigen: Im Detektoren-Subsystem ist an vielen Stellen unsauber oder unnötig kompliziert programmiert worden. Moderne Programmiersprachen erlauben es, einen Algorithmus auf viele verschiedene Arten zu realisieren. Abgesehen von Unterschieden, die aus dem ästhetischen Empfinden der jeweiligen Entwickler resultieren, gibt es Realisierungsmöglichkeiten, die schlechter sind als andere.

Wenn es das Verständnis, die Effizienz oder die Wartbarkeit des Codes verbessert, ist es sinnvoll, eine andere Realisierung desselben Algorithmus einzusetzen.

## 4.5 Das Geheimnisprinzip

Im gesamten XCTL-System werden Grundprinzipien der objektorientierten Programmierung verletzt.

Das Geheimnisprinzip (information hiding):

„Eine Änderung oder Abfrage des Zustandes eines Objekts ist nur über seine Operationen möglich, d.h. die Attributwerte und Verbindungen sind außerhalb des Objektes nicht sichtbar.“[17] Die Details der Implementation sollen vor dem Nutzer einer Klasse verborgen werden; er soll nur das Interface sehen. Dadurch wird die Komplexität des Codes verringert, da dem Anwender nur der für ihn relevante Teil angeboten wird.

Durch die Trennung von Interface und Implementation gewinnt man die Flexibilität, die Implementation ändern zu können. Hinter dem Interface verbergen sich auswechselbare Codeteile. In C++ stellt nur das Interface einer Klasse den abstrakten Datentyp dar. Typ und Wert der Attribute hingegen sind Interna der Klasse.

Sind Attribute eines Objektes öffentlich, so können sie von anderen Objekten modifiziert werden, ohne dass das Objekt davon erfährt. Da sich der Objektzustand oft aus mehreren Attributen zusammensetzt, die voneinander abhängen, kann hier ein inkonsistenter Zustand entstehen. So werden Daten, Zustand und Objektverhalten voneinander getrennt. Es ist aber gerade ein geschätzter Vorteil der objektorientierten Programmierung, Daten und Verhalten zusammen zu betrachten und auf die Klasse als Organisationsform zu konzentrieren. Damit wird das Verstehensaufwand des dahinterliegenden Konzeptes, die Fehlersuche und der Änderungsaufwand lokal beschränkt.

[17] [BAL98], Seite 156

Um die Einhaltung des Geheimnisprinzips wieder durchzusetzen, habe ich den Code so transformiert, dass er dem Grundsatz „Es darf keine öffentlichen Attribute geben“ entsprach. In Fowlers Maßnahmenkatalog existiert dafür die Refaktorisierung „Feld kapseln“[18]. Dazu habe ich sukzessiv den Zugriffsschutz für alle ehemals als public deklarierten Attribute auf protected geändert. Anschließend hilft der Compiler, alle Stellen im Code aufzufinden, an denen bisher direkt auf Attribute zugegriffen wurde.

In einigen Fällen meldete der Compiler keine Verletzung von Zugriffsregeln, dann wurde auf das Attribut höchstens innerhalb der Klassenhierarchie zugegriffen. In allen anderen Fällen habe ich nun für das entsprechende Attribut Zugriffsmethoden hinzugefügt: eine lesende Accessor-(Get-) und eventuell eine schreibende Mutator-(Set-)Methode. Die Codepassagen mit Attribut-Direktzugriffen habe ich dann durch die entsprechenden Zugriffsmethoden ersetzt.

Das Interface des Detektorensubsystems war von dieser Maßnahme stark betroffen, war es doch bis dato üblich, Interna des Subsystems aus anderen Subsystemen heraus direkt zu modifizieren. In einem zweiten Schritt habe ich mit einem Cross-Referenzer (doxygen) ermittelt, welche der Methoden und Attribute in den Zugriffsmöglichkeiten weiter eingeschränkt werden konnten, indem sie als protected bzw. private deklariert wurden (Fowlers Refaktorisierung „Methode verbergen“[19]). Damit konnten einige der Zugriffsmethoden wieder aus dem Subsystem-Interface herausgelöst werden, da sie nur innerhalb der Klassenhierarchie verwandt werden. Neben der beseitigten Nachteile, die sich aus öffentlichen Attributen ergeben (siehe oben), trugen diese Refaktorisierungen sowohl zum Interface-Recovery für das Subsystem als auch zur Trennung von Logik und Präsentation bei.

## 4.6 Falsche Freunde

In einem weiteren Schritt habe ich mich dem Problem der friend-Deklarationen angenommen. Sie ermöglichen einer Klasse oder einer Funktion uneingeschränkten Zugriff auf die Attribute einer anderen Klasse. Es ist also ein vorsätzliches Durchbrechen des Geheimnisprinzips.

[18] [FOW00], Seite 209f

[19] [FOW00], Seite 312f

Daher ist die Rolle der friend-Deklarationen sehr zwiespältig: Beim Überladen von Operatoren sind friend-Deklarationen zum Teil unverzichtbar, in allen anderen Fällen sind sie eher ein Zeichen für schlechtes Design. Die Erfinder von Java haben z. B. ein Konstrukt wie friend in ihrer Sprache gar nicht vorgesehen, weil es nicht gutem objektorientierten Stil entspricht.

Mit friend-Deklarationen kann man ein kleineres Klasseninterface erreichen, wenn nur sehr wenige Klassen oder Funktionen tatsächlich Zugriff auf Klasseninterna haben sollen. Man deklariert dann die entsprechenden Methoden als protected bzw. private, und erlaubt einigen explizit angegebenen Klassen bzw. Funktionen den ungehinderten Zugriff auf diese Methoden.

Die Nachteile eines solchen Konstruktes sprechen aber klar dagegen. Um die Klasse A zum friend von Klasse B zu machen, muss der Quellcode von B modifiziert werden[20]. Bei Verwendung von Bibliotheken oder kommerziellem Code ist das aber oft gar nicht möglich oder nicht erlaubt. Friend-Beziehungen sind nicht transitiv und nicht vererbbar. Somit stehen sie der Wiederverwendung von Code im Wege. Sie verschleiern Abhängigkeiten und erhöhen die Kopplung zwischen Subsystemen.

Im Falle des XCTL-Systems sind friend-Beziehungen über Subsystemgrenzen hinweg durchaus nicht selten anzutreffen, manchmal hat eine Klasse sieben verschiedene friends. Friends wurden hier aber in erster Linie eingesetzt, um den Änderungsaufwand auf Kosten des Designs und der Codequalität gering zu halten.

Das Auflösen von friend-Beziehungen verlief ähnlich wie das Wiederherstellen des Geheimnisprinzips. Dafür gibt es in Fowlers Katalog keine explizit aufgeführte Refaktorisierung. Dem Sinn nach entspricht es aber den Refaktorisierungen „Feld kapseln“ bzw. „Methode verbergen“. Ich habe einfach sukzessiv die friend-Deklarationen auskommentiert und den Compiler nach Zugriffsverletzungen suchen lassen. Die illegalen Direktzugriffe wurden dann wieder durch die entsprechenden Zugriffsmethoden ersetzt. Ähnlich wie die Refaktorisierungen im vorhergehenden Abschnitt trugen auch dieses Refaktorisierungen sowohl zum Interface-Recovery als auch zur Trennung von Logik und Präsentation bei.

[20] [CSTF], „You cannot (in a standard conforming program) grant yourself access to a class without modifying its source.”

## 4.7 GUI-Entkopplung

Das objektorientierte Paradigma der Trennung von Anwendungs- und Präsentationslogik wurde im Detektoren-Subsystem verletzt. Für jedes Oberflächen-Dialogfenster existiert eine Klasse, die die Präsentationslogik repräsentieren soll. Im XCTL-Projekt sind allerdings sind auch Teile der Anwendungslogik in den Oberflächenklassen enthalten. Für jedes Eingabeelement der Oberfläche wird eine Plausibilitätsprüfung vorgenommen, bevor der Wert in der Dokumentklasse gespeichert wird. Durch die Art und Weise, wie der Wert aus dem Oberflächenelement ausgelesen wird, wird in gewisser Näherung der erforderliche Datentyp erzwungen.

Beispiel: Aus einem Feld, das eine Realzahl enthalten soll, wird der Datenwert mit einer Funktion ermittelt, die eine Realzahl liefert. Das ist sicher sinnvoll und stellt nur eine minimale Abhängigkeit von der Anwendungslogik dar.

Allerdings wurden in den Oberflächenklassen auch Intervallprüfungen für die Werte vorgenommen, und falls notwendig, die Werte so modifiziert, dass sie in das zulässige Intervall passten. Um diese Intervallprüfung vorzunehmen, muss die Oberflächenklasse Informationen über die zulässigen Bereichsgrenzen haben. Unterschiedliche Oberflächenklassen, die dieselbe Dokumentklasse repräsentieren, können so mit verschiedenen zulässigen Bereichen arbeiten, die für die jeweils anderen Oberflächenklassen bzw. für die Dokumentklassen ungültige Werte zulassen. Das ist eine nicht mehr hinzunehmende Abhängigkeit von der Anwendungslogik. Diese Intervallprüfung musste in die Dokumentklassen verschoben werden und ist somit für alle Repräsentationen verbindlich.

Dieses Problem ging in den meisten Fällen mit dem Problem der Verletzung des Geheimnisprinzips einher. Die Oberflächenklassen waren als friend der Detektorklassen deklariert und schrieben einfach direkt in die entsprechenden Attribute.

Zur Problemlösung habe ich das im vorigen Abschnitt erläuterte Verfahren zur Auflösung von friend-Beziehungen angewandt. In die neue Mutatormethode wurde dann der Code für die Intervallprüfung verschoben, so dass die Einhaltung der Intervallgrenzen immer(!), an einer Stelle(!) und mit denselben(!) Grenzwerten geprüft wird. Anschließend konnte der Code für den Intervallgrenzentest aus den Oberflächenklassen entfernt werden.

Im Ergebnis wurden die Methoden der Oberflächenklassen kleiner und verständlicher, die Methoden der Anwendungslogik sicherer und plausibler.

Zu Beginn der Überarbeitung der Oberflächenelemente des Detektorensubsystems stand die Zusammenführung aller UI-Bestandteile in eine einzige Übersetzungseinheit. Die Dialogklassen befinden sich jetzt in den Dateien detecgui.cpp und detecgui.h.

Im Interface des Detektorensubsystems findet man jetzt die neue Utility-Klasse TDetectorGui, mit der den Klienten des Subsystems alle erlaubten Aktionen bezüglich der Oberfläche zur Verfügung gestellt werden (Fassaden-Strukturmuster / Facade-Pattern)[21]. Über ein statisches Methodenobjekt kann ein Subsystemklient jetzt das Erstellen und Ausführen eines Dialogobjektes anfordern.

Das Interface der GUI-Elemente des Detektoren-Subsystems wird nun ausschließlich innerhalb des Subsystems verwandt. Die Datei detecgui.h wird nur in den Implementationsdateien des Detektorensubsystems eingebunden. Nutzer der Subsystem-GUI brauchen keinerlei Kenntnis mehr über die entsprechenden Klassen und Strukturen. Damit wird die Kopplung zu anderen Subsystemen verringert.

Die GUI-Komponenten des Detektorensubsystems bestehen aus den Klassen

- TCommonDevParamDlg
- TScsParametersDlg
- TPsdParametersDlg
- TCalibratePsdDlg
- TDetectorGUI

Für alle Dialogfensterklassen des Projektes existieren die gemeinsamen Basisklassen TModalDlg und TModelessDlg für modale und nichtmodale Dialogfenster. In ihrem Wesen als Fensterklassen sind natürlich auch die Dialogklassen des Detektoren-Subsystems von diesen beiden Basisklassen abgeleitet. Die Funktionalität, die allen Dialogen gemein sein soll, ist in der Datei dlg_tpl.cpp implementiert. Um ihre Funktionalität nutzen zu können, wurde diese Datei ursprünglich für jedes Subsystem separat kompiliert und zum Zielcode der jeweiligen Erstellungseinheit gelinkt. Der aus der Implementationsdatei erstellte Code war also fünfmal identisch im Projektcode vorhanden.

[21] [GOF96], Seite 212ff

Der Versuch, die Basisfunktionalität der Dialogklassen in ein eigenes Subsystem zu transferieren, schlug fehl: Das System funktionierte nur dann, wenn für die Daten der Dialogbasisklassen in jedem Subsystem eine eigene Instanz angelegt wurde. Für jedes Subsystem existierte dann ein Zeiger theModeless, der auf das (einzige) momentan geöffnete nichtmodale Fenster zeigte. Innerhalb eines Subsystems sollte allein durch disziplinierte Implementation sichergestellt werden, dass genau ein nichtmodales Fenster verwaltet werden konnte und dass das aktuelle Fenster geschlossen wurde, bevor ein neues geöffnet wurde. Genau das wurde allerdings nicht sichergestellt, wenn man ein Dialogsubsystem implementierte. Dann existierte nur noch ein einziger theModeless-Zeiger, dessen Modifikation aus verschiedenen Subsystemen nicht mehr ausreichend synchronisiert wurde.

Mit diesem Problem befassen sich zurzeit Günther Reinecker und Thomas Kullmann. Sie haben die Situation dadurch entschärft, dass sie u. a. eine vernünftige Verwaltung der offenen Fenster implementierten.

Zumindest konnte ich die Projektdatei so abändern, dass der Code für die Dialogbasisklassen nur bei der Erstellung der splib.dll kompiliert wird, und zu allen weiteren Subsystemen nur noch gelinkt wird, was zumindest eine Zeitersparnis bei der Erstellung des Projektes bedeutet. Das Problem des fünfmaligen Vorhandenseins von Code kann erst dann gelöst werden, wenn die Oberflächenklassen in ein eigenes Subsystem umgezogen sind.

## 4.8 Hardware-Abstraktion

Das XCTL-Programm steuert Motoren und Detektoren. Alle Motoren und Detektoren verfügen über eine entsprechende Interface-Karte, die in den Messplatz-PC eingebaut ist. Das Programm muss also Steuersignale an diese Karten senden und die entsprechenden Antworten verarbeiten.

Auf Intel-basierten PC-Systemen existiert dafür ein spezieller, vom Speicher-Adressraum getrennter E/A-Speicherbereich, auf den nicht mit den üblichen Speichermanipulations-Operationen zugegriffen werden kann. Auf den so genannten Ports können nur mit den Assemblerbefehlen IN bzw. OUT 8, 16 bzw. 32 Bit breite Daten ein- bzw. ausgegeben werden.

Die Interface-Karten müssen so konfiguriert werden, dass sie Daten auf Ports austauschen, die von keiner anderen PC-Komponente genutzt werden. Kennt das XCTL-Programm die entsprechenden Ports, kann die Kommunikation stattfinden. Solche Einstellungen erhält das Programm in den Detektor- bzw. Motor-Abschnitten der hardware.ini.

Port-Ein- und Ausgaben wurden im XCTL-Projekt bisher immer mit inline-Assembler-Abschnitten im Quelltext nach folgendem Schema ausgeführt:

```
void foo(void)
{
   asm
  {
    // Ausgeben eines 16bit-Wertes
    mov ax, Wert
    mov dx, Port-Adresse
    out dx, ax

    // Einlesen eines 16bit-Wertes
    in ax, dx
  }
  return _AX;
}
```

Solcher Code ist prozessorabhängig, weil er nur auf i8086-kompatiblen PC-Systemen funktioniert. Das ist für das XCTL-Projekt allerdings kein Hindernis, da die Zielplattform auf absehbare Zeit ein solches PC-System sein wird.

Solcher Code ist compilerabhängig, weil der benutzte Compiler einerseits inline-Assembler unterstützen muss, und weil andererseits die Art, wie Prozessorregister innerhalb von C/C++ direkt manipuliert werden können, zwischen verschiedenen Compilern differiert. Man sollte nicht erwarten, dass bei einem anderen Compiler auch per _AX auf den Inhalt des Prozessorregisters AX zugegriffen werden kann.

Solcher Code ist betriebssystemabhängig, weil es auch auf Intel-Plattformen Betriebsysteme gibt, die den direkten Zugriff eines Nutzerprogramms auf die Hardware nicht gestatten. Als Vertreter dieser Kategorie seien Microsoft Windows NT, Windows 2000 und Windows XP genannt. Um die Systemsicherheit zu erhöhen, werden hier Features von Intel-Prozessoren ab dem 80286 verwandt, die im so genannten Protected Mode genutzt werden können. Die IOPL-Bits (Input Output Privileg Level) legen die aktuelle Privilegstufe beim Zugriff auf den E/A-Speicherbereich fest. Jedes Code-Segment kann mit einer Prioritätsstufe zwischen 0 und 3 versehen werden. Nur wenn diese Prioritätsstufe kleiner oder gleich dem IOPL-Wert ist, dürfen die Hardware-Operationen IN und OUT ausgeführt werden.

Damit wird durchgesetzt, dass nur Code des Betriebssystemkerns (Kernel-Mode-Prozesse) direkt auf die Hardware zugreifen kann. Wollen Nutzerprogramme ( User-Mode-Prozesse ) auf die Hardware zugreifen, müssen sie einen Kernel-Mode-Treiber benutzen, der eventuell konkurrierende Hardwarezugriffe koordinieren muss.

Da auf absehbare Zeit mit Windows 3.11 bzw. Windows 98 gearbeitet werden soll, ist das Problem zum jetzigen Zeitpunkt nicht vakant. Direkter Portzugriff bleibt vorerst die Regel.

Die Hardware-Abstraktion bestand aus vier Problemen:

- Alle Codeabschnitte für Port-Kommunikation waren in Anweisungen für bedingte Übersetzung eingeschlossen. Die zugehörige Bedingung war die Übersetzung mit Zielcode für 16-Bit-Windows. In Übersetzungen mit 32-Bit-Zielcode würden also gar keine Anweisungen für die Hardwarekommunikation enthalten sein, eine Nutzung von Detektoren bzw. Motoren in einer 32-Bit-Version der XCTL-Software wäre einfach nicht möglich.
- Diese Codeabschnitte waren in proprietärem Inline-Assembler formuliert. Hier sollte einerseits die Abhängigkeit von der Borland-C++-Entwicklungsumgebung beseitigt und andererseits die Wartbarkeit durch Transformation von einer 2GL-Sprache in die 4GL-Sprache C++ erhöht werden.
- Der Code für die Port-Kommunikation soll möglichst einfach erweiterbar sein, um später eventuell auch unter Windows NT und Nachfolgern lauffähig zu sein, bei denen kein direkter Portzugriff möglich ist.
- Die oben dargestellten inline-Assemblerabschnitte sind über den gesamten Code der Detektoren- und Motoren-Subsysteme verteilt

Hier hilft Fowlers Refactoring „Methode extrahieren“, um duplizierten Code in einer Funktion zusammenfassen zu können. Mehrfach angewandt für 8- und 16-Bit-Ein- und Ausgabefunktionen entstanden vier neue Funktionen. Da alle diese Funktionen auf gemeinsamen Daten operieren (dem Port), entstand die neue Klasse TIOPort. Alle Port-Ein- und Ausgaben wurden durch Methodenrufe eines TIOPort-Objekts ersetzt.

```
class _export TIOPort
{
  private:
    unsigned int port;

  public:
    TIOPort(unsigned int p);

    BYTE In8Bit( void );
    WORD In16Bit( void );
    void Out8Bit( BYTE d);
    void Out16Bit( WORD d);
};
```

Anschließend konnte das Refactoring „Algorithmus ersetzen“ angewandt werden: statt Assembler werden die Windows-API-Funktionen inp und inpw ( 8bit- bzw. 16bit-Porteingabe) bzw. outp und outpw (8bit- bzw. 16bit-Portausgabe) verwandt, so dass der Code für Windows 3.1 bis Windows ME übersetzt werden kann.

Der Code für Ein- und Ausgabe liegt also genau einmal in C++ vor. Soll zu einem späteren Zeitpunkt das Projekt so portiert werden, dass es unter Windows NT und Nachfolgern lauffähig ist, muss an *einer* Stelle der entsprechende Kernel-Mode-Treiber eingebunden werden - sicherlich ein klarer Vorteil gegenüber dem alten Zustand. Die Klasse TIOPort wurde in einer eigenen DLL zur Verfügung gestellt, damit die Software durch Austausch einer einzigen Bibliothek auf Windows NT und Nachfolgern zum Laufen gebracht werden kann.

Diese neue Klasse wird auch vom Motoren-Subsystem genutzt, es handelt sich also um eine Refaktorisierung über Subsystemgrenzen hinaus.

Und letztlich möchte ich es dem Urteil des Lesers überlassen, welche der folgenden Codepassagen mit gleicher Funktionalität die verständlichere ist:

Alt:

```
//Byte zum Controller uebertragen (Datenregister schreiben)
#ifndef __WIN32__
    _asm
    {
      mov dx,rd;  //Portadresse des Datenregisters in dx schreiben
      mov ax,d;   //zu uebertragendes Byte in ax schreiben
      out dx,al;  //Byte in das Datenregister des Controllers
                  //schreiben
    }
#endif
      //outp(rd,d); //zum Assemblercode aequivalente Windowsfunktion
```

Neu:

```
//Byte zum Controller uebertragen (Datenregister schreiben)
dataPort.Out8Bit(d);
```

### 4.8.1 Refaktorisierung des Radicon-Detektortreibers

Zu Beginn meiner Arbeiten bestand der Treiber für den Radicon-Detektor aus einer geringfügig modifizierten Beispielimplementation des Geräteherstellers. Der Hersteller, Radicon Ltd., hatte den Steuercode als Beispiel für eine mögliche Verwendung in eigenen Programmen in C formuliert. Zwei C-Dateien und drei Headerdateien realisieren eine Hardwaresteuerung in einer Dreischichtenarchitektur. Heute liefert der Hersteller dagegen eine DLL und zwei VisualBasic-Include-Dateien.

Die untere Schicht stellt Dienstprimitive zur Ein- und Ausgabe einzelner Zeichen bereit.

Die mittlere Schicht nutzt die Dienste der unteren, um Zeichenfolgen gesichert zu übertragen. Die obere Schicht nutzt die Dienste der darunterliegenden Schichten, um Steuerbefehle an den Controller zu senden bzw. Messwerte auszulesen. Im eigentlichen XCTL-Code nutzt dann die Klasse TRadicon Funktionen dieser Schichten.

Die Implementation des Treibercodes stellt insofern ein Problem dar, als dass sie aufgrund der Nutzung mehrerer Programmiersprachen und vor allem mehrerer Programmierparadigmen die Nutzung solcher CASE-Tools behindert, die entweder mit C oder mit C++ umgehen können. Da der Code in einer prä-OO-Sprache implementiert ist, wird es Entwicklern leicht gemacht, schlechten OO-Code zu entwickeln. So ist es zum Beispiel möglich, am Interface des Treibers vorbei direkt auf die mittleren und unteren Schichten zuzugreifen. Es fehlen wichtige Vorzüge der objektorientierten Programmierung: jede Funktion ist öffentlich nutz- und jede Variable direkt modifizierbar. Globale Zeiger und Datenpuffer stellen seit den Kinderzeiten der C-Programmierung eine der größten Fehlerquellen dar.

Wie in C üblich werden zahlreiche Konstanten per #define eingeführt. Alle diese Konstanten sind gleichwertig nach außen sichtbar, auch wenn sie eventuell nur innerhalb des Treibercodes eine Rolle spielen. Einigen der Konstantennamen lässt sich nur schwer ihre Bedeutung entnehmen ( so zum Beispiel CF5, CF9, CFB ).

Die Funktionsdefinitionen sind im veralteten Kernighan/Ritchie-Stil implementiert:

(Beispiel aus kisl1.c)

```
int CALLTYPE setprm( rd, rs, cph, cpl, ve, ftmr, imp, intg, snd, ntrp
)

  int rs, rd;
  unsigned short cph, cpl;
  int ve;
  double ftmr;
  unsigned long imp;
  int intg;
  int snd;
  int ntrp;
{
...
}
```

Einige CASE-Tools würden schon allein deshalb die Mitarbeit verweigern.

Im obigen Beispiel erkennt man noch ein zweites Problem: weder der Funktionsname noch die Namen der Parameter lassen ein black-Box-Verstehen des Codes zu. Keiner der Namen führt zu einer vernünftigen Vermutung über seine Semantik.

Im Bezug auf die Prüfung von Vorbedingungen lässt sich der Code problemlos optimieren:

( Beispiel aus kisl1.c: )

```
int CALLTYPE getinf( fun, rd, rs, ftmr, imp )
  int fun, rd, rs;

  double * ftmr;
  unsigned long * imp;

{
  register int i, j;
  double wftmr;
  unsigned long wimp, wtmr, ww;
  unsigned char * p;
  int l;
  /* buffer receive message */
  unsigned char bufmsg[LNGMSG];

#ifdef DEBUG
  retspot_set = retspot_beg = retspot_get = retspot_init =
    retspot_tr = retspot_rc = retspot_out = retspot_in = 0;
#endif // def DEBUG

  if (FakeDevice)
  {
    *ftmr = 0.0;
    *imp = 0;
    return 0;
  }
```

Sollte nun die Auswertung von FakeDevice tatsächlich wahr ergeben, wurden 8 Variablen und ein Datenpuffer umsonst angelegt und 8 Variablen umsonst mit 0 initialisiert. Der Grund für dieses Phänomen liegt eben in der Implementation in C. In C wird verlangt, dass alle in einem Block verwendeten Variablen deklariert werden, bevor der erste Code ausgeführt wird.

C++ lässt hier deutlich größere Freiheiten: Stroustrup empfehlen hier, eine Variable so spät wie möglich einzuführen und sie im selben Atemzug auch zu initialisieren[22]. Zuerst sollten also alle Vorbedingungen geprüft werden, und nur wenn sie alle erfüllt sind, und weitere Berechnungen tatsächlich erforderlich sind, sollten diese auch durchgeführt werden (lazy evaluation).

[22] [STRO00], Seite 14 : „Don't declare a variable before you need it so that you can initialize it immediately. A declaration can occur anywhere a statement can …"

Um eine einheitliche Codebasis für das Projekt zu bilden, sollte der Treibercode also nach C++ transformiert und an das Projekt angepasst werden. Bei Fowler gibt es dafür keine originäre Refaktorisierung, da Fowler sich in erster Linie auf die Java-Welt bezieht, in Java das Problem einer Umstellung von prozeduraler auf objektorientierte Programmierung jedoch keine Rolle spielt. Für die Radicon-Treiber entstand die neue Klasse TRadiconHW, die von der Adapterklasse TRadicon genutzt wird.

In einem ersten Schritt habe ich also alle Funktionen in eine Utility-Klasse überführt, bei der alle Methoden statisch sind. Das war deshalb sehr einfach, weil in C Daten und die Operationen darauf getrennt betrachtet werden. Bei einem Zusammenfassen verschiedener Funktionen zu einem Objekt muss daher nur auf den inhaltlichen Zusammenhang zwischen den neuen Methoden geachtet werden. Im Zuge dieser Umstellung habe ich die Funktionsdefinitionen im alten Kernighan/Ritchie-Stil durch ihre moderneren C++-Äquivalente ersetzt.

In einem zweiten Schritt habe ich entsprechende Zugriffsrechte auf die Methoden vergeben: nur die Interface-Methoden sind öffentlich deklariert, alle anderen privat. So wird sichergestellt, dass Code, der die neue Schichtenarchitektur nutzt, nur auf Dienste der obersten Schicht zugreifen kann; die Funktionen aller darunterliegenden Schichten bleiben dagegen verborgen.

Anschließend habe ich mich den #defines zugewandt: alle Konstanten, die nur innerhalb des Treibers verwendet werden sollen, sind innerhalb der Implementationsdatei mit dem Schlüsselwort const als Konstanten deklariert. Alle öffentlich sichtbaren Konstanten sind jetzt im Kontext der Klasse als enum deklariert. Damit können Fehler, die bisher erst zur Laufzeit festgestellt werden konnten, nun schon vom Compiler gefunden werden.

Beispiel:

In der alten Version waren viele Steuercodes als #define implementiert.

```
/*  receive result time and pulses */
#define CF9 0x09

/* receive current time and pulses */
#define CFB 0x0b
```

Eine Funktion, die einen solchen Wert als Parameter braucht, musste integer-Werte verarbeiten:

```
int CALLTYPE begwrk( fun, rd, rs )
  int fun, rs, rd;
{
...
}
```

Es war also möglich, Werte als Parameter für den Funktionscode zu übergeben, denen gar keine Funktion zugeordnet war. Um robuste Software zu schreiben, musste also Code hinzugefügt werden, der zur Laufzeit alle Werte abfing, die keine gültige Operation darstellten.

In der neuen Fassung ( und in C++ ) ist es dagegen möglich, mehrere konstante Werte als benannte Aufzählung zusammenzufassen, die einen eigenen Typ darstellen.

```
//! tells the detector's controller what to do
enum EOperationMode {
    fixedExposureCounts,  // count pulses during exposure time
    fixedImpulseCounts,  // count time when number of pulses is set
    intensimeter,    // intensimeter
    stop      // stop controller
};
```

Eine Funktion, die entsprechende Codes als Parameter erwartet, kann als Typ den Typ der Aufzählung nutzen:

```
int TRadiconHW::Execute( EOperationMode fun )
```

Der Compiler kann durch die strenge Typprüfung von C++ einen Fehler bei der Parameterübergabe zur Erstellungszeit erkennen; zusätzlicher Code zum Abfangen ungültiger Werte ist nicht erforderlich.

In einem vierten Schritt musste aus der Utility-Klasse eine echte Klasse gemacht werden:

Daten und die Methoden zu ihrer Manipulation mussten zusammengeführt werden. Das war insbesondere im Falle des Radicon-Detektors nicht trivial. Objekte mit eigener Identität können zum Beispiel auf unterschiedlichen Ports mit der Controllerhardware kommunizieren. Im alten Code war das aber gar nicht vorgesehen: die zu nutzenden Ports wurden in zwei globalen Variablen für alle Radicon-Detektoren gemeinsam gespeichert. Auch wenn die Initialisierungsdatei anderes suggerierte und vielleicht auch anderes beabsichtigt war, war es im alten Modell nicht möglich, mehr als einen Radicon-Detektor zu nutzen. Hätte man es dennoch versucht, wäre das nur durch unerwartetes Programmverhalten aufgefallen.

In TRadiconHW stehen nun für jede Operation auf der Hardware entsprechende Methoden bereit. Im Gegensatz zur C-Implementierung wurden einige Funktionen und Datenelemente zwischen diesen beiden Klassen verschoben.

Oft wurden die genutzten Ports in Methodenrufen als Parameter übergeben (Beispiel: Klasse TRadicon ). Zur Steigerung der Effizienz wurden die von den Detektoren genutzten Ports der entsprechenden Klasse als neue private-Attribute hinzugefügt (Refaktorisierung „Feld verschieben“[23] bzw. „Parameter entfernen“[24]).

Beispiel:

TRadicon verarbeitet die Initialisierungsdatei hardware.ini. Daraus ermittelt die Klasse, mit welchen I/O-Ports die Kommunikation mit der Radicon-Hardware stattfinden soll. Mit diesen Informationen wird ein Objekt der Klasse TRadiconHW erzeugt. Ab diesem Zeitpunkt sind die zu verwendenden I/O-Ports für TRadicon uninteressant, das sind Details der Hardwarekommunikation, um die sich die Hardwareklasse kümmern soll. Die beiden Ports sind jetzt Attribute der Klasse TRadiconHW, daher müssen sie nicht wie früher bei jeder Funktion als Parameter übergeben werden.

Zu guter letzt musste sich die Klasse einer semantischen Überprüfung unterziehen:

Wurden alle Methoden genutzt? Gab es Methoden, die zu viel bzw. zu wenig taten? Gab es Methoden, die lieber mit fremden Objekten arbeiteten als mit dem eigenen? Stellten Daten und Methoden zusammen genau ein Konzept dar?

[23] [FOW00], Seite 144

[24] [FOW00], Seite 283

In jedem Falle stand eine Umbenennung fast aller Bezeichner an, um ihnen Bedeutung einzuhauchen. Aus dem eher kryptischen begwrk(CF2) ( „begin work“ – eine Methode, die die Arbeit nicht nur beginnen, sondern auch beenden konnte ) wurde zum Beispiel die Methode

```
Execute(fixedImpulseCounts).
```

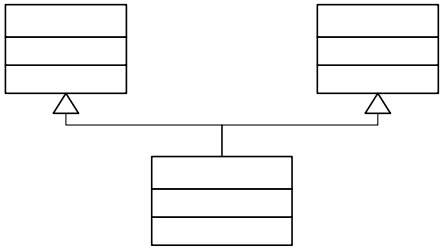
Da sich am Treiber fast alles geändert hatte, musste mit jedem Umstellungsschritt auch der nutzende Code modifiziert und an die neuen Verhältnisse angepasst werden. Zum Glück betraf das im Wesentlichen immer die Klasse TRadicon.

Anschließend war mir klar, dass das Konzept der Zusammenarbeit einer Hardwareklasse mit einer Messadapterklasse auch auf alle anderen Detektoren projiziert werden konnte. Hier bildete sich eine Basiskomponente der neuen Architektur heraus. Die Messadapterklasse ist per Komposition mit der Hardwareklasse verbunden und nutzt deren Dienste für die Kommunikation mit der Hardware, um ihrerseits die Anforderungen der nutzenden Subsysteme zu erfüllen und Messwerte bereitzustellen.

TRadicon im Ausgangszustand war ein schönes Beispiel von „C mit Klassen“. Die Mechanismen zur Hardwaresteuerung, die jetzt in TRadiconHW gefasst sind, lagen als C-Funktionen vor. Die Klasse TRadicon versuchte nun, *einen* Radicon-Detektor zu verkörpern. Die Hardwareadressen des einen unterstützten Controllers waren globale Variablen; allein die Erzeugung eines weiteren Objektes vom Typ TRadicon hätte ein heilloses Durcheinander bewirkt. Die Tatsache, dass eine Klasse nur eine Schablone für die Erstellung mehrerer Objekte ist, wurde hier völlig ignoriert. Das Konzept der Klasse wurde nur genutzt, um einige Funktionen unter dem Namensraum TRadicon zusammenzufassen.

### 4.8.2 Refaktorisierung des TAm9513-Detektortreibers

Diese neue Architektur konnte recht problemlos auch auf die Treiberkomponenten für die TAm9513-Multi-I/O-Karte realisiert werden. Ursprünglich wurden die zugehörigen Treiber aus zwei Komponenten gebildet:

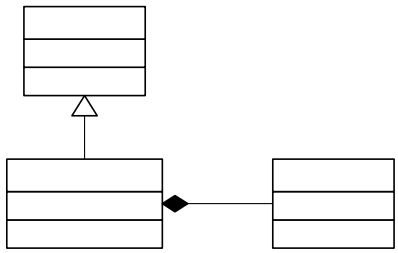


Die Klasse TAm9513 war für die eigentliche Hardwarekommunikation verantwortlich. Sie soll nun aber nicht mehr durch (Mehrfach-)Vererbung, sondern durch Komposition genutzt werden. Dadurch entsteht eine reine gerichtete Nutzungsbeziehung: TGenericDetector nutzt Funktionalität von TAm9513.

TAm9513 habe ich mit denselben Methoden, die schon für die Radicon-Treiber verwandt wurden, derart modifiziert, dass alle Hardwarezugriffe nur noch über die neuen TIOPort-Objekte erfolgen. Damit konnten bedingte Kompilierung und inline-Assembler völlig aus der Implementation dieser Klasse verschwinden.

In einem zweiten Schritt habe ich die Vererbungsbeziehung zwischen TAm9513 und TGenericDetector entfernt und durch eine Komposition ersetzt. Dazu erzeugt jedes TGenericDetector-Objekt jetzt bei der Initialisierung ein TAm9513-Objekt, das es für die Hardwarekommunikation nutzt.

Somit erreiche ich die Einführung einer Zweischichtenarchitektur, deren Schichten in den beiden Klassen TGenericDetector und TAm9513a implementiert sind.



Die beiden Hardwareklassen TRadiconHW und TAm9513 habe ich in die neuen Dateien detec_hw.cpp und detec_hw.h verschoben. Damit findet sich die unterste Softwareschicht der Detektoren-Schichtenarchitektur in eigenen Dateien.

## 4.9 Der Detektormanager

Der Detektormanager ist für das Detektoren-Subsystem von zentraler Bedeutung. Alle Subsysteme verschaffen sich Zugriff auf die gewünschten Detektoren, indem sie die Verwaltungsfunktionen des Detektormanagers aufrufen. Daher ist seine korrekte Funktion essentiell.

Zu Beginn meiner Arbeiten war der Detektormanager in der Klasse TDList implementiert. TDList hatte die Eigenart, dem System Beschränkungen aufzuerlegen, die nur durch die Art der Implementation existierten, also nicht der Anwendungsdomäne entstammten.

Die Detektoren wurden in einem Array von Zeigern auf Detektor-Objekte verwaltet:

```
aDevice = new TDevice* [nMaxNumber];
```

Die maximale Anzahl verwaltbarer Detektoren war im Code mit drei fest eingestellt. Diese Festlegung war rein willkürlich, weder Beschränkungen der Hardware noch Beschränkungen der Einrichtung der Messplätze liegen dieser Zahl zugrunde. Daher habe ich eine Lösung geschaffen, die genau so viele Detektoren verwaltet, wie es der Systemnutzer durch Einträge in der Initialisierungsdatei wünscht.

Die Speicherung der Detektorobjekte in einem Array habe ich durch eine dynamische Speicherverwaltung ersetzt. Meine erste Implementation nutzte dazu den Containertyp vector aus der C++-STL ( Standard Template Library ). Sie funktionierte hervorragend unter Borland C++ 5.0, leider jedoch nicht unter Borland C++ 4.5, da die Borland-C++-Compiler vor Version 5 die STL noch nicht kennen. Da Rückwärtskompatibilität zur Version 4.5 beibehalten werden musste, habe ich die Verwaltung der Detektorobjekte in einem zweiten Anlauf als verkettete Liste implementiert. Diese Verwaltungsstruktur ist als eingebettete private Klasse in TDetectorManager realisiert, damit sie später problemlos durch einen STL-Containertyp ersetzt werden kann. In Fowlers Katalog kennt man dafür die Refaktorisierung „Array durch Objekt ersetzen“[25].

Die Klasse TDList führte Attribute, die den Index von Detektoren eines bestimmten Typs im Detektoren-Array festhielten:

```
int         MD_Monitor;
int         MD_Encoder;
int         MD_Psd;
int         MD_Counter;
int         MD_Generic;
```

Über diese Indexwerte konnte der zum jeweiligen Detektortyp gehörende Detektor ermittelt werden. Daher konnte für jeden dieser Detektortypen auf genau einen Detektor zugegriffen werden. Waren mehrere Detektoren angeschlossen, die zu derselben Detektorenklasse gehörten, konnte immer nur der Detektor mit dem numerisch größten Index ( der zuletzt verarbeitete ) angesprochen werden. TDList konnte also maximal drei Detektoren verwalten, die verschiedenen Detektorklassen angehören mussten.

[25] [FOW00], Seite 186ff

Von welchem Typ ein Detektor ist, ist an sich eine Eigenschaft des Detektors. Es ist keine Rolle, die sich im Laufe des Lebens eines Detektorobjektes ändern könnte. Der Typ eines Detektors ist in der Initialisierungsdatei vorgegeben und während des Programmablaufes statisch. Daher habe ich TDetector ein neues Attribut und eine Zugriffsmethode darauf hinzugefügt, damit der Typ eines Detektors jederzeit ermittelt werden kann.

Im Detektormanager wurde die Methode GetDetector()dahingehend verändert, dass sie man ihr mit einem optionalen zweiten Attribut sagen kann, den wievielten Detektor des entsprechenden Typs sie zurückgeben soll.

```
TDetector* GetDetector( int n, int nFilterDimension = -1 ) const;
```

Diese Methode durchläuft den Detektorencontainer und ruft für jedes Detektorobjekt die Methode GetDetectorType() auf, bis der gewünschte Detektor gefunden ist. Somit kann die explizite Speicherung in Indexvariablen entfallen.

Inhaltlich gesehen haben hier eine ganze Reihe von Refaktorisierungen stattgefunden: „Feld verschieben“ von TDetectorManager nach TDetector, „Feld kapseln“ durch das Hinzufügen der entsprechenden Zugriffsmethode GetDetectorType(), „Temporäre Variable durch Abfrage ersetzen“, „Algorithmus ersetzen“ und „Methode parametrisieren“ durch das Überarbeiten von GetDetector().

In der Indexvariable MD_Monitor wurde der Index des Detektors festgehalten, der als Monitordetektor genutzt werden sollte. Der Monitordetektor wird bei einer Messung zusätzlich eingesetzt, um die Intensität des Röntgenlichtes festzuhalten. Mit diesen zusätzlichen Referenzdaten können die Messwerte entsprechend der tatsächlich verwendeten Beleuchtungsintensität normalisiert werden.

Die Tatsache, dass ein Detektor als Monitordetektor verwendet wird, ist jedoch eine Eigenschaft des entsprechenden Messvorgangs, keine Eigenschaft des Detektors. Der Detektor hat lediglich korrekte Messwerte zu liefern. Bis zum Programmende war der Detektor also nur als Monitor nutzbar, sein eigentlicher Typ wird vom System ignoriert. Um ihn als normalen Detektor zu nutzen, war eine Modifikation der hardware.ini und ein Programmneustart notwendig.

Die Lösung dieses Problems war insofern schwierig, als dass dazu Verantwortlichkeiten zwischen Klassen aus verschiedenen Subsystemen transferiert werden mussten. Nicht mehr der Detektormanager sollte wissen, welcher der Detektoren als Monitor genutzt werden sollte, sondern die Klasse, die die jeweilige Untersuchungsmethode repräsentiert. Sie sollte die Messergebnisse des Detektors in seiner Rolle als Monitordetektor interpretieren und anschließend als normalen Detektor wieder zur Verfügung stellen.

Zuerst habe ich also in die entsprechenden Scan-Klassen gerichtete Assoziationen integriert, mit der ein Scan-Objekt auf seinen Monitor-Detektor zugreifen kann. Anschließend habe ich in den Dialogressourcen der Scan-Konfigurationsdialoge die bisher vorhandene Schaltfläche entfernt, mit der die Nutzung des als Monitor deklarierten Detektors aktiviert bzw. deaktiviert werden konnte. Stattdessen habe ich Kombinationsfeld eingefügt, mit dem bei Bedarf ein geeigneter Detektor als Monitordetektor ausgewählt werden kann.

Wenn ein Detektor nun als Monitordetektor ausgewählt wird, zeigt die Monitor-Assoziation auf das gewählte Detektoren-Objekt; andernfalls auf 0. Damit hatte sich ein semantischer Wandel vollzogen. Bis dato war der Detektor, der als Monitor genutzt werden konnte, durch das ini-File festgelegt. Eine zusätzliche boolesche Variable zeigte an, ob der festgelegte Monitor auch tatsächlich genutzt werden sollte. Nun war die Wahl ( und der Wechsel ) des Monitordetektors während der Programmlaufzeit möglich. Ob ein Monitor genutzt werden sollte, läßt sich einfach durch den Test des entsprechenden Zeigers auf 0 feststellen.

Mit der neuen Flexibilität waren allerdings viele Änderungen am Client-Code nötig. Alle Stellen, wo bisher der Wert der booleschen Variable überprüft worden war, musste ich nun durch einen Vergleich des Monitor-Zeigers mit dem Nullzeiger ersetzen.

Aufwendiger war noch der Kampf mit dem schlechten Klassendesign der Scanklassen. Zu jeder Scan-Klasse existiert eine entsprechende ScanParameters-Klasse. Diese ScanParameters-Klasse ist vom Wesen her eine Oberflächenklasse, deren Aufgabe es ist, die interaktiven Eingaben in einem entsprechenden Oberflächendialog zu verarbeiten und zu speichern. Die Oberflächenklasse speichert die ermittelten Werte aber nicht, wie in einem vernünftigen objektorientierten Design ( z.B. Doc/View- bzw. Model/View/Controller-Pattern[26] ) üblich, in einer Daten-/Dokumentklasse, sondern direkt in sich selbst. Um der Scan-Klasse Zugriff auf diese Daten zu vermitteln, wurden nicht etwa Zugriffsmethoden eingesetzt, sondern die Datenklasse von der Oberflächenklasse abgeleitet. Für einige der aus den Basisklasse ererbten Attribute (u. a. den Monitorzeiger) wurden in den Scan-Klassen eigene lokale Kopien angelegt.

[26] [GOF96], Seite 5ff

Dieses völlig aus Sicht der Softwarequalität völlig inakzeptable Konstrukt wurde aber durch einen weiteren Aspekt noch verschlechtert: Die Funktionalität der drei LineScan-Verfahren StandardScan, Omega2ThetaScan und ContinuousScan, die man vernünftigerweise als Subklassen von TScan ( eine der im vorigen Abschnitt beschriebenen Scan-Klassen) implementiert hätte, ist in einer einzigen Klasse untergebracht. In den meisten Methoden wird dann durch gigantische switch-Anweisungen das zum jeweiligen Typ gehörende Verhalten ausgewählt. Das ist zwar C++, weil es von einem C++-Compiler akzeptiert wird, aber nicht objektorientiert. Vier verschiedene Konzepte und die dazu gehörende Oberfläche sind in zwei voneinander abgeleiteten Klassen integriert, von Oberflächen-Anwendungslogik-Trennung kann keine Rede sein. Diesem Gewirr mit den Methoden des Refactorings zu Leibe zu rücken sollte ausreichend Stoff für eine weitere Diplomarbeit bieten.

In dieser Situation habe ich mich für eine minimalinvasive Herangehensweise entschieden. Ich habe die ScanParameters-Klassen so modifiziert, dass sie die zugehörige Scan-Klasse veranlassen, den Zeiger auf das Monitordetektor-Objekt selbst zu speichern. Damit entfiel die lokale Kopie dieses Attributes, und es wurde klarer, welche Monitorattibute modifiziert werden mussten. Anschließend habe ich vor jedem Zugriff auf den Monitor-Zeiger einen Test integriert, ob dieser Zeiger auf einen Detektor zeigt. In späteren Versionen könnte dieser Test durch die Nutzung eines Nullzeiger-Objekt oder die Nutzung von C++-Exceptions erweitert werden.

Anschliessend konnte das Konzept des Monitordetektors vollständig aus dem Detektorensubsystem entfernt werden, weil weder die einzelnen Detektoren noch der Detektormanager wissen muss, in welcher Semantik die Messwerte eines Detektors verarbeitet werden.

Diese drei TDList-Methoden

```
HINSTANCE GetModuleHandle( void );
BOOL InitializeModule( void );
BOOL SaveModuleSettings( void );
```

sind kennzeichnend für ein weiteres Problem der ursprünglichen Implementation:

Die Implementation des Detektormanagers war zu eng mit der betriebssystemspezifischen Verwaltung der Bibliothek counters.dll verbunden.

Der Grund dafür war in der Art und Weise zu suchen, wie dafür gesorgt wurde, dass genau ein Detektormanager-Objekt existiert. Dazu wurden DLL-Initialisierungsmechanismen genutzt, die beim Laden der DLL einen Zeiger auf ein globales TDList-Objekt anlegten ( lpDList ). Alle anderen Subsysteme fanden den Detektormanager, indem sie diesen Zeiger nutzten. Er konnte überall im Code beliebig überschrieben werden.

Das Problem habe ich in zwei Stufen gelöst. Um sicherzustellen, dass von einer Klasse genau ein Objekt existiert, hat die „Gang of Four“[27] das Singleton-Pattern bekanntgemacht:

```
class TDetectorManager
{
public:
  TDetectorManager& DetectorManager()
  {
    // die Singleton-Instanz
    static TDetectorManager theDetectorManager;

    return theDetectorManager;
  }

private:
  //! Singleton-Konstruktor
  TDetectorManager();

  // Verbotene Operationen
  TDetectorManager(const TDetectorManager&);
  TDetectorManager& operator=(const TDetectorManager&);
}
```

Der Default-Konstruktor, der Copy-Konstruktor und der Zuweisungsoperator der Klasse werden verborgen. Die Klasse bekommt eine öffentliche statische Methode GetInstance(), die ein statisches Objekt vom Klassentyp anlegt und eine Referenz darauf zurückliefert. Damit erreicht man mehrere Ziele auf einmal: Diese Methode kann man von überall durch Klasse::GetInstance() aufrufen. Das statische Objekt in der statischen Methode wird beim ersten Aufruf der Methode angelegt. So kann man genau den Erstellungszeitpunkt festlegen, wenn zum Beispiel Vorbedingungen erfüllt sein müssen oder Abhängigkeiten von anderen Objekten bestehen. Bei jedem Aufruf der Methode wird dasselbe Objekt zurückgegeben, ein weiteres kann nicht erzeugt werden. Da eine Referenz immer auf dasselbe Objekt zeigt, war die bisher mögliche Zeigermanipulation nun nicht mehr möglich.

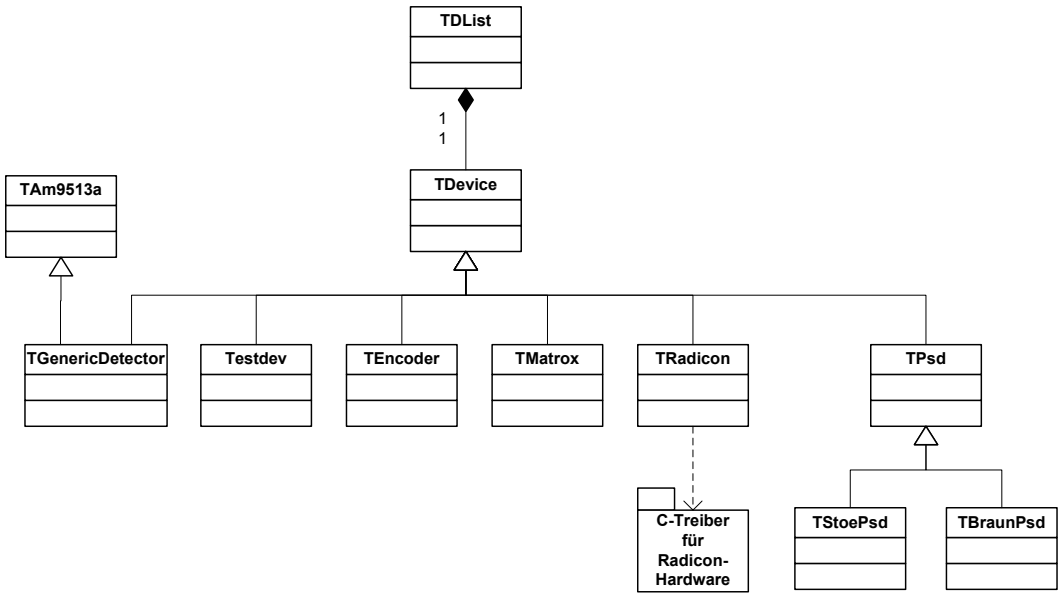
Nach der Integration des Singleton-Musters mussten 128 Referenzen auf lpDList durch die Nutzung von TDetectorManager::DetectorManager() ersetzt werden. Da es nur ein Symbol dieses Namens gab, konnten lexikalische Suchen-und-Ersetzen-Methoden hier angewandt werden.

[27] [GOF96], Seite 157ff

Zum Detektorenmanager sind noch weitere Methoden hinzugekommen, die in den Abschnitten über die Sanierung der GUI-Komponenten bzw. über das Interface-Redesign beschrieben werden.

## 4.10 Die Detektoren-Klassenhierarchie

Betrachtet man die Hierarchie der Klassen, die von TDevice (jetzt TDetector) abgeleitet sind, zum Zeitpunkt des Beginns meiner Arbeiten, so lassen sich verschiedene Designprobleme feststellen.




Der Klasse TDevice lastet der Geruch einer „großen Klasse“[28] an. Diese Klasse hat einfach zu viele Aufgaben. Sie ist gleichzeitig:

- Wurzelklasse aller Detektoren, jedoch nicht abstrakt
- Interface für alle null- und eindimensionalen Detektoren
- Messwertermittlungs- und Berechnungsklasse
- Container für Hardware-Einstellungen
- Container für Einstellungen, die den Messvorgang und die Messwertermittlung betreffen
- Einfacher nulldimensionaler Testzähler
- Verwalter für das Zählerfenster

Von funktionaler Bindung kann keine Rede sein.

[28] [FOW00], Seite 71

### 4.10.1 Neue Basisklassen

Von TDevice sind alle nulldimensionalen Detektoren direkt abgeleitet. Für die eindimensionalen Detektoren existiert die Basisklasse TPsd, die ebenfalls direkt von TDevice abgeleitet ist.

Die Klienten des Subsystems nutzen nur Objekte vom Typ TDevice*. Sie können somit nur Methoden aufrufen, die in TDevice bereits deklariert sind. Das Ergebnis ist das, was Stroustrup ein „fettes Interface“ nennt[29]: Für alle Methoden der Unterklassen sind Delegationen in der Basisklasse vorhanden. Infolgedessen lassen sich Methoden für Objekte aufrufen, die diese Funktionalität semantisch gar nicht unterstützen. Oft sind deshalb nur Methodenrümpfe implementiert, die lediglich generische Rückgabewerte wie 1 oder true liefern. Die Aufgabe, die der Name der Methode beschreibt, kann und soll das Objekt ja gar nicht erfüllen.

Beispiel:

Lediglich zwei der Detektoren können tatsächlich ein akustisches Feedback zur aktuellen Höhe des ermittelten Wertes realisieren: TRadicon und TGenericDetector, beides nulldimensionale Detektoren. Inhaltlich ist so eine akustische Rückkopplung auch nur für nulldimensionale Detektoren sinnvoll. Um eine solche Funktion bieten zu können, muss die Basisklasse sie für alle Detektoren zur Verfügung stellen, auch für die, bei denen es keinen Sinn macht.

Noch ein Beispiel:

Die Detektoren liefern naturgemäß unterschiedliche Messwerte: nulldimensionale Detektoren eben nulldimensionale Werte und eindimensionale Detektoren entsprechend eindimensionale. Auch wenn es eventuell noch sinnvoll ist, aus den Messwerten eines eindimensionalen Detektors per Integration einen nulldimensionalen Wert zu berechnen, ist es gänzlich unsinnig, wenn von einem nulldimensionalen Detektor ein eindimensionaler Messwert abgefragt werden kann. Beide Möglichkeiten werden jedoch für alle Detektoren von TDevice zur Verfügung gestellt.

Andererseits muss für jede neue Methode, die in einer der abgeleiteten Klassen definiert wird, eine Delegation in TDevice hinzugefügt werden, damit diese neue Methode über das fette Interface aufrufbar ist. Dadurch würde eine Neuerstellung aller Programmteile, die das Detektoren-Subsystem nutzen, notwendig werden – eine zusätzliche Erstellungsabhängigkeit.

Auch dieses Problemgebilde wurde wieder durch Zerlegung in Einzelprobleme gelöst.

[29] Vgl. dazu [STRO98], Seite 467ff und Seite 819ff

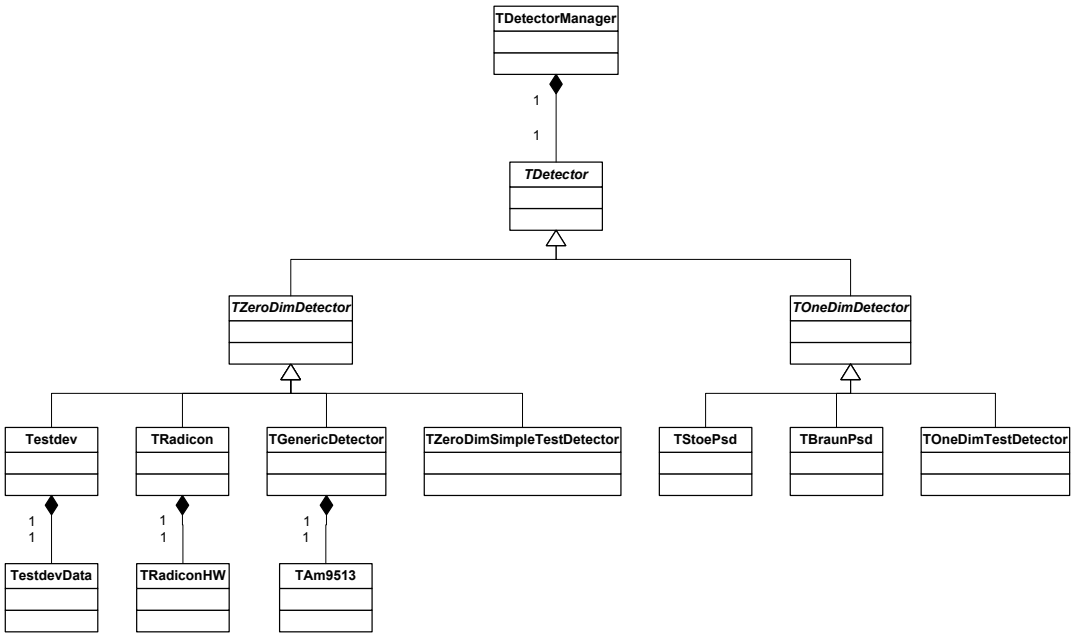
Äquivalent zur Klasse TPsd habe ich für alle nulldimensionalen Detektoren die Klasse TZeroDimDetector geschaffen, die direkt public von TDevice abgeleitet ist. Dementsprechend habe ich TPsd in TOneDimDetector umbenannt. Damit ein solches Konstrukt nicht nur eine zusätzliche Abstraktionsebene darstellt, musste TZeroDimDetector Verantwortung übernehmen. Der Mechanismus der akustischen Wiedergabe der gemessenen Intensität ist wie oben erläutert nur für nulldimensionale Detektoren sinnvoll. Also habe ich die Refaktorisierungen „Methode verschieben“ und „Feld verschieben“ angewandt, um alle zur Kontrolle der akustischen Rückkopplung gehörenden Elemente in die neue Klasse zu verschieben.

Damit tritt automatisch ein neues, wesentlich größeres Problem auf: Das Konzept des fetten Interfaces wurde verletzt. Für Objekte vom Typ TDevice* konnte nun SetSound() nicht mehr aufgerufen werden, weil diese Methode nicht mehr für alle Detektoren existierte.

Damit wurde eine massive Umstrukturierung des Subsysteminterface und der nutzenden Subsysteme ausgelöst. Ein Fassadenmuster für alle von TDevice abgeleiteten Klassen war schon deshalb nicht sinnvoll, weil diese abgeleiteten Klassen ihrem Wesen nach vollkommen unterschiedliche Methoden anbieten mussten. In Klassenstrukturen, in denen alle abgeleiteten Klassen mit dem Interface der Basisklasse auskommen und nur verschiedenes Verhalten zeigen, ist so ein Design einsetzbar.

Die Struktur der von TDevice abgeleiteten Klassen war aber nicht derart homogen. Die nulldimensionalen Detektoren müssen teilweise völlig andere Methoden unterstützen als die eindimensionalen. Die von TZeroDimDetector bzw. TOneDimDetector abgeleiteten Messungsadapterklassen hingegen erfüllen das Interface ihrer Basisklasse.

Daher sieht die neue Klassenstruktur wie folgt aus:




Die drei Klassen TDevice, TZeroDimDetector und TOneDimDetector bilden nun das neue Subsysteminterface des Detektoren-Subsystems. Für jeden Subsystemklienten hat das weit reichende Folgen: Dem Nutzer muss klar sein, ob er einen generischen Detektor mit einem eingeschränkten Satz an allgemeinen Methoden oder einen in seiner Dimension festgelegen Detektor mit spezifischen Methoden nutzen möchte.

Spätestens zum Zeitpunkt der Verschiebung der Sound-Kontrolle in die Klasse TZeroDimDetector ließ sich das Projekt nicht mehr fehlerfrei übersetzen. Daher musste ich für jede Klasse, die bisher mit TDevice*-Objekten gearbeitet hat, überprüfen, mit welcher Art von Detektoren sie nun arbeiten würde. Diese Überprüfung konnte drei mögliche Ergebnisse haben:

- Die Klasse nutzt nur die Methoden eines generischen Detektors, in diesen Fällen konnte mit TDevice weitergearbeitet werden.
- Die Klasse nutzt entweder null- oder eindimensionale Detektoren, dann konnte entsprechend mit TZeroDimDetector*- oder mit TOneDimDetector*-Objekten gearbeitet werden.
- Die Klasse nutzt beide Detektortypen. Jetzt war die Lage bedeutend schwieriger: ein RTTI-Mechanismus (run-time type information) musste genutzt werden, um entsprechend des dynamischen Typs eines TDevice*-Objektes einen downcast entweder zu TZeroDimDetector oder zu TOneDimDetector zu machen.

Da Borland C++ 4.52 Exceptions noch nicht vernünftig unterstützt, entstanden relativ umständliche, nicht portable dynamic_cast-Konstrukte. Borland C++ 4.52 (und Borland C++ 5 mit deaktivierter Exception-Unterstützung) liefert dann, wenn der dynamic_cast fehlschlägt, einen Nullpointer zurück. Andere Compiler würden hier einfach eine bad_cast-Exception zurückgeben, die nicht verarbeitet wird und das Programm damit zum Absturz bringt.

Da die in den beiden Compilern integrierte RTTI-Unterstützung proprietär und selbst zwischen den beiden Compilern nicht portabel war, habe ich einen eigenen Mechanismus implementiert, um vor einem dynamic_cast sicherstellen zu können, dass diese Typumwandlung funktionieren würde. Mit anderen Worten, das Problem der unterschiedlichen Reaktion auf einen fehlgeschlagenen dynamic_cast ist in keiner Weise beseitigt, es wird nur sichergestellt, dass ein solcher Fall nicht eintritt.

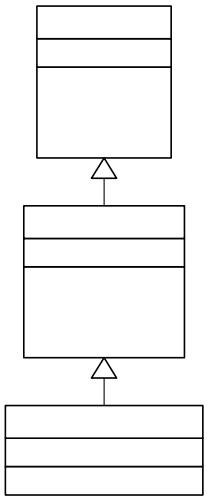
Jedes TDetector-Objekt weiß, in welcher Dimension es Messwerte ermittelt. Mit der Methode TDetector::GetDetectorDimension() kann vor einem dynamic_cast abgefragt werden, ob es sich um einen null- oder eindimensionalen Detektor handelt und dementsprechend den richtigen downcast wählen. Die Nutzung von Typinformationen in Attributen entspricht zwar nicht den Grundsätzen der OO-Programmierung und wird als schlechter Stil angesehen (sogar als Codegeruch), stellt aber eine Notwendigkeit dar, wenn man mit den relativ veralteten Borland-Compilern portablen Code erstellen möchte. Nach einer Portierung auf eine modernere Entwicklungsumgebung muss dieses Detail also noch einmal überarbeitet werden.

### 4.10.2 Testdetektoren

Im XCTL-System existieren drei Testdetektoren[30]. Das ist Code, der einen realen Zähler simuliert, jedoch keinerlei Hardware nutzt. Ursprünglich war der Testdetektor „Simulant“ direkt in der Klasse TDevice und der Testdetektor „PSD“ in der Klasse TPsd integriert. Diese beiden Klassen sollen als Basisklassen und als Interface fungieren, sie sollten also abstrakt sein. Um mit den integrierten Testdetektoren arbeiten zu können, muss man aber Objekte dieser Klassen anlegen können.

Daher habe ich die Refaktorisierung „Unterklasse extrahieren“ genutzt, um die Testdetektorfunktionalität aus den Basisklassen heraus in je eine neue Klasse zu transferieren.

[30] siehe Abschnitt 1.3



Anschließend brauchte nur der Detektorobjekt-Erzeugungsmechanismus in TDetectorManager angepasst werden, so dass für die entsprechenden Testdetektoren die jeweilige neue Klasse genutzt wurde.

Diese Refaktorisierung betraf nur das Detektoren-Subsystem, andere Projektteile waren nicht betroffen. Sie erhöhte zwar die Anzahl der Klassen und die Breite der Klassenhierarchie, verringerte aber wesentlich die Komplexität der betroffenen Basisklassen. Die Funktionalität der Testdetektoren konnte in zwei einzelnen semantisch gebundenen Klassen von geringer Komplexität abgelegt werden. Insbesondere wurde mit der Deklaration der Basisklassen als abstrakte Klassen die Schaffung echter Interface-Klassen möglich.

### 4.10.3 Kommunikation

Im XCTL-Programm werden verschiedene Konzepte zur Nachrichtenübermittlung angewandt. Dazu kennt man im Software-Engineering eine ganze Reihe von Mechanismen. In erster Linie muss hier synchrone und asynchrone Kommunikation unterschieden werden.

Synchrone Nachrichtenmechanismen blockieren den Sender einer Nachricht, bis der Empfänger die Nachricht verarbeitet hat. Ein Methodenaufruf an einem C++-Objekt stellt zum Beispiel synchrone Kommunikation dar. Der Kontrollfluss geht zum aufgerufenen Objekt über und kehrt erst zurück, wenn die aufgerufene Methode abgearbeitet ist. Das Ergebnis ist also eine streng sequentielle Arbeitsweise.

Bei asynchroner Kommunikation hingegen wird dem Empfängerobjekt eine Nachricht gesandt, die es sofort oder später bearbeiten kann. Dadurch wird die Arbeit des Senders von der des Empfängers entkoppelt. Mehrere Kontrollflüsse können parallel und auch verteilt existieren. Der Empfänger hat keinen direkten Einfluss auf den Sender. Diese Unabhängigkeit wird mit komplizierteren Mechanismen zum Message-Queuing und zur Prozesssynchronisation erkauft.

In Windows-Systemen wird asynchrone Kommunikation u. a. über Betriebssystem-Schnittstellen ermöglicht. So genannte Windows-Messages können von einem Fenster an ein anderes Fenster gesandt werden. Damit CASE-Werkzeuge diese von der Implementationssprache unabhängige Objektkommunikation berücksichtigen können, müssen sie zusätzliches Wissen über die Besonderheiten des jeweiligen Betriebssystems haben. Insbesondere Windows-Programme, die nicht unter Nutzung der MFC (Microsoft Foundation Classes) bzw. deren Nachfolgern entwickelt wurden, stellen für CASE-Tools in der Hinsicht Probleme dar. Andererseits fesselt die Nutzung eines solchen proprietären Nachrichtenmechanismus an das Betriebsystem und in diesem Fall auch an das GUI-System.

Wann synchrone und wann asynchrone Kommunikation eingesetzt wurde, scheint im XCTL-Programm in erster Linie vom Kenntnisstand und von persönlichen Neigungen des jeweiligen Entwicklers abhängig zu sein. Insbesondere bei der Nutzung von Timern wurden manchmal Fenster für Vorgänge genutzt, die mit Fenstern überhaupt nichts zu tun haben, nur weil die im Windows-API eingebauten Timer Nachrichten nur direkt an Fenster senden.

Andererseits ist für die Nutzung mehrerer Kontrollflüsse asynchrone Kommunikation unbedingt erforderlich. Hier sollte ein weiteres Projektteam vielleicht einmal über alternative Lösungen nachdenken und eine Lösung für das gesamte XCTL-Projekt schaffen.

### 4.10.4 Synchronisation

Im gesamten XCTL-Programm stellen Fenster Werte aus Dokumenten dar: Messwerte, Motorpositionen, Graphen, Konfigurationseinstellungen etc. In vielen Fällen ist der zu präsentierende Wert im Zeitverlauf nicht konstant. Die Präsentation muss also in irgendeiner Form mit der zeitlichen Änderung der dargestellten Werte synchronisiert werden. Dazu existieren im Wesentlichen zwei Mechanismen:

- Pull-Synchronisation
  Die Präsentation muss in regelmäßigen Intervallen den aktuellen Wert der dargestellten Größe neu ermitteln und bei Bedarf neu darstellen. Dazu braucht nur die Präsentation eine Assoziation zum beobachten Objekt und einen Synchronisations-Zeitgeber. Der beobachtete Wert kann sich inzwischen geändert haben, trotzdem wird mit dem veralteten Wert weitergearbeitet, bis die nächste Synchronisation stattgefunden hat.

- Push-Synchronisation
  Das beobachtete Objekt benachrichtigt bei jeder Zustandsänderung alle seine Präsentationen (Publisher-Subscriber-Muster[31]). Dazu müssen sich alle Präsentationen beim beobachteten Objekt registrieren und später wieder abmelden.

Natürlich kann keiner der beiden Formen generell der Vorzug gegeben werden. Pull-Synchronisation ist bei lokalen, einfachen Zusammenhängen deutlich einfacher zu realisieren. Push-Synchronisation ist ressourcenschonender, weil sie nur dann abläuft, wenn tatsächlich eine Veränderung aufgetreten ist. Sie funktioniert auch mit asynchroner Kommunikation in nebenläufigen und verteilten Systemen.

Im Zuge einer evolutionären Entwicklung wird Pull-Synchronisation mit dem Wachsen des Projektes zu irgendeinem Zeitpunkt in Push-Synchronisation geändert werden müssen.

Im Detektoren-Subsystem werden beide Verfahren eingesetzt: das Zählerfenster wird per Push-Synchronisation benachrichtigt, die Scan-Fenster per Pull-Synchronisation. In keinem der Fälle gibt es einen einheitlichen Standard, weil insbesondere das Problem der Fenster-Basisklassen noch nicht geklärt ist.

Problematisch für das XCTL-Projekt ist dabei die Art der Realisierung einer solchen Push-Synchronisation. Hier spielt das Publisher-Subscriber-Muster eine große Rolle: Der Publisher muss in einer geeigneten Weise einen Container aller Subscriber-Objekte führen, denen er mit einem gemeinsamen Mechanismus die Änderung mitteilt (ob er damit nur den Fakt der Änderung oder gleich den neuen Wert mitteilt, ist für die Problematik unerheblich).

Mit den Mitteln der Objektorientierung müssten also alle Subscriber-Objekte eine gemeinsame Schnittstelle implementieren, mit deren Methoden sie gegebenenfalls über die Änderung benachrichtigt werden können. Sind die (relevanten) Subscriber nicht von einer gemeinsamen Basisklasse abgeleitet, bleibt in C++ nur das Mittel der (zu vermeidenden) Mehrfachvererbung, weil C++ im Gegensatz zu Java kein reines Interface-Konzept kennt.

Nutzt man Nachrichtenmechanismen, die nicht Teil von C++ sind, so ist das gemeinsame Interface nicht erforderlich. Zum Beispiel könnte man das Problem in der Windows-Welt mit Windows-Botschaften lösen, die allerdings nur an Fenster gesandt werden können. Andererseits hat auch dieses Verfahren seine Beschränkungen und Nachteile.

[31] Für weitere Informationen siehe [GOF96], Seite 287, dort als Beobachter(Observer)-Pattern dargestellt

Die Implementation des Publisher-Subscriber-Musters ist essenziell für die Einführung des Doc-View- bzw. eventuell auch des Model-View-Controller-Musters. Beide Muster versuchen eine strikte Trennung zwischen Logik und Präsentation zu erreichen. Insbesondere im Hinblick auf eine eventuell bevorstehende Adaption der Software auf Microsoft Visual C++ und die MFC ist das Doc-View-Muster vorzuziehen, weil es das einfachere der beiden und das native Modell von Visual C++[32] ist.

Ein Dokument (Doc) mit seinen Werten kann dabei von einer beliebigen Anzahl von Ansichten (Views) dargestellt werden. Das Dokument bietet Schnittstellen, mit denen sein Zustand (die Attributwerte) verändert werden kann. Es kennt alle seine beobachtenden Ansichten und kann ihnen mitteilen, dass sich sein Zustand geändert hat.

### 4.10.5 Das Zählerfenster

In TDetector gibt es die Konzepte des aktuellen und des geöffneten Detektors. Diese Konzepte haben mit der alten Architektur des XCTL-Systems zu tun und haben in erster Linie historische Berechtigung. Ein Detektor ist dann geöffnet, wenn für ihn das Zählerfenster (TCounterWindow) angezeigt wird. Das Zählerfenster ist ein Fenster, das den derzeit mit dem aktuellen Detektor ermittelten Messwert als Zahl anzeigt. Hier liegen also schlecht synchronisierte bidirektionale 1:1-Assoziationen vor, die konzeptionell nicht so recht zum Rest des Systems passen.

Es kann also genau ein offenes Zählerfenster geben, und es kann zu einem Zeitpunkt mit genau einem Detektor gemessen werden. Das sind sicher Beschränkungen, die aus den frühesten Systemmodellen erhalten geblieben sind.

Inhaltlich ist das Zählerfenster ein Ansichtsfenster (View-Komponente), das die Werte eines Dokuments repräsentiert. Es gibt tatsächlich keinen Grund, warum nicht mehrere Zählerfenster zum selben Detektor (möglicherweise mit unterschiedlichen Darstellungen) geöffnet sein sollten. Es gibt auch keinen Grund, warum nicht parallel dazu weitere Zählerfenster den Messwert eines anderen Detektors darstellen können sollten.

Durch eine konsequente Implementation des Doc-View-Musters sollte sich hier eine wesentliche Verbesserung erreichen lassen.

[32] Zur Kurzdarstellung des Doc-View-Musters und seiner Rolle innerhalb der MFC: Vgl. Abschnitt „Die Dokument-/Ansicht-Architektur“ in [SPHA00], Seite 284

In seinem Wesen als darstellendes Fenster gehört das Zählerfenster nicht zum Detektoren-Subsystem. Eine Abhängigkeit von ihm ist für das Subsystem unerwünscht. Leider lässt es sich auch keinem der anderen Subsysteme zuordnen.

### 4.10.6 Einstellungen für die Belichtung

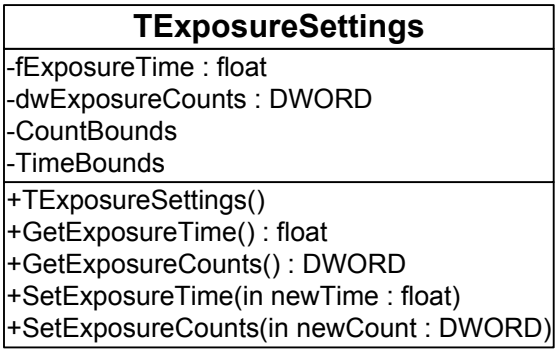
Die beiden Werte für die maximale Belichtungsdauer und die maximale Anzahl der dabei zu messenden Photonen stellen für alle Detektoren Limits für den Messvorgang dar. Dabei dürfen beide Werte in je einem detektorspezifischen Intervall liegen.

Die Einhaltung dieser Limits wurde für jeden Detektor an zwei Stellen durchgesetzt: bei der Verarbeitung des entsprechenden Wertes aus der Initialisierungsdatei und innerhalb der zugehörigen Oberflächenklasse.

Hier waren drei Probleme zu lösen:

- Jeder Detektor besitzt spezifische Limits für die Belichtungswerte.
- Die Einhaltung der Bereichsgrenzen muss bei jeder Modifikation der Werte überprüft werden.
- Beider Werte werden üblicherweise zusammen übergeben und genutzt. Hier riecht man den Codegeruch „Datenklumpen“.

Alle drei Probleme wurden gelöst, indem man die Belichtungsdaten in einer eigenen Klasse kapselte. Bei der Erstellung eines Detektor-Objektes wird ein zugehöriges TExposureSettings-Objekt erzeugt. Im TExposureSettings-Konstruktor gibt der Detektor seine spezifischen Bereichsgrenzen für die Belichtungswerte an.




Die Belichtungswerte können nun nur noch über Mutatormetoden modifiziert werden, die vor dem Verändern einer Belichtungseinstellung die Einhaltung der Bereichsgrenzen sicherstellt.

Bei der Übergabe der Belichtungswerte in einer Methode muss nun nur noch eine Objektreferenz angegeben werden. Inhaltlich gesehen ist diese Refaktorisierung eine Umstellung von strukturiertem C-Code auf objektorientiertes C++.

## 4.11 Das Subsystem-Interface

Ich habe das alte Interface aus C-Funktionen entfernt und vollständig durch das neue objektorientierte Interface ersetzt, was oftmals Modifikationen außerhalb des Detektoren-Subsystems verlangte.

Für das Detektoren-Subsystems habe ich die Trennung von Interface und Implementation vorangetrieben. Das neue Subsysteminterface kann man aus dem Inhalt der Datei „detecuse.h“ erkennen. Nur dieser Header muss integriert werden, damit das komplette Interface genutzt werden kann. Erst in der Implementation der Interface-Funktionen, die von außen nicht sichtbar sein muss, werden weitere Headerdateien der Subkomponenten des Detektoren-Subsystems verwandt. Daher ist es jetzt erstmals möglich, die counters.dll im Binärcode zu verwenden und weiterzugeben. Der Nutzer braucht nur noch die Headerdatei, in der das Interface deklariert ist.

Das Interface des Detektoren-Subsystems besteht nun aus drei Komponenten: den für eine Windows-DLL nötigen Verwaltungsfunktionen, den drei abstrakten Basisklassen TDetector, TZeroDimDetector, TOneDimDetector und der Utility-Klasse TDetectorGUI, mit der sich von außen die Konfigurationsdialoge ausführen lassen.

# 5 Das neue Detektoren-Subsystem

Das Detektoren-Subsystem ist eine Service-Komponente, die von den Programmnutzern nur indirekt genutzt wird, indem zur Durchführung eines Scans die entsprechenden Detektoren zur Nutzung bereitstehen. Der Nutzer kommt nur an zwei Stellen mit dem Subsystem direkt in Kontakt, und zwar in Form der Detektor-Deklarationen in der Initialisierungsdatei hardware.ini und in Form der Oberflächenelemente, mit denen Einstellungen an den Detektoren vorgenommen werden können.

Daher stellt das folgende Kapitel in erster Linie eine Informationsquelle für Entwickler dar. Dazu sollen für die Teile des Detektoren-Subsystems eine Verhaltens- und eine Designbeschreibung zu finden sein.

Das folgende Kapitel soll das Detektoren-Subsystem in erster Linie inhaltlich beschreiben. Ein viele Seiten lange Referenz mit einer Beschreibung aller Strukturen, Klassen, Methoden, Attribute, Aufzählungen etc. halte ich als Teil dieser Diplomarbeit für unangebracht.

Eine Dokumentation in Schriftform über Ist-Zustände ist meiner Meinung nur für einen Augenblick von Wert. Sie stellt nur eine Zeitpunktaufnahme dar, die nach der nächsten Codeänderung schon nicht mehr der Realität entspricht. Im Laufe der Zeit wächst die Diskrepanz zwischen Dokumentation und tatsächlich vorhandenem Code immer weiter.

Den Aufwand, nach jeder Änderung die Ist-Zustands-Dokumentation manuell abzugleichen, halte ich für unangemessen hoch. Solche Dokumentation lässt sich hervorragend automatisch generieren. Als einen solchen Dokumentationsgenerator habe ich während meiner Arbeit doxygen verwandt, das mir innerhalb weniger Minuten eine vollständige, aktuelle Ist-Zustands-Dokumentation erzeugt.

Entwicklern steht damit zusammen mit der überarbeiteten Code-Kommentierung und der von doxygen automatisch erstellten Projektdokumentation eine neue Informationsquelle aus erster Hand zur Verfügung. Damit soll der vor Beginn meiner Arbeit herrschende Mangel an Dokumentation über das Detektoren-Subsystem beseitigt sein.

## 5.1 Die Detektoren-Konfigurationsdatei hardware.ini

Um dem XCTL-Programm mitzuteilen, welche Hardware für Untersuchungen verwendet werden soll, existiert die Datei hardware.ini. Für jeden Detektor, der genutzt werden soll, muss in dieser Datei ein Abschnitt mit Konfigurationsdaten existieren. Am Programmende werden die Detektoreinstellungen wieder in die Initialisierungsdatei zurück geschrieben, um Einstellungen über mehrere Messsitzungen hinaus persistent zu halten.

Verschiedene Messplätze haben also, je nachdem welche Detektoren angeschlossen sind und welche davon genutzt werden sollen, unterschiedliche Initialisierungsdateien. Benennt man die Datei um bzw. verschiebt sie in einen anderen Pfad, können mehrere Versionen einer Systemkonfiguration gespeichert werden. Die Konfiguration eines Messplatzes kann vorab vorbereitet und dann einfach durch Einspielen der richtigen Initialisierungsdatei umgesetzt werden.

Die Datei ist im klassischen Stil einer Windows-Initialisierungsdatei gehalten:

Jeder Abschnitt wird durch eine Zeile eingeleitet, in der der Name des Abschnittes in eckigen Klammern steht.

Beispiel:

[device0]

In den folgenden Zeilen können dann Schlüssel/Wert-Paare folgen, bis das Dateiende erreicht ist oder ein neuer Abschnitt begonnen hat.

Beispiel:

Schluessel=Wert

Die Abschnitte für die am Messplatzrechner angeschlossenen Detektoren tragen die Bezeichnung [deviceX], wobei X eine nichtnegative ganze Zahl ist. Das XCTL-Programm sucht nach diesen Detektorabschnitten, indem eine Variable bei null beginnend jeweils um eins hochgezählt wird. Bei ersten Wert, zu dem kein entsprechender Detektorabschnitt gefunden wird, wird davon ausgegangen, dass nun alle zu nutzenden Detektoren erfasst wurden; weitere Detektorabschnitte werden nicht berücksichtigt. Fortlaufende Nummerierung aller [deviceX]-Abschnitte ist also zwingend erforderlich.

Basierend auf dem Wert des Schlüssels „Type=“ wird ein entsprechender Detektor vom Programm verwandt. Die Werte in einem Detektorabschnitt sind für unterschiedliche Abstraktionen eines Detektors bestimmt. Es gibt allgemeine Einstellungen, die für jeden Detektor möglich bzw. notwendig sind, es gibt Einstellungen, die nur für nulldimensionale bzw. nur für eindimensionale Detektoren gelten und es gibt Einstellungen, die nur zu einem spezifischen Detektor gehören.

Jede dieser Abstraktionsstufen sucht eigenständig nach entsprechenden Schlüssel/Wert-Paaren. Dabei werden nur solche Paare verarbeitet, die die jeweilige Abstraktionsstufe kennt und deshalb danach sucht. Andere Schlüssel/Wert-Paare werden schlicht ignoriert, solange sie den Syntaxregeln für Windows-Initialisierungsdateien genügen.

Jede Abstraktionsstufe besitzt die Methoden LoadDetectorSettings() und SaveDetectorSettings(), die die entsprechenden Einstellungen lesen bzw. speichern.

Einige Schlüssel/Wert-Paare müssen vorhanden sein, zum Beispiel das Paar „Type=“, sonst kann das Programm den Detektor nicht erkennen. Einige Werte können vorhanden sein, sie müssen es aber nicht. Fehlen sie, werden an ihrer Stelle vom Programm Standardwerte genutzt. Für den beim Programmende aktuellen Wert wird dann ein neues Schlüssel/Wert-Paar in die Initialisierungsdatei eingefügt. Man kann also mit einer Initialisierungsdatei beginnen, die nur die Pflichtwerte enthält, und daraus wird dann eine gültige Konfiguration mit allen optionalen Werten erstellt.

### 5.1.1 Einstellungen für alle Detektoren

**Schlüssel: Type** (Pflichtfeld)

| Typ des Wertes | Limits | Standardwert | Beispiel | Bemerkungen |
|---|---|---|---|---|
| String | | „“ | Type=Radicon | |

Anhand des Wertes wird der entsprechende Detektor verwendet. Welche Werte möglich sind, hängt von der jeweiligen Programmversion und den darin berücksichtigten Detektoren ab. Zurzeit sind das: „GENERIC“, „STOE-PSD“, „RADICON“, „BRAUN-PSD“, „PSD“ und „TEST“.

**Schlüssel: Name** (Pflichtfeld)

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| String | | „Counter“ |

Dieser Name erscheint in den Detektorauswahl-Kombinationsfeldern. Er ist also eher informellen Charakters.

**Schlüssel: Debug**

| Typ des Wertes | Limits | Standardwert | Bemerkungen |
|---|---|---|---|
| Integer | (0, 1) | 0 | 0 = aus<br>1 = ein |

Dieser Wert bestimmt, ob für diesen Detektor in erhöhtem Masse Statusinformationen erzeugt werden sollen.

**Schlüssel: ExposureTime**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Realzahl | [0.1, 500] | 4 |

Dieser Wert legt die maximale Länge eines Messintervalls, die Belichtungsdauer, fest.

**Schlüssel: ExposureCounts**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [10, 300000] | 10000 |

Dieser Wert legt die maximale Anzahl von Impulsen fest, die in einem Messintervall gemessen werden darf.

### 5.1.2 Einstellungen für alle nulldimensionalen Detektoren

**Schlüssel: Sound**

| Typ des Wertes | Limits | Standardwert | Bemerkungen |
|---|---|---|---|
| Integer | (0, 1) | 1 | 0 = aus<br>1 = ein |

Dieser Wert bestimmt, ob für diesen Detektor die Höhe der ermittelten Impulsrate auch akustisch repräsentiert werden soll.

### 5.1.3 Einstellungen für alle eindimensionalen Detektoren

**Schlüssel: BaseAddr**

| Typ des Wertes | Limits | Standardwert | Bemerkungen |
|---|---|---|---|
| Integer | | 0x300 | Der Wert sollte in hexadezimaler Schreibweise ( Beispiel 0x300 ) angegeben werden. |

Dieser Wert legt den ersten der zu verwendenden I/O-Ports für die Kommunikation mit der Detektor-Hardware fest.

**Schlüssel: OverflowIntensity**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Realzahl | | 50000.0 |

Dieser Wert legt fest, bei welcher ermittelten Intensität davon ausgegangen werden muss, dass ein Überlauf stattgefunden hat.

**Schlüssel: SignalGrowUp**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | (0, 1) | 1 |

Dieser Wert legt fest, ob der Messwert durch Aufsummieren mehrerer Teilmessungen ermittelt werden soll.

**Schlüssel: HVRegelung**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | (0, 1) | 0 |

Dieser Wert legt fest, on die Hochspannung des Detektors vom Programm geregelt werden soll.

**Schlüssel: ReadLeftFirst**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | (0, 1) | 1 |

Dieser Wert legt fest, ob die Daten von links nach rechts oder umgekehrt ausgelesen werden.

**Schlüssel: AngleStep**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Realzahl | | 1.0 |

Dieser Wert legt fest, welcher Winkelabstand von einem Kanal erfasst werden kann.

**Schlüssel: Unit**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| String | | Sekunden |

Einheit, in der Winkelangaben interpretiert werden sollen.

**Schlüssel: AddedChannels**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [1, 4095] | 4 |

Legt fest, wie viele Kanäle des PSDs zu einer Kanalgruppe zusammengefasst werden sollen, für die ein gemeinsamer Messwert ermittelt wird.

**Schlüssel: FirstChannel**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 4095] | 0 |

Erster Kanal des PSDs, mit dem gemessen werden soll..

**Schlüssel: LastChannel**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 4095] | 4095 |

Letzter Kanal des PSDs, mit dem gemessen werden soll..

### 5.1.4 Einstellungen für einen Detektor vom Typ „Radicon SCSCS“

**Schlüssel: IOAddr**

| Typ des Wertes | Limits | Standardwert | Bemerkungen |
|---|---|---|---|
| Integer | | 0x100 | Der Wert sollte in hexadezimaler Schreibweise ( Beispiel 0x100 ) angegeben werden. |

Dieser Wert legt den I/O-Ports für den Datenkanal der Kommunikation mit der Detektor-Hardware fest. Für den Steuerungskanal wird dann der nächsthöhere I/O-Port verwandt.

**Schlüssel: LowerThresh**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 1024] | 150 |

Der Wert legt den unteren Schwellwert für den Digital-Analog-Konverter(DAC) fest.

**Schlüssel: UpperThresh**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 1024] | 950 |

Der Wert legt den oberen Schwellwert für den Digital-Analog-Konverter(DAC) fest.

**Schlüssel: HighVoltage**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [400, 900] | 640 |

Dieser Wert legt die Höhe der Beschleunigungsspannung fest.

### 5.1.5 Einstellungen für einen Detektor vom Typ „Braun PSD“

**Schlüssel: EnergyScale**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 3] | 2 |

Diese Einstellung legt fest, um welche Zweierpotenz Energiedaten skaliert werden sollen.

**Schlüssel: PositionScale**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 3] | 2 |

Diese Einstellung legt fest, um welche Zweierpotenz Positionsdaten skaliert werden sollen.

**Schlüssel: AbbruchMitShutter**

| Typ des Wertes | Limits | Standardwert | Bemerkungen |
|---|---|---|---|
| Integer | (0, 1) | 0 | 0 = aus<br>1 = ein |

Dieser Wert legt fest, ob die Messung mit dem PSD durch Schließen des Shutters beendet werden soll.

**Schlüssel: EnergyLow**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 4095/EnergyScale ] | 526 |

Untere Schranke für das zu ermittelnde Energiespektrum.

**Schlüssel: EnergyHigh**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 4095/EnergyScale ] | 870 |

Obere Schranke für das zu ermittelnde Energiespektrum.

**Schlüssel: Ratemeter**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | (0, 1) | 0 |

Dieser Wert legt fest, ob das externe Ratemeter genutzt werden soll.

**Schlüssel: DeathTime**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | [0, 99] | 10 |

Der Wert legt den minimalen Abstand zwischen zwei Impulsen fest. Die Zeit ist in Mikrosekunden angegeben.

**Schlüssel: DelayFast**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | | 2 |

Dieser Wert beschreibt, wie lange das Programm die Hardwarekommunikation verzögert, um der Schnittstellenkarte Zeit zum Rechnen zu lassen. Die Schnittstellenkarte wird im Polling-Modus abgefragt und würde bei zu geringer Verzögerung überlastet.

**Schlüssel: DelaySlow**

| Typ des Wertes | Limits | Standardwert |
|---|---|---|
| Integer | | 4 |

Siehe DelayFast.

## 5.2 Die Komponente TDetectorManager

### 5.2.1 Verantwortung der Klasse

Der Detektoren-Manager verwaltet alle Detektor-Objekte; er erzeugt sie und zerstört sie am Programmende. Er ist Teil des Interfaces des Detektoren-Subsystems. Klienten des Subsystems nutzen den Detektoren-Manager, um Zugriff auf die nutzbaren Detektoren zu erhalten.

### 5.2.2 Lebenslauf des Detektoren-Managers

Um Probleme mit konkurrierenden Hardwarezugriffen zu vermeiden, darf im Arbeitsspeicher des Messplatzrechners nur genau eine Instanz des Detektoren-Managers und genau ein Detektor-Objekt für jeden Detektor-Abschnitt der hardware.ini existieren.

Durch Verwendung des Singleton-Musters wird dafür Sorge getragen, dass genau eine Instanz des Detektoren-Managers erzeugt werden kann. Der Klassenkonstruktor ist private deklariert, er kann also von außen nicht direkt aufgerufen werden.

Zugriff auf die Instanz des Detektoren-Managers erhält man durch die Klassenmethode DetectorManager(). In dieser Methode wird ein statisches Detektoren-Manager-Objekt erzeugt und eine Referenz darauf zurückgeliefert. Die Objekterzeugung findet beim ersten Aufruf von DetectorManager() statt.

Die Objektzerstörung findet beim Programmende automatisch statt. Dazu wird der Destruktor benötigt, er muss also public deklariert sein. Trotzdem darf er von Klienten nicht direkt aufgerufen werden.

### 5.2.3 Erzeugen der Detektor-Objekte

Für jeden Detektor, der nutzbar sein soll, muss in der Datei hardware.ini ein Abschnitt vom Typ „[device*X*]“ existieren. Für jeden dieser Abschnitte erstellt der Detektoren-Manager das entsprechende Detektor-Objekt und fügt es seinem Container hinzu. Die Detektor-Abschnitte müssen dabei bei 0 beginnend aufsteigend und fortlaufend nummeriert sein. Der Detektoren-Manager nutzt dafür einen Schleifenzähler. Beim ersten Zählerwert, für den kein Detektor-Abschnitt existiert, geht der Detektoren-Manager davon aus, dass er alle Detektor-Abschnitte bearbeitet hat, weitere Abschnitte werden nicht berücksichtigt.

Um zu prüfen, ob die deklarierten Detektoren auch nutzbar sind, ruft der Detektoren-Manager für jeden Detektor die Methode Initialize() auf. Diese Methode gibt, je nachdem, ob die Initialisierung erfolgreich verlaufen ist, TRUE bzw. FALSE zurück. Stellt der Detektoren-Manager fest, dass eine Detektor-Initialisierung fehlgeschlagen ist - zum Beispiel, weil ein Detektor deklariert war, der nicht physisch an den Messplatzrechner angeschlossen ist, und somit keine Hardwarekommunikation hergestellt werden kann - meldet der Detektoren-Manager dem Programmbenutzer, dass einer der gewünschten Detektoren nicht nutzbar ist. Für diesen ini-File-Abschnitt wird kein Detektor erstellt.

Sollte nach vollständiger Verarbeitung der hardware.ini kein Detektor nutzbar sein, wird das Programm mit einem Hinweis darauf beendet.

Der erste Detektor wird als der aktive festgelegt.

### 5.2.4 Zugriff auf die nutzbaren Detektoren

Der Detektoren-Manager bietet Zugriff auf den aktuellen Detektor und bietet Möglichkeiten an, einen anderen Detektor als den aktiven festzulegen.

Wenn die Verarbeitung der hardware.ini abgeschlossen ist, und das Programm nicht beendet wurde, weil keine nutzbaren Detektoren vorhanden sind, steht also mindestens ein nutzbarer Detektor zur Verfügung. Einige der im XCTL-System implementierten Untersuchungsverfahren nutzen grundsätzlich nur einen Teil der Detektoren. So wird zum Beispiel in der Topographie nur mit nulldimensionalen Detektoren gearbeitet. Diese Subsysteme brauchen also eine Möglichkeit herauszufinden, ob unter den nutzbaren Detektoren auch mindestens ein Detektor mit der von ihnen benötigten Dimension vorhanden ist.

Die Scan-Dialoge bieten üblicherweise ein Kombinationsfeld an, in dem der für die Messung zu benutzende Detektor ausgewählt werden kann. Das Füllen dieser Detektorauswahl-Kombinationsfelder ist eine Funktion des Detektoren-Managers. Über einen optionalen Parameter der entsprechenden Methode können Entwickler festlegen, dass nur Detektoren einer bestimmten Dimension in diesem Kombinationsfeld erscheinen sollen.

Das Ergebnis der Auswahl eines Detektors aus einem solchen Kombinationsfeld ist immer der Index der gewählten Zeile. Eine Methode des Detektoren-Managers ermittelt für diesen Index und das Filterkriterium, dass bei der Erstellung des Kombinationsfeldes genutzt worden war, den entsprechenden Detektor. Der Programmbenutzer wählt den Detektor also über seinen Namen aus, das entsprechende Subsystem wählt den Detektor dann über den Kombinationsfeld-Index.

Die Ablaufsteuerung braucht eine Möglichkeit, um Zugriff auf einen Detektor zu erhalten, von dem sie nur den Namen kennt. Der Detektoren-Manager muss hier basierend auf dem Namen einen Zugriffsschlüssel liefern, aus dem der zugehörige Detektor rekonstruiert werden kann.

### 5.2.5 Existenz von Detektoren einer bestimmten Dimension

Einige Untersuchungsverfahren arbeiten nur mit Detektoren, die Messwerte in einer bestimmten Dimension ermitteln können. Zu Beginn des entsprechenden Gesamtvorgangs sollte also sinnvollerweise geprüft werden, ob für diese Untersuchung überhaupt ein entsprechender Detektor nutzbar ist. Die Methode DimAvailable(int dimension) liefert genau diese Information: Sie liefert TRUE zurück, wenn wenigstens ein Detektor der gewünschten Dimension verfügbar ist, sonst FALSE.

### 5.2.6 Zugriff auf die nutzbaren Detektoren

Im einfachsten Fall wird nur der Zugriff auf den aktiven Detektor gewünscht. Diesen erhält man mit der Methode GetDetector(). Ebenso einfach kann man den aktuellen Detektor modifizieren, wenn man schon ein Detektor-Objekt hat (SetDetector(const TDetector*)).

Möchte man Zugriff auf einen bestimmten Detektor, so nutzt man die Methode GetDetector(int index). Wenn der gewählte Detektor existiert, so erhält man einen Zeiger darauf, andernfalls einen Zeiger auf den aktiven Detektor.

Möchte man nur Detektoren, die Messwerte in einer bestimmten Dimension ermitteln können, so nutzt man die Methode GetDetector(int index, int dimension).

Nach demselben Schema existieren die Methoden SetDetector(int index) und SetDetector(int index, int dimension).

### 5.2.7 Zugriff über einen Identifikationsschlüssel

Die Ablaufsteuerung benötigt eine Möglichkeit, einen Detektor über seinen Namen auswählen zu können. Der ermittelte Detektor kann aber nicht direkt gespeichert werden, sondern nur ein Identifikationsschlüssel dafür. Die Methode GetIdByDescription(LPCSTR name) verwendet dafür den Listenindex des Detektors, da sich die Liste der Detektoren nach ihrer Erstellung nicht mehr ändert. Der zugehörige Detektor kann über GetDetector(int id) ausgewählt werden. Um prüfen zu können, ob ein Schlüssel korrekt auf einen der nutzbaren Detektoren verweist, existiert die Methode IsValidId().

## 5.3 Die Basisklasse für alle Detektoren TDetector

### 5.3.1 Verantwortung der Klasse

TDetector stellt eine Schnittstelle für die Operationen dar, die von allen Detektoren ausgeführt werden können. Diese Klasse ist Teil des Interfaces des Detektoren-Subsystems.

Sie wird als Basistyp für polymorphe Zeigeroperationen genutzt, wenn die auszuführenden Operationen auf allen Detektoren möglich sind.

### 5.3.2 Lebenslauf eines Detektors

Objekte vom Typ TDetector können nicht erzeugt werden, da diese Interfaceklasse als abstrakt deklariert ist. Jedoch erzeugt und verwaltet der Detektoren-Manager einen Container von Zeigern auf TDetector-Objekte. Alle Detektorobjekte, die vom XCTL-System genutzt werden können, werden beim Programmstart vom Detektoren-Manager erzeugt und beim Programmende zerstört.

Da die Verwaltung ausschließlich dem Detektoren-Manager vorbehalten ist, ist das Kopieren und Zuweisen von Detektoren nicht möglich. Das wird dadurch erreicht, dass der Kopierkonstruktor und der Zuweisungsoperator in der Basisklasse ohne Methodenkörper als private deklariert sind. Daher verweigert der Compiler jeden Versuch, ein Objekt vom Typ TDetector oder einer davon abgeleiteten Klasse zu kopieren.

### 5.3.3 Konfigurationsdaten

Alle Detektoren können bzw. müssen mit spezifischen Daten konfiguriert werden. Wenn der Detektoren-Manager seine Arbeit aufnimmt, wird der Typ des zu erstellenden Detektors ermittelt und dementsprechend ein neues Detektorobjekt erstellt. Die zugehörigen Daten können ihrem Geltungsbereich nach hierarchisch organisiert werden:

- Daten für alle Detektoren, zum Beispiel der Name oder der Typ des Detektors
- Daten für alle Detektoren einer bestimmten Dimension, z. B. ob das akustische Feedback aktiviert werden soll
- Daten für einen spezifischen Detektor, z. B. die Hardwareadressen des Radicon-Controllers

Günstigerweise wird bei der Objekterstellung unter C++ der Klassenhierarchie folgend genau so vorgegangen: zuerst wird ein TRadicon-Objekt als generischer Detektor erstellt, dann als nulldimensionaler und zum Schluss als Radicon-Detektor. Daher können für jede Erstellungstufe die obligatorischen und die optionalen Parameter ermittelt und für die Konfiguration des jeweiligen Objektzustandes verwandt werden.

Umgekehrt dazu läuft die Objektzerstörung ab: von der am weitesten abgeleiteten zu Basisklasse hin können bei der Zerstörung der entsprechenden Stufe die Konfigurationsdaten wieder gesichert werden.

Wie es für Softwaresysteme unter Windows 3 üblich war, wird auch die Konfiguration der XCTL-Software in Initialisierungsdateien bewahrt. Ab Windows 95 rät Microsoft dazu, solche Daten in der Registrierungsdatenbank des Betriebssystems abzulegen. Daher wird es über kurz oder lang notwendig, für das gesamte System einen abstrahierenden Mechanismus zu entwickeln, mit dem Konfigurationsdaten ermittelt und gespeichert werden können, unabhängig davon, wie das tatsächlich geschieht.

Die Verwendung von Initialisierungsdateien hat allerdings auch einen entscheidenden Vorteil: Mit mehreren Dateien kann man mehrere Konfigurationen vorbereiten, verteilen, speichern oder archivieren. Vielleicht wäre ja auch hier XML eine gute Lösung...

Hier bietet sich die Entwicklung und Integration einer Klassenhierarchie nach dem Bridge-Pattern an: Von einer Basisklasse mit dem gemeinsamen Interface werden weitere Klassen abgeleitet: z. B. eine für die Datenhaltung in der ini-Datei, eine für die Datenhaltung in der Registry, eine für Datenhaltung in XML. Diese Struktur wird so integriert, dass an allen Stellen im XCTL-Code, an denen bisher die API-Funktionen zum Lesen und Schreiben in ini-Dateien verwandt wurden, nun die entsprechenden Methoden der neuen Basisklasse genutzt werden. Nun kann einfach durch die Nutzung einer anderen Implementation transparent auf die Nutzung einer anderen Datenhaltung umgestellt werden.

Im Detektoren-Subsystem besitzt jede Stufe der Detektoren-Klassenhierarchie die nichtvirtuellen Methoden LoadDetectorSetting() und SavaDetectorSettings(). Mit LoadDetectorSettings wird versucht, einen Wert für alle in der jeweiligen Abstraktionsstufe gültigen Konfigurationsvariablen zu ermitteln. Ist kein derartiger Wert vorhanden, wird stattdessen ein Standardwert genutzt. Einstellungen, die für diesen Detektor ungültig sind, werden einfach ignoriert.

Mit SaveDetectorSettings werden die aktuellen Einstellungen des Detektors in die Initialisierungsdatei zurückgeschrieben. Schlüssel/Wert-Paare, die beim Programmstart nicht vorhanden waren, werden neu geschrieben. Das Konfigurationssystem ist somit mehr oder weniger selbstreparierend.

### 5.3.4 Der Messvorgang

TDetector bietet mit den virtuellen Methoden Initialize(), MeasureStart(), MeasureStop(), PollDetector() und GetData() eine Schnittstelle, mit denen eine Messung unter Nutzung von Pull-Synchronisation ermöglicht wird. Die jeweiligen konkreten Detektoren überschreiben diese Methoden mit dem Verhalten, das die Messung mit dem spezifischen Detektor ermöglicht. PollDetector() weist das Detektor-Objekt an, die aktuellen Intensitätswerte aus der Hardware zu lesen und daraus die Messwerte zu ermitteln. GetData() liefert dann die ermittelten Intensitätswerte in einem für die jeweilige Dimension geeigneten Datentyp.

## 5.4 Die Basisklasse für alle nulldimensionalen Detektoren TZeroDimDetector

Als Basisklasse für alle nulldimensionalen Detektoren soll TZeroDimDetector TDetector um die Funktionalitäten erweitern, die für alle nulldimensionalen Detektoren kennzeichnend ist.

So wird die Unterstützung des akustischen Feedback hier realisiert. Damit die Höhe des aktuellen Messwertes als akustisches Signal wiedergegeben werden kann, existieren die beiden Methoden GetSound() und SetSound().

Eine zweite Eigenart der nulldimensionalen Detektoren ergibt sich aus den Besonderheiten des ContinuousScan. Beim ContinuousScan wird die Messung gestartet und die Probe gleichmäßig bewegt. In regelmäßigen Abständen meldet dann das Detektor-Objekt dem ContinuousScan den aktuellen Messwert, ohne die Messung zu beenden.

Das ist ein Gegensatz zum „herkömmlichen“ Verfahren, das bei allen anderen Scans genutzt wird: Statt der Pull-Synchronisation aus MeasureStart(), MeasureStop(), PollDetector() und GetData() wird eine Push-Synchronisation mit InitializeEvent(), EventHandler() und KillEvent() genutzt.

Der ContinuousScan wurde als letztes Scanverfahren realisiert. Dazu mussten die Methoden zur Messwertgewinnung nachträglich in das bestehende System integriert werden. Da zu diesem Zeitpunkt leider nur unzureichende Dokumentation zum Subsystem existierte, konnte leider nur eine *funktionierende* Lösung gefunden werden, keine *gute*.

Leider ist dieser an sich sinnvolle Mechanismus nur für einen Teil der Detektoren und nur für einen einzigen Scan-Vorgang mit sozusagen proprietäeren Mitteln erreicht worden. Es stellt eine Besonderheit dar, die sich nicht einfach auf alle Detektoren verallgemeinern lässt und so bei jeder Änderung separat berücksichtigt werden muss. Ich konnte diesen Misstand aus zwei Gründen nicht beseitigen:

Erstens werden zur Realisierung Timer aus dem Windows-Multimedia-API eingesetzt. Diese Timer können nach Ablauf des Zeitintervalls per Callback eine angegebene Funktion aufrufen. Wohlgemerkt ist hier nur eine C-Funktion gemeint, keine Methode eines Objektes.

Damit der Mechanismus trotzdem verwendet werden kann, wird beim Start der kontinuierlichen Messung ein Zeiger auf das eine Detektor-Objekt, mit dem diese Messung durchgeführt wird, in einem Attribut gespeichert. Das bringt die Gefahr des Überschreibens mit sich, weil eben nur genau ein derartiger Zeiger verwaltet wird. Ich habe das unter der Voraussetzung getan, dass der ContinuousScan bisher nie mit zwei Detektoren gleichzeitig durchgeführt werden konnte und wurde.

Um der Schnittstelle der Timer-API-Funktion gerecht zu werden, wird also eine statische Klassenmethode zurechtgecastet, sicher kein schönes C++. Diese Klassenmethode ermittelt nach ihrer Aktivierung das zu verwendende Detektor-Objekt und führt die notwendigen Operationen darauf aus.

Es fehlt dem gesamten XCTL-Projekt also ein Timermechanismus, der a) unabhängig von Fenstern arbeitet und b) mit objektorientierter Programmierung klarkommt.

Der zweite Grund liegt in der ungeheuren Komplexität der Klasse TScan, von der der ContinuousScan nur ein Teil ist. Jede Änderung daran konnte ich nur unter größter Vorsicht vornehmen. An ein komplettes Umstellen der Zusammenarbeit zwischen TScan und TZeroDimDetector ist im Moment nicht zu denken.

## 5.5 Der Testdetektor TZeroDimSimpleTestDetector

TZeroDimSimpleTestDetector stellt einen einfachen Testdetektor zur Verfügung. Er berechnet aus der aktuellen Position auf den Achsen Omega und Psi über eine „magische“ Formel einen Messwert. Wie diese Formel zustande kam, lässt sich nicht mehr nachvollziehen.

Dazu überschreibt diese Klasse lediglich die Methode PollDetector().

## 5.6 Der Testdetektor Testdev

Testdev ermöglicht die Nutzung eines Testdetektors, der eine Messung mit Referenzdaten simuliert. Die Messdaten einer früheren realen Messung wurden geeignet aufbereitet in der Textdatei testdev.dat abgelegt. Testdev.dat muss bei der Initialisierung eines Detektors von Typ Testdev im gleichen Verzeichnis wie die ausführbare Projektdatei liegen.

Ein Objekt vom Typ Testdev übernimmt an seinem Interface Anforderungen des XCTL-Programms und erfüllt diese Aufgaben mit Hilfe eines TestdevData-Objekts. TestdevData liest die Daten aus der Messwertdatei und füllt damit einen Datencontainer. Dieser Container wird durch dynamische Speicherverwaltung und mehrfach indirekte Zeiger realisiert, da die entsprechende STL-Unterstützung fehlt. Die Messdaten werden also, wenn ein Testdev-Detektor genutzt werden soll, vollständig in den Arbeitsspeicher geladen.

Zur Ermittlung eines Messwertes werden nun die aktuellen Positionen der Motoren Kollimator, Tilt und Beugung fein ausgewertet und daraus ein Messwert aus der Messdatenstruktur interpoliert.

Zu einem späteren Zeitpunkt könnten die Softwarekomponenten für diesen Detektor in zweierlei Hinsicht optimiert werden:

Einerseits sollte die Verwendung von Zeigerarithmetik durch einen geeigneten Container aus der C++-STL ersetzt werden. Andererseits könnte hier ein intelligenter Caching-Mechanismus dafür sorgen, dass nur Teile der Messdaten im Speicher gehalten werden müssen.

## 5.7 Der Detektor TGenericDetector

TGenericDetector ermöglicht die Messung mit nulldimensionalen Detektoren, die an einer russischen Controllerkarte angeschlossen sind. Auf dieser Karte arbeitet ein AM9513[33] der Firma AMD, ein Zähler/Timer-Chip mit 5 separaten 16bit-Zählern.

[33] Details siehe http://www.measurementcomputing.com/PDFmanuals/9513A.pdf, dies ist die Dokumentation zu einer identischen Lizenzproduktion. AMD selbst scheint sich nicht daran erinnern zu können, so einen Chip je hergestellt zu haben, man sucht vergebens auf search.amd.com

Ein Objekt vom Typ TGenericDetector übernimmt an seinem Interface Anforderungen des XCTL-Programms und erfüllt diese Aufgaben mit Hilfe eines TAm9513-Objekts. Die Klasse überschreibt dazu lediglich die Methoden Initialize(), MeasureStart(), MeasureStop() und PollDetector().

## 5.8 Der Detektor TRadicon

TRadicon ermöglicht die Messung mit dem russischen Radicon-Detektor.

Dazu arbeiten, äquivalent zur Klasse TGenericDetector im vorigen Abschnitt zwei Objekte TRadicon und TRadiconHW zusammen, indem TRadicon an seinem Interface Anforderungen der Klienten übernimmt und diese mit Hilfe eines TRadiconHW-Objekts erfüllt.

Damit die Radicon-Controllerkarte genutzt werden kann, muss zuerst eine Firmware in die Controllerkarte geladen werden. Die Datei mit der Firmware muss sich im selben Verzeichnis wie die ausführbare XCTL-Software befinden und scs.prg heißen. Während der Abarbeitung der Initialize()-Methode wird der Inhalt dieser Datei zum Controller übertragen.

Spezifische Einstellungen für den Radicon-Detektor können über ein Instanzfenster der Klasse TScsParametersDlg festgelegt werden.

In der Klasse TRadicon werden die Methoden zur kontinuierlichen Messung überschrieben. Offenbar hat das Team, das die Integration des ContinuousScan vorgenommen hat, dies zuerst nur für den Radicon-Detektor getan und dann auf alle Detektoren verallgemeinert (auch wenn das nur für nulldimensionale Detektoren Sinn macht).

Für einen Radicon-Controller können einige Parameter konfiguriert werden, die es nur für diesen Detektor gibt: die Intervallgrenzen und die zu nutzende Beschleunigungsspannung. Diese Werte werden entweder über die Initialisierungsdatei hardware.ini oder über das Dialogfenster für spezifische Einstellungen gesetzt. Beide Mechanismen befinden sich innerhalb des Detektoren-Subsystems und müssen daher nicht über das Subsystem-Interface auf die Detektoren zugreifen. Im Gegenteil, sie kennen die Klasse TRadicon und ihr Interface und können direkt mit Objekten von diesem Typ interagieren.

## 5.9 Die Basisklasse für alle eindimensionalen Detektoren TOneDimDetector

Als Basisklasse für alle eindimensionalen Detektoren soll TOneDimDetector in erster Linie das Interface für alle Operationen realisieren, die für eindimensionale Detektoren möglich sein sollen. Außerdem wird in dieser Klasse vererbbares, gemeinsames Verhalten für eindimensionale Detektoren festgelegt.

### 5.9.1 Messkanäle

Eindimensionale Detektoren teilen die Sensorfläche in einer Dimension in mehrere gleichbreite Kanäle (Streifen). Für jeden dieser Kanäle wird ein eigener Intensitätswert ermittelt. Die Anzahl der vorhandenen Kanäle ist eine gerätespezifische, fixe Größe. Allerdings lassen sich mehrere nebeneinander liegende Kanäle zu einer Kanalgruppe zusammenschließen, für die ein gemeinsamer Intensitätswert ermittelt wird. Durch die Verwendung solcher Kanalgruppen lässt sich die Anzahl der logischen Kanäle eines eindimensionalen Detektors praktisch bis auf eins reduzieren.

TOneDimDetector bietet dementsprechend verschiedene Methoden, um die Verwendung von Kanälen zu steuern. GetAddedChannels() und SetAddedChannels() kontrollieren die Anzahl der Kanäle, die zu einer Gruppe zusammengefasst sind. GetAngleStep() und SetAngleStep() kontrollieren, welcher Winkelabstand von einem Kanal erfasst werden kann. GetWidth() liefert den Winkel, der von einer Kanalgruppe erfasst wird.

GetFirstChannel(), GetLastChannel() und SetChannelRange beeinflussen die Größe der Detektorfläche, indem sie steuern, ab welchem Kanal bis zu welchem Kanal gemessen werden soll. Mit GetAngleRange() lässt sich abfragen, welcher Winkelbereich mit dem aktuellen Sensorausschnitt erfasst werden kann. Somit lässt sich also der Ort, die Breite und die Granularität der Messwerterfassung genau festlegen.

GetReadLeftFirst() und SetReadLeftFirst() kontrollieren, ob beim Auslesen der Messwerte aus der Hardware von links nach rechts oder umgekehrt vorgegangen werden soll. Im Ergebnis ergibt sich dadurch ein horizontal gespiegelter Kurvenverlauf.

### 5.9.2 Messwerterfassung

Zur Abstraktion von der spezifischen Hardware implementiert TOneDimDetector ein Feld von Werten, in das die von TOneDimDetector abgeleiteten Messadapterklassen die aus ihrer Hardware ermittelten Werte für jeden verwendeten Kanal ablegen. Um das Konzept der logischen Kanäle, also der Kanalgruppen zu realisieren, berechnet TOneDimDetector aus diesen Kanal-Rohdaten dann die Messwerte für Kanalgruppen und erstellt daraus eine Datenstruktur vom Typ TCurve.

Sollte die Hardware einen Überlauf des gültigen Messwertebereiches gemeldet haben, so ergibt IsHardOverflow() wahr. Sollte sich erst bei der Berechnung der Messwerte ein solcher Überlauf ergeben, so gibt IsSoftOverflow() wahr zurück. In beiden Fällen müssen die ermittelten Werte als zu stark fehlerbehaftet angesehen werden.

Während der Verarbeitung der Rohdaten wird der Kanal mit der höchsten Einzelintensität und das zugehörigen Intensitätsmaximum ermittelt. Diese Werte können mit GetMaximumChannel() und GetMaximumIntensity() abgefragt werden.

## 5.10 Der Testdetektor TOneDimSimpleTestDetector

Äquivalent zum TZeroDimSimpleTestDetector stellt TOneDimSimpleTestDetector einen einfachen eindimensionalen Testdetektor zu Verfügung. Dazu überschreibt diese Klasse die Methode PsdReadOut, um simulierte Messdaten in den Messwertepuffer zu schreiben.

Der Testdetektor berücksichtigt dabei die aktuelle Position auf der Theta-Achse. Mithilfe einer „magischen“ Formel, deren Herkunft sich auch hierbei nicht mehr erklären lässt, werden für alle Kanäle Messwerte ermittelt und im Messwertepuffer abgelegt.

## 5.11 Der Detektor TStoePsd

TStoePsd ermöglicht die Messung mit einem PSD der amerikanischen Firma Stoe. Dieser Detektor wird nicht mehr verwendet. Leider wurde dieser Fakt erst gegen Ende meiner Arbeiten bekannt, so dass ich den zugehörigen Code nicht mehr selbst entfernen konnte. Er sollte demnächst aus dem Projektcode entfernt werden.

## 5.12 Der Detektor TBraunPsd

TBraunPsd ermöglicht die Messung mit einem PSD der deutsch/österreichischen Firma M. Braun. Die Softwareunterstützung für diesen Detektor stellt insofern eine Besonderheit dar, als dass nicht alle für die Nutzung des Detektors notwendigen Funktionen von der XCTL-Software zur Verfügung gestellt werden können.

Zum Kalibrieren des Detektors wird eine Software des Herstellers verwandt, deren Interna nicht bekannt und deren Funktion somit nicht einfach innerhalb des XCTL-Systems nachgebildet werden kann.

Zum Auslesen der Messwerte des Detektors kommt ebenfalls eine Komponente des Detektoren-Herstellers zum Einsatz: die asa.dll. Diese Bibliothek bietet vier Funktionen zum Konfigurieren der Kommunikation mit der Controllerkarte und zum Auslesen der Daten. Alle vier Funktionen werden dynamisch zum Softwaresystem gebunden. Auch die internen Mechanismen dieser Bibliothek sind nicht genau bekannt, so dass eine nicht zu behebende Abhängigkeit von Fremdsoftware besteht. Insbesondere bei der Verwendung unter Windows NT und Nachfolgern kann die Art des Hardwarezugriffes innerhalb der DLL zu Problemen führen, die die Projektgruppe nicht selbst beheben kann.

Ich habe diesen Code mit besonderer Vorsicht behandelt, da keine Informationen über das Hardware-Kommunikationsprotokoll oder die impliziten Annahmen der Fremdsoftware verfügbar sind. Der Code ist, bis auf geringfügige Anpassungen an C++, weitestgehend unverändert.

Als Spezialisierung von TOneDimDetector muss TBraunPsd:

- die Hardware einsatzbereit machen
- die generischen Operationen eines eindimensionalen Detektors in Anweisungen des spezifischen Detektors umsetzen
- die vom Detektor ermittelten Werte in ein eindimensionales Feld von Messwerten in den Messdatenpuffer von TOneDimDetector schreiben.

# 6 Ausblick

## 6.1 Verbesserungspotential

Softwaresysteme vom E-Typ müssen sich kontinuierlich ändern und bieten immer Gelegenheit für Verbesserungen. Im Folgenden werde ich Hinweise für mögliche Verbesserungen geben. Diese Hinweise betreffen meist das gesamte Projekt, auch wenn sie natürlich in erster Linie aus meinen Arbeiten am Detektoren-Subsystem entstanden sind.

Wenn es möglich war und wenn der Aufwand für die Arbeit einer Einzelperson nicht zu groß war, habe ich diese Verbesserungsvorschläge im Detektoren-Subsystem umgesetzt.

Wenn ich eine mögliche Verbesserung nicht umgesetzt habe, dann im Wesentlichen aus folgenden Gründen:

- Die Rückwärtskompatibilität zu Borland C++ 4.5 verhinderte die Umstellung
- Die Nutzung von Borland C++ im allgemeinen verhinderte die Umstellung
- Für die Änderung wären zu umfangreiche Umstellungen im mehreren Teilen des Projektcodes nötig gewesen
- Eine vernünftige Lösung für das gesamte Projekt müsste zuvor gefunden werden

### 6.1.1 Umgang mit Fehlern

In der Art, wie mit Programmfehlern umgegangen wird, sieht man die differierenden Einstellungen der verschiedenen Entwickler zu diesem Thema. Selbst innerhalb des Detektoren-Subsystems existieren verschiedene Ansätze zur Fehlererkennung und Fehlerbehandlung.

Die Fehlererkennung wird unterschiedlich intensiv betrieben. Zu oft werden Werte oder Systemzustände nicht als fehlerhaft erkannt. Dazu wäre es notwendig, mehr testenden Code einzufügen.

Wenn die Fehlerbehandlung nicht in der Funktion stattfindet, in der der Fehler erkannt wurde – und das sollten die meisten Fälle sein, insbesondere in Bibliotheken möchte man normalerweise dem Nutzer der Bibliothek überlassen, wie er auf den Fehler reagieren möchte – muss das Auftreten des Fehlers in irgendeiner Form angezeigt werden. Dazu werden im Projekt entweder Funktionen mit einem Rückgabewert vom Typ bool ( fehlerfrei/fehlerhaft ) oder bei differenzierteren Rückmeldungen Funktionen mit dem Rückgabetyp int genutzt. Außer einem Fehlercode können keine weiteren Informationen zurückgeliefert werden.

In beiden Fällen bleibt das Problem, dass man diesen Fehler so leicht ignorieren kann, weil man nicht darauf reagieren muss.

In C++ wurde dafür der Exception-Mechanismus geschaffen: ein Codeabschnitt kann über das Auftreten eines Fehlers benachrichtigen, indem ein Exception-Objekt erstellt und zurückgegeben werden kann. Da man dieses Objekt ganz nach eigenen Vorstellungen entwerfen kann, kann man in ihm beliebige Zusatzinformationen speichern bzw. beliebige Aktionen ausführen.

Der zweite Vorteil besteht darin, das Exceptions behandelt werden müssen; möchte man sie ignorieren, so muss man das explizit tun.

Durch die sicherzustellende Rückwärtskompatibilität zu Borland C++ 4.5 können Exceptions praktisch nicht genutzt werden. Von den Vorteilen dieser Möglichkeit kann also erst profitiert werden, wenn nicht mehr mit Borland C++ entwickelt werden muss.

Auch die Fehlerbehandlung fällt unterschiedlich aus: Je nach Geschmack des Programmierers wurde ein Meldungsfenster mit einer Fehlermitteilung geöffnet oder eine Nachricht in der Statuszeile im unteren Bereich des Hauptfensters eingeblendet. Dabei ist leider keinerlei konsistentes Schema festzustellen.

Insgesamt betrachtet sind zu diesem Thema recht aufwendige weitere Arbeiten nötig. Zuerst müssten Wege festgelegt werden, wie auf Fehler reagiert werden soll. Dazu müssten für das gesamte Projekt einheitliche Methoden implementiert werden.

Anschließend würde eine schwierige Phase beginnen: jeweils ein Subsystem nach dem anderen wird auf die Nutzung von Exceptions umgestellt und alle nutzenden Subsysteme müssen sie behandeln. Diese Umstellung lässt sich leider nicht lokal Subsystem für Subsystem vollziehen; sie verändert die Subsystem-Interfaces und betrifft immer mehrere Subsysteme gleichzeitig.

## 6.2 Nutzung der Möglichkeiten von C++

### 6.2.1 Nutzung von const

Wenn Objekte nicht verändert werden sollen bzw. dies tatsächlich auch nicht getan wird, so sollte das explizit festgelegt werden. So kann der Compiler diese Absicht auch überprüfen und ihre Einhaltung forcieren, wodurch Fehler vermieden werden. Die Information über die Konstanz eines Objektes stellt eine Zusicherung an andere Entwickler dar. Sie können sich darauf verlassen, dass am Objekt keine Änderungen vorgenommen werden.

Insbesondere Methoden, die lediglich Werte zurückgeben bzw. Werte aus vorhandenen Werten berechnen, können einfach in konstante Methoden umgewandelt werden. Dabei muss allerdings beachtet werden, dass die Konstanz einer Methode Teil ihrer Signatur wird. Derartige Änderungen an Interface-Funktionen sind also Änderungen am Interface.

Beispiel für eine Methode, bei der die const-Nutzung problemlos möglich ist:

```
UINT TOneDimDetector::GetMaximumChannel( void ) const
{
  if ( bReadLeftFirst )
    return uMaximumChannel;

  return ( GetChannelNumber( ) - uMaximumChannel );
}
```

Diese Verbesserungen sind dann einfach und sofort umsetzbar, wenn sie Subsystem-lokale Objekte betreffen. Bei Interface-Funktionen sind wieder mehrere Subsysteme gleichzeitig betroffen. Hier muss zuerst überprüft werden, ob Funktionalitätsanbietern und –nutzern die implizite Konstanz bei der Programmierung bewusst war.

### 6.2.2 Nutzung von Referenzen

In C++ wurde das Konzept der Referenz eingeführt. Eine Referenz verweist – ähnlich einem konstanten Zeiger – auf einen vorhandenen Speicherbereich.

Im Unterschied zu einem Zeiger ist eine Referenz immer mit einem Speicherbereich verknüpft. Es gibt keine Nullreferenz. Die Dereferenzierung eines Nullzeigers lässt ein Programm sofort abstürzen – mit Referenzen ist das nicht möglich.

Viele Aufgaben, die C-Programmierer mit Zeigern lösen würden, lassen sich mit C++-Referenzen sicherer lösen. Da Referenzen syntaktisch wie normale Variablen behandelt werden, entfallen potentielle Fehlerquellen wie Zeigerarithmetik oder die inkorrekte Referenzierung/Dereferenzierung von Zeigern.

Im nächsten Beispiel werden zwei funktional äquivalente Funktionen dargestellt, die den Tausch des Inhaltes zweier int-Variablen realisieren. Eine Version ist mit Zeigern realisiert, die zweite mit Referenzen.

```
void swap_ptr(int* i, int* j)
{
      int tmp = *i;
      *j = *i;
      *i = tmp;
}

void swap_ref(int& i, int& j)
{
      int tmp = i;
      j = i;
      i = tmp;
}


main()
{
      int a=7, b=12;

      swap_ptr(&a, &b);
      swap_ref(a, b);
}
```

Sowohl in der Funktionsdefinition der Zeigervariante als auch bei deren Aufruf muss der Entwickler sehr genau überlegen, ob er im Moment mit der Zeigeradresse oder seinem Inhalt arbeiten möchte. Fehler sind hier sehr wahrscheinlich, und nur mit einem Debugger zu finden. Noch problematischer ist die Tatsache, dass beim falschen Einsatz von Zeigern die Speicherbereiche von anderen Programmdaten überschrieben werden können. Durch den (falschen) Einsatz von Zeigern wird das Lokalitätsprinzip verletzt, was Fehlersuche und Wartung enorm erschweren kann.

Tatsächlich führt genau das Problem der fälschlich überschriebenen Speicherbereiche zu großen Problemen im XCTL-Projekt, doch dazu später mehr.

Insbesondere bei der Übergabe von Objekten als Parameter bzw. bei der Rückgabe eines Objektes aus einer Methode können Referenzen sinnvoll, wenn nicht gar unverzichtbar sein. Wird ein Objekt als Parameter übergeben bzw. als Rückgabewert zurückgeliefert, so erfordert die C++-Semantik immer die Übergabe einer Objektkopie. Im einfachsten Fall ist das nur ein Effizienznachteil, in anderen Fällen ist das Kopieren eines Objektes sogar problematisch oder unerwünscht.

In allen diesen Fällen können Referenzen helfen. Da sie den Speicherbereich des entsprechenden Objektes *referenzieren*, ist das Kopieren unnötig. Um die Semantik einer Übergabe „by value“ zu erreichen, bieten sich const-Referenzen an.

### 6.2.3 Zeichenfelder mit konstanter Länge

Beispiel aus m_scan.cpp:

```
char szMsgLine132[] = "Durch die Einbeziehung des Beschleunigungs-
bzw.\nBremsweges (%s) kommt es zu ungültigen Motorpositionen!";

...

BOOL TSetupContinuousScanDlg::CanClose( void )
{
        char buf[ MaxString ];
  ...
  sprintf(buf2,mGetDF(),dAccelerationDistance);
  sprintf(buf,szMsgLine132,buf2);
  MessageBox(GetFocus(),buf,szMsgFailureScan,MBINFO);
  ...
}
```

Bei Versuchen mit dem Debugger von Visual C++ 6 stellte sich heraus, dass das Programm unerwünscht Speicherbereiche überschrieb. Das Problem liegt in der Verwendung von Zeichenfeldern mit fester Länge. Wenn nun, wie es in den älteren Versionen des Projektcodes war, MaxString mit 80 definiert war, passte schon der Formatstring nicht in den Puffer. Er überschrieb teilweise Speicherbereiche hinter der Feldgrenze.

Das Problem wurde zu Ungunsten des Speicherverbrauchs entschärft ( nicht gelöst ), indem MaxString auf 512 gesetzt wurde. Manchen Programmierern schien das Problem bewusst zu sein, denn sie verwendeten semantisch fragwürdige Konstrukte wie

```
char FileName[2*MaxString];
```

Für jeden Puffer müsste durchdacht werden, wie lang sein potentieller Inhalt werden kann. Dabei muss aus Sicherheitsgründen immer das Maximum verwandt werden, auch wenn empirisch nur ein Bruchteil davon benötigt wird. Das ist sicherlich nicht immer leicht.

Das Problem lässt sich nun auf zweierlei Art lösen: Sicherheitsbewusste C-Programmierer würden zum Beschreiben von Pufferspeichern Funktionen nutzen, die die maximale Anzahl zu schreibender Zeichen berücksichtigen. Damit wird das Überschreiben fremder Speicherbereiche vermieden. Ist eine Zeichenkette zu lang für den Puffer, würde sie einfach abgeschnitten. Für jeden verwendeten Puffer ( und das können sehr viele sein ) müsste seine Maximallänge im Code abgelegt werden und bei jeder Schreiboperation genau die Maximallänge verwandt werden.

Mit den Mitteln von C++ könnte man mit Streams arbeiten. Streams sind abstrakte Datenströme mit variabler Länge. Für die korrekte Speicherverwaltung sorgt die C++-Standardbibliothek im Hintergrund. Insbesondere für Zeichenketten existiert die Klasse strstream. Spezielle Kompatibilitätsmethoden sollen ermöglichen, einfach solche strstream-Objekte statt Zeichenpuffern zu verwenden.

Die konsequente Ersetzung aller Zeichenpuffer fester Länge wurde durch den Einsatz von Borland C++ 4.5 verhindert.

### 6.2.4 Nutzung der C++-STL

C++ bietet mit seiner Standard-Template-Library Vorlagen für Datenstrukturen und Algorithmen, sie nicht von jedem Programmierer neu erdacht oder implementiert werden müssen. Dabei werden Operationen einfach möglich, für die C-Programmierer viel eigenen Code entwickeln und testen müssten. Außerdem wird der Austausch unter den Programmierern erleichtert, da es sich dabei um standardisierte, bekannte Muster handelt.

Im XCTL-Projekt besteht an vielen Stellen Bedarf an Datencontainern wie Vektoren, Listen oder Assoziativen Speichern bzw. an standardisierten Algorithmen wie Iterieren, Suchen, Sortieren etc.

Beispiele:

- TDetectorManager verwaltet die nutzbaren Detektoren in einer selbstimplementierten Datenstruktur, die die Operationen „Element hinzufügen“ und „Struktur durchsuchen“ ermöglichen muss.
- An allen Stellen, wo Push-Synchronisation eingesetzt werden sollte, müssen Datenstrukturen gepflegt werden, die die zu benachrichtigenden Datenempfänger verwalten.

- In der Fensterverwaltung haben die Herren Kullmann und Reinecker eigene Datenstrukturen entwickeln müssen, um die offenen Fenster zu verwalten.
- Der Transferdatentyp TCurve für eindimensionale Messdaten ist ein Beispiel für solch einen selbst erstellten Datencontainer.

Auch der Einsatz dieser Mittel wurde durch Borland C++ 4.5 verhindert.

## 6.3 Einsatz von Entwurfsmustern

Der Einsatz von Entwurfsmustern kann die Verständlichkeit und Wartbarkeit von Programmcode erhöhen. Statt der aufwendigen Selbstentwicklung können standardisierte Problemlösungen eingesetzt werden, für die ein kommunizierbarer Name existiert.

Neulingen im Projekt kann der Einstieg so erleichtert werden, da ein komplexer Zusammenhang allein mit dem Namen des Entwurfsmusters erklärt werden kann.

# 7 Anhang

## 7.1 Quellenverzeichnis

[BAL98] Balzert, Helmut:
Lehrbuch der Softwaretechnik. Band 2 Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Heidelberg, Berlin. Spektrum Akademischer Verlag. 2000

[BAL00] Balzert, Helmut:
Lehrbuch der Softwaretechnik. Band 1 Softwareentwicklung. 2. Auflage. Heidelberg, Berlin. Spektrum Akademischer Verlag. 2000

[TDM97] DeMarco, Tom:
Warum ist Software so teuer? Und andere Rätsel des Informationszeitalters. München, Wien, Hanser 1997

[TDM99] DeMarco, Tom:
Wien wartet auf Dich! : Der Faktor Mensch im DV-Management. 2. aktualisierte und erweiterte Auflage. München, Wien. Hanser 1999

[ECK98] Eckel, Bruce:
In C++ denken. München, Prentice Hall 1998

[FOW00] Fowler, Martin:
Refactoring – Wie sie das Design vorhandener Software verbessern. Addison-Wesley 2000

[GOF96] Gamma, Erich:
Entwurfsmuster : Elemente wiederverwendbarer objektorientierter Software. Bonn. Addison-Wesley Longman, 1996

[FWS2] Lehman et al:
Preprints of the (first) three FEAST Workshops, Departement of Computing, Imperial College of Science, Technologie and Medicine.
http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/2feastwks.pdf

[LEH96] Lehman, M. M.:
Laws of Software Evolution Revisited, pos. pap., European Workshop on Software Process Technoology 1996, Springer Verlag 1997, pp. 108-124.
http://www.doc.ic.ac.uk/~mml/feast2/papers/pdf/556.pdf

[MEY95] Meyers, Scott:
Effektiv C++ programmieren : 50 Möglichkeiten zur Verbesserung Ihrer Programme. Bonn; Paris; Reading Mass. Addison-Wesley 1995

[MEY97] Meyers, Scott:
Mehr effektiv C++ programmieren : 35 neue Wege zur Verbesserung Ihrer Programme und Entwürfe. Bonn; Paris (u.a.). Addison-Wesley-Longman 1997

[SNE91] Sneed, Harry:
Softwarewartung und –wiederverwendung. Band 1 Softwarewartung. Köln. Verlagsgesellschaft Rudolf Müller 1991

[SPHA00] Sphar, Chuck:
Visual C++ 6. Unterschleißheim. Microsoft Press Deutschland 2000

[STRO98] Stroustrup, Bjarne:
Die C++-Programmiersprache. 3. aktualisierte und erweiterte Auflage. Bonn. Addison-Wesley 1998

[STRO00] Stroustrup, Bjarne:
The C++ Programming Language. Special 3rd Edition. Addison Wesley Publishing, 2000

[CSTF] Stroustrup, Bjarne:
Bjarne Stroustrup's C++ Style and Technique FAQ
http://www.research.att.com/~bs/bs_faq2.html#friend

[WS] Williams, Sam:
http://www.salon.com/tech/feature/2002/04/08/lehman/index.html

## 7.2 Automatische Dokumentation mit doxygen

Um mein Projektziel zu erreichen, musste ich das Problem der Redokumentation lösen.

Das Grundproblem aller Dokumentation ist die Zeitpunktbezogenheit. Dokumentation von gestern kann heute schon veraltet und daher (fast) wertlos sein. Im Sinne des XCTL-Projektes ergeben sich daraus für mich zwei Konsequenzen: Es hat wenig Sinn, veraltete Ist-Zustände zu dokumentieren und ein in Frage kommendes Dokumentationssystem muss in der Lage sein, jederzeit den aktuellen Ist-Zustand zu dokumentieren.

Aktuelle Änderungen in viele alte Dokumente einzuarbeiten ist enorm aufwendig. Besser wäre es, wenn man den alten Satz an Dokumenten archivieren und den aktuellen automatisch von Grund auf neu erstellen könnte. Zusätzlich könnte man dieses System in Zusammenarbeit mit einer guten Quellcode-Versionsverwaltung auch auf beliebige ältere Entwicklungsstände anwenden.

Zu diesem Zwecke habe ich oft und gern doxygen eingesetzt. Doxygen ist ein solches automatisches Dokumentationssystem, das erfolgreich mit vielen Open-Source-Projekten eingesetzt wird. Es ist freie Software unter der GPL und deshalb gerade für Zero-Budget-Projekte eine gute Wahl.

Doxygen kann Quelltexte in C, C++, Java, IDL, PHP und C# verarbeiten und daraus Entwicklerdokumentation in HTML, Latex, RTF, Postscript, CHM, Unix Manpage und XML generieren.

Ein Script extrahierte dabei zweimal täglich den aktuellen Stand der Entwicklung aus dem CVS-System und generierte daraus, wenn sich Quellen geändert hatten, Online-Dokumentation.