Diplomarbeit

Automatisierte Ermittlung und Bewertung von Subsystemschnittstellen

Tobias Thiel

Betreuer: Dipl.-Inf. Kay Schützler

Humboldt-Universität zu Berlin
Institut für Informatik
Lehr- und Forschungsgebiet Softwaretechnik
Prof. Dr. Klaus Bothe

Erklärungen

Selbständigkeitserklärung

Hiermit erkläre ich, dass die vorliegende Diplomarbeit ausschließlich unter Verwendung der angegebenen Quellen selbständig von mir erstellt wurde.

Berlin, den 12. Januar 2004

Tobias Thiel

Einverständniserklärung

Hiermit erkläre ich mein Einverständnis mit der öffentlichen Ausstellung meiner Diplomarbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin.

Berlin, den 12. Januar 2004

Tobias Thiel

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Subsysteme und Schnittstellen

Ein größeres Programmsystem ist im Allgemeinen zu komplex, um es als Einheit zu verstehen. Daher wird es in mehrere Komponenten, genannt Subsysteme, eingeteilt, wobei jedes Subsystem eine Teilaufgabe übernimmt. Die Einteilung kann sich dabei nach verschiedenen Aspekten richten: so kann, zum Beispiel, jeder Anwendungsfall durch ein Subsystem realisiert werden. Subsysteme können aber auch Objekte der realen Welt, wie z.B. technische Geräte, wiederspiegeln oder einfach nur Funktionen nach Aufgabenbereichen zusammenfassen.

Die jeweilige Teilaufgabe sollte von einem Subsystem möglichst eigenständig und unabhängig von anderen Subsystemen realisiert werden, wobei das Prinzip des „Information Hiding“ angewendet werden sollte. Das bedeutet, dass den anderen Subsystemen soviel Details wie möglich verborgen bleiben sollten, also die interne Struktur und Arbeitsweise des Subsystems nicht bekannt sein sollte. Die einzelnen Subsysteme müssen aber miteinander kommunizieren, um ihre Arbeit zu koordinieren. Dies geschieht im Allgemeinen durch Aufrufen von Funktionen des jeweils anderen Subsystems, was auf der Ebene des Quelltextes bedeutet, dass bestimmte Teile (z.B. einzelne Klassen oder Funktionen) jedes Subsystems anderen Subsystemen bekannt gemacht werden müssen. Diese Menge der allgemein bekannten Teile eines Subsystems bildet die Schnittstelle des Subsystems.

Beim Entwurf eines Systems sollte auf eine gute Subsystemeinteilung geachtet werden, da dies viele Vorteile bringt: Ein einzelnes Subsystem ist leichter zu verstehen als das Gesamtsystem, Fehler sind schneller zu lokalisieren und zu beheben, Änderungen betreffen meist nur ein Subsystem; wenn die Schnittstellen festgelegt sind, kann die Arbeit am System auf mehrere Entwickler aufgeteilt werden und die Realisierung der Funktionalität eines Subsystems kann auch durch den kompletten Austausch des Subsystems verändert werden.

Wie sollen die Schnittstellen aussehen? Aus dem Prinzip des „Information Hiding“ folgt, dass die Schnittstellen eines Subsystems nur so viel wie zur Benutzung des Subsystems nötig ist bekannt geben sollten. Dabei sollte vor allem Funktionalität angeboten werden aber keine Details wie interne Datenstrukturen. Daraus kann man folgern, dass Schnittstellen also möglichst klein und übersichtlich sein sollten, denn eine kleine Schnittstelle erleichtert es, die Benutzung des Subsystems zu verstehen oder auch das Subsystem auszutauschen.

Andererseits muss die Schnittstelle aber so viele Funktionen umfassen, dass eine sinnvolle Benutzung möglich ist und die einzelnen Funktionen verständlich sind. So könnte man, zum Beispiel, die gesamte Funktionalität des Subsystems in einer einzigen Funktion zur Verfügung stellen, deren Verhalten über zahlreiche Parameter

gesteuert wird. Damit hätte man zwar eine kleine Schnittstelle, die aber weder leicht zu verstehen noch einfach zu benutzen ist. Dagegen wäre eine Schnittstelle mit mehreren Funktionen und einfacheren Parametern zwar größer aber besser zu verstehen.

Wie eine „optimale“ Schnittstelle aussieht, läßt sich allgemein nicht endgültig festlegen. Beim Entwurf der Schnittstellen muss ein Kompromiss zwischen leichter Verständlichkeit mit einfacher Benutzung und dem Verbergen von Details mit einer möglichst kleinen Schnittstelle gefunden werden. Außerdem können noch weitere Faktoren eine Rolle spielen: Wenn das Programm später erweitert werden oder leicht zu modifizieren sein soll, muss dies bei den Schnittstellen berücksichtigt werden, indem z.B. zusätzliche Funktionen für spätere Erweiterungen angeboten werden.

1.1.1 Begriffe

Da im Weiteren einige Begriffe häufig benutzt werden, sollen diese hier zunächst erklärt werden:

Symbol Ein Symbol ist ein im Quelltext definiertes Element. Jedes Symbol ist von einen bestimmten Typ (z.B. Klasse, Funktion, Variable, etc.) und kann durch einem Namen(Bezeichner) und einen Sichtbarkeitsbereich eindeutig identifiziert werden.

Subsystem Teil eines Systems, das eine bestimmte Teilaufgabe übernimmt. In jedem Subsystem wird eine Menge von Symbolen definiert.

Referenz Eine Referenz zeigt die Verwendung eines Symbols an, das heißt jedes Auftreten eines Symbolnamens außerhalb der Symboldefinition ist eine Referenz des Symbols. Zum Beispiel ist ein Funktionsaufruf eine Referenz einer Funktion, eine Wertzuweisung eine Referenz einer Variablen.

Subsystemschnittstelle Teilmenge, der in einem Subsystem definierten Symbole, die in anderen Subsystemen bekanntgemacht wird, bzw. die Menge der Symbole, die in anderen Subsystemen referenziert werden.

1.1.2 Subsysteme und C/C++

Im Gegensatz zu Java, mit der Möglichkeit Packages anzulegen, ist es in C/C++ weitaus schwieriger Subsysteme auszudrücken. Ein direktes Konzept wird nicht angeboten, womit es vollständig in der Verantwortung des Entwicklers liegt, die Subsysteme und deren Schnittstellen darzustellen. In C allein bietet es sich nur an, den Quellcode durch Aufteilung auf verschiedene Dateien zu strukturieren. Da ein Subsystem meist zu groß ist, um in einer Quellcodedatei realisiert werden zu können, werden mehrere Dateien zu einem Subsystem zusammengefasst. Damit dies als Subsystem erkennbar ist, sollten diese Dateien in einem gemeinsamen Verzeichnis angeordnet werden oder durch entsprechende Benennung als zusammengehörig gekennzeichnet sein.

Für die Darstellung der Schnittstellen können Headerdateien verwendet werden. Mit Hilfe des Include-Mechanismus können Quelltextteile an verschiedenen Stellen eingefügt werden. So können Programmteile in anderen Subsystemen bekannt gemacht werden. Wie man die Schnittstellen darstellt, ist von den jeweiligen Anforderungen an das Design abhängig: Wenn man die Schnittstellen besonders hervorheben möchte, bietet es sich an, für jedes Subsystem eine Headerdatei anzulegen. Um die Schnittstellen noch weiter zu strukturieren, können auch mehrere Headerdateien für ein Subsystem benutzt werden.

In C++ besteht zusätzlich die Möglichkeit Programmteile in Namensbereichen (Namespaces) zusammenzufassen. Wenn für jedes Subsystem ein eigener Namespace verwendet wird, sind die Subsysteme als Komponenten zumindest erkennbar, ohne dass aber die Schnittstellen ablesbar sind. Um auch die Schnittstellen auszudrücken und von der Implementation zu trennen, ist es nötig weitere (Unter-)-Namensbereiche anzulegen, wobei für Schnittstelle und Implementation eigene Namespaces verwendet werden (siehe auch [Stroustrup]).

In C/C++ wird dem Entwickler also keine direkte Unterstützung bei der Bildung von Subsystemen gegeben. Eine Subsystem-Aufteilung ist dann nur eine zusätzliche Sichtweise des Entwicklers auf das Programm, der Compiler „sieht“ diese Subsysteme nicht und kann damit auch keinerlei Überprüfungen auf Korrektheit der Subsysteme vornehmen. Insbesondere muss der Entwickler allein darauf achten, die Schnittstellen korrekt auszudrücken, damit nicht unnötig Teile eines Subsystem öffentlich werden. Daher ist es besonders wichtig, die Subsysteme bei der Programmentwicklung von Anfang an mit zu berücksichtigen und vor allem die Schnittstellen eindeutig festzulegen. Nur dann können die Subsysteme und ihre Schnittstellen auch im Quellcode erkennbar dargestellt werden.

1.2 Ermittlung von Schnittstellen

Im Bereich des Reverse-Engineering stellt sich oft die Aufgabe, ein unbekanntes Programm ohne ausreichende Dokumentation zu untersuchen, die Architektur zu bestimmen und Weiterentwicklungen vorzunehmen. Dabei wird man zunächst versuchen, Subsysteme zu finden, um erste Aussagen über die Architektur machen zu können. Anschließend kann dann die weitere Arbeit auf je ein Subsystem begrenzt werden. Das Bestimmen der Subsysteme kann problematisch sein, wenn das vorliegende Programm bei der Entwicklung gar nicht oder nur ungenügend in Subsysteme eingeteilt wurde oder die Subsystemeinteilung im Quelltext nicht umgesetzt wurde. Dann wird im Quelltext keine Struktur erkennbar sein und es bleibt nur übrig, einzelne Quelltextteile anhand ihrer Funktionalität einem Subsystem zuzuordnen.

Bevor Änderungen an einem Subsystem gemacht werden können, müssen die Schnittstellen bekannt sein, damit Veränderungen an den Schnittstellen nicht zu Fehlern beim Zugriff von anderen Subsystemen führen. Also ergibt sich das Problem die Schnittstellen zu bestimmen. Dabei ist gemeint, die tatsächlich benutzten Schnittstellen (Used-Interface) zu finden, die von den bei der Programmentwicklung festgelegten Schnittstellen (wenn dabei überhaupt welche festgelegt wurden) durchaus abweichen können, da beim Entwurf der Subsysteme oftmals geplant wird, mehr Funktionalität zur Verfügung zu stellen, als später tatsächlich verwendet wird oder während der Entwicklung Änderungen vorgenommen wurden. Das Used-Interface ist also konkret die Menge der Symbole eines Subsystems, die in anderen Subsystemen refrenziert wird.

Eine weitere Aufgabe bei Architekturuntersuchengen wäre es, die Elemente eines Subsystems zu bestimmen, die auf andere Subsysteme zugreifen (Using-Interface), also die Symbole, die Symbole in anderen Subsystemen referenzieren. Dies soll hier aber nicht weiter betrachtet werden.

Auch wenn das zu untersuchende Programm bereits in Subsysteme eingeteilt ist, werden die Schnittstellen oftmals nicht direkt erkennbar sein. Diese müssen dann wie in [Thiel] beschrieben mühsam herausgearbeitet werden. Da diese Aufgabe bisher kaum von Tools unterstüzt wird, wäre ein Tool, das die Subsystemschnittstellen ermittelt, eine große Hilfe. Da dann die Schnittstellen sehr schnell bestimmt werden können, kann solch ein Tool auch eingesetzt werden, um verschiedene Varianten der Subsystemeinteilung des Programmes auszuprobieren, die damit gefundenen Schnittstellen zu vergleichen und somit die Subsystemeinteilung zu verbessern.

1.3 Bewertung von Schnittstellen

Ein Tool, dass die Möglichkeit bietet Subsystemschnittstellen schnell zu bestimmen, kann man benutzen, um für ein Programm nach einer „optimalen“ Subsystemeinteilung zu suchen. Dabei wird man verschiedene Subsystemeinteilungen ausprobieren und versuchen, deren Vor- und Nachteile zu erkennen. Außerdem kann man anhand der Schnittstellen Informationen über die Struktur der Subsysteme gewinnen und es lassen sich Rückschlüsse auf die Programmarchitektur ziehen. Damit man zu verlässlichen Aussagen gelangt, müssen die Schnittstellen und Subsystemeinteilungen bewertet werden können. Dazu müssen messbare Größen gefunden werden, auf die sich die Bewertung stützt. Nur dann ist es möglich, bei zwei verschiedenen Subsystemeinteilungen zu entscheiden, welches die „bessere“ ist.

An den gefundenen Schnittstellen und dem vorliegenden Quellcode müssen also einige Größen bestimmt werden. Dabei muss unterschieden werden, zwischen Größen die sich nur auf eine Schnittstelle und Größen die sich auf eine Subsystemeinteilung beziehen. Um die Schnittstellen der einzelnen Subsysteme zu bewerten und zu vergleichen, kommt zunächst die Schnittstellengröße (Anzahl der in der Schnittstelle enthaltenen Symbole) in Betracht, für die Bewertung von Subsystemeinteilungen als Ganzes muss eine entsprechende Größe gefunden werden, die alle Subsysteme zusammenfasst.

Außerdem kann man sich die Struktur der einzelnen Schnittstellen anschauen: treten mehr Funktionen als Klassen auf, sind Variablen enthalten oder andere Elemente überhaupt nicht? Wenn man untersucht, welche Art von Symbolen die Schnittstelle enthält und wie häufig diese auftreten, lassen sich Informationen über das Programm gewinnen. Besonders interessant sind dabei Häufungen von einer Art. Dann lassen sich zum Beispiel anhand der Häufigkeit von Klassen in der Schnittstelle, Aussagen über den Grad der Objektorientierung des Programmes treffen. Falls die Schnittstellen alle Arten von Symbolen in etwa gleich großer Zahl enthält kann dies ein Zeichen dafür sein, dass das entsprechende Subsystem gegenüber dem Rest des Programms schlecht abgegrenzt oder die Subsystemeinteilung noch nicht optimal ist.

1.4 Das XCTL-System als Beispiel

Das XCTL-System [XCTL] ist eine am Institut für Physik der HU-Berlin, in C++ entwickelte Software zur Steuerung von Meßanlagen zur Röntgenstrukturanalyse. Es wird im Rahmen eines Reverse-Engineering Projektes am Lehrstuhl für Softwaretechnik des Instituts für Informatik überarbeitet. Dabei wurden in [Schützler] die Subsysteme bestimmt, wobei sich herausstellte, dass die Schnittstellen nicht vollständig erkennbar sind. Daher wurden in [Thiel] erstmals die Schnittstellen dieser Subsysteme manuell herausgearbeitet, wobei der Wunsch nach einer automatisierten Bestimmung der Schnittstellen entstand, um weitere Untersuchungen in Bezug auf die Subsystemeinteilung vornehmen zu könnnen. Das im Folgenden beschriebene Tool wird daher speziell im Hinblick auf die Anwendung auf das XCTL-System entwickelt.

1.5 Verändern der Subsystemeinteilung

Wenn ein Tool zur Verfügung steht um die Schnittstellen zu bestimmen, können die Quelldateien “beliebig“ verschiedenen Subsystemen zugeordnet werden, um unterschiedliche Subsystemeinteilungen auszuprobieren. Dabei stellt sich die Frage, ob dies eventuell Auswirkungen auf die Lokalität und Sichtbarkeit und damit auf die eindeutige Bestimmbarkeit der in den Dateien definierten Symbole hat.

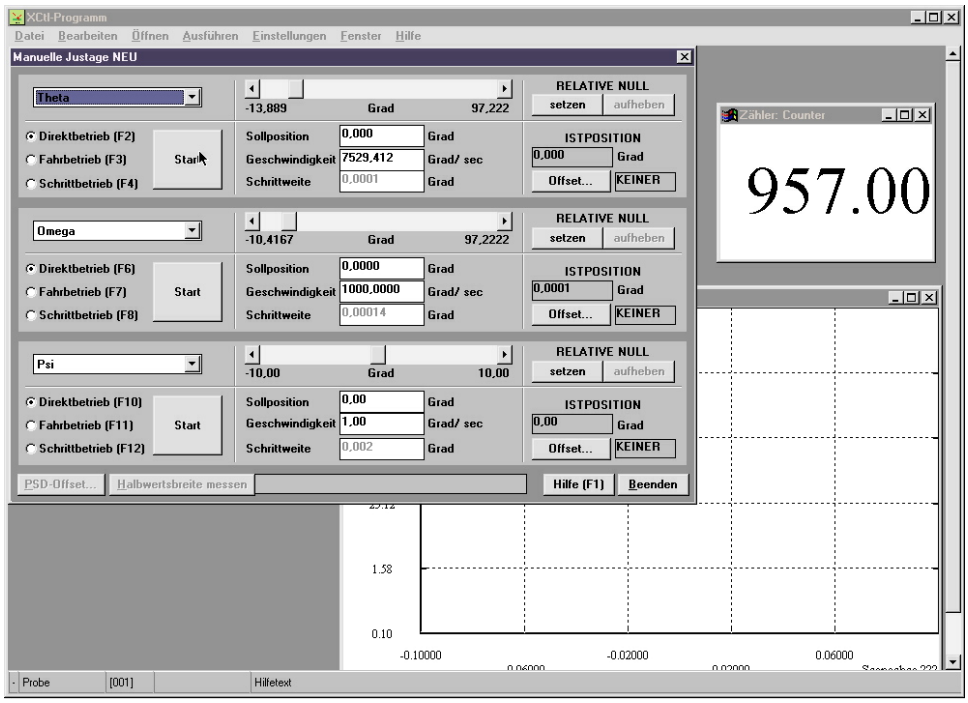


Abbildung 1.1: XCTL

Da, wie oben beschrieben, Subsysteme kein C++ -Sprachkonzept sind, ändert sich durch eine (veränderte) Subsystemeinteilung nichts an der Sichtbarkeit(für den Compiler) von Symbolen und damit auch nichts an der Compilierbarkeit.

Trotzdem ist es möglich ein Symbol als *subsystemlokal* zu bezeichnen: Wenn die Sichtbarkeit eines Symbols nicht weiter eingeschränkt ist (funktionslokal, klassenlokal), so erstreckt sich die Sichtbarkeit des Symbols zunächst auf die Datei in der es definiert wurde. Diese Sichtbarkeit kann auf weitere Dateien ausgedehnt werden, wenn die Definitionsdatei in anderen Dateien inkludiert wird oder zusätzliche Deklarationen (`extern` bei Variablen, Prototypen von Funktionen) das Symbol in anderen Dateien bekannt machen. Somit kann ein Symbol als *subsystemlokal* bezeichnet werden, genau dann wenn alle Dateien, in denen es sichtbar ist, zum gleichen Subsystem gehören. Das bedeutet aber, dass dieses Symbol dann nicht in Dateien anderer Subsysteme sichtbar ist und somit dort nicht verwendet werden kann, womit das Symbol nicht zur (tatsächlich benutzten) Subsystemschnittstelle gehören kann. Anders herum sind alle Symbole die nicht zur Schnittstelle gehören natürlich subsystemlokal.

Weiterhin kann natürlich eine Veränderung der Subsystemeinteilung bedeuten, dass Subsystemlokalität entsteht oder verloren geht. Wenn z.B. eine Datei, in der ein, bisher subsystemlokales, Symbol sichtbar ist, zu einem anderen Subsystem kommt, geht Subsystemlokalität verloren und das Symbol taucht in der Schnittstelle auf.

Für ein Tool, welches Subsystemschnittstellen berechnet, ist es wichtig alle Symbole eindeutig unterscheiden zu können. Es ist möglich, zwei Symbole mit gleichen Namen zu definieren, wenn diese in verschiedenen Übersetzungseinheiten stehen. Solange alle Dateien eine gemeinsame Übersetzungeinheit bilden, dürfen zwei Symbole, die in mehr als einer Datei sichtbar sind, nie den gleichen Namen haben damit das Programm korrekt compiliert werden kann. Damit bleibt aber auch bei jeder beliebigen Subsystemeinteilung die Eindeutigkeit bestehen. Probleme können ent-

stehen, wenn in verschiedenen Dateien, die nicht gemeinsam compiliert werden, Symbole mit gleichen Namen definiert werden. Ein Tool, das die Übersetzungseinheiten nicht erkennt, wird möglicherweise die Symbole nicht unterscheiden können und somit ein Symbol fälschlicherweise in die Subsystemschnittstelle aufnehmen. Dies ist aber unabhängig von der Subsystemeinteilung, da wie oben beschrieben, die Subsystemeinteilung auf der Quellcodeebene nicht sichtbar ist. Das „Problem“ tritt dann also bei jeder Einteilung auf.

Kapitel 2

Toolentwicklung

Im vorangegangenen Kapitel wurde die Notwendigkeit der Bestimmung von Subsystemschnittstellen erläutert, und der Wunsch geäußert, ein Tool zu entwickeln, welches diese Aufgabe übernimmt. Im Folgenden soll nun solch ein Tool entwickelt werden.

2.1 Grundproblem

Wie kann ein solches Tool arbeiten? Das Grundproblem besteht darin festzustellen, ob ein Symbol (Klasse, Funktion, etc.) außerhalb des Subsystems in dem es definiert wurde verwendet(referenziert) wird, wenn dies der Fall ist, gehört dieses Symbol zur Subsystemschnittstelle. Um dieses Problem zu lösen, müssen zunächst alle Symbole erkannt und einem Subsystem zugeordnet werden. Anschließend muss für jedes Symbol herausgefunden werden, ob es in einem anderen Subsystem verwendet wird.

Das Erkennen der Symbole und deren Verwendung, d.h das Bestimmen der Referenzen, erfordert ein Parsen des Quelltextes und das Anlegen einer Symboltabelle. Diese Arbeit wird auch von Compilern und anderen Sourcecode-Analyse-Tools ausgeführt. Daher sollte geprüft werden, ob ein vorhandenes Tool, welches die Symbol- und Referenzinformationen ermittelt und zur Verfügung stellt, genutzt bzw. erweitert werden kann.

2.2 Vorhandene Tools

Es existiert bereits eine große Vielzahl von Sourcecode-Analyse-Tools, wovon hier zunächst zwei betrachtet werden sollen im Hinblick auf die Möglichkeit, Subsystemschnittstellen zu finden, das Tool um eine solche Funktionalität zu erweitern, bzw. als Hilfe zur Ermittlung von Symbol- und Referenzinformationen zu nutzen. Außerdem wird überprüft, inwieweit sich das XCTL-System mit dem jeweiligen Tool bearbeiten lässt.

2.2.1 objectiF

Das Programm objectiF von Microtool (siehe [objectiF]) ist ein Entwicklungswerkzeug, welches einen großen Teil des Softwareentwicklungsprozesses unterstützt. Ausgehend von Anwendungsfällen können Use-Case-Diagramme entworfen und zugehörige Beschreibungen angefertigt werden, woraus direkt mögliche Klassen erkannt und angelegt werden können. Desweiteren stellen Aktivitätsdiagramme eine weitere Spezifizierungsmöglichkeit dar. ObjectiF kann dann aus den Klassen-, Use-Case-

und Aktivitätsdiagrammen Sourcecode erzeugen. Bei Änderungen wird die Konsistenz zwischen Sourcecode und Objektmodell von objectiF stets sichergestellt. Über eine Reverse-Engineering-Komponente ist es möglich bereits vorhandenen Quellcode einzulesen und zu bearbeiten.

Hinter dem Menupunkt *Referenzen* verbirgt sich ein Cross-Referencer. Damit kann zu jedem Symbol eines Systems ermittelt werden wo dieses verwendet wird. Nach dem Aufrufen des Menupunktes werden die gefundenen Referenzen in einem Fenster angezeigt. Mit dieser Funktion ist eine Voraussetzung erfüllt, objectiF als Hilfe bei der Ermittlung von Subsystemschnittstellen einzusetzen, wenn die Möglichkeit besteht die Referenzinformationen mit einem eigenen Tool, oder einer Erweiterung von objectiF, zu nutzen.

Es gibt zwei Möglichkeiten objectiF zu erweitern: So genannte *Code-Skripte* ermöglichen eine individuelle Codegenerierung, über *Action-Skripte* kann die Programmfunktionalität von objectiF erweitert werden, wobei die damit realisierten Funktionen im entsprechenden Kontextmenü von objectiF angeboten werden. Diese Skripte müssen in VisualBasic geschrieben werden. Bei beiden Skriptvarianten können Informationen, die objectiF über ein System hat, über ein COM-Interface ausgelesen werden. Das COM-Interface ermöglicht einen direkten Zugriff auf die Informationen, so kann z.B nach den Membern einer bestimmten Klasse gefragt werden, oder auch nach dem Quelltext einer Funktion, etc.

Für die Ermittlung der Subsystemschnittstellen böte es sich an ein *Action-Skript* zu schreiben, welches dann über das Kontextmenu aufgerufen werden kann. Die gefundenen Schnittstellen könnten in einem Fenster als Liste ausgegeben werden, welche auch ausgedruckt oder gespeichert werden könnte.

Der Versuch das XCTL-System(Version vom 20.01.2002) in objectiF über die Reverse-Engineering-Komponente einzulesen gelang nicht ganz problemlos. Zur Vorbereitung mussten alle im XCTL-Quellcode definierten Makros (`#define`) in eine Konfigurationsdatei aufgenommen werden, damit diese von objectiF erkannt werden. Desweiteren führten bedingte Codeabschnitte, wie z.B.:

```
#ifdef GermanVersion
char strFailure[] = "Fehler";
...
#else
char strFailure[]   = "Failure";
...
#endif
```

zu Problemen, wenn sowohl im if- als auch im else-Abschnitt die gleichen Bezeichner deklariert wurden. In solchen Fällen musste entweder ein Abschnitt auskommentiert, oder das Parsen eines Abschnittes mittels spezieller Makros ausgeschaltet werden.

Einige Eigenarten des XCTL-Quelltextes werden von objectiF nicht verarbeitet. So führt die C-Syntax bei `struct-` Deklarationen

```
typedef struct {
...
} mystruct;
```

zu dem Problem, dass objectiF einen namenlosen (nur mit einer Nummer versehen) struct-Typen anlegt und zusätzlich einen Typedef `mystruct`. Diese Deklarationen müssen also in die C++-Syntax

```
struct mystruct {...};
```

überführt werden.

Desweiteren darf bei `enum`-Deklarationen nach dem letzten Element kein Komma mehr stehen (wie z.B. bei `enum readoutMode` in RADICOHW.H) und bei Konstanten muss stets der Typ angegeben sein (z.B. `static const ACK=0x06` in RADICOHW.CPP).

Diese Anpassungen könnten größtenteils dauerhaft im XCTL-System vorgenommen werden, so dass bei einem erneuten Einlesen in objectiF der Aufwand geringer wäre. Änderungen am XCTL, z.B. neue Makros oder bedingte Codeabschnitte, müssten aber jeweils besonders berücksichtigt werden.

Wie könnten nun die Subsystemschnittstellen ermittelt werden? ObjectiF ermöglicht es ein System in Packages einzuteilen. Die Idee ist nun, jedes Subsystem als ein Package darzustellen. Dann könnten nacheinander für jedes Element eines Packages die Referenzen ermittelt werden. Wenn dabei Referenzen von außerhalb des Packages auftreten muss das Element in das Interface aufgenommen werden. Allerdings ist die Einteilung des Systems in Packages aufwendig und nur in Handarbeit möglich, womit insgesamt nicht viel gewonnen wäre, schließlich soll ja durch die Automatisierung Aufwand gespart werden. Auch das Ausprobieren verschiedener Subsystemeinteilungen wäre entsprechend aufwendig, da jedesmal die Packages neu sortiert werden müssten.

Inwieweit das Anlegen von Packages automatisert werden kann wurde nicht weiter untersucht, da noch ein weitaus größeres Problem besteht: Es wurde keine Möglichkeit gefunden die Referenzen über das COM-Interface zu erfragen. Mit größerem Aufwand wäre es zwar möglich diese Funktion eigenständig nachzuprogrammieren, damit würde aber die Verwendung von objectiF keine wesentlichen Vorteile mehr bieten.

2.2.2 SNiFF+

Als Alternative zu objectiF bietet sich SNiFF+ von Windriver an, welches hier in der Version 4.0.1 betrachtet werden soll. SNiFF+ ist eine Entwicklungsumgebung, die neben den „klassischen“ Funktionen einer IDE erweiterte Darstellungsmöglichkeiten eines Systems anbietet. Insbesondere können Klassendiagramme angezeigt werden, ein Symbolbrowser ermöglicht die einfache Navigation im gesamten System und es existiert ein Cross-Referencer der eine Funktion zur Darstellung der Referenzen eines Symbols zur Verfügung stellt.

Der Funkionsumfang von SNiFF+ kann nicht direkt erweitert werden, stattdessen gibt es aber die Möglichkeit mit eigenen Java-Programmen Informationen von SNiFF+ abzufragen. Sämtliche Informationen die SNiFF+ über ein System besitzt werden über eine Java-Schnittstelle, die sogenannte SniffAPI, zur Verfügung gestellt. Eine Klasse `SniffController` stellt dabei die Verbindung zum SNiFF+-Programm her und startet die Informationsübertragung an das Java-Programm. Von verschiedenen so genannten Handlerklasssen (`ProjectHandler, FileHandler, SymbolHandler, ReferenceHandler, PreprocessorHandler` und `ScopeHandler`) werden die Informationen empfangen, indem dort entsprechende Methoden (z.B. `handleClass(ClassData data)` aus der Klasse `SymbolHandler`) aufgerufen werden. Die Parameter dieser Funktionen können zusätzliche Informationen enthalten. Im eigenen Programm werden Ableitungen der Handlerklassen benutzt und die Methoden überschrieben, um die Informationen auswerten zu können.

SNiFF+ verwendet zur Identifizierung der einzelnen Symbole Ids, welche auch als Schlüssel benutzt werden können, wenn die Symbolinformationen in einer eigenen Datenbank gespeichert werden. Referenzinformationen erhält man von SNiFF+ als Paar von Ids in der Form: [ReferencingId, ReferencedId].

Alle Informationen zu einem System werden von SNiFF+ in einem Durchlauf übertragen, die relevanten Informationen müssen also herausgefiltert und evtl. zwi-

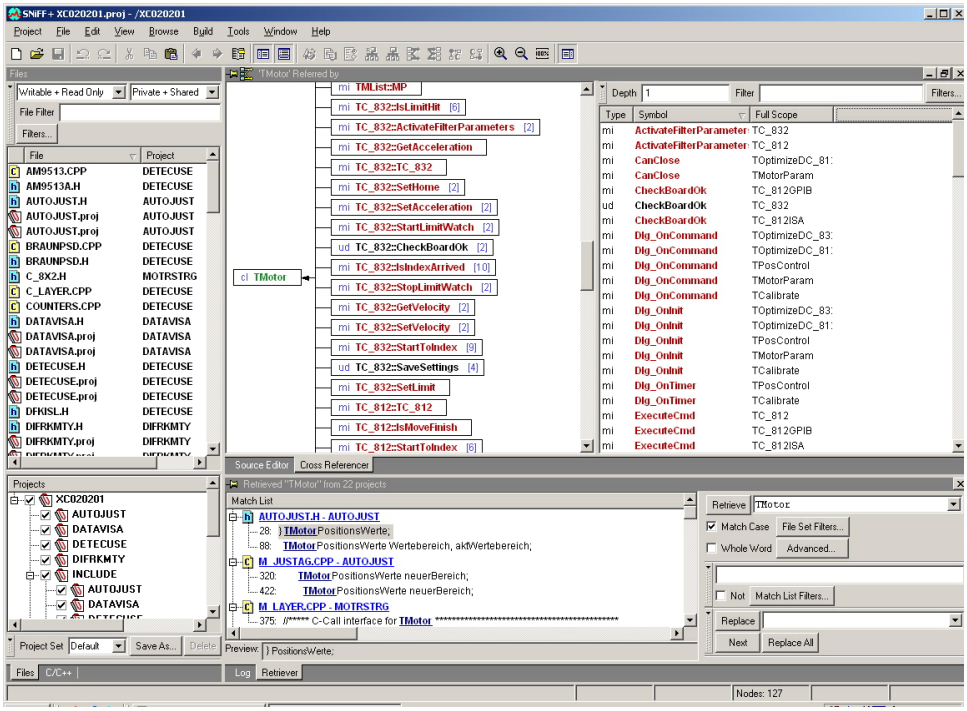


Abbildung 2.1: SNiFF+ mit geöffnetem XCTL-Projekt

schengespeichert werden. Ein direktes Abfragen von einzelnen Informationen, wie über das COM-Interface von objectiF, ist nicht möglich.

Das Einlesen der XCTL-Quellen in SNiFF+ erfolgte völlig problemlos. Der „Import-Wizard“ von SNiFF+ erkennt nach dem Angeben des Verzeichnisses alle Quelldateien, liest diese ein, erzeugt eine SNiFF+ -Projektdatei und legt ein SNiFF+ -Working-Environment an. Bei Änderungen am XCTL-System, die nicht innerhalb von SNiFF+ erfolgen, muss der Quellcode allerdings immer wieder neu eingelesen, das heißt ein neues Projekt angelegt werden.

Die Besonderheiten des XCTL-Quellcodes stellen für SNiFF+ kein Problem dar. Wenn Structs mit C-Syntax definiert werden, legt SNiFF+ einen Struct und einen Typedef mit dem Namen an. Über die SniffAPI erhält man allerdings einen namenlosen Struct und einen Typedef mit dem Namen. Dies stellt aber kein Problem dar, da dies keinen Einfluß hat für das Bestimmen von Subsystemschnittstellen, solch ein Struct wird in der Ausgabe der Schnittstellen dann auch namenlos und mit zugehörigen Typedef auftreten.

Andererseits unterstützt die Sniff-API nicht den vollen C++-Sprachumfang. So werden weder Templates noch Namespaces behandelt. Da beides aber im XCTL-System nicht verwendet wird, ist dies keine wirkliche Einschränkung. Problematischer ist da schon die Tatsache, dass einige Schlüsselwörter (`auto`, `explicit`, `extern`, `mutable`, `register`, `volatile`) nicht unterstützt werden, wovon jedoch nur `extern` für die Schnittstellenbestimmung von Bedeutung ist (bei Variablen, die mittels „`extern`“ in verschiedenen Dateien verwendet werden). Für die Behandlung von `extern` müsste daher das eigene Tool eine Lösung finden.

Im Gegensatz zu den Action-Skripten von objectiF ist die Möglichkeit die Informationen hier mit einem eigenen Java-Programm auswerten zu können wesentlich flexibler, da die Informationen mit weiteren, nicht in SNiFF+ enthaltenen, Informationen kombiniert werden können. So ist es für das Ermitteln der Subsystemschnittstellen nicht notwendig (und auch nicht möglich) die Subsysteme in SNiFF+

darzustellen. Es ist ausreichend, wenn das Java-Programm die Subsystemeinteilung kennt und jedes Symbol einem Subsystem zuordnen kann. Ein weiterer Vorteil ist auch, dass das Java-Programm ohne SNiFF+ compiliert werden kann, also bei der Entwicklung nicht immer SNiFF+ vorhanden sein muss.

Das Ermitteln von Subsystemschnittstellen könnte von einem Java-Programm mit Hilfe von SNiFF+ folgendermaßen realisiert werden:

- gewünschte Subsystemeinteilung einlesen
- alle Symbol- und Referenzinformation von SNiFF+ übertragen und evtl. zwischenspeichern
- jedes Symbol einem Subsystem zuordnen
- jede Referenz überprüfen, ob verschiedene Subsysteme beteiligt, wenn ja, referenziertes Symbol in Schnittstelle aufnehmen
- Schnittstellen ausgeben

Da SNiFF+ alle Voraussetzungen zur Benutzung als Hilfsmittel bei der Ermittlung von Subsystemschnittstellen erfüllt und einfacher zu verwenden ist als objectiF, ist damit entschieden SNiFF+ zu verwenden und ein entsprechendes Java-Programm zu entwickeln.

2.3 Anforderungsanalyse

In diesem Abschnitt werden die Anforderungen an das Tool und erste Ideen zur Arbeitsweise allgemein beschrieben. Im nächsten Abschnitt folgt dann das Pflichtenheft, in dem die Anforderungen konkret festgelegt werden. Das Tool wird ab jetzt **InterfaceFinder** genannt.

2.3.1 Allgemeine Anforderungen

Aufgabe

Aufgabe des Tools **InterfaceFinder** ist es die Subsystemschnittstellen eines Programmsystemes zu bestimmen. Das Programmsystem liegt im Quelltext vor und ist durch Anordnen der Quelldateien in Unterverzeichnisse in Subsysteme eingeteilt. Der Nutzer sollte die Möglichkeit haben, die Subsystemeinteilung leicht zu verändern, damit verschiedene Subsystemeinteilungen ausprobiert werden können.

Die ermittelten Schnittstellen müssen in einer angemessenen Form ausgegeben werden und sollten gespeichert werden können.

Subsystemeinteilungen

Jedes auftretende Symbol muss eindeutig einem Subsystem zugeordnet werden können. Diese Zuordnung muss vom Nutzer bestimmt und dem Tool bekannt gemacht werden. Da es für den Nutzer zu aufwändig wäre jedes Symbol einzeln zuzuordnen, werden jeweils ganze Quelldateien, und damit die darin enthaltenen Symbole, einem Subsystem zugeordnet. Dabei gilt, dass der Ort der *Definition* eines Symbols über die Zuordnung zu einem Subsystem entscheidet, zusätzliche *Deklarationen*, möglicherweise in verschiedenen Dateien, werden nicht beachtet. Da jedes Symbol genau einmal definiert werden muss, ist so eine eindeutige Zuordnung möglich.

Solch eine Subsystemeinteilung auf Dateibasis ist sehr flexibel, da sowohl Header- als auch .cpp-Dateien gleich behandelt werden und normalerweise kein Eingriff in

den Quelltext erforderlich ist. Nur falls Symbole einer Datei zu verschiedenen Subsystemen zugeordnet werden sollen, muss die Datei aufgeteilt und der entsprechende Code-Abschnitt verschoben werden. Für den Nutzer ist die Einteilung einfach vorzunehmen und zu verändern, z.B durch Anordnen der Quelldateien in Unterverzeichnissen.

Eine Zuordnung von Dateien zu Subsystemen muss nicht notwendigerweise physisch durch Anordnung der Quelldateien in Unterverzeichnissen vorhanden sein. Es reicht aus die Zuordnung im Tool logisch vorzunehmen und in einer passenden Datenstrukur zu halten. Dazu muss ein Dialog angeboten werden, in dem Subsysteme angelegt und Dateien zugeordnet werden können. Die Zuordnung gilt dann nur solange das Tool läuft und hat keinerlei Auswirkungen auf die wirkliche Dateistruktur. Damit können dann verschiedene Einteilungen mit geringem Aufwand ausprobiert werden.

Referenzen

Die Referenzinformationen werden von SNiFF+ in Form von Paaren von Symbol-IDs [referencingID, referencedID] geliefert. Das referenzierende Symbol ist dabei die umgebende Funktion, Klasse oder Datei. Eine Referenz wird von SNiFF+ in folgenden Fällen erzeugt:

Typen Klassen, Struct, Unions und Enums definieren eigene Typen. Eine Referenz wird generiert, wenn der Typname direkt genannt wird, z.B. beim Anlegen eines Objektes mit `new`, oder eine Variable des Typs verwendet wird. In allen Zeilen des folgenden Beispiels werden Referenzen auf die Klasse A erzeugt:

```
1   A * a, * b;
2   a = new A ();
3   b = a;
4   a->doSomething();
5   print(b);
```

Funktionen Funktionsreferenzen werden beim Aufruf einer Funktion (auch Memberfunktion) generiert, im obigen Beispiel also in den Zeilen 4 und 5.

Variable Referenzen von Variablen werden beim Lesen und Schreiben (Wertzuweisung) von Variablen (auch Membervariablen) erstellt. Im folgenden Beispiel werden in den Zeilen 2 bis 4 Referenzen auf die Variable i erzeugt.

```
1   int i;
2   i = 1;
3   i++;
4   print(i);
```

Typedef Typedefs definieren einen neuen Namen für einen Typ. Eine Referenz wird generiert wenn der Typedef-Name verwendet wird.

Für Makros(`#define`'s) erzeugt SNiFF+ keine Referenzen. Die Verwendung der Makros muss also anders bestimmt werden. SNiFF+ erzeugt sogenannte „Undefined References“, wenn das referenzierte Symbol nicht bekannt ist.

Datenhaltung

Es wird notwendig sein, verschiedene Informationen über den Quelltext und die Symbole zwischenzuspeichern. Dafür müssen entsprechende Datenstrukturen angelegt werden. Datenstrukturen werden benötigt für die Symbole (Namen und die

Art des Symbols, zugehöriges Subsystem, Verwendung), die Dateien des Systems, die Subsystemeinteilung und die ermittelten Schnittstellen. Alle Informationen über Dateien, Symbole und Referenzen werden von SNiFF+ geliefert. Da SNiFF+ sämtliche Informationen über das System auf einmal überträgt, müssen die relevanten Informationen herausgefiltert und so zwischengespeichert werden, dass ein effektiver Zugriff möglich ist. Dabei können die von SNiFF+ mitgelieferten Ids als Schlüssel benutzt werden.

Ausgabe

Für die Form und den Umfang der Ausgabe sind verschiedenste Möglichkeiten denkbar, wobei im Allgemeinen für jedes Subsystem eine Liste von Symbolen(Bezeichner) auszugeben ist. Solche Listen könnten nach verschiedenen Kriterien sortiert oder um ergänzenden Informationen erweitert werden. Die Ausgabe sollte sowohl auf den Bildschirm, als auch in eine Datei möglich sein. Außerdem könnten Statistikinformationen (z.B. Größe der Schnittstellen) ausgegeben werden.

Nutzersicht/Oberfläche

Das Tool sollte für den Nutzer durch eine grafische Oberfläche zu bedienen sein. Es müssen Dialoge angeboten werden, mit denen das zu untersuchende System bekannt gemacht werden kann (Pfad der Quelldateien oder SNiFF+ -Projektdatei) und eine Subsystemeinteilung vorgenommen oder bearbeitet werden kann. Die Ausgabe der Schnittstellen sollten in einem Fenster erfolgen, mit Möglichkeit der Bearbeitung (sortieren, kopieren).

2.3.2 Pflichtenheft

Das Pflichtenheft mit den konkreten Anforderungen an das zu entwickelnde Tool befindet sich in Anhang A auf Seite 61.

2.3.3 UseCase-Diagramm

Ausgehend von den Anforderungen des Pflichtenheftes lassen sich für **InterfaceFinder** folgende Anwendungsfälle bestimmen, die im UseCase-Diagramm in Abbildung 2.2 gezeigt werden.

Subsystemschnittstellen ermitteln Dieser UseCase umfasst neben der eigentlichen Berechnung der Schnittstellen auch die Verbindung mit SNiFF+, die Informationsübertragung sowie das Anlegen und Verwalten der Datenstrukturen.

Projekt öffnen Bereitstellen von Dialogen mit denen der Nutzer die SNiFF+ -Projektdatei auswählen und das SNiFF+ -WorkingEnvironment festlegen kann.

Subsysteme zuordnen Möglichkeit für den Nutzer eine Subsystemeinteilung vorzunehmen/ zu verändern

Schnittstellen ausgeben Ausgabe der ermittelten Schnittstellen.

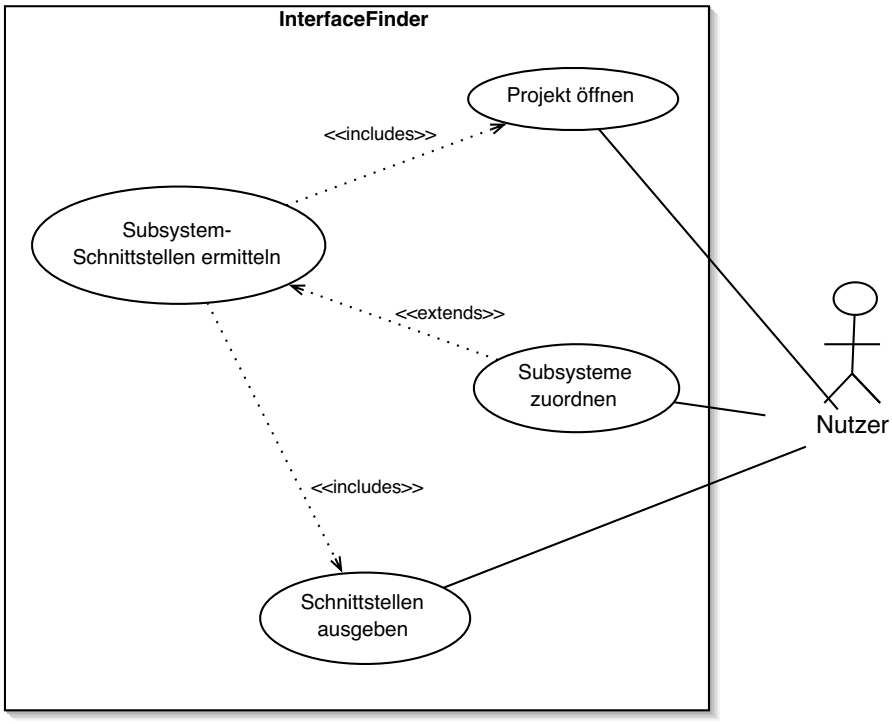


Abbildung 2.2: Use-Case Diagramm

2.4 Architektur

In diesem Abschnitt soll die Architektur von **InterfaceFinder** beschrieben werden. Die Architektur eines Systems ist nach [Bass] die Struktur (bzw. die Strukturen) des Systems. Zur Dokumentation der Architektur müssen die verschiedenen Strukturen in Form von Sichten (Views) und die Beziehungen zwischen den Sichten beschrieben werden. Mögliche Sichten können z.B. die Struktur in Form von Implementationseinheiten (Module, Klassen), die Struktur zur Laufzeit (Prozesse, Threads) oder die Systemumgebung zeigen.

Im Falle von **InterfaceFinder** ist vor allem die Zerlegung in Komponenten, die verschiedenen Teilfunktionen übernehmen, von Bedeutung. Die weitere Verfeinerung dieser Zerlegung führt zur Klassenstruktur des Programms.

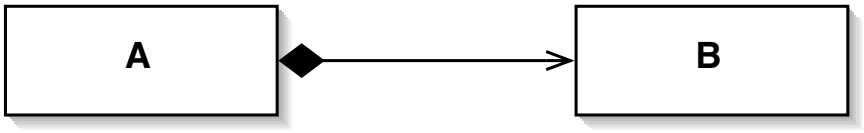
2.4.1 Diagrammsymbolik

Alle folgenden Diagramme werden in UML-Notation dargestellt.

Im Klassendiagramm werden dabei zum Teil auch Klassen aus anderen Komponenten des Systems, der SNiFF+ -API oder aus Standard-Java-Packages gezeigt. Solche Klassen werden grau dargestellt. Um Übersichtlichkeit zu bewahren, werden die Beziehungen zwischen den Klassen nur gezeigt, soweit es für des Verständnis hilfreich ist, die Rollennamen werden meist weggelassen.

Die Beziehungen zwischen den Klassen werden als Aggregation oder Composition dargestellt. Nach dem UML-Standard bedeutet Composition, dass das „Gesamt“-Objekt für die Beseitigung seiner Teile verantwortlich ist. Da in Java die Beseitigung von Objekten durch die garbage-collection durchgeführt wird, kann dies hier so verstanden werden, dass mit dem Verlust der letzten Referenz auf das Gesamtobjekt auch keine Referenzen mehr auf die Teile vorhanden sind. Desweiteren

kann bei **InterfaceFinder** die Bedeutung der Composition so interpretiert werden, dass das Gesamtobjekt seine Teile anlegt, also:



auch gelesen werden kann als A erzeugt B. Die Pfeile geben an, in welcher Richtung Methoden gerufen werden können, im Beispiel kann die Klasse A also Methoden von B aufrufen.

2.4.2 Komponentenzerlegung

Aus den UseCases und den Anforderungen im Pflichtenheft lassen sich Teilaufgaben für verschiedene Komponenten ableiten.

Da der UseCase *Subsystemschnittstellen ermitteln* mehr als das eigentliche Berechnen der Schnittstellen umfaßt, werden daraus mehrere Komponenten abgeleitet: *Schnittstellen-Ermittlung*, *Datenhaltung*, *SNiFF-Client*. Außerdem beinhaltet dieser UseCase Teile der *Programmsteuerung*. Die UseCases *Projekt öffnen* und *Subsysteme zuordnen* werden durch verschiedene Dialoge realisiert, sie bilden keine eigenen Komponenten sondern sind Teil der *Benutzeroberfläche*, der *Programmsteuerung* und der *Schnittstellenermittlung* Der UseCase *Schnittstellen ausgeben* wird durch eine eigene Komponente realisert, steht aber in Verbindung mit der *Benutzeroberfläche*.

Daraus ergibt sich der in Abbildung 2.3 dargestellte Aufbau des Systems aus Komponenten.

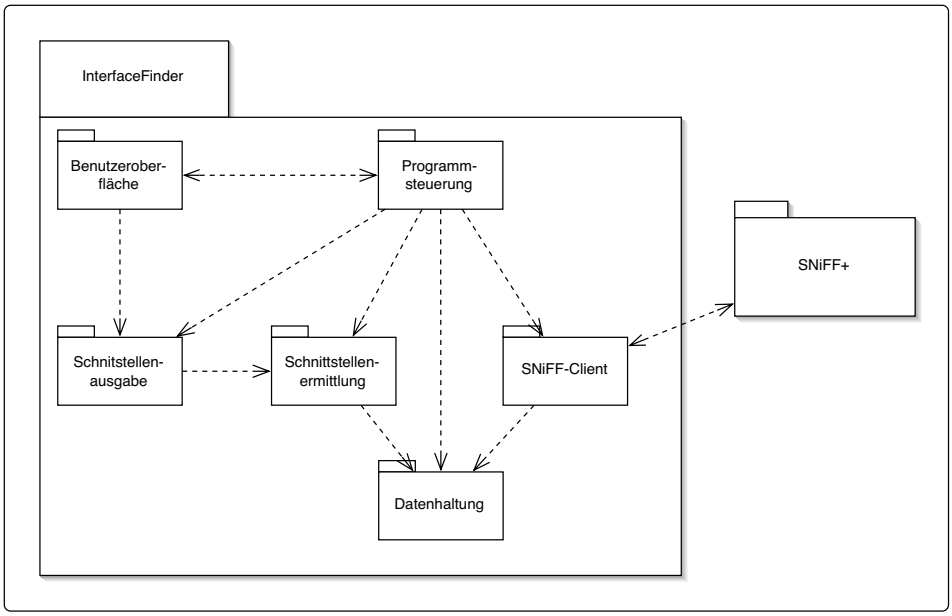


Abbildung 2.3: Zerlegung in Komponenten. Die Pfeile zeigen den Zugriff (Funktionsaufrufe) auf Komponenten an.

Im Folgenden werden nun die einzelnen Komponenten genauer spezifiziert, wobei für jede Komponente folgendes Schema verwendet wird: Zunächst wird die Aufgabe

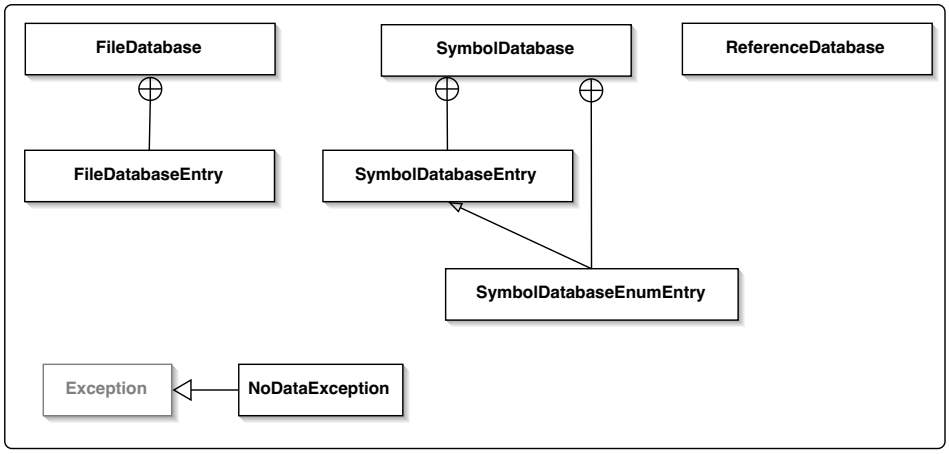
der Komponente kurz beschrieben. Dann folgt ein Diagramm, welches die Klassen dieser Komponente zeigt. Im Anschluß werden die einzelnen Klassen kurz beschrieben.

Eine Übersicht über die Beziehungen zwischen den Klassen verschiedener Komponenten wird im folgenden Abschnitt in Abbildung 2.4 auf Seite 20 gezeigt.

Datenhaltung

Diese Komponente ist verantwortlich für die Zwischenspeicherung der über das zu untersuchende System vorliegenden Informationen. Die Daten werden dabei von der Komponente *SNiFF-Client* übernommen und anderen Komponenten über entsprechende Methoden zur Verfügung gestellt.

Klassen der Datenhaltung:



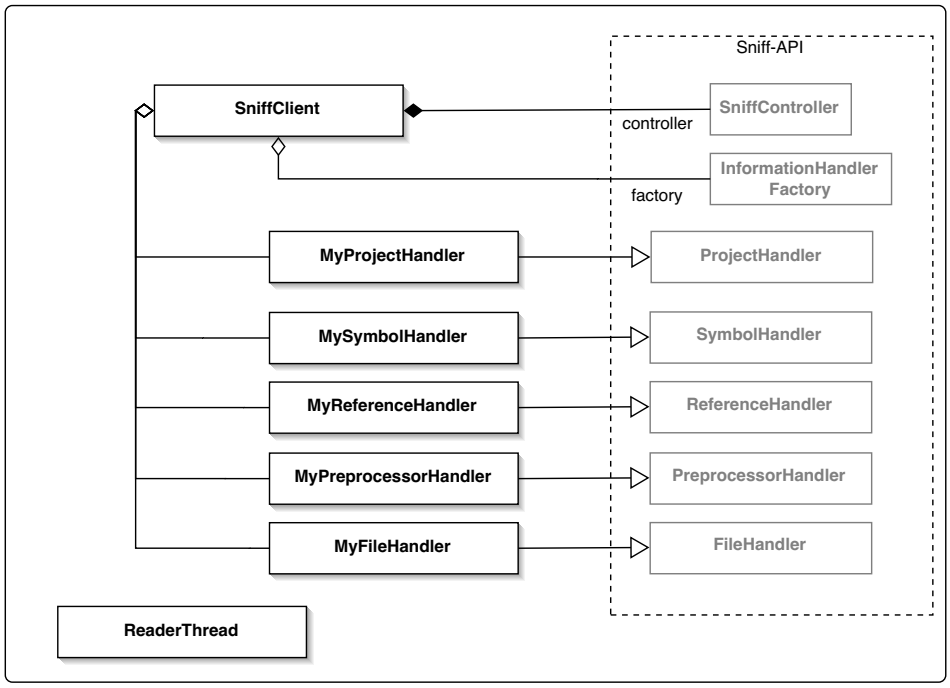
Bei der Datenhaltung werden unterschiedliche Anforderungen (Art und Umfang der Informationen, Zugriffsmöglichkeiten) in Bezug auf Dateien, Symbole und Referenzen gestellt. Daher wird die Datenhaltung durch die drei verschiedenen Klassen `FileDatabase`, `SymbolDatabase` und `ReferenceDatabase` realisiert. Die Datensätze der Datei- und Symboldatenbank sind so umfangreich, dass zur Aufnahme der Informationen die Klassen `FileDatabaseEntry` bzw. `SymbolDatabaseEntry` benutzt werden. Zur Speicherung von `Enum`-Symbolen dient die Klasse `SymbolDatabaseEnumEntry`. In der Referenzdatenbank werden die Informationen in einem `Vector` gespeichert.

Die `NoDataException` dient zum Anzeigen der Situation, dass ein angeforderter Datensatz nicht verfügbar ist. Dies wird sowohl in der Symbol- als auch in der Dateidatenbank benutzt.

SNiFF-Client

Die Aufgaben dieser Komponente umfassen den Verbindungsaufbau zu SNiFF+, die Informationübertragung und den Empfang der Informationen. Der Verbindungsaufbau und der Start der Informationsübertragung werden dabei von der Komponente *Programmsteuerung* ausgelöst. Falls das SNiFF+ -Programm noch nicht läuft, wird versucht, es beim Verbindungsaufbau zu starten. Die empfangenen Daten werden der *Datenhaltung* zur Zwischenspeicherung übergeben.

Klassen:

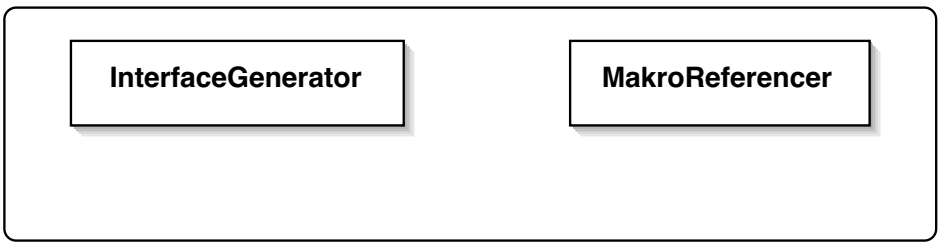


Die Klasse `SniffClient` stellt die Verbindung zum Sniff-Programm her und startet die Datenübertragung. Die verschiedenen `Handler`-Klassen dienen zum Empfang der entsprechenden Informationen und Weitergabe der Daten an die *Datenhaltung*. Die Klasse `ReaderThread` ist eine Hilfsklasse, die das Blockieren des SNiFF+ - Prozesses verhindert.

Schnittstellenermittlung

Mit dieser Komponente werden aus den Informationen der *Datenhaltung* die Subsystemschnittstellen ermittelt und den Ausgabekomponenten zur Verfügung gestellt. Desweiteren ist die *Schnittstellenermittlung* für das Bestimmen der Makrobenutzung zuständig, da SNiFF+ zwar die Definition von Makros (`#define`-Anweisungen) übermittelt, aber nicht die Verwendung dieser Makros.

Klassen:



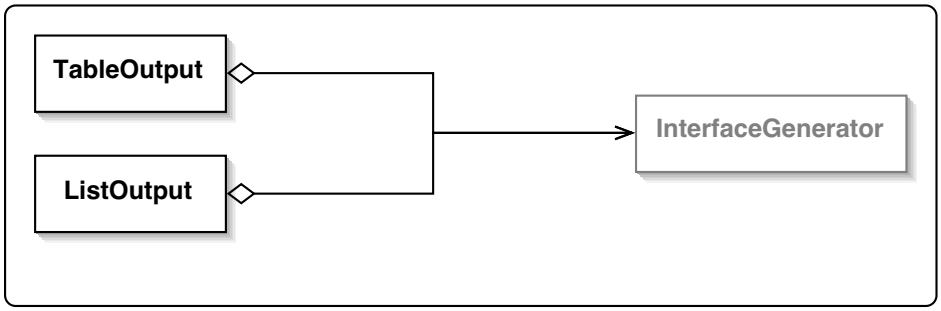
Die Klasse `InterfaceGenerator` ermittelt die Schnittstellen. Mit `MakroReferencer` wird die Benutzung der Makros ermittelt, indem der Quelltext nach den definierten Makros durchsucht wird.

Schnittstellenausgabe

Die *Schnittstellenausgabe* besteht aus verschiedenen Ausgabemodulen (Klassen), die die Schnittstellen in einer passenden Form (z.B. Tabelle oder Liste) ausgeben. Die notwendigen Informationen erhalten die Ausgabemodule von der Komponente *Schnittstellenermittlung*.

An dieser Stelle kann **InterfaceFinder** durch das Hinzufügen von weiteren Ausgabemodulen erweitert und so die Ausgabe an spezielle Erfordernisse angepasst werden.

Klassen:

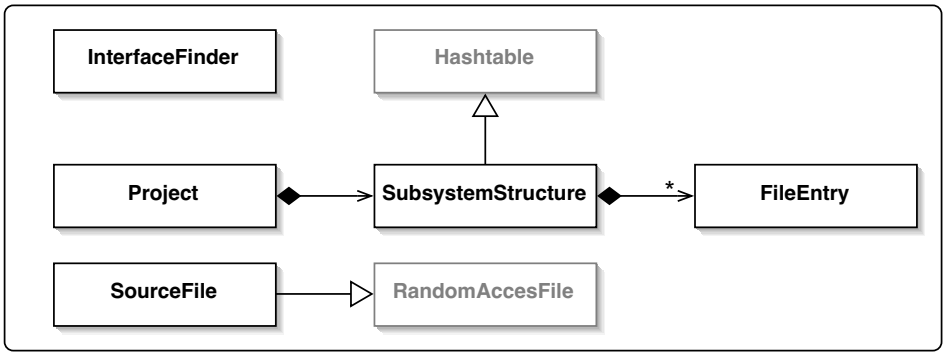


Die Klasse `TableOutput` erzeugt eine Datenstruktur, die von Elementen der *Benutzeroberfläche* direkt als Tabelle ausgegeben werden kann. Mit der Klasse `ListOutput` werden die Schnittstellen als Liste auf dem Bildschirm oder in eine Datei ausgegeben.

Programmsteuerung

Der allgemeine Programmablauf wird von dieser Komponente gesteuert, insbesondere umfasst dies den Programmstart, die Verwaltung von Daten zum aktuellen Projekt und die Steuerung der Projektbearbeitung.

Klassen:

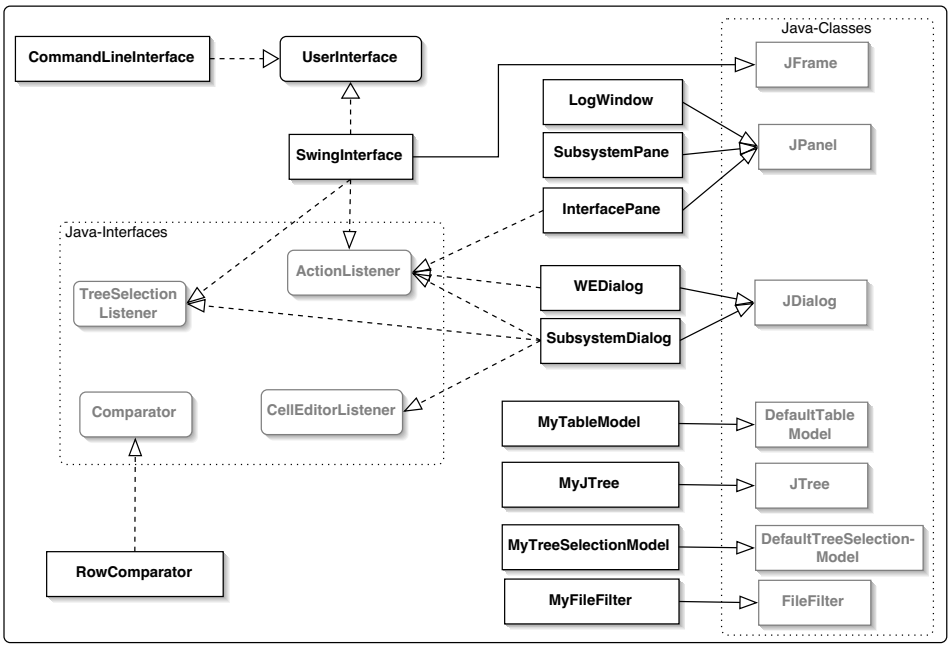


Die Klasse `InterfaceFinder` enthält die `main()`-Methode des Programms, die den Start der weiteren Komponenten vorbereitet. Desweiteren dient diese Klasse für grundlegende Ausgabeoperationen und zum Lesen und Schreiben der Voreinstellungsdatei. In der Klasse `Project` werden alle Informationen zum aktuell zu bearbeitenden Projekt gehalten und die Projektbearbeitung gesteuert. `SubsystemStructure` ist die Datenstruktur für die Subsystemeinteilung des Projektes, die einzelnen Datensätze werden dabei als `FileEntry` gespeichert. Die Klasse `SourceFile` ermöglicht ein einfaches Lesen in Quelldateien.

Benutzeroberfläche

Die Interaktion mit dem Nutzer wird von dieser Komponente übernommen. Es kann zwischen einer grafischen Oberfläche und einer Kommandozeilenversion gewählt werden. Um diese Austauschbarkeit zu ermöglichen, muss die *Benutzeroberfläche* relativ unabhängig vom restlichen Programm sein. Es wird daher nur mit der *Programmsteuerung* und den Ausgabemodulen kommuniziert.

Klassen:



Um die Verwendung verschiedener Nutzeroberflächen zu ermöglichen, beschreibt `UserInterface` Methoden, die angeboten werden müssen, wenn eine Klasse eine Oberfläche implementieren soll. Mit `CommandLineInterface` wird die Kommandozeilenoberfläche realisiert, `SwingInterface` ist die Hauptklasse für die grafische Oberfläche. Die Klassen `WEDialog` und `SubsystemDialog` realisieren die Dialoge zur Eingabe des Working-Environments bzw. für die Subsystemeinteilung. Die drei Bereiche des Hauptfensters werden durch die Klassen `SubsystemPane`, `InterfacePane` und `LogWindow` dargestellt. Die Klassen `MyJTree`, `MyTreeSelectionModel` und `MyTableModel` beschreiben Datenstrukturen für die Darstellung von Baumstrukturen bzw. Tabellen. `RowComparator` ist eine Hilfsklasse zum vergleichen von zwei Tabellenzeilen nach dem Wert aus einer bestimmten Spalte. Die Klasse `MyFileFilter` wird zur Festlegung des auswählbaren Dateityps im File-Open-Dialog benutzt.

2.4.3 Beziehungen zwischen den Komponenten

Das folgende Klassendiagramm in Abbildung 2.4 zeigt die Beziehungen der Klassen verschiedener Komponenten. Dieses Diagramm stellt somit die Verfeinerung von Abbildung 2.3 auf der Ebene der Klassen dar. Zur besseren Übersicht wurden hier Klassen, die nur innerhalb einer Komponente verwendet werden, weggelassen. Für die Symbolik des Diagramms gilt weiterhin das in 2.4.1 auf Seite 14 gesagte.

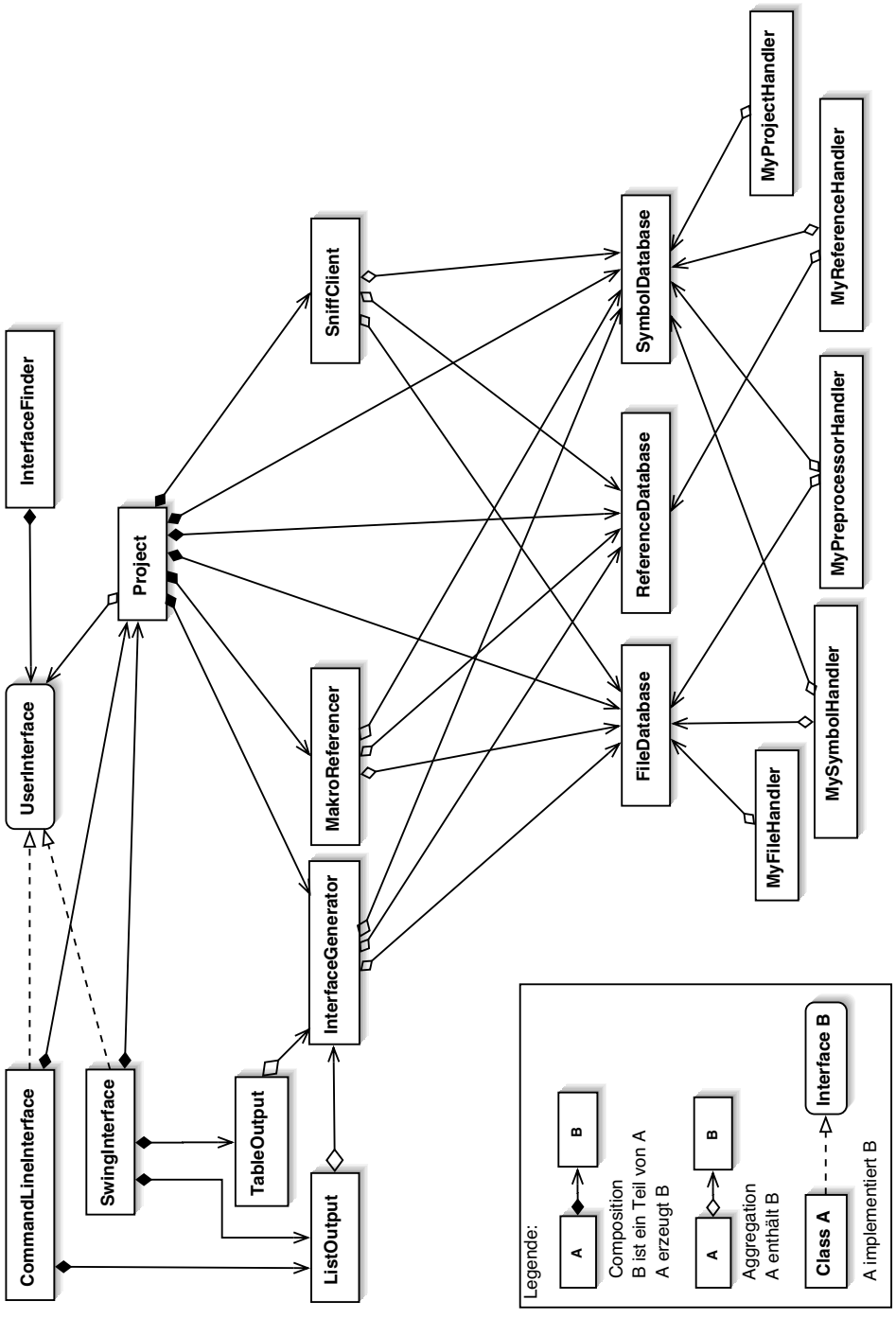


Abbildung 2.4: Beziehungen zwischen den Komponenten

2.5 Implementation

Die Details der Implementation lassen sich anhand des Quelltextes erschließen.Für ein besseres Verständnis sollen hier aber einige Punkte genauer erklärt werden.

Die Quelldateien befinden sich auf der beigelegten CD-ROM im Verzeichnis `InterfaceFinder/Quellen/source`.

2.5.1 Datenhaltung

SymbolDatabase

Neben den vom Symbolhandler empfangenen Informationen zu den Symbolen werden in der Symboldatenbank auch alle definierten Makros gespeichert. Die Makrodefinitionen werden in `MyPreprocessorHandler.handleDefine(...)` empfangen. Da die von SNiFF+ gelieferten Ids der Makrodefinitionen nicht bei anderen Symbolen wiederverwendet werden, wird bei Makros zur ID der Wert `getMaxID()` hinzuaddiert, um Eindeutigkeit zu erreichen. Ansonsten werden Makros wie andere Symbole behandelt.

Zur Unterscheidung der verschiedenen Arten von Symbolen werden in der Klasse `SymbolDatabase` Konstanten definiert, die auch in anderen Programmteilen verwendet werden.

ReferenceDatabase

In der Referenzdatenbank werden die Referenzen gespeichert. Für die Subsystemschnittstellen sind nur Referenzen relevant, bei denen das referenzierte Symbol in einem anderen Subsystem definiert ist als das referenzierende Symbol. Daher werden nur solche Referenzen in die Datenbank aufgenommen. Die Überprüfung, ob verschiedene Subsysteme beteiligt sind, wird bereits im Referenzhandler vorgenommen. Der Vorteil gegenüber dem Speichern aller Referenzen besteht darin, dass die Referenzdatenbank klein bleibt (bei einem Programm wie dem XCTL treten mehrere tausend Referenzen auf). Außerdem ist das Berechnen der Schnittstellen anschließend sehr einfach, die gespeicherten Referenzen müssen nur noch ausgewertet werden (siehe InterfaceGenerator). Der Nachteil ist, dass bei einer Neueinteilung der Subsysteme die Referenzen erneut von SNiFF+ übertragen werden müssen.

2.5.2 SNiFF-Client

MySymbolHandler

In `MySymbolHandler` werden die Symbolinformationen von SNiFF+ empfangen. Die empfangenen Symbole werden in die Symboldatenbank aufgenommen. Je nach Typ des Symbols ist teilweise eine besondere Behandlung notwendig:

`handleFriendDecl()`: Hier wird bei einer Operatorüberladung als Friend-Funktion auf den Sniff-Fehler hingewiesen.

`handleFunction()`: Bei Funktionen muss darauf geachtet werden, dass nur die Datei der *Definition* festgehalten wird, weitere *Deklarationen* interessieren nicht. Bei Memberfunktionen, muss die zugehörige Klasse festgehalten werden.

`handleVariable()`: Neben „normalen“ Variablen können solche mit `extern` oder `static` deklarierte auftreten. Um dies festzustellen, wird im Sourcecode an der Stelle der Variablendefinition nach `extern` und `static` gesucht. Wenn `static` gefunden wird, so wird dies in der Datenbank festgehalten. Bei `extern` ist das aktuelle Vorkommen der Variable keine Definition sondern nur eine Deklaration. Es muss die Definition gefunden werden, damit eindeutig ein Subsystem

zugeordnet werden kann. Dazu werden alle Vorkommen der Variable (gleicher Name) in der Symboldatenbank extra festgehalten und, wenn die Definition gefunden wird, das Subsystem für alle zusammen gesetzt.

MyReferenceHandler

Hier werden die Referenzinformationen empfangen. Für jede Referenz wird geprüft, ob diese über verschiedene Subsysteme reicht, wenn ja, wird das referenzierte Symbol in der Referenzdatenbank festgehalten.

In `handleUndefinedReference(...)` wird eine Warnung ausgegeben, dass das referenzierte Symbol unbekannt ist. Eine weitere Behandlung solcher Referenzen ist nicht möglich.

ReaderThread

Diese Klasse liest den InputStream eines Prozesses aus. Damit wird verhindert, dass der Prozess blockiert. `ReaderThread` wird verwendet, wenn das Sniff-Programm von **InterfaceFinder** gestartet wird, da dieser Prozess sonst blockieren würde.

2.5.3 Schnittstellenermittlung

MakroReferencer

Diese Klasse dient zur Bestimmung der Makrobenutzung. Da SNiFF+ zwar die Informationen zur Makrodefinition liefert, aber nicht deren Benutzung müssen diese Informationen hier selbst gewonnen werden. Makrobenutzung bedeutet, dass der Makrobezeichner irgendwo im Quelltext steht. Somit ist folgende Vorgehensweise möglich: Die Makrodefinitionen sind bekannt und in der Symboldatenbank eingetragen. Dann wird in den Quelldateien nach dem Vorkommen dieser Makrobezeichner gesucht. Wird ein solches Vorkommen gefunden, wird dies wie eine normale Referenz behandelt und wenn nötig in die Referenzdatenbank aufgenommen. Zu beachten ist dabei, dass Makros mit gleichem Bezeichner mehrfach definiert werden können (in verschiedenen Dateien). Dann muss versucht werden, die gültige Definition zuzuordnen, damit auch ein Subsystem zugeordnet werden kann. Dabei wird geprüft, ob eine der Definitionen in der gleichen oder einer der inkludierten Dateien steht. Falls dabei keine gültige Definition gefunden wird, wird die erste vorhandene Definition verwendet und eine Warnung ausgegeben, dass die Makrobenutzung nicht korrekt ermittelt werden kann.

InterfaceGenerator

Die Klasse `InterfaceGenerator` extrahiert aus den Informationen in der Datenbank die eigentlichen Schnittstelleninformationen und stellt diese den Ausgabemodulen zur Verfügung. Da in die Referenzdatenbank nur Symbole aufgenommen werden, die aus einem anderen Subsystem referenziert werden, gehören alle diese Symbole zu einer Subsystemschnittstelle. So muss für jedes in der Referenzdatenbank enthaltene Symbol nur das Subsystem bestimmt werden, um es in die Schnittstellenliste für das entsprechende Subsystem aufzunehmen.

2.5.4 Schnittstellenausgabe

Die Klassen der Schnittstellenausgabe greifen auf die Informationen des `InterfaceGenerator` zurück und bereiten diese für die entsprechende Ausgabeform auf.

Die Ausgabemodule stellen ein Bindeglied zwischen Schnittstellenermittlung und Nutzeroberfläche dar. Hier bietet sich somit ein guter Ansatzpunkt für Erweiterungen, z.B. könnte ein Ausgabemodul für Statistikinformationen entwickelt werden.

2.5.5 Programmsteuerung

InterfaceFinder

In der `main()`-Methode des Programms werden die Programmparameter ausgewertet. Die Option „-c“ schaltet die Swing-Oberfläche aus und als zweiter Parameter kann das Verzeichnis des SNiFF+ -Programms angegeben werden.

Project

Die Klasse `Project` verwaltet Daten zum aktuellen SNiFF+ -Projekt und steuert dessen Bearbeitung.

Die Projektbearbeitung erstreckt sich über zwei Methoden: `startProject()` übernimmt den ersten Teil mit dem Verbindungsaufbau zu SNiFF+ dem Einlesen der Dateiinformationen und dem ersten Zuordnen der Subsysteme nach der Verzeichnisstruktur. Fortgesetzt wird in `processProject(...)` mit dem Bearbeiten der Subsystemeinteilung, dem Einlesen der Symbol- und Referenzinformationen, der Bestimmung der Makrobenutzung, der Bestimmung der Schnittstellen und schließlich deren Ausgabe. Die Aufteilung der Projektbearbeitung auf zwei Methoden ist notwendig, um nach einem Verändern der Subsystemeinteilung wieder einen Einstiegspunkt zu haben. Beide Methoden werden außerdem in eigenen Threads ausgeführt, da sonst die Swing-Oberfläche blockieren würde und somit die Ausgabe von Meldungen nicht mehr erfolgen würde.

SourceFile

`SourceFile` ist eine Klasse zum einfachen Lesen in Quelldateien. Dabei wird ein FileCounter verwaltet, der unabhängig von der Darstellung der Zeilenenden (Unix/Windows Zeilenenden) ist. Für jedes Zeilenende wird der Zähler immer nur um einen Schritt weitergesetzt. Der normale File-Counter zählt dagegen die Anzahl der Zeichen, also evtl. zwei Schritte beim Zeilenende. Damit wird Übereinstimmung mit den Positionsangaben von SNiFF+ erreicht. Mit `getFileCounter()` kann die aktuelle Position in der Datei abgefragt werden.

2.5.6 Benutzeroberfläche

SubsystemDialog

`SubsystemDialog` realisiert den Dialog zum Bearbeiten der Subsystemzuordnung. Die Subsystemzuordnung wird als `SubsystemStructure` übergeben, die als Originalzuordnung erhalten bleibt. Die Subsystemzuordnung wird als Baum (`MyJTree`) dargestellt, der die Struktur intern als `DefaultTreeModel` verwaltet. Änderungen werden zunächst nur am `TreeModel` vorgenommen, erst zum Abschluss wird daraus wieder eine `SubsystemStructure` erzeugt, die andere Objekte abfragen können.

MyTableModel

Als Datenmodell für die Schnittstellentabelle erweitert `MyTableModel` das `DefaultTableModel` um die Möglichkeit die Tabelle nach beliebigen Spalten zu sortieren (Mausklick auf Spaltenkopf). Die Methode `sortByColumn(...)` vergleicht mit Hilfe

von `RowComparator` alle Zeilen nach dem Wert der ausgewählten Spalte und ersetzt den Tabelleninhalt durch die sortierten Zeilen.

2.6 Test

Um die korrekte Arbeitsweise von **InterfaceFinder** zu überprüfen, muss ein Test des Programms durchgeführt werden. Dabei kann zwischen dem Test der Funktionalität, d.h. der korrekten Bestimmung der Subsystemschnittstellen, und dem Test der Oberfläche (GUI) unterschieden werden.

2.6.1 Test der Funktionalität

Um die Funktionalität eines Programmes zu testen, werden im Allgemeinen Testdaten und dazu passende Ergebnisse festgelegt, die Testdaten dem Programm als Eingabe gegeben, und die Ausgaben des Programms mit den erwarteten Ergebnissen verglichen. Im Falle von **InterfaceFinder** bestehen Testdaten aus einem C++-Programm als SNiFF+ -Projekt und die zu erwartenden Ergebnisse sind Subsystemschnittstellen.

Um **InterfaceFinder** zu testen werden zwei verschiedene solcher Testdatensätze verwendet: Zum einen wird ein C++-Programm (im Folgenden *Testsystem* genannt) mit bekannten Subsystemschnittstellen geschrieben und darauf **InterfaceFinder** angewendet. Zum anderen wurden in [Thiel] die Subsystemschnittstellen des XCTL-System manuell herausgearbeitet, so dass die dortigen Ergebnisse verglichen werden können mit einer Anwendung von **InterfaceFinder** auf die entsprechende XCTL-Version.

Testsystem

Das Testsystem ist ein korrekt compilierbares C++-Programm, dass aus zwei Subsystemen, *Sub_A* und *Sub_B*, besteht.

Im Subsystem Sub_B werden alle von **InterfaceFinder** zu erkennenden C++-Sprachelemente definiert und diese im Subsystem Sub_A referenziert, womit alle diese Elemente die Subsystemschnittstelle von Sub_B bilden. Zusätzlich werden in Sub_B alle Elemente in einer zweiten Version definiert, die aber nicht in Sub_A referenziert wird sondern nur in Sub_B. Diese gehören somit nicht zur Schnittstelle von Sub_B.

Folgende Elemente (mit entsprechender Benutzung) müssen getestet werden:

- Klassen/Strukturen/Unions:
  - Zugriff als Typ (nur Klassenname)
  - Zugriff auf Membervariable
  - Aufruf von Memberfunktionen
- Funktionen
  - Aufruf
- Variablen
  - globale Variable
  - externe Variable

- Enums
  - Benutzung des Enums
  - Benutzung von Enum-items
- Typedefs
  - Benutzung
- Makros
  - Benutzung

Zusätzlich werden static Variable und Makros, die nur innerhalb einer Datei sichtbar sind, mit gleichem Namen in beiden Subsystemen definiert, um zu testen, ob **InterfaceFinder** die Elemente unterscheidet und bei Makros eine entsprechende Warnung ausgibt. Ein Warnhinweis wird von **InterfaceFinder** auch erwartet bei einer Operatorüberladung als friend-Funktion, weshalb diese auch im Testsystem enthalten ist.

Der Quelltext des Testsystems Der Quelltext des Testsystems wird in Anhang C wiedergegeben. Die zugehörigen Dateien befinden sich auf der beigelegten CD-ROM im Verzeichnis `Test/Testsystem`.

Subsystemschnittstellen des Testsystems Die Subsystemschnittstellen des Testsystem lassen sich leicht am Quelltext ablesen. Sie sehen nun folgendermaßen aus:

- Sub_A

  keine Elemente

- Sub_B

Variable	var_ext_b
Klasse	Klasse_B_ex
Klasse	Test_Operator
Membervariable	klasse_B_ex.member_b_2
Memberfunktion	klasse_B_ex.function_b_2
Struct	Struct_B_ex
Structmember	struct_B_ex.member_b_5
Union	Union_B_ex
Unionmember	union_B_ex.member_b_7
Unionmember	union_B_ex.member_b_8
Funktion	function_b_ex
Funktion	function_b_ex_2
Enum	enum_b_ex
Enumitem	e3
Enumitem	a1
Enumitem	a2
Typedef	type_b_ex
Makro	makro_b_ex

Da **InterfaceFinder** statt der einzelnen Enumitems die entsprechenden Enums als Schnittstellenelement anzeigt werden in der Ausgabe die Items `e3, a1, a2` nicht erscheinen, aber der Enum `enum_b_ex_2`, da dieser die Items `a1` und `a2` enthält, `e3` gehört zu `enum_b_ex`.

Testablauf Nun kann das Testsystem in SNiFF+ eingelesen und anschließend **InterfaceFinder** mit der vorgegebenen Subsystemeinteilung darauf angewendet werden. Die von **InterfaceFinder** berechneten Subsystemschnittstellen müssen den oben aufgeführten entsprechen. Außerdem muß **InterfaceFinder** Warnungen in Bezug auf die Operatorüberladung und die Definition des Makros `makro_in` ausgeben.

Testergebnis Die folgende Abbildung 2.5 zeigt **InterfaceFinder** nach der Bearbeitung des Testsystems. Der Testlauf wurde durchgeführt am 16.06.2003.

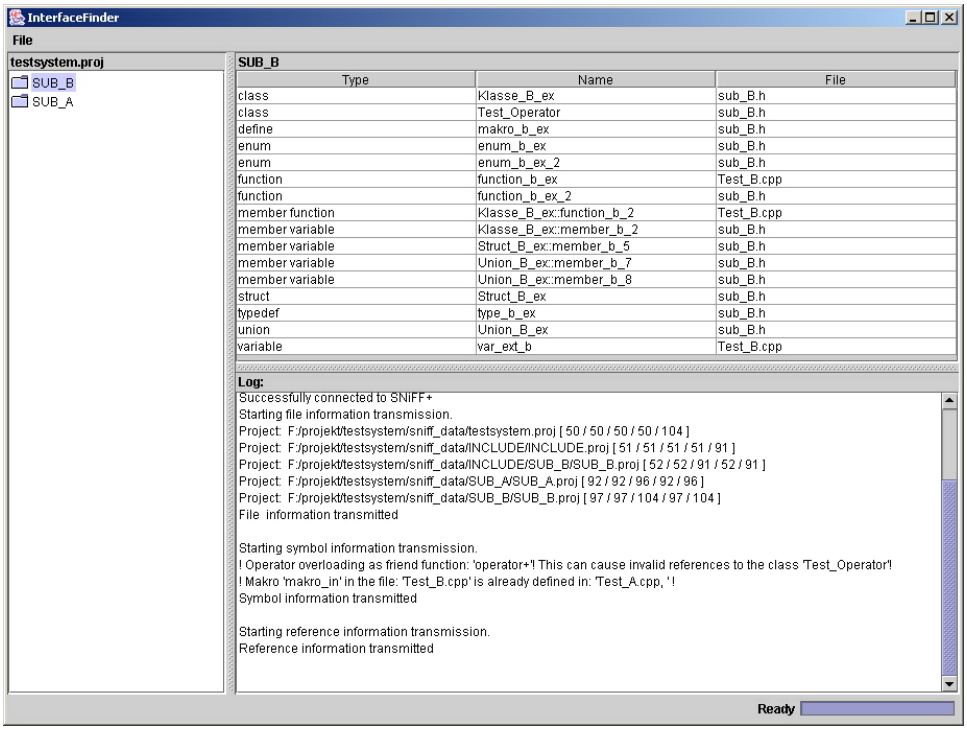


Abbildung 2.5: Ergebnis des Testlaufs

Die berechneten Subsystemschnittstellen entsprechen den obigen Erwartungen (die im Bild nicht sichtbare Schnittstelle des Subsystems Sub_A ist leer), und die entsprechenden Warnmeldungen zur Operatorüberladung und zur Makrodefinition werden ausgegeben.

Der Test ist somit erfolgreich verlaufen.

XCTL

Die in [Thiel] manuell herausgearbeiteten Subsystemschnittstellen des XCTL-Systems, sind jeweils in den Subsystem-Headerdateien enthalten. Diese Schnittstellen werden nun mit den Ergebnissen der Anwendung von **InterfaceFinder** auf die entsprechende XCTL-Version verglichen. Dabei ist zu beachten, dass die Headerdateien auch Elemente enthalten, die nicht außerhalb des Subsystem benutzt werden, aber in der Headerdatei enthalten sein müssen, da andere Elemente in der Datei diese benötigen. Diese Elemente werden nicht zur Subsystemschnittstelle gezählt und werden somit nicht von **InterfaceFinder** gefunden. Um den Vergleich durchführen zu können, wurden aus den Headerdateien manuell die eigentlichen Schnittstellenelemente herausgesucht und in den folgenden Tabellen aufgelistet.

Diese Tabellen enthalten jeweils drei Spalten. Die erste Spalte enthält den Typ und den Namen der auftretenden Elemente. In der zweiten Spalte „Headerdatei“ ist eine Markierung '*' gesetzt, wenn das Element in der entsprechenden Headerdatei enthalten ist, während die dritte Spalte „**InterfaceFinder**“ markiert ist, wenn von **InterfaceFinder** das Element als zugehörig zur Schnittstelle erkannt wurde.

Im Idealfall sollten immer beide Spalten markiert sein. Abweichungen werden jeweils im Anschluss unter der Tabelle erklärt.

Sämtliche Variable die mittels „`extern`“ in verschiedenen Subsystemen benutzt werden, und somit zu einer Subsystemschnittstelle gehören, sind nicht in den Headerdateien enthalten, da diese in .cpp-Dateien definiert sind. Diese Variablen werden aber von **InterfaceFinder** gefunden. In der Tabelle ist daher nur in der Spalte „**InterfaceFinder**“ eine Markierung gesetzt.

AUTOJUST

Element	Headerdatei	**InterfaceFinder**
class TAutomaticAngleControl	*	*
class TMatrix		*

In der Klasse `TMatrix` wird der *-Operator als friend-Funktion überladen. Dies erkennt SNiFF+ nicht korrekt und erzeugt bei Verwendung des '*' (z.B. bei Multiplikationen) Referenzen auf die Klasse `TMatrix`, wodurch **InterfaceFinder** dies zur Schnittstelle des Subsystems zählt.

DATAVISA

Element	Headerdatei	**InterfaceFinder**
class TCurve	*	*
class TCurveShowParam	*	*
class TBitmapSource	*	*
class TDataBase	*	*
class TPlotData	*	*
struct CPoint	*	*
struct TBMContens	*	*
struct TCoorSystem	*	*
struct TDisplay	*	*
enum TOrder	*	*
enum TOutputType	*	*
enum TSaveFormat	*	*
typedef LPCurve	*	*
typedef LPDataBase	*	*
typedef TData		*
typedef TKSystem	*	*
Makro ArToRLx	*	*
Makro ArToRLy	*	*
Makro DegToRad	*	*
variable lpDataBase		*

Die Variable `lpDataBase` ist definiert in der Datei `M_CURVE.CPP` (DATAVISA) und wird „`extern`“ verwendet in `L_LAYER.CPP` (INTERNLS).

Die Struktur `TData` ist sowohl in `M_DATA.CPP` (DATAVISA), als Typedef mit einer unbenannten Struktur, als auch in `M_ARSCAN.CPP`(DIFRKMTY) definiert, wird

aber nur in `M_ARSCAN.CPP` verwendet. SNiFF+ sieht diese Verwendung als Referenz auf die Definition in `M_DATA.CPP`, womit **InterfaceFinder** `TData` zur Schnittstelle des Subsystems *Datavisa* hinzufügt. Die Definition in `M_DATA.CPP` ist wahrscheinlich überflüssig und sollte entfernt werden.

DETECUSE

Element	Headerdatei	**InterfaceFinder**
class TDevice	*	*
class TDList	*	*
class TPsd	*	*
enum TDeviceType	*	*
enum TPsdDataType	*	*
function dGetExposureValues	*	*
function dlGetDevice	*	*
function dlGetIdByName	*	*
function dlGetInstance	*	*
function dlGetVersion	*	*
function dlIsDeviceValid	*	*
function dlParsingDevice	*	*
function dlSetDevice	*	*
function dMeasureStart	*	*
function dMeasureStop	*	*
function dSetExposureValues	*	*
function GetCounterListPtr	*	*
function InitializeCountersDLL	*	*
typedef LPDevice	*	*
typedef LPDList	*	*
Makro D_DiffractionScan	*	*

DIFRKMTY

Element	Headerdatei	**InterfaceFinder**
class TAreaScan	*	*
class TAreaScanCCD	*	*
class TAreaScanParameters	*	*
class TAreaScanCCDParameters	*	*
class TScan	*	*
class TScanParameters	*	*
struct TReportUse	*	*
variable nMaxScaleIdx	*	*
variable nMouseAction		*
variable NoAskForSave	*	*
variable PointX		*
variable PointY		*
variable SaveWithNewName	*	*

Die Variablen `nMouseAction, PointX` und `PointY` sind definiert in `M_ARSCAN.CPP` (DIFRKMTY) und werden „`extern`“ verwendet in `M_DATA.CPP` (DATAVISA).

INTERNLS

Element	Headerdatei	**InterfaceFinder**
class TIOPort		*
class TMain	*	*
class TMDIWindow	*	*
struct THKL	*	*
struct TKSProperties	*	*
enum TScaleType	*	*
enum TUnitType	*	*
enum TxScanType	*	*
function CreateDefaults	*	*
function Delay	*	*
function DelayTime	*	*
function FileOpen	*	*
function FileSave	*	*
function GetCFile	*	*
function GetClientHandle	*	*
function GetFileLine	*	*
function GetFrameHandle	*	*
function GetHKL	*	*
function GetHWFile	*	*
function GetMacroFile	*	*
function GetMainInstance	*	*
function GetTestDevFile	*	*
function NewWindow		*
function ParsingXScanType		*
function SetFPOnData	*	*
function SetHKL	*	*
function SetInfo	*	*
function SetStaticInfo	*	*
function SetStatus	*	*
function UnitEnum	*	*
typedef HPBYTE	*	
typedef HPFLOAT	*	
typedef HPSTR	*	
Makro DeviceTimerIdStart	*	*
Makro EOL	*	*
Makro LEFT	*	*
Makro MaxString	*	*
Makro MBASK	*	*
Makro MBFAILURE	*	*
Makro MBINFO	*	*
Makro MBSTOP	*	*
Makro NO	*	*
Makro OFF	*	*
Makro O_Flush	*	*
Makro ON	*	*
Makro O_PaintPoint	*	*
Makro P_X	*	*
Makro P_Y	*	*
Makro P_Z	*	*
Makro R_EndOfFile	*	*
Makro R_Failure	*	*

Element	Headerdatei	**InterfaceFinder**
Makro R_HardOverflow	*	*
Makro RIGHT	*	*
Makro R_MeasInProcess	*	*
Makro R_MeasOk	*	*
Makro R_OK	*	*
Makro R_Overflow	*	*
Makro R_SoftOverflow	*	*
Makro R_TimeOut	*	*
Makro R_UnknownCmd	*	*
Makro TimerIdInformation	*	*
Makro W_AdjustmentWindow	*	*
Makro W_CounterWindow	*	*
Makro W_DiffractionWindow	*	*
Makro W_EditWindow	*	*
Makro W_PSDWindow	*	*
Makro YES	*	*
variable bManualMovesCorrected		*
variable bModulLoaded		*
variable lpDBase		*
variable lpDList		*
variable Main		*
variable wm_WakeUpSteering		*

Die Klasse `TIOPort` ist in `HWIO.H` (INTERNLS) definiert und wird in der Datei `RADICOHW.H` (DETECUSE) verwendet. An dieser Stelle ist also die Headerdatei unvollständig, **InterfaceFinder** bringt hier das korrekte Ergebnis.

Die Funktion `NewWindow` wird in `L_LAYER.H` (INTERNLS) definiert und in verschiedenen anderen Subsystemen verwendet. **InterfaceFinder** zählt diese Funktion also korrekt zur Subsystemschnittstelle von INTERNLS. Es sollte in `EVRYTHNG.H` ein Prototyp der Funktion eingefügt werden.

Die Funktion `ParsingXScanType` ist in `L_LAYER.CPP` (INTERNLS) definiert und wird in `M_STEERG.CPP` (WORKFLOW) aufgerufen. Auch hier sollte in `EVRYTHNG.H` ein Prototyp stehen.

Die Definition der drei Typedefs `HPSTR, HPBYTE` und `HPFLOAT` stehen in einem Codeabschnitt der nur compiliert wird, wenn `__FLAT__` definiert ist. SNiFF+ übergeht diesen Abschnitt und erkennt daher diese drei Typedefs nicht als Symbole, womit auch **InterfaceFinder** diese Typedefs nicht kennt.

`wm_WakeUpSteering`, `bManualMovesCorrected Main`, `lpDBase`, `bModulLoaded` und `lpDList` sind Variablen, die in `M_MAIN.CPP` (INTERNLS) definiert sind und in verschiedenen Subsystemen „`extern`“ verwendet werden. Eine Variable `lpDList` wird auch in `COUNTERS.CPP` (DETECUSE) definiert. SNiFF+ ist nicht in der Lage beide Definitionen zu unterscheiden.

MOTRSTRG

Element	Headerdatei	**InterfaceFinder**
enum TAxisType	*	*
enum TMParameter	*	*
enum TValueType	*	*
function InitializeMotorsSimulation		*
function mActivateDrive	*	*
function mGetAxisName	*	*

Element	Headerdatei	**InterfaceFinder**
function mGetAxisUnit	*	*
function mGetDF	*	*
function mGetDistance	*	*
function mGetDistanceProcess	*	*
function mGetMoveFinishIdx	*	*
function mGetMoveScan	*	*
function mGetScanSize	*	*
function mGetSF	*	*
function mGetUnitType	*	*
function mGetValue	*	*
function mExecuteCmd	*	*
function mMoveToDistance	*	*
function mPopSettings	*	*
function mPushSettings	*	*
function mSetCorrectionState	*	*
function mSetRelativeZero	*	*
function mSetValue	*	*
function mStopDrive	*	*
function mIsRangeHit	*	*
function mIsCalibrated	*	*
function mIsDistanceRelative	*	*
function mIsMoveFinish	*	*
function mlGetAxis	*	*
function mlGetAxisNumber	*	*
function mlGetDistance	*	*
function mlGetIdByName	*	*
function mlGetInstance	*	*
function mlGetOffset	*	*
function mlGetValue	*	*
function mlGetVersion	*	*
function mlInitializeMotorsDLL	*	*
function mlInquireReferencePointDlg	*	*
function mlIsAxisValid	*	*
function mlIsMoveFinish	*	*
function mlMoveToDistance	*	*
function mlOptimizingDlg	*	*
function mlParsingAxis	*	*
function mlPositionControlDlg	*	*
function mlSaveModuleSettings	*	*
function mlSetAngleDefault	*	*
function mlSetAxis	*	*
function mlSetParametersDlg	*	*
function UnInitializeMotorsSimulation		*
variable hModuleInstance		*
vairable szMessage		*
variable szMsgFailure		*
variable szMsgLine0[01...12]		*

Die beiden Funktionen `InitializeMotorsSimulation` und `UnInitializeMotors-Simulation` werden in der Datei `MSIMSTAT.CPP` (MOTRSTRG) definiert und in `M_MAIN.CPP` (INTERNLS) aufgerufen.

Die Variablen `szMsgLine001...szMsgLine012, szMessage` und `szMsgFailure` werden unabhängig voneinander in den Dateien `M_ARSCAN.CPP` (DIFRKMTY) und

`MOTORS.CPP` (MOTRSTRG) definiert und verwendet (die beiden Dateien gehören zu verschiedenen Übersetzungseinheiten/DLL's, werden also nicht zusammen kompiliert). In SNiFF+ kann das XCTL-Projekt nicht in verschiedene Übersetzungseinheiten aufgeteilt werden, was dazu führt, dass SNiFF+ die Variablen nicht unterscheiden kann, und die Verwendung in `M_ARSCAN.CPP` als Referenz auf die jeweilige Variable in `MOTORS.CPP` gesehen wird.

Eine Variable `hModuleInstance` ist in mehreren Dateien definiert, unter anderem in `M_LAYER.CPP` (MOTRSTRG). Außerdem wird `hModulInstance` in der Datei `DLG_TPL.CPP` (SWINTRAC) als `externe` Variable benutzt. **InterfaceFinder** muss beim Auftreten der `extern`-Version der Variable diese einer Definition zuordnen. Wenn es mehrere Definitionen gibt, kann die korrekte Zugehörigkeit nicht festgestellt werden und **InterfaceFinder** benutzt die erste gefundene Definition, in diesem Fall die in `M_LAYER.CPP`.

SWINTRAC

Element	Headerdatei	**InterfaceFinder**
class TAngleControl	*	*
class TCounterShowParam	*	*
class TCounterWindow	*	*
class TExecuteCmd	*	*
class TGetData	*	*
class TMeasurementParam	*	*
class TModalDlg	*	*
class TModelessDlg	*	*
class TPsdParameters	*	*
variable TheDialog		*
variable TheModeless		*

`TheModeless` und `The Dialog` sind `extern` verwendete Variable aus `DLG_TPL.CPP` (SWINTRAC).

TOPOGRFY

Element	Headerdatei	**InterfaceFinder**
class TTopography	*	*
class TTopographyExecute	*	*
class TTopographySetParam	*	*
variable Topography		*

Die `extern`-Variable `Topography` ist in `M_TOPO.CPP`(TOPOGRFY) definiert.

WORKFLOW

Element	Headerdatei	**InterfaceFinder**
class TAdjustmentExecute	*	*
class TAdjustmentWindow	*	*
class TAreaScanCmd		*
class TMacroExecute	*	*
class TSaveDataCmd		*
class TScanCmd		*
class TSetAdjustmentParam	*	*

Element	Headerdatei	**InterfaceFinder**
class TSetupScanCmd		*
class TSteering	*	*
struct TCmdTag	*	*
struct TMacroTag	*	*
enum TCParam	*	*
enum TCmdId	*	*
enum TMacroId	*	*
variable Steering		*

Die Klassen `TSaveDataCmd, TAreaScanCmd, TSetupScanCmd` und `TScanCmd` werden in der Definition der Klasse `TAreaScan` als `friend`-Klassen gennant. Dies sieht SNiFF+ als Referenz auf diese Klassen, womit **InterfaceFinder** diese Klassen in die Subsystemschnittstelle einordnet.

Die Variable `Steering` ist in `M_STEERG.CPP` (WORKFLOW) definiert und wird in verschiedenen Subsystemen als `extern`-Variable benutzt.

Testergebnis

Die gefundenen Abweichungen beruhen auf dem bereits oben erklärten Problem der externen Variablen, auf fehlerhaften Referenzinformationen von SNiFF+ oder auf Unvollständigkeiten in den Headerdateien. Die Ergebnisse von **InterfaceFinder** sind aber in allen Fällen korrekt, so dass der Test als erfolgreich gewertet werden kann.

Die Export-Datei der gefundenen Subsystemschnittstellen und die für den Test verwendete XCTL-Version befinden sich im Verzeichnis `Test/XCTL-Vergleich` auf der beigelegten CD.

2.6.2 Test der Oberfläche

Beim Test der Benutzerberfläche muss das korrekte Verhalten der Oberflächenelemente (Fenster, Dialoge, Buttons, ...) überprüft werden. Dazu werden im Folgenden Testfälle definiert, in dem die vom Benutzer auszuführenden Aktionen und die dazu erwarteten Reaktionen des Programms beschrieben werden. Außerdem werden zum Teil „Bedingungen“ genannt, die vom Programm erfüllt werden müssen. Der Tester muss diese überprüfen, ohne dass ihm dafür genaue Handlungsanweisungen gegeben werden.

Die Dialoge „File Open“ und „File Save“ müssen nicht genauer getestet werden, da dies Standarddialoge von Java sind und in ihrer Funktionalität nicht erweitert wurden.

Die Testfälle sind so angelegt, dass jedes Oberflächenelement mindestens einmal „aktiviert“ wird.

Testfall 1: Hauptfenster

Benutzeraktion	**erwartete Reaktion(-en)**
InterfaceFinder starten	„Welcome“-Splashscreen erscheint und verschwindet wieder nach kurzer Zeit Hauptfenster erscheint in der Mitte des Bildschirms: Text „no open project“ im Subsystemanzeigefeld, andere Felder leer
horizontalen und vertikalen Trennbalken verschieben	Bereiche passen sich an

Benutzeraktion	erwartete Reaktion(-en)
Fenster vergrößern, verkleinern, maximieren	Bereiche werden entsprechend größer oder kleiner
Fenster schließen	Fenster verschwindet, Programm wird beendet

Testfall 2: Menu

Benutzeraktion	erwartete Reaktion(-en)
bei allen folgenden Aktionen die Bedingungen (rechte Spalte) überprüfen	Die Menupunkte dürfen nur aktiv sein, wenn dies sinnvoll ist: • **Open...** und **Open recent** nur wenn kein Projekt geöffnet ist. • **Close, Export...** und **Reassign...** nur nach abgeschlossener Projektbearbeitung • **Quit** ist immer aktiv

InterfaceFinder starten	siehe: Testfall 1
aus dem Menu **File** → **Open...** wählen	File-Open-Dialog erscheint
Cancel-Button drücken	File-Open-Dialog verschwindet, Hauptfenster wieder wie zuvor
File → **Open...** wählen	File-Open-Dialog erscheint
im Textfeld Namen einer nicht existierenden Datei eingeben, **Open**-Button drücken	File-open-Dialog verschwindet, im Log-Fenster erscheint Fehlermeldung, dass die Datei nicht existiert.
File → **Open...** wählen	File-Open-Dialog erscheint
Datei `testsystem.proj` suchen und auswählen	(nur .proj-Dateien können ausgewählt werden)
Open-Button drücken	File-open-Dialog verschwindet, Dialog zur Eingabe des Working-Environments erscheint, dessen Textfeld enthält bereits Vorschlag (`/testsystem`) für das Working-Environment. Im Log-Fenster erscheint der Pfadname der Projektdatei. Statusbalken beginnt sich vorwärts zu bewegen und erhält Beschriftung „Working“
OK-Button drücken	Dialog „Working Environment“ verschwindet, Projektbearbeitung beginnt
Projektbearbeitung abwarten, im Dialog „Assign Subsystems“ **OK**-Button drücken	Im Log-Fenster werden Meldungen zur Projektbearbeitung ausgegeben. Falls nötig wird das SNiFF+-Programm gestartet. Der Statusbalken schreitet voran und erreicht mit dem Abschluß der Projektberabeitung sein Maximum, die Beschriftung wechselt zu „Ready“. Die Subsystemstruktur und die Schnittstelle des ersten Subsystems wird angezeigt.

Benutzeraktion	**erwartete Reaktion(-en)**
File → **Close** wählen	Subsystem- und Schnittstellenanzeige werden gelöscht, Statusbalken wird zurückgesetzt.
File → **Open recent** öffnen	Eintrag `testsystem.proj` muss vorhanden sein
testsystem.proj auswählen	Projektbearbeitung wie nach **Open...**
File → **Export** wählen	File-Save-Dialog erscheint
Dateinamen eingeben, gewünschten Speicherort auswählen, **Save**-Button drücken	File-Save-Dialog schließt sich, Datei mit Schnittstellenliste wird geschrieben
File → **Close** wählen	Subsystem- und Schnittstellenanzeige werden gelöscht, Statusbalken wird zurückgesetzt.
File → **Open recent** → **Clear Menu** wählen	Einträge im **Open recent**-Menu werden gelöscht.
File → **Quit** wählen	Fenster schließt sich, Programm wird beendet

Testfall 3: Dialog „Assign Subsystems“

Benutzeraktion	erwartete Reaktion(-en)
InterfaceFinder starten, die Datei `testsystem.proj` öffnen und WE-Dialog bestätigen	Projektbearbeitung beginnt, der Dialog „Assign Subsystems“ erscheint. Die angezeigte Subsystemstruktur entspricht der Verzeichnisstruktur der Quelltextdateien. **Up-**, **Down-** , **Remove-** und **Revert-**Buttons sind inaktiv.
nach Belieben Subsysteme auswählen, öffnen, schließen, hinzufügen, löschen, umbenennen; Dateien auswählen, nach oben (unten) verschieben	folgende Bedingungen müssen eingehalten werden: • Dateien können nicht über das oberste (unterste) Subsystem hinaus nach oben (unten) verschoben werden (Buttons werden entsprechend [in-]aktiv) • nur leere Subsysteme können gelöscht werden • Namen für Subsysteme dürfen nicht doppelt benutzt werden (bei mehrmaligen **New** werden Ziffern an „untitled“ angehängt) • bei mehrfacher Auswahl: – entweder nur Dateien oder nur Subsysssteme – nur zusammenhängend – bei Dateien nur innerhalb eines Subsystems • wenn Liste zu lang wird muss eine Scrollbar erscheinen
Revert-Button drücken	Subsystemstruktur wie zu Anfang wird wiederhergestellt, alle Buttons bis auf **New** und **Ok** sind inaktiv
Ok-Button drücken	Dialog schließt sich, Projektbearbeitung wird fortgesetzt Subsystemstruktur wurde übernommen
File → **Reassign** wählen	Dialog „Assign Subsystems“ erscheint
Subsystemstruktur verändern, **OK**-Button drücken	Dialog schließt sich, Subsystemschnittstellen werden neu berechnet neue Subsystemstruktur wurde übernommen

Testfall 4: Subsystem- und Schnittstellenanzeige

Benutzeraktion	erwartete Reaktion(-en)
InterfaceFinder starten, die Datei `testsystem.proj` öffnen und bearbeiten lassen	Subsystemschnittstellen werden berechnet linker Teil des Hauptfensters zeigt Subsystemstruktur an
Subsystem auswählen	Subsystemschnittstelle wird als Tabelle angezeigt
im Tabellenkopf auf eine Spalte klicken	Tabelle wird nach dieser Spalte sortiert
`Shift`-drücken und dabei im Tabellenkopf auf eine Spalte klicken	Tabelle wird nach dieser Spalte umgekehrt sortiert
Subsystem öffnen (doppelklicken)	zugehörige Dateien werden sichtbar
Datei auswählen	Inhalt der Datei wird angezeigt
	zu überprüfende Bedingungen • in allen Anzeigen erscheint Scrollbar wenn der Platz nicht ausreicht • Subsystemanzeige erlaubt keine Mehrfachauswahl • Tabellenspalten können verschoben(umsortiert) werden • oberhalb der Subsystemanzeige steht der Projektname • oberhalb der rechten Anzeigefläche steht der Name des ausgewählten Subsystems bzw. der Dateiname

Testergebnis

Der Test wurde am 12.06.2003 durchgeführt. Dabei konnten alle Testfälle erfolgreich abgeschlossen werden.

2.7 Probleme

Zu Problemen bei der Verwendung von **InterfaceFinder** führt die Tatsache, dass das Programm auf verschiedenen Rechnern, trotz gleichem Betriebssystem, unterschiedliches Verhalten zeigt. Die Ursache für das unterschiedliche Verhalten liegt vermutlich an verschiedenen Java-Versionen auf den einzelnen Rechnern.

Auffallend, aber ansonsten unproblematisch, sind kleine Darstellungsunterschiede: Text wird auf einem Rechner schwarz, auf anderen blau dargestellt.

Unterschiedliches Zeitverhalten führt dazu, dass beim Starten von SNiFF+ unterschiedlich lange gewartet wird, zum Teil so kurz, dass SNiFF+ den Startvorgang noch nicht abgeschlossen hat.

Problematisch ist das Verhalten beim Schreiben der Voreinstellungsdatei (Einträge des **Open recent** - Menus). Auf manchen Rechnern werden Leerzeilen in die Datei eingefügt, was beim Wiedereinlesen zu Fehlern führte, erst ein zusätzlicher Test auf leere Eingaben im Programm schafft hier Abhilfe.

Kapitel 3

Anwendung

Mit **InterfaceFinder** ist es nun möglich Subsystemschnittstellen schnell zu bestimmen. Damit können leicht verschiedene Subsystemeinteilungen ausprobiert und nach einer „optimalen“ Einteilung gesucht werden. Im Folgenden soll nun die Anwendung an einigen Beispielen mit verschiedenen Subsystemeinteilungen gezeigt werden. Dabei soll versucht werden, anhand der gefundenen Schnittstellen Aussagen über die Struktur des Programms und die Vor- und Nachteile der einzelnen Subsystemeinteilungen zu treffen. Die Informationen dazu, werden aus Größen gewonnen, die an den Schnittstellen messbar sind. Die Bedeutung und Funktion der Elemente in den Schnittstellen wird nicht mit berücksichtigt. Dies führt zu einer Bewertung der Schnittstellen, die objektiv und nachvollziehbar ist und daher eventuell auch automatisiert erstellt werden könnte. Mit solch einer Bewertung kann aber keine Aussage darüber getroffen werden, ob eine Subsystemeinteilung im konkreten Fall sinnvoll ist, da hier die Bedeutung und Funktion der Subsysteme und Schnittstellen nicht erfasst wird.

3.1 Vorgehensweise

Nach dem bestimmen der Schnittstellen mit **InterfaceFinder** ist eine mehrstufige Bewertung der Subsystemeinteilung möglich: Zunächst werden die Schnittstellen einzeln betrachtet. Dabei wird untersucht, welche Arten von Symbolen in der Schnittstelle auftreten und wie häufig diese vorkommen. Aus der Anzahl der einzelnen Symbole und deren Gesamtanzahl lassen sich bereits Aussagen über die Struktur des Systems oder auch den Grad der Objektorientierung ableiten. Als nächstes werden die Schnittstellen der Subsysteme miteinander verglichen, wofür inbesondere die Größe der Schnittstellen betrachtet wird. Daraus ergeben sich Erkenntnisse über die Subsystemeinteilung, z.B. lässt sich an der Größe der Schnittstellen erkennnen, wie gut die Subsysteme voneinander abgegrenzt sind. Schließlich kann die aktuelle Subsystemeinteilung als Ganzes betrachtet und mit anderen Einteilungen verglichen werden.

Damit die einzelnen Schnittstellen und Subsystemeinteilungen miteinander verglichen werden können, müssen anhand der ermittelten Schnittstellen und des Quelltextes einige Größen bestimmt werden. Für jedes Subsystem wird daher dessen Größe s_i in 1000 Lines of Code (kLOC), die Anzahl der Symbole in der Schnittstelle insgesamt (*absolute Schnittstellengröße*, bezeichnet als k_i) und die Anzahl der einzelnen Symbole, geordnet nach Typen, bestimmt, wobei $i = 1 \ldots n$ und n die Anzahl der Subsysteme ist.

Alle diese Größen lassen sich mit Hilfe der UNIX-Kommandos `grep` und `wc` aus den Quell- bzw. Schnittstellen-Export-Dateien bestimmen.

3.1.1 Betrachtung einzelner Schnittstellen

Anhand der Struktur einer Schnittstelle, das heißt der Häufigkeit der einzelnen Symboltypen in der Schnittstelle, lassen sich einige Erkenntnisse über das System gewinnen.

Bei objektorientierter Programmierung sollte die Kommunikation der einzelnen Programmteile über Klassen und deren Memberfunktionen erfolgen. Also sollten auch die Schnittstellen der Subsysteme möglichst nur Klassen (und Memberfunktionen) enthalten. Wenn hingegen viele Funktionen in den Schnittstellen auftreten, deutet dies auf einen nicht objektorientierten Programmierstil hin. Andere Subsysteme sollten auch nicht auf Membervariablen direkt zugreifen können, es sei denn, es handelt sich dabei um Konstanten.

Structs und Enums dienen in C++ zur Definition von eigenen Typen. Die Benutzung von Structs und Enums, auch in Schnittstellen, ist positiv zu bewerten. Es wird damit der Wertebereich von Variablen und Parametern eingeschränkt, was die Verständlichkeit erhöht und Fehler verringert.

Variable sollten nicht in verschiedenen Subsystemen benutzt werden, da dies die Verständlichkeit stark verschlechtert. Wenn Variable an mehreren Stellen benutzt werden, lässt sich nur schwer nachvollziehen wie darauf zugegriffen wird. Stattdessen können die Variablen in Klassen gepackt und mit entsprechenden Zugriffsfunktionen benutzt werden.

In C werden oft `#define`-Anweisungen benutzt, um Konstanten zu definieren. Dies sollte in C++ besser direkt mittels `const` erfolgen. Define-Anweisungen in Schnittstellen deuten daher meist auf einen schlechten Programmierstil hin.

Wenn in einer Schnittstelle alle Typen von Symbolen häufig vorkommen, ist dies ein Zeichen dafür, dass das Subsystem nicht gut entworfen wurde und schlecht abgegrenzt ist. Ein gut entworfenes Subsystem sollte einen deutlichen Schwerpunkt bei Klassen und deren Memberfunktionen (bei nicht objektorientierter Programmierung bei Funktionen) erkennen lassen.

3.1.2 Vergleich von Schnittstellen

Zum Vergleich der Schnittstellen der verschiedenen Subsysteme einer Einteilung wird vor allem die Größe der Schnittstellen betrachtet. Die absolute Größe einer Schnittstelle k_i ist für den Vergleich mit den Schnittstellen anderer Subsysteme noch wenig aussagekräftig. Daher wird diese Größe in Bezug zur Größe des Subsystems s_i gesetzt und so eine *relative Schnittstellengröße* r_i bestimmt:

$$r_i = \frac{k_i}{s_i}$$

Mit dieser relativen Schnittstellengröße r_i können die Subsysteme einer Einteilung untereinander verglichen werden. Prinzipiell gilt dabei, dass eine Schnittstelle und damit auch die relative Schnittstellengröße möglichst klein sein sollte.

Wenn die Subsystemeinteilung gut entworfen wurde, sollten für alle Subsysteme die Schnittstellen klein sein. Einzelne Subsysteme, die sehr stark von den Anderen abweichen, zeigen eine nicht einheitliche Vorgehensweise beim Entwurf der Subsysteme. Große Werte der relativen Schnittstellengröße und starke Abweichungen zwischen den Subsystemen deuten also auf eine schlechte Einteilung hin.

Es wäre auch denkbar, eine relative Schnittstellengröße als das Verhältnis der absoluten Schnittstellengröße zur Gesamtzahl der in dem Subsystem definierten Symbole zu bilden:

$$r'_i = \frac{k_i}{\text{Anzahl der Symbole im Subsystem } i}$$

Die Anzahl der definierten Symbole im Subsystem kann aber nicht so einfach ermittelt werden, dafür würde ein entsprechendes Tool benötigt.

3.1.3 Vergleich von Subsystemeinteilungen

Ziel soll es sein, auch etwas über die Güte einer Subsystemeinteilung sagen zu können, also die Einteilung als Ganzes zu bewerten. Dies würde dann auch einen Vergleich verschiedener Einteilungen ermöglichen.

Dazu muss eine Größe gefunden werden, mit der alle Subsysteme zusammengefasst werden. Eine solche Größe, genannt M erhält man z.B. indem man die absolute Schnittstellengröße über alle Subsysteme aufsummiert und durch die Gesamt-LOC teilt:

$$M = \frac{\sum_{i=1}^{n} k_i}{\sum_{i=1}^{n} s_i}$$

Eine Subsystemeinteilung würde man damit um so besser bewerten, je kleiner M ist. Dabei hat allerdings die Größe der Subsysteme einen starken Einfluß auf M: Ein sehr großes Subsystem mit kleiner Schnittstelle kann mehrere kleine Subsysteme dominieren. Hingegen wird die Güte der einzelnen Schnittstellen zu wenig berücksichtigt.

Eine andere Möglichkeit besteht darin, statt M eine Größe R als *Durchschnitt der relativen Schnittstellengröße* zu bilden:

$$R = \frac{\sum_{i=1}^{n} r_i}{n}$$

Mit R werden alle Subsysteme, unabhängig von der Größe, als gleichwertig angesehen und somit stärker die einzelnen Schnittstellen bewertet. Dabei besteht allerdings die Gefahr, dass kleine Subsysteme überbewertet werden.

Da zur Bewertung einer Subsystemeinteilung die Schnittstellen aller Subsysteme einbezogen werden sollten und die Größe der Subsysteme dabei weniger eine Rolle spielt, ist die Größe R zur Bewertung einer Subsystemeinteilung besser geeignet als M. In den folgenden Beispielen wird daher R als Bewertunggröße verwendet.

Was noch fehlt ist ein Bezugswert, um bei einem Wert von R sagen zu können, wie „gut“ dieser ist. Im Allgemeinen ist natürlich ein Wert um so „besser“, je kleiner dieser ist, da ja die Schnittstellen möglichst klein sein sollten. Doch muss zunächst einmal festgelegt werden, was ein kleiner bzw. großer Wert ist.

Wenn man davon ausgeht, dass eine Subsystemeinteilung das System strukturieren soll und die Subsysteme dabei möglichst stark voneinander abgegrenzt werden, so sollte bei einer völlig unstrukturierten Subsystemeinteilung ein besonders „schlechter“, großer Wert herauskommen. Also kann man diesen Wert als Bezugswert nehmen, eine Subsystemeinteilung ist dann um so „besser“, je stärker sie von diesem Bezugswert nach unten abweicht. Daher muss also zunächst eine unstrukturierte Einteilung gefunden werden. Wenn man dazu die Dateien zufällig den Subsystemen zuordnet sollte dies eine unstrukturierte Einteilung ergeben. Der Mittelwert von mehreren zufälligen Einteilungen bildet dann den Bezugswert. Bei den zufälligen Einteilungen muss darauf geachtet werden, dass die Anzahl der Subsysteme die gleiche ist wie bei der später zu vergleichenden Subsystemeinteilung, da die Anzahl der Subsysteme die Größe der Schnittstellen direkt beeinflußt. Daher können auf diese Weise nur verschiedene Subsystemeinteilungen verglichen werden, die die gleiche Anzahl von Subsystemen haben.

3.1.4 Darstellung

Die Ergebnisse der Anwendung von **InterfaceFinder** werden jeweils in einer Tabelle, wie z.B. Tabelle 3.1 auf Seite 43, dargestellt. Für jedes Subsystem wird eine Zeile

verwendet: In der ersten Spalte stehen die Namen der Subsysteme. Die nächsten neun Spalten unter „Anzahl der Symbole“ zeigen jeweils die Anzahl der einzelnen Symbole der verschiedenen Typen in der Schnittstelle des Subsystems. Die Symbole werden dabei so gezählt, wie sie in der Schnittstellen-Export-Datei auftreten, das heißt jede Zeile der Export-Datei wird als ein Vorkommen gezählt. Für Klassen wird dabei auch das Auftreten von Membern gezählt, wobei ein Eintrag bei Membern einen Eintrag bei Klassen voraussetzt.

Die letzten drei Spalten der Tabelle enthalten die absolute Schnittstellengröße k_i, die Subsystemgröße s_i und die relative Schnittstellengröße $r_i = \frac{k_i}{s_i}$. In der letzten Tabellenzeile „gesamt“ werden die darüberliegenden Zeilen aufsummiert, in der letzten Spalte ist hier der Durchschnitt der relativen Schnittstellengröße R gebildet.

Die von **InterfaceFinder** gefundenen Subsystemschnittstellen der folgenden Anwendungsbeispiele, sind jeweils in den Export-Dateien auf der beigelegten CD-ROM im Verzeichnis `Examples` zu finden. Dort befinden sich auch verschiedene Varianten des XCTL-Systems bei denen die Verzeichnisstruktur den untersuchten Subsystemeinteilungen entspricht.

3.2 Anwendung auf das XCTL-System

Als erstes soll die aktuelle Version des XCTL-Systems untersucht werden. Die verwendete Version wurde am 18. Juni 2003 dem CVS entnommen.

3.2.1 Normale Subsystemeinteilung

Im ersten Schritt wird die durch die Verzeichnisstruktur gegebene Subsystemeinteilung* verwendet. Dabei sind die 113 Quelltextdateien auf 16 Subsysteme unterschiedlicher Größe aufgeteilt. Die Ergebnisse sind in Tabelle 3.1 dargestellt.

Bei den Ergebnissen fällt zunächst auf, dass die Größe und die Struktur(Typ der Symbole) der Schnittstellen der einzelnen Subsysteme recht unterschiedlich ist. So bieten einige Subsysteme ihre Funktionalität nur, oder zumindest hauptsächlich, über Klassen an (AUTOJUST, SWINTRAC, TOPOGRFY, MANJUST, HARDWARE, MESPARA, DIFRKMTY, WORKFLOW, DATAVISA und DETECUSE), während bei anderen Subsystemen (PROTOCOL, INTERNLS, UTILS, MOTORSTRG) überwiegend Funktionen benutzt werden.

Bei objektorientierter Programmierung sollten die Subsystme ihre Funktionalität möglichst durch Klassen zur Verfügung stellen. Im diesen Sinne fällt hier das Subsystem PROTOCOL besonders negativ auf, da hier ausschließlich Funktionen verwendet werden, obwohl dieses Subsystem neu entwickelt wurde und somit ein konsequent objektorientierter Entwurf möglich gewesen wäre. Bei den anderen drei Subsystemen (INTERNLS, UTILS, MOTORSTRG) werden zumindest nicht ausschließlich Funktionen benutzt. Die von INTERNLS und UTILS benutzten Funktionen sind Hilfsfunktionen für die es sich nicht unbedingt anbietet, eine Klasse zu entwerfen und für die Subsysteme INTERNLS und MOTORSTRG ist die vielfache Verwendung von Funktionen auch eine Folge der ursprünglich nicht vollständig objektorientierten Entwicklung des XCTL-Systems.

Die Benutzung von Variablen in verschiedenen Subsystemen ist kritisch zu sehen. Die Variablen werden mittels `extern`-Deklarationen in anderen Subsystemen sichtbar gemacht, ohne das dies in den Headerdateien, die die Subsystemschnittstelle enthalten sollen, zu erkennen ist. Besser wäre es diese Variablen in Klassen zu packen und nur über diese Klassen darauf zuzugreifen.

Immerhin werden, wie in der Tabelle sichtbar, nur relativ wenige Variablen benutzt. Beim Subsystem MOTRSTRG werden nicht wirklich 19 Variablen verwendet,

*im Weiteren als *normale* Einteilung bezeichnet

Subsystem	Anzahl der Symbole									k_i	s_i	r_i
	CLASS	STRUCT	FUNCTION	ENUM	TYPEDEF	VARIABLE	DEFINE	MEMBER VAR	MEMBER FCT			
PROTOCOL			57			2				59	8,096	7,3
AUTOJUST	2									2	2,476	0,8
WINRESRC							714			714	1,239	576,3
HELP							7			7	0,122	57,4
SWINTRAC	8					4	4	1	23	40	2,104	19,0
TOPOGRFY	5					1			1	7	2,682	2,6
INTERNLS	2	1	23	3	3	5	21	24	12	94	2,943	31,9
MANJUST	3									3	4,577	0,7
HARDWARE	1								4	5	0,175	28,6
MESPARA	2						1		16	19	0,492	38,6
UTILS	3		17			1	2		4	27	1,298	20,8
MOTRSTRG	2		49	3	1	19				74	6,502	11,4
DIFRKMTY	4	1				6		27	5	43	8,167	5,3
WORKFLOW	2	2		3		1		6	19	33	4,072	8,1
DATAVISA	4	4		2	3		3	40	29	85	5,520	15,4
DETECUSE	6		2	2		1			24	35	7,019	5,0
gesamt	44	8	148	13	7	40	752	98	137	1247		R: 51, 8
gesamt[a]	44	8	148	13	7	40	31	98	137	526		R: 14, 0

[a] ohne WINRESRC und HELP

Tabelle 3.1: Ergebnisse XCTL-System(normale Subsystemeinteilung)
k_i: Gesamtanzahl der Symbole in der Schnittstelle,
s_i: Größe des Subsystems in 1000 LOC,
r_i: relative Schnittstellengröße: $r_i = \frac{k_i}{s_i}$
R: Durchschnitt der rel. Schnittstellengröße

SNiFF+ kann hier Variablen mit gleichen Namen im Subsystem DIFRKMTY nicht unterscheiden (siehe auch Abschnitt 2.6.1 XCTL/MOTORSTRG auf Seite 32).

Wenn man sich die relative Schnittstellengröße r_i in der letzten Tabellenspalte anschaut, findet man für WINRESRC und HELP die größten Werte. Die Dateien in diesen Verzeichnissen stellen aber keine Funktionalität, sondern nur einige `#define`'s zur Verfügung und bilden daher kein richtiges Subsystem. Deshalb werden diese „Subsysteme“ hier nicht weiter beachtet. Für die anderen Subsysteme kann man erkennen, dass AUTOJUST und MANJUST sehr kleine Schnittstellen haben, während INTERNLS, HARDWARE und MESPARA größere Schnittstellen aufweisen.

Die großen Werte für die relative Schnittstellengröße ergeben sich bei HARDWARE und MESPARA dadurch, dass die Subsysteme sehr klein sind (wenige LOC). Die Schnittstellen bestehen in beiden Subsystemen aber nur aus einer, bzw. zwei, Klassen, sind also eigentlich doch recht klein und sollen daher hier nicht negativ bewertet werden. Da alle anderen Subsysteme wesentlich größer sind, stellt sich die Frage, ob es überhaupt sinnvoll ist HARDWARE und MESPARA als eigenständige Subsysteme zu führen. Eine Antwort darauf, lässt sich aber nur geben, wenn man weiß, nach welchen Kriterien die Subsysteme gebildet wurden. Daher muss auch ein großer Wert bei der Schnittstellengröße nicht zwangsläufig auf schlechtes Design hinweisen, auch wenn im Allgemeinen kleinere Subsystemschnittstellen wünschenswert sind.

Unter Nichtbeachtung von HELP und WINRESRC ergibt sich für die Schnittstellengröße der Durchschnittswert R zu 14,0. Um mit dieser Zahl etwas anfangen zu können und einen Vergleich zu ermöglichen, wird im nächsten Abschnitt der Wert für eine zufällige Subsystemeinteilung als Bezugswert bestimmt.

Wenn man sich die in den Subsystemschnittstellen enthaltenen Symbole im Einzelnen anschaut (siehe Export-Dateien), fällt auf, dass das in [Thiel] angestrebte Prinzip der Darstellung der Subsystemschnittstellen in entsprechenden Subsystem-Headerdateien nicht mehr eingehalten wird. So gehören zum Beispiel beim Subsystem SWINTRAC Klassen aus `M_DLG.H` zur Subsystemschnittstelle (`TCounterWindow, TExecuteCmdDlg, TGetDataDlg`). In anderen Subsystemen (TOPOGRFY, MANJUST) ist ein Teil der Schnittstelle in einer Datei `*_GUI.H` enthalten, weitere Subsysteme haben keine Subsystem-Headerdatei (z.B. HARDWARE). In der Datei `MOTRSTRG.H` werden weitere Headerdateien inkludiert, die Teile zur Subsystemschnittstelle beisteuern.

Das Konzept, die Schnittstellen in einer Headerdatei je Subsystem unterzubringen, stellt nicht unbedingt eine optimale Lösung dar. Damit war aber wenigstens ein einheitliches Prinzip vorhanden, bei dem die Schnittstellen leicht erkennbar sind, was besonders wichtig ist, wenn Modifikationen am Programm durch verschiedene Entwickler vorgenommen werden. Daher wäre es wünschenswert, wenn für die Darstellung der Subsystemschnittstellen wieder zu einem einheitlichen Konzept für alle Subsysteme zurückgefunden werden würde.

3.2.2 Zufällige Subsystemeinteilung

Zum Vergleich zur normalen Subsystemeinteilung wird nun eine zufällige Einteilung der Subsysteme gewählt. Um die verschiedenen Einteilungen vergleichen zu können, muss die Anzahl der Subsysteme beibehalten werden. Es werden also 14 Subsysteme angelegt und die Quelldateien (ohne die Dateien in HELP, WINRESRC, PORTING) zufällig auf diese Subsysteme verteilt, indem für jede Datei eine Zufallszahl zwischen 1 und 14 generiert wird. Die so entstandene Einteilung ist in Tabelle 3.2 dargestellt und die Ergebnisse der Schnittstellenbestimmung befinden sich in Tabelle 3.3.

Offensichtlich hat sich die Struktur der Schnittstellen gegenüber der normalen Einteilung geändert. Während bei der normalen Einteilung sich für die meisten

1	2	3	4
L_LAYER.H	AUTOJUST.H	M_DEVICE.CPP	detecmes.cpp
MESPARA.H	DETECGUI.CPP	M_JUSTAG.CPP	L_LAYER.CPP
MESPARAW.CPP	DETECGUI.H	MJ_GUI.H	m_dlgdif.cpp
MJ_OLD.CPP	HWIO.CPP	PRTOPARA.H	MATRIX.H
MOTRSTRG.H	MATRIX.CPP	Range.h	MSIMSTAT.H
PRDIPARA.H	PROTDIF.CPP		PRDIPRNW.H
PROCOMBW.CPP	PROTOW.CPP		
PROCOMBW.H	PROTPARW.CPP		
PROPRNW.H	PROTPASW.CPP		
PROTPARW.H	PROTTOPW.H		
TRANSFRM.H	TP_GUI.H		

5	6	7	8
C_8X2.H	Detec_hw.cpp	HWIO.H	P_LANG.H
EVRYTHNG.H	detecmes.h	IEEE.H	PROTDIFW.CPP
M_DLG.H	M_ARSCAN.H	M_LAYER.CPP	PROTTOP.H
M_DLGDIF.H	M_LAYER.H	MJ_OFUNK.CPP	U_TIMER.H
MESPARA.CPP	MJ_OLD.H	PROTDIFW.H	WORKFLOW.H
MJ_OFUNK.H	PROTO.H	PRTOPRNW.H	
MJ_OGUI.CPP	PROTTOPW.CPP	TRANSFRM.CPP	
MJ_OGUI.H			
U_VALUES.CPP			

9	10	11	12
M_DATA.H	detec_hw.h	DIFRKMTY.H	DLG_TPL.CPP
MESPARAW.H	PROTPASW.H	dllmain.cpp	M_JUSTAG.H
MOTORS.CPP	PRTOPRNW.CPP	M_CURVE.CPP	M_MOTCOM.H
MSIMSTAT.CPP	TP_GUI.CPP	M_TOPO.CPP	M_SCAN.H
PROPRNW.CPP		MJ_FUNK.H	M_STEERG.H
PROTDIF.H		PRDIPRNW.CPP	MJ_FUNK.CPP
TOPOGRFY.H		PROTOCOL.H	P_LANG.CPP
TP_FUNK.CPP		PROTTOP.CPP	PROTO.CPP
TP_FUNK.H		U_TIMER.CPP	PROTOCOL.CPP
U_VALUES.H			U_FILES.CPP
			U_FILES.H

13	14
DATAVISA.H	M_ARSCAN.CPP
detecuse.cpp	M_MOTHW.H
DETECUSE.H	
M_DATA.CPP	
M_DLG.CPP	
M_MAIN.CPP	
M_SCAN.CPP	
M_STEERG.CPP	
MJ_GUI.CPP	
PROTOW.H	
SWINTRAC.H	
U_UTILS.H	

Tabelle 3.2: zufällige Subsystemeinteilung

Subs.	Anzahl der Symbole											
	CLASS	STRUCT/UNION	FUNCTION	ENUM	TYPEDEF	VARIABLE	DEFINE	MEMBER VAR	MEMBER FCT	k_i	s_i	r_i
1	5	1		5	2	2	4	50	19	88	1,775	49,6
2	8	2		1				61	28	100	3,514	28,5
3	3		1	3		4		12		23	1,666	13,8
4	5		28			4		10	3	50	4,448	11,2
5	12	2	9	3	3	23	21	102	40	215	1,936	111,1
6	10			3	4	1	6	83	26	133	2,673	49,8
7	3		45	1	1	2		15	42	109	1,995	54,6
8	10	2		4				74	6	96	1,577	60,9
9	10		2	1		16	2	65	80	176	6,100	28,9
10	3			3				12	2	20	1,161	17,2
11	5	1	2	2	3	5		77	30	125	3,625	34,5
12	21	1	71	4	2	65	1	119	135	419	5,959	70,3
13	17	4		9	3	12	19	113	121	298	16,032	18,6
14	10	1				4		63	16	94	3,662	25,7
ges.	122	14	158	39	18	138	53	856	548	1946		R: 41, 1

Tabelle 3.3: Ergebnisse XCTL-System (zufällige Subsystemeinteilung)

Schnittstellen ein Schwerpunkt entweder bei Klassen oder Funktionen ausmachen lässt, ist dies bei der zufälligen Einteilung nicht mehr möglich. Die Schnittstellen sind weitaus weniger strukturiert, so wie es für die zufällige Einteilung auch gewollt ist, um aus dieser Einteilung einen Bezugswert für R zu erhalten.

Desweiteren sind die Abweichungen bei der absoluten Schnittstellengröße k_i zwischen den einzelnen Subsystemen geringer als bei der normalen Einteilung, was bei einer zufälligen Zuordnung aber auch zu erwarten ist. Die stärkeren Abweichungen bei der normalen Subsystemeinteilung können auch darauf hinweisen, dass bei der Entwicklung der Subsysteme nicht immer einheitlich vorgegangen wurde, oder deren Funktion verschieden ist.

Die Anzahl der in den Subsystemschnittstellen enthaltenen Symbole ist bei der zufälligen Subsystemeinteilung größer als bei der normalen Einteilung. Es werden insgesamt mehr als dreimal so viele Symbole über Subsystemgrenzen hinweg benutzt. Während bei den meisten Symboltypen sich die Anzahl etwa verdoppelt bis verdreifach hat, ist bei Funktionen aber nur ein geringer Unterschied zu sehen (158 statt 148).

Was kann das bedeuten? Wenn man sich die verwendeten Funktionen anschaut (siehe Export-Dateien) wird erkennbar, dass 138 dieser Funktionen bei beiden Subsystemeinteilungen auftreten. Wenn Funktionen weit verbreitet im gesamten XCTL verwendet würden, hätte sich deren Anzahl in den Schnittstellen bei der zufälligen Einteilung auch erhöhen müssen. Da dies nicht der Fall ist, lässt sich der Schluss ziehen, dass einerseits Funktionen nicht weit verbreitet genutzt, sondern hauptsächlich nur die gefundenen 138, die bei der normalen Subsystemeinteilung offenbar speziell für Schnittstellenaufgaben angeboten werden. Andererseits bedeutet das, dass, abgesehen von diesen Funktionen, die Kommunikation verschiedener Programmteile über Klassen erfolgen muss, und tatsächlich hat sich ja die Anzahl der Klassen in den Schnittstellen bei der zufälligen Einteilung auch fast verdreifacht. Bestätigt wird dies auch durch die sehr hohe Anzahl von Membervariablen und Memberfunktionen bei der zufälligen Subsystemeinteilung: die Zahl der Memberfunktionen hat sich vervierfacht, die der Membervariablen gar mehr als verachtfacht. Die überwiegende Verwendung von Klassen statt Funktionen ist positiv zu sehen und zeigt, dass das XCTL-System zumindest in diesem Punkt objektorientiert ist.

Wenn man sich die relative Schnittstellengröße r_i in der letzten Spalte der Tabelle anschaut, fallen auch hier zunächst die größeren Werte gegenüber der normalen Subsystemeinteilung auf. Der Durchschnittswert R, der ja als Bezugswert genutzt werden soll, liegt hier bei 41,1. Zwei weitere zufällige Einteilungen brachten ähnlich große Werte von 47,5 und 53,7. Gegenüber der normalen Einteilung hat sich dieser Wert also fast verdreifacht. Daraus lässt sich folgern, dass die normale Einteilung eine Programmstruktur bildet, bei der die Subsystemschnittstellen klein sind, die Subsysteme also gut abgegrenzt sind. Die normale Einteilung kann also grundsätzlich positiv bewertet werden.

3.2.3 Drei-Schichten-Einteilung

Es wird nun versucht das XCTL-System in drei Schichten einzuteilen: Eine unterste Schicht, HARDWARE, für Programmteile mit Hardwarezugriffen (Motor- und Detektorsteuerung), eine mittlere Schicht FUNKTIONALITÄT, welche die auf der Hardwareschicht aufbauende Funktionalität enthält und schließlich eine GUI-Schicht mit der Benutzeroberfläche.

Solch eine Einteilung ist hilfreich zur Untersuchung der Programmarchitektur, sie ermöglicht festzustellen, inwieweit eine Schichtenarchitektur vorliegt oder wie gut einzelne Schichten abgrenzbar sind; sie dient aber nicht als Alternative zur normalen Einteilung, da sich eine Subsystemeinteilung im Allgemeinen an Anwendungsfällen

bzw. Funktionalität orientieren sollte, wofür eine Drei-Schichten-Einteilung meist nicht ausreichend ist.

Die Zuordnung der Dateien zu den einzelnen Schichten kann zum Teil nach dem Dateinamen erfolgen (Dateien mit „GUI“, „FUNK“ bzw. „HW“ im Namen), bei anderen Dateien muss der Inhalt beachtet werden, wobei gilt, dass Klassen, die für Motor- oder Detektorzugriff zuständig sind, zur HARDWARE-Schicht gezählt werden. Alle Klassen die Dialoge realisieren („`*Dlg`“), kommen zur GUI-Schicht, alle anderen zur FUNKTIONALITÄTs-Schicht.

Dabei ist eine eindeutige Zuordnung einer Datei zu einer Schicht nicht immer möglich, da manche Dateien Elemente von verschiedenen Schichten enthalten. In solchen Fällen wird zunächst die Schicht ausgewählt, zu der die überwiegende Anzahl der Symbole gehört, verschiedenen Zuordnungen ausprobiert und nach der besten Möglichkeit gesucht.

Durch solche „schichtübergreifenden“ Dateien ist eine scharfe Abgrenzung der Schichten nicht möglich, was sich auch auf die Schnittstellen auswirken wird: Eventuell werden Symbole in den Schnittstellen auftreten, die eigentlich zu einer anderen Schicht gehören. Um dies zu vermeiden müssten die Symbole aus solchen Dateien auf mehrere Dateien aufgeteilt werden.

Die Übersicht über die Einteilung gibt Tabelle 3.4 und die Ergebnisse sind in Tabelle 3.5 dargestellt.

Die Werte für die Schnittstellengröße können hier nicht mit den Werten der normalen Einteilung verglichen werden, da hier nur noch drei, dafür aber sehr große, Subsysteme gebildet sind. Insbesondere das Subsystem FUNKTTIONALITÄT ist extrem groß, es umfaßt mehr als die Hälfte des XCTL-Systems (gemessen an LOC). Um Vergleichswerte zu bekommen, wurde für verschiedene zufällige Einteilungen mit drei Subsystemen die durchschnittliche Schnittstellengröße ermittelt (siehe Tabelle 3.6). Die Einteilungen A und B wurden wie die zufälligen Einteilungen im vorigen Abschnitt erzeugt, man erhält bei beiden ähnliche Durchschnittstwerte R von ca. 32. Bei den Einteilungen C und D wurde die Größe der Subsysteme wie bei der Schichteneinteilung gewählt, indem bei Einteilung C die Anzahl der Dateien und bei Einteilung D die Anzahl der Quelltextzeilen der Subsysteme den jeweiligen Werten der Schichteneinteilung entsprechen.

Die Schichteneinteilung hat im Vergleich zu diesen zufälligen Einteilungen wesentlich kleinere Schnittstellen, im Durchschnitt nur ein Drittel so groß. Besonders klein sind die Schnittstellen der GUI- und HARDWARE-Subsysteme.

Auffällig ist, das die Schnittstelle der FUNKTIONALITÄT Symbole aller Typen enthält, also völlig unstrukturiert ist. Dies ist auch eine Eigenschaft der zufälligen Subsystemeinteilungen, könnte hier aber auch eine Folge der oben beschriebenen unscharfen Abgrenzung der Schichten sein. Die Schnittstellen der Schichten GUI und HARDWARE sind dagegen besser strukturiert, insbesondere die GUI-Schicht ist gut abgegrenzt und wird fast nur über Klassen angesprochen. Die FUNKTIONALITÄTS-Schicht als einzelnes Subsystem zu betrachten, scheint hingegen keine optimale Lösung zu sein, eventuell wäre eine weitere Aufteilung dieser Schicht günstiger.

3.2.4 Fazit

Als Fazit des Vergleichs lässt sich zunächst festhalten, dass es möglich ist, die Subsystemeinteilungen nur durch Untersuchung der Struktur und Größe der ermittelten Schnittstellen zu bewerten und sich Aussagen über das Programmdesign ableiten lassen.

Dabei zeigt sich für das XCTL-System, dass die normale Subsystemeinteilung als günstige Einteilung zu bewerten ist. Die Schnittstellen sind immer kleiner als bei der zufälligen Einteilung, was zeigt, dass hier Elemente lokal bleiben, und Funktionalität

GUI			
AUTOJUST.H	MJ_OLD.H	PROCOMBW.H	PROTTOPW.CPP
DETECGUI.CPP	M_ARSCAN.H	PROPRNW.CPP	PROTTOPW.H
DETECGUI.H	M_DEVICE.CPP	PROPRNW.H	PRTOPRNW.CPP
DLG_TPL.CPP	M_DLG.CPP	PROTDIFW.CPP	PRTOPRNW.H
MESPARAW.CPP	M_DLG.H	PROTDIFW.H	SWINTRAC.H
MESPARAW.H	M_DLGDIF.H	PROTOW.CPP	TP_GUI.CPP
MJ_GUI.CPP	M_JUSTAG.CPP	PROTOW.H	TP_GUI.H
MJ_GUI.H	M_SCAN.H	PROTPARW.CPP	m_dlgdif.cpp
MJ_OGUI.CPP	PRDIPRNW.CPP	PROTPARW.H	
MJ_OGUI.H	PRDIPRNW.H	PROTPASW.CPP	
MJ_OLD.CPP	PROCOMBW.CPP	PROTPASW.H	
FUNKTIONALITÄT			
DATAVISA.H	MSIMSTAT.CPP	PROTDIF.CPP	TP_FUNK.CPP
DIFRKMTY.H	MSIMSTAT.H	PROTDIF.H	TP_FUNK.H
EVRYTHNG.H	M_ARSCAN.CPP	PROTO.CPP	TRANSFRM.CPP
L_LAYER.CPP	M_CURVE.CPP	PROTO.H	TRANSFRM.H
L_LAYER.H	M_DATA.CPP	PROTOCOL.CPP	U_FILES.CPP
MATRIX.CPP	M_DATA.H	PROTOCOL.H	U_FILES.H
MATRIX.H	M_JUSTAG.H	PROTTOP.CPP	U_TIMER.CPP
MESPARA.CPP	M_MAIN.CPP	PROTTOP.H	U_TIMER.H
MESPARA.H	M_SCAN.CPP	PRTOPARA.H	U_UTILS.H
MJ_FUNK.CPP	M_STEERG.CPP	P_LANG.CPP	U_VALUES.CPP
MJ_FUNK.H	M_STEERG.H	P_LANG.H	U_VALUES.H
MJ_OFUNK.CPP	M_TOPO.CPP	Range.h	WORKFLOW.H
MJ_OFUNK.H	PRDIPARA.H	TOPOGRFY.H	dllmain.cpp
HARDWARE			
C_8X2.H	HWIO.H	M_LAYER.CPP	detec_hw.h
DETECUSE.H	IEEE.H	M_LAYER.H	detecmes.cpp
Detec_hw.cpp	MOTORS.CPP	M_MOTCOM.H	detecmes.h
HWIO.CPP	MOTRSTRG.H	M_MOTHW.H	detecuse.cpp

Tabelle 3.4: 3-Schichten-Einteilung

Subsystem	Anzahl der Symbole											
	CLASS	STRUCT	FUNCTION	ENUM	TYPEDEF	VARIABLE	DEFINE	MEMBER VAR	MEMBER FCT	k_i	s_i	r_i
GUI	29	1	1			6	4	38	10	89	13,751	6,5
FUNK.	20	4	46	12	5	25	17	48	305	482	30,196	16,0
HARDWARE	9		52	6	6	2			38	113	12,176	9,3
gesamt	58	5	99	18	11	33	21	86	353	684		$R : 10,6$

Tabelle 3.5: Ergebnisse 3-Schichten-Einteilung

Einteilung	Subsystem	k_i	s_i	$r_i = \frac{k_i}{s_i}$	R
A	1	396	20,655	19,2	33,1
	2	732	11,483	63,7	
	3	391	23,985	16,3	
B	1	630	25,574	24,6	31,7
	2	661	15,109	43,7	
	3	414	15,440	26,8	
C	1	463	20,539	22,5	24,5
	2	758	24,972	30,4	
	3	218	10,612	20,5	
D	1	426	13,367	31,9	24,0
	2	462	30,722	15,0	
	3	302	12,034	25,1	

Tabelle 3.6: zufällige Einteilungen mit drei Subsystemen

in den Subsystemen gekapselt ist. Auch ist ein gewisser Grad der Objektorientierung erkennbar. Die Drei-Schichten-Einteilung macht deutlich, dass zumindest eine GUI-Schicht abgrenzbar ist.

Offen bleibt, ob die normale Subsystemeinteilung noch zu verbessern oder eine völlig andere Einteilung günstiger wäre. Da Subsystemeinteilungen immer einem logischen Konzept entsprechen sollten, und so Subsysteme zum Beispiel eine bestimmte Teilfunktionalität des Systems, bzw. einen Anwendungsfall, repräsentieren, muss bei der Zuordnung der Dateien zu Subsystemen ein Verständnis der Funktionalität der Dateien vorhanden sein. Maße wie Art und Größe der Schnittstellen stellen zwar eine Hilfe zur Bewertung da, geben aber keine Auskunft, ob die gewählte Subsystemeinteilung sinnvoll ist in Bezug auf die Funktionalität.

3.3 Anwendung auf ein anderes System

Nun soll auch gezeigt werden, wie **InterfaceFinder** auf ein anderes Programm angewendet werden kann und bei der Bildung von Subsystemen hilfreich ist. Als Beispiel dient das Programm Tinyxml, ein kleiner XML-Parser in C++ (siehe [TinyXml]). Das Programm liegt in 7 Quelldateien, ohne Subsystemeinteilung, vor:

tinyxml.cpp
tinyxml.h
tinyxmlparser.cpp
tinyxmlerror.cpp
tinystr.cpp
tinystr.h
xmltest.cpp

Das Beispiel Tinyxml ist mit sieben Quelltextdateien und zusammen ca. 4000 Zeilen (LOC) sehr klein, dafür aber sehr anschaulich. Trotz des geringen Umfangs ist es damit möglich, die Anwendung von **InterfaceFinder** zu demonstrieren.

Da die Dateinamen recht aussagekräftig sind, kann versucht werden, daraus auf die Funktion zu schließen. So wird vermutlich „tinystr“ mit Stringverarbeitung zu tun haben. Daher wird zunächst ein Subsystem „STRING“ mit den Dateien `tinystr.h` und `tinystr.cpp` gebildet und die restlichen Dateien in einem Subsystem „TINY“ zusammengefasst:

TINY	STRING
tinyxml.cpp	tinystr.cpp
tinyxml.h	tinystr.h
tinyxmlparser.cpp	
tinyxmlerror.cpp	
xmltest.cpp	

Die Exportdatei nach der Anwendung von **InterfaceFinder** sieht folgendermaßen aus:

```
----------------------------------------
Interface for subsystem: STRING

 * class TiXmlString
 * class TiXmlOutStream
 * member function TiXmlString::append
 * enum TiXmlString::{anon:tinystr.h:158}
 * member function TiXmlString::reserve
```

Das Subsystem „TINY“ besitzt also keine Subsystemschnittstelle, und die von „STRING“ besteht aus zwei Klassen. Dies zeigt, das „STRING“ als eigenständige Komponente betrachtet werden kann: „STRING“ ist gegenüber den anderen Komponenten stark abgegrenzt und stellt nur über die beiden Klassen `TiXmlString` und `TiXmlOutStream` Funktionalität bereit.

Als Nächstes wird versucht ein weiteres Subsystem zu bilden. Dazu werden die Dateien `tinyxml.h` und `tinyxml.cpp` in einem Subsystem „XML“ zusammengefasst:

TINY	STRING	XML
tinyxmlparser.cpp	tinystr.cpp	tinyxml.cpp
tinyxmlerror.cpp	tinystr.h	tinyxml.h
xmltest.cpp		

Das Ergebnis ist dann Folgendes:

```
----------------------------------------
Interface for subsystem: TINY

 * member function TiXmlDocument::Parse
 * member function TiXmlBase::GetEntity

----------------------------------------
Interface for subsystem: STRING

 * class TiXmlString
 * class TiXmlOutStream
 * member function TiXmlString::append
 * enum TiXmlString::{anon:tinystr.h:158}
 * member function TiXmlString::reserve

----------------------------------------
Interface for subsystem: XML

 * define TIXML_STRING
 * define TIXML_LOG
 * class TiXmlDocument
```

```
 * member function TiXmlDocument::LoadFile
 * class TiXmlElement
 * class TiXmlNode
 * class TiXmlText
 * member function TiXmlNode::InsertAfterChild
 * member function TiXmlNode::InsertEndChild
 * member function TiXmlNode::InsertBeforeChild
 * class TiXmlComment
 * member function TiXmlElement::SetAttribute
 * member function TiXmlNode::RemoveChild
 * class TiXmlBase
 * member variable TiXmlBase::condenseWhiteSpace
 * member function TiXmlBase::GetChar
 * enum TiXmlBase::{anon:tinyxml.h:224}
 * member variable TiXmlBase::entity
 * enum TiXmlBase::{anon:tinyxml.h:196}
 * member function TiXmlDocument::SetError
 * member variable TiXmlNode::value
 * class TiXmlUnknown
 * member variable TiXmlAttribute::value
 * class TiXmlAttribute
 * member variable TiXmlAttribute::document
 * member variable TiXmlAttribute::name
 * member function TiXmlNode::LinkEndChild
 * member function TiXmlBase::Parse
 * member variable TiXmlDeclaration::standalone
 * class TiXmlDeclaration
 * member variable TiXmlDeclaration::encoding
 * member variable TiXmlDeclaration::version
 * member variable TiXmlNode::parent
 * member function TiXmlAttribute::SetDocument
```

Die Schnittstelle von „XML“ ist relativ umfangreich und unübersichtlich, zusätzlich hat jetzt auch „TINY“ eine Schnittstelle. Dies bedeutet, das zwischen Dateien aus „TINY“ und „XML“ enge Beziehungen bestehen und somit „XML“ keine gut abgegrenzte Komponente darstellt. Zur möglich Verbesserung wird als Nächstes versucht die Datei `tinyxmlparser.cpp` mit in das Subsystem „XML“ aufzunehmen:

TINY	STRING	XML
tinyxmlerror.cpp	tinystr.cpp	tinyxml.cpp
xmltest.cpp	tinystr.h	tinyxml.h
		tinyxmlparser.cpp

Dann erhält man diese Schnittstellen:

```
----------------------------------------
Interface for subsystem: XML

 * class TiXmlDocument
 * member function TiXmlDocument::LoadFile
 * class TiXmlElement
 * class TiXmlNode
 * member function TiXmlDocument::Parse
 * class TiXmlText
 * member function TiXmlNode::InsertAfterChild
```

```
 * member function TiXmlNode::InsertEndChild
 * member function TiXmlNode::InsertBeforeChild
 * class TiXmlComment
 * member function TiXmlElement::SetAttribute
 * member function TiXmlNode::RemoveChild

----------------------------------------
Interface for subsystem: STRING

 * class TiXmlString
 * class TiXmlOutStream
 * member function TiXmlString::append
 * enum TiXmlString::{anon:tinystr.h:158}
 * member function TiXmlString::reserve
```

Die Schnittstelle von „XML“ ist nun wesentlich kleiner geworden und „TINY“ hat wieder keine Schnittstelle. Somit scheint dies eine günstigere Einteilung als die vorherige zu sein.

Der Versuch `xmltest.cpp` mit in das Subsystem „XML“ aufzunehmen, führt dazu, dass sowohl „XML“ als auch „TINY“ keine Subsystemschnittstellen mehr haben. Ein Blick in die Datei `xmltest.cpp` zeigt, das diese ein Testprogramm für TinyXML ist, und daher natürlich Funktionalität der anderen „XML“-Dateien abfragt. Wenn man in `tinyxmlerror.cpp` hineinschaut, erkennt man, dass diese Datei zur „XML“ Komponente gehört.

Zusammenfassend lässt sich also eine Aufteilung in drei Subsysteme als günstig annehmen:

XML	STRING	TEST
tinyxml.cpp	tinystr.cpp	xmltest.cpp
tinyxml.h	tinystr.h	
tinyxmlparser.cpp		
tinyxmlerror.cpp		

„XML“ wozu die Dateien `tinyxml.h, tinyxml.cpp, tinyxmlparser.cpp` und `tinyxmlerror.cpp` gehören, die zusammen die eigentliche XML-Funktionalität bereitstellen

„STRING“ mit den Dateien `tinystr.h` und `tinystr.cpp` für string-spezifische Funktionalität.

„TEST“ bestehend aus der Datei `xmltest.cpp`

Bei dieser Einteilung erhält man die gleichen Schnittstellen wie zuvor, die Ergebnisse sind in Tabelle 3.7 dargestellt. Es ist zu erkennen, dass die beiden Subsysteme hier relativ kleine Schnittstellen im Vergleich zum XCTL-System haben. Die Schnittstellen bestehen, bis auf einen Enum, nur aus Klassen und deren Memberfunktionen, was auf gutes objektorientiertes Design hinweist.

Subsystem	Anzahl der Symbole											
	CLASS	STRUCT	FUNCTION	ENUM	TYPEDEF	VARIABLE	DEFINE	MEMBERVAR	MEMBERFCT	k_i	s_i	r_i
XML	5								7	12	3,046	3,9
STRING	2			1					2	5	0,536	9,3
TEST											0,492	
gesamt	7			1					9	17		$R: 6{,}6^a$

[a] ohne TEST

Tabelle 3.7: Ergebnisse Tinyxml

Kapitel 4

Ergebnisse und Ausblick

4.1 Das Tool

Mit **InterfaceFinder** ist ein einfach zu benutzendes Tool entwickelt worden, dass die Bestimmung von Subsystemschnittstellen ermöglicht. Durch das schnelle Ermitteln der Schnittstellen können unterschiedliche Subsystemeinteilungen ausprobiert und miteinander verglichen werden.

4.1.1 Aufwand

Der Aufwand für die Schnittstellenberechnung ist abhängig von der Größe des zu untersuchenden Systems. Genauer gesagt, ist die benötigte Rechenzeit abhängig von der Anzahl der definierten Symbole und der Referenzen. Der zugrundeliegende Algorithmus sieht vereinfacht folgendermaßen aus:

1. Symbole und Referenzen übertragen
2. Für jede Referenz:
   (a) bestimme referenzierendes Subsystem
   (b) bestimme referenziertes Subsystem
   (c) Wenn referenziertes Subsystem $\neq$ referenzierendem Subsystem
      i. füge referenziertes Symbol zur Schnittstelle des Subsystems hinzu

Der erste Schritt, das Übertragen der Symbole und Referenzen mit dem Eintragen der Informationen in die Datenbank, erfordert einen linearen Zeitaufwand in Abhängigkeit von der Anzahl der Symbole und Referenzen. Das Bestimmen des Subsystems zu einem Symbol (Schritte 2.(a) und 2.(b)) ist in einer festen Zeit möglich, zumindest solange, wie bei den für die Datenbanken verwendeten Hashtabellen keine Kollisionen auftreten. Auch das Hinzufügen eines Symbols zur Subsystemschnittstelle benötigt nur einen konstanten Zeitaufwand. Somit ist der Aufwand für den gesamten Schritt 2 linear abhängig von der Anzahl der Referenzen. Also ist der Zeitaufwand für den gesamten Algorithmus linear abhängig von der Anzahl der Symbole und Referenzen.

Bei **InterfaceFinder** kommt zu obigen Algorithmus noch das Bestimmen der Subsystemeinteilung und die Ausgabe der Schnittstellen hinzu. Für Ersteres ist der Aufwand dafür abhängig von der Anzahl der Subsysteme und Dateien. Für der Ausgabe hängt der Aufwand von der Größe der Schnittstellen ab.

Auch die Größe der Datenstrukturen ist abhängig von der Anzahl der Symbole und Referenzen. Beim XCTL-System treten mehrere tausend Symbole und ca.

50 000 Referenzen auf. Da nicht alle Referenzen gespeichert werden, sondern nur die für die Schnittstellen relevanten, bei denen verschiedene Subsysteme beteiligt sind, kann die Referenzdatenbank klein gehalten werden. Dafür müssen aber bei einer geänderten Subsystemeinteilung die Referenzen erneut übertragen werden.

4.1.2 Korrektheit

Die mit **InterfaceFinder** ermittelten Schnittstellen sind, wie der Test zeigte, soweit korrekt, wie dies in der Zuständigkeit von **InterfaceFinder** möglich ist. Auftretende Fehler oder Unvollständigkeiten in den Schnittstellen beruhen auf entsprechend fehlerhaften Informationen von SNiFF+. Die Schwierigkeiten mit SNiFF+ umfassen:

Operatorüberladung: Wird der *-Operator als friend-Funktion in einer Klasse überladen, erzeugt SNiFF+ auch bei einfachen Multiplikationen Referenzen auf die entsprechende Klasse, wodurch diese Klasse evtl. unnötig zur Schnittstelle gezählt wird. **InterfaceFinder** gibt bei solch einer Operatorüberladung eine Warnung aus.

Undefined references: Es werden Referenzen empfangen, bei denen das referenzierte Symbol nicht bekannt ist. Solche Referenzen können für die Schnittstellenbestimmung nicht weiter berücksichtigt werden. Es wird eine Warnung ausgegeben.

Symbole mit gleichen Namen: Wenn in Dateien aus verschiedenen Übersetzungseinheiten Symbole mit gleichen Namen definiert werden (z.B. die Variable `szMessage` im XCTL) kann SNiFF+ diese nicht immer korrekt unterscheiden. Sämtliche Referenzen auf solche Symbole beziehen sich dann nur auf eine der Definitionen. Das Symbol wird dann in die Schnittstelle dieses Subsystems aufgenommen, auch wenn alle Verwendungen außerhalb des Subsystems zu einer anderen Definition gehören.

Makros: Da SNiFF+ keine Informationen zur Makrobenutzung liefert, muss **InterfaceFinder** die Benutzung der Makros selbst ermitteln. Dabei kann bei Makros die in verschiedenen Dateien mit gleichen Namen definiert sind nicht immer die gültige Definition gefunden werden. In solch einem Fall wird der Nutzer durch eine Warnung darauf hingewiesen.

Nicht unterstützte Schlüsselwörter: SNiFF+ unterstützt nicht den gesamten C/C++ -Sprachumfang: weder Templates noch Namespaces und die Schlüsselwörter `auto`, `explicit`, `extern`, `mutable`, `register` und `volatile` werden unterstützt. Für die Bestimmung von Schnittstellen ist von den Schlüsselwörtern nur `extern` relevant, Templates und Namespaces kommen jedoch im XCTL nicht vor. Das Schlüsselwort `extern` wird von **InterfaceFinder** im Zusammenhang mit Variablen beachtet.

Den von **InterfaceFinder** gelieferten Ergebnissen kann also nicht blind vertraut werden, vielmehr müssen diese kritisch betrachtet und anhand des Quelltextes des untersuchten Systems überprüft werden. Die genannten Probleme betreffen im Allgemeinen aber nur einen sehr kleinen Anteil der Symbole, so dass die Ergebnisse trotzdem brauchbar sind.

4.1.3 Mögliche Erweiterungen

Im Verlauf der Entwicklung und Anwendung von **InterfaceFinder** entstanden mehrere Ideen für Erweiterungen des Programms:

Subsystemeinteilungen

Um die Arbeit mit dem Tool zu vereinfachen, wäre es günstig, wenn man die Subsystemeinteilung abspeichern und später wiederwenden könnte. Die gewünschten Einteilungen müssen mit Hilfe des Dialoges „Subsysteme zuordnen“ angelegt werden. Wenn die Einteilung dabei sehr stark von der Verzeichnisstruktur abweicht und viele Dateien und Subsysteme umfasst, ist es relativ mühsam die Einteilung anzulegen. Diese Arbeit muss wiederholt werden, wenn die Einteilung zu einem späteren Zeitpunkt nochmals verwendet werden soll.

Als Arbeitserleichterung bietet es sich bei einer vielbenutzten Einteilung daher an, eine Kopie der Quellen anzulegen und die Einteilung in einer entsprechenden Verzeichnisstruktur der Quelldateien zu realisieren.

Für die Realisierung einer Funktion zum Wiederverwenden von Subsystemeinteilungen müsste in der Oberfläche ein Menupunkt zum Abspeichern und ein weiterer Menupunkt zum Öffnen einer Subsystemeinteilung hinzugefügt werden. Das Sichern der Datenstrukturen, die die Subsystemeinteilung beschreiben, ist an sich nicht weiter schwierig, es müssen aber einige Dinge beachtet werden: Zum einen muss sichergestellt werden, dass die gesicherte Einteilung eindeutig dem entsprechenden SNiFF-Projekt zugeordnet bleibt, damit ein Wiederverwenden möglich wird. Zum anderen müssen Änderungen an der Dateistruktur des Projektes zwischen Sichern und Wiederverwenden der Einteilung erkannt und berücksichtigt werden.

Statistik

Das Bewerten der ermittelten Subsystemschnittstellen erfordert das Erfassen von verschiedenen Größen. Diese Größen müssen bisher manuell anhand der Export-Dateien bestimmt werden, wobei die meisten dieser Größen von **InterfaceFinder** auch „nebenbei“ bestimmt werden könnten. Dazu müssten von den verschiedenen `...Handler`-Klassen, in den Datenbank-Klassen und in `InterfaceGenerator` entsprechende Informationen gesammelt und berechnet werden. Ein spezielles Ausgabemodul könnte dann die Statistikinformationen in einem Fenster ausgeben.

Die möglichen Größen wären: Anzahl der definierten Symbole (insgesamt und je Subsystem oder Datei), Anzahl der Referenzen und für jede Schnittstelle die Anzahl der enthaltenen Symbole (insgesamt und je nach Typ des Symbols). Die Größe der Subsysteme in Lines of Code kann so nicht ermittelt werden, da SNiFF+ diese Information nicht überträgt. Wenn **InterfaceFinder** auch die relative Schnittstellengröße berechnen soll, müsste diese also auf die Gesamtzahl der im Subsystem definierten Symbole bezogen werden anstatt auf die Größe des Subsystems in Lines of Code.

Ausgabe

Durch weitere Ausgabemodule lässt sich die Ausgabe jeder gewünschten Form anpassen. So könnten z.B. die Ausgabe in eine Datei geschrieben werden, in einer Form die eine weitere Verarbeitung mit einem anderen Programm ermöglicht.

Sprachumfang

Der mögliche Sprachumfang für zu untersuchende Systeme könnte erweitert werden. Insbesondere könnten **InterfaceFinder** auch für in Java implementierte Systeme angepasst werden. Eine volle Unterstützung des C++ -Sprachumfangs (Templates, Namespaces) ist nicht ohne weiteres möglich, solange dies nicht von SNiFF+ unterstützt wird. Die Unterstützung für diese Sprachelemente selbständig zu ergänzen würde erheblichen Aufwand bedeuten, wobei letzendlich ein Parser für den Quelltext benötigt würde. Wenn aber ein eigener Parser zur Verfügung steht, bringt die

Verwendung von SNiFF+ keine größeren Vorteile mehr, so dass dann auf SNiFF+ auch verzichtet werden könnte.

Ersetzen von SNiFF+

Da Sniff die Informationen nicht ganz fehlerfrei liefert und derzeit nicht den gesamten C/C++ -Sprachumfang unterstützt, könnte über einen Ersatz nachgedacht werden. Wenn ein anderes Tool das den Quelltext parst und die Referenzen ermittelt zur Verfügung steht, muss bei **InterfaceFinder** die Schnittstelle zu SNiFF+ ersetzt bzw. dem neuen Tool angepasst werden. Dazu müssten die Klasse `SniffClient` und die verschiedenen Handlerklassen geändert werden.

4.2 Bewertung von Schnittstellen

Die von **InterfaceFinder** ermittelten Schnittstellen können benutzt werden, um das Design der Schnittstelllen und die Subsystemeinteilung des Programms zu untersuchen und zu bewerten. Dazu wurden einige Größen bestimmt, um Vergleiche zu ermöglichen. Mit Hilfe der relativen Schnittstellengröße r_i, die die Anzahl der Symbole in der Schnittstelle ins Verhältnis setzt zur Größe des Subsystems, können die Schnittstellen der einzelnen Subsysteme miteinander verglichen werden. Der Durchschnittswert R der relativen Schnittstellengröße über alle Subsysteme liefert eine Größe, mit der verschiedene Subsystemeinteilungen verglichen werden können. Wird R für eine zufällige Subsystemeinteilung bestimmt, erhält man einen Bezugswert, mit dem andere Einteilungen bewertet werden können.

Bei der Bestimmung der Größe R trägt jedes Subsystem mit seiner relativen Schnittstellengröße r_i den gleichen Anteil bei, unabhängig von der Größe des Subsystems. Damit besteht die Gefahr, dass kleine Subsysteme überbewertet werden. Zur Verbesserung könnte daher versucht werden, bei der Bestimmung von R eine Gewichtung nach der Größe der Subsysteme vorzunehmen.

Es sollte auch untersucht werden, ob es günstiger ist die relative Schnittstellengröße auf die Anzahl der im Subsystem definierten Symbole zu beziehen, anstatt auf die Größe des Subsystems s_i. Damit würde mehr über den in der Schnittstelle veröffentlichen Anteil der Symbole des Subsystems ausgesagt als es in der jetzigen Variante der Fall ist. Zur Bestimmung der Anzahl der definierten Symbole muss allerdings ein Tool zur Verfügung stehen.

Mit der Größe R können bisher nur Einteilungen mit der gleichen Anzahl von Subsystemen miteinander verglichen werden, da die Anzahl der Subsysteme die Größe der Schnittstellen beeinflußt. Es stellt sich daher die Frage, ob es möglich ist eine Größe zu finden, mit der Subsystemeinteilungen unabhängig von der Anzahl der Subsysteme bewertet werden können. Wenn dies gelingt, kann dies genutzt werden, um nach einer optimalen Einteilung für ein System zu suchen. Die Suche könnte dann auch automatisiert mit einem Tool, dass auf **InterfaceFinder** aufbauend nacheinander die Schnittstellen verschiedener Einteilungen bestimmt und bewertet, durchgeführt werden.

Eine andere Vorgehensweise wäre, zunächst zu versuchen die „optimale“ Anzahl von Subsystemen für ein System zu bestimmen. Die mögliche Anzahl liegt zwischen nur einem Subsystem das alle Dateien enthält und einem Subsystem je Datei. Es ist jedoch zu bezweifeln, dass eine optimale Subsystemanzahl nur anhand des Quelltextes, also ohne Kenntnis der Funktionalität der Subsysteme, bestimmt werden kann. Die Subsystemanzahl ist zu sehr von anderen Faktoren abhängig, da mit einer Subsystemeinteilung immer auch ein bestimmtes Konzept oder eine Designidee umgesetzt werden soll.

Desweiteren stellt sich die Frage nach dem „optimalen“ Wert für die relative Schnittstellengröße. Die einfache Formel „Je kleiner desto besser.“ gilt sicherlich nur begrenzt, schließlich kann die kleinste (leere) Schnittstelle erreicht werden, wenn alle Dateien zu einem Subsystem gehören. Bei großen Subsystemen mit sehr kleiner Schnittstelle muss daher untersucht werden, ob vielleicht das Subsystem zu umfangreich ist und eine Aufteilung in mehrere Subsysteme günstiger wäre.

4.3 Schnittstellen des XCTL-Systems

Durch die Betrachtung der Schnittstellen des XCTL-Systems konnte die Subsystemeinteilung grundsätzlich bestätigt werden, insofern, dass mit dieser Einteilung der Quelltext strukturiert und Funktionalität in Subsystemen abgegrenzt wird. Es zeigt sich, dass das XCTL in großen Teilen objektorientiert programmiert ist, wobei aber immer noch Reste eines C-typischen Programmierstils vorhanden sind. Die unterschiedliche Struktur der Schnittstellen und der Subsysteme deutet darauf hin, das bei der Entwicklung nicht einheitlich vorgegangen wurde und verschiedene Konzepte verfolgt wurde. Generell sollte bei der zukünftigen Weiterentwicklung auf mehr Einheitlichkeit geachtet werden, wozu entsprechende Konzepte entwickelt und feste Vorgaben in Bezug auf die Architektur und den Programmierstil gemacht werden müssen.

Dies gilt insbesondere auch für die Darstellung der Schnittstellen im Quelltext, die nicht für alle Subsystem einheitlich ist. Zum Teil sind die Schnittstellen nicht als solche zu erkennen oder nur teilweise in entsprechenden Headerdateien untergebracht. Hier sollte unbedingt ein einheitliches Konzept gefunden werden.

Die Frage nach einer besseren Subsystemeinteilung bleibt weiter offen, da für die Suche danach erst, wie im vorigen Abschnitt beschrieben, eine Möglichkeit gefunden werden muss, Subsystemeinteilungen mit unterschiedlicher Anzahl von Subsystemen zu vergleichen.

Anhang A

Pflichtenheft

Aufgabe

Das Tool **InterfaceFinder** soll zu einem im Quellcode gegebenen Programmsystem, welches in Subsysteme eingeteilt ist, die tatsächlich benutzten Schnittstellen der Subsysteme (Used-Interface) ermitteln. Die Subsystemeinteilung orientiert sich dabei an Dateien, das heißt jede Quelldatei ist genau einem Subsystem zugeordnet.

Das Programm SNiFF+ von Wind River wird zu Hilfe genommen, um Strukturinformationen über das zu untersuchende Programmsystem zu erhalten.

Grundlegende Funktionen

1. Öffnen eines Projektes

   (a) Auswahl einer SNiFF+ -Projektdatei:
   Eine SNiFF+Projektdatei kann mit Hilfe einer Standard-File-Open-Dialogbox ausgewählt werden.

   (b) Festlegen des SNiFF+ -Working-Environments:
   Eine Dialogbox ermöglicht die Eingabe des SNiFF+ -Working-Environments. Dabei soll dem Nutzer ein Vorschlag angeboten werden, der sich aus dem Namen der ausgewählten Projektdatei ergibt.

2. Festlegen der Subsystemeinteilung

   Jede Quellcodedatei des zu untersuchenden Programmsystems muß eindeutig einem Subsystem zugeordnet werden. Das Tool bestimmt zunächst eine Subsystemeinteilung anhand der Verzeichnisstruktur der Quellen, wobei von einer Verzeichnisstruktur wie beim XCTL-System ausgegangen wird:

```
/... XC010701/
        AUTOJUST/
            M_JUSTAG.CPP
            MATRIX.CPP
            TRANSFRM.CPP
        DATAVISA/
            M_CURVE.CPP
            M_DATA.CPP
        ...
        INCLUDE/
```

```
    AUTOJUST/
        MATRIX.H
        TRANSFRM.H
        AUTOJUST.H
        M_JUSTAG.H
    ...
INTERNLS/
    M_MAIN.CPP
...
sniff_data/
    XC010701.proj
...
```

Das Verzeichnis, in welchem sich das `sniff_data`-Verzeichnis befindet (hier: `XC010701`), wird zum Basisverzeichnis. Alle Unterverzeichnisse, in denen sich Quellcodedateien befinden, werden als Subsystem betrachtet. Der Teil des Pfadnamens ab dem Basisverzeichnis wird zum Namen des Subsystems. Subsystemnamen müssen eindeutig sein, was durch die Verwendung des Pfadnamens sichergestellt ist.

Der Inhalt des `INCLUDE`-Verzeichnisse wird so behandelt, als befände er sich direkt im Basisverzeichnis, d. h. alle Quellcodedateien, aus Unterverzeichnissen von `INCLUDE` mit gleichem Namen wie ein Verzeichnis höherer Stufe, werden zu dem entsprechenden Subsystem hinzugefügt.

(a) Zuordnen von Dateien zu Subsystemen

Die so vom **InterfaceFinder** bestimmte Subsystemeinteilung kann vom Nutzer verändert werden.

In einer Baumdarstellung werden die Subsysteme und die zugehörigen Dateien angezeigt. Jede Datei kann in ein anderes Subsystem verschoben werden. Dabei ist sicherzustellen, dass jede Datei genau einem Subsystem zugeordnet ist.

(b) Hinzufügen, Umbenennen und Entfernen von Subsystemen

Der Nutzer kann neue Subsysteme anlegen und vorhandene umbenennen. Die Namen der Subsysteme müssen eindeutig sein, ein schon benutzter Name darf nicht ein zweites Mal verwendet werden.

Subsysteme, denen keine Datei zugeordnet ist, können vom Nutzer entfernt werden.

Es ist nicht direkt vorgesehen Subsysteme hierarchisch anzuordnen, das heißt Subsubsysteme, und so weiter, zu bilden. Jedoch können Hierarchien von Subsystemen trotzdem verarbeitet werden, da die Namen der Subsysteme aus dem Pfadnamen gebildet werden, womit auch bei einer tiefer geschachtelten Verzeichnisstruktur die Eindeutigkeit gewahrt bleibt. Dabei werden aber die Schnittstellen berechnet wie bei einzelnen, nicht hierarchisch angeordneten, Subsystemen.

3. Bestimmen der Subsystemschnittstellen

InterfaceFinder startet das SNiFF+ -Programm und überträgt die benötigten Strukturinformation mit Hilfe der SNiFF+ -Java-API. Es werden dazu Informationen über die Dateien des Projektes, alle deklarierten Symbole und Referenzen übertragen.

   (a) Zuordnung von Subsystemen zu Symbolen
   Alle deklarierten Symbole müssen eindeutig einem Subsystem zugeordnet werden. Jedem Symbol wird das Subsystem der Datei in der das Symbol *definiert* wird zugewiesen.

   (b) Aus den von SNiFF+ gelieferten Informationen berechnet **InterfaceFinder** die Subsystem-Schnittstellen. Dabei wird ein Symbol genau dann in die Subsystem-Schnittstelle aufgenommen, wenn es aus einem anderen Subsystem referenziert wird: Liefert SNiFF+ eine Referenz der Form $A \rightarrow B$ (A referenziert B) und gehört A zu einem anderen Subsystem als B, so wird B in die Schnittstelle des Subsystems von B aufgenommen.

4. Ausgabe der Subsystemschnittstellen

   Zu jedem Subsystem wird eine Liste der in anderen Subsystemen verwendeten Symbole (Art und Name des Symbols, evtl. zugehörige Datei) ausgegeben. Dies umfaßt: Klassen, Funktionen, Variable, Membervariable, Memberfunktionen, Structs, Enums, Unions, Makros (define's) und Typedefs.

   Die Ausgabe kann in eine Datei geschrieben werden.

Oberfläche

Für den Nutzer wird eine grafische Oberfläche angeboten, wobei die Java-Swing-API benutzt wird.

Ferner soll es möglich sein, **InterfaceFinder** ohne grafische Oberfläche zu benutzen. Dabei wird nur die Verzeichnisstruktur als Subsystemeinteilung verwendet, ein Ändern ist nicht möglich.

Optionale Funktionen

1. Ausgabe der Schnittstellen

   (a) Anpassen der Ausgabe
   Die Ausgabe der Subsystemschnittstellen kann modifiziert werden. So kann der Nutzer festlegen welche Elemente in der Ausgabe erscheinen sollen.

   (b) weitere Ausgabeformen
   Eventuell ist es möglich die Schnittstellen als Headerdateien auszugeben.

2. Sichern von Subsystemeinteilungen

   Die (vom Nutzer vorgenommene) Subsystemeinteilung kann gespeichert und bei späteren Programmläufen wieder benutzt werden.

3. Statistik

   Das Programm gibt Statistik-Informationen zu den Schnittstellen aus, wie z.B Größe der Schnittstellen, Anzahl der verschiedenen Elemente

Voraussetzungen

Zum Einsatz von **InterfaceFinder** müssen folgende Voraussetzungen erfüllt sein:

- Windows 2000/NT/XP
- Java 1.3.1 oder höher

- SNiFF+ Version 4.0.1 oder höher
- Das zu untersuchende Programmsystem muß in C oder C++ geschrieben und korrekt compilierbar sein. Ein zugehöriges SNiFF+ -Projekt muss vorhanden sein.

Einschränkungen

Da SNiFF+ zum Teil fehlerhafte Informationen liefert, kann die Korrektheit der Schnittstellen nicht garantiert werden. Wenn möglich werden von **InterfaceFinder** Warnhinweise ausgegeben.

Es wird nicht der volle C++-Sprachumfang unterstützt, insbesondere werden keine Templates und Namespaces behandelt, da diese durch die Sniff-API nicht unterstützt werden. Da **InterfaceFinder** im Hinblick auf die Untersuchung des XCTL-Systems entwickelt wird, brauchen auch nur dort verwendete C++-Sprachelemente unterstützt werden.

Umgebung

Implementation und Test erfolgen unter Windows 2000 mit Java 1.3.1 und SNiFF+ 4.0.1.

Anhang B

Kurzanleitung InterfaceFinder

Voraussetzungen

Voraussetzung zum Verwenden von **InterfaceFinder** ist Java (ab der Version 1.3.1) und ein lauffähiges SNiFF+ (ab Version 4.0.1).

Installation

1. Die Dateien `InterfaceFinder.jar` und `if_start.cmd` können in ein beliebiges Verzeichnis kopiert werden.

2. Die Datei `if_start` kann mit einem Texteditor bearbeitet werden. In der Datei müssen die dort angegebenen Pfade (Sniff- und Programmverzeichnis) an die aktuellen Verhältnisse angepasst werden.

Vorbereitungen

Bevor **InterfaceFinder** auf ein System angewendet werden kann, muss für dieses System in SNiFF+ ein Projekt angelegt werden. Dazu muss SNiFF+ gestartet und mit Hilfe des Import-Wizards die Quelldateien eingelesen werden. Dabei wird im Verzeichnis der Quelldateien ein Unterverzeichnis `sniff_data` angelegt, das eine `.proj`-Datei enthält. Diese Datei muss später mit **InterfaceFinder** geöffnet werden.

SNiFF+ verwaltet Projekte in sogenannten Working-Environments. **InterfaceFinder** erwartet die Eingabe des Working-Environments beim Öffnen eines Projektes. Das Working-Enviroment eines Projektes kann beim Anlegen des Projektes angegeben werden und ist nach dem Öffnen des Projektes in der Titelzeile des SNiFF+ -Fensters sichtbar.

Starten von InterfaceFinder

Das Programm kann durch einen Doppelklick auf `if_start.cmd` gestartet werden. Nach kurzer Zeit erscheint das **InterfaceFinder** -Hauptfenster.

Das Programm kann auch von der Kommandozeile aus durch Eingabe von `if_start` gestartet werden.

Mit der Option „-c“ kann dabei die Swing-Oberfläche abgeschaltet werden, es wird dann die Kommandozeilenversion verwendet. Bei der Kommandozeilenversion ist es allerdings nicht möglich, die Subsystemeinteilung zu bearbeiten, es wird nur die durch die Verzeichnisstruktur gegebene Einteilung benutzt.

Ermitteln von Subsystemschnittstellen

Zum Ermitteln der Subsystemschnittstellen eines Systems sind folgende Schritte auszuführen:

1. Nach Auswählen des Menupunktes **File→Open...** erscheint ein Standard-File-Open-Dialog. Hier muss die `.proj`-Datei des zu untersuchenden Systems gesucht und ausgewählt werden.

2. Es erscheint ein Dialog, der zur Eingabe des Working-Environments des zu öffnenden Projektes auffordert. Das Working-Environment muss wie in SNiFF+ sichtbar, inklusive des vorangestellten Schrägstriches, angegeben werden. In den meisten Fällen kann der angezeigte Vorschlag durch Drücken von **OK** übernommen werden.

   Anschließend versucht **InterfaceFinder** eine Verbindung mit SNiFF+ aufzubauen, falls nötig wird dabei das SNiFF+-Programm gestartet. Nach einem erfolgreichen Verbindungsaufbau werden erste Daten zum Projekt übertragen. Nach einer Weile erscheint der Dialog für die Einteilung der Subsysteme.

3. Der Dialog zeigt zunächst die aus der Verzeichnisstruktur bestimmte Subsystemeinteilung. Dateien können ausgewählt und mit Hilfe der entsprechenden Buttons (**Up** bzw. **Down**) in ein anderes Subsystem verschoben werden. Das Anlegen und Löschen von Subsystemen erlauben die Buttons **New** bzw. **Remove**. Mit **Revert** kann die ursprüngliche Einteilung wieder hergestellt werden. Nach dem Anlegen der gewünschten Subsystemeinteilung muss der Dialog mit **OK** bestätigt werden.

   Nun setzt **InterfaceFinder** die Arbeit mit der Übertragung weiterer Daten und der Berechnung der Schnittstellen fort. Wenn die Berechnung abgeschlossen ist, wird die Schnittstelle des ersten Subsystems angezeigt.

Weitere Funktionen

Subsystemeinteilung verändern

Nach dem erstmaligen Bestimmen der Subsystemschnittstellen kann mit dem Menupunkt **File→Reassign...** wieder der Dialog zur Bearbeitung der Subsystemeinteilung geöffnet werden. Anschließend werden die Schnittstellen neu berechnet.

Schnittstellen Exportieren

Mit dem Menupunkt **File→Export...** können alle Schnittstellen in eine Textdatei exportiert werden.

Zuletzt benutzte Projekte

Unter dem Menupunkt **File→Open recent...** werden die letzten zehn benutzten Projekte angezeigt. Diese können damit schneller geöffnet werden.

Anzeige der Schnittstellen

Die Subsystemschnittstellen werden in einer Tabelle angezeigt. Die Tabelle kann durch einen Klick auf den Kopf einer Spalte nach dem Inhalt dieser Spalte sortiert werden. Mit `SHIFT`-Klick erfolgt die Sortierung in umgekehrter Reihenfolge.

Der linke Teil des Fensters zeigt die Subsysteme und die zugehörigen Dateien. Durch Auswahl einer Datei wird der Inhalt der Datei angezeigt. Um wieder die Schnittstelle eines Subsystems anzuzeigen, muss ein Subsystem ausgewählt werden.

Anhang C

Quelltext des Testsystems

Im Folgenden wird der Quelltext des Testsystems aufgelistet. Das Testsystem besteht aus den Dateien: `Test_A.cpp`, `Test_B.cpp` und `sub_B.h`, die in folgender Verzeichnisstruktur angeordnet sind:

```
/... Testsystem/
        SUB_A/
            Test_A.cpp
        SUB_B/
            Test_B.cpp
        INCLUDE/
            Sub_B/
                sub_B.h
```

Die Dateien des Testsystems befinden sich auch auf der beigelegten CD im Verzeichnis `Test`.

Test_A.cpp

```
/************************************************
 Testsystem, Subsystem A

 benutzt Elemente aus Subsystem B
************************************************/
#include <iostream>
#include "sub_B.h"


//Makro, mit gleichen Namen auch in Test_B.cpp
#define makro_in 4;

//Variable
//externe Variable
extern int var_ext_b;


//static Variable
static int var_stat_b;
```

```
int main()
{
std::cout << "Hello! This is the Testsystem" << std::endl << std::endl;


    // Zugriff auf Elemente des Subsystems B

    var_ext_b = 2;

    var_stat_b = 3;


    //Klassen
    Klasse_B_ex * klasse_b_ptr = new Klasse_B_ex();;
    Klasse_B_ex klasse_b;

    klasse_b.member_b_2 = 0.125;
    klasse_b.function_b_2('a');


    //struct
    Struct_B_ex struct_b;
    struct_b.member_b_5 = 0;

    //union
    Union_B_ex union_b;
    union_b.member_b_7 = 'a';
    int i = union_b.member_b_8;


    // Funktionen
    i =  function_b_ex(0);

    function_b_ex_2(1);

    //enums
    enum_b_ex e = e3;
    e = enum_b_ex(a1 + a2); //indirekte Benutzung von enum_b_ex_2

    //typedef
    type_b_ex d;
    d = 'd';

    //Makros
    int m = makro_b_ex + makro_in;

return 0;
}
```

Test_B.cpp

```
#include "sub_B.h"

//Makros
#define makro_in 3;


//Variablen
//globale Variable
int var_b;

//externe Variable
int var_ext_b;

//static Variable
static int var_stat_b;



// Definition der Memberfunktionen
// dabei Benutzung von Elementen aus sub_B.h

void Klasse_B_ex::function_b_1(int i) {
    i = makro_b_in + makro_in + makro_b_ex;
    i = function_b_in(i);
    i++;
}

int Klasse_B_ex::function_b_2(char c) {
    return c + 1;
}


void Klasse_B_in::function_b_3(int i) {
    i++;
}

int Klasse_B_in::function_b_4(char c) {
    function_b_3(2);
    member_b_3 = 17;
    return c + 2;
}


// Nicht-Member Funktionen
int function_b_in(int i) {
    Klasse_B_in klasse_b;
    Struct_B_in struct_b;
    Union_B_in union_b;

    klasse_b.function_b_4('c');
    klasse_b.member_b_4 = 12;
```

```
    struct_b.member_b_6 = 0.12;

    union_b.member_b_9 = 100;
    union_b.member_b_10 = 1.1001;

    enum_b_in e = a;
    type_b_in tb = 0.123;
    function_b_in_2(i);
    return i;
}

int function_b_ex(int i) {
i++;
    return i;
}

Test_Operator operator + (double fakt, const Test_Operator & mat) {
    Test_Operator t;
return t;
}
```

sub_B.h

```
/**************************************
 Headerdatei für Testsystem, Subsystem B
**************************************/

#ifndef _SUBB_H_
#define _SUBB_H_

//Makros
#define makro_b_ex 1;
#define makro_b_in 2;


// Klasse mit Zugriff von Außen
class Klasse_B_ex {

private:
    int member_b_1 ;
    void function_b_1(int i);

public:
double member_b_2;
    int function_b_2(char c);
};


// Klasse ohne Zugriff von Außen
class Klasse_B_in {

private:
    int member_b_3;
    void function_b_3(int i);

public:
double member_b_4;
    int function_b_4(char c);
};


// Struct mit Zugriff von Außen
struct Struct_B_ex {
    long member_b_5;
};


// Struct ohne Zugriff von Außen
struct Struct_B_in {
    float member_b_6;
};


// Union mit Zugriff von Außen
```

```
union Union_B_ex {
    char member_b_7;
    int member_b_8;
};


// Union ohne Zugriff von Außen
union Union_B_in {
    long member_b_9;
    double member_b_10;
};


//Funktionen
int function_b_in(int i);
int function_b_ex(int i);

inline void function_b_in_2(int i){ i++; };
inline void function_b_ex_2(int i){ i++; };


// Enums
enum enum_b_ex {e1, e2, e3};
enum enum_b_ex_2 {a1, a2, a3, a4 };
enum enum_b_in {a, b, c};

//typedefs
typedef char type_b_ex;
typedef double type_b_in;


// Operatorüberladung
class Test_Operator {

public:

friend Test_Operator operator + (double fakt, const Test_Operator & mat);
};


#endif
```

Literaturverzeichnis

[Bass] Len Bass, Paul Clements, Rick Kazman, „Software Architecture in Practice“, Second Edition, Addison-Wesley, 2003

[objectiF] MicroTool Webseite:
http://www.microtool.de/objectif/de/index.htm

[Schützler] Kay Schützler, „Wiedergewinnung von Subsystemen durch Use-Case-Analyse und Dateistrukturierung am Beispiel des XCTL-Systems“, Diplomarbeit am Institut für Informatik, Humboldt-Universität zu Berlin, 2001

[SNiFF] SNiFF/SNiFF+ Pro User's Guide and Reference, 4.0.1, Wind River Systems Inc. 2001

[Stroustrup] Bjarne Stroustrup, „Die C++-Programmiersprache“, Addison-Wesley Verlag, 2000

[Thiel] Tobias Thiel, „Verbesserung der Headerdateistruktur und Herausarbeiten der Subsystemschnittstellen am Beispiel des XCTL-Systems“, Studienarbeit am Institut für Informatik, Humboldt-Universität zu Berlin, 2002

[TinyXml] TinyXml Webseite: http://www.sourceforge.net/projects/tinyxml

[XCTL] XCTL-Projekt Webseite:
https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98y/index.html