DIPLOMARBEIT

zur Erlangung des akademischen Titels
Diplom-Informatiker

Metriken zur Portabilitätsanalyse Windows-basierter Software-Systeme

von

Michael Müller

März 2002

Betreuer: Prof. Dr. Klaus Bothe

Institut für Informatik
Math.-Naturwiss. Fakultät II
Lehrstuhl für Softwaretechnik
Rudower Chaussee 25
Humboldt-Universität zu Berlin

Selbstständigkeitserklärung

Hiermit erkläre ich, Herr Michael Müller, die vorliegende Diplomarbeit zum Thema

Metriken zur Portabilitätsanalyse Windows-basierter Software-Systeme

selbständig und ausschließlich unter Verwendung der im Quellenverzeichnis aufgeführten Literatur- und sonstigen Informationsquellen verfaßt zu haben.

Berlin, am 28.3.2002 Unterschrift (Michael Müller)

Einverständniserklärung

Hiermit erkläre ich mein Einverständnis mit der öffentlichen Ausstellung meiner Diplomarbeit in der Bibliothek des Instituts für Informatik.

Berlin, am 28.3.2002 Unterschrift (Michael Müller)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Entwicklung von Software-Systemen ist mit ständig steigenden Kosten verbunden. Aus Herstellersicht wird daher die Ausdehnung der Zykluszeit von Anwendungssystemen und deren weite Verbreitung zunehmend wichtiger. Auch aus Anwendersicht ist die lange Lebensdauer und die weite Verbreitung mit geringeren Anschaffungskosten und verringertem Einarbeitungsaufwand verbunden, wodurch sich schließlich die Amortisierungsdauer von Anwendungssoftware verkürzt.

Einer ausgedehnten Zykluszeit für Anwendungssoftware steht aber eine wesentlich niedrigere Zykluszeit für Systemsoftware und eine noch geringere Hardware-Zykluszeit entgegen. Entwickelt man heute Anwendungssoftware, dann bedeutet dies, daß während der Lebenszeit der Software mindestens einmal die zugrunde liegende Systemsoftware und mindestens zweimal die Hardware ausgetauscht wird. Die weite Verbreitung einer Anwendungssoftware muß neben der Lebenszykluszeit von Komponenten eines Computersystems zusätzlich den Aspekt der länderspezifischen Varianten berücksichtigen.

Um die Lebensdauer und die weite Verbreitung bereits bestehender Software-Systeme zu ermöglichen, ist es notwendig, diese auf verschiedene Computersysteme zu portieren. Hierbei entstehen ebenfalls Kosten, die mit dem Aufwand für eine Neuimplementierung verglichen werden müssen. Ist der Portierungsaufwand für ein bestehendes Software-System bekannt, so wird aus Herstellersicht eine Entscheidung, bezüglich der Alternativen Portierung oder Neuimplementierung, objektiv erleichtert.

Zur Klärung des Portierungsaufwands soll im Rahmen eines Reengineering-Projekts ein Software-System zur Steuerung einer Röntgen-Topographie-Kamera mit Hilfe von Metriken näher untersucht werden. Ziel der Arbeit ist die Ermittlung des Aufwands der Portierung eines 16-Bit-Windows-basierten Software-Systems, auf die 32-Bit-Plattform des Betriebssystems Windows 98.

Das zweite Kapitel der Arbeit soll zunächst einen Überblick über das Gebiet der Software-Portabilität geben. Hierbei sollen allgemeine Aspekte zur Portabilität von Software-Systemen erläutert und die für die Fragestellung relevanten Begriffe geklärt werden. Es werden im Anschluß daran allgemeine Strategien, Konzepte und Werkzeuge zur Unterstützung der Portierung vorgestellt. Den Abschluß dieses Teils bildet die Vorstellung allgemeiner Portierungsmodelle, welche als Grundlage für die Ermittlung des Portierungsaufwands von Windows-basierten Software-Systemen dienen können und die wesentlichen Einflußfaktoren einer Portierung im betrachteten Portierungsfeld berücksichtigen.

Der zweite Teil dieses Kapitels wird im Speziellen die Portabilität von Software-Systemen, auf der Basis des Betriebssystems Windows, der Firma Microsoft, behandelt. Hierbei werden insbesondere Software-Systeme beleuchtet, die auf der Basis einer 16-Bit-Windows-Plattform entstanden sind und auf die 32-Bit-Plattform von Windows portiert werden sollen. Es beschäftigt sich mit den grundlegenden Konzepten von Windows-Systemen und beschreibt, ausgehend von deren Historie, Strategien zur Programmierung von Software-Systemen unter Windows. Hiernach werden auf der Grundlage der Programmiersprachen C und C++ sowie den Architekturen des betrachteten Portierungsfelds, Windows-spezifische Portierungsprobleme identifiziert und klassifiziert. Im Anschluß daran werden Strategien und Werkzeuge zur Unterstützung des Portierungsprozesses vorgestellt.

Im dritten Kapitel wird die Verwendung von Metriken hinsichtlicht der Portierungsanalyse untersucht. Hierbei soll ein bestehender, evolutionärer Ansatz für ein Portierungsmaß an bestehende Verhältnisse angepaßt werden. Das Portierungsmaß wird hierbei um die Verwendung objektorientierter Metriken erweitert. Der erste Teil der allgemeinen und Windows-spezifischen Portabilitätsbetrachtungen sowie der zweite Teil der Untersuchung vorhandener Metriken, die den Prozeß der Portabilitätsanalyse unterstützen können, bilden die Grundlage für die Untersuchung eines Software-Systems zur Steuerung einer Röntgen-Topographie-Kamera (XCTL-System).

Das vierte und letzte Kapitel soll die Erkenntnisse der vorangegangenen Untersuchungen im Rahmen eines Reengineering-Projekts auf ein 16-Bit-Windows-basiertes Software-System anwenden. Hierbei wird der Versuch unternommen, den Aufwand zu ermitteln, der im Falle einer geplanten Portierung auf die Win32-Plattform entsteht.

2 Portabilität von Software-Systemen

Dieses Kapitel dient der Einführung in die Thematik der Portabilität von Software-Systemen. Zunächst wird anhand von Definitionen versucht, einen allgemeinen Sprachgebrauch für die Arbeit zu finden. Es werden allgemein Probleme behandelt, die bei der Portierung von Software-Systemen auftreten können und welche Werkzeuge den Prozeß der Portierung unterstützen. Danach wird im Speziellen die Portierung von Windows-basierten Software-Systemen behandelt. Hierbei werden die Probleme einer Windows-basierten Portierung aufgezeigt und klassifiziert. Es werden u.a. Werkzeuge vorgestellt, die eine Analyse der Quellen von Software-Systemen, im Hinblick auf die vorgestellten Portierungsprobleme, vornehmen.

2.1 Portabilität im Allgemeinen

Um korrekte und effiziente Software zu produzieren, sind große Anstrengungen erforderlich. Arbeitet ein Programm in einer bestimmten Umgebung, möchte man den ganzen Aufwand bei einem Wechsel des Compilers, des Prozessors oder des Betriebssystems nicht wiederholen. Im Idealfall sollten hierbei überhaupt keine Änderungen erforderlich sein.

Dieser Idealfall ist bei vielen Software-Systemen nicht gegeben. Manche Systeme sind so plattformspezifisch und erfordern so viele Anpassungen, daß eine völlige Neuentwicklung, einer Portierung vorgezogen wird. Um solche Probleme in den Griff zu bekommen und um Neuentwicklungen möglichst plattformunabhängig zu gestalten, beschäftigt man sich mit der Portabilität von Software-Systemen.

2.1.1 Der Portabilitätsbegriff

Im besten Fall sind bei einer Übertragung eines Software-Systems in eine andere System-Umgebung keine Überarbeitungen der Quellen nötig. Man spricht in der Praxis häufig schon dann von einem portablen Programm, wenn es bei der Übertragung der Software zu keiner Neuentwicklung kommt, sondern die vorhandenen Quellen angepaßt werden können. Es ist dann um so portabler, je weniger Überarbeitungen erforderlich sind.

Diesem, eher intuitiven, Verständnis der Portabilität von Software-Systemen stehen eine Reihe von Definitionen gegenüber. Die Begriffe ’Portabilität’ und ’Portierung’ werden jedoch in der einschlägigen Literatur in unterschiedlicher Bedeutung verwendet. Neben diesen Definitionen existieren noch weitere Begriffe wie ’Adaptivität’ und ’Transportabilität’, die entweder synonym oder ergänzend gebraucht werden.

Ziel dieses Abschnitts ist eine Übersicht über die einzelnen, in der Literatur vorhandenen, Definitionen zu geben. Anhand dieser Übersicht soll dann eine Eingrenzung der unterschiedlichen Begrifflichkeiten, zugunsten des einheitlichen Sprachgebrauchs in dieser Arbeit, vorgenommen werden.

2.1.1.1 Was ist Portabilität?

In den 60er Jahren wurden erste Projekte wie SLANG [Sibley 61] oder UNCOL [SHARE 68] durchgeführt, die eine Übertragung von Software von einem Rechner auf einen anderen zum Thema hatten. Eine erste, allgemeine Präzisierung des Portabilitätsbegriffs wurde 1968 von Alan Perlis, im Zuge einer NATO-Arbeitstagung, zum Thema Software-Engineering vorgenommen [Perlis 68]:

> "Portability is the property of a system which permits it to be mapped from one environment to a different environment."

Danach steht Portabilität für die Eigenschaft von Systemen, die es erlaubt, diese von einer Umgebung in eine andere Umgebung zu übertragen. Aus dieser Definition geht allerdings nicht hervor, welche Eigenschaften ein zu übertragendes System haben muß.

Eine weitere Definition wurde von Poole und Waite in den 70er Jahren formuliert [Poole 73]:

> "Portability is a measure of the ease with which a program can be transferred from one environment to another: If the effort required to move the program is much less than that required to implement it initially, then we say this is highly portable."

Hier wird Portabilität als ein Maß der Einfachheit angesehen, mit der ein Programm von einer Umgebung in eine andere übertragen wird. Ist der Aufwand für die Übertragung geringer als eine Neuimplementierung, so bezeichnet man ein Programm als portabel. Hier wird der Aspekt des Aufwands hervorgehoben, der in den meisten Definitionen, entweder in Bezug auf die ursprüngliche Implementation oder einer Neuimplementation, für die Zielumgebung berücksichtigt wird. Allen Definitionen ist gemeinsam, daß der Aufwandsbegriff nicht näher präzisiert wird.

Anfang der 80er Jahre definiert Hommel die Portabilität von Programmen und benutzt dabei den Begriff der 'Adaptierung', den er jedoch nicht konkretisiert [Hommel 80]:

> "Portabilität ist eine Eigenschaft von Software (Programme und Daten), die es erlaubt, diese mit relativ geringem Aufwand von einem Grundsystem auf ein anderes zu übertragen. Bei dieser Übertragung dürfen andere Software-Eigenschaften nicht (Korrektheit) oder nur unwesentlich (Effizienz) beeinträchtigt werden. Es kann erforderlich sein, zusätzlichen Aufwand für die Adaptierung zu betreiben."

Adaptierung kann hier als zusätzlicher Aufwand verstanden werden, der zusätzlich zum Änderungsaufwand betrieben werden muß, falls die genannten Software-Eigenschaften beeinträchtigt werden. In einem Portabilitätsmaß P wird von Hommel der Aufwandsbegriff näher präzisiert, indem er eine Beziehung zum Implementationsaufwand herstellt:

$$P = \frac{AI}{AI + AP + AA} \quad ,$$

wobei AI der Aufwand für die erste Implementierung eines Software-Produktes ist, AP der Aufwand für die Portierung und AA der Aufwand für die Adaptierung.

Ein Software-Produkt gilt dann als portabel, wenn die Bedingung $0,5 \leq P \leq 1$ erfüllt ist. Hiermit wird verdeutlicht, was mit dem "relativ geringem Aufwand" in der Definition gemeint ist, auf den in bisherigen Definitionen nicht eingegangen worden ist.

Andere Definitionen bestimmen ebenfalls ein Portabilitätsmaß um den Aufwand näher zu beschreiben. Jedoch wird nicht von einem Maß gesprochen, sondern es wird der Begriff 'Portabilitätsgrad' eingeführt. Hierbei wird zusätzlich zum Konvertierungsaufwand auch die Konvertierungsmethode berücksichtigt. Eine Definition von Boyle geht näher auf die 'Adaptierung' von Software ein [Boyle 80]:

> "Software adaption consists of making specification-preserving changes to a piece of software in order to make it work, or to make it work efficiently, in a particular hardware and software-environment."

Für ihn ist die Adaptierung ein spezifikationserhaltender Prozeß. Wird eine Spezifikation verändert, so nennt er das eine 'Software-Modifikation'. Veränderungen in Bezug auf die Adaptabilität, sind für Poole und Waite aber gerade nicht spezifikationserhaltend [Poole 73].

> "Adaptability is a measure of the ease with which a program can be altered to fit different user images an system constraints."

Adaptierungsänderungen beziehen sich hierbei auf die Änderung der Funktion des Programms, bei gleichbleibender Umgebung und nicht, wie in ihrer Definition des Portabilitätsbegriffs, auf einen Umgebungswechsel.

In vorangegangenen Definitionen wird eine Portierung, von einer Umgebung in eine andere vorgenommen. Auch wird immer von der Software-Eigenschaft 'Portabilität' gesprochen. Eine Definition von Kaindl definiert daher den Begriff der 'Portierung' eines Programms und präzisiert dazu den Umgebungsbegriff näher [Kaindl 88]:

> "An environment E is a triple $(\{l_1,...,l_i\},o,m)$, where $\{l_1,...,l_i\}$, $i \in \mathbb{N}$, is a set of language processors, *o* is an operating system and *m* is a machine."

> "A port is a mapping $(P, E(\{l_1,...,l_i\},o,m)) \dashrightarrow (P', E'(\{l'_1,...,l'_i\},o',m'))$, where P is the program to be ported, P' the resulting program after the port, E the original and E' the target environment. The following condition must be hold:
> $(\{l_1,...,l_i\} \neq \{l'_1,...,l'_i\}) \vee (o \neq o') \vee (m \neq m')$.

Hierbei besteht eine Umgebung aus einer Menge von Compilern, zusammen mit dem Betriebssystem und dem Rechner, auf dem diese installiert sind. Damit ist eine Portierung die Übertragung eines Programms von einer ursprünglichen Umgebung in eine Zielumgebung, bei der die obige Bedingung gilt. Dadurch wird ausgeschlossen, daß eine Übertragung ohne Änderungen als Portierung bezeichnet wird.

Ford definiert im Zusammenhang mit erforderlichen Adaptierungsänderungen den Begriff 'Transport' von Software [Ford 77]:

> "A program is transportable between configurations a and b if it will compute to prescribed levels of accuracy and efficiency in each computing environment with automated changes made by a computer program."

Lassen sich erforderliche Änderungen automatisch durchführen, so spricht Ford nicht von Portierung, sondern von 'Transport' von Software.

In den 90er-Jahren definiert Mooney die Portabilität von Programmen in Abhängigkeit zum vorhandenen Aufwand, der für die Adaptierung und den Transport notwendig ist [Mooney 90]:

> "An application is portable across a class of environments to the degree that the effort required to transport and adapt it to a new."

Auf die Begriffe 'Transport' und 'Adaptierung' von Programmen, im Zusammenhang mit dem Prozeß einer Portierung, geht er wie folgt ein:

> "The porting process has two principal components which may be called adaption and transportation. Adaption is any modification that must be performed on the original version. Transportation is physical movement, with low-level adaptation."

Adaptierung bedeutet also die Änderung von Programmen im Prozeß der Portierung, und Transport bezeichnet hierbei die physikalische Übertragung von Programmen auf die Zielumgebung. Mit 'low-level adaption' sind Anpassungen bzgl. eines unterschiedlichen Datenformats gemeint.

Den Begriff 'Portierung' definiert Mooney wie folgt [Mooney 97]:

> "Porting is the act of producing an executable version of a software unit or a system in a new environment, based on an existing version."

Portierung ist demnach die Erstellung eines ausführbaren Software-Systems, basierend auf einer bereits vorhandenen Version dieses Systems. Unter dem Begriff 'Software-Unit' versteht er Anwendungssoftware, System-Software und Teile dieser Software. System-Software ist dann eine Sammlung von Software-Units.

Balzert beschreibt Portabilität mit folgenden Worten [Balzert 96]:

> "Portabilität, genauer gesagt Anwendungs-Portabilität, bedeutet, daß die Konzepte, die bei der Erstellung der Anwendungssoftware benutzt werden, auf unterschiedlichen Systemen verschiedener Hersteller zur Verfügung stehen."

Des weiteren werden hier Abstufungen zur Portabilität unterschieden: Objektcode-Portabilität (binäre Portabilität), Quellcode-Portabilität und Entwurfs-Portabilität. Objektcode-Portabilität bezeichnet die Ausführbarkeit von lauffähigen Anwendungen auf verschiedenen Plattformen. Quellcode-Portabilität kennzeichnet die Übertragbarkeit eines Quellcodes von einer Plattform auf eine andere Plattform, wobei der Quellcode auf der neuen Plattform neu übersetzt und anschließend ausgeführt werden kann.

Entwurfs-Portabilität ist der Entwurf einer Anwendung, basierend auf Konzepten, die leicht in andere Implementierungen transformiert werden können.

Es zeigt sich, daß noch kein einheitliches Verständnis in Bezug auf den Portabilitätsbegriff besteht. Es lassen sich jedoch Tendenzen erkennen, die für die Festlegung eines Sprachgebrauchs dieser Arbeit Verwendung finden können. Aus diesem Grund soll im nächsten Abschnitt der Versuch unternommen werden, den begrifflichen Rahmen dieser Arbeit aus den einzelnen Definitionen und Begriffsvorstellungen abzuleiten.

2.1.1.2 Festlegung des Sprachgebrauchs

In diesem Abschnitt soll festgelegt werden, welche Bedeutung die Begriffe ‘Portabilität’ und ‘Portierung’ im Rahmen dieser Arbeit haben sollen. Weiterhin soll auch geklärt werden, in welcher Beziehung die verschiedenen aufgeführten Begrifflichkeiten zum Thema Portabilität stehen. Wichtig bei der Festlegung des Begriffs Portabilität ist zum einen das Verhältnis zur alternativen Neuimplementation und zum anderen die Beschränkung auf konkrete Umgebungen, in deren Rahmen eine Portierung stattfindet.

Unter Portabilität eines Software-Systems soll deshalb die Übertragbarkeit des Systems oder Elemente des Systems, von einer Umgebung in eine davon verschiedene Umgebung, verstanden werden. Portierung kennzeichnet den Prozeß des Übertragens des Software-Systems oder der Systemelemente. Ein Software-System besteht demnach aus Systemelementen bzw. Systemkomponenten. Ein Software-System kann entweder Systemsoftware – ermöglicht den Betrieb und die Wartung von Hardware, z.B. Betriebssystem, Compiler etc. – oder Anwendungssoftware sein. Anwendungssoftware löst die Aufgaben des Anwenders mit Hilfe eines Computersystems und setzt auf Systemsoftware und Hardware auf.

Anwendungssoftware, Systemsoftware und Hardware bilden zusammen ein Computersystem bzw. DV-System. Unter Umgebung soll aber nicht nur die Hardware- und Software-Umgebung, im Sinne eines Computersystems verstanden werden, sondern auch das Umfeld, in dem das Computersystem existiert. Dazu gehören technische- und organisatorische Systeme (Benutzer, Anwender, Mitarbeiter), welche zusammen mit dem Computersystem ein rechnergestütztes Informationssystem/Anwendungs-system bilden. Unter einer Umgebung soll somit ein rechnergestütztes Informationssystem verstanden werden, also der komplette Umfang an Elementen, der mit dem zu portierenden Software-System interagiert. Der von vielen Autoren benutzte Begriff ’Plattform’ bezieht sich nur auf das Betriebssystem und die verwendete Hardware. Der Hardwarebegriff bezieht sich auf alle materiellen Teile eines Computersystems (Zentraleinheit, Ein/Ausgabesteuerung, Peripheriegeräte).

Die Übertragbarkeit eines Software-System wird durch den Portabilitätsgrad gekennzeichnet. Ein Software-System ist in dem Maße portabel, je höher die Kosten – verursacht durch den Änderungsaufwand – für eine Neuentwicklung, im Vergleich zur Portierung sind. Ein Software-System wird durch seinen Portabilitätsgrad – Funktion von Portierungs- und Neuentwicklungskosten, bzgl. einer bestimmten Zielumgebung – charakterisiert. Eine Übertragung die keine Kosten verursacht, soll nicht als Portierung verstanden werden.

Eine Adaptierung kennzeichnet die spezifikationserhaltende Änderung des Software-Systems im Prozeß der Portierung bzgl. Funktion, Korrektheit oder Effizienz. Eine Modifikation ändert die Spezifikation aufgrund von Benutzeranforderungen, bei gleichbleibender Umgebung. Unter Transport soll die physikalische Übertragung des Software-Systems, von einer Umgebung in eine andere Umgebung, verstanden werden.

Die Umgebung aus der das Software-System portiert werden soll, wird als Hostumgebung bezeichnet. Ein Software-System wird immer in eine Zielumgebung portiert. Wird ein konkretes Software-System von einer bestimmten Hostumgebung in eine konkrete Zielumgebung portiert, so soll der spezifische Rahmen, in dem die Portierung stattfindet, als Portierungsfeld bezeichnet werden.

2.1.2 Allgemeine Probleme bei der Portierung

In diesem Abschnitt soll ein Überblick darüber gegeben werden, welche Probleme im Rahmen einer Portierung auftreten können. Portierungsprobleme entstehen meist dadurch, daß sich Host- und Zielumgebung durch die Hardware des Computersystems und der darauf aufsetzenden Systemsoftware (Betriebssystem, Compiler und Programmiersprachen) unterscheiden.

2.1.2.1 Hardware eines Computersystems

Bei der Portierung spielt die Rechnerarchitektur eine wesentliche Rolle. Rechnerarchitekturen unterscheiden sich primär in der internen Darstellung von Daten, der Registerverwendung sowie der Speicherorganisation. Ein weiteres Problem sind die Vielzahl verschiedener Peripheriegeräte der einzelnen Computersysteme.

Rechnerarchitektur

Die interne Darstellung von Daten unterscheidet sich in deren Wortlänge. Man unterscheidet hierbei Wortlängen von 8-, 16-, 32- oder 64 Bit. Von ihr hängt z.B. die Größe der darstellbaren, ganzzahligen Datentypen, die Genauigkeit von Fließkommawerten oder die mögliche Anzahl von Zeichen eines Zeichensatzes pro Maschinenwort ab. Es gibt verschiedene Formate für die Repräsentation von numerischen Daten, die zu speziellen Problemen führen. Dazu gehören vorzeichenlose, ganzzahlige Werte (binär, binär-dezimal usw.) sowie negative, ganzzahlige Werte im 2er- oder 1er-Komplement. Durch das 2er-Komplement lassen sich, z.B. durch binäre Shift-Operationen, schnelle Rechenoperationen durchführen, die beim 1er-Komplement, jedoch nur für positive Zahlen, korrekte Werte liefern. Bei den Fließkommazahlen wurden diesbezüglich schon formale Standards verabschiedet, welche die interne Darstellung von Fließkommazahlen mit den darauf verwendeten Operationen definieren (z.B. IEEE 754: Binary Floating Point Arithmetic). Des weiteren können unterschiedliche Zeichensätze ebenfalls zu Problemen bei der Portierung führen. Der ASCII-Standard hat hier zwar eine weitverbreitete, gemeinsame Basis für viele Rechner geschaffen, jedoch existieren sehr viele inkompatible Zeichensätze sowie Erweiterungen für Sprachen mit umfangreichem Zeichensatz und Sonderzeichen.

Bei der Registerverwendung entstehen die Probleme infolge der unterschiedlichen Nutzung durch den Rechner. Die Hauptunterschiede liegen hierbei in der Verwaltung von Zwischenergebnissen. Diese

müssen explizit gespeichert werden, falls nur ein Arithmetikregister zur Verfügung steht. Bei Rechnern mit wenigen Registern und einem Stack für Operanden und Ergebnisse, erfolgt die Speicherung automatisch. Explizite Speicherung von Zwischenergebnissen, z.B. bei Rechnern mit einer Anzahl von Allzweckregistern-Operanden – werden aus dem Speicher geholt – oder bei Rechnern mit Registersets – Operanden müssen in den Registern stehen –, ist nur dann erforderlich, wenn die Anzahl der gleichzeitigen Zwischenergebnisse zu groß wird. Somit kann ein Software-System, welches Stack-Instruktionen verwendet, nicht so leicht auf einen Zielrechner portiert werden, der über keine Stack-Instruktionen verfügt. Hierbei ist eine Anpassung, in Form einer Stack-Simulation, notwendig. Weiterhin können unterschiedliche Instruktionssätze ebenfalls zu Portierungsproblemen führen.

Die Art der Speicherorganisation hat ebenfalls großen Einfluß auf die Portierbarkeit eines Software-Systems. Man unterscheidet hierbei linearen, segmentiert-linearen und hierarchischen Speicher. Bei linearen Speichern ist der Adreßraum fortlaufend adressierbar. Bei segmentiert-linearem Speicher ist der Adreßraum in einzelne, unabhängige Segmente aufgeteilt und jede Adressierung läuft über Segmentadresse und Offset. Der hierarchische Speicher verfügt über mehrere Adreßraumebenen mit unterschiedlichen Geschwindigkeiten, wobei jede Ebene unabhängig von den anderen adressiert wird. Dabei ist ein zusätzlicher Mechanismus für den Datentransfer zwischen den Ebenen notwendig. Auch spielt, neben der Organisation des Speichers, die unterschiedliche Größe der adressierbaren Speichereinheiten, und der Umfang des jeweils zur Verfügung stehenden Arbeitsspeichers eine Rolle.

Peripheriegeräte

Eine große Anzahl von Peripheriegeräten kann ebenfalls ein Maß an Portierungsproblemen verursachen. In der Klasse der externen Speichermedien – magnetisch, optisch oder hybrid – stellt sich das Problem des Transports von zu portierenden Software-Systemen. Während man bei Netzwerkverbindungen mit einheitlichen Übertragungsprotokollen im Allgemeinen keine Transportprobleme zu erwarten hat, ist die Verwendung von externen Speichermedien meist mit Problemen behaftet. Nicht jeder Rechner kann mit der breiten Palette der auf dem Markt befindlichen Speichermedien, umgehen oder bietet hierfür physikalische Schnittstellen. Hierbei entsteht auch das Problem mit unterschiedlichen Dateisystemen umzugehen (siehe Abschnitt 2.1.2.2). In der Klasse der Ausgabegeräte (Drucker, Bildschirm usw.) existieren Portierungsbarrieren im Bereich der unterschiedlichen Zeichendarstellungen. Bei einem Drucker können beispielsweise die maximale Zeilenlänge, die Steuerzeichen oder die benutzten Zeichensätze differieren. Auch bei der Bildschirmausgabe hat sich noch kein einheitlicher Standard etabliert, der die zeichen-, zeilen- oder sogar seitenweise Verarbeitung von Zeichenketten definiert. Auch führen unterschiedliche End-of-Line-Konventionen oder verschiedene Steuerzeichen zu Portierungsproblemen.

2.1.2.2 Systemsoftware

Systemsoftware ist Software, die für eine spezielle Hardware oder eine Hardwarefamilie entwickelt wurde, um den Betrieb und die Wartung der Hardware zu ermöglichen, bzw. sie zu erleichtern. Zur Systemsoftware zählt man das Betriebssystem, in der Regel aber auch Compiler, Datenbanken, Kommunikationsprogramme und spezielle Dienstprogramme. In diesem Abschnitt sollen die Portierungsprobleme angesprochen werden, die im Zusammenhang mit unterschiedlichen Compilern bzw. Programmiersprachen und den darunterliegenden Betriebssystemen auftreten können.

Ein Betriebssystem läßt sich aus zwei Perspektiven betrachten. Zum einen ist es ein Software-System, welches die realen Eigenschaften der Hardware vor dem Benutzer bzw. Programmierer versteckt und eine einfache Abstraktion auf hohem Niveau, in Form von benannten Dateien und Schnittstellen für die Programmierung liefert (Top-Down-Sicht). Unter diesem Blickwinkel ist es die Aufgabe des Betriebssystems, dem Benutzer ein Äquivalent einer erweiterten Maschine bzw. einer virtuellen Maschine zu präsentieren, die über geeignete Schnittstellen leichter zu programmieren ist, als die darunterliegende Hardware. Zum anderen hat das Betriebssystem die Aufgabe der Verwaltung aller Bestandteile eines komplexen Computersystems (Bottom-Up-Sicht). Dazu gehören Prozeßverwaltung, Verwaltung der Ein/Ausgabe-Geräte, Speicherverwaltung und Dateiverwaltung. Anhand dieser beiden Sichtweisen lassen sich die Portierungsprobleme in Bezug auf die Systemsoftware in zwei Klassen aufteilen: Probleme aus Programmierer- bzw. Benutzersicht und Probleme aus betriebssysteminterner Sicht.

Betriebssystem aus Sicht des Programmierers (Top-Down)

Aus Sicht des Programmierers nutzt man die Schnittstellen eines Betriebssystems größtenteils über bereitgestellte Schnittstellen-Bibliotheken. Die Nutzung erfolgt durch Einbindung in eine bestimmte, von der Bibliothek unterstützte, höhere Programmiersprache (z.B. C, C++, FORTRAN, Ada). Ein Compiler erzeugt aus den System-Bibliotheken und dem Programm eine ausführbare Binärdatei für ein bestimmtes Computersystem. In besonderen Fällen nutzt man zusätzlich die Hilfe von Assemblersprachen. Damit ist es möglich spezielle Zugriffe auf die Hardware zu realisieren, die die Abstraktionsebene der Betriebssystem-Schnittstelle umgehen, um spezielle Anforderungen an die Effizienz eines Software-Systems zu erfüllen. Solche Programmteile sind in der Regel immer mit Portierungsproblemen behaftet.

Bei der Benutzung von Betriebssystemschnittstellen durch höhere Programmiersprachen, stellt sich zunächst das syntaktische Problem der unterschiedlichen Sprachdialekte. Die Ursache hierfür sind Spracherweiterungen oder das Weglassen von Sprachkonstrukten bei verschiedenen Compilern. Dabei treten Portierungsprobleme immer dann auf, wenn die Portierung auf einen Zielrechner erfolgt, dessen Compiler einen geringeren Sprachumfang realisiert, als der Compiler auf dem Hostrechner. Eine Lösung hierzu bieten Sprachstandards, die einen einheitlichen Sprachdialekt für verschiedene Compiler einer Programmiersprache definieren. Aber auch hier treten Portierungsprobleme durch unterschiedliche Repräsentation von eingebauten, maschinenabhängigen Datentypen auf, die nicht vollständig im Sprachstandard erfaßt sind.

Neben den Syntaxproblemen gibt es aber auch Probleme im semantischen Bereich von Programmiersprachen, welche dahingehend zumeist nicht exakt genug definiert sind. Dazu gehören z.B. die Reihenfolge der Auswertung bei Mehrfachbedingungen, die Werte von Schleifenvariablen nach dem Verlassen einer Schleife sowie Mechanismen zur Parameterübergabe. Hier sollte beispielsweise überprüft werden, ob eine voreingestellte Art zur Parameterübergabe, wie z.B. ’call by value’ oder ’call by refererence’ etc. existiert.

Ein weiterer Aspekt von verwendeten Compiler auf dem Hostsystem, ist das Nichtvorhandensein dieser auf dem Zielsystem. Dieser Fall tritt häufig dann ein, wenn auf einem neuen Computersystem ältere Programmiersprachen nicht mehr unterstützt werden. Es ist z.B. nicht trivial, dynamische Datenstrukturen – wie z.B. Zeiger – auf ein Zielsystem zu portieren, dessen Compiler diesen Datentyp

nicht unterstützt. Weitere Compilerprobleme betreffen die Durchführung von Laufzeit-Prüfungen, z.B. die Überprüfung von Array-Grenzen, zulässige Länge von Bezeichnern oder auch die Fehlerbehandlung allgemein.

Betriebssysteminterne Sicht (Bottom-Up)

Betrachtet man ein Betriebssystem aus der Sicht der Verwaltung der Bestandteile des zugrundeliegenden Computersystems, so lassen sich die Portierungsprobleme den einzelnen Aufgabenbereichen zuordnen.

Bei der Prozeßverwaltung geht es um die Organisation von, in der Ausführung befindlichen, Programmen bzw. Prozessen. Hierbei entscheidet ein Scheduling-Algorithmus – Round-Robin, Shortes-Job-First etc. – über die Reihenfolge, mit der die einzelnen Prozesse bearbeitet werden. Dies kann bei einer Portierung, wobei sich Host- und Zielsystem durch unterschiedliche Zuteilungsalgorithmen auszeichnen, dazu führen, daß die Laufzeit eines Software-Systems nicht mehr den gestellten Anforderungen entspricht.

Weiterhin ist die Überwachung und Steuerung von Ein/Ausgabegeräten eine zentrale Aufgabe eines Betriebssystems. Hierbei können Probleme auftreten, wenn gewisse Restriktionen in Bezug auf die Zahl von zulässigen Betriebsmitteln bestehen oder sich die Anzahl der zur Verfügung stehenden Betriebsmittel unterscheidet. Beispielsweise könnte ein Betriebssystem eine virtuelle Speicherverwaltung, wie z.B Paging, unterstützen, wodurch umfangreiche Speicheranforderungen ermöglicht werden. Bei anderen Betriebssystemen, die diese Funktion nicht unterstützen, kann dies zu größeren Portierungsproblemen führen.

Die Speicherverwaltung hat die Aufgabe, den freien und belegten Speicherplatz zu verwalten. Hierbei treten, neben den bereits angesprochenen Problemen in Bezug auf die Speicherorganisation, Portierungsprobleme durch unterschiedliche Speicherverwaltungssysteme auf. Man unterscheidet zwischen Verwaltungssystemen, die Prozesse während ihrer Ausführung zwischen dem Hauptspeicher und der Platte hin und her transferieren, Swapping oder Paging, und solchen die es nicht tun, z.B. Systeme mit Einprogrammbetrieb. Ein Portieren zwischen solchen Computersystemen kann ebenfalls Probleme verursachen.

Dateiverwaltung und unterschiedliche Dateisysteme verursachen bei der Portierung ebenfalls viele Schwierigkeiten. Die wichtigsten Unterschiede zwischen Dateiverarbeitungssystemen sind:

- Dateinamen, d.h. deren zulässige Länge, sowie die gültigen Zeichen,
- Zugriffsmethoden auf einzelne Datensätze (sequentielles Lesen oder wahlfreier Zugriff),
- Sicherungsmethoden für Dateien, d.h. in der Art, wie Zugriffsrechte vergeben werden,
- Operationen für die Bearbeitung von Dateien sowie Restriktionen in Bezug auf deren Verarbeitung (maximale Blockgröße oder maximal gleichzeitig geöffneter Files).

Probleme bei der Portierung treten häufig immer dann auf, wenn Software-Systeme nicht die allgemein verbreiteten Dateiverarbeitungsfunktionen gebrauchen.

Sonstiges

Weitere Probleme treten auch immer dann auf, wenn Betriebssysteme eigene Mechanismen in Bezug auf Sonderfunktionalität anbieten, beispielsweise die Unterstützung der Kommunikation zwischen laufenden Software-Systemen.

Auch können die Menge der Systemaufrufe sowie der verwendete Kommandointerpreter, Probleme bereiten. Bei den Systemaufrufen, die bestimmte Funktionen des Betriebssystems zur Verfügung stellen, können je nach Betriebssystem, z.B. die Abfragen von Systeminformationen, auf unterschiedlichen Funktionsaufrufen basieren. Kommandointerpreter können sich in ihrer Syntax, Semantik, Mächtigkeit oder in einzelnen Befehlen erheblich voneinander unterscheiden und somit Portierungsprobleme verursachen.

Auch die Fehlerbehandlung eines Betriebssystems spielt für die Portierung eine Rolle und kann Probleme verursachen. Einige Betriebssysteme haben spezielle Fehlerbehandlungsroutinen, andere ignorieren bestimmte Fehler oder führen zum sofortigen Programmabbruch, wodurch viele Portierungsfehler im Verborgenen bleiben.

Es hat sich gezeigt, daß die Probleme bei der Portierung von Software-Systemen relativ vielschichtig sind und eine Aussage über die Portabilität von Software-Systemen eine Vielzahl der allgemein möglichen Portierungsprobleme berücksichtigen muß.

2.1.3 Strategien, Konzepte und Werkzeuge

Um Portierungsprobleme zu bewältigen, existieren eine Reihe von Konzepten, Strategien und Werkzeugen. Dieses Kapitel soll einen Überblick darüber geben, welche Lösungen für Portierungsprobleme existieren. Hierbei spielt das Prinzip der Uniformität eine zentrale Rolle, wobei man versucht, durch einheitliche Schnittstellen ein hohes Maß an Portabilität zu gewährleisten.

2.1.3.1 Sprachebene

Dieser Teil beschreibt die Konzepte, Strategien und Portierungswerkzeuge für Probleme, die den Bereich der Compiler und Programmiersprachen betreffen. Hierzu gehören in erster Linie die Entwicklung von einheitlichen Sprachstandards und Standardbibliotheken sowie die Verwendung von Werkzeugen wie Filtern, Prä- und Makroprozessoren, portablen Compilern und Translatoren.

Standards

Die Idee bei der Entwicklung von Sprachstandards besteht darin, einen einheitlichen Sprachdialekt für eine gegebene Programmiersprache zu definieren, der von allen Compilern dieser Sprache akzeptiert wird. Damit kann eine einheitliche Basis für die Übertragung von Software-Systemen, die innerhalb dieses Sprachstandards entwickelt wurden, hergestellt werden. Ein Standard sollte eine Vielzahl von Systemabhängigkeiten verbergen und durch eine einheitliche Definition handhabbar machen. Wichtige Sprachstandards existieren für die Sprachen: FORTRAN (ISO/IEC 1539: Programming Language FORTRAN (1991)), COBOL (ANSI X3.23: Programming Language COBOL (1985)), Ada (ISO

8652: Programming Language Ada(1995)) oder C (ANSI X3.159: Programming Language C) bzw. C++ (ISO/IEC: 14882, 1998-09-01, International Standard, Programming Languages C++).

Einige Probleme, die bei der Entwicklung eines Sprachstandards auftauchen, sollen am Beispiel der Sprache FORTRAN aufgezeigt werden. Hierbei wurden erste Erfahrungen bei der Entwicklung von Sprachstandards bereits in den 70er-Jahren gesammelt [Infotech 80]. Ein älterer FORTRAN-Standard ist FORTRAN ANSI 66 bzw. FORTRAN IV. Dieser Standard wurde aber von den meisten FORTRAN-Programmen nicht eingehalten und man legte restriktivere Untermengen fest, die eine höhere Portabilität von FORTRAN-Programmen gewährleisten sollten. Einer dieser Standards ist Portable- FORTRAN (PFORT), für den ein eigener Filter implementiert wurde, welcher FORTRAN-Programme auf die Einhaltung des PFORT-Standards überprüft. Ein anderer FORTRAN-Dialekt wurde als Compatible-FORTRAN (CF) bezeichnet. Die Erfahrungen gingen dann in einen neuen Standard (FORTRAN 77) ein, der einige Erweiterungen gegenüber FORTRAN 66 enthielt. Da aber der 66er-Standard keine echte Teilmenge des 77er-Standards war, gab es Probleme bei der Portierung von alten FORTRAN-Programmen auf Rechner mit FORTRAN-Compilern, die den 77er Standard unterstützten. Bei der Entwicklung eines Sprachstandards mußten also Kompromisse in Bezug auf existierende Programme eingegangen werden und es hat sich gezeigt, daß die nachträgliche Festlegung eines Standards für eine existierende Programmiersprache nicht unproblematisch ist. Hierbei müßten entweder alle Dialekte bei der Definition berücksichtigt werden – mit zum Teil widersprüchlichen Forderungen – oder bestehende, ältere Programme werden als nicht portabel deklariert.

Nicht nur die Entwicklung und Einführung eines Sprachstandards gestaltet sich schwierig, sondern auch die Einhaltung bei der Entwicklung von Software-Systemen, ist keine Garantie für dessen Portabilität. Zwar werden sehr viele Systemabhängigkeiten von einem Sprachstandard heute bereits abgedeckt, jedoch gibt es weitere Abhängigkeiten, die aus Portierungssicht immer noch nicht zufriedenstellend vereinheitlicht wurden. Hierzu gehören die bereits aufgeführten Probleme wie: Steuerung von Ein/Ausgabegeräten, Dateisysteme, Wertebereiche von Datentypen, Adressierung sowie die Prozeßverwaltung (Prozeß-Scheduling und -Synchronisation).

Ein weiteres Problem im Zusammenhang mit Sprachstandards ist darin zu sehen, daß sie meist keine Aussagen über die Zulässigkeit von Spracherweiterungen enthalten. Auch ist die allgemeine Akzeptanz bei den Compilerherstellern von entscheidender Bedeutung für die Definition eines Standards und dessen Verbreitung.

Standardbibliotheken

Die Idee bei Standardbibliotheken folgt ebenfalls dem Einheitlichkeitsprinzip. Hierbei sollen uniforme Schnittstellen zum Betriebssystem und weiteren Funktionalitäten, die Portabilität von System-Software, welche mit Hilfe solcher Bibliotheken entwickelt wurden, sichern. Es existieren eine Reihe von standardisierten Bibliotheken (C++ Standard Template Bibliothek, NAG für Numerische Bibliotheken, CORBA-Implementierungen etc.), die strenggenommen kein Teil der zugrunde liegenden Programmiersprache sind. Sie werden jedoch meist gleichzeitig mit ihr definiert und in jeder Umgebung, in der die Sprache eingesetzt wird, vorausgesetzt. Bibliotheken decken ein breites Spektrum an Funktionalitäten ab, z.B. Ein/Ausgabe, String-Operationen, dynamische Speicherverwaltung. Diese arbeiten aber häufig eng mit dem Betriebssystem zusammen, sodaß sie trotzdem nichtportable Teile beinhalten können. Standardbibliotheken stellen in diesem Zusammenhang die Interaktion mit dem Betriebssystem auf eine portable Basis und sichern zugleich

die einheitliche Funktionalität der benutzten Bibliotheksfunktionen. Hält man sich bei der Entwicklung eines Software-Systems streng an vorhandene Standardbibliotheken, so sollte sich bei einem Umgebungswechsel das Verhalten und die Funktionalität nicht ändern. In der Praxis reicht die Funktionalität von Standardbibliotheken meist nicht aus, sodaß auf zusätzliche, und häufig nichtportable, Bibliotheksfunktionen außerhalb des Standards zurückgegriffen werden muß.

Werkzeuge

Ein Teil der Portabilitätsprobleme, die durch unterschiedliche Sprachdialekte auftreten, werden mit Hilfe von Werkzeugen wie Filtern, die eine Einhaltung von Sprachstandards überprüfen, angezeigt [Brown 77], [LeCarme 89]. Filter arbeiten mit syntaktischer- und teilweise mit semantischer Analyse. Bei der syntaktischen Analyse wird der Quelltext auf Übereinstimmung mit der Syntax einer zugrundeliegenden Programmiersprache überprüft. Die semantische Analyse überprüft den inhaltlichen Teil, dazu gehört die statische Semantik, z.B. Variablentypen oder Gültigkeitsbereiche und die dynamische Semantik, z.B. Ausführungsreihenfolge und Initialisierungen. Allen Filtern ist gemeinsam, daß sie ein Software-System mit einem gegebenen Sprachstandard vergleichen, wobei der Analyseumfang recht unterschiedlich sein kann. In diesem Zusammenhang wird auch der Begriff 'Verifier' gebraucht. Zum einen versteht man darunter dynamische Debugging-Tools für Programme, zum anderen bezeichnen sie Werkzeuge für die entsprechenden Teile der dynamische Analyse. Legt man die erstere Auffassung zugrunde, so gehört zum Analyseumfang des Filters auch die dynamische Analyse.

Neben dem Einsatz von Filtern gibt es Werkzeuge wie Präprozessoren, die verschiedene Sprachdialekte ineinander transformieren [LeCarme 89]. Diese übernehmen Aufgaben wie Makro-Bearbeitung, Einkopieren von Dateien, Anreicherung älterer Sprachen mit modernen Möglichkeiten der Kontrollflußmanipulation, Datenstrukturierung (rationale Präprozessoren) sowie Erweiterung von existierenden Sprachen. In den letzten beiden Aufgaben kommt der Portabilitätsaspekt zum Tragen. Rationale Präprozessoren ermöglichen das Einbinden neuer Kontroll- und Datenstrukturierungskonzepte. Ein solcher Präprozessor könnte z.B. eingebaute Makros für while- oder if-Anweisungen anbieten, falls diese in der Programmiersprache selbst nicht existieren. Präprozessoren, die den Erweiterungsaspekt in Programmiersprachen abdecken, versuchen, der Sprache in Form eingebauter Makros, neue Möglichkeiten hinzuzufügen. Die Idee von Präprozessoren, im Rahmen der Portierung von Software-Systemen, ist es, bestehende Sprachen so auszubauen, daß sie den modernen Anforderungen entsprechen. Mit Hilfe solcher Konzepte wurden weitverbreitete Sprachen, wie FORTRAN, so ausgebaut, daß mit ihnen ältere Software-Systeme an neue Konzepte angepaßt werden konnten (RATFOR). Auch können in einigen Sprachen mit Hilfe von Präprozessoren bestimmte Systemabhängigkeiten durch bedingte Kompilierung, effektiv behandelt werden. Diese sind jedoch schwierig zu verwalten und sollten nur in Software-Systemen mit geringem Umfang verwendet werden. Präprozessoren sind größtenteils eng mit einer Programmiersprache verbunden (z.B. C-Präprozessor) oder sind explizit für eine Programmiersprache entwickelt worden (z.B. RATFOR).

Eng im Zusammenhang mit Präprozessoren stehen die meist darin enthaltenen oder auch eigenständigen Makroprozessoren. Diese untersuchen ein Programm auf die Benutzung sogenannter Makros, bzw. deren Aufrufe [LeCarme 89]. Makroaufrufe werden vom Makroprozessor durch definierte Zeichenketten ersetzt, die in Makrodefinitionen angegeben worden sind. Im Rahmen der Portabilität von Software-Systemen werden Makroprozessoren dafür eingesetzt,

Systemabhängigkeiten durch entsprechende Makrodefinitionen zu beherrschen. Man unterscheidet hierbei drei Kategorien von Makroprozessoren: Makroassembler, syntaktische Makroprozessoren und General-purpose-string-processing-Makroprozessoren (z.B. STAGE2). Letztere spielen bei der Portabilität die wichtigste Rolle, weil sie nicht an eine bestimmte Assemblersprache gebunden sind und somit eine größere Umgebungsunabhängigkeit bieten. Makrodefinitionen, die Systemabhängigkeiten beinhalten, werden hierbei als Definition einer Art abstrakten Maschine aufgefaßt. Auf dieser läuft dann die portable Software, welche in der entsprechenden Makrosprache formuliert wurde. Um das Software-System in verschiedenen Systemumgebungen ausführen zu können, müssen die Makrodefinitionen jeweils angepaßt werden. Stehen mehr sprachliche Aspekte im Vordergrund, wird eher der Begriff ’Zwischensprache’ anstatt ’abstrakte Maschine’ benutzt.

Ein anderer Ansatz, Portierungsprobleme bzgl. der Unterschiede der Sprachdialekte in den Griff zu bekommen, besteht darin, den Compiler selbst so portabel zu gestalten, daß er mit seinem akzeptierenden Sprachdialekt in jeder beliebigen Umgebung zur Verfügung steht (z.B. Unix Portable C Compiler, Amsterdam-Compiler-Kits (Tanenbaum)). Sind Compiler in ihrer Architektur so angelegt, daß sie schnell an verschiedene Umgebungen angepaßt werden können, so spricht man von portablen Compilern. Der entscheidende Faktor für die Portierung von Compilern ist die Schnittstelle zwischen Front- und Back-End. Die Front-End-Komponente – lexikalische, syntaktische und semantische Analyse – kann dabei meist maschinenunabhängig realisiert werden und ist bei portablen Compilern der unkritischere Teil. Die Back-End-Komponente von Compilern – Zwischencode, Codeoptimierung, Codeerzeugung – realisiert die Transformation eines Software-Systems in eine lauffähige und systemabhängige Binärversion. Dieser Teil des Compilers ist stark maschinenabhängig und muß für jede Umgebung entsprechend angepaßt werden. Hauptproblem bei der Anpassung von portablen Compilern sind die unterschiedlichen Betriebssystemaufrufe der einzelnen Umgebungen, die jeweils ein hohes Maß an Einarbeitungsaufwand erfordern. Somit bleibt die umfangreiche Schnittstelle zwischen Compiler und Betriebssystem ein offenes Portierungsproblem.

Translatoren sind Portierungswerkzeuge, welche die Quellen eines zu portierenden Software-Systems so verändern, daß sie von Compilern auf der Zielmaschine verarbeitet werden können. Man unterscheidet Higher-Level-Sprach-Translatoren und die bereits angesprochenen Präprozessoren. Erstere übersetzen ein Programm, das auf einer höheren Programmiersprache implementiert wurde, in eine andere, höhere Programmiersprache. Präprozessoren transformieren aus Portierungssicht zwischen verschiedenen Sprachdialekten. Bei den Higher-Level-Translatoren ergeben sich die größten Probleme durch die einzelnen Levelunterschiede bei den Programmiersprachen. Übersetzt man zum Beispiel ältere Software-Systeme in eine Umgebung mit einer mächtigeren Programmiersprache als die, die dem bisherigen Software-System zugrunde liegt, können die Unterschiede der einzelnen Konstrukte komplexe Kontrollflußanalysen erforderlich machen. Solche und andere Schwierigkeiten – z.B. erhebliche Performanceverluste im umgekehrten Fall – sorgen dafür, daß ein Translator nur ein unterstützendes und kein vollautomatisches Portierungswerkzeug sein kann.

2.1.3.2 Betriebssystemebene

Weitere Strategien auf dem Weg zur Portabilität von Software-Systemen sind auf der Ebene der Betriebssysteme zu finden. Auch hier versucht man durch einheitliche Standards, dem Prinzip der Vereinheitlichung zu folgen und entwickelt standardisierte Schnittstellen für Betriebssysteme. Auf der anderen Seite versucht man, das zugrunde liegende Betriebssystem so portabel zu gestalten, daß es

ohne größeren Aufwand auf verschiedene Umgebungen übertragen werden kann. Man spricht dann von portablen Betriebssystemen.

Standards

Die Idee bei der Standardisierung von Betriebssystemschnittstellen sind vereinheitlichte Zugriffe auf Komponenten des Betriebssystem und dessen peripheren Ressourcen, die eine Lauffähigkeit von System-Software in verschiedenen Umgebungen sicherstellen sollen.

Ein anerkannter Vertreter solcher Standards ist POSIX. Das ‘Portable Operating System Interface’ ist ein Dokument, das von der IEEE erarbeitet und von der ANSI und ISO standardisiert wurde. Die ’POSIX-Familie’ besteht aus mehreren Teilen, wobei der ’IEEE 1003.1: POSIX Part 1: System API’ den Teil der Betriebssystemschnittstelle für die Anwendungsprogrammierung spezifiziert und somit die Portabilität von Software-Systemen auf Quellcode-Niveau unterstützt. Ein zweiter Teil – ISO 9945-2: Shell- and Utility-Application-Interface – definiert Schnittstellen für die Kommandointerpretation auf Quellcode-Ebene sowie Betriebssystemdienste und -unterstützungen für System-Software, als Ergänzung zum ersten Teil.

Ein weiterer Standard für Betriebssystemschnittstellen ist MOSI (IEEE 855: Microprocessor Operating System Interfaces (MOSI) (1990)). Die Arbeiten an MOSI starteten in der Unix-Komune zur gleichen Zeit wie POSIX, Anfang der frühen 80er-Jahre. POSIX wurde dann von der IEEE im Jahre 1985 übernommen und steht seitdem unter dessen Obhut, was zu einer breiten Anerkennung dieses Standards führte und viele Unix-Systeme, aber auch Windows NT, POSIX-konforme Subsysteme anbieten.

Portable Betriebssysteme

Wie oben bereits erwähnt, ist die Schnittstelle zum Betriebssystem ein entscheidender Faktor für die Portabilität von Software-Systemen. Dies betrifft sowohl System- als auch Anwendungssoftware. Die oben vorgestellten Standards für Betriebssystemschnittstellen bieten einen Weg, diese Schnittstellen zu vereinheitlichen. Eine andere Strategie verfolgt das Konzept der portablen Betriebssysteme. Hierbei wird die einheitliche Schnittstelle durch das Betriebssystem selbst geschaffen, indem es auf die jeweilige Zielumgebung portiert und somit die Verantwortung für die Portabilität von Software-Systemen, auf das zugrunde liegende Betriebssystem verlagert wird.

Ein Beispiel für ein portables Betriebssystem ist Unix. Hierbei werden maschinenabhängige Funktionen in einzelnen Komponenten zusammengefaßt, die für den Rest des Betriebssystems die Hardware-Schnittstelle darstellen. Hardwarenahe Funktionen – Interrupts, Systemaufrufe etc. – werden mittels Assemblerroutinen realisiert, die zusammen mit einigen, in C implementierten, maschinennahen Funktionen, z.B. Treiber, den sogenannten Betriebssystemkern bilden und bei einer Portierung in andere Umgebungen angepaßt werden müssen. Somit beschränkt sich die Maschinenabhängigkeit nur auf einige wenige Komponenten. Der Rest des Unix-Betriebssystems ist völlig maschinenunabhängig implementiert und kann ohne weitere Anpassungen für unterschiedlichste Zielumgebungen übernommen werden.

Ein weiteres Beispiel ist MUSS [Frank 79], das Ende der 70er-Jahre an der Manchaster-Universität entworfen wurde. Die Portabilität des Betriebssystems wird durch mehrere unabhängige Komponenten

erreicht, die als virtuelle Maschinen, z.B. Ein/Ausgabe-Manager, Disk-Manager etc., dargestellt werden. Die maschinenabhängigen Funktionen, wie z.B. Speicherverwaltung und Interrupt-Handling etc., sind ähnlich zu Unix in einem Betriebssystemkern zusammengeführt. Die virtuellen Maschinen greifen ausschließlich über Bibliotheksfunktionen des Betriebssystemkerns auf Hardwarekomponenten zu, laufen jeweils unabhängig voneinander in eigenen Prozessen und kommunizieren über spezielle Mechanismen des Nachrichtenaustauschs.

Portable Betriebssysteme bieten somit nicht nur entscheidende Vorteile in Bezug auf die einheitliche Schnittstelle zum Betriebssystem, sondern auch bezüglich des geringeren Aufwands – im Vergleich zu einer Neuimplementierung –, der für die Übertragung eines portablen Betriebssystems auf bestimmte Zielumgebungen notwendig ist.

Emulatoren

Ein, im Vergleich zu portablen Betriebssystemen, umgekehrtes Konzept, nämlich die Hardware an die Software anzupassen, erschließt sich durch die Realisierung von Emulatoren. Hierbei werden die Eigenschaften der Hostumgebung auf dem Zielrechner abgebildet und ermöglichen somit die Portierung von Software-Systemen, ohne dessen explizite Anpassung. Wird die Nachbildung softwaretechnisch realisiert, so ist dies, nach Gewald [Gewald 79], ein Simulator. Bei hardwaretechnisch realisierten Abbildungen spricht man von Emulatoren. Diese Unterscheidung trifft man heutzutage jedoch nur noch selten und es hat sich für beide Formen der Begriff ’Emulator’ etabliert. Man spricht im Allgemeinen von Simulatoren, wenn reale Abläufe rechnergestützt simuliert werden. Somit soll hier nur von Emulatoren die Rede sein.

Softwaretechnisch realisierte Emulatoren werden häufig dann eingesetzt, wenn das zu portierende Software-System keine allzu großen Anforderungen an die Ausführungsgeschwindigkeit stellt. Hierin liegt auch der größte Nachteil von softwaremäßig realisierten Emulatoren. Die Ausführungsgeschwindigkeit der darauf ablaufenden Programme, ist im Vergleich zur realen Hostumgebung stark verringert. Ein Beispiel für einen solchen Emulator ist der von Doyle und Mandelberg entwickelte PDP-11-Emulator [Doyle 84]. Dieser Emulator wurde entwickelt, um Unix auf eine SPERRY-UNIVAC 90/80 – einem 32-Bit-Rechnersystem – und auf eine NOVA 3 – einen 16-Bit Minicomputer– zu portieren. Die durchschnittliche Instruktionszeit auf der SPERRY-UNIVAC stand zur PDP-11/10 im Verhältnis 120 zu 1. Dieses Verhältnis konnte durch Ersetzung bestimmter FORTRAN-Routinen durch Assemblerroutinen noch verbessert werden und man war mit der Leistung insgesamt zufrieden. Neuere Emulatoren haben die Geschwindigkeitsunterschiede auf ein Verhältnis von 2:1 verbessern können. Eine vom Autor durchgeführte Vermessung eines kommerziellen softwaretechnisch realisierten Emulators für Unix- und Windows-Plattformen, der Firma VMWare [VMWare 01], ergab beispielsweise ein Verhältnis von 3:1.

Bei der hardwaretechnischen Abbildung werden mit Hilfe von Mikroprogrammen und spezieller Hardwarekomponenten, bedeutende Geschwindigkeitsvorteile gegenüber Softwarelösungen erzielt. Der Nachteil von Hardware-Emulatoren wird darin gesehen, daß sie nur dann sinnvoll einsetzbar sind, wenn die beteiligten Maschinen eine ähnliche Architektur aufweisen [Leong 83].

2.1.3.3 Architekturebene

Wie oben gezeigt, versucht man auf Hochsprachenebene dem Prinzip der Vereinheitlichung durch einheitliche Repräsentation von Software-Systemen nachzukommen. Auf der Ebene der Betriebssysteme ist man bestrebt, einheitliche Schnittstellen zu definieren. Aber auch auf der Ebene der Maschineninstruktionen existieren Konzepte und Strategien, die Portabilität von Software-Systemen zu verbessern. Bei den hier vorgestellten Konzepten geht es ebenfalls um die Entwicklung einheitlicher Architekturstandards oder Strategien, auf der Basis generischer Architekturen.

Standards

Ist man bestrebt die Architektur von Rechnersystemen zum Zwecke einheitlicher Maschineninstruktionen zu standardisieren, bzw. Teile davon, erreicht man bei der Realisierung dieses Konzepts ein hohes Maß an Portabilität. Die Übertragung von Software-Systemen auf einzelne Zielumgebungen wäre dann ohne größere Anpassungen möglich. Eine einheitlich definierte Hardware-Architektur erlaubt die gemeinsame Nutzung verschiedener Systemsoftware auf homogen Hardwaresystemen und sorgt gleichsam für Objektcode-Portabilität dieser Software-Systeme.

Ein weitverbreiteter De-facto-Standard ist der, von vielen Firmen im Laufe der Zeit weiterentwickelte, IBM-PC. Der erste Personal Computer wurde von IBM, im Jahre 1981, auf dem Markt eingeführt und definierte damit, auf der Basis des weitverbreiteten Industriestandards, ISA, einen weltweiten, offenen Standard für sogenannte 'Personal Computer'. Dieser ist in seinen Grundzügen immer noch aktuell. Der IBM-PC basiert im wesentlichen auf einer CISC-Prozessorarchitektur, deren wesentlicher Repräsentant die Intel-X86-Familie ist. Hier steht die Objektcode-Kompatibilität, mittels Aufwärtskompatibilität durch Familienbildung im Vordergrund, ausgehend vom 8-Bit-Mikroprozessor 8080, über den 16-Bit-Mikroprozessor 8086, die 32-Bit-Mikroprozessoren 386, 486, Pentium, Pentium Pro, bis hin zum 64-Bit-Mikroprozessor P7 (Itanium). Objektcode, der für einen bestimmten Prozessor X eines IBM-PC erzeugt wird, läuft ohne weitere Modifikation auch auf einem IBM-PC mit Prozessoren der älteren Generationen.

Ein von der Firma Sun, 1987 eingeführter und von der IEEE (Std. 1754) weiterentwickelter Architekturstandard ist SPARC (Scalable Processor Architecture) [Catanzaro 91]. Dieser war die erste offene RISC-Prozessorarchitektur und ist für den Einsatz in Laptop-, Desktop-, High-End- und Multiprozessorsystemen bestimmt. Für alle SPARC Systeme, die mit UNIX-System-V laufen, wird Binärkompatibilität garantiert. Der ABI-Standard (Application Binary Interface) beschreibt das Format und den Inhalt der SPARC-ABI-Binaries. Die SPARC-Architektur definiert den Zweck der Integer-, Floating-Point- und Coprozessor-Register, den Prozessorzustand und die Statusregister, 69 Basisbefehle und einen 32 Bit breiten, virtuellen Adreßraum für User-Applikationen. Die Systemkomponenten, wie z.B. I/O-Interface, die Cache/Memory-Architektur oder eine MMU, werden nicht definiert. Die erste SPARC wurde 1987 von Fujitsu implementiert und hieß MB86900.

Viele Anstrengungen werden unternommen, die Architektur von Rechnersystemen zu vereinheitlichen. Eine Vielzahl von Architekturphilosophien und die rasche Entwicklung der Hardwaretechnologie haben aber bisher die Entwicklung eines universellen Standards verhindert.

Generische Architekturen

Bei dem Konzept der generischen Architekturen geht es um die Realisierung einer einheitlichen Pseudo-Hardware-Architektur. Diese setzt auf realen Hardware-Architekturen auf und wird mittels Laufzeitinterpreter oder Übersetzern mit ihr verbunden. Hierbei übernimmt die Pseudo-Architektur die Rolle einer universellen Hardware-Architektur und ermöglicht somit Objektcode-Portabilität, unabhängig von der zugrundeliegenden, realen Architektur von Rechnersystemen.

Ein gutes Beispiel für die Realisierung von generischen Architekturen bietet die zu Anfang der 90er-Jahre entwickelte Programmiersprache Java. Diese wurde mit dem Ziel entwickelt, plattformunabhängig zu sein. Ein Java-Compiler übersetzt hierbei ein Quellprogramm in sogenannten Java-Byte-Code. Dieser Objektcode ist plattformunabhängig und nicht direkt vom Prozessor ausführbar. Zwischen den generierten Objektcode – auch als Zwischencode bezeichnet – und die reale Hardware-Architektur wird ein Laufzeitinterpreter geschaltet, der den Zwischencode für eine bestimmte Rechnerarchitektur schrittweise analysiert und dann direkt auf dieser ausführt. Der jeweilige Java-Interpreter selbst, wird vom Prozessor ausgeführt und verdeckt die Eigenschaften des jeweiligen Prozessortyps in Form einer virtuellen Maschine. Eine andere Möglichkeit besteht darin, den Zwischencode zur Laufzeit zu übersetzen. Hierbei wird über ein Zusatzprogramm (z.B. Web-Browser) ein sogenannter Just-in-time-Compiler bereitgestellt, der den Zwischencode in ein Objektprogramm für einen speziellen Prozessortyp übersetzt und dann direkt auf diesem ausführt, ohne den Umweg über einen Interpreter. Weitere Beispiele für Java-ähnliche Konzepte ist der an der Universität von Kalifornien (Sant Diago) entwickelte, portable Pascal-Compiler (UCSD-Pascal) mit P-Code oder aber der portable BCPL-Compiler, mit O-Code als Zwischensprache [Peck 78].

Bei portablen Betriebssystemen wie MUSS, wird die generische Architektur ebenfalls durch virtuelle Maschinen realisiert, die als Zwischenschicht, bis auf die Ebene der Maschineninstruktionen abstrahieren. Der generierte Objektcode wird dann von diesen virtuellen Maschinen interpretiert und direkt auf dem zugrunde liegenden Prozessortyp ausgeführt wird.

Ein weiteres Beispiel für eine generische Architektur, ist das eingesetzte Konzept der Hardware-Abstraktionsschicht – Hardware Abstraction Layer, kurz HAL – von Windows NT. Diese zusätzliche Schicht abstrahiert Teile der darunterliegenden, realen Hardware-Komponenten und spiegelt den, bei einer Portierung auf ein anderes Rechnersystem, anzupassenden Betriebssystemkern wieder. Der Unterschied zu Betriebssystemen wie MUSS besteht darin, daß hier nur bestimmte Teile der Hardware einer generischen Architektur unterworfen sind, wie z.B. Karten-Treiber. Es wird nicht bis auf die Ebene der Maschineninstruktionen, in Form von virtuellen Maschinen, abstrahiert. Der Hardware Abstraction Layer erlaubt beispielsweise die Entwicklung von Software-Systemen, ohne Kenntnis des späteren Prozeß-Interface sowie eine dynamisch-anpassungsfähige Verkabelung. Die Zuordnung der einzelnen Prozeßsignale, zu der in Software-Systemen implementierten I/O-Funktionen, ist frei konfigurierbar und somit ohne Änderungen an unterschiedliche Schnittstellenkarten anpaßbar. Dies führt zu einer strikten Codetrennung zwischen Software-Systemen und Prozeß-Interface-Software, in diesem Fall die Karten-Treiber.

Ein Problem bei generischen Architekturen besteht in der eindeutigen Festlegung des Abstraktionsniveaus. Ist es zu hoch angesiedelt, wird unter Umständen nicht viel gewonnen Bei einem zu niedrigen Niveau leidet die Anpassung an die verschiedenen Hardware-Architekturen. Ein anderes Problem betrifft die Laufzeit von Software-Systemen. Da der bei der Übersetzung entstandene

Objektcode über einen Objektcode-Interpreter vom Prozessor verarbeitet wird, laufen solche Software-Systeme in der Regel langsamer als ihre binärkompatiblen Gegenstücke. Das Ziel hierbei ist es also, den Vorteil generischer Architekturen, mit einem Mindestmaß an Geschwindigkeitseinbußen bzw. Übersetzungsaufwand nutzbar zu machen.

2.1.4 Portierungsmodelle

Modelle haben im allgemeinen die Aufgabe, komplexe Zusammenhänge von Situationen oder Problemen überschaubar und somit handhabbar zu machen. Dabei wird von den unwesentlichen Faktoren der zu modellierenden Realität abgesehen und nur die Faktoren berücksichtigt, die für eine reale Situation oder ein reales Problem von Bedeutung sind. Um ein Verständnis von den Zusammenhängen und Abläufen eines Portierungsprozesses zu bekommen, ist es notwendig, Modelle, die diesen realen Prozeß beschreiben, aufzustellen. Ein solches Prozeßmodell bildet eine gute Grundlage für weitere Überlegungen in Richtung Meßbarkeit oder Optimierung von Portierungsprozessen. Für Portierungsprobleme existieren Modelle, die die Portabilität von Software-Systemen zum einen über dessen Beziehungen zur Systemumgebung beschreiben und zum anderen Modelle, die den eigentlichen Prozeß der Portierung behandeln. Im folgenden soll eine Auswahl an Modellen vorgestellt werden, die für das Verständnis des Portierungsprozesses von Nutzen sind.

Ein Modell, welches die Beziehungen eines Software-Systems zu seiner Systemumgebung darstellt, wird von Mooney beschrieben [Mooney 94]. Seine Sicht konzentriert sich auf die Schnittstellen zwischen dem zu portierenden Software-System und den sich ändernden Umgebungskomponenten. Die Schnittstellen-Sicht bezieht sich nicht auf den Prozeß der Portierung, liefert jedoch Hinweise auf zu berücksichtigende Abhängigkeiten zwischen den einzelnen Komponenten.

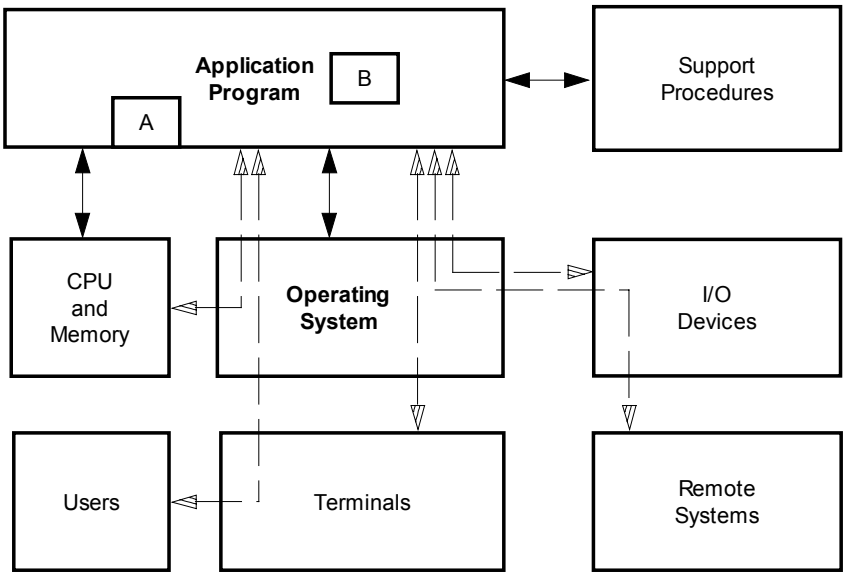


Abbildung 1: Schnittstellenmodell für die Portabilität von Software-Systemen

Das Modell in Abbildung 1 zeigt das Software-System und dessen Interaktion mit Systemkomponenten über deren Schnittstellen. In einer weiteren Verfeinerung des Modells wird das Software-System in einzelne Systemkomponenten aufgeteilt, die entweder mit der Systemumgebung

interagieren oder nicht, d.h. unabhängig von dieser sind. Die Abbildung zeigt beispielsweise die Beziehung der Software-Systemkomponente A zur Schnittstelle des Betriebssystems (direkte Schnittstelle, ausgefüllter Pfeil), welche wiederum mit dem Speicher der Systemumgebung über dessen Schnittstellen interagiert (indirekte Schnittstellen, schraffierter Pfeil). Systemkomponente B hat dagegen keine Systemabhängigkeiten. Die Portabilität eines Software-Systems ist somit direkt abhängig von den benutzten Schnittstellen seiner Komponenten. Ein wichtiger Aspekt, der bei diesem Modell nicht direkt aufgezeigt wird, ist die unterschiedliche Repräsentation solcher Schnittstellen. Einige repräsentieren sich auf Quellcodeebene, andere wiederum auf Objektcode- oder Ausführungsebene (Maschineninstruktionen). Jede Ebenenrepräsentation ist hierbei mit einem spezifischen Verlust an Portabilität behaftet. Aus diesem Grund wird dieses Modell als statisches Schnittstellenmodell bezeichnet.

Ein von ihm aufgestelltes Prozeßmodell für die Portierung zeigt Abbildung 2. Das Modell repräsentiert ein einfaches Beispiel für die Portierung eines Software-Systems auf Quellcodeebene. Hierbei wird das Quellprogramm zunächst in die Zielumgebung transportiert und dann entsprechend an diese angepaßt. Im Allgemeinen werden jedoch nur einzelne Software-Systemkomponenten portiert und die Anpassung der einzelnen Komponenten erfolgt je nach Zielstellung entweder in der Hostumgebung (Original Environment) oder in der Zielumgebung (Target Environment).

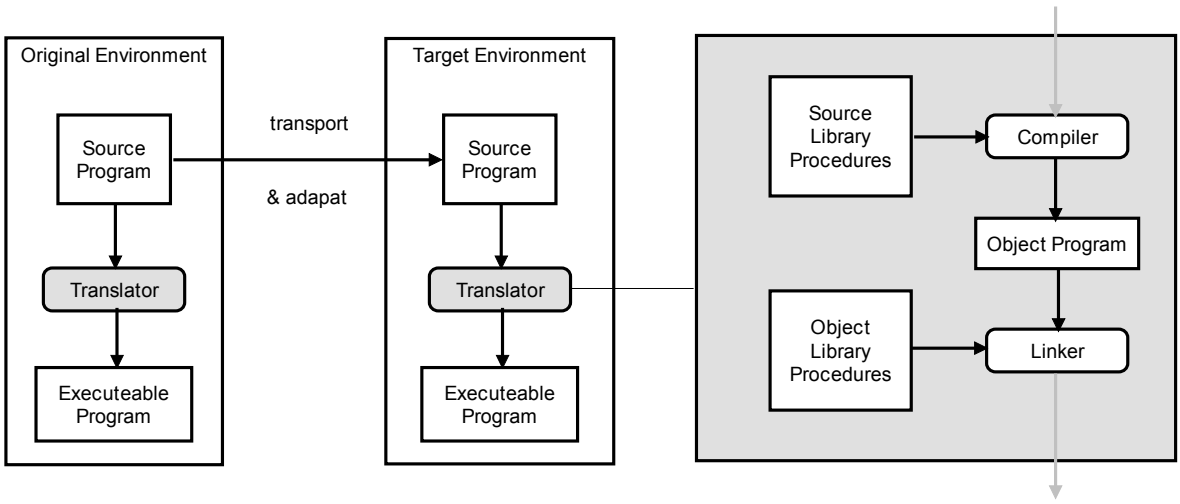


Abbildung 2: Prozeßmodell für die Portierung nach Mooney

Der Übersetzer beinhaltet den Compile- und Link-Prozeß, wobei jeweils einzelne Bibliotheken – auf Quellcode- oder Objektebene – hinzugenommen werden können. Jede Transformationsebene eines Quellprogramms beinhaltet Anpassungsmöglichkeiten an die Zielumgebung. Das Modell kann dafür genutzt werden, um festzustellen, was portiert werden muß und wo im Portierungsprozeß die Anpassungen ihren Eingang finden. Findet die Anpassung an die Zielumgebung in der Hostumgebung statt, ergibt sich ein alternatives Modell, welches in Abbildung 3 dargestellt wird und in der die Anpassung des Software-Systems vollständig in der Hostumgebung vorgenommen wird. Diese Vorgehensweise ist z.B. bei integrierten Systemen notwendig, da diese nur eingeschränkte oder überhaupt keine Mechanismen für eine Programmerstellung bereitstellen.

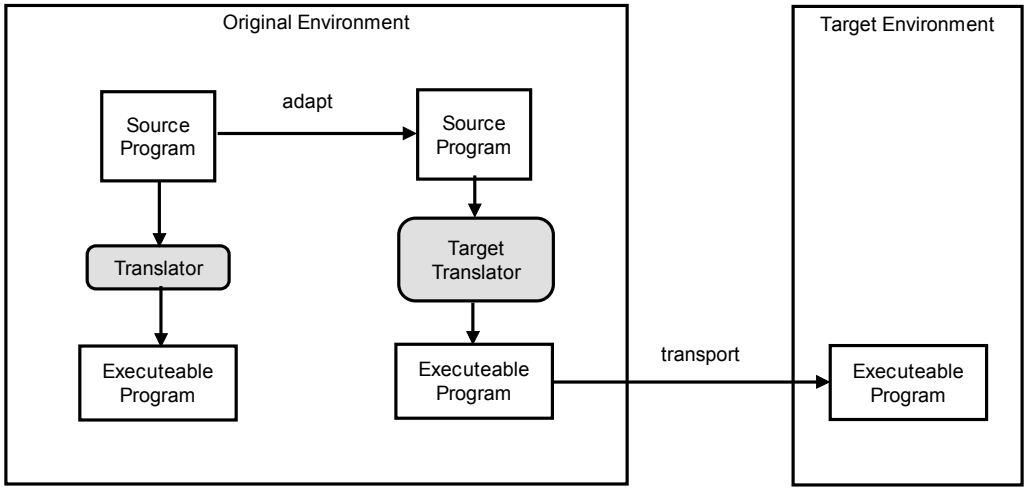


Abbildung 3: Alternatives Prozeßmodell für die Portierung (Anpassung in Hostumgebung)

Ein weiteres Modell für den Portierungsprozeß findet sich in [Buschhorn 88]. Hierbei werden bestimmte Faktoren bestimmt, die den Portierungsaufwand beeinflussen. Dazu gehören Programmiersprachen, Betriebssysteme, Hardware, Komplexität, Dokumentation, Portierungstools sowie Erfahrungen und Kenntnisse des Porteurs. Alle Faktoren werden dann in drei Gruppen eingeteilt. Die erste Gruppe besteht aus den verwendeten Programmiersprachen, den zugrundeliegenden Betriebssystemen und der Hardware des Computersystems. Alle drei Faktoren haben direkten Einfluß auf den Aufwand einer Portierung, da die Anzahl an Codestatements, die geändert werden müssen, von den Unterschieden zwischen Host- und Zielrechner abhängt. Die zweite Gruppe bilden die indirekt auf den Portierungsaufwand einflußnehmenden Faktoren. Dazu gehört die Komplexität eines Software-Systems. So sind z.B. Folgeänderungen, die Portierungsänderungen an wiederum anderen Programmanweisungen nach sich ziehen können, schwierig zu erkennen. Außerdem geht man davon aus, daß in einem ersten Schritt nicht alle anzupassenden Anweisungen erkannt werden, sodaß umfangreiche Verifikationstests erforderlich sind, die noch dadurch erschwert werden, daß keine ausreichende Dokumentation zu den Quellen vorhanden ist. Die dritte Gruppe, welche aus Dokumentation, Tools und Erfahrungen besteht, ist ebenfalls indirekt, da durch sie ebenfalls der Änderungsaufwand entweder erhöht oder verringert werden kann. Im Unterschied zur Komplexität sind sie aber außerhalb des zu portierenden Software-Systems angesiedelt und werden als externe Faktoren bezeichnet. Zur eigentlichen Dokumentation der Quellen eines Software-Systems zählt auch die Portierungsdokumentation. Hierbei sollte in erster Linie eine detaillierte Beschreibung der erforderlichen Portierungsschritte und die Bereitstellung von Testdaten, zur Verifizierung des Portierungsprozesses, im Vordergrund stehen.

Mit Hilfe von Portierungstools wird der Portierungsprozeß teilweise automatisiert. Dazu gehören ganz allgemeine Tools wie Texteditoren, aber auch Spezielle, wie Präprozessoren oder Filter. Einen gewichtigen Faktor für den Portierungsaufwand stellen die Erfahrungen und Kenntnisse eines Porteurs dar. Diese können in allgemeiner Form, als Programmiererfahrung in früheren Portierungsprojekten vorliegen. Es kann sich aber auch um spezielle Kenntnisse, bezüglich der verwendeten Programmiersprache oder des Betriebssystems, mit der zugrundeliegenden Hardware, handeln. Der Aufwand für die Einarbeitung in ein konkretes Portierungsprojekt und dessen Durchführung wird also maßgeblich durch den Porteur und dessen Erfahrungen bestimmt. Diese sind um so wichtiger, je unvollständiger oder ungeeigneter die Portierungsdokumentation ist.

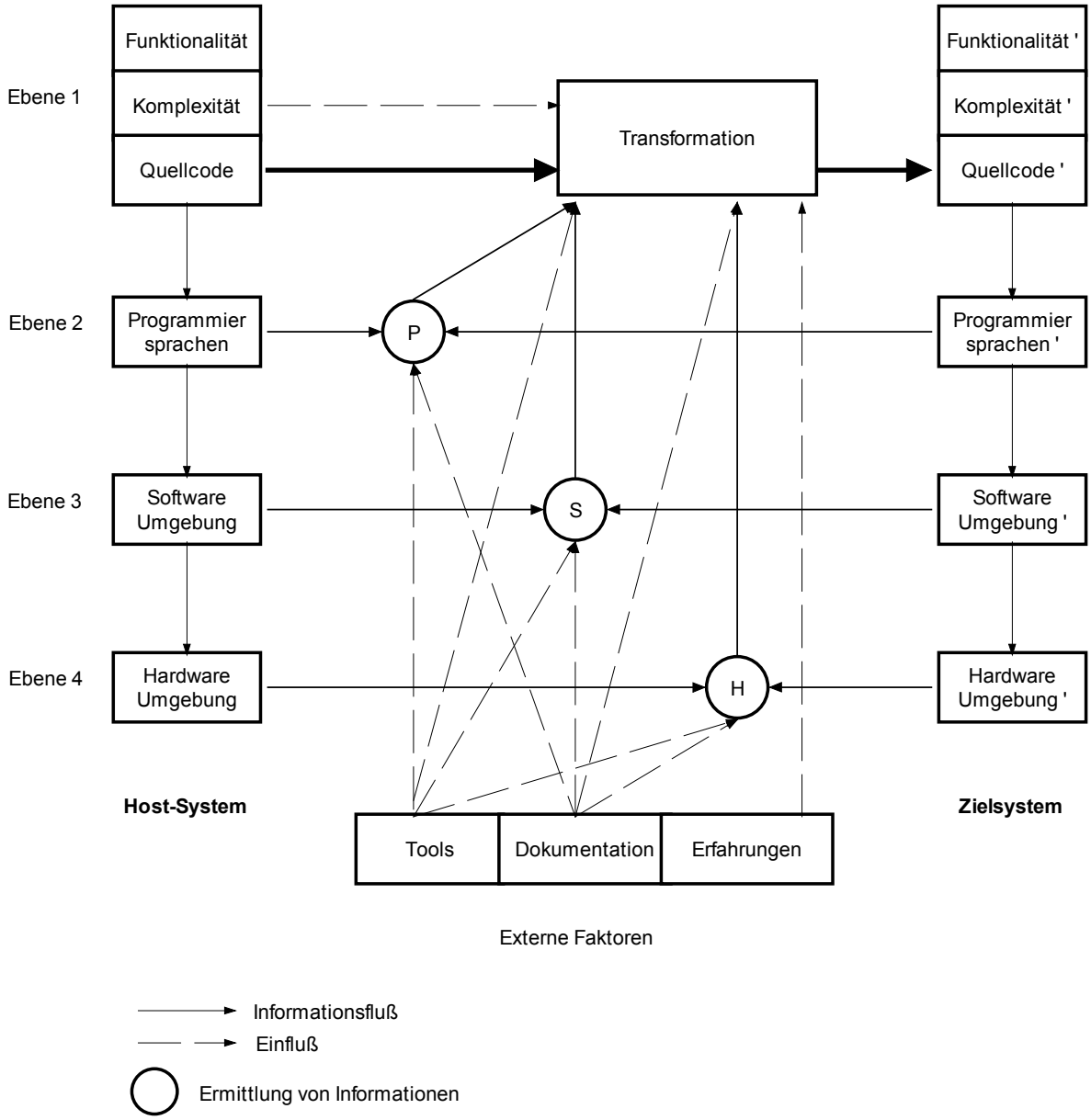


Abbildung 4: Portierungsmodell nach Buschhorn

Das Modell besteht zunächst aus den Komponenten Host- und Zielsystem, die zusammen mit der Software, bzw. der portierten Software, eine Hierarchie aus vier abstrakten Maschinen bilden. Auf der ersten Ebene finden sich die abstrakten Maschinen, die das zu portierende Programm darstellen. Die für eine Portierung relevanten Merkmale werden durch Quellcode, Funktionalität und Komplexität charakterisiert. Die programmimanente Komplexität eines Programms kann sich durch eine Portierung verändern, der Quellcode verändert sich in jedem Fall. Finden keine Änderungen am Quelltext statt, so wird hier nicht von einer Portierung gesprochen. Um von einer Portierung eines Software-Systems zu sprechen, müssen folgende Bedingungen erfüllt sein:

$$Komplexität = Komplexität' \wedge Komplexität \neq Komplexität' \text{ und } Quellcode \neq Quellcode'.$$

Auf der zweiten Ebene befinden sich die Programmiersprachenkomponenten von Host- und Zielsystem. Sie repräsentieren die Schnittstellen zu den darüberliegenden Programmen und werden durch die Konstrukte realisiert, die die jeweilige Programmiersprache zur Verfügung stellt. Die dritte

Ebene der Softwareumgebung stellt sowohl die Funktionen des jeweiligen Betriebssystems, als auch die anderer Komponenten dar, z.B. zusätzliche Bibliotheken. Auf der letzten Ebene befinden sich die abstrakten Maschinen der Hardwareumgebung. Die Schnittstelle zur darüberliegenden Softwareumgebung wird durch die Menge der Maschinenbefehle realisiert. Außerhalb der Hierarchien von Host- und Zielsystem befinden sich die externen Einflußfaktoren. Einflüsse werden im Modell mit gestrichelten Pfeilen dargestellt, Informationsflüsse mittels durchgezogener Pfeile. Der fettgedruckte Pfeil repräsentiert die eigentliche Übertragung des zu portierenden Programms. Die Pfeile zwischen den Ebenen der abstrakten Maschinen auf Host- und Zielumgebungsseite, kennzeichnen den im obigen Sinn angedeuteten, hierarchischen Aufbau der betroffenen Komponenten. Mit den Kreisen wird ausgedrückt, daß an diesen Stellen Informationen, die für die Portierung erforderlich sind, ermittelt werden. So werden z.B. bei der Informationsermittlung P, die Unterschiede zwischen Host- und Zielumgebung ermittelt. Dazu können Werkzeuge, aber auch Informationen aus der Dokumentation des Software-Systems hilfreich sein. Die so gewonnenen Information fließen dann in der Transformation zusammen. Hier werden die erforderlichen Änderungen am Quellcode des Software-Systems sowie das Testen des angepaßten Software-Systems durchgeführt. Diese Schritte wiederholen sich zyklisch, bis das Software-System, unter Einhaltung der gestellten Anforderungen, auf dem Zielsystem korrekt läuft.

2.2 Portabilität Windows-basierter Software-Systeme

Dieses Kapitel beschreibt die Probleme der Portierung von Software-Systemen, die auf der Basis einer 16-Bit-Windows-Umgebung entstanden sind und in eine 32-Bit-Windows-Umgebung portiert werden sollen. Hierbei wird zunächst auf die historische Entwicklung der einzelnen Windows-Systeme eingegangen. Danach werden die wichtigsten Windows-Konzepte und Strategien für die Windows-Programmierung beleuchtet und somit insgesamt der Rahmen für die Betrachtung der verschiedenen Portierungsprobleme abgesteckt. Anschließend werden, ausgehend von der Programmiersprache C und C++ und den Architekturen der betroffenen Windows-Systeme, die spezifischen Portierungsprobleme behandelt und in einzelnen Problemklassen zusammengefaßt. Ausführungen über Strategien und unterstützende Werkzeuge – u.a. das Werkzeug ’Porttool’ der Firma Microsoft – bei der Portierung, schließen sich dann an diesen Teil an.

2.2.1 Historie von Windows-Systemen

Im Herbst 1981 wurde der erste IBM-PC, mit zwei Betriebssystemen, von der Firma IBM auf den Markt gebracht. Hierbei konnte sich das Betriebssystem für PC’s und Kompatible, MS-DOS, schließlich am Markt behaupten. Diese Variante von MS-DOS stellte dem Benutzer eine Eingabeaufforderung zur Verfügung, über die er entsprechende Kommandos an das Computersystem übermitteln durfte und verschiedene Anwendungsprogramme in den Hauptspeicher laden konnte. Das ganze Betriebssystem bestand für Programmierer demnach aus einer Gruppe von Funktionen, die den Datentransfer von und zu einzelnen, externen Speichermedien übernahm und den Rest über direkte Zugriffe auf die Hardware realisierte (z.B. Textausgabe oder auch schon Grafiken).

Im Januar 1983 wurde von der Firma Apple der erste, mißlungene Versuch unternommen, textbasierten ‘Diskettenbetriebssystemen’ eine grafisch orientierte Programmierumgebung gegenüberzustellen. Im Jahre 1994 konnte aufgrund der jetzt zur Verfügung stehenden,

leistungsfähigeren Hardware, der Macintosh mit einem grafisch orientierten Betriebssystem zur Marktreife gebracht werden. Dieses Betriebssystem setzte damit den Standard für grafisch orientierte Benutzeroberflächen.

Windows erschien, mit der Versionsnummer 1.01, im November 1985 – also noch vor dem Macintosh– auf dem Markt für IBM-PC's und wurde durch einige Modifikationen in der Benutzeroberfläche durch die Version 2.0.l im Jahre 1987 abgelöst. Beide Versionen kamen mit den Prozessormodellen 8086 und 8088 von Intel zurecht, die im Real-Mode liefen und maximal 1 MB Speicher adressieren konnten. Kurz nach dem Erscheinen von Version 2.0 kam ein Windows/386, im Jahre 1988, für den Prozessor 80386 auf den Markt, das den Virtual-8086-Mode unterstützte. Dadurch war es zum erstenmal möglich, DOS-Programme durch Simulation mehrerer, voneinander unabhängiger Computer parallel ablaufen zu lassen, wobei diese auch direkt auf die zugrundeliegende Hardware zugreifen konnten.

Die Version 3.0 von Windows wurde im Mai 1990 vorgestellt. Sie vereinigte die 386er- und 286er-Version – Bezeichnung der Nachfolgeversion von Windows 2.0, die konsequenterweise als Windows/286 eingeführt wurde – und unterstützte den Protected-Mode für Modelle ab dem 80286. Diese Version fand im Heim- und Bürobereich eine weite Verbreitung. Mit Hilfe des Protected-Mode konnten Anwendungsprogramme einen Hauptspeicher von bis zu 16 MB adressieren und somit die 640-KB-Grenze von MS-DOS überschreiten.

Windows 3.1 erschien im April 1992 und hatte wesentliche Neuerungen aufzuweisen. Dazu gehörten die TrueType-Technologie – skalierbare Outline-Schriften, auch für den Bildschirm –, ein Grundgerüst für Mutimedia-Anwendungen, das Einbetten und Verknüpfen von Objekten (OLE) sowie die Definition von Dialogen für Standard-Prozeduren, wie das Laden und Speichern von Dateien. Windows 3.1 läuft im Vergleich zu seinen Vorgängern jedoch nur noch im Protected-Mode und setzt einen 80286-Prozessor sowie Hauptspeicher von mindestens 1 MB voraus. Eine Unterstützung für Peer-to-Peer-Netzwerke wurde mit der Version Windows für Workgroups 3.1, ab November 1992 realisiert (WfW).

Windows NT (New Technology), in der Version 3.1, wurde im April 1994 vorgestellt und ist speziell auf den Protected-Mode mit 32-Bit-Prozessoren – dazu gehören die Modelle 80386, 80486, Pentium und alle Nachfolger – zugeschnitten. Es war von Anfang an als Netzwerkbetriebssystem konzipiert, weshalb es in zwei Varianten erhältlich war: als Workstation- oder Client/Server-Variante. Anwendungsprogramme, die für dieses Betriebssystem geschrieben werden, haben Zugriff auf einen linearen Adreßraum mit 32 Bit und verwenden ausschließlich 32-Bit-Befehle. Windows NT ist für den professionellen Einsatz auf Workstations, bis hin zu Client/Server-Umgebungen gedacht und bietet daher, im Vergleich zu seinen Vorgängern, zusätzliche Sicherheitsmechanismen und hohe Stabilität. Windows NT wurde als portables Betriebssystem entwickelt und läuft sowohl auf Intel-Plattformen, als auch auf einer Reihe von RISC-Workstations, die den Alpha-Prozessor der Firma DEC verwenden. Es wurde zunächst in den Versionen 3.1 bis 3.5 ausgeliefert und hatte noch die Oberfläche von Windows 3.1. Mit Version 4.0, die im Jahr 1995 auf den Markt kam, wurde die Oberfläche an Windows 95 angeglichen. Als Nachfolger von Windows NT 4.0 ist Windows 2000 seit Februar 2000 auf dem Markt. Es zeichnet sich vor allem durch eine, im Vergleich zu seinem Vorgänger, verbesserte Technik aus und bietet darüber hinaus eine wesentlich verbesserte Bedienoberfläche.

Windows 95 erschien im August 1995 und unterstützt, wie Windows NT, das 32-Bit-Programmiermodell des 80386 und seiner Nachfolger. Im Vergleich zu Windows NT fehlen allerdings Vorteile wie: Sicherheit, Stabilität oder Portabilität zu anderen Hardware-Plattformen. Dafür sind die Hardwareanforderungen jedoch wesentlich geringer, als bei Windows NT. Der Nachfolger, Windows 98, wurde im Juni 1998 vorgestellt und zeichnet sich, im Vergleich zu Windows 95, durch höhere Performance, ein größeres Spektrum unterstützter Hardware sowie eine bessere Anbindung an das Internet aus. Der nächste Nachfolger, im Bereich der sogenannten Heimanwender-Betriebssysteme, ist Windows ME und wurde im Juli 2000 auf dem Markt eingeführt. Es liefert eine Reihe von neuen Funktionen, u.a. für die Bereiche Multimedia, Internet und kleine Heimnetzwerke. Allen Versionen dieser Windows-Reihe ist gemeinsam, daß sie als technische Basis noch immer das Betriebssystem MS-DOS nutzen.

Mit der aktuellen Windows-Version, Windows XP, hat Microsoft auch das 64-Bit-Zeitalter eingeleitet. Diese Windows-Version wird sowohl für 32-Bit-x86-Systeme als auch für Intels zukünftige 64-Bit-Prozessoren ‘Itanium’ zur Verfügung stehen. Hierbei wurde die technische Basis von Windows 9x, ME und Windows 2000 vereinigt, um so eine einheitliche Basis für alle Windows-Versionen zu schaffen. Eine Übersicht zur Historie von Windows zeigt Abbildung 5.

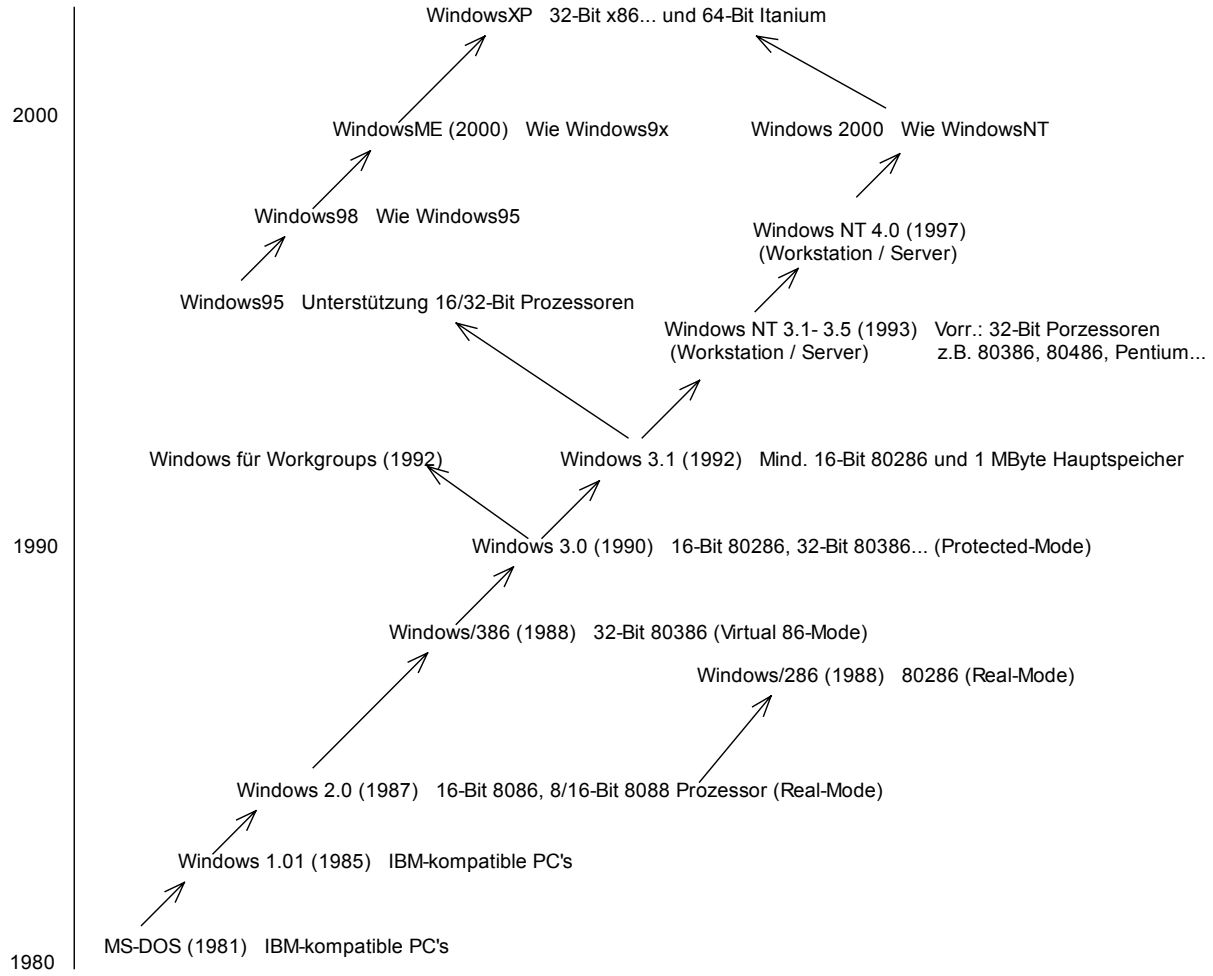


Abbildung 5: Übersicht zur Historie von Windows

2.2.2 Windows-Konzepte und Strategien zur Windows-Programmierung

Die Windows-Betriebssystem definiert sich durch eine Vielzahl unterschiedlicher Konzepte, die bei der Programmierung zu berücksichtigen sind. In diesem Abschnitt sollen die wichtigsten dieser Konzepte kurz vorgestellt werden und es soll gezeigt werden, welche Möglichkeiten bestehen, Windows-basierte Software-Systeme zu entwickeln

2.2.2.1 Konzepte von Windows-Systemen

API (Application Programming Interface)

Aus Programmierersicht definiert sich ein Betriebssystem über dessen Programmierschnittstelle. Im Falle von Windows ist dies die Windows-API, die mittlerweile mehrere tausend Funktionen umfaßt und sämtliche, bei diesen Aufrufen verwendbaren, Strukturen und Datentypen festlegt. Für die Oberflächen, ab Windows 95 bzw. Windows NT (ab Version 3.5), wird diese Programmierschnittstelle zusammenfassend als Win32-API bezeichnet. Im Gegensatz dazu werden die 16-Bit-Versionen bis Windows 3.1 und Windows für Workgroups als Win16-API bezeichnet.

Der Grund für diese Unterteilung soll hier kurz aufgeführt werden, näheres dazu siehe Abschnitt 2.2.3. Die Versionen Windows 1.0, bis einschließlich 3.1, mußten mit dem Adressierungsmodell der Prozessoren 8086, 8088 bis 80286 – von den nachfolgenden Prozessoren aus Kompatibilitätsgründen weiterhin unterstützt – auskommen, das eine Segmentunterteilung des Speichers in jeweils 64 KB vornimmt. Das Modell legt die Registerbreite des Prozessors auf 16 Bit fest, d.h. ein ganzzahliger C-Datentyp belegt 16 Bit Adreßspeicher und setzt seine Speicheradresse aus zwei 16-Bit-Komponenten zusammen (einem Segment und einem Offset, innerhalb dieses Segments). Die 32-Bit-Versionen von Windows, wie Windows NT oder Windows 9x sowie deren Nachfolger, verwenden den 32-Bit-Modus der Prozessormodelle 80386, 80486, Pentium etc. Der C-Datentyp ’int’ umfaßt hierbei 32 Bit und wird in einem linearen Adreßraum angeordnet, ohne Segment und Offset.

Alle Funktionen der Windows-API lassen sich größtenteils in folgende Kategorien einteilen: Systemkern, Oberfläche und Grafik. Der als Kernel bezeichnete Systemkern enthält Funktionen für die Speicherverwaltung, Datei-Management und die Verwaltung von Prozessen. Die im sogenannten User-Modul zusammengefaßten Funktionen, sind für die Benutzeroberfläche und die Fensterlogik zuständig. Das Modul GDI stellt die grafische Geräteschnittstelle dar und seine Funktionen übernehmen die Darstellung von Text und Grafik, sowohl auf dem Bildschirm als auch auf dem Drucker. Alle anderen Funktionen können den anderen, zur Verfügung stehenden, Subsystemen (z.B. POSIX, Console-API etc.) zugeordnet werden.

GUI (Graphical User Interface)

Alle heutzutage üblichen, grafisch orientierten Benutzeroberflächen – inklusive des Macintosh und Windows – beruhen auf Vorarbeiten des Palo Alto Research Centers (PARC) der Firma Xerox. Das dahinter stehende Konzept war bereits Mitte der 70er Jahre dort entwickelt worden und wurde zunächst in einer, auf Smalltalk basierenden, Umgebung realisiert. 10 Jahre später wurde es von Firmen wie Microsoft und Apple vermarktet. Die zentrale Idee von GUI’s ist die symbolische oder grafische Repräsentation von Informationen und deren Manipulation. Neben der so entstehenden Interaktion zwischen Mensch und Maschine liegt der größte Vorteil von grafischen

Benutzeroberflächen darin, daß sie Programme überwiegend einheitlich repräsentieren. Durch das Einhalten des Prinzips der Uniformität bei der Repräsentation und Manipulation von Informationen, steigt die Lernkurve des Benutzers exponentiell an und erleichtert somit auch das Einarbeiten in ähnliche Umgebungen mit grafischer Benutzeroberfläche. Aus Programmiersicht zeigt sich die Uniformität darin, daß das System nicht nur die Funktionen zum Anlegen von Fenstern, Menüs, Dialogen etc. definiert, sondern auch die Teile der Funktionalität dieser Komponenten. So haben z.B. Menüs deshalb eine einheitliche Maus- und Tastaturschnittstelle, weil diese Arbeit von Windows übernommen wird und nicht Aufgabe der jeweiligen Anwendung ist.

Multitasking

Hinter diesem Konzept steht die Idee der quasi-parallelen Ausführung von Software-Systemen, die bei Bedarf untereinander Informationsaustausch betreiben können und durch jeweils ein Hauptanwendungsfenster auf der Benutzeroberfläche repräsentiert werden. Benutzer können diese Anwendungsfenster dann, z.B. in ihrer Größe verändern, verschieben oder beliebig von einem Anwendungsfenster in ein anderes wechseln und Informationen zwischen diesen austauschen. Man unterscheidet zwischen kooperativem und verdrängendem Multitasking. Windows-Versionen der 16-Bit-Generationen folgen zwar den Prinzipien der quasi-parallelen Ausführung von Programmen, arbeiten jedoch mit der Technik des kooperativem Multitasking. Hierbei wird jeder Anwendung durch den Benutzer die vollständige Kontrolle über das Computersystem gegeben, bis diese an eine andere Anwendung übergeben wird usw. Hierbei werden der übergebenden Anwendung solange jegliche Ressourcen entzogen, bis diese wieder die Kontrolle erhält. Anders sieht es bei der Technik des verdrängendem (preemptiven) Multitasking, in 32-Bit-Versionen von Windows, aus. Hier werden den einzelnen Prozessen, vom System selbst, bestimmte Rechenzeiten zugewiesen und eine entsprechende Kontrolle über bestimmte Ressourcen gegeben. Anwendungen können sich hierbei in mehrere Ausführungsstränge (Threads) unterteilen und werden dann von Windows in Einprozessorsystemen quasi-parallel, bzw. in Mehrprozessorsystemen, echt parallel, ausgeführt. Dies gilt jedoch nicht für Windows 9x bzw. Windows ME, da diese Betriebssysteme keine Mehrprozessorunterstützung aufweisen.

Dynamische Speicherverwaltung

Die Fragmentierung von Speicherplatz stellt bei allen Betriebssystemen ein Problem dar. So ist z.B. der Hauptspeicher eines Computersystems nach kurzer Zeit so stark fragmentiert, daß ohne eine dynamische Reorganisation, welche die unregelmäßige Folge freier und belegter Speicherplätze bereinigt, nur einzelne Teile des Speichers allokiert werden könnten und nicht als gemeinsamer Speicherplatz nutzbar wären (absolute Speicheradressen, statisch verwaltet). Die Idee der dynamischen Speicherverwaltung löst das Problem in Form von Indirektion. Jeder Speicheradresse wird eine eindeutige Kennziffer zugewiesen, über eine Tabelle miteinander verknüpft und entsprechend verwaltet. Somit kann das Betriebssystem die realen Speicherbereiche über die Kennziffern beliebig umverteilen und nicht benötigte Bereiche auf die Festplatte auslagern, d.h. virtuellen Speicher simulieren. Die Idee der dynamischen Speicherverwaltung wurde seit der ersten Windows-Version konsequent umgesetzt. Später durch die Unterstützung von Speichererweiterungskarten (Expanded Memory Support) in der Version Windows 2.0 erweitert und in der Version 3.0 auf die Verwaltung von bis zu 16 MB Hauptspeicher ausgebaut. Voraussetzung für die dynamische Speicherverwaltung bis zur Windows-Version 3.1 waren, aufgrund der Prozessorarchitekturen, eine Vielzahl von Hilfskonstruktionen, die den Speicher in einzelne Segmente

unterteilen mußten und die Speicherplätze über sogenannte Offsets in einem Segment eindeutig adressierten. In den 32-Bit-Versionen von Windows wird die notwendige Indirektion nicht mehr über Hilfskonstruktionen, wie Segmentarithmetik, abgehandelt, sondern direkt vom Prozessor vorgenommen. Die Adressierung geschieht aus der Sicht von Programmen über die eindeutige Angabe einer 32 Bit breiten Speicheradresse. Diese sind somit nicht mehr Offsets in einen begrenzten, segmentierten Bereich, sondern in den bis zu 4GB großen, linearen Adreßraum.

Dynamische Bibliotheken

Dynamische Link-Bibliotheken (Dll's) stellen von Beginn an ein zentrales Konzept für die gemeinsame Verwendung von Codeteilen dar. Ausgangspunkt für die Überlegungen waren z.B. Probleme bei gemeinsamen Betriebssystemaufrufen unter MS-DOS. Nach dem Compilieren und Binden haben selbstdefinierte Funktionen oder Funktionen statischer Bibliotheken, eine jeweils festgelegte Speicheradresse. Bei dynamischen Aufrufen dieser Funktionen in Systembibliotheken, zur Laufzeit, werden die Funktionsnummern in ein Register geladen. Funktionen, die auf diese Systembibliotheken zugreifen, brauchen somit die exakte Speicheradresse der aufzurufenden Funktion, und diese muß darüber hinaus für alle Clients konstant gehalten werden, Da Windows nun aber mehrere tausend Funktionen zur Verfügung stellt und diese nach außen hin, statt Funktionsnummern, selbstbeschreibende Funktionsnamen besitzen, war hier ein neuer Mechanismus notwendig, der als dynamisches Binden bezeichnet wird. Somit erscheinen Funktionen des Betriebssystems in Windows-Programmen genauso wie Aufrufe normaler Bibliotheksfunktionen. Benutzt man nun eine Betriebssystemfunktion in Windows-Programmen, so läuft das Einbinden über eine sogenannte Importbibliothek, die nicht den spezifischen Code dieser Funktion enthält, sondern nur einen Verweis auf eine dynamische Bibliothek, die Windows bereitstellt. Wird das Programm gestartet, sucht das Betriebssystem die benötigten Bibliotheken heraus, lädt sie in den Speicher und setzt die tatsächlichen Adressen der einzelnen Funktionen in das Programm ein. Sämtliche Funktionen der Windows-API befinden sich in solchen dynamischen Link-Bibliotheken. Die drei wichtigsten Dll's sind: die Systemkernbibliothek 'Kernel32.dll' (für die 16-Bit-Windows-Systeme, das Modul Kernel386.exe), die Bibliothek für die Verwaltung der Benutzeroberfläche 'User32.dll' (User.exe für 16 Bit) sowie die Grafik- und Textausgabebibliothek 'GDI32.dll' (GDI.exe bei 16 Bit). Darüber hinaus enthält Windows weitere Dll's mit Funktionen für speziellere Aufgaben.

Das Konzept der dynamische Bibliotheken ist aber nicht nur bei der Nutzung von Betriebssystemfunktionen nützlich, sondern eröffnet auch eine Vielzahl von weiteren Anwendungsmöglichkeiten. Die verschiedenen Anwendungsfelder ergeben sich aus den unterschiedlichen Gründen, Dll's in seinen Software-Systemen einzusetzen. Ein wichtiger Grund ist die anwendungserweiternde Eigenschaft von dynamischen Bibliotheken. So könnte beispielsweise ein Unternehmen ein bestimmtes Software-Produkt erstellen und anderen Unternehmen erlauben, dieses in Form von Dll's zu erweitern oder zu verbessern. Sie vereinfachen darüber hinaus auch das Projektmanagement und aus Portabilitätssicht ergeben sich Gründe für die Verwendung von Dll's, aus der Sprachunabhängigkeit und der teilweisen Überwindung von Plattformunterschieden. Man kann z.B. diejenige Sprache auswählen, die für eine bestimmte Aufgabe am besten geeignet erscheint. Das Betriebssystem erlaubt dann beispielsweise einem Visual-Basic-Programm, eine C++-Dll, eine Cobol-Dll oder eine Fortran-Dll etc. zu laden. Aber auch einzelne Plattformunterschiede von Software-Systemen zu einzelnen Windows-Versionen, können mit Hilfe von Dll's zum Teil überwunden werden. So könnte man neue Mechanismen aktueller Windows-Versionen durch spezielle Funktionen bereitstellen, die vom Entwickler in einer aktuellen Version des Software-Systems nutzbar gemacht

werden sollen. Faßt man solche Funktionen in einer Dll zusammen, können Anwendungen unter einer älteren Windows-Version trotzdem ausgeführt werden, auch wenn für die neueren Funktionen keine entsprechende Unterstützung existiert. Würde man in einem solchen Fall auf die Nutzung von Dll's verzichten, könnte der Lader des Betriebssystems die Anwendung nicht ausführen, da der Quellcode einen Aufruf einer Funktion enthält, welche in der Windows-Version, in der die Anwendung ausgeführt wird, nicht unterstützt wird. Außerdem tragen Dll's zur sparsamen Verwendung von Speicher bei und ermöglichen die gemeinsame Verwendung von Ressourcen.

Treiberbibliothek

Der IBM-PC war von Anfang an als ein offenes System konzipiert, d.h. es ließen sich beliebige PC-kompatible Hardwarekomponenten, verschiedenster Hersteller, in das Computersystem einbinden. Der größte Nachteil hierbei war, das jedes Software-System eine eigene Treiberbibliothek mitbringen mußte, welche die eingesetzte Hardware in allen Details der Ansteuerung zu kennen und zu unterstützen hatte. Nicht selten hatten Software-Systeme, die nach diesem Konzept die Hardware verwalteten, 80% ihres Installationsumfangs mit Treibern, z.B. für eine Vielzahl verschiedener Grafikkarten oder Hunderten von Druckern, belegt. Das Konzept der Windows-Treiber und einer zentralen Treiberbibliothek führte ab der ersten Windows-Version dazu, daß das Betriebssystem die Verwaltung der Hardware übernahm. Software-Systeme greifen hierbei nicht direkt auf die Hardware zu, sondern benutzen einheitliche Treiberschnittstellen des Systems. Damit ist die Verantwortung für die Bereitstellung der Treiber von den Software-Entwicklern auf die Hardware-Entwickler übergegangen. Diese stehen nun ihrerseits in der Pflicht, die angebotene Hardware mit entsprechenden Treibern für die einzelnen Windows-Versionen auszuliefern, auf die dann alle Software-Systeme zentral zurückgreifen, um die jeweilige Hardware anzusprechen.

2.2.2.2 Programmierung Windows-basierter Software-Systeme

Für die Programmierung von Software-Systemen unter Windows benutzt man die jeweilige Programmierschnittstelle, hier die Windows-API, über die sich aus Programmierersicht ein Betriebssystem definiert. Mit Hilfe von integrierten Entwicklungsumgebungen, als automatisches Werkzeug, nicht nur für den Erstellungsprozeß, sondern auch für den gesamten Entwicklungsprozeß, werden komplexe Software-Systeme für Windows realisiert. Abstrakte Entwicklungsumgebungen und Programmierbibliotheken verbergen jedoch meist die internen Abläufe eines Systems und verhindern somit ein tieferes Verständnis der zugrunde liegenden Strukturen und Datentypen. Da Windows selbst überwiegend in der Programmiersprache C geschrieben ist, liefert diese, in Kombination mit direkten Aufrufen der Windows-API, natürlicherweise die bestmögliche Performance und Flexibilität. Darüber hinaus schult der direkte Umgang mit der Windows-API auch das Wissen über die internen Abläufe und macht sich bei einem Wechsel auf eine abstraktere Entwicklungsumgebung, durch ein solide geschultes API-Wissen bezahlt. Demgegenüber steht die einfache und zeitsparende Programmerstellung mittels einer Programmierumgebung und -bibliothek. Dieser erlauben die Konzentration auf das Wesentliche, z.B. die Elemente der Benutzeroberfläche, die dann über relativ einfache Mechanismen mit entsprechenden Aktionen verbunden werden. Somit kommt man relativ schnell zu komplexen Software-Systemen unter Windows, ohne die Komplexität des Systems zu berühren.

Für die Programmierung mittels abstrakter Entwicklungsumgebungen und entsprechenden Bibliotheken bieten sich eine Vielzahl kommerzieller Lösungen an. Hier wäre als erste Microsoft's Visual C++ in Kombination mit den Microsoft Foundation Classes (MFC) – als Alternative zum reinen C und der Windows-API – zu nennen. Diese Entwicklungsumgebung besteht aus einem C- und C++-Compiler, zusammen mit Bibliotheken und Werkzeugen, die man zum Übersetzen und Binden von Programmen braucht. Außerdem bietet sie eine integrierte Oberfläche für die Erstellung und Bearbeitung der Quellen und benötigten Ressourcen. Ebenfalls enthalten ist ein interaktiver Debugger für die Testphase des Entwicklungsprozesses. Die MFC-Bibliothek abstrahiert von den komplexen Teilen des Windows-Systems und kapselt viele Aspekte der Windows-Programmierung in einer Sammlung von C++-Klassen. Legt man weitere Programmiersprachen zugrunde, so lassen sich mit Hilfe von Visual-Basic-Entwicklungsumgebungen (z.B. Microsoft Visual Basic) ebenfalls komplexe Software-Systeme entwickeln. Das gleiche gilt für die von der Firma Sun Microsystems, in Anlehnung an C++ erfundene, prozessorunabhängige Programmiersprache Java (Microsoft Visual Java++).

Weitere Entwicklungsumgebungen sind Borland C++ oder der C++-Builder, der Firma Borland. Auch hier werden eine integrierte Entwicklungsumgebung (IDE) und alle Werkzeuge für die Erstellung und Entwicklung Windows-basierter Software-Systeme bereitgestellt. Die von der Windows-API abstrahierende Programmierbibliothek ist hierbei die Objekt Windows Library (OWL), bzw. bei C++-Builder, die Visual Component Library (VCL). Legt man eine andere Programmiersprache zugrunde, so bieten sich weitere Lösungen. Die gleichfalls von der Firma Borland angebotene Delphi-Entwicklungsumgebung – eine objektorientierte Pascal-Variante – ermöglicht es ebenfalls, Oberflächen-Design mit Codeerstellung effizient zu verbinden. Sie bietet gleichfalls eine umfangreiche Klassenbibliothek und die Möglichkeit, diese mit direkten Aufrufen der Windows-API zu kombinieren.

Der Einsatz von Entwicklungsumgebungen ist, wie bereits kurz angesprochen, mit Vor- und Nachteilen verbunden. So ändert eine zusätzliche Schicht, oberhalb der Windows-API, nichts an der Komplexität des Systems, sondern versteckt sie lediglich und ist zwangsläufig mit Einschränkungen verbunden, wenn sie nicht den gleichen Komplexitätsgrad erreicht, wie die Windows-API selbst. So setzen zusätzlich gewünschte Mechanismen, die die eingesetzte Bibliothek überfordern, eine solide Kenntnis der zugrunde liegenden Programmierschnittstelle voraus und verlangen einen direkten Gebrauch der Windows-API. Demgegenüber stehen alle Vorteile der objektorientierten Programmierung beim Einsatz von objektorientierten Programmierbibliotheken, wie beispielsweise der MFC. Software-Systeme können in kürzerer Zeit entwickelt werden, als dies mit der reinen API möglich wäre. Durch das Zurückgreifen auf ausgetesteten Code, sinkt die Fehlerquote in den damit entwickelten Software-Systemen. Die Kosten für Pflege, Wartung und Weiterentwicklung werden erheblich reduziert und sind damit ein entscheidender, wirtschaftlicher Faktor. Aber auch hier steigt der Komplexitätsgrad, durch den immer größer werdenden Funktionsumfang der Bibliotheken, stetig an. Damit wächst auch der Einarbeitungsaufwand für den Entwickler.

Betrachtet man den Aspekt der Portabilität von Windows-basierten Software-Systemen, so sind viele der Bibliotheken auch auf anderen Plattformen erhältlich und bieten somit eine Art von bibliotheksbedingter Plattformunabhängigkeit, die bei klassischer Programmierung, über die Windows-API, nicht gegeben ist. Man kann jedoch zusammenfassend feststellen, daß die gemeinsame Basis aller Windows-basierten Software-Systeme, sich in der, dem Betriebssystem zugrunde liegenden, Programmierschnittstelle widerspiegelt und der Weg zur besten Performance in der Regel über einen C-Compiler führt. Somit soll die Windows-API, als unterste Abstraktionsebene des

Betriebssystems, zusammen mit C bzw. C++ als Programmiersprache, die gemeinsame Grundlage für weitere Betrachtungen bilden.

2.2.3 Portierungsprobleme Windows-basierter Software-Systeme

In diesem Kapitel geht es um die Klassifikation von Portierungsproblemen, die auftreten können, wenn Software-Systeme, aus einer 16-Bit-Windows-Umgebung (Win16) heraus, in eine 32-Bit-Windows-Umgebung (Win32) portiert werden sollen. Der Ausgangspunkt für die in diesem Kapitel angesprochenen Probleme sind 16-Bit Software-Systeme, die in C oder C++ geschrieben sind und direkt auf der Win16-API aufsetzen. Der Vorteil von C bzw. C++ zu anderen Programmiersprachen, wie z.B. Java – die von einer Standardisierung noch weit entfernt ist – oder Pascal – es existiert zwar ein ISO-Standard für Pascal, dieser ist aber nicht mächtig genug, sodaß ihn jeder Hersteller mit zahlreichen Erweiterungen versieht – liegt zum einen darin, daß diese durch anerkannte und präzise Sprachstandards sowie standardisierte Laufzeitbibliotheken definiert werden, zum anderen aber hohe Optimierungsfähigkeiten und Verfügbarkeit für Windows-Systeme aufweisen. Unter Benutzung der Standardbibliotheken und des ANSI-Sprachstandards, sowie konsequenter Beachtung bestimmter Portabilitätsrichtlinien geschriebene Software-Systeme unter Windows, sind mit vergleichsweise geringem Aufwand auf andere Plattformen zu portieren, sofern dort ein entsprechendes ANSI-C- bzw. ANSI-C++-Entwicklungssystem zur Verfügung steht. So werden zunächst grundlegende Richtlinien für portable C bzw. C++-Programme vorgestellt. Anschließend werden die Architekturunterschiede der zu betrachtenden Host- (Win16) und Zielumgebungen (Win32) behandelt, um anschließend auf die Portierungsprobleme in diesem Portierungsfeld einzugehen und diese zu klassifizieren.

2.2.3.1 Portable C und C++-Software-Systeme

Blickt man zunächst auf die Sprachebene von Windows-Programmen, unabhängig von spezifischen Betriebssystemen oder Entwicklungswerkzeugen, auf der Basis der Programmiersprachen C bzw. C++, so lassen sich allgemeine Richtlinien erkennen, die konsequent angewendet, zu portableren Software-Systemen führen. Portabilität von Software-Systemen hängt nicht nur von der Uniformität der Programmierschnittstellen ab, sondern ist auch ein kennzeichnendes Merkmal von Quelltexten, unabhängig vom Zielsystem. Der Aufwand für die Portierung eines Windows-basierten Software-Systems hängt einerseits von der möglichst rückwärtskompatiblen Gestaltung der zur Verfügung stehenden Windows-API ab. Andererseits müssen die zu portierenden Quellen bestimmten, allgemeinen Portabilitätsgrundsätzen genügen, damit die Portierung nicht zu aufwendig wird.

Sprachstandards

Der erste Schritt zu portablen Software-Systemen führt über die Benutzung einer standardisierten Hochsprache. Ein solcher Sprachstandard wird dann entwickelt, wenn eine Reihe von widersprüchlichen Implementierungen existieren, die es zu vereinigen gilt. So gibt es mehrere Lieferanten oder Versionen von Compilern für bestimmte Hochsprachen und Betriebssysteme. Dies führt dazu, daß einige Compiler die Quellen nicht einheitlich übersetzen. Die Einhaltung eines Sprachstandards stellt aber noch keine Garantie für die Portabilität von Software-Systemen, dar. Viele Standards sind unvollständig und definieren verschiedene Merkmale einer Sprache bewußt unvollständig. Somit suggerieren Sprachstandards den Eindruck einer genauen Spezifikation. Sie

definieren eine Sprache aber niemals vollständig, sodaß unterschiedliche Implementierungen gültige, aber dennoch inkompatible Interpretationen vornehmen können. Dennoch ist die Einhaltung von Sprachstandards, wie der offizielle ANSI/ISO-Standard für C bzw. der ISO-Standard für C++, eine notwendige Bedingung für portable Programmierung.

Den unterschiedlichen Spracheigenschaften und verschiedenen Implementationen eines Sprachstandards in bestimmten Compilern, begegnet man größtenteils mit der Programmierung im sogenannten 'Mainstream'. Hierbei benutzt man nur die Eigenschaften eines Sprachstandards, für welche die Sprachdefinition eindeutig ist und die Implementierung durch den Compiler feststeht. Konstrukte, die sich außerhalb des Mainstreams befinden und von den aufgestellten Standards nicht eindeutig definiert werden, sind z.B.:

- die Größe von Datentypen,
- die Reihenfolge der Auswertung von Operanden, Seiteneffekten und Funktionsargumenten,
- die Vorzeichenbehaftung von Zeichen,
- arithmetisches oder logisches Verschieben,
- die Byte-Reihenfolge,
- die Ausrichtung von Strukturen und Klassenelementen und
- Bit-Felder.

Eine Faustregel besagt: keine Eigenschaften in seine Quellen einzubauen, die so ungewöhnlich oder unklar sind, daß man ein Experte auf dem Gebiet von Sprachdefinitionen sein muß, um sie zu verstehen. Ein Beispiel hierfür ist die uneinheitliche Nutzung von komplizierten Zeiger- und Vektorausdrücken in der Programmiersprache C bzw. C++, wie '*(a+i)', statt 'a[i]', falls man auf den Inhalt eines Vektors verweist oder f(int (*vektor)[12]), statt f(int vektor[2][12]) vereinbart, um klar zu machen, daß hier ein Vektor an eine Funktion zu übergeben ist, wobei der Sprachstandard diese Äquivalenz wohlwollend duldet. Auch sollte man mehrere Compiler ausprobieren und alle Compiler-Warnungen aktivieren, um so weitere Portabilitätsprobleme zu erkennen, die sonst vielleicht im Verborgenen bleiben würden.

Standardbibliotheken

Neben einer präzisen Sprachdefinition, die nur in relativ wenigen und eindeutig identifizierbaren Punkten implementationsabhängig ist, sind eindeutig definierte Standardbibliotheken, die grundlegende Funktionsbereiche abdecken, ebenfalls eine wichtige Voraussetzung für die Portabilität von Software-Systemen. Standardbibliotheken werden meist gleichzeitig mit einer Sprache definiert und mit ihr zusammen in der Systemumgebung angeboten, sind aber kein Teil der Sprache selbst. Bibliotheken arbeiten häufig eng mit dem zugrunde liegenden Betriebssystem zusammen und können daher trotzdem unportable Teile enthalten. Auch bei den Implementierungen verschiedener Standardbibliotheken gibt es eine Reihe von Merkmalen, die nicht Bestandteil eines Standards sind. So ist z.B. die, in vielen standardkonformen Umgebungen vorhandene, String-Kopierfunktion 'strdup' kein Teil des ANSI-C-Standards, wird aber dennoch, fälschlicherweise, als Standardfunktion angenommen und routinemäßig eingesetzt. Somit ist es auch hier aus Portabilitätssicht sehr wichtig, sich strikt an die Definition der benutzen Standardbibliothek zu halten und das Software-System in einer Vielzahl von Umgebungen zu testen.

Wie bereits erläutert, gibt es zahlreiche weitere Standards, die ebenfalls nicht Teil einer Programmiersprachendefinition sind. Dazu zählen zum Beispiel Betriebssystem- und Netzwerkschnittstellen, Grafikschnittstellen usw. Einige davon, wie POSIX, sind für mehr als ein Betriebssystem bestimmt, andere hingegen, wie die verschiedenen Windows-API's, für ein bestimmtes Betriebssystem spezifisch. Auch hier sollten, möglichst auf breiter Ebene, etablierte Standards ausgewählt und nur bestimmte Hauptkomponenten sowie die am häufigsten benutzten Teile, Verwendung finden.

Datenaustausch

Die einfachste Form des Datenaustausches zwischen beliebigen Rechnersystemen stellen Textdaten dar. Diese können durch verschiedene Werkzeuge leicht gelesen und manipuliert werden. Beim Austausch von Text kommt es jedoch immer wieder zu bestimmten Verwirrungen. So nutzen z.B. PC-Systeme ein 'carriage return' und ein 'newline' (CRLF), um eine Zeile zu beenden, wobei Unix-Systeme nur 'newline' verwenden. Ohne diese Markierungen des Zeilenendes, könnte man eine Textdatei als eine riesige Zeile betrachten und die Zahl der Zeilen bzw. Zeichen wäre somit falsch oder würde sich unerwartet ändern. Werden für dieses Portabilitätsproblem Standardschnittstellen eingesetzt, die CLRF's in jeder Umgebung konsistent behandeln, wären damit weitere Portierungsprobleme eliminiert. Wobei für häufig zu transportierende Textdateien, Programme zur Umwandlung der beiden Formate notwendig sind.

Binärdateien hingegen benötigen spezialisierte Werkzeuge und können, manchmal sogar auf dem gleichen Rechnersystem, nicht zusammen verwendet werden. Sie sind mit großen Portabilitätsproblemen behaftet (Reihenfolge der Bytes). Um Binärdateien von einem Rechnersystem auf ein anderes zu übertragen, sollten diese, auch wegen der Fehlererkennung, für den Transport portabel kodiert werden. Binäre Daten sind bei vielen Aspekten in Netzwerken unentbehrlich, da sie erheblich schneller und kompakter zu dekodieren sind. Benutzt man für den Datenaustausch eine festgelegte Byte-Reihenfolge für Sender und Empfänger, sind die wichtigsten Portabilitätsprobleme bereits behoben. Verallgemeinert läßt sich dieses Vorgehen auch auf Strukturen anwenden. Man muß sich jedoch darauf verständigen, in welcher Reihenfolge die Bytes und wie viele davon übertragen werden.

Organisation der Programme und Isolation

Um portable Software-Systeme zu erhalten bedient man sich zweier Ansätze, die als 'Vereinigung' und 'Schnitt' bezeichnet werden. Vereinigung bedeutet, für jede Zielumgebung speziellen Code zu schreiben und alles so gut wie möglich zusammenzufügen, zum Beispiel mit bedingter Übersetzung. Die Vereinigung benutzt die Eigenschaften eines Systems und macht die Übersetzung und Installation von den lokalen Gegebenheiten des Rechnersystems abhängig. Damit wird der entstandene Quellcode mit einer Vielzahl von Umgebungen fertig und nutzt die Stärken jedes Systems. Der größte Nachteil dieser Vorgehensweise liegt in der Größe und Komplexität des Installationsprozesses und der Komplexität des zugrunde liegenden Quellcodes, der eine Vielzahl von bedingten Übersetzungen enthält. Bedingte Übersetzungen sind, wie Präprozessoranweisungen, meist schwierig zu verwalten und die Informationen neigen dazu, sich über den gesamten Quelltext zu verteilen. Außerdem benötigen sie für jede Umgebung einen Strang für bedingte Übersetzung. Sie können sich mit Steuerungen des Ausführungsflusses vermischen, was zu einer schlechten Lesbarkeit der Quellen führt, und so gut wie gar nicht getestet werden. Je freier ein Software-System von bedingten

Übersetzungen ist, desto einfacher und zuverlässiger ist dessen Wartung, Konfiguration und Installation.

Bei der Vorgehensweise nach dem Schnitt-Ansatz, gestaltet man seinen Code möglichst so, daß er auf jedem Rechnersystem ohne Anpassung funktioniert. Systemabhängigkeiten werden in einzelnen Quelltextdateien gekapselt, die dann als eine Art Schnittstelle zwischen dem Software-System und dem zugrunde liegenden Rechnersystem agieren. Der Schnitt verwendet nur die Merkmale, die auf allen Zielumgebungen vorhanden sind. Nachteilig hierbei ist die Forderung nach universeller Verfügbarkeit, was die möglichen Zielumgebungen oder die Fähigkeiten des resultierenden Software-Systems reduzieren kann, bzw. in bestimmten Umgebungen zu Performanceverlusten führt. Der größte Vorteil des Schnitt-Ansatzes ist jedoch die vollständige Vermeidung von bedingtem Code und die saubere Trennung von bestimmten Abhängigkeiten in separaten Dateien. Diese können dann je nach Bedarf ausgetauscht werden, für jede Umgebung eine Datei. Auch sollten innerhalb der Dateien Systemabhängigkeiten hinter Schnittstellen verborgen bleiben und, nach dem Prinzip der Abstraktion, nichtportable Teile eines Software-Systems kapseln. Das Software-System nutzt dann nur Funktionen aus dieser Schnittstelle, welche wiederum alle Fähigkeiten des zugrunde liegenden Computersystems nutzt. Das erfordert zwar recht unterschiedliche Implementierungen der einzelnen Schnittstellen, aber das Software-System, das nur die Schnittstellen verwendet, ist davon unabhängig und sollte bei einer Portierung keine weiteren Änderungen erforderlich machen.

Portabilität und Upgrades

Verschiedene Portabilitätsprobleme treten immer dann auf, wenn die zugrunde liegende System-Software, wie z.B. das Betriebssystem, während der Lebensdauer des Software-Systems modifiziert wird und somit Inkompatibilitäten zwischen vorhandenen Programmversionen hervorruft. Somit können Änderungen an der System-Software unterschiedliche Versionen eines bestimmten Software-Systems erzeugen, die sich zwar absichtlich unterschiedlich verhalten, aber auch zu unabsichtlichen Portabilitätsproblemen und Versionskonflikten führen. Eine konsistente, auf die einzelnen Versionen abgestimmte Namensgebung, kann hier viel zur Übersichtlichkeit beitragen.

Ein weiteres Problem ist die Kompatibilität zu vorhandenen Software-Systemen und deren Daten. So ist es allgemein üblich, daß neuere Versionen einer bestimmten Software, Daten aus älteren Versionen uneingeschränkt verarbeiten können (Abwärtskompatibilität). Somit ist es ebenfalls notwendig, darauf zu achten, daß man nicht in Konflikt mit bestimmten Vorgängerversionen und den Daten, die davon abhängen, in Konflikt gerät.

Internationalisierung

Betrachtet man das menschliche Umfeld, in das Computersysteme eingebettet sind, so zeigt sich hier eine Vielfalt an Sprach- und Kultureigenheiten, auf die bei der Portierung von Software-Systemen ebenfalls Rücksicht genommen werden muß. Das Prinzip der Internationalisierung bedeutet im Bereich der Portabilität von Software-Systemen, daß dieses so gestaltet werden und gemäß der Spezifikation funktionieren muß, daß keine Annahmen über das kulturelle Umfeld zugrunde liegen.

Ein erstes Problem sind hierbei die reichhaltigen Zeichensätze in vielen Teilen der Welt. So reicht hier die bereits angesprochenen ASCII-Codierung der verschiedenen Zeichen nicht aus. Auch sind die in vielen Teilen Europas benutzten, vollen 8-Byte-Erweiterungen des ASCII-Zeichensatzes, z.B. Latin -

1-Kodierung, für viele Sprachen nicht ausreichend. So erfordern Kodierungen im asiatischen Raum beispielsweise 16 Bit pro Zeichen. Um diese Situation zu verbessern, wurde der 16-Bit-Unicode-Zeichensatz geschaffen, wodurch eine konsistente Kodierung für alle Sprachen der Welt möglich wird. Dadurch entsteht aber wieder ein neues Problem, welches daraus resultiert, daß die Zeichen nicht mehr länger in ein Byte passen und somit Verwirrungen um die Byte-Reihenfolge auftreten. Um dieses Problem zu vermeiden, benutzt man für Unicode-Dokumente, vor der Übertragung zwischen Software-Systemen oder über Netzwerke, eine Byte-Strom-Kodierung namens UTF-8. Hierbei kann man von einer Abwärtskompatibilität zur ASCII-Kodierung ausgehen, was Software-Systemen, die Text als nicht interpretierten Byte-Strom ansehen, ermöglicht, mit Unicode-Text in jeder Sprache zu funktionieren. C und C++ unterstützen 'wide characters' (d.h. breite Zeichen, die aus ganzzahligen Werten mit 16 oder noch mehr Bits bestehen), jedoch ist die Interpretation des Zeichensatzes und die Definition der Byte-Strom-Kodierung gut in den Bibliotheken versteckt und nur schwer zu ermitteln. Die Situation ist zur Zeit bestenfalls unbefriedigend und es herrscht keine Übereinstimmung darüber, welcher Zeichensatz benutzt werden soll. Eine Entscheidung für einen bestimmte Zeichensatz-Kodierung, sollte jedoch immer über das Standard-ASCII hinausgehen.

Mithin sollten bestimmte Sprachannahmen bei der Nutzung von Benutzeroberflächen ebenfalls vermieden werden. Dazu gehören: die Breite von Dialogfeldern für Meldungen in verschiedenen Sprachen, das Einhalten des Prinzips der Lokalität für die Verwaltung der einzelnen Meldungen sowie der länderspezifischen Datumsformate oder kulturabhängigen Icons in grafischen Benutzeroberflächen.

Aus den hier betrachteten Problemen lassen sich somit bestimmte, allgemeine Grundsätze für portablen C bzw. C++-Code ableiten:

- Einhaltung der vorhandenen Standards (ANSI-C bzw. C++),
- Programmierung im C- bzw. C++-Mainstream,
- Kompilieren mit unterschiedlichen Compilern auf höchster Warnstufe,
- Nutzung von C- und C++-Standardbibliotheken,
- Benutzung von Text für den Datenaustausch,
- Benutzung einer festgelegten Byte-Reihenfolge für den Datenaustausch,
- Nutzung überall verfügbarer Merkmale,
- Vermeidung von bedingter Übersetzung (Vereinigung),
- Konzentration von Systemabhängigkeiten in separaten Dateien (Schnitt),
- Verstecken von Systemabhängigkeiten hinter Schnittstellen (Kapselung),
- Namensänderungen bei Spezifikationsanpassungen,
- Bewahren der Kompatibilität zu vorhandenen Software-Systemen und Daten,
- ASCII-Kodierung und Landessprache nicht voraussetzen.

2.2.3.2 Architekturen des betrachteten Portierungsfelds

Portiert man Software-Systeme von einer Hostumgebung in eine bestimmte Zielumgebung, so muß festgestellt werden, welche System-Software (Betriebssystem, Programmiersprachenwerkzeuge etc.) aus der Software-Umgebung und welche Hardware aus der Hardware-Umgebung, dem Portierungsprozeß zugrunde liegen (Plattform). Aus diesen Kenntnissen definiert sich dann im weiteren, das spezifische Portierungsfeld des Portierungsprozesses. Auf die System-Software im

programmiersprachlichen Bereich und die zugrunde liegenden Rechnersysteme wurde bereits eingegangen. Es soll nun der Teil der System-Software aus der Software-Umgebung betrachtet werden, der als Schnittstelle zwischen Rechnersystem und Software-System agiert. Dies sind die, jeweils dem Portierungsprozeß zugrunde liegenden Windows-Betriebssysteme und dessen Architekturen. Auf der 16-Bit-Seite des Portierungsfelds (Hostumgebung) findet sich Windows 3.1, inklusive naher verwandter, wie z.B. Windows for Workgroups. Die 32-Bit-Seite des betrachteten Portierungsfelds (Zielumgebung) stellen die Betriebssysteme Windows 9x und Windows NT mit ihren jeweiligen Architekturen dar. Im folgenden sollen deshalb die Schlüsselkonzepte dieser Architekturen näher beschrieben werden (siehe hierzu auch Anhang A).

Windows 3.x

Windows 3.x ist ein ereignisgesteuertes 16-Bit-Betriebssystem. Die Philosophie der Ereignissteuerung von Windows ist es, auf bestimmte Ereignisse – Tastatur- oder Mausereignis etc. – mit entsprechenden Prozeßoperationen zu antworten, bis das Ereignis, entsprechend seiner Zielstellung, behandelt worden ist. Die Anwendungen liegen nicht in einer Warteschleife und fragen nach, ob der Benutzer irgendwelche Eingaben getätigt hat. Statt dessen werden sie von Windows benachrichtigt, sobald ein derartiges Ereignis eintritt, z.B. eine Taste gedrückt wird. Dieser, gegenüber herkömmlichen DOS-Anwendungen, veränderte Ansatz wird daran deutlich, daß nicht mehr der Entwickler den Programmfluß vorgibt, sondern daß primär der Anwender darüber bestimmt. Die Aufgabe der Ereignisbehandlung und -steuerung übernimmt der sogenannte Ereignis-Manager. Er ist die zentrale Instanz für die Vermittlung zwischen der Anwendung und den Basisdiensten des Betriebssystems. Betrifft ein Ereignis ein Objekt der grafischen Oberfläche, so wird diese Nachricht an das GDI (Graphics Device Interface) weitergeleitet und dort verarbeitet. Das GDI ist für die grafische Darstellung von Objekten auf der Benutzeroberfläche zuständig. Wird über ein Maus-Ereignis z.B. die Größe eines Fensters verändert, werden nicht nur Nachrichten an das GDI, sondern auch an den Kernel geschickt, der die neuen Daten über die zugrunde liegende Hardware verarbeitet und im Speicher ablegt. Der Kernel ist der Teil des Systemkerns, der alle peripheren Elemente des Rechnersystem rund um die CPU kontrolliert. Bei Windows 3.1 unterscheidet man zwischen dem Protected-Mode für Prozessoren mit der Architektur einer 80286-CPU und dem für 80386-Modelle. Somit wird für jeden Prozessortyp ein entsprechend kompatibler Kernel – KRNL286 (Standard-Mode) oder KRNL386 (Enhanced-Mode) – geladen.

Um eine bestmögliche Anpassung an die Prozessoren zu erreichen, kennt Windows drei Betriebsarten, die sich an den Fähigkeiten und den Betriebsarten der jeweils zugrundeliegenden CPU des Rechnersystems orientieren. An erster Stelle wäre der `Real-Mode´ zu erwähnen, der aus Kompatibilitätsgründen zu Rechnersystemen mit 8086/88-Prozessoren existiert. Im Real-Mode laufen nicht nur ältere DOS-Anwendungen, es lassen sich sogar, speziell für die vorherige Fassung (2.x) erstellte Treiberprogramme, zur Anpassung von Fremd-Hardware direkt verwenden. Im Real-Mode wird für Speicherweiterungen, über die engen Verwaltungsgrenzen von DOS – 640 KB – hinaus, prozessorbedingt, die Expanded-Memory-Technologie benutzt. Alle weiteren Modi arbeiten mit verbesserten Technologien, wie dem Extended-Memory, welche durch nachfolgende Prozessoren unterstütz wird. Eine Grundvoraussetzung für den erfolgreichen Aufruf der nachfolgenden Modi ist die Verwaltung des externen Speichers gemäß XMS-Spezifikation (eXtendend Memory Specification). Damit wird der Datenaustausch zwischen Erweiterungsspeicher der Speichermodule und den unteren 640 KB des DOS-Speichers realisiert und ermöglicht somit – anders als bei der

Bereitstellung von Erweiterungsspeicher über externe Expandend Memory Speicherkarten – die direkte Nutzung des erweiterten Adreßraum, von bis zu 16 MB

Der Standard-Mode ist auf die speziellen Eigenschaften von Rechnersystemen mit 80286-CPU's ausgerichtet. Alle Anwendungen laufen hierbei, wie auch im folgenden Enhanced-Mode, im 16-Bit-Protected-Mode dieses Prozessors. Wesentliche Unterschiede zum Real-Mode liegen in Schutzmechanismen für die Speicherverwaltung, verbunden mit einer segmentorientierten Adressierung des Speichers über Deskriptor-Tabellen, sowie grundsätzlichem Support von Task-Wechseln, durch spezielle Strukturen und Befehle. Der segmentorientierte Adressierungsmechanismus der 80286-Prozessoren kann bis zu 16 MB an Speicher adressieren, wobei das erste Megabyte für konventionellen und hohen Speicher reserviert ist. Die anderen 15 MB können über den, bereits angesprochenen, Extended-Memory-Mechanismus direkt als physisch vorhandener Speicher verwaltet oder als virtueller Speicher, auf einer Festplatte ausgelagert, adressiert werden. Ebenfalls verwaltet über den Extended-Memory-Manager. Weder im Real- noch im Standard-Modus laufen DOS-Programme im Multitasking-Betrieb. Hierbei handelt es sich um eine einfache Umschaltmöglichkeit zwischen exklusiv ablaufenden Programmen, die nur dann laufen, wenn der Anwender sie aktiviert.

Aus softwaretechnischer Sicht ist der `erweiterte 386-Modus´ (Enhanced-Mode) die aufwendigere, aber auch leistungsfähigere Windows-Betriebsart, im Vergleich zum Standard- oder Protected-Mode. Sie macht starken Gebrauch von den Fertigkeiten des 80386-Prozessors, insbesondere des Virtual-86-Mode, einer speziellen Variante des Protected Mode. Der Prozessor kann so, als Task, mehrere Programme, die für den 8086 konzipiert sind, ausführen. Er geht über die im Standard-Modus vorhandenen Verwaltungs- und Schutzmechanismen hinaus, indem er die virtuelle Speicherverwaltung des 80386 nutzbar macht. Im erweiterten 386-Modus besteht Windows aus virtuellen Maschinen (VM) und virtuellen Geräten, deren Miteinander von dem sogenannten `Virtual Machine Manager´ (VMM) koordiniert wird. Der Windows-Funktionsvorrat residiert in einer virtuellen Maschine, der System-VM. Die Anwendungen laufen in jeweils eigenen VM's ab, die sowohl aus 8086-Code, als auch aus Code im 16-Bit-Protected-Mode bestehen können. Nur im erweiterten 386er-Modus laufen unter Windows mehrere DOS-Tasks simultan zu Windows-Anwendungen. Eine Verwaltungsinstanz billigt ihnen reihum Rechenzeit zu. Windows behandelt sich selbst, einschließlich aller laufenden Windows-Anwendungen, als eine Task in diesem Multitasking-Gefüge. Der VMM selbst läuft im Protected-Mode, regelt die Speicherverwaltung, verteilt die Rechenzeit auf einzelne VM's und stellt den Kontakt zur Hardware mit Hilfe der virtuellen Geräte her. Virtuelle Geräte dienen hauptsächlich dazu, gleichzeitige Zugriffe auf Peripheriegeräte zu serialisieren, d.h. die Zugriffe die mehrere Tasks wechselseitig durchführen, in eine vernünftige Reihenfolge zu bringen, ohne daß dabei Konflikte auftreten. So können sich die Anwendungen beispielsweise einen Drucker konfliktfrei teilen. Dies ist um so entscheidender, da DOS-Anwendungen aufgrund ihrer exklusiven Ausführung, auf solch eine Unterstützung angewiesen sind und die belegten Ressourcen den anderen Anwendungen ansonsten nicht gemeinsam zur Verfügung stehen könnten. Die Virtualisierung ist somit unerläßlich, wenn Windows- und DOS-Programme, voneinander ungestört, in einer Umgebung kooperativ zusammenarbeiten sollen. Die herkömmlichen Windows-Treiber arbeiten eng mit den virtuellen Gerätetreibern zusammen. Beispielsweise schlägt der Treiber für die serielle Schnittstelle Alarm, wenn der VCD (virtuelles COM-Device) feststellt, daß eine Windows-Anwendung versucht, einen COM-Port zu nutzen, obwohl dieser bereits durch eine DOS-Applikation beansprucht wird. Allen drei Modi ist gemein, daß Windows-Applikationen eine identische Schnittstelle zu den diversen Windows-API-Funktionen sowie eine einheitliche, abwärtskompatible Speicherverwaltung nutzen.

Viele der benutzten Begriffe gehören eindeutig in den Bereich `Benutzeroberfläche´, so z.B. die Maus- oder Fenstersteuerung. Bereichsfremd sind aber Multitasking und virtueller Speicher, denn dabei handelt es sich eher um Leistungsmerkmale eines Betriebssystems. So kann man Windows, keiner der beiden Kategorien eindeutig zuordnen. Ein direkter Vergleich mit anderen Oberflächensystemen, wie zum Beispiel mit dem unter Unix gebräuchlichen `X-Window-System´, wäre folglich falsch. Windows 3.1 ist nicht nur eine grafische Benutzeroberfläche, sondern auch eine Erweiterung des Betriebssystems MS-/PC-DOS.

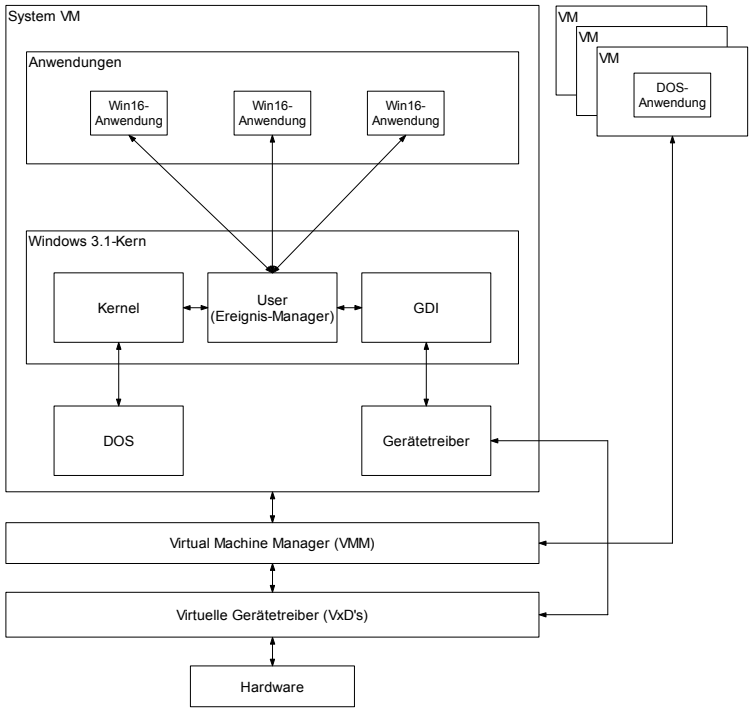


Abbildung 6: Systemarchitektur von Windows 3.1 (Enhanced-Mode)

Windows 9x (Windows 95 und Windows 98)

Windows 9x ist ein 32-Bit-Betriebssystem und zeichnet sich durch höhere Arbeitsgeschwindigkeit sowie robusteren Aufbau, im Vergleich zu Windows 3.1, aus. Das Architekturdesign von Windows 9x basiert im Wesentlichen auf der von Windows 3.1. Es beinhaltet eine Reihe von Kompromissen, die getroffen werden mußten, um als modernes Betriebssystems möglichst kompatibel zu seinen Vorgängern zu sein. Die wesentlichen Änderungen betreffen die neuartige Architektur für Gerätetreiber, das Dateisystem sowie die Grafik-Engine, zusammen mit dem Subsystem für Druckausgabe, der Kommunikation und Multimedia. Auch sind Netzwerkfunktionen jetzt direkter Bestandteil des Betriebssystems. Windows 9x verfügt, im Vergleich zu Windows 3.1, über:

- ein voll integriertes 32-Bit-Betriebssystem für den Protected-Mode (Vorteil: Es macht die Benutzung einer separaten Kopie von MS-DOS überflüssig);
- eine Unterstützung von preemptiven Multitasking und Multithreading (Vorteil: Verringerung der Antwortzeiten des Systems und störungsfreier Hintergrundbetrieb von Programmen);

- die Möglichkeit 32-Bit-Dateisysteme zu installieren, wie VFAT, CDFS und Netzwerk-Redirector mit besserer Arbeitsgeschwindigkeit, langen Dateinamen und einer offenen Architektur für zukünftige Erweiterungen;
- 32-Bit-Gerätetreiber (Vorteil: höherer Geschwindigkeit und Stabilität sowie intelligente Ausnutzung des Hauptspeichers);
- erhöhte Stabilität von Anwendungen und Aufräumaktionen beim Absturz eines Programms (Vorteil: bessere Stabilität des gesamten Systems);
- eine dynamische Konfiguration der Umgebung (Vorteil: seltenere Anpassung des Systems durch den Anwender nötig) sowie
- erhöhte Systemkapazitäten (Vorteil: verbesserte, gleichzeitige Ausführung von Programmen).

Alle Funktionen werden durch einen modularen Aufbau unterstützt und in Abbildung 7 dargestellt.

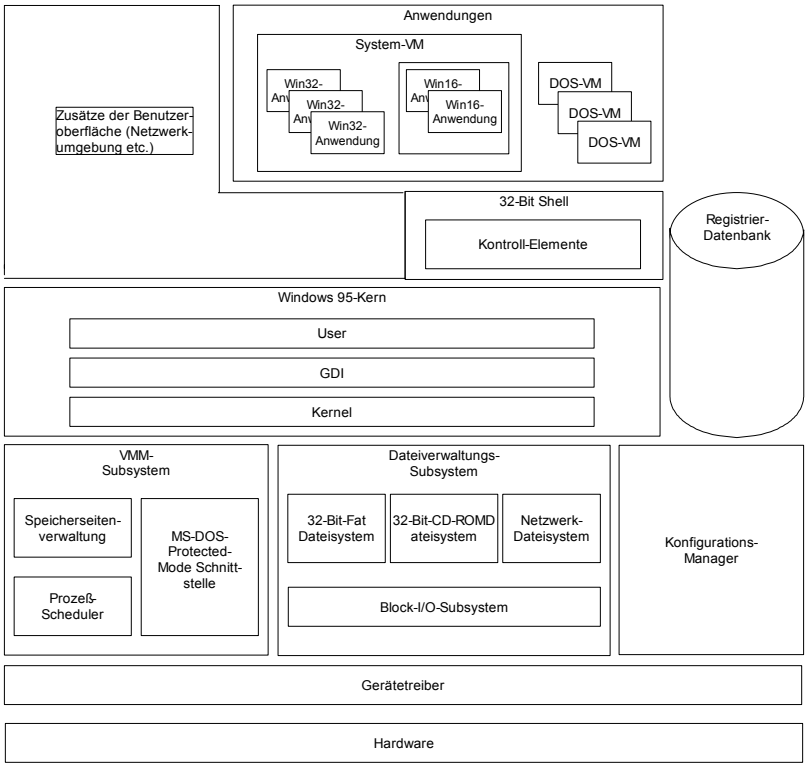


Abbildung 7: Systemarchitektur von Windows 9x

Die zentrale Speicherstelle für Systeminformationen ist die Registrierdatenbank. Sie löst die Dateienvielfalt und dezentrale Speicherung von system- oder anwendungspezifischen Daten durch 'Autoexec.bat', 'Config.sys' und INI-Dateien aus Windows 3.1, vollständig ab und erscheint aus Anwendersicht als eine einzige Speicherstelle für Systeminformationen. Sie wird auf Systemebene durch zwei Dateien, nämlich mit Informationen über das Rechnersystem selbst (System.dat) und die Anwenderdaten (User.dat) repräsentiert. Die Registrierdatenbank ist eine hierarchische Datenbank, die sich aus einzelnen Registrierschlüsseln zusammensetzt. Sie ermöglicht den Fernzugriff auf Win32-Anwendungen sowie deren Fernverwaltung über das Netzwerk und dient als zentrale Speicherstelle für gerätespezifische Daten, die u.a. von den Plug-and-Play-Komponenten ausgewertet werden.

In Windows 9x werden einzelne Geräte, wie im Enhanced Mode von Windows 3.1, über virtuelle Gerätetreiber angesprochen. Auch hier bezieht sich 'VxD' auf einen allgemeinen virtuellen Treiber (Universal-Treiber), wobei das 'x' für den jeweiligen Gerätetyp steht (Virtual Display Driver, Virtual Timer Driver etc.). Die meisten virtuellen Gerätetreiber verwalten Hardware, aber es gibt auch solche, die Software verwalten, z.B. DOS-Gerätetreiber oder speicherresidente Programme. Die neuartige Architektur der Gerätetreiber teilt sich auf in Unitreiber und Minitreiber. Für die meisten Gerätetypen werden hierbei vom Betriebssystem allgemein gültige Universaltreiber angeboten und enthalten Schnittstellen zu den jeweiligen Komponenten von Windows 9x (z.B. Drucksubsystem etc). Hardware-Firmen können nun einen entsprechenden Minitreiber entwickeln, der zusätzliche Funktionen unterstützt, die der Unitreiber nicht kennt. In Windows 9x werden VxD's dynamisch geladen, in Windows 3.1 statisch, d.h. das System lädt immer nur diejenigen Treiber in den Speicher, die gerade benötigt werden. Das virtuelle Gerät, das ein jeweiliger Treiber erzeugt, verwaltet den Gerätezustand für jede Anwendung und stellt somit sicher, daß eine Anwendung ohne Probleme auf das Gerät zugreifen kann.

Um den Informationsaustausch mehrerer Geräte, unterschiedlichen Typs, über ein Bussystem und auch Plug-and-Play zu ermöglichen, ist ein entsprechendes Konfigurationsmanagement erforderlich. Dieser Konfigurationsprozeß wird mit Hilfe des Konfigurations-Managers organisiert, der sicherstellt, daß jedes Gerät einen Hardware-Interrupt, eine I/O-Adresse und andere Ressourcen, ohne Konflikte mit anderen Geräten, nutzen kann. Der Konfigurationsmanager überwacht außerdem Anzahl und Typ der angeschlossenen Geräte sowie die Konfiguration bei Hardware-Änderungen.

Der 'Virtual-Machine-Manager' verteilt im Vergleich zum Konfigurationsmanager nicht die Ressourcen für alle Geräte, sondern für alle Anwendungen und Systemprozesse. Er erzeugt und verwaltet virtuelle Maschinen, in denen die Anwendungen und Systemprozesse arbeiten. Er ist ein separater Speicherbereich, der aus der Perspektive einer Anwendung, wie ein einzelnes Computersystem erscheint. Windows 9x besitzt eine System-VM, in der alle Systemprozesse und alle Win32-Anwendungen – in jeweils einem eigenem Adreßraum –, und Win16-Anwendungen – in einem gemeinsam genutzten Adreßraum– laufen. Eine MS-DOS-Anwendung läuft in ihrer eigenen DOS-VM, da diese zum Teil exklusiven Zugriff – MS-DOS-Modus des VMM – auf die Systemressourcen erfordern und währenddessen keiner anderen Anwendung ein Zugriff auf Rechenzeit oder Systemressourcen zugeordnet wird. Der Prozeß-Scheduler ordnet die Systemressourcen den Anwendungen und anderen Prozessen zu und legt für deren gleichzeitige, d.h. preemptive, Ausführung, bestimmte Zeitscheiben fest. Die Speicherseitenverwaltung erfolgt, wie bei Windows NT, nach der Methode des Demand-Paging (Code und Daten werden seitenweise aus dem physischen Speicher in eine Datei oder auf die Festplatte ausgelagert). Dieser Mechanismus arbeitet auf einem flachen, linearen Adreßraum, auf den mit 32-Bit-Adressen zugegriffen wird. Das Modell der linearen Speicheraddressierung vereinfacht die Entwicklung von Software-Systemen und beseitigt die Leistungseinbußen durch segmentierten Speicher (das Betriebssystem und die Anwendungen unter Windows 3.1 können Speicher außerhalb eines Segments nur unter Leistungsverlusten ansprechen). Es ermöglicht jeder 32-Bit-Komponente den Zugriff auf 4 Gigabyte adressierbaren Speicher, wobei jede Anwendung bis zu 2 Gigabyte 'privaten' Speicher anfordern kann.

Windows 9x unterstützt durch seine geschichtete Dateisystemarchitektur unterschiedliche Dateisysteme, wie VFAT (Virtual File Allocation Table) oder CDFS (CD-ROM File System) und erhöht damit, im Vergleich zu Windows 3.1, die Geschwindigkeit beim Zugriff auf Laufwerke und Dateien. Zu den Funktionen der Architektur gehören, unter anderem, die Unterstützung langer

Dateinamen und ein dynamischer Cache für Zugriffe auf Dateien und das Netzwerk. Die Dateisystemverwaltung wird durch den IFS-Manager geregelt (Installable File System Manager). Das Block-I/O-Subsystem von Windows 9x erweitert die 32-Bit-Plattenzugriffe gegenüber Windows 3.1. Damit erhöht sich die Geschwindigkeit des gesamten Dateisystems.

Ähnlich wie unter Windows 3.1, besteht der Systemkern von Windows 95 aus drei Komponenten: User, Kernel und GDI (Graphical Device Interface). Jede Komponente besteht aus einem Paar von Dll's (einer Dll für 16-Bit-Systeme und einer für 32-Bit-Systeme), die alle Dienste für Anwendungen enthalten. Hier spiegelt sich der Kompromiß zur Rückwärtskompatibilität wieder. Windows 9x benutzt weiterhin 'alten' 16-Bit-Code, wenn entweder die Kompatibilität zu alten Anwendungen erhalten bleiben soll oder der 32-Bit-Code die Speicheranforderungen erhöht, ohne dabei die Ausführungsgeschwindigkeit zu verbessern. Alle I/O-Subsysteme, z.B. Netzwerkroutinen oder die Dateisysteme, und die Gerätetreiber sind komplett in 32 Bit gehalten. Ebenso die Komponenten zur Zeitscheiben- und Speicherverwaltung, inklusive des Kernel und des VMM. Die Module laufen, obwohl sie zum System gehören, nicht im Kernel-Mode des Prozessors, denn ein ständiger Privilegwechsel würde erhebliche Leistungseinbußen bei Systemaufrufen mit sich bringen.

Windows 9x besitzt eine 32-Bit-Benutzerschnittstelle, die auf dem Windows-Explorer basiert und eine Reihe von Werkzeugen, wie die Netzwerkumgebung, als Zusatz zur Benutzeroberfläche zur Verfügung stellt. Alle Anwendungen und Werkzeuge (arbeiten auf demselben Systemniveau wie Win32-, Win16- und MS-DOS-Anwendungen) können die Standard-Kontrollelemente der Shell benutzen (z.B. Standarddialogfelder, Symbolleisten oder Baumanzeigen).

Windows NT

Windows NT bietet, aufgrund seiner auf Portabilität und Systemsicherheit ausgerichteten Architektur, eine, im Vergleich zu Windows 9x, grundlegend andere Struktur. Zwei wesentliche Elemente beherrschen hierbei das Architekturfeld: die Subsysteme, die die verschiedenen Betriebssystememulationen bereitstellen und das eigentliche Basisbetriebssystem.

Das NT-Basissystem läuft im Kernel-Modus des Prozessors und in einem, von den Subsystemen und Anwendungen, isolierten Adreßraum. Es besteht aus drei wesentlichen Funktionsgruppen: Einem teilweise hardwareabhängigen Kern, der sich um das Scheduling und die Interrupt-Vorverarbeitung kümmert, der Hardware Abstraktionsschicht (HAL), die viele Systemdesign-bezogene Eigenschaften, wie etwa DMA-Spezifika, vor dem Rest des Systems kapselt und der Executive, die alle restlichen prozessor- und architekturunabhängigen Komponenten enthält. Unter Windows NT können Anwendungen laufen, die für verschiedenen Betriebssysteme geschrieben worden sind (Win32, OS/2, DOS, Win16, POSIX). Um diese Flexibilität zu erreichen, mußte der Betriebssystemkern recht allgemein gehalten werden. In seinen Aufgabenbereich fallen deshalb nur elementare Dienste, wie Prozeß- und Threadverwaltung, Speicherverwaltung etc. Auf Rechnersystemen ohne INTEL-Befehlssatz – etwa auf den Alpha- oder Mips-Systemen – wird der Binärcode der DOS-, Windows- und OS/2-Programme zur Laufzeit interpretiert und in entsprechende Anweisungen für den Zielprozessor umgesetzt. Genauso werden Aufrufe an das OS/2-API zur Laufzeit so modifiziert, daß sie auf die entsprechenden Funktionen im OS/2 Subsystem verweisen. Auch innerhalb der Executive verwendet NT oft mehrere Schichten. So handelt es sich bei Dateisystemen letztendlich um Treiber, die auf dem Treiber für blockorientierte Geräte aufsetzen, der seinerseits wieder auf den z.B. SCSI-Treiber zurückgreift. Sie haben somit, für den NT-Kern, den Status von Gerätetreibern. Für die

Subsysteme werden abstrakte Systemdienste exportiert, aus denen diese die notwendigen Funktionen, für die Emulation, ableiten müssen. Die Gerätetreiber und installierbaren Dateisysteme sind so vom Rest des Systems konsequent abgetrennt und werden durch ein einheitliches Protokoll vom Kernel-I/O-Manager angesprochen. Treiber können durch die Bildung 'multipler Layer' aufeinander aufbauen und ein installierbares Dateisystem, wie z.B. das New Technology File System (NTFS), kann damit mehrere Geräte über die zugehörigen Treiber gleichzeitig bedienen. Weitere Gerätetreiber können jederzeit dynamisch hinzugefügt bzw. entfernt werden. Ein laufendes NT-System muß daher nicht zwangsläufig heruntergefahren werden, um neue oder verbesserte Komponenten, z.B. weitere Dateisysteme, verfügbar zu machen.

Eine Spezialform für Gerätetreiber bieten die Virtuellen DOS Maschinen (VDM), die NT zum Ausführen von DOS und 16-Bit-Windows Programmen verwendet. Um proprietärer Hardware, wie Faxmodemkarten oder Hand-Scanner nicht vollends den Weg zu versperren – direkte Hardwarezugriffe sind unter NT nicht zulässig –, lassen sich diese über spezielle virtuelle Gerätetreiber (VDD) in ein NT-System einbinden. Ein solcher VDD kann direkte Hardwarezugriffe in einer VDM abfangen und an einen Kernel-Mode-Treiber weiterleiten.

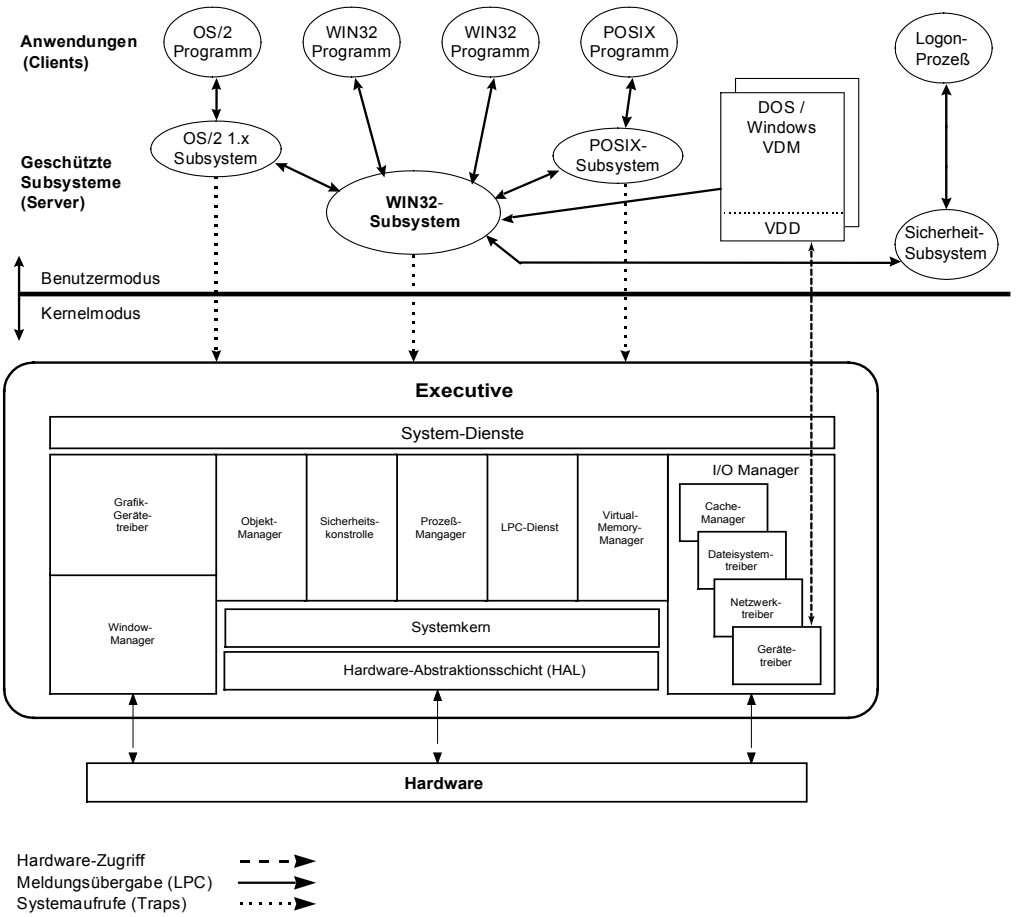


Abbildung 8: Systemarchitektur Windows NT

Die NT-Executive stellt den aufbauenden Subsystemen alle Funktionen zur Verfügung, die diese benötigen, um ihr jeweiliges API nach außen hin zu implementieren. Alle höheren Funktionen des Betriebssystems, wie die grafische Oberfläche, werden von den Subsystemen implementiert, die wie eine normale Anwendung im User-Mode des Prozessors laufen und als Server ihre Clients bedienen.

Windows NT erlaubt so ein beliebiges nebeneinander von API's und Softwareumgebungen. Schnittstellen zu anderen Betriebssystemen können jederzeit problemlos nachgerüstet werden. Jede Anwendung läuft, genau wie die Subsystem-Server, in einem separaten Adreßraum. Sie müssen deshalb über spezielle Kanäle miteinander kommunizieren, die als 'Local Procedure Calls' (LPC) bezeichnet werden.. Die Abarbeitung der LPC's kann asynchron erfolgen. Dies ist insbesondere bei Ein-/Ausgabeoperation von Vorteil, da die Anwendung in ihrer Ausführung nicht unterbrochen wird und erst dann eine Nachricht erhält, wenn die angeforderten Daten eingetroffen sind. Aus Entwicklersicht spielen die Kernel-Funktionen der NT-Executive keine große Rolle, da sie in Win32-Programmen nur in Ausnahmefällen aufgerufen werden. Auch existieren wenig Informationen zu den Kernel-Funktionen und der Entwickler wird für die Entwicklung von Software-Systemen auf die Win32-API verwiesen. Die zusätzliche Abstraktionsebene durch die Win32-API und deren Trennung vom Kernel hat den Vorteil, Kernel-Funktionen (Systemdienste) anzupassen oder zu erweitern, ohne daß die laufenden Software-Systeme von Änderungen der Datentypen, Parameter etc. bei Kernel-Funktionen direkt betroffen sind. Somit legt sich das Win32-Subsystem, als zusätzliche Schale, zwischen den Entwickler und die tiefer liegenden Kernel-Funktionen.

Das Win32-Subsystem ist der Mittelpunkt des NT-Systems. Es bestimmt als Instanz, die das geräteunabhängige grafische Ausgabesystem GDI (Graphical Device Interface) beherbergt, maßgeblich das 'Look-and-Feel' des gesamten Betriebssystems. Alle anderen Subsysteme erledigen die Ausgabe von Daten auf Bildschirm und Drucker über das Win32-Subsystem und greifen somit ebenfalls auf dessen Dienste zurück. Man unterscheidet bei NT zwischen zwei Subsystemen: den sogenannten 'environment subsystems', wie z.B. das Win32-Subsystem, und den 'integral subsystems'. Erstere stellen ein definiertes Betriebssystem-API zur Verfügung, während letztere anderweitige Dienste bereitstellen, z.B. wäre hier das Sicherheits-Subsystem einzuordnen. Das Win32-Subsystem ist ein, im User-Mode ablaufendes Programm (Server) und stellt allen anderen Programmen (Clients) einen umfassenden Satz von Diensten, nach dem Client/Server-Prinzip, zur Verfügung. Windows NT macht beim Aufbau sowohl Anleihen beim Schichtenmodell als auch beim Client/Server-Modell. Während der Betriebssystemkern im Wesentlichen aus mehreren, übereinanderliegenden Schichten aufgebaut ist, wird für die darüberliegenden Aufgaben das Client/Server-Prinzip verwendet. Die Dienste des Betriebssystems werden dabei in einzelne Server aufgeteilt, die im User-Mode und im eigenen Adreßraum laufen. Dies bietet einige große Vorteile: So kann ein Server ausfallen und wieder neu gestartet werden, ohne das der Rest des Systems oder die laufenden Anwendungen in Mitleidenschaft gezogen werden. Wenn der Betriebssystemkern fehlerfrei läuft, kann das System praktisch nicht mehr blockieren. Außerdem können einzelne Server, auch im Netzwerk verteilt, auf anderen Rechnern liegen. Windows NT stellt hierfür die Remote Procedure Calls (RPC) zur Verfügung. Da NT auch multiprozessorfähig ist, kann jeder Server auf einem anderen Rechnersystem ausgeführt werden. Das Win32-Subsystem enthält folgenden Bestandteile:

- User-Modul (Window-Manager),
- GDI-Modul und –Treiber (Grafik-Erzeugung und –Ausgabe),
- Kernen-Modul (Betriebssystemdienste (z.B. Dateiverwaltung, I/O) und Basisdienste (z.B. Speicher- und Taskverwaltung),
- Schnittstellen zu anderen Subsystemen wie OS/2 oder Posix,
- Console API (Textmodus-Ausgabe).

Auch sorgt das Win32-Subsystem dafür, daß alle Sicherheitsregeln und -prüfungen, die den Kernel absichern, strikt eingehalten werden. Funktionen des Win32-Subsystems liegen in zwei Formen vor:

entweder sie sind in einer Win32-Subsystem-DLL implementiert, die in den Adreßraum eines Software-Systems abgebildet und aufgerufen wird oder sie finden sich auf dem Win32-Subsystem-Server. Hierbei muß dann das Software-System über den LPC-Mechanismus mit dem Server kommunizieren, wobei aber auch hier System-Dll's, mit entsprechenden Schnittstellen-Funktionen, eigenhändig die LPC-Kommunikation regeln und den Entwickler von der Arbeit mit dem LPC-Protokoll entlasten. Somit hat sich an der Oberfläche für den Entwickler, obwohl die Implementation im Vergleich zu Windows 3.1 eine völlig andere ist, nichts Wesentliches geändert. Damit sind auch die Unterschiede zu Windows 9x, für den Entwickler, nur in seltenen Fällen von Bedeutung, da die Win32-API eine gemeinsame Programmierschnittstelle für beide Betriebssysteme definiert und wesentliche Unterschiede nur den internen Aufbau, u.a. den System-Kernel, der Systeme betreffen.

2.2.3.3 Identifikation und Klassifikation Windows-spezifischer Portierungsprobleme

Die grundlegenden Architekturunterschiede des Portierungsfelds, welche im vorangegangenen Abschnitt betrachtet wurden, ergeben für die jeweilige Programmierschnittstelle – Windows 3.1: Win16-API und Windows 9x, Windows NT: Win32-API – gewisse Unterschiede, in Bezug auf die jeweilige Nutzung und erfordern gewisse Anpassungen der Quellen, sobald eine Portierung innerhalb des betrachteten Portierungsfelds, als Zielstellung, vorgegeben wird. Die notwendigen Änderungen an den Quellen lassen sich nach [Lauer 93] in folgende Problembereiche unterteilen:

- die Benutzung eines virtuellen, linearen 32-Bit-Adressraumes,
- die strikte Separation aller Prozesse bzw. ihrer Adreßräume voneinander,
- globale Änderungen am Modell für Benutzereingaben über Tastatur und Maus,
- erweitertes Koordinatensystem und weitere Änderungen im GDI,
- der Wegfall MS-DOS- und 80x86-spezifischer Aufrufe,
- die Benutzung undokumentierter Win16-Eigenschaften,
- aus den vorangegangenen Problembereichen resultierende Probleme, bei der Nutzung von Dll's,

Der größte Teil (ca. 95%) aller Portierungsprobleme läßt sich einer dieser Kategorien zuordnen. So lassen sich z.B. einzelne, geringfügige Änderungen bei den Parametern oder der Funktionsweise von Win16-Funktionen unter Win32, keiner grundlegenden Problemkategorie zuordnen. Ein Beispiel hierfür wären Threads, die aufgrund des preemptiven Multitaskings gewisse Anpassungen notwendig machen, die sich ebenfalls schlecht einer bestimmten Kategorie zuordnen lassen. Aus diesem Grund werden alle identifizierten Portierungsprobleme im betrachteten Portierungsfeld, zentral in einer Problembibliothek verwaltet und bei der späteren Portierungsanalyse, falls möglich, einem der Problembereiche zugeordnet (siehe Anhang B).

Linearer 32-Bit-Adressraum

32-Bit-Software-Systeme laufen, wie bereits an verschiedenen Stellen angesprochen, im Vergleich zu 16-Bit-Anwendungen, in einem virtuellen, linearen Adreßraum. Windows 9x benutzt, wie Windows NT, den virtuellen Speicher nach dem Prinzip des Demand Paging, der auf einem flachen, linearen Adreßraum beruht. Auf diesen Adreßraum wird mit 32-Bit-breiten Adressen zugegriffen. Demand Paging bezeichnet hierbei eine Methode, bei der Code und Daten seitenweise aus dem physischen Speicher, in eine Datei auf der Festplatte ausgelagert werden. Sobald ein Prozeß den Code und die Daten wiederbenötigt, werden die Seiten in den physischen Speicher verschoben. Jeder Prozeß besitzt

einen eigenen, virtuellen Adreßraum von 4 GB. Die oberen 2 GB dieses Adreßraums stehen allen Anwendungen gemeinsam zur Verfügung, die unteren 2 GB sind für die jeweilige Anwendungen reserviert. Dieser virtuelle Adreßraum wird in gleich große Blöcke aufgeteilt, die sogenannten Speicherseiten. Die Speicherverwaltung bildet die virtuellen Adressen aus dem Adreßraum eines Prozesses, auf die physisch vorhandenen Speicherseiten des Computers ab und verbirgt auf diese Weise die physische Organisation. Dadurch ist sichergestellt, daß ein Thread auf den Speicher seines eigenen Prozesses zugreifen kann, nicht aber auf den Speicher eines anderen Prozesses. Aus diesem Grund ist die Organisation des virtuellen Speichers, aus der Sicht eines Threads, wesentlich einfacher, als die tatsächliche Organisation der Seiten im physischen Speicher.

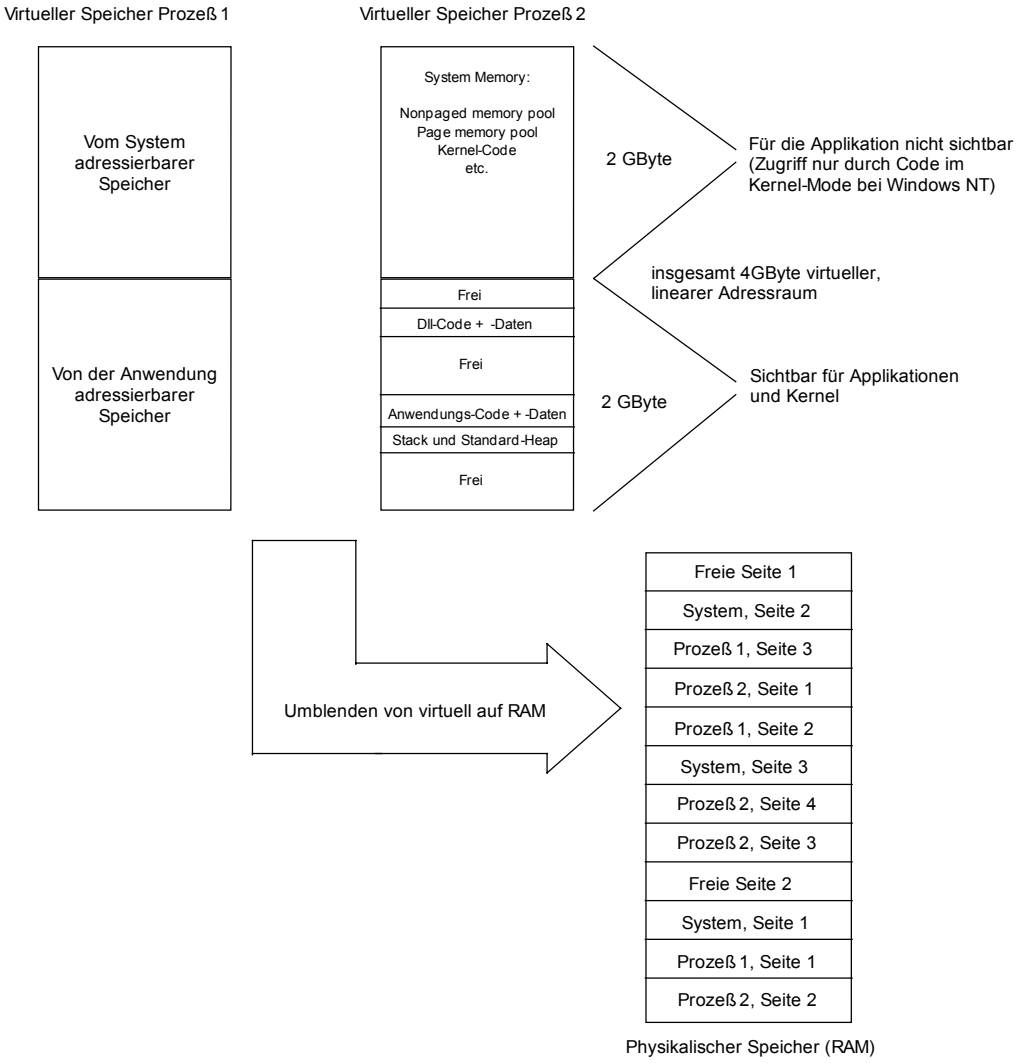


Abbildung 9: Adreßraum einer Win32-Anwendung und Verwaltung von Speicherseiten

Um 16-Bit-Betriebssysteme, wie Windows 3.1, zu unterstützen, benutzen Intel-Prozessoren einen sogenannten Segment-Mechanismus. Der frei verfügbare Speicher ist hierbei in 64 KB große Segmente aufgeteilt und wird mit Hilfe von zwei 16-Bit-Adressen angesprochen: der Segmentadresse und der Offsetadresse, für den Speicher innerhalb dieses Segments. Aufgrund der Systemarchitektur von Prozessoren der Intel-Baureihe <80386 können hierbei insgesamt 16 MB Speicher adressiert werden, obwohl theoretisch 4 GB möglich wären. Das Betriebssystem und die Anwendungen können Speicher außerhalb eines Segments, nur unter Leistungseinbußen ansprechen. Windows 9x und

Windows NT nutzen die 32-Bit-Fähigkeiten des 80386-Prozessors und seiner Nachfolger, deren Architektur ein flaches, lineares Speichermodell für die Funktionen eines 32-Bit-Betriebssystems und seiner 32-Bit-Anwendungen unterstützt. Das Modell der linearen Speicheradressierung vereinfacht die Entwicklung von Anwendungen und beseitigt die Leistungseinbußen durch segmentierten Speicher. Durch dieses lineare Speichermodell ermöglicht Windows jeder 32-Bit-Komponente des Betriebssystems, den Zugriff auf 4 Gigabyte adressierbaren Speicher, wobei jede 32-Bit-Anwendung bis zu 2 Gigabyte 'privaten' Speicher anfordern kann.

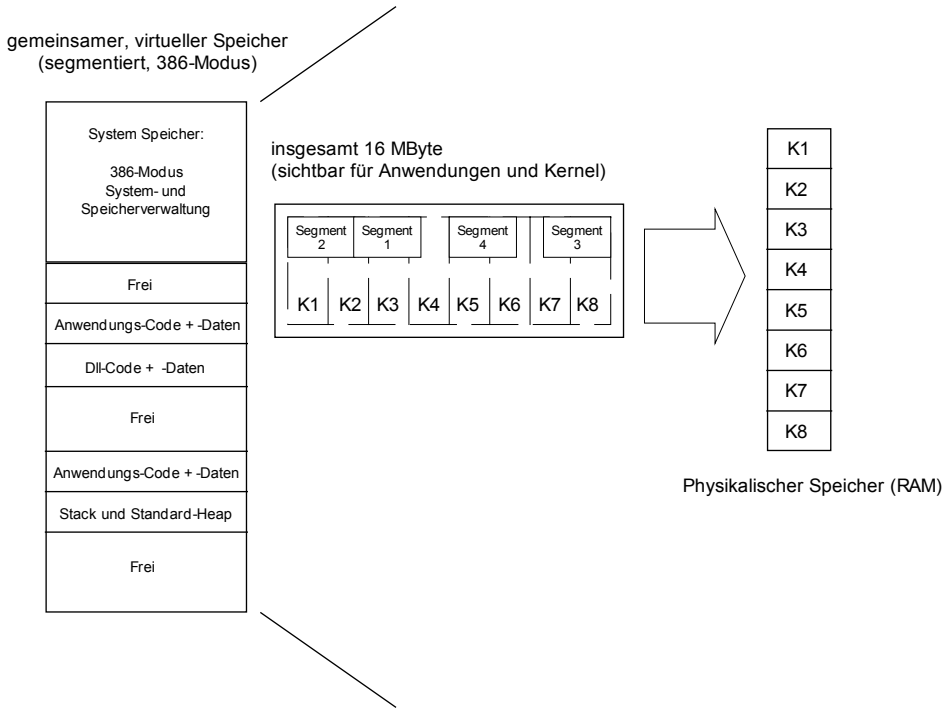


Abbildung 10: Adreßraum einer Win16-Anwendung und Verwaltung von Speicherseiten

Die lineare Speicherarchitektur führt u.a. dazu, daß einige der wichtigsten Datentypen von 16 auf 32 Bit erweitert sind (siehe Anhang C – Wichtige Datentypen im Vergleich) und somit Unterschiede in den jeweiligen API's erzwingen. Dies betrifft vor allem die Benutzung folgender Datentypen:

- vorzeichenbehaftete und vorzeichenlose Integer-Werte,
- NEAR-Zeiger,
- fast alle Handle-Typen,
- sowie die polymorphen Typen: LPARAM und WPARAM.

Unter Windows 3.1 reicht ein vorzeichenloser Integer-Wert aus, um die einzelnen 16-Bit-Segmente (mit jeweils 64 KB) byteweise zu adressieren. Größere Speicherbereiche werden hierbei durch eine spezielle Segmentarithmetik (Segment und Offset) adressiert. Mit dieser Adressierungsmethode über 16-Bit-Integer-Werte ist es jedoch nur schwer möglich, einen 4 GB großen Adreßraum effizient zu adressieren. Somit werden in 32-Bit-Windows-Systemen alle Integer-Werte von den entsprechenden Compilern mit einer Breite von 32 Bit angelegt.

Die gleiche Aussage kann für NEAR-Zeiger gelten, welche in 16-Bit-Windows-Systemen dazu verwendet werden, den 16 Bit breiten Offset, in ein festgelegtes, maximal 64 KB großes, Speichersegment zu realisieren. Unter der 32-Bit-Plattform von Windows gehen die vermeindlichen

'Offsets' direkt in einen virtuellen und linearen Adreßraum von 4 GB Länge. Um diesen zu adressieren wird ein 32 Bit breiter Datentyp benötigt. Ein FAR-Zeiger – auch unter Win16, 32 Bit breit – aus einer 16-Bit-Umgebung, verwandelt sich in einer 32-Bit-Umgebung in einen einfachen NEAR-Zeiger – allerdings mit einer Breite von 32 Bit –, sodaß sowohl der Größenunterschied, als auch konzeptionelle Diskrepanzen zwischen einem Win16-NEAR-Zeiger und einem Win16-FAR-Zeiger dort aufgehoben sind, d.h. sizeof(NEAR)==sizeof(FAR). Das völlige Auflösen von Segmentwerten und die Vergrößerung des Offsets auf 32 Bit führen auch dazu, daß fast alle nicht standardisierten Zeigermanipulationen unmöglich werden. Abbildung 11 zeigt die verschiedenen Win16-Zeigertypen und deren Größen im Vergleich zu Win32-Zeigern. Diese entsprächen einem 48-Bit-Zeiger, mit 16-Bit für den Selektor, plus 32-Bit-Offset. Ein 32-Bit-Compiler für Windows-Systeme, kennt jedoch keinen eingebauten Datentyp dieses Formats. Dieser wird im Win32-API weder benutzt noch benötigt, da Segmente hier vom Betriebssystem einmalig initialisiert und vom Programmierer weder benutzt, noch geändert werden können.

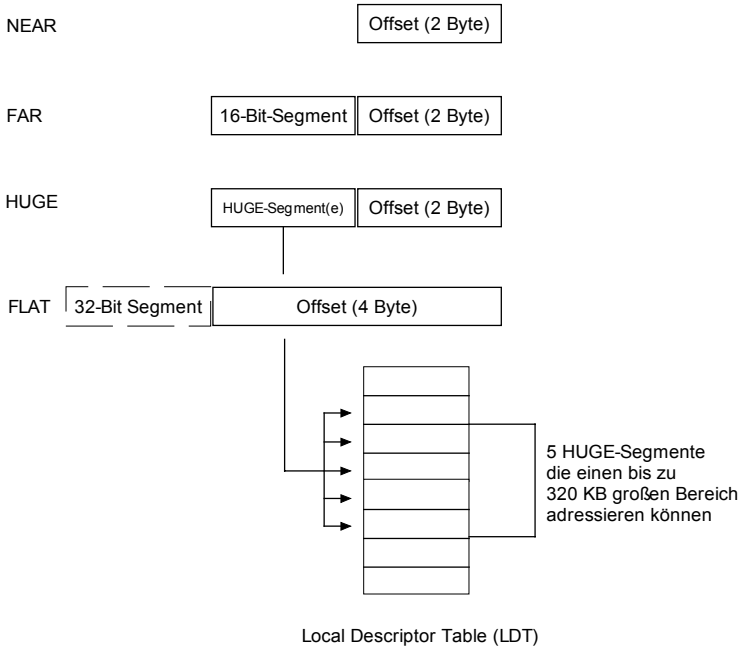


Abbildung 11: Win16-Zeigertypen und deren Größen im Vergleich zu Win32-Zeigern.

Ein Handle-Typ, wie z.B. HWND, beschreibt als Parameter in zahlreichen Windows-Funktionen die Eigenschaften eines bestimmten Fensters. Für das Betriebssystem ist dieser Handle ein NEAR-Zeiger auf eine interne Datenstruktur und folglich in 32-Bit-Systemen ebenfalls verbreitert. Da ein NEAR-Zeiger in 16-Bit-Systemen immer aus einem Segment und einem Offset besteht, zeigt dieser in einem solchen Fall in die Datensegmente, der jeweilig beteiligten Systemkomponenten (User.exe, GDI.exe oder Kernelx86.exe) und ist z.B. bei HWND ein NEAR-Zeiger in ein User-Datensegment. Da diese Systemkomponenten in einem 32-Bit-System nun in einem eigenen virtuellen Speicherbereich, von insgesamt 4 GB laufen, ist es notwendig, die Handle-Typen dementsprechend zu verbreitern.

Die Verbreiterung der Datentypen hat insbesondere Konsequenzen im Bereich der sogenannten Callback-Prozeduren (z.B. WNDPROC, DLGPROC). Dies sind Routinen, die nicht von anderen Stellen des Programms, sondern von Windows aus, aufgerufen werden und sich außerhalb des programmeigenen Adreßraums befinden. Hierbei ist insbesondere der dritte Parameter entscheidend,

dieser wächst von 16 Bit (WORD) auf 32 Bit (WPARAM). WPARAM ist in beiden Systemen, 16 und 32 Bit, als vorzeichenloser Integer-Wert definiert und zusammen mit LPARAM – der jedoch keiner Änderungen bedarf, da jeweils als 4-Byte breiter Integer-Typ (LONG) definiert – für den Nachrichtenaustausch zuständig. Da viele Nachrichten in den beiden Parametern mehrere Informationen verpacken und diese unter einem 32-Bit-Windows-System mehr Platz brauchen, mußte das Format der betreffenden Nachrichten angepaßt werden. Das führt dazu, daß die Informationen unter Windows 3.1 anders gepackt sind, als unter Windows 9x oder Windows NT. Somit ist eine portable und klare Formulierung von Quellen schwieriger, da je nach Zielsystem, unterschiedliche Zugriffe auf die Nachrichtenparameter notwendig sind.

Getrennte Adreßräume

Die vollständige Trennung der Adreßräume unter Win32 sorgt dafür, daß jede Anwendung ihren eigenen privaten Adreßraum erhält. Auf diesen Adreßraum haben Prozesse anderer Anwendungen keinen Zugriff, weder lesend noch schreibend. Damit sind Speicherzugriffe auf die Adreßräume anderer Prozesse – und somit die gemeinsame Nutzung von Speicherbereichen, wie unter Win16 – nicht mehr möglich. Der Datenaustausch zwischen den Prozessen wird über verschiedene IPC-Mechanismen (d.h. nur über vom System definierte Kommunikationswege bzw. Interprozeß-Kommunikation) realisiert. Dazu gehören Mechanismen wie z.B. 'Local Procedure Call' (LPC, eine für den lokalen Betrieb optimierte Variante des 'Remote Procedure Call' (RPC)), 'Named Pipes', Mechanismen wie 'Dynamic Data Exchange' (DDE) – Datenaustausch erfolgt über gemeinsame Speicherbereiche – oder OLE (siehe Abbildung 12).

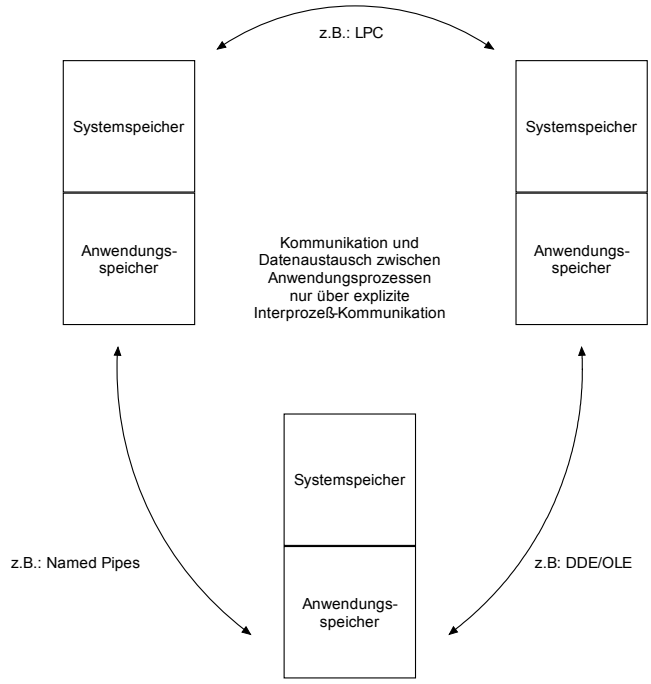


Abbildung 12: Getrennte Adreßräume unter Win32 und Interprozeß-Kommunikation

Die Separation der Adreßräume erzeugt einen Problembereich, der im wesentlichen folgende Portierungsprobleme aufwirft:

- Zugriff auf vorhergehende Instanzen einer Anwendung,

- Speicherverwaltung: lokaler und globaler Heap,
- Abweichungen beim Windows Management: DDE und Datenaustausch.

Verschiedene Anwendungen benötigen Informationen über bereits laufende Instanzen, um entweder sicherzustellen, daß nur eine Instanz geladen werden kann (z.B. Task-Manager) oder um Informationen aus schon laufenden Instanzen zu lesen, bzw. an diese zu übertragen. Die völlige Separation der Adreßräume führt dazu, daß sich jeder Anwendungsprozeß als grundsätzlich erste gestartete Instanz eines Programms identifiziert und nicht, wie unter Windows 3.x, als weitere Instanz einer Anwendung registriert wird. So ist zum Beispiel die Praxis von einigen Grafikprogrammen, häufig benötigte Grafikobjekte nur einmal anzulegen und in mehreren Instanzen zu benutzen, unter Win32 unzulässig.

Global definierte Daten werden unter Win32 anders verwaltet als im 16-Bit-Windows. So existieren unter Win16 zwei Mechanismen zur dynamischen Anforderung von Speicherbereichen: der lokale Heap – mit einem theoretischen Maximum von 64 KB, einem Zeigertyp Offset (2 Byte) und anwendungsweiter Verfügbarkeit –, für kleinere Objekte und der globale Heap – mit einem, je nach Betriebsmodus, maximal allokierbarem Speicher von mehreren Megabyte (im Rahmen der physischen Verfügbarkeit), einem Zeigertyp 'Selektor und Offset' (4 Byte) und systemweiter Nutzung der Speicherbereiche –, der in erster Linie größere Bereiche allokiert oder aber zum Austausch von Daten zwischen Anwendungen benutzt wird.

Die Verfügbarkeit des lokalen und globalen Heaps beschränkt sich unter Win32 – wie unter Win16 der lokale Heap – auf den Speicherbereich der Anwendung (hier aber bis zu maximal 2 GB über einen Offset (4 Byte) adressierbar). Somit befindet sich die Speicherverwaltung von Win32 konzeptionell näher am lokalen Heap von Windows 3.x, als am globalen Heap und Portierungsprobleme entstehen hierbei, u.a. bei Zeigerzuweisungen an verbreiterte Datentypen, verschicken von NEAR-Zeigern in privaten Nachrichten oder Datensegment-Modifikationen, im Sinne des Konzepts mehrerer lokaler Heaps. Auch ist das sogenannte 'Locking' von allokierten, globalen und verschiebbaren Speicherbereichen über korrespondierende Lock-Aufrufe, unter Win32 notwendig – Ausnahme: 'Fixed Memory', hier werden keine Speicher-Handles sondern direkt Zeiger zurückgeliefert –, um an gültige Adressen in Form von Speicher-Handles zu kommen. Hin- und Hersenden von globalen Speicherbereichen durch z.B. globale Speicher-Handles – notwendig unter Windows 3.x für lokale und als 'moveable' gekennzeichnete Speicherallokationen und zwingend für global genutzte Speicherbereiche – oder noch drastischer, von FAR-Zeigern, zwischen Anwendungen, um Datenaustausch zu realisieren, ist unter Win32 nicht mehr möglich. Damit stellen alle 'shared Memory'-Ansätze, die auf dem Austausch globaler Handles, durch private Nachrichten beruhen, potentielle Portierungsprobleme dar.

Bedient man sich beim Datenaustausch des standardisierten DDE-Protokolls, bzw. der DDE Managment Library (DDEML), als API rund um das DDE-Protokoll, übersetzt das Win32-Subsystem die Handles der Nachrichten für den Informationsaustausch von einem virtuellen Adreßraum in den anderen. Die DDEML selbst, ist, bis auf kleinere Unterschiede zwischen den betrachteten Windows-Systemen, völlig portabel. Jedoch hat sich das Format, der in den beiden DDE-Nachrichten transportierten Informationen, für Win32 geändert. So erzeugen Anwendungen, die statt der DDEML, das Nachrichtenprotokoll selbst benutzen, ebenfalls Probleme bei der Portierung.

Benutzereingabe: Maus und Tastatur

Eingabenachrichten werden unter Windows 3.x über Hardware-Interrupts und im Zuge des Interrupts, über generierte Input-Event's (z.B. WM_KEYDOWN) behandelt. Diese werden in einer systemweiten Nachrichtenschlange hinterlegt und von den jeweiligen Anwendungen, zu denen die Eingabenachricht gehört, ausgelesen und in die private Nachrichtenschlange kopiert. Unter Win32 wird das Konzept des lokalen Eingabemodus benutzt. In diesem Eingabemodell werden die Eingabenachrichten im Moment ihrer Erzeugung bereits in die private Nachrichtenschlange des Anwendungsprozesses kopiert. Das neue Eingabemodell hat den wichtigen Vorteil, daß Anwendungen, die nicht regelmäßig ihre Nachrichtenschleife bedienen, nicht mehr zur scheinbaren Blockade des gesamten Systems führen (Sanduhr-Effekt unter Win16).

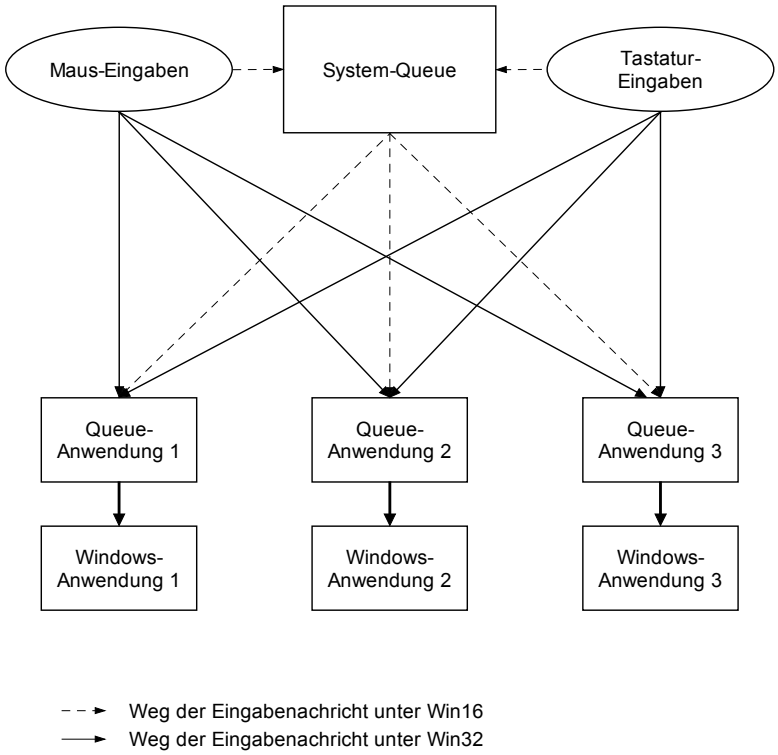


Abbildung 13: Eingabenachrichten unter Win16 und Win32

Portierungsprobleme, die sich aufgrund des veränderten Eingabemodells (lokales Eingabemodell) ergeben, finden sich im Wesentlichen bei den Funktionen, die sich mit der Verwaltung und Veränderung des Eingabestatus befassen. Dies betrifft insbesondere die Verwaltung des Fokus, bzw. der aktiven Fenster sowie das systemweite Abfangen aller Mausnachrichten.

Jeder Anwendungsprozeß besitzt eine eigene Nachrichtenschlange für alle, an ihn bestimmten – abhängig vom aktuellen Eingabefokus, welcher auf ein aktives Fensters eines Anwendungsprozesses gerichtet ist –, eingehenden Nachrichten. Unter Win16 ist es jederzeit möglich, das Handle des Fokus-Windows zu erfragen (GetFocus()). Unter Win32 nur dann, wenn das Fenster auch zum aufrufenden Prozeß der Anwendung gehört. Auch kann unter Win16 jeder beliebigen Anwendung, sobald ihr Handle bekannt ist, der Fokus zugewiesen werden (SetFocus()). Dies ist unter Win32 nur noch für die, vom Anwendungsprozeß eigens erzeugten, Fenster möglich. Auch das Erfragen oder Setzen von

aktiven Fenstern bezieht sich unter Win32 nur noch auf den lokalen Eingabestatus von Anwendungsprozessen.

Zusätzlich zum standardmäßigen Abfangen von Mausnachrichten über das jeweils aktive Fenster eines Anwendungsprozesses, besteht für Fenster unter Win16 jederzeit die Möglichkeit sich dem Eingabestrom der Mausnachrichten zu bemächtigen (SetCapture()). Wie der Fokus ist der 'mouse capture', eine Art Pseudo-Ressource, in deren Besitz immer nur ein einziges Fenster kommen kann. Unter Win32 kann nur das Fenster – oder eines seiner Child-Windows – den 'mouse capture' erhalten, das sich gerade im Vordergrund befindet, bzw. aktiv ist. Für alle nicht im Vordergrund befindlichen Fenster, ist ein Versuch, die Mausnachrichten abzufangen, unter Win32, wirkungslos und führt zu Portierungsproblemen. Das systemweit eigenmächtige Abfangen von Mausnachrichten verletzt das Prinzip von Win32, jederzeitige Bedienbarkeit aller gegenwärtig laufenden Anwendungen zu gewährleisten. Dazu gehört auch die Nutzung modaler Dialogboxen.

Änderungen am GDI

Das GDI unter Win32 ist nicht nur komplett neu implementiert -– in C++, statt wie unter Win16 in C – und somit objektiv verbessert worden, sondern es ist auch mit seinen Koordinaten gewachsen. Die im Zusammenhang mit dem GDI auftretenden Problemfälle, betreffen im Wesentlichen die verbreiterten GDI-Koordinaten und das Verpacken von Koordinaten.

Die Funktionen für die Darstellung von Text und Grafik, die über die grafische Geräteschnittstelle zur Verfügung gestellt werden, benutzen unter Windows 3.x, 16 Bit breite GDI-Koordinaten, für die Darstellung von Informationen auf dem jeweiligen Ausgabemedium. Diese wurden unter Win32 auf 32 Bit verbreitert. Portierungsprobleme ergeben sich hierbei in den Strukturen, die mit Koordinaten bzw. Größenangaben umgehen (POINT, RECT etc.). Während die meisten Strukturkomponenten, die Koordinaten beschreiben, konsistent auf 32 Bit erweitert wurden und eine systemkonforme Behandlung von GDI-Koordinaten sicherstellen, werden eine Vielzahl von Windows-Nachrichten mit gepacktem Win16-Format versorgt. Dieses Format liefert Koordinaten- und Größenangaben verpackt im LPARAM-Nachrichtenparameter von Windows-Funktionen. Hierbei liefert LOWORD (lparam) die x-Komponente und HIWORD die y-Komponente. Auch liefern diverse Win16-GDI-Funktionen ihre Resultate so codiert zurück. Unter Win16 werden die x/y-Paare, als zwei getrennte Integer-Werte, in 4 Byte breiten Integer-Datentypen (DWORD oder LONG) verpackt, an aufrufende Funktionen zurückgeliefert. Da ein DWORD-Datentyp unter Win32 aber genauso breit ist wie ein Integer-Datentyp und die beiden 32-Bit-Werte für die Koordinaten nicht mehr in einem DWORD untergebracht werden können, funktioniert das Packen der Koordinaten hier nicht mehr. Somit können alle Stellen, an denen ein Übergang von gepackten 16-Bit-Koordinaten auf das neue 32-Bit-Format stattfindet, mit Portierungsproblemen behaftet sein.

DOS- und CPU-Spezifisches

Der fast vollständige Verzicht auf MS-DOS- oder 80x86-spezifische Aufrufe unter Win32 eröffnet ein weiteres Feld an Portierungsproblemen. Hierbei ist Windows NT, bedingt durch seine Architektur, insbesondere durch seine Hardware-Abstraktionsschicht, noch strikter, was den direkten Zugriff auf Hardwarekomponenten angeht, als Windows 9x. Wird unter Win32 versucht, durch eine einheitliche und konsistente Betriebssystem-API, alle Ressourcen des Computersystems nutzbar zu machen, ist es unter Win16 notwendig, über eine Vielzahl von direkten BIOS-Aufrufen (MS-DOS-

Funktionsinterrupt), Peripheriegeräte des Computersystems zu steuern (bzw. über Funktionen aus der Laufzeitbibliothek von Compilerherstellern). Für Software-Systeme, die nach Windows NT portiert werden sollen, kann folgende Aussage getroffen werden: Jeder direkte Hardware-Zugriff außerhalb von Gerätetreibern ist hier nicht mehr möglich und erfordert entsprechende Anpassungen. Unter Windows 9x ist der direkte Hardware-Zugriff zwar noch möglich, jedoch sollten im Sinne der Portabilität von windows-basierten Software-Systemen direkte Hardware-Zugriffe – und somit systemabhängig – vermieden werden. Weitere Problempunkte in diesem Bereich betreffen neben den Hardware-Zugriffen auch das erweiterte Dateisystem und das Konzept der zentralen Registrierungsdatei unter Win32. Insgesamt ergeben sich in erster Linie folgende Problembereiche:

- direkte Hardwarezugriffe außerhalb von Gerätetreibern,
- Speicherung und Bearbeitung von Dateinamen,
- Behandlung von Intitialisierungsdaten über das Konzept der zentralen Registrierungsdatenbank.

Bei der Betrachtung von Portierungsproblemen, im Bereich von Hardwarezugriffen, spielt die unterschiedliche Architektur der beiden Win32-Systeme, Windows NT und Windows 9x, eine wesentliche Rolle. Ist es unter Windows 9x noch möglich, aus einem Anwendungsprozeß heraus, direkt auf Hardware-Komponenten zuzugreifen – da Windows 9x als 'Update' zu Windows 3.x zu betrachten ist und direkte Hardware-Zugriffe aus diesem Grunde hier noch akzeptiert werden –, so ist diese Möglichkeit unter Windows NT – durch den zusätzlich eingeführten 'Hardware-Abstraction-Layer' – nur noch, mit speziellen Rechten versehenen, Hardware-Treibern gegeben. Anwendungen laufen hier im sogenannten User-Mode und haben keinerlei Rechte auf die zugrundeliegende Hardware direkt zuzugreifen. Somit gelten die Aussagen über diesen Problembereich in erster Linie für Software-Systeme, deren Portierungsziel Windows NT darstellt. Aufgrund von vorteilhaften Portabilitätsmerkmalen können diese Aussagen aber auch für Anwendungen gelten, die nach Windows 9x portiert werden sollen. Der Problembereich der Hardware-Zugriffe läßt sich in zwei Teilbereiche klassifizieren: Zugriffe über Funktionsinterrupts, für die in der Win32-API eine kompatible Funktion bereitstellt wird und solche, für die in der Win32-API keine portable Funktion implementiert werden kann. Womit ansonsten eines der Ziele von Windows NT, die weitgehende Quelltext-Portabilität zwischen den unterstützten Plattformen, nicht mehr gegeben wäre (z.B. direkte Zugriffe auf bestimmte I/O-Ports). So steht z.B. im Bereich der Datei-I/O ein komplett kompatibles API zur Verfügung und auch der Bereich der MS-DOS-Funktionen, über den Interrupt 21, wird größtenteils über Win32-Funktionen abgedeckt. Für die zweite Gruppe der Hardware-Zugriffe werden unter Windows-NT eigene Gerätetreiber benötigt, welche im Kernel-Mode auf die einzelnen Hardwarekomponenten zugreifen.

Eng mit der Hardware und dem Betriebssystem verbunden, ist die Speicherung und Bearbeitung von Dateinamen. Unter Windows NT werden verschiedene Dateisysteme unterstützt (NTFS, HPFS etc.), die sich von dem Dateisystem unter Windows 3.1 (FAT) unterscheiden. Das Problem der unterschiedlichen Dateisysteme macht sich in den meisten Fällen dann bemerkbar, wenn man implizit Annahmen über die Länge und das Format von Dateinamen macht (80-Zeichen-Limit unter Windows 3.1, im 8.3-Format). Hierbei entstehen Probleme bei der nach Win32 portierten Version dieser Anwendung, sobald diese auf einem Win32-System, ohne FAT-Dateisystem, läuft und einen Dateinamen verwendet, der länger als die vorgegebene Größe von z.B. 80 Byte ist, bzw. das 8.3-Format nicht einhält. Weiterhin ist die unterschiedliche Semantik einzelner Zeichen sowie verschiedenartige Konventionen, in Bezug auf Dateinamen in den unterschiedlichen Dateisystemen,

zu berücksichtigen. Auch müssen alle Programmteile, die Pfad- oder Dateinamen zeichenweise analysieren, entsprechend den Namenkonventionen angepaßt werden.

Das Konzept der zentralen Registrierungsdatenbank führt zu weiteren Problemstellen bei der Portierung. So ist unter Windows 3.1 jede Anwendung und auch das Windows-System selbst, über eine Reihe von sogenannten Initialisierungsdateien, in denen Initialisierungs- und sonstige Informationen abgelegt werden, konfigurierbar. Das Konzept der Initialisierungsdateien führt jedoch dazu, daß eine Vielzahl solcher Dateien auf einem Rechnersystem entstehen und keine zentrale Instanz für die Registrierungs- und Verwaltungsdaten der Anwendungen und des Rechnersystems zuständig ist. Das Konzept von Win32-Systemen ist die zentrale Registrierungsdatenbank. In ihr werden alle Arten von benutzer- und anwendungsspezifischen Informationen sowie dem Datenaustausch der Anwendungen untereinander (z.B. OLE), als auch die gesamte Hard- und Software-Konfiguration eines Rechnersystems, festgehalten und verwaltet. Hierbei treten in erster Linie dann Probleme auf, wenn man direkt auf den Inhalt von INI-Dateien zugreift – nicht standardisierte Zugriffsmethoden, sprich direkte Dateioperationen – und mittels selbstdefinierter I/O-Funktionen auf deren Inhalt zugreift, anstatt auf die dafür vorgesehenen API-Funktionen zurückzugreifen. Dieses Vorgehen ist im Zusammenspiel mit der Win32-Registry nicht mehr möglich und man benötigt für den Umgang mit Systeminformationen, die hierfür vorgesehenen API-Funktionen.

Nichtdokumentierte Eigenschaften

Die Nutzung undokumentierter Eigenschaften eines Windows-Systems, in Software-Systemen, ist meist mit größeren Anpassungen bei der Portierung verbunden und verursacht somit Probleme. Hierbei unterscheidet man zwischen undokumentierten Eigenschaften in Win16-Systemen, welche unter Win32 offiziell vorhanden und entsprechend dokumentiert sind – z.B. Nachrichten wie WM_[GET/SET]HOTKEY oder Funktionen, wie keypd_event(), bzw. Funktionen aus Bibliotheken, wie Toolhelp.dll oder Stress.dll sowie diverse Konstanten etc. – oder man vertraut auf die inoffizielle Übernahme der benutzten Eigenschaften. Hierbei ist in erster Linie die Verwendung undokumentierter, bzw. interner Win16-Datenstrukturen, zu nennen, welche durch die Erweiterung aller grundlegenden Datentypen auf 32 Bit und die Trennung der Adreßbereiche, mit großer Wahrscheinlichkeit, kein Win32-Äquivalent aufweisen. Es können somit keine grundlegenden Aussagen darüber getroffen werden, welche Eigenschaften im einzelnen portierbar sind und wo Portierungsprobleme zu erwarten sind.

Programmierung von Dll's

Die Probleme in diesem Bereich ergeben sich aus der strikten Trennung der Adreßräume von Prozessen und der Verwendung von preemptivem Multitasking unter Win32. Jeder Prozeß muß benötigte Dll's in seinen privaten Adreßraum einblenden (Dll attaching), um deren Daten nutzen zu können. Diese Daten – globale und statische Variablen – befinden sich im Datenbereich der entsprechenden Dll und werden für jeden Prozeß, separat in dessen Adreßraum, neu angelegt (privat-global). Die Codebereiche einer Dll werden dagegen nur einmal in den physikalischen Speicher geladen und durch die Win32-Speicherverwaltung (Virtual Memory Manager) für die Prozesse über das sogenannte 'Paging' der CPU, in deren Adreßraum abgebildet. Von Portierungsproblemen ist deshalb nur der Umgang mit den Datenbereichen einer Dll betroffen. So wird für jeden Prozeß eine eigene, private Kopie einer Dll, in dessen Adreßraum geladen und initialisiert. Aus diesem Grund

können hierbei keine, für alle Prozesse, wichtigen Informationen (public-global) in den Dll's gehalten werden (siehe Abbildung 14).

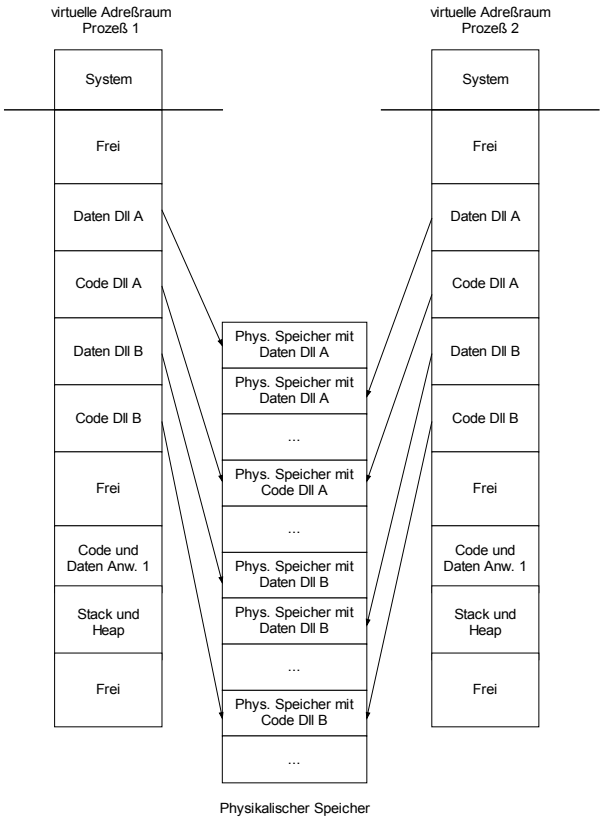


Abbildung 14: Dll-Speicherverwaltung unter Win32

Unter Win16 wird eine Dll einmalig in den Speicher geladen, ist danach systemweit sichtbar und jeder Prozeß kann auf deren Datensegment zurückgreifen bzw. dieses manipulieren (z.B. für dynamisch globale Speicherallokationen) (siehe Abbildung 15).

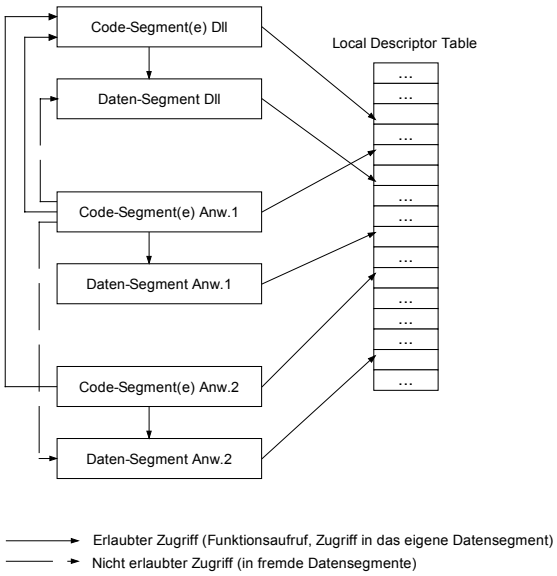


Abbildung 15: Dll-Speicherverwaltung unter Win16

Ein weiteres Problem betrifft die Ausführung von Dll-Aufrufen, bei verändertem Multitasking-Mechanismus. Werden unter Win16, aufgrund des kooperativen Multitaskings, Dll-Aufrufe jeweils komplett ausgeführt (atomic function call), so können unter Win32 während der Abarbeitung von Dll-Aufrufen, Prozeßumschaltungen vorkommen, die entsprechende Synchronisationsmechanismen erforderlich machen. Insbesondere Dll's, welche mehrere Anwendungs-Clients gleichzeitig bedienen und zentrale Datenstrukturen manipulieren, müssen den korrekten Zugriff geeignet synchronisieren.

Aus den Unterschieden zwischen den betrachteten Windows-Systemen ergeben sich bei der Portierung von Dll's, insbesondere Probleme bei:

- der Nutzung globaler und dynamischer Speicherbereiche,
- deren Initialisierung und Terminierung
- sowie der Verwendung globaler Windows-Klassen.

Eine Reihe von Dll's gehen über die Zielstellung der effizienten und zentralen Bereitstellung von Funktionen hinaus und benötigen für die Verwaltung von prozeßspezifischen, privat-globalen Datenblöcken, zusätzliche Datenbereiche, die nur einmal angelegt und global sichtbar sein müssen. Unter Win32 sind Dll's nach dem Laden nicht global bekannt, sondern müssen sich jeweils im Adreßraum des Prozesses, in den diese geladen wurden, bewegen. Statische Datenbereiche der Dll – globale- und static-Variablen – werden hierbei für jeden Prozeß neu angelegt Sobald diese benutzt werden, um einer Dll die Koordination und Verwaltung mehrerer Prozesse zu ermöglichen, führt dies zu einer Reihe von Portierungsproblemen. Dynamische Speicheranforderungen werden unter Win32 immer privat, im Adreßraum des verursachenden Prozesses, ausgeführt. Konnte man unter Win16, durch die grundsätzlich globale Allokation, von allen Anwendungen aus, gemeinsam auf die dynamisch angeforderten Speicherbereiche zugreifen, so ist dieses Vorgehen unter Win32 nicht mehr möglich. Auch die unter Win16 bestehende Möglichkeit, die Lebensdauer von globalen Speicherallokationen selbst zu kontrollieren, ist unter Win32 wirkungslos – GMEM_(DDE)SHARE-Flags – und verursacht, wie die Annahme, daß dynamische Speicheranforderungen innerhalb einer Dll grundsätzlich global sind, weitere Probleme in Bezug auf die Portierung. Auch verursachen Allokationen, die in den lokalen Heap einer Win16-Dll gelenkt werden, um Daten zwischen mehreren Client-Anwendungen auszutauschen, gewisse Anpassungsprobleme. Die Speicherverwaltung unter Win32 trennt nicht mehr zwischen lokalem und globalem Heap, und Dll's müssen sich im Adreßraum ihrer Client-Anwendung mit Speicher versorgen. Dieses Problem entsteht dadurch, daß Dll's, aber auch allgemein alle Anwendungsprozesse, unter Win32 kein eigenes Datensegment mehr besitzen – statt dessen ein Datenbereich im Adreßraum des aufrufenden Anwendungsproszesses –, welches unter Win16 immer mit einem eigenen, lokalen Heap der entsprechenden Dll verknüpft ist.

Portierungsprobleme ergeben sich auch aus der Sicht der Initialisierung und Terminierung von Dll's. So ist unter Win16 die Initialisierungsfunktion LibMain(), als Haupteintrittspunkt einer Dll, für das Entsperren des Datensegments und das Einrichten globaler Variablen – z.B. bei der Registrierung globaler Windows-Klassen – zuständig. Der Initialisierungscode wird nur für die erste Anwendung, die eine Dll benutzt, ausgeführt. und dann aufgerufen, wenn die Bibliothek zum ersten Mal geladen wird. Falls erforderlich können über eine Terminierungsfunktion, WEP(), eventuelle Aufräumarbeiten, beim Entladen der Dll aus dem Speicher vorgenommen werden. Unter Win32 existiert für die Initialisierung und Terminierung von Dll's eine gemeinsame Funktion, welche beliebig benannt werden kann, z.B. DllEntryAndExitPoint(). Hierbei werden im ersten Parameter die Instance-Handle

der Dll angegeben. Im zweiten Parameter werden mittels übergebener Konstanten Initialisierungs- oder Terminierungsroutinen, innerhalb der Funktion, ausgelöst. Je nachdem, welcher Grund zum Aufruf der Funktion führt. Hierbei können nicht nur einzelne Anwendungsprozesse bei der Dll an- und abgemeldet werden, sondern auch einzelne Threads. Die Portierung von Software-Systemen, welche mit einer Vielzahl von Dll's arbeiten, erfordern somit auch in diesem Bereich gewisse Anpassungen und ist mit Problemen behaftet.

Bei der Verwendung anwendungsglobaler Windows-Klassen, welche innerhalb einer Dll registriert sind und z.B. mittels der Funktion CreateWindow() genutzt werden, ergeben sich ebenfalls Portierungsprobleme, aufgrund des notwendigen, expliziten Ladens der Dll. Entweder explizit über die Funktion LoadLibrary() oder implizit über einen externen Lader beim Systemstart. Das zweite Verfahren über einen externen Lader, funktioniert unter Win32 nicht mehr in der gewünschten Form. Jeder Prozeß muß alle benötigten Dll's selbst laden, da hier, außer den System-Bibliotheken, keine global bekannten Dll's mehr existieren. Anwendungslokal registrierte Windows-Klassen, ohne CS_GLOBALCLASS-Flag, verursachen ebenfalls Portierungsprobleme, aufgrund der unterschiedlichen Verwendung der Instance-Handles.

2.2.4 Strategien und Werkzeuge zur Portierung von Win16 nach Win32

Jede Portierung findet innerhalb eines festgelegten Portierungsfelds statt. Der Ausgangspunkt hierbei ist ein funktionierendes Software-System, in einer bestimmten Systemumgebung. Die Zielsetzung besteht in der Portierung dieses Software-Systems in seine zuvor bestimmte Zielumgebung. Hierbei können unterschiedliche Strategien und Werkzeuge zur Zielerreichung genutzt werden. In diesem Kapitel sollen deshalb zunächst mögliche Ausgangspunkte und Zielsetzungen näher eingegrenzt und mögliche Strategien beleuchtet werden. Darüber hinaus werden Werkzeuge identifiziert, die den Portierungsprozeß unterstützen, jedoch nicht automatisieren.

2.2.4.1 Ausgangspunkt und Zielsetzung

Für ein Portierungsprojekt und deren Planung muß zunächst der Ausgangspunkt, d.h. die Beschaffenheit der Quellen des zu portierenden Software-Systems und die Systemumgebung ‚von der aus portiert werden soll, eindeutig bestimmt werden. Windows-basierte Software-Systeme können unter Win16 allgemein vier verschiedenen Varianten, bzw. deren Mischformen, zugeordnet werden:

- Software-Systeme, die noch unter Windows 2.x laufen (Real-Mode) und nicht an Windows 3.x angepaßt wurden,
- Software-Systeme, die unter Windows 3.0 im Protected-Mode arbeiten und nicht auf die erweiterte API von Windows 3.1 umgestellt wurden,
- solche Software-Systeme, die unter Windows 3.1 laufen,
- zusätzlich ohne ernste Warnmeldungen des Compilers mit dessen STRICT-Option compiliert sind und
- darüber hinaus ANSI-C bzw. C++-kompatibel sind.

Liegen Software-Systeme vor, die noch unter Windows 2.x im Real-Mode laufen, sollten diese zunächst an den Protected-Mode von Windows 3.x angepaßt werden, bevor diese im nächsten Schritt in eine 32-Bit-Umgebung portiert werden. Den besten Ausgangspunkt für eine Portierung nach Win32

bilden Software-Systeme, die im Protected-Mode des zugrundeliegenden Prozessors – Standard-Mode für 286-Prozessoren oder Enhanced-Mode für 386-Prozessoren und älter – fehlerfrei arbeiten und zusätzlich mit der STRICT-Option des verwendeten Compilers erstellt worden sind.

Die STRICT-Option versetzt den Compiler in die Lage, durch präzise Definition aller verwendeten Datentypen, strengere Typ- und Fehlerprüfungen vorzunehmen, als sonst üblich bei Funktionsaufrufen, Zuweisungen etc. Damit kann man dem Compiler die Details der Datentyp-Kompatibilität übertragen. Software-Systeme, die in C++ geschrieben sind, können nur bedingt auf die Vorteile der STRICT-Option des Compilers zurückgreifen. Ein C++-Compiler hängt an die, von ihm erzeugten, Funktionsnamen in den Objektdateien, noch Informationen über die Datentypen der Parameter sowie des Funktionsresultats. Diese Zusatzinformationen ermöglichen die Unterscheidung von überladenen Funktionen und werden zusätzlich für das spätere, typensichere Linken benötigt. Somit werden zwei C++-Module, wovon eines mit STRICT, das andere jedoch herkömmlich compiliert wird, nicht mehr auf die gleiche Definition des Datentyps (z.B. HWND), selbst bei Aufrufen der gleichen Funktion, zurückgreifen. Objektdateien, welche die unterschiedlichen Namen der externen Funktionen exportieren, können dann nicht mehr korrekt zusammengelinkt werden. Software-Systeme, auf der Basis von C++ müssen demzufolge, im Ganzen, an die STRICT-Option angepaßt und portiert werden. Die einzige Ausnahme bilden hierbei Funktionen, die explizit als ’extern C’ definiert sind und sowohl aus Dateien aufgerufen werden, die mit STRICT compiliert sind, als auch von herkömmlichen Dateien heraus angesprochen werden. Software-Systeme auf C-Basis können im Vergleich dazu, unter konsequenter Nutzung von ANSI-C-Prototypen – zumindest die Prozedurköpfe sollten, wegen der Parameterprüfungen, ANSI-C kompatibel formuliert sein –, schrittweise an die STRICT-Option angepaßt werden.

Die STRICT-Kompatibilität von Software-Systemen ist keine notwendige Bedingung für die Portierung nach Win32. Der Vorteil ’strikter’ Software-Systeme liegt vor allem in der grundsätzlich besseren Typprüfung, welche die zugrundeliegenden Quellen automatisch portabler machen. Dagegen spricht der erhöhte Zeitaufwand für den zusätzlichen Arbeitsaufwand und die Tatsache, daß trotz der STRICT-Option nicht alle Portabiliätsprobleme erkannt werden. Die Forderung nach fehlerfreier Ausführung des Software-Systems, im Protected-Mode einer Win16-Umgebung, ist hingegen notwendig. Somit müssen Software-Systeme für den Real-Mode zunächst an den entsprechenden Stellen an den Protected-Mode angepaßt werden. Der beste Ausgangspunkt für die Portierung eines Software-Systems ist somit deren ANSI-C bzw. C++-kompatibler Code und seine Fehlerfreiheit im Protected-Mode von Windows 3.1. Optional hierzu ist die STRICT-Kompatibilität, welche den Vorteil der höheren Sicherheit und Portabilität der Quellen mit sich bringt, jedoch aus Zeitgründen vernachlässigbar ist.

Bei der Zielsetzung steht die Win32-Plattform im Vordergrund. Hierbei stellt sich die Frage, ob das Software-System auch weiterhin in der Host-Umgebung laufen soll (Mehrweg-Portierung bzw. portable Programmierung einer Anwendung) oder ob es vollständig portiert und auf der Ursprungsplattform nicht weiterentwickelt wird (Einweg-Portierung bzw. reine Portierung).

Bei der Mehrweg-Portierung wird der Quelltext des Software-Systems parallel für die Host- und die Zielumgebung erstellt und gepflegt. Hierbei werden aus einem Satz von Quelltexten sowohl lauffähige Anwendungen für die Host- als auch die Zielumgebung generiert. Diese Zielsetzung wird mit Methoden der bedingten Compilierung, der Definition portabler Makros oder der Isolierung von portablen Codeteilen in selbständigen Funktionen, erreicht. Jedoch ist die vollständige Compilierung,

trotz aller eingesetzten Methoden, nicht immer zu erreichen. So sind Eigenschaften der Zielumgebung, die unter der Hostumgebung nicht existieren – z.B. multiple Threads unter Win32, welche unter Win16 nicht existieren –, dort nur sehr aufwendig oder gar nicht realisierbar. Dies erzwingt in den meisten Fällen eine zusätzliche Aufspaltung in getrennte Quellen (Schichtenarchitektur), welche dann jeweils Plattform-spezifisch sind und vom Rest der Quellen (portable Schicht) isoliert werden. Nach der erfolgreichen 'Portabilisierung' des Software-Systems, muß weiterhin regelmäßig Aufwand für die Aufrechterhaltung der Kompatibilität, zu beiden Zielplattformen, betrieben werden.

Die Einweg-Portierung geht von der Annahme aus, daß ein zu portierendes Software-System nur noch in der Zielumgebung zur Verfügung steht. Hierbei werden nicht portable Codeteile gestrichen und durch adäquaten Programmtext für die Zielumgebung ersetzt. Dieses Vorgehen stellt eine einmalige Anstrengung dar und ist mit der erfolgreichen Portierung beendet.

2.2.4.2 Strategien zur Portierung

Bei der Portierung von Software-Systemen unterscheidet man grundsätzlich zwei Arten von Strategien: Bottom-Up, vom Detail zum Ganzen, und Top Down ,vom Ganzen (Zentrum) und anschließendem, schrittweisem Verfeinern, zum Detail. Portiert man nach der Bottom-Up-Strategie, werden alle Quelldateien jeweils nacheinander angepaßt, compiliert und je nach ausgegebenen Fehlermeldungen und Warnungen solange verändert, bis eine Objektdatei, inklusive benötigter Ressourcen, erzeugt wird. Anschließend werden alle Objektdateien gelinkt und die Anwendung wird in ihrer Zielumgebung getestet. Hierbei führen Fehler zu erneuter Anpassung der Quellen, bis die Anwendung fehlerfrei läuft, womit die Portierung abgeschlossen ist. Die Vorteile der Bottom-Up-Strategie machen sich vor allem bei Anwendungen bemerkbar, deren Portierungsziel eine Mehrweg-Portierung darstellt und diese z.B. Rückwärtskompatibel zur Host-Plattform bleiben müssen:

- Erkennung wiederkehrender Code-Muster,
- Mechanisierung von Änderungsaufgaben (Master-Slave-Arbeitsteilung) bei großen Projekten.

Die Bottom-Up-Strategie erlaubt es, wiederkehrende Code-Muster, die entweder in einer Portabilitätsbibliothek isoliert oder mit Hilfe bedingter Compilierung, bzw. Makrodefinitionen, behandelt werden können, zu erkennen. Auch kann bei großen Projekten das einseitige Ändern von ähnlichen Code-Sequenzen, unter der Anweisung von erfahrenen Entwicklern, von relativ unerfahrenen Programmieren vorgenommen werden. Hierbei findet ein enges Wechselspiel zwischen Testen (Entwickler) und Ändern (Programmierer) statt.

Verfolgt man die Top-Down-Strategie, wird zunächst ein funktionierendes Minimalprogramm umgesetzt. Hierbei wird vom Hauptprogramm ausgehend, für alle Funktionsaufrufe – dies setzt eine prozedurale Struktur der Quellen voraus – eine leere Funktion gesetzt und dieses soweit an die Zielumgebung angepaßt, bis ein funktionierendes Hauptgerüst entsteht. Dieses Gerüst wird anschließend schrittweise mit Funktionalität der leeren Funktionen gefüllt, wobei diese jeweils zusammen mit den bereits funktionierenden Teilen angepaßt, compiliert und getestet werden. Der Top Down Ansatz bringt folgenden Vorteile mit sich:

- Vereinfachtes Debugging,
- motiviert und

- fördert Teamarbeit und Erfahrungsaustausch.

Durch die schrittweise und kontrollierte Anpassung der Quellen, können die einzelnen Teile des Software-Systems besser getestet und Fehler leichter beseitigt werden. Der Entwickler konzentriert sich immer auf einen bestimmten Teil und versucht diesen korrekt zu portieren. Somit gelangt er schnell zu Erfolgserlebnissen, welches auch die Motivation in positivem Maße beeinflußt. Auch können nach portiertem Hauptprogrammgerüst und definierten Schnittstellen, zur weiteren Funktionalität, alle Teile relativ unabhängig voneinander portiert und getestet werden. Hinzu kommen die im Laufe der Portierung gesammelten Erfahrungen der allerersten Anwendungsteile, welche sich auf die Portierung der noch ausstehenden Teile positiv auswirken.

Beide Strategien stellen einen jeweils idealisierten Fall dar und bei der Portierung eines Software-Systems wird man jeweils einzelne Elemente beider Strategien vereinigen müssen. Dennoch ist es vorteilhaft, sich zu Beginn der Portierung für eine grundlegende Vorgehensweise zu entscheiden. Entweder in einer umfassenden Bearbeitung, zunächst alle Quellen anzupassen und danach die Anwendung zu erzeugen oder umgekehrt, ein funktionierendes Grundgerüst zu schaffen um dieses anschließend, schrittweise mit Funktionalität zu füllen.

2.2.4.3 Hilfsmittel und Werkzeuge

Für die Portierung von Windows-basierten Software-Systemen gibt es eine Reihe von Hilfsmitteln und Werkzeugen, die diese unterstützen können. Neben der Verwendung von Zusatzbibliotheken, der Schaffung einer eigenen Portabilitätsbibliothek sowie der Verwendung geeigneter Informationsquellen, liefert das Quellenanalyse-Werkzeug 'PORTTOOL', Hinweise auf mögliche Portierungsprobleme und unterstützt somit die Erkennung von Portierungsproblemen, innerhalb der Quellen des Software-Systems.

Zusatzbibliotheken

Für die Portabilität von Software-Systemen existieren eine Reihe von, größtenteils kommerziellen Bibliotheken, welche als, mehr oder weniger stark abstrahierende Zusätze zur Windows-API, angeboten werden. Diese Zusatzbibliotheken ermöglichen ein gewisses Maß an portabler Programmierung, stehen aber in geeigneter Form nur dem C++-Entwickler zur Verfügung. Eine geeignete und portable GUI-Bibliothek, z.B. für C-Entwickler, ist entweder nicht vorhanden oder man kann nur wenig von ihr profitieren. Somit liegt der Einsatz von Zusatzbibliotheken im Bereich der objektorientierten Programmierung und C++ (z.B. Microsoft Foundation Classes von Microsoft oder Object Windows Library von Borland). Aber auch bei dem Einsatz objektorientierter Zusatzbibliotheken gibt es Verluste, hauptsächlich im Bereich der Flexibilität. Eine portable Zusatzbibliothek kann nur den kleinsten gemeinsamen Nenner aller unterstützten Systeme wirklich brauchbar umsetzen. Je größer die Unterschiede, d.h. je mehr Systeme unterstützt werden, desto kleiner wird auch der Kern, der von dieser Bibliothek portabel abgedeckt wird. Darüber hinaus bezahlt man die Portabilität seines Software-Systems mit der Abhängigkeit zu einem bestimmten Hersteller und die Einarbeitung in das entsprechende Programmiermodell erfordert zusätzlichen Zeitaufwand. Insgesamt erhöht der Einsatz von Zusatzbibliotheken jedoch die Portabilität und Wiederverwendbarkeit von Software-Systemen, bringt aber einen Verlust an Flexibilität, im Vergleich zur direkten Nutzung der einzelnen API's, mit sich.

Portabilitätsbibliotheken

Selbst erschaffene Portabilitätsbibliotheken stellen ein weiteres Hilfsmittel für die Portierung von Software-Systemen dar. Diese können neben Funktionen, die bestimmte Bereiche des Win32-API's portabel abdecken, wie z.B. systemspezifische Hilfsfunktionen, auch an das Zielsystem angepaßte Makrodefinitionen enthalten, z.B. für portable Datentypdefinitionen. Globale Programmier- und Portabilitätsrichtlinien für die Umsetzung bestimmter Programmierkonstrukte, erleichtern zusätzlich den Erfahrungs- und Informationsaustausch. Hierzu gehört u.a. die Verwendung der ungarischen Notation, als einheitliches Benennungsschema in den Quellen.

Informationsquellen

Neben den bisher vorgestellten Hilfsmitteln, die Portabilität von Windows-basierten Software-Systemen zu erhöhen, sind natürlich die diversen Herstellerdokumentationen und Hilfstexte, als Hilfsmittel für die Portierung zu verstehen. Dazu gehört in erster Linie die jeweils aktuellste Version der Bibliothek von Hilfetexten, des zugrunde liegenden Entwicklungssystems (z.B. Microsoft Software Development Network) und somit die vollständige Dokumentation der entsprechenden Programmierschnittstelle, der Betriebssysteme, des betrachteten Portierungsfelds.

PORTTOOL-Werkzeug

Zu den wichtigsten Werkzeugen für die Portierung Windows-basierter Software-Systeme, neben den der Entwicklungsumgebung und Programmerstellung (Compiler, Editor etc.), gehört das Werkzeug 'PORTTOOL', der Firma Microsoft. Dieses, zusammen mit der Win32-SDK ausgelieferte, Hilfsprogramm für die Portierungsanalyse, erlaubt das automatische Auffinden von Portierungsproblemen, in Bezug auf das hier betrachtete Portierungsfeld. Es erlaubt das Laden von Quelltexten in einen Editor und untersucht diesen auf Codeteile, die entweder gar nicht portabel oder nur eingeschränkt portabel sind. Die Quellen werden hierbei sukzessiv auf bestimmte Schlüsselworte hin untersucht und zusammen mit Kommentaren und Änderungsvorschlägen angezeigt. Die Schlüsselwörter und begleitenden Anmerkungen werden in einer zentralen Problembibliothek (Datei 'Port.ini', siehe Anhang B) erfaßt, welche als Grundlage für den Suchmechanismus verwendet wird. Der Suchalgorithmus ist einfach dem, eines normalen Editors und findet somit auch Stellen, die keinerlei Portierungsprobleme verursachen. Das Werkzeug 'PORTTOOL' ist deshalb nur als halbautomatisches Werkzeug für die Portierungsanalyse von Windows-basierten Software-Systemen einzustufen, da der größte Teil der Portierungsanalyse weiterhin manuell durchgeführt werden muß.

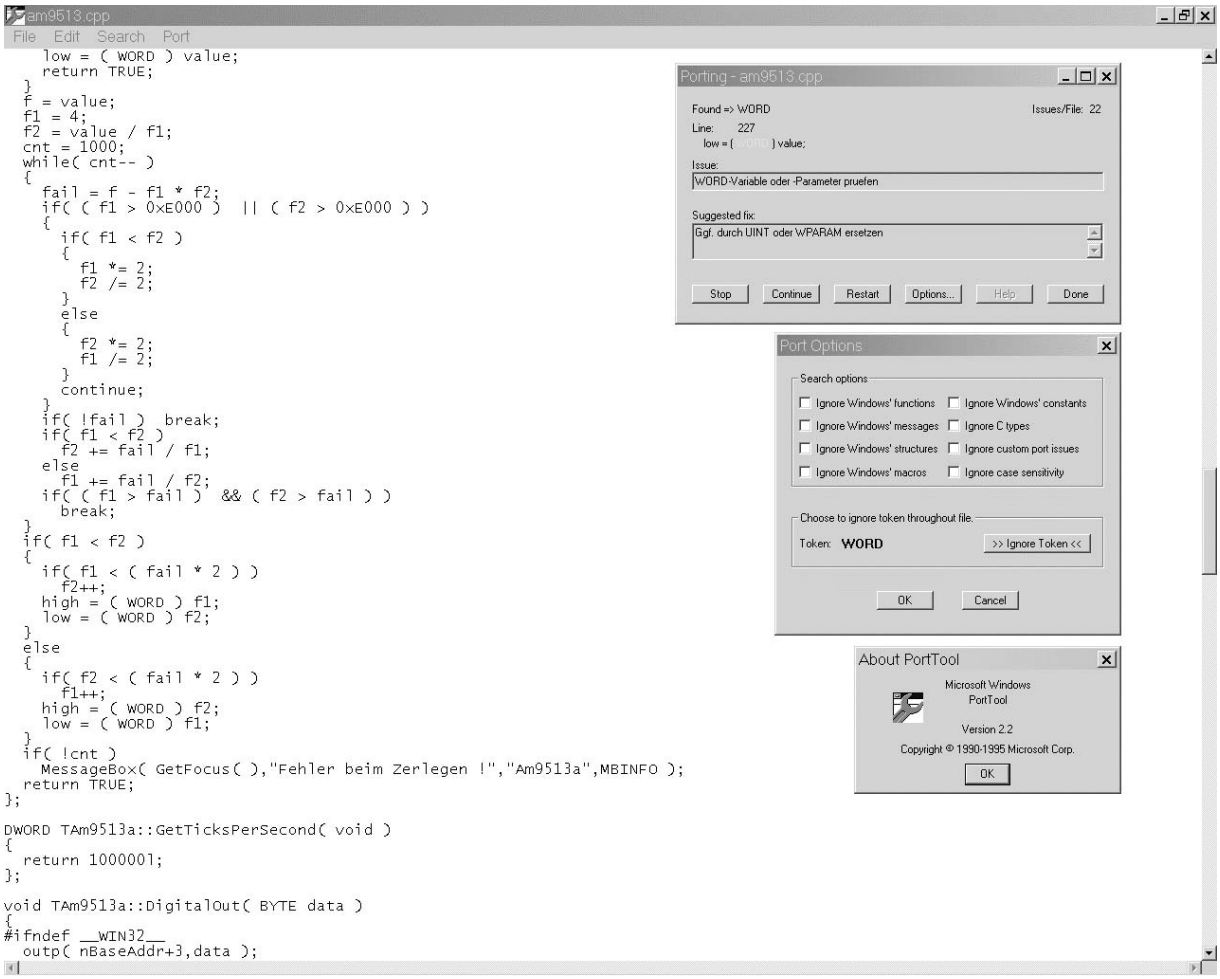


Abbildung 16: Werkzeug für die Portabilitätsanalyse: PORTTOOL

Kithara-Werkzeuge

Ein entscheidendes Kriterium für die Portabilität und die Portierung von hardwarenahen Software-Systemen, ist die Verfügbarkeit von geeigneten Treibern in den entsprechenden Betriebssystemumgebungen. Die Entwicklung von Hardwaretreibern, für bestimmte Zielumgebungen, ist in vielen Fällen mit einem hohen Maß an Kosten und Entwicklungszeit verbunden. Um hardwarenahe Software-Systeme, ohne den Aufwand einer Treiberentwicklung für die Zielumgebung, portieren zu können, existieren Werkzeuge, welche eine Art Universaltreiber zur Verfügung stellen, die, in das Software-System eingebunden, weiterhin hardwarenahe Programmierung zulassen. Für die Portierung von hardwarenahen Software-Systemen, unter Windows, existiert u.a. eine kommerzielle Lösung der Firma Kithara, die ’Driver Collection’-Toolkits [Kithara 01]. Diese umfaßt eine Reihe von Werkzeugen zur Erstellung von hardwarenahen 32-Bit-Anwendungen und Gerätetreibern. Der Aufwand für die Entwicklung von eigenen Kernel-Treibern – z.B. bei Windows NT durch dessen Architektur notwendig, um Kernel-Mechanismen nutzen zu können –, bei der Portierung von 16-Bit-Anwendungen in eine 32-Bit-Umgebung, kann somit, größtenteils entfallen. Der zeitliche Aufwand für die Portierung von Windows-Anwendungen wird damit verkürzt und die Portierung vereinfacht. Es ist somit möglich, ohne Einarbeitung in die Kernel-Treiber-Programmierung, hardwarenahe und 16-Bit-Windows-basierte Software-Systeme über eine gemeinsame Hardware-Schnittstelle, in eine 32-Bit-Umgebung zu portieren.

Die ’Driver Collection’ von Kithara wird über spezielle Kernel-Treiber (unter Windows 9x: VxD’s) realisiert, welche die zugrundeliegenden Mechanismen des jeweiligen Windows-Betriebssystemkerns und der Prozessorarchitektur nutzen. Die Funktionen der einzelnen Werkzeuge werden über Dll’s verfügbar gemacht und können in verschiedenen Programmiersprachen (C/C++, Pascal, Java etc.) und Entwicklungsumgebungen (Visual C++, Borland C++, Borland Delphi etc.) genutzt werden, sofern der Zugriff auf Dll-Funktionen unterstützt wird.

Die Sammlung an Werkzeugen enthält: den ’I/O Accelerator’ für Zugriffe auf I/O-Ports, physischen Speicher und PCI-Daten; das ’Hardware Toolkit’, für die Behandlung von Hardware-Interrupts innerhalb der Anwendung; ein ’Timer Toolkit’, zur Programmierung von genauen Timern und hochauflösenden Ermittlung der Systemzeit; das ’Serial Toolkit’, für die Programmierung der seriellen Schnittstellen sowie den ’Dos-Enabler’, für den Betrieb von hardwarenahen 16-Bit-Anwendungen unter Windows NT und Windows 2000. Folgende Funktionen werden von den Werkzeugen abgedeckt:

- Direkter Zugriff auf beliebige I/O-Ports,
- Zugriff auf physischen Speicher,
- Genaue Zeitmessungen,
- Präzise Timer in hoher Auflösung,
- Interrupt-Behandlung,
- Serielle Kommunikation,
- Code-Ausführung auf der Kernel-Ebene,
- Sonstige Funktionen (z.B. Zugriff auf ’Shared Memory’-Bereiche etc.).

3 Metriken und Portierungsaufwand

Ausgehend von allgemeinen Aspekten, bezüglich Software-Metriken, geht es in diesem Kapitel im Besonderen um die Darstellung von Metriken, die zur Bestimmung des Portierungsaufwands von Software-Systemen herangezogen werden können. Auf der Grundlage von bereits bestehenden Portabilitätsmaßen werden die Möglichkeiten der Vermessung von Software-Systemen unter dem Aspekt der Portabilität aufgezeigt und um aktuelle Entwicklungen im Bereich der objektorientierten Programmierung ergänzt.

3.1 Software-Metriken

In diesem Abschnitt geht es um die allgemeine Darstellung von Software-Metriken und deren Nutzen, in Bezug auf die Verwendung bei der Portabilitätsanalyse von Software-Systemen. Zunächst wird in einer Übersicht das Gebiet der Software-Metriken eingegrenzt. Die darauffolgende Klärung unterschiedlicher Begriffsvorstellungen, in Bezug auf Metriken, soll in einem weiteren Schritt den Sprachgebrauch dieser Arbeit festlegen.

3.1.1 Überblick

Die Zunahme der Anforderungen an Software-Systeme, durch enorme Leistungssteigerungen im Hardwarebereich sowie die gleichzeitige Forderung nach Fehlerfreiheit der Systeme, als ein wichtiger Qualitätsaspekt, erfordert eine umfangreiche Qualitätssicherung und ist somit ein wichtiger Aufgabenbereich der industriellen Software-Entwicklung. Auf der Grundlage von Qualitätsmodellen unternimmt man den Versuch, allgemeine Qualitätsbegriffe, durch Ableiten von Unterbegriffen zu operationalisieren und zu spezifizieren. Hierbei wird die Qualität eines Software-Systems durch bestimmte Qualitätsmerkmale, wie z.B. Funktionalität, Zuverlässigkeit, Übertragbarkeit, charakterisiert und mittels Teilmerkmalen, wie beispielsweise Übertragbarkeit, Anpaßbarkeit, Austauschbarkeit, verfeinert. Diese werden in einem nächsten Schritt, durch sogenannte Qualitätsindikatoren bzw. Metriken, meß- und bewertbar gemacht und mit diesen in Beziehung gesetzt. Die Aufgabe von Metriken besteht somit darin, eine quantitative Beschreibung der wesentlichen Qualitätsmerkmale von Software-Systemen zu ermöglichen, sodaß diese klassifiziert, verglichen und einer mathematischen Analyse unterzogen werden können. Eine Metrik oder ein Software-Qualitätsmaß gibt außerdem an, wie der Wert für einen solchen Qualitätsindikator bestimmt werden kann.

Software-Metriken werden aufgrund verschiedener Sichtweisen unterschiedlich kategorisiert. Eine allgemein anerkannte Einteilung findet man bei Fenton [Fenton 92], der Software-Metriken in drei Kategorien unterteilt: Ressource-Metriken, Prozeß- und Produkt-Metriken. Ressource-Metriken sind demzufolge Metriken, welche sich auf die zur Verfügung stehenden Hilfsmittel, innerhalb der Lebenszyklen eines Software-Systems (Entwicklung, Integration, Test, Support) beziehen und schließt den personellen Bereich mit ein. Prozeß-Metriken dienen der Messung von quantitativen Eigenschaften des Programmentwicklungsprozesses und der Entwicklungsumgebungen. Produkt-Metriken messen im Vergleich dazu, bestimmte Qualitätsmerkmale von Software-Systemen, welche zum einen die Codeteile selber und zum anderen die Dokumentation von Software-Systemen

betreffen. Aufgrund der Tatsache, daß sich Ressource- und Prozeß-Metriken auf den Entwicklungsprozeß von Software-Metriken beziehen, der Portierungsaufwand von Software-System jedoch in erster Linie von deren Qualitätsmerkmalen abhängig ist, sollen im folgenden nur die Produkt-Metriken näher betrachtet werden.

Produkt-Metriken erlauben somit, quantitative Aussagen über die Produktqualität von Systemkomponenten zu treffen. Jede Metrik quantifiziert dabei einen begrenzten Bereich einer Systemkomponente (intramodular) oder des Systems (intermodular) und erlaubt nicht, ein Software-System als Ganzes zu erfassen. In Balzert [Balzert 98] findet sich deshalb, auf intramodularer Ebene, auch der Begriff 'Komponentenmetriken'. Auf intermodularer Ebene spricht man hier von 'Kopplungsmetriken'.

Der größte Teil der Produkt-Metriken bezieht sich auf die Komplexität der zu analysierenden Software-Systeme. Die Zahl der publizierten Metrikvorschläge, allein im Bereich der klassischen Komplexitätsmetriken, wird mit über 200 angegeben [Zuse 91]. Hierbei unterscheidet man auf intramodularer Ebene die semantische und prozedurale Komplexität von Systemkomponenten. Die semantische Komplexität einer Systemkomponente wird durch ihre semantische Bindung – Verbindung zwischen den Systemelementen und die Aufgabenvielfalt einer Systemkomponente – ausgedrückt und läßt sich durch Metriken noch nicht direkt messen. Man kann jedoch sagen, daß eine gute Bindung immer dann vorliegt, wenn nur solche Elemente zu einer Einheit zusammengefaßt werden, die auch zusammengehören. Es gibt daher Ansätze von qualitativen Maßen, welche die Kompaktheit einer Prozedur oder Funktion in Form von Bindungsarten beschreiben (funktional, sequentiell, kommunikativ etc.). Zur Messung der prozeduralen Komplexität werden traditionell fünf Klassen unterschieden: Umfangsmetriken, logische Strukturmetriken, Datenstrukturmetriken, Stilmetriken und Interne Bindungsmetriken (syntaktische Bindung).

Umfangsmetriken vermessen die Größe von Software-Systemen (textuelle Komplexität). Diese haben ihren Ursprung in der zentralen Idee, daß die Größe eines Software-System und dessen Komponenten viel darüber aussagt, wie einfach der Umgang mit dem System ist und wie hoch das darauf bezogene, allgemeine Verständnis sein wird. Das diesbezüglich erste verfügbare Maß dieser Art war der Umfang an Quellcode, gemessen in LOC (Lines of Code). Es gibt Auskunft über die Anzahl der Quelltextzeilen eines Software-Systems und läßt sich einfach durch Auszählung ermitteln. Weitere frühe Maße waren der Umfang der Programmdateien oder die Anzahl der Funktionen. Basierend auf den Umfangsmetriken, stellte Halstead eine Anzahl Metriken auf, die auf der Zahl, der in einem Software-System verwendeten, unterschiedlichen Operanden und Operatoren sowie auf deren Gesamtzahl basiert.

Logische Strukturmetriken oder Kontrollflußmetriken setzen auf dem Kontrollfluß eines Software-Systems auf und basieren auf der Idee, je komplizierter der Kontrollfluß eines Software-Systems ist, desto höher ist seine Komplexität. Das am meisten verwendete Maß ist hierbei die zyklomatische Komplexität nach McCabe. Dabei wird, ausgehend von aufgestellten Kontrollflußgraphen, die hiernach leicht zu ermittelnde zyklomatische Zahl, als Maß für die Kompliziertheit von Kontrollflüssen berechnet.

Datenstrukturmetriken messen die Komplexität der Struktur und der Komponenten eines Software-Systems, indem sie die Zugriffe auf Daten und Datenstrukturen analysieren. Sie vermessen die Anzahl an Variablen eines Software-Systems, deren Gültigkeit und Lebensdauer. Außerdem überprüfen sie,

ob diese auch referenziert werden. Eine wichtige Datenstrukturmetrik ist die ’Segment-global usage pair’-Metrik. Sie vermißt die Anzahl der Lese- oder Schreibzugriffe von Komponenten, auf globale Variablen und wird als relative Häufigkeit (RUP, relative usage pair) ausgedrückt. Eine weitere Datenstrukturmetrik ist die Datenbindungsmetrik, welche die Beziehung zwischen den Komponenten eines Software-Systems bestimmt, indem sie deren Informationsaustausch über globale Variablen mißt.

Stilmetriken vermessen die psychologische Komplexität von Software-Systemen und beziehen sich somit auf die Faktoren, die es dem Menschen erschweren, Software-Systeme zu verstehen. Hierbei werden Software-Systeme z.B. daraufhin untersucht, ob der zugrundeliegende Quellcode richtig eingerückt ist und bestimmte Namenskonventionen eingehalten werden. Ein Beispiel für eine Stilmetrik ist die von Halstead definierte Berechnung der Schwierigkeit, ein Programm zu schreiben oder zu verstehen. Hierbei wird, ausgehend vom verwendeten Vokabular und der Anzahl der Operanden, ein Maß für den Aufwand zum Schreiben von Programmen, den Aufwand bei Code-Reviews und das Verstehen von Programmen bei Wartungsvorgängen ermittelt.

Interne Bindungsmetriken vermessen die syntaktische Bindung von Software-Systemen, durch Prüfung des Codes einer jeden Systemkomponente. Im Vergleich zur semantischen Bindung, die durch die Analyse der Bindungsart gekennzeichnet ist und bisher noch nicht direkt berechnet werden kann, existieren hier beispielsweise Metriken, die das Bindungsverhältnis von Systemkomponenten, aus dem Verhältnis der Anzahl funktional gebundener Komponenten, zur Anzahl aller Komponenten errechnen.

Auf der intermodularen Ebene von Software-Systemen, betrachtet man die Kopplung von Komponenten eines Systems (im Vergleich zur Bindung auf intramodularer Ebene, innerhalb einer Systemkomponente). Die Kopplung ist hierbei ein qualitatives Maß für die Schnittstellen zwischen den einzelnen Komponenten. Man unterscheidet den Kopplungsmechanismus (Aufruf, Verzweigung, externe Verbindung), die Schnittstellenbreite (Anzahl Parameter, Datentyp der Parameterelemente) und die Kommunikationsart (über Daten oder Steuerinformation). Kopplungen zwischen den Systemkomponenten und die Bindungen jeder Systemkomponente, bestimmen im Wesentlichen die Struktur eines Software-Systems. Diese ist um so ausgeprägter und die Modularität um so höher, je stärker die Bindungen der Systemkomponenten, im Vergleich zu den Kopplungen zwischen diesen, sind. Je ausgeprägter die Struktur eines Software-Systems ist, desto geringer ist deren Komplexität. Geringe Komplexität bedeutet einen hohen Grad an Einfachheit, gute Verständlichkeit und leichte Einarbeitung. Dieses wird erreicht, indem man die Kopplungen minimiert und die Bindungen maximiert.

3.1.2 Maß- und Metrikbegriff

In der Literatur, so hat der Überblick gezeigt, haben sich für die Vermessung von Software-Systemen eine Vielfalt an Begrifflichkeiten herausgebildet. Auch werden die Begriffe ’Maß’ und ’Metrik’ nicht einheitlich benutzt. Aus diesem Grund sollen in diesem Abschnitt die genannten Begriffe dargestellt und für den Rahmen dieser Arbeit festgelegt werden.

Nach [Zuse 98] finden sich für die Vermessung von Software-Systemen die Begriffe ’software metrics’, ’software engineering metrics’, ’software engineering measurement’ oder auch ’software

metrication'. Im deutschen Sprachraum benutzt man hierfür Begriffe wie: 'Softwaremetriken' bzw. 'Software-Metriken', 'Softwaremessung' oder 'Softwaremetrie'. Zuse hält den Begriff 'Software-Metriken' in diesem Bereich als für ungeeignet und schlägt die Verwendung von ,'software measurement' vor, was sich am besten mit dem Begriff 'Softwaremessung' ins Deutsche übersetzen läßt. Aus diesem Grund wird in dieser Arbeit die Vermessung von Software-Systemen im allgemeinen als 'Softwaremessung' bezeichnet.

Die Begriffe 'Maß' und 'Metrik' werden ebenfalls uneinheitlich definiert. Zuse sieht das Software-Maß als strukturverträgliche Abbildung mit homomorphen Eigenschaften, zwischen der empirischen Welt und der Welt der Zahlen. Ein Maß wird von ihm definiert, als empirische Relation zwischen den beteiligten Objekten dieser beiden Mengen.

Schmidt [Schmidt 87] definiert den Begriff 'Maß', allgemein wie folgt: Sei A eine beliebige Menge und $\mathcal{P}(A)$ deren Potenzmenge. Eine Abbildung $m:\mathcal{P}(A)\to\mathbb{R}$ heißt ein Maß auf $\mathcal{P}(A)$, wenn gilt:

- $\exists m(A_i)\ \forall A_i \in \mathcal{P}(A)$,
- $m(A_i)\begin{cases}\geq 0\ \forall A_i \in \mathcal{P}(A)\\ = 0,\ \text{falls } A_i = \varnothing\end{cases}$,
- $m(A_1 \cup A_2 \cup ... \cup A_n) = m(A_1) + m(A_2) + ... + m(A_n)$ für jede Folge paarweise disjunkter Teilmengen $A_1, A_2, ..., A_n \in \mathcal{P}(A)$.

Bei der Komplexität nach McCabe beispielsweise, wird eine zyklomatische Zahl ermittelt, um so die logische Komplexität eines Software-Systems zu messen. Die zyklomatische Komplexität eines sogenannten Kontrollgraphen, wird durch die Formel $v(G) = e - n + 2p$ ermittelt (siehe Kapitel 3.1.3.5). Hat man zwei Kontrollgraphen G_1 mit $v(G_1) = 2$ und G_2 mit $v(G_2) = 1$, so kann folgendes gesagt werden: Die Menge der Kontrollgraphen eines Software-Systems sei A und im Beispiel ist somit $A = \{G_1, G_2\}$. Die Abbildung $v: A \to \mathbb{R}$ sei eine Funktion, die jeden Graphen auf seine zyklomatische Zahl abbildet. Weiterhin sei eine Funktion $m:\mathcal{P}(A)\to\mathbb{R}$ mit $m(B) = \sum_i v(a_i)$ definiert, wobei $B \subseteq A$ und $a_i \in B$. Die Potenzmenge von A ist $\mathcal{P}(A) = \{\{\}, G_1, G_2, \{G_1, G_2\}\}$, d.h.: $m(\{\}) = 0$, $m(G_1) = v(G_1) = 2$, $m(G_2) = v(G_2) = 1$ und $m(\{G_1, G_2\}) = 2 + 1 = 3$, sodaß m die Anforderungen an ein Maß erfüllt.

Neben einer Vielzahl von Definitionen zum Begriff 'Metrik', wie z.B. die des IEEE [IEEE 93]:

> "A 'metric' is synonymous with a 'software quality metric' and defines a software quality metric as a function with input and output. Software quality metric have software data as inputs and a single numeric value as output. The output is interpreted as the degree to which software processes a given attribute that affect its quality.",

welche Metriken ebenfalls als Funktion, mit bestimmten Eingabedaten durch ein Software-System und einem entsprechend numerischen Wert als Ausgabe, sieht. Wobei der Ausgabewert als Grad für ein bestimmtes Qualitätsmerkmal eines Software-Systems angesehen wird.

Die ISO (ISO/IEC 9126, 1991) [ISO 91] beschreibt den Begriff der Software-Qualtitätsmetriken folgendermaßen:

> "A software quality metric is a quantitative scale and method that can be used to determine the value a feature takes for a specific product.",

als Definition einer quantitative Skala, mit einer entsprechenden Methode, den Wert für das entsprechende Produkt, auf dieser zu bestimmen.

Ein großer Teil der textuellen Definitionen von Metriken bezieht sich somit auf die Messung bestimmter Qualitätsmerkmale eines Produkts oder Prozesses und deren numerischer Repräsentanz.

In [Balzert 98] wird folgende Beschreibung für Metriken angegeben:

> "Eine Metrik soll in kompakter Form über technisch oder wirtschaftlich interessierende Sachverhalte informieren und meßbare Eigenschaften dieser Sachverhalte in Ziffern ausdrücken. Eine Software-Metrik definiert, wie eine Kenngröße eines Software-Produkts oder eines Software-Prozesses gemessen wird."

Weiterhin wird gesagt:

> "Um von einem Software-Maß sprechen zu können, muß eine Software-Metrik bestimmte Gütekriterien erfüllen."

Aufgeführt werden hierbei die Gütekriterien: Objektivität, Zuverlässigkeit, Validität, Normierung, Vergleichbarkeit Ökonomie und Nützlichkeit. Hierbei sei das am schwierigsten nachzuweisende Kriterium, das der Validität. D.h. der Nachweis, daß ein Maß auch tatsächlich das mißt, was es vorgibt zu messen. Hierbei wird deutlich und in qualitativer Hinsicht, zwischen dem Begriff 'Maß' und 'Metrik', unterschieden.

Orth [Orth 74] beschreibt im Gegensatz hierzu eine Metrik als Hilfe dafür, eine vollständige Rangordnung von Objektpaaren, aufgrund ihrer Ähnlichkeit bzgl. einzelner Merkmale aufzustellen. Der Begriff 'Metrik' wird von ihm wie folgt definiert:

Sei A eine nichtleere Menge und d eine reelle Funktion auf $A \times A$. Die reelle Funktion d heißt Metrik (oder Abstandsfunktion, Abstandsmaß), genau dann, wenn für alle $a, b, c \in A$ gilt:

- $d(a,a) = 0$ und wenn $a \neq b$, dann $d(a,b) > 0$ (Positivität),
- $d(a,b) = d(b,a)$ (Symmetrie),
- $d(a,b) + d(b,c) \geq d(a,c)$ (Dreiecksungleichung).

Die Elemente aus A heißen Punkte und die reelle Zahl $d(a,b)$ heißt Abstand zwischen a und b.

Beispielsweise kann in einer Menge von Software-Systemen $\mathcal{S} = (S_1, ..., S_4)$ mit dem Merkmal m_1 (Länge des Software-Systems, gemessen in Lines of Code) und den gemessenen Werten: $m_1(S_1) = 20,\ m_1(S_2) = 27,\ m_1(S_3) = 100,\ m_1(S_4) = 75$, folgende Abstandsmatrix bzgl. des

Merkmals m_1 berechnet werden: $d(S_i, S_j) = \left| S_i - S_j \right|,\ i, j = 1...4$. Daraus ergibt sich die Rangfolge: $d(S_1, S_2) < d(S_3, S_4) < d(S_2, S_4) < d(S_1, S_4) < d(S_2, S_3) < d(S_1, S_3)$, da $7 < 25 < 48 < 55 < 73 < 80$.

Mit dieser Definition ist ebenfalls ein gewisser Unterschied. zwischen dem Begriff ’Maß’ und ’Metrik’ erkennbar. Während ein Maß als absoluter Wert, bezüglich der Objekte eines bestimmten Qualitätsmerkmals, eines untersuchten Systems oder einer Komponente aufgefaßt werden kann, implizieren Metriken eine Art Rangordnung zwischen den Objekten eines betrachteten Merkmals. Sie lassen somit keine Aussage über die absolute Qualität eines Objekts, bzgl. dieses Qualitätsmerkmals zu. So kann z.B. die Aussage ’ Software-System S_1 ist portabler als S_2 ’ wenig Aufschluß darüber geben, wie portabel S_1 oder S_2 wirklich ist und wie hoch der jeweilige Portierungsaufwand tatsächlich sein wird. Allgemein kann jedoch gesagt werden, daß die Begriffe ’Maß’ und ’Metrik’ synonym gebraucht werden und die dargestellte Abgrenzung der Begrifflichkeiten nur in Einzelfällen stattfindet (z.B. bei der Herausstellung qualitativer Unterschiede). Da in den aktuellen Lehrbüchern zumeist der Begriff ’Metrik’ und nicht ’Maß’ verwendet wird, um quantitative Bewertungen von Software-Produkten oder Software-Prozessen zu bezeichnen, soll im Rahmen dieser Arbeit, dem allgemeinen Kanon folgend, der Begriff ’Metrik’ bzw. ’Software-Metrik’ verwendet werden. Explizite Herausstellung qualitativer Aspekte werden mit Hilfe der Maß-Definition nach Balzert berücksichtigt. Beispiele aus der Literatur werden, falls sie dem begrifflichen Rahmen dieser Arbeit nicht entsprechen, in der vom jeweiligen Autor gewählten Begrifflichkeit dargestellt.

3.1.3 Traditionelle Produkt-Metriken für strukturelle Komplexität

In diesem Abschnitt werden, ausgehend von der obersten Ebene einer Systemumgebung, zunächst die Kopplungsmaße zwischen prozedural strukturierten Systemkomponenten eines Software-Systems betrachtet. Danach werden auf Komponentenebene die Bindungen von Prozeduren und Funktionen (semantische Bindung) behandelt und anschließend verschiedene Metriken zur Messung der prozeduralen Komplexität vorgestellt.

3.1.3.1 Analyse der Kopplungsart

Die Kopplung ist ein qualitatives Maß für die Komplexität der Beziehungen zwischen Systemkomponenten (Prozeduren oder Funktionen) eines Software-Systems und spielt für deren Festlegung der Produktqualität eine entscheidende Rolle. Der Grad der Kopplung drückt Abhängigkeiten einzelner Systemkomponenten zu ihrer Systemumgebung aus (z.B. stellen Prozeduraufrufe in Prozeduren eine Kommunikation zwischen diesen dar, die zu Abhängigkeiten, Verbindungen und Kopplungen führen). Hierbei spiegelt eine Zunahme der Kopplung, höhere Abhängigkeit zur Systemumgebung (anderen Prozeduren) wieder. Geringe Kopplung von Systemkomponenten erhöht deren Verständlichkeit und ist ein entscheidender Faktor für reduzierte Einarbeitungszeiten. Ebenso können Änderungen in Systemkomponenten, mit geringer Kopplung, einfacher durchgeführt werden, da diese meist auf entsprechende Komponenten beschränkt bleiben. Das Ziel ist, Kopplungen zwischen Systemkomponenten zu minimieren. In [Myers 75] werden sechs Kopplungsarten unterschieden: Inhaltskopplung, Common-Kopplung, externe Kopplung, Kontroll-Kopplung, Stamp-Kopplung und Daten-Kopplung. Einige Kopplungsarten von Myers enthalten

verschiedene Aspekte von Kopplungen, die nicht explizit Erwähnung finden (z.B. liegt bei einer Inhaltskopplung auch eine Kontrollkopplung vor, wenn beispielsweise eine Komponente die Anweisung einer anderen ändert).

Stevens hingegen versucht in seinem Ansatz diesen Aspekt zu berücksichtigen, in dem er die Kopplung zwischen den Systemkomponenten in verschiedenen Dimensionen differenziert. Nach Stevens [Stevens 81] unterscheidet man folgende Dimensionen: Kopplungsmechanismus (type of connection), Schnittstellenbreite (size) und Kommunikationsart (what is communicated). Es sollen deshalb die drei Dimensionen nach Stevens näher betrachtet werden. Er erwähnt zusätzlich eine vierte Dimension, die Klarheit (clarity), welche sich auf die Einfachheit der Kopplung bezieht und für das Verständnis einer Systemkomponente relevant ist. Diese Dimension soll hier nicht näher betrachtet werden, da sie keine meßbare Größe darstellt und im Wesentlichen durch die drei anderen Dimensionen bereits erfaßt wird.

Kopplungsmechanismus

Stevens unterscheidet drei Arten, um Komponenten untereinander zu koppeln: Aufruf (CALL, Prozedur- bzw. Funktionsaufruf), Verzweigung (PERFORM, goto) und externe Verbindung. Ein Aufruf mit der expliziten Angabe von Parametern ist die einfachste, verständlichste, flexibelste und am wenigsten fehleranfällige Form der Kopplung. In den meisten Programmiersprachen wird dieser Kopplungsmechanismus über das Prozedur- oder Funktionskonzept realisiert. Der Datenaustausch erfolgt über Parameter, die in einer Parameterliste enthalten sind. Kennt eine Programmiersprache diesen Aufrufmechanismus mit Parameterkonzept nicht, können auch andere Mechanismen verwendet werden, die aber zu einer Erhöhung der Kopplungsstärke führen.

Der wesentliche Unterschied zwischen einer Verzweigung in der Praxis einer ’goto’-Anweisung (z.B. ’PERFORM’-Anweisung in Cobol) und dem Aufruf einer Prozedur besteht darin, daß im ersten Fall keine explizite Parameterübergabe stattfindet, bzw. möglich ist und die Daten zwischen der aufrufenden und der aufgerufenen Systemkomponente über gemeinsame Datenbereiche (shared data area) ausgetauscht werden (in FORTRAN beispielsweise durch die ’COMMON’-Anweisung, d.h. verschiedene Komponenten greifen auf eine gemeinsame Datenstruktur zu). Der Nachteil dieser Kopplung besteht darin, daß die Daten nicht explizit übergeben werden und damit die Schnittstellen nicht eindeutig identifizierbar bzw. schwer ersichtlich sind.

Eine externe Datenverbindung liegt immer dann vor, wenn eine Prozedur direkt auf ein Datenelement oder eine Anweisung, innerhalb einer anderen Prozedur, zugreift (Inhaltskopplung, z.B. Verzweigung einer Systemkomponente in eine andere, ohne daß ein externer Einsprungspunkt definiert ist). In PL/1 kann diese Zugriffsart, beispielsweise mittels der ’EXTERNAL’-Anweisung, realisiert werden. Der Kopplungsmechanismus der externen Datenverbindung hat den Nachteil, daß ein Fehler in einer extern gekoppelten Komponente, durch eine mit ihr gekoppelten Komponente entstanden sein kann und deshalb in manchen Fällen alle extern gekoppelten Komponenten betrachtet werden müssen. Dieser Kopplungsmechanismus ist der komplexeste, verwirrendste und fehleranfälligste und sollte daher nach Möglichkeit vermieden werden.

Schnittstellenbreite

Diese wird bei Stevens, durch die Anzahl der einzelnen ausgetauschten Datenelemente und deren Datentyp, bestimmt. Eine Prozedurkopplung wird um so geringer, je weniger Parameter vorhanden und je mehr Parameterelemente, durch elementare Datentypen definiert sind. Je weniger Daten eine Schnittstelle passieren, desto geringer ist die Kopplungsstärke. Der Umfang der Daten bezieht sich hierbei auf die Schnittstellenbreite (Anzahl der Parameter). Bei Prozeduraufrufen ist die Anzahl der ausgetauschten Elemente, gleich der Anzahl der Parameter, solange nicht weitere Zugriffe auf globale Datenbereiche hinzukommen. Erfolgt eine Kopplung über globale Speicherbereiche (Common-Bereich) ist die maximale Anzahl der Kopplungen $KO_{\max} = K(K-1)\cdot N$, mit K als Anzahl der Komponenten und N als Anzahl der einzelnen Datenelemente im Common-Bereich. Meist ist die Kopplung zwischen zwei Komponenten aber geringer, da nicht jede Komponente alle Datenelemente des Common-Bereichs verwendet. Probleme, die durch den Zugriff auf gemeinsame Datenbereiche entstehen, sind:

- die Änderung einer Komponente, die eine Änderung des Common-Bereichs zur Folge hat und sich auf alle Komponenten auswirken kann, die auf den Common-Bereich zugreifen;
- eine Beschränkung der Zugriffsmöglichkeiten einer Komponente auf die Daten, die im Fall einer Common-Kopplung nicht möglich ist, da jede Common-gekoppelte Systemkomponente auf jedes Datenelement des Common-Bereichs zugreifen kann (In FORTRAN ist eine Reduzierung dieser Kopplungsform durch die Definition von Named-Common-Bereichen möglich);
- die Verwendung einer Systemkomponente in verschiedenen Software-Systemen, welche auf Common-Bereiche zugreift, da hier meist keine äquivalente Umgebung vorhanden ist.

Generell sollten keine gemeinsam benutzten Datenbereiche und keine globalen Variablen verwendet werden.

Betrachtet man die Parameter einer Schnittstelle, so könnte man versuchen, durch Zusammenfassen von Datentypen in Datenstrukturen, eine Minimierung der Schnittstellenbreite und damit der Kopplung zu erreichen. Deshalb spielt der Datentyp der einzelnen Datenelemente einer Schnittstelle, eine wesentliche Rolle. So sind z.B. zusammengesetzte Datentypen, wie Arrays oder Records, auf dessen Elemente einzeln zugegriffen werden kann, nur in homogenen Datenreihungen (jedes Element trägt die gleiche Art von Information), bzw. zusammengehörigen Datenreihungen, sinnvoll. Durch die Bündelung von Parametern (Datenstrukturkopplung), sinkt zwar die Anzahl der Parameter, die Kopplungsstärke wird aber erhöht, da die Verständlichkeit der Schnittstelle beeinträchtigt wird, falls man mehrere, nicht zusammengehörige Daten, zufällig bündelt. Deshalb ist Stevens der Ansicht, daß die Anzahl der Datenelemente an der Schnittstelle, in dem Sinne maximiert werden sollte, daß jedes benötigte Datenelement, das eine logische Einheit darstellt, auch explizit aufgeführt sein sollte. Je mehr Daten zu einer Struktur künstlich zusammengefaßt und übergeben werden, um so unverständlicher und schlechter wartbar wird eine Prozedur. Weiterhin wird durch Übergabe komplexer Datenstrukturen das Geheimnisprinzip unterlaufen.

Kommunikationsart

Stevens [Stevens 81] unterteilt die Kommunikationsart in drei Gruppen: Hybridkopplung (Kontroll- und Datenkopplung), reine Kontrollkopplung und einfache Datenkopplung, wobei er bei der reinen Kontrollkopplung unterscheidet zwischen: direkter Verzweigung, Code-Kopplung und

Kontrollparametern. Danach geschieht die Kommunikation zwischen Systemkomponenten auf zwei Arten: über Daten und/oder über Steuerinformation (Kontrollinformation).

Hybrid-Kopplung liegt immer dann vor, wenn eine Komponente den Code einer anderen verändert und der Hybrid-Begriff sich aus der Tatsache ergibt, daß die modifizierende Systemkomponente den Code der anderen als Daten betrachtet, die geänderte Komponente aber dadurch kontrolliert wird. Diese Kommunikationsart ist in den meisten höheren Programmiersprachen nicht realisierbar, sodaß sie in erster Linie nur bei Assemblerprogrammen auftritt.

Die Kontrollkopplung bezieht die rufende Prozedur in die Steuerung der gerufenen Prozedur mit ein und widerspricht somit dem Geheimnisprinzip. Die Form der direkten Verzweigung tritt auf, wenn von einer Komponente, direkt in eine andere Komponente verzweigt wird und Änderungen an einer Komponente, in welche direkt verzweigt wird, oft erst dann durchführbar sind, wenn alle Komponenten, die direkt in diese Komponente verzweigen, überprüft wurden. Dieser Kopplungsmechanismus ist deshalb problematisch, da Komponenten, die auf diese Art verzweigen, nicht so einfach aufzufinden sind. Im Falle eines Fehlers kann daher jede, an der Verzweigung beteiligte Komponente, ein potentieller Verursacher des Fehlers sein.

Die Code-Kopplung ist gegeben, falls verschiedene Komponenten gleiche Code-Segmente verwenden, wie dies z.B. mittels Paragraphen in Cobol (mehrere Einsprungspunkte) der Fall sein kann. Liegt beispielsweise ein Paragraph B zwischen den Paragraphen A und C und verwendet man dann eine Anweisung der Art ’PERFORM A TROUGH C’, so kann eine Änderung am Paragraphen B zu einer fehlerhaften Ausführung dieser PERFORM-Anweisung führen.

Kontrollparameter haben Einfluß auf die Verarbeitungsreihenfolge, der aufgerufenen Komponente und setzen Kenntnisse über deren Inhalt voraus. Wird z.B. über eine Systemkomponente die Steuerung mehrerer Drucker realisiert, so kann die Auswahl des Druckers von anderen Komponenten, mit Hilfe einer Kontrollvariablen übernommen werden und man erhält hier eine Form der Kopplung, die zumeist nicht notwendig ist. So wäre es denkbar, daß man die Druckersteuerung in mehrere Komponenten aufteilt. Jeder Drucker erhält seine eigene, ihm zugehörige Systemkomponente, wobei die Auswahl des Druckers in die aufrufende Komponente verlegt wird. Kontrollkopplungen über Kontrollparameter sollten, wie auch die anderen Fälle, möglichst vermieden werden.

Eine reine Datenkopplung liegt vor, wenn der Informationsaustausch über Parameter erfolgt, die keine Kontrolldaten und keine strukturierten Daten darstellen. Die Übergabe von reinen Daten ist somit die einfachste Kommunikationsart und für das Funktionieren eines Systems ausreichend. Sobald ein Prozedurparameter dazu verwendet wird, anderen Prozeduren mitzuteilen, was zu tun ist, handelt es sich schon nicht mehr um eine reine Datenkopplung, sondern um eine Kontrollkopplung. Dadurch werden die Verbindungen zwischen Prozeduren erhöht, da die Prozedur, die auf eine andere zurückgreift, über Kenntnisse des Inneren dieser Prozedur, verfügen muß. Die reine Datenkopplung ist die schwächste Kopplungsart, da die gegenseitige Beeinflussung der betroffenen Komponenten hierbei am geringsten ist und zusätzlich auf nur zwei Komponenten beschränkt bleibt. Somit ist die anzustrebende Kopplung zwischen zwei Systemkomponenten, der Aufruf mit Parametern, wobei die Parameter jeweils nur aus reinen Daten bestehen. Der Aufruf wird benutzt, um eine minimale Anzahl von Datenparametern auf die verständlichste Weise zu übertragen. Folgende Voraussetzungen müssen nach [Balzert 98] erfüllt sein, um diese Art der Kopplung zu erreichen (schmale Datenkopplung):

- Prozedurkopplungen werden nur durch Aufruf anderer Prozeduren hergestellt;
- Die eigentliche Kommunikation erfolgt nur über explizite Parameter;
- Globale Größen oder gemeinsam benutzte Datenbereiche gibt es nicht;
- Das Geheimnisprinzip wird eingehalten.

Sind diese Voraussetzungen erfüllt, kann man demnach die Überprüfung von Prozedurkopplungen auf die Fragen: ’Liegt eine reine Datenkopplung vor?’ und ’Handelt es sich um eine schmale Schnittstelle?’ konzentrieren. Sie sind nur durch manuelle Prüfmethoden beantwortbar. Es kann u.U. recht schwierig sein, reine Daten und Steuerdaten zu unterscheiden. Hilfreich ist es hierbei, auf die Beschreibung der Parameter zu achten. Datenparameter werden meist durch Substantive, Zustandsparameter, welche von Kontrolldaten (Steuerdaten) zu unterscheiden sind, durch Adjektive und Kontrollparameter durch Verben beschrieben. Ist eine Prozedur durch eine Kontrollkopplung mit anderen Prozeduren gekoppelt, muß man die Systemstruktur überprüfen, was dazu führt, alle betroffenen Prozeduren auf funktionale Bindung zu untersuchen. Eine besonders unübersichtliche und änderungsunfreundliche Steuerungskopplung hat man, wenn verschiedenen Wertebereichen eines Parameters, unterschiedliche Bedeutungen zugeordnet werden.

Abschließend kann gesagt werden, daß jedes Software-System ein Minimum an Kopplung zwischen den einzelnen Komponenten benötigt, um die ihm gestellte Aufgabe zu erfüllen. Ziel der Software-Entwicklung muß es daher sein, zusätzliche und unnötige Kopplung zu vermeiden oder ganz zu eliminieren.

3.1.3.2 Analyse der Bindungsart

Die Bindung ist ein qualitatives Maß für die Kompaktheit einer Systemkomponente und spielt für deren Produktqualität eine entscheidende Rolle. Man betrachtet hierbei die Beziehungen zwischen den Elementen einer Systemkomponente und untersucht, wie eng diese miteinander verbunden sind. Außerdem wird über semantische Bindungsmetriken untersucht, wie groß die Anzahl der Aufgaben ist, die in einer Systemkomponente erledigt werden.

Nach [Myers 75] werden hierbei folgende Bindungsarten unterschieden: funktionale Bindung, informale Bindung, kommunikative Bindung, prozedurale Bindung, klassische Bindung, logische Bindung und zufällige Bindung. Die Bindungsstärke nimmt hierbei von links nach rechts ab, d.h. funktionale Bindung bindet stärker als informale Bindung, wobei die zufällige Bindung, die schwächste Form der Bindung von Elementen einer Systemkomponente darstellt.

Bei der zufälligen Bindung stehen die Elemente in keiner besonderen Beziehung zueinander. D.h. jedes Element arbeitet autonom und ist nicht über Beziehungen mit anderen Elementen der Systemkomponente verbunden. Dies ist z.B. dann der Fall, wenn ein Software-System aus Speicherplatzgründen in Komponenten aufgeteilt wird.

Eine logische Bindung einer Systemkomponente besteht, wenn zwischen den Elementen eine logische Beziehung besteht. So stellt beispielsweise die Zusammenfassung aller Eingabefunktionen, in einer Systemkomponente, eine logische Bindung dar.

Mit der klassischen Bindung wird die logische Bindung um den zeitlichen Aspekt erweitert. D.h., daß die Elemente, neben der logischen Bindung, auch noch zeitlich miteinander verbunden sind. Ein Beispiel hierfür sind Initialisierungskomponenten, deren unterschiedliche Initialisierungen eine logische Beziehung aufweisen und zusätzlich zu einem bestimmten Zeitpunkt ausgeführt werden.

Prozedurale Bindung liegt immer dann vor, falls eine Komponente mehrere Funktionen ausführt, die einen bestimmten Bezug zu einem gemeinsamen, übergeordneten Problem haben.

Eine Komponente ist kommunikativ gebunden, wenn über die prozedurale Bindung hinaus, alle Elemente miteinander kommunizieren. D.h. sie greifen gemeinsam auf bestimmte Daten zu oder tauschen Daten miteinander aus, beispielsweise bei der Übergabe von Funktionsparametern, in der die Ausgabe einer Funktion, die Eingabe einer anderen Funktion darstellt.

Informal gebundene Komponenten führen verschiedene Funktionen aus, welche durch den Einsprungspunkt, in der jeweiligen Komponente repräsentiert werden. Jeder Einsprungspunkt hat hierbei die Eigenschaften einer funktional gebundenen Komponente.

Eine funktionale Bindung besteht, wenn alle Elemente einer Komponente an der Ausführung einer einzigen, abgeschlossenen Funktion beteiligt sind. So sind z.B. einfache mathematische Funktionen, wie die Berechnung der Quadratwurzel, Systemkomponenten mit funktionaler Bindung. Funktionale Bindung ermöglicht aber nicht nur die Realisierung einfacher, sondern auch komplexer Funktionen. Diese werden mit Hilfe weiterer Prozeduren (importierte Prozeduren) realisiert, welche ebenfalls funktional gebunden sein sollten. Nach [Balzert 98] besitzt eine funktionale Bindung folgende Kennzeichen:

- Alle Elemente tragen dazu bei, ein einzelnes, spezifisches Ziel zu erreichen;
- Es gibt keine überflüssigen Elemente;
- Die Aufgabe läßt sich mit genau einem Verb und genau einem Objekt vollständig beschreiben;
- Leichter Austausch gegen ein anderes Element, das denselben Zweck erfüllt;
- Hohe Kontextunabhängigkeit, d.h. einfache Beziehungen zur Umwelt.

Neben der hohen Kontextunabhängigkeit sind geringe Fehleranfälligkeit bei Änderungen, ein hoher Grad an Wiederverwendbarkeit, leichte Erweiterbarkeit und Wartbarkeit, wichtige Vorteile einer funktionalen Bindung und führen zu einer wesentlichen Verfestigung der internen Prozedurstruktur. Problematisch bei der Bestimmung der Bindungsart ist jedoch, daß diese noch nicht automatisch ermittelt werden kann, bzw. keine objektiv meßbaren Kriterien zu deren Bestimmung vorliegen und nach Balzert, nur durch manuelle Prüfmethoden bestimmbar ist (Entwurfs- oder Codeüberprüfung). Im Entwurf wird anhand der Aufgabenbeschreibung analysiert, welche Bindungsart vermutlich vorliegt. Eine endgültige Entscheidung darüber, wird anhand der Implementierung getroffen. Das Ziel einer guten Bindung liegt demnach darin, nur solche Elemente zu einer Systemkomponente zusammenzufassen, die logisch zusammengehören und funktional gebunden sind.

3.1.3.3 Programmlänge

Die Programmlänge $L(P)$ ist ein einfacher Vertreter der Umfangsmetriken und mißt die Anzahl an Programmzeilen eines Software-Systems. Die Idee bei der Ermittlung der Komplexität eines Software-

Systems mittels Programmlänge beruht im Wesentlichen auf der Annahme, daß eine Erhöhung der Anzahl an Programmzeilen auch eine Erhöhung der Komplexität zur Folge hat. Der Vorteil der Messung der Programmlänge liegt in seiner einfachen Bestimmbarkeit. So ist die Programmlänge von Software-Systemen relativ einfach zu bestimmen, bzw. wird von den meisten Compilern automatisch ermittelt. Die Programmlänge wird häufig für Kostenabschätzung oder Produktivitätsuntersuchungen eingesetzt.

Wesentliche Nachteile entstehen aber, sobald man Leer- oder Kommentarzeilen mitzählt oder Mehrfachanweisungen und mehrfache Funktionsaufrufe in einer Zeile ignoriert und somit u.U. unterschiedliche Arten von Software-Systemen über ein gemeinsames Programmlängenmaß vergleicht. Auch wird hier nicht die Komplexität der Beziehung zwischen den Programmkomponenten bewertet. Somit ist die Programmlänge eines Software-Systems kein aussagekräftiges Komplexitätsmaß, um die verschiedenen Einflußfaktoren der Komplexität zu erfassen.

3.1.3.4 Halstead-Metriken

Aufwendigere Umfangsmetriken, als solche auf einfacher Informationsbasis, wie die Programmlänge, beruhen auf der Basis der textuellen Komplexität. So unternahm Halstead, Ende der 70er-Jahre, erste Versuche, eine Reihe von Software-Metriken für den Software-Entwicklungsprozeß aufzustellen. Die von Halstead [Halstead 77] vorgeschlagenen Metriken sind ein gutes Beispiel dafür, wie auf der Basis einfacher Metriken, eine Vielzahl weiterer Metriken ableitbar sind. Die Grundlage für die Ableitung weiterer Metriken bilden folgende Basismetriken:

- η_1 : Anzahl der unterschiedlichen Operatoren,
- η_2 : Anzahl der unterschiedlichen Operanden,
- N_1 : Gesamtzahl der verwendeten Operatoren,
- N_2 : Gesamtzahl der verwendeten Operanden.

Operatoren sind alle Symbole und Schlüsselwörter, die eine Aktion kennzeichnen. Dazu gehören z. B. arithmetische Symbole oder Funktionsnamen. Als Operanden werden alle Symbole angesehen, die Daten repräsentieren, z.B. Variablen, Konstanten oder Labels. Ein Nachteil ist jedoch, daß bei der Ermittlung der Basismetriken keine Operatoren und Operanden berücksichtigt werden, die in Deklarationen oder Ein-/Ausgabefunktionen vorkommen.

Ausgehend von diesen Basismetriken hat Halstead eine Reihe von Metriken entwickelt, die unterschiedliche Eigenschaften eines Software-Systems erfassen. Dazu gehören:

- $\eta = \eta_1 + \eta_2$: Größe des Vokabulars;
- $N = N_1 + N_2$: Länge der Implementierung;
- $N' = \eta_1 \cdot \log_2 \eta_2 + \eta_2 \cdot \log_2 \eta_2$: alternative Längengleichung;
- $V = N \cdot \log_2 \eta$: Volumen einer Implementierung;
- $V^* = (2 + \eta_2^*) \cdot \log_2 (2 + \eta_2^*)$: potentielle Volumen;
- $L = V / V^*$: Programmebene;
- $E = V / L$: Aufwand für die Implementierung eines Software-Systems der Länge N ;
- $T = E / \beta$: Implementierungsdauer.

Die ersten, von den Basismetriken, abgeleiteten Metriken sind die Summe der Operatoren und Operanden und werden als 'Vokabular' η eines Software-Systems bezeichnet. Die Programmlänge N wird durch die Gesamtzahl aller Zeichen berechnet und bildet ebenfalls ein einfaches Komplexitätsmaß. Eine Alternative zur einfachen Programmlänge stellt die sogenannte 'Längen-Gleichung' dar, die auf der Annahme beruht, daß die Länge eines strukturierten Software-Systems, eine Funktion der Anzahl verschiedener Operatoren und Operanden ist. Untersuchungen haben jedoch gezeigt, daß die 'Längen-Gleichung' eine schlechte Näherung für N, bei kleinen oder sehr umfangreichen Software-Systemen, ist. Die Genauigkeit der Vorhersage von N durch N' hängt einerseits vom Programmierstil, andererseits aber auch von der Häufigkeit der Verwendung einzelner Operatoren und Operanden ab. Bessere Näherungen für N erhielt man mit N', wenn die in Deklarationen enthaltenen Operatoren und Operanden, bei der Ermittlung der Basismetriken η_1 und η_2, mitgezählt wurden.

Die 'Volumen-Gleichung' wird in zwei Varianten interpretiert. Die erste sieht das Volumen V als die minimale Anzahl an Bits, die für die Codierung des Software-Systems benötigt werden. In der zweiten wird das Volumen als die Anzahl der geistigen Vergleiche aufgefaßt, die notwendig sind, um das Software-System zu implementieren. Bei dieser Interpretation geht man von der Annahme aus, daß der Programmierer die Auswahl des nächsten Zeichens aus der Menge η (Summe der Operatoren und Operanden) durch binäre Suche trifft. Dies entspricht jedoch nicht unbedingt der Realität, da beim Schreiben von Anweisungen, meist nur bestimmte Operatoren und Operanden sinnvoll sind und nicht erst eine Anzahl von Vergleichen durchgeführt werden, um sich dann für das passende Zeichen zu entscheiden. Legt man als Hypothese zugrunde, daß eine minimale Implementierung eines Software-Systems nur aus Funktionsaufrufen und Ein-/Ausgabeparametern besteht, so ist diese bei Halstead durch das potentielle Volumen V^* definiert. Hierbei steht η_2^* für die Anzahl der Parameter, wobei der zweite Summand in der Klammer, aus dem Funktions- bzw. Prozedurnamen sowie einem Trennzeichen, daß zwischen dem Funktionsnamen und den Parametern steht, besteht.

Aus dem Verhältnis des potentiellen Volumens zum Volumen eines Software-Systems, bestimmt man dessen 'Programmebene' E und beschreibt somit gewisse Effektivitätseigenschaften. Die Metrik besitzt den Wert 1, wenn das Software-System auf höchstmöglicher Ebene geschrieben wurde (d.h. bei gewünschter Funktionalität, minimale Größe besitzt).

Ausgehend von der Annahme, daß mit wachsender Programmgröße und fallender Programmebene der Aufwand einer Implementierung eines Software-Systems steigt, definiert Halstead diesen, durch das Verhältnis von Programmvolumen zu Programmebene. Die erforderliche Zeitdauer der Implementierung T wird dann mit der sogenannten 'Stroudschen Zahl', ins Verhältnis gesetzt. Die Stroudsche Zahl gibt die Anzahl elementarer geistiger Vergleiche an, die ein Mensch, physisch, pro Sekunde durchführen kann. In der Programmierpraxis hat sich hier ein Wert von 18 als realistisch erwiesen.

Neben den hier angesprochenen Metriken existieren noch weitere Metriken und Alternativen, auf die hier nicht weiter eingegangen werden soll. Die von Halstead definierten Metriken besitzen den Vorteil, daß sie einfach zu ermitteln und zu berechnen sind, für alle Programmiersprachen eingesetzt werden können und als größtenteils signifikant eingestuft werden. So korrelieren beispielsweise η_1 und η_2 stark mit der Programmgröße und der Fehlerrate, d.h. daß mit zunehmender Programmgröße auch ein

entsprechender Anstieg von η_1 und η_2 zu verzeichnen war. Andere Metriken wie das potentielle Volumen V^* oder die Programmebene E scheinen nicht darauf hinzudeuten, das zu messen, was sie scheinbar vorgeben zu messen. Ein grundlegendes Problem besteht auch in den Mehrdeutigkeiten des Meßansatzes. Dies betrifft z.B. die Klassifikationsregeln für Operanden und Operatoren und wirkt sich auf alle weiteren Metriken aus. Dieser Nachteil wurde bereits bei der Betrachtung der Basismetriken angesprochen. Auch werden hier ausschließlich Implementierungsaspekte betrachtet.

3.1.3.5 Zyklomatische Komplexität nach McCabe

Eine Gruppe von Metriken, die auf der Anzahl der Entscheidungen in einem Software-System beruhen, werden u.a. als logische Strukturmetriken bezeichnet. Das wohl bekannteste Maß dieser Gruppe ist das Komplexitätsmaß von McCabe [McCabe 76]. Von ihm wurde zur Messung der strukturellen Komplexität von Software-Systemen, eine Metrik vorgeschlagen, die als quantitative Basis für die Abschätzung von Test- und Wartungsaufwand eines Software-Systems dient. Ausgehend von gerichteten Kontrollflußgraphen, welche die logische Struktur eines Software-Systems widerspiegeln, wird die sogenannte zyklomatische Zahl ermittelt.

Die zyklomatische Zahl $v(G)$ eines Graphen $G(\mathcal{N},\mathcal{E})$, mit n Knoten, e Kanten und p Zusammenhangskomponenten, wird mit Hilfe der Formel

$$v(G) = e - n + 2p\,,$$

ermittelt. Mit der zyklomatischen Komplexität wird die Anzahl der Pfade bestimmt, die mindestens notwendig sind, um alle möglichen Pfade durch ein Modul, mit Hilfe der Linearkombination zu berechnen (Basispfade). Graphen, in denen jeder Knoten von jedem anderen Knoten aus erreicht werden kann, werden als streng zusammenhängend bezeichnet und spielen bei der Definition der zyklomatischen Komplexität eine wichtige Rolle. Eine Zusammenhangskomponente oder auch verbundene Komponente, ist hierbei ein Untergraph, in dem jeder Knoten von jedem anderen Knoten erreichbar ist. So besteht ein Kontrollflußgraph eines Software-Systems ohne Prozeduren oder Funktionen, aus einer verbundenen Komponente (p=1). Ein Software-System, das aus mehreren Prozeduren oder Funktionen besteht und jede durch einen eigenen Untergraph dargestellt wird, hat somit eine entsprechende Anzahl verbundener Komponenten.

Reduziert man einen Graphen, in dem man alle Untergraphen entfernt die nur einen Ein- und einen Ausgangsknoten haben (primitive Kontrollstrukturen der strukturellen Programmierung: Sequenz, If, Case, While, Repeat), so bezeichnet man die zyklomatische Komplexität des reduzierten Graphen G' als essentielle Komplexität:

$$ev(G) = v(G') = v - n\,.$$

Diese drückt den Grad der Unstrukturiertheit eines Graphen bzw. Moduls aus und wird mittels Differenz von zyklomatischer Komplexität, des vollständigen Graphen v und der minimalen Anzahl von Untergraphen n ermittelt. Die essentielle Komplexität sollte für strukturierte Programme immer den Wert 1 haben, da nach einer vollständigen Reduzierung eines Graphen, nur eine Sequenz übrig bleiben sollte. McCabe formuliert in diesem Zusammenhang vier Fälle, die zu unstrukturierten

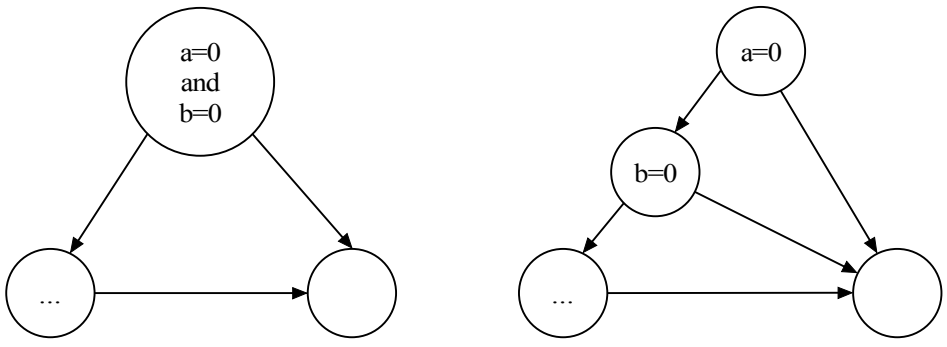
Programmen führen: Aus- und Einsprung aus einer oder in eine Schleife sowie Aus- und Einsprung aus einer oder in eine Entscheidung. Wobei er zeigt, daß jeweils zwei der aufgeführten Fälle zusammen auftreten und die zyklomatische Zahl immer um einen Faktor drei erhöht wird, wenn einer der genannten Fälle auftritt.

Ist die zyklomatische Zahl einer Systemkomponente bestimmt, so gibt McCabe zusätzlich eine Strategie zum Testen dieser Komponente an. Ist die Differenz $v - ac$, wobei v die zyklomatische Komplexität der Systemkomponente ist und ac die Anzahl der getesteten Pfade bezeichnet, positiv, so können nach McCabe folgende drei Situationen gegeben sein:

- Es müssen weitere Tests durchgeführt werden;
- Die Komplexität des Kontrollgraphen für die Systemkomponente oder das Software-System kann zu $v - ac$ reduziert werden;
- Die Komplexität von Teilen einer Systemkomponente oder des Software-Systems kann durch das Einfügen von Code vermindert werden.

Ein großer Vorteil der McCabe-Metrik ist zunächst in deren einfachen Berechnung zu sehen. So kann man beispielsweise die zyklomatische Zahl eines Kontrollflußgraphen, der nur aus einer einzigen Komponente besteht, allein durch Abzählen der Bedingungen ermitteln. Die zyklomatische Zahl ist dann immer um den Wert 1 größer, als die Anzahl der Bedingungen. Zudem weist sie eine hohe Korrelation zwischen der Anzahl der Entscheidungen einer Komponente, der Komponenten-Komplexität und deren Fehlerhäufigkeit auf. Darüber hinaus hat sie den Vorteil, daß man über die Anzahl linear unabhängiger Programmpfade, eine minimale Anzahl von Testfällen finden kann.

Neben den betrachteten Vorteilen, existieren aber auch gewisse Schwachpunkte der McCabe-Metrik. Ein Punkt betrifft beispielsweise die Behandlung von Entscheidungen mit Mehrfachbedingungen. So existieren bei der Anweisung ’if a=0 and b=0 then...’ zwei Möglichkeiten der graphischen Darstellung:



Fall a) Fall b)

Abbildung 17: Behandlung von Entscheidungen mit Mehrfachbedingungen

Berechnet man im Fall a) die zyklomatische Komplexität, so erhält man den Wert 2, im Fall b) den Wert 3. Für McCabe sind zusammengesetzte Entscheidungen komplexer, als Entscheidungen mit einer einzigen Bedingung und entscheidet sich somit für die Behandlung über Variante b). Meyers [Meyers 77] führt in seinen Arbeiten jedoch an, daß die drei Entscheidungen: ’if (x=0) then...else...’, ’if (x=0) and (y<1) then...else...’, ’if (x=0) then if (y<1) then...else...else...’, nach der Interpretation von McCabe die Werte 2,3,3 hätten und nach seiner Meinung so nicht korrekt ist, da die dritte Anweisung

komplexer ist als die Zweite. Für die zusammengesetzte Entscheidung in der zweiten if-Anweisung würde man die Werte 2,2,3 erhalten. Dies steht nach Meyers aber im Widerspruch zu der Auffassung, daß die zweite Entscheidung komplexer ist, als die Erste. Somit schlägt er vor, die zyklomatische Komplexität als Intervall anzugeben, dessen Grenzen durch die Anzahl der Entscheidungen (untere Grenze) und die Anzahl der Bedingungen (obere Grenze) gegeben sind. Für die erste Entscheidung ergebe sich dann der Wert (2:2), für die zweite der Wert (2:3) und für die dritte (3:3).

Eine weitere Schwachstelle besteht in der Tatsache, daß die Komplexität der Knoten selbst nicht berücksichtigt wird. So ist jeder Knoten, mit Ausnahme der Knoten von Entscheidungen, nach McCabe ein sequentielles Code-Segment mit jeweils einem Eingangs- und einem Ausgangsknoten. Das hat zur Folge, daß keine Unterscheidung zwischen Knoten, die eine Anweisung oder eine Folge von Anweisungen enthalten, stattfindet. Hansen [Hansen 78] schlägt vor, als weitere Maßzahl die Anzahl der Operatoren mit anzugeben, da ein Software-System mit mehr Operatoren, sowohl größer als auch komplexer ist.

Neben diesen exemplarischen Beispielen für Schwachpunkte der McCabe-Metrik, existieren eine Reihe weiterer Kritikpunkte. So werden nach [Balzert 98] unterschiedliche Programmerkmale zu stark vereinfacht und das Quellprogramm wird als zentrales Meßobjekt zu stark überbetont. Außerdem wird nur das Programmgerüst und nicht die Komplexität einzelner und verschiedener Anweisungen berücksichtigt. Die aufgeführten Nachteile haben dazu geführt, daß zahlreiche Weiterentwicklungen der McCabe-Metrik stattgefunden haben, bzw. diese den Ausgangspunkt für weitere Metrikentwicklungen bildete.

Roth schlägt zur Berechung der zyklomatischen Komplexität beispielsweise vor, jede Komponente entsprechend ihrer Aufrufe in verschiedenen Pfaden, in die Berechung einfließen zu lassen. D.h. jeder Aufruf einer Systemkomponente in einem neuen Pfad, erhöht die Anzahl der Knoten und Kanten. Die Zahl p für die Zusammenhangskomponenten, entspricht dann der Anzahl der Komponentenaufrufe. Adamov [Adamov 85] schlägt in diesem Zusammengang vor, für jeden Aufruf einer Komponente, außer dem ersten, die zyklomatische Komplexität um zwei zu erhöhen. Grund hierfür ist die Entwicklung eines Programmkontroll-Graphen (siehe Kapitel 3.1.3.8) zu einem streng zusammenhängenden Graphen, wenn der Endknoten des Graphen, mit dessen Anfangsknoten verbunden wird und bei jedem Komponentenaufruf ein Ein- und Ausgangspfeil hinzukommt. Beide Ansätze erhöhen jedoch die zyklomatische Komplexität, sodaß die ursprüngliche Interpretation, als Anzahl minimaler Pfade, nicht mehr gültig ist.

3.1.3.6 'Segment global usage pair '– Metrik

Die 'Segment global usage pair'-Metrik gehört zur Gruppe der Datenstruktur-Metriken, welche die Zugriffe auf Daten und Datenstrukturen in einem Software-System analysieren und so die Komplexität der Struktur des Software-Systems oder dessen Komponenten messen. Mit dieser Metrik wird die Anzahl der Lese- oder Schreibzugriffe von Komponenten, auf globale Variablen, gemessen [Basili 80].

Die Messung von Zugriffen auf globale Variablen erfolgt durch Paarbildung *(P,R)* (Segment-global usage pair), wobei *P* eine Komponente ist, die auf die globale Variable *R* zugreift. Die Menge aller in einem Software-System auftretenden 'Segment-global usage pairs', bezeichnet man als AUP (actual

usage pair). Sind in einem Software-System alle globalen Variablen einer jeden Komponente bekannt und alle Variablen R_i, jeweils mit Komponenten P_i, durch Bildung sogenannter potentieller Benutzungspaare (P_i, R_i) verknüpft, so bilden diese die Menge aller potentiellen Benutzungspaare (PUP, possible usage pair). Das Maß für die relative Häufigkeit der Verwendung globaler Variablen (RUP, relative usage pair), ist das Verhältnis der Anzahl der aktuellen zu den potentiellen Benutzungspaaren $RUP = AUP / PUP$.

3.1.3.7 Datenbindungsmetrik

Die Datenbindungsmetrik ist wie die 'Segment-global usage pair'-Metrik, eine Datenstruktur-Metrik und bestimmt die Bindung zwischen zwei Komponenten über eine globale Variable [Basili 80]. Diese Bindung wird durch ein Tripel (P, R, Q) beschrieben und als 'Segment-global-segment data binding' bezeichnet. Folgende Bedingungen müssen für ein Tripel gelten: in der Komponente P wird die globale Variable R verändert, in der Komponente Q wird auf die Variable R zugegriffen und die Komponenten P und Q dürfen nicht gleich sein, d.h. $P \neq Q$.

Die aktuelle Datenbindung (ADB, actual data binding) ist die Menge aller existierenden Datenbindungen im Software-System. Die Menge der potentiellen Datenbindungen beinhaltet alle möglichen Datenbindungen (PDB, possible data binding). Das Verhältnis der Anzahl der existierenden Datenbindungen, zu der Anzahl der möglichen Datenbindungen, wird als relative Datenbindung (RDB, relative data binding) bezeichnet und beschreibt den Informationsaustausch zwischen den Komponenten eines Software-Systems über globale Variablen $RDB = ADB / PDB$.

Die 'Segment-global usage pair' Metrik sowie die Datenbindungsmetrik, als Vertreter der Datenstruktur-Metriken, berücksichtigen jeweils nur einen bestimmten Aspekt, der, ein Software-System beeinflussenden, Komplexität und sind daher allein nicht ausreichend, um genaue Aussagen darüber zu machen. So werden z.B. Abschätzungen über die Komponentengröße eines Software-Systems oder den Aufrufbeziehungen zwischen diesen, nicht berücksichtigt. Die Datenstruktur-Metriken stellen jedoch eine sinnvolle Ergänzung zu anderen Komplexitätsmaßen dar, wie z.B. der zyklomatischen Komplexität von McCabe.

3.1.3.8 Komplexitätsmaß nach Adamov

Ausgehend von der Erkenntnis, daß die Komplexität eines Software-Systems nicht nur von einem Merkmal beeinflußt wird und verschiedene Aspekte bzw. Einflußfaktoren zu berücksichtigen sind, hat Adamov ein Komplexitätsmaß entwickelt, welches den Versuch unternimmt, eine Vielzahl der Einflüsse in das Maß einfließen zu lassen. Nach Adamov wird die Gesamtkomplexität eines Software-Systems im Wesentlichen durch die Komplexität der Komponenten und deren Beziehungen untereinander bestimmt.

Sein Komplexitätsmaß bezieht sich in erster Linie auf Software-Systeme, mit streng hierarchischer Struktur, d.h. mit einer Art Ebenen- oder Schichtenarchitektur, in der die Systemkomponenten (Prozeduren) einer Schicht, nur auf Komponenten niedrigerer Schichten zugreifen können. Die Darstellung der Systemstruktur erfolgt mit Hilfe sogenannter Prozeduraufruf-Graphen, in dem, aufgrund der Bedingung der strengen Hierarchie, keine Rückpfeile erlaubt sind.

Betrachtet man zunächst die Komplexität der Systemkomponenten, so unterscheidet Adamov hier folgende vier Aspekte: interne Komplexität, Positionskomplexität, nach innen orientierte Schnittstelle und nach außen orientierte Schnittstellenkomplexität.

Die interne Komplexität wird nach Adamov durch die Komplexität des Kontrollflusses bestimmt. Hierbei kann z.B. die Ermittlung der zyklomatischen Zahl nach McCabe ein brauchbares Maß liefern.

Die Postitionskomplexität wird durch die Ebene bestimmt, in der sich der Prozeduraufrufgraph befindet. Hierbei legt man die Annahme zugrunde, daß streng hierarchische Systeme auf den oberen Ebenen, wenige komplexe, und auf den unteren Ebenen, viele einfache Systemkomponenten besitzt (Pyramidenstruktur). Auch haben die Abstände zwischen den Komponenten, bzw. sich aufrufenden Prozeduren, Einfluß auf die Komplexität des Systems, da sie die Abhängigkeit der Komponenten von der Systemstruktur widerspiegeln. So ist beispielsweise bei zunehmenden Abstand (Anzahl der zwischenliegenden Schichten) der beteiligten Komponenten, die Abhängigkeit der aufrufenden Komponente zur Umgebung immer größer, da nicht nur Komponenten einer direkt benachbarten Schicht aufgerufen werden.

Die nach innen orientierte Schnittstellenkomplexität, wird durch die Anzahl der Lesezugriffe auf die Daten der aufgerufenen Systemkomponente sowie deren Rückgabewerte bestimmt, und entsprechend gewichtet. Die Gewichtungsfaktoren bilden sich aus dem Abstand der Rufenden, zur aufgerufenen Systemkomponente, im Systemstrukturgraphen. Die nach außen orientierte Schnittstellenkomplexität ergibt sich, analog zur nach innen orientierten Schnittstellenkomplexität, aus den übergebenen Parametern und den Schreibzugriffen auf Datenstrukturen anderer Systemkomponenten, mit entsprechender Gewichtung. Um die Schnittstellenkomplexität, bestehend aus: nach innen- und außen orientierter Schnittstellenkomplexität, zu bestimmen, erweitert Adamov den Prozeduraufrufgraphen zum Programmaufrufgraphen, da die globalen Daten und Datenstrukturen im Prozeduraufrufgraphen nicht enthalten sind. Auf der untersten Ebene des Programmaufrufgraphen befinden sich die Datenstrukturen.

Der Vorteil des Komplexitätsmaßes von Adamov ist, daß es im Vergleich zu anderen Maßen, welche jeweils nur einen Faktor beachten, die entscheidenden Faktoren der Komplexität von Systemen berücksichtigt. Ein Nachteil ist jedoch darin zu sehen, daß dieses Maß nur für streng hierarchische Systeme entwickelt wurde und somit in seiner Anwendung, auf diese Systemarchitektur beschränkt bleibt.

3.1.4 Produkt-Metriken für objektorientierte Software-Systeme

Analog zu den traditionellen Produktmetriken für die Vermessung prozeduraler Software-Systeme, werden in diesem Abschnitt Metriken für objektorientierte Software-Systeme betrachtet.. Software-Metriken im objektorientierten Bereich, haben zum Teil andere Eigenschaften, als klassische Metriken. Unabhängig davon werden Metriken für prozedural, strukturierte Systeme, teilweise unverändert oder, in Teilen modifiziert, übernommen. Auf der Grundlage von Überlegungen zur Übertragbarkeit klassischer Metriken, auf den objektorientierten Bereich, werden, für die praktische Verwendung vorgeschlagene, System- und Komponentenmetriken, vorgestellt.

3.1.4.1 Übertragbarkeit klassischer Produkt-Metriken auf OO-Software-Systeme

Eine wichtige Fragestellung ist, ob die klassischen Metriken für den objektorientierten Bereich übernommen werden können, oder ob neue Metriken definiert werden müssen.

Bindungsmetriken und Analyse der Bindungsart in objektorientierten Software-Systemen

Neben den vorgestellten, traditionellen Metriken für prozedurale Systemkomponenten (strukturelle Komplexität), existieren eine Reihe neuerer oder modifizierter Produktmetriken für objektorientierte Systemkomponenten. Hierbei wurden die meisten der traditionellen Metriken in unveränderter oder modifizierter Form, aus der Vermessung von prozeduralen Systemkomponenten übernommen. Auch findet man hier, wie bei den Metriken zur Vermessung der strukturellen Komplexität von Systemkomponenten, die bereits aufgeführte Klasseneinteilung. Jedoch erweitert um zentrale, objektorientierte Konzepte, wie Klasse und Vererbung. Die Bindung wird hinsichtlich abstrakter Datenstrukturen, Klassen, Vererbungsstrukturen und Objekten untersucht. Vorschläge zur Festlegung von entsprechenden Bindungsarten für die semantische Bindung, befinden sich noch im Forschungsstadium und sind noch nicht einheitlich definiert [Balzert 98].

Kopplungsmetriken und Analyse der Kopplungsart in objektorientierten Software-Systemen

Auf intermodularer Ebene müssen Besonderheiten des objektorientierten Paradigmas, wie Kopplung von Klassen durch Vererbungsgraphen oder Kopplung von Operationen durch Assoziationen und Aggregationen sowie temporärer Botschaftswege berücksichtigt werden. Auch muß hierbei die Vererbung unter der Sichtweise ’Vererbung als Kopplung’ und Vererbung als Bindung’ unterschieden werden. Wobei die zweite Alternative dann einer externen Bindungsmetrik (intramodulare Ebene) entspricht und der erste Fall im Bereich der Kopplungsmetriken, und somit auf intermodularer Ebene anzusiedeln ist. Auch die Klassifizierung von Kopplungsarten zwischen Datenabstraktionen, Klassen und Objekten befindet sich noch im Forschungsstadium und ist ebenfalls noch nicht einheitlich definiert [Balzert 98].

Metriken für Komponenten

Auf intramodularer Ebene und dem Gebrauch von Umfangsmetriken, der prozeduralen Komplexität, für objektorientierte Systemkomponenten, stellt sich primär die Frage, nach der Einbeziehung von Vererbung und Polimorphismus in die Zeilenzählung. Neben diesen Metriken sind spezielle Maße für die Breite und Höhe von Vererbungshierarchien, die Anzahl der Klassen, die eine spezielle Operation erben, den Anteil wiederverwendeter Systemkomponenten, die Anzahl der Objekt- und Klassenattribute sowie die Anzahl der Objekt- und Klassenoperationen notwendig.

Logische Strukturmetriken für objektorientierte Komponenten wurden bisher in unveränderter Form übernommen. Betrachtet man die Anwendung der zyklomatischen Komplexität, so stellt sich heraus, daß die Kontrollflußkomplexität objektorientierter Operationen in der Regel gering ist (V(G)=1) und deshalb die zyklomatische Zahl, als Maßzahl für die Komplexität einer Klasse, nur bedingt geeignet ist [Balzert 98].

In [Tegarden 92] wird die Verwendung der LOC-Metrik, die Halstead-Metriken und die zyklomatische Komplexität nach McCabe untersucht. Hierbei wird die Ansicht vertreten, daß diese

Metriken im objektorientierten Bereich sinnvoll angewendet werden können. Der Nachweis wird über zahlreiche Beispielsysteme erbracht, die sich in Polymorphismus und Vererbung voneinander unterscheiden. Die Duplizierung von Programmteilen in prozeduralen Software-Systemen, erhöhte die Meßwerte bei klassischen Metriken, infolge des fehlenden Polymorphismus und der Vererbung. Dies steht im Einklang mit der Forderung nach reduzierter Komplexität, durch Einführung objektorientierter Konzepte, wie Polymorphismus oder Vererbung. Somit haben sich die untersuchten, klassischen Metriken als bedeutsam für objektorientierte Programme erwiesen. Trotzdem ist die Definition spezieller Metriken für diesen Bereich notwendig.

In [Lake 94] werden mit Hilfe von Korrelationsberechungen Aussagen zur Übertragbarkeit klassischer Metriken getroffen. Sie kommen zu dem Ergebnis, daß mit Hilfe klassischer Metriken Aussagen über die Softwarequalität objektorientierter Software-Systeme gewonnen werden können. Darüber hinaus gebe es aber weitere Einflußgrößen, die nur mit objektorientierten Metriken beschreibbar sind.

Bei [Moreau 89] wird die Übertragbarkeit klassischer Metriken verneint, da diese nicht die Strukturelemente des objektorientierte Modells betrachten. Es sei in diesem Bereich nicht möglich, die Gesamtkomplexität mit der Summe der Teilkomplexitäten zu bestimmen. Werden klassische Metriken, wie z.B. die zyklomatische Komplexität nach McCabe, auf objektorientierte Software-Systeme angewandt, dann haben sie nur für Methoden in einzelnen Objekten, ihre Anwendungsberechtigung. Auch in [Abreu 94] wird die Übertragbarkeit der zyklomatischen Komplexität nach McCabe angezweifelt und sei nur auf Methodenebene direkt anwendbar, da nur dort ein Kontrollflußgraph im Sinne von McCabe konstruierbar ist. Andere Autoren befürworten die Anwendung der zyklomatischen Komplexität, auch auf Klassenhierarchien und schlagen hierfür eigene Maße vor [Fetcke 95]. Chidamber et.al.[Chidamber 94] machen den Vorschlag, das Maß von McCabe als Gewichtung für die Berechung der WMC-Metrik einzusetzen und somit für die Verwendung auf Klassen- oder Systemniveau nutzbar zu machen.

Eine interne Bindungsmetrik für objektorientierte Systemkomponenten bestimmt die Anzahl, der durch die Operationen einer Klasse, gemeinsam benutzten Objektattribute. (LCOM, lack of cohesion in methods). Diese ist bislang schlecht definiert und bereits mehrmals modifiziert worden und daher als Qualitätsindikator ebenfalls nur bedingt geeignet [Balzert 98].

Die Frage nach der Übertragbarkeit klassischer Produkt-Metriken ist umstritten. Die Tendenz geht eher in Richtung Anpassung vorhandener klassischer Maße. Diese haben den Vorteil, von Anwendern und Forschern wohlverstanden zu sein und ruhen auf empirischen Erfahrungen, die diese Metriken zusätzlich unterstützen. Auch wird versucht, klassische Metriken durch Akkumulation in neue Vorschläge für den objektorientierten Bereich einzubinden (z.B. McCabes zyklomatische Komplexität als Gewichtung für die WMC-Metrik). Insgesamt findet man ca. 150 Maßvorschläge für objektorientierte Software-Systeme [Fetcke 95] [Zuse 98]. Erste empirische Untersuchungen zeigen die Eignung einiger der vorgeschlagenen Metriken, als Qualitätsindikatoren. Ein Nachteil ist jedoch, daß die Metriken bisher unzureichend validiert sind, bzw. die Ziele, für die diese Metriken entwickelt wurden, in der Regel nicht angegeben sind. Somit fehlt der Gültigkeitsnachweis, daß eine Metrik auch tatsächlich das Softwareattribut mißt, welches es vorgibt zu messen (interne Validierung), bzw. der empirische Nachweis eines Zusammenhanges, zwischen dem Softwaremaß und einer externen Variablen, wie z.B. den Entwicklungskosten (externe Validierung). Eine Auswahl vorgeschlagener Metriken stellt sich dabei als signifikant für die Vermessung, der Qualität einer Klasse heraus [Balzert

98]. Aus dieser Auswahl werden im folgenden diejenigen vorgestellt, die in das Portierungsmaß mit eingehen sollen.

3.1.4.2 DIT (Depth of Inhertiance Tree)

Ein Maß für die Anzahl der Vorfahren einer Klasse liefert die DIT-Metrik (Depth of Inheritance Tree) und wird von Shyam Chidamber und Chris Kemerer [Chidamber 94] wie folgt definiert:

$DIT(C) :=$ "Länge des maximalen Weges in der Klassenhierarchie von der Klasse C bis zur Wurzel"

Hierbei wird die Tiefe in der Klassenhierarchie, von der Wurzel an, mit 0 gezählt und bei der Mehrfachvererbung der maximale Weg. Das Maß ist damit eine Kenngröße für die Anzahl Vorgänger, die Einfluß auf die betrachtete Klasse haben können. Um die Eigenschaften einer Klasse vollständig zu ermitteln, muß die Vererbungshierarchie bis zur Wurzelklasse durchsucht werden, da eine Basisklasse auch die Eigenschaften für abgeleitete Klassen definiert. Bei der Verwendung von Programmiersprachen, in denen Klassen stets als Spezialisierung von Systemklassen definiert werden (z.B. in Smalltalk 'Object', als generelle Oberklasse), ergeben sich allgemein höhere Meßwerte für die Klassen. Dies gilt auch bei der Verwendung von Klassenbibliotheken. Es wird empfohlen, die Zählung unterhalb der Klassenbibliotheken zu beginnen. Man kann sagen, je größer der Wert von DIT für eine Klasse C ist, desto größer ist die Wahrscheinlichkeit des Auftretens von Fehlern (hohe Signifikanz) [Balzert 98].

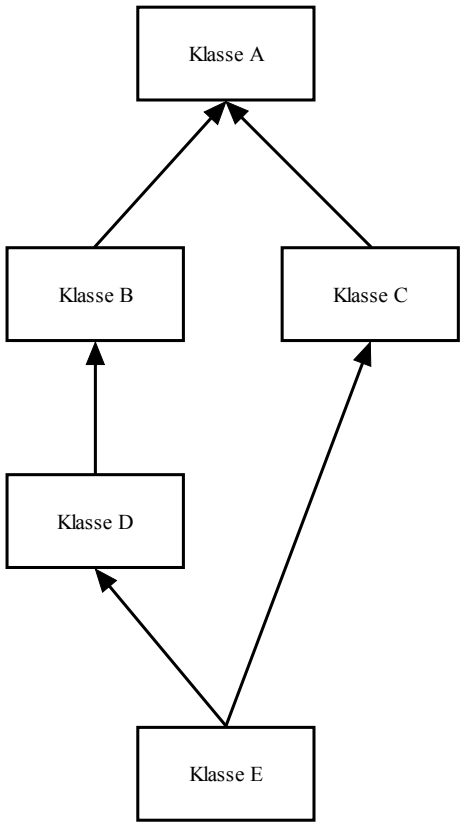


Abbildung 18: Beispiel für das Maß DIT

Ein Beispiel soll die Verwendung der DIT-Metrik veranschaulichen. Hierbei ist die Wurzel der Hierarchie die Klasse A:

$$DIT(A) = 0.$$

Für die Klassen B und C ergeben sich:

$$DIT(B) = DIT(C) = 1.$$

Demzufolge hat die Klasse E den Wert

$$DIT(E) = 3,$$

weil der maximale Weg von Klasse A zu Klasse E, über die Klassen B und D verläuft.

3.1.4.3 CBO (Coupling between Object Classes)

Eine weitere, ebenfalls von Shyam Chidamber und Chris Kemerer [Chidamber 94], definierte OO-Metrik ist CBO (Coupling between Object Classes). Sie wird wie folgt definiert:

$CBO(C) :=$ "Anzahl der Klassen, mit denen die Klasse C gekoppelt ist."

Zwei Klassen sind hierbei miteinander gekoppelt, wenn Methoden der einen Klasse, Methoden oder Instanzvariablen der anderen Klasse aufrufen bzw. verwenden. Die CBO-Metrik drückt also die Anzahl der Klassen aus, die in einer gegenseitigen Benutzt-Beziehung zur Klasse C stehen. Je mehr Klassen eines Software-Systems sich gegenseitig benutzen bzw. gekoppelt sind, um so empfindlicher reagiert es auf Änderungen, wodurch der Wartungsaufwand und die Fehleranfälligkeit signifikant steigt. Exzessive Kopplung zwischen Klassen verstößt gegen die Prinzipien der Modularisierung und verhindert deren Wiederverwendbarkeit.

Auch hier soll ein kurzes Beispiel die Verwendung der CBO-Metrik verdeutlichen.

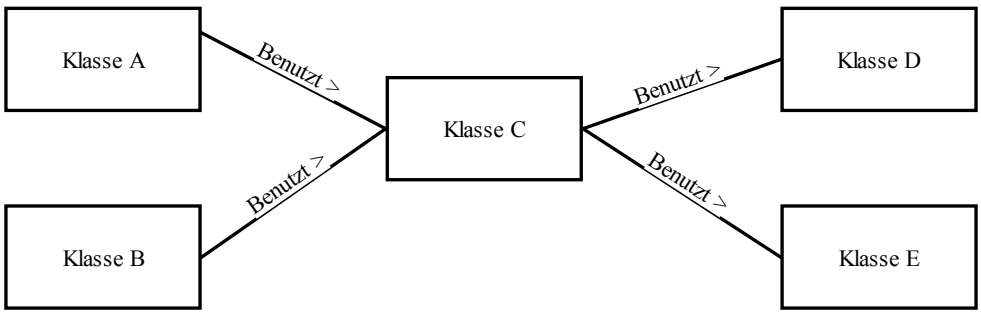


Abbildung 19: Beispiel für das Maß CBO

In dem Beispiel ist die Klasse C mit insgesamt vier Klassen gekoppelt. Sie wird einerseits von den Klassen A und B benutzt und nutzt ihrerseits Methoden oder Instanzvariablen der Klassen D und E. Danach ergibt sich für die Klasse C der Wert:

$$CBO(C) = 4 .$$

3.1.4.4 RFC (Response for a Class)

[Chidamber 94] definieren die RFC-Metrik (Response for a Class) wie folgt:

$RFC(C) := |$ Responce Set $| = | RS_C |$,
mit $RS_C :=$ Menge aller Methoden der Klasse C, vereinigt mit der Menge aller Methoden der Klassen, die von den Methoden der Klasse C aufgerufen werden. Wobei die zweite Menge eine Teilmenge aller Methoden der von C benutzten Klassen ist.

Die RFC-Metrik ist somit ein Maß für die Anzahl der eigenen Operationen einer Klasse C, erweitert um die Anzahl der internen und externen Aufrufe. Je größer die Zahl der aufrufbaren Methoden einer Klasse ist, desto höher ist deren Komplexität und um so schwieriger werden das Testen und Debugging. Die Metrik gibt somit auch eine obere Schranke für die Anzahl der Testfälle an, die durch die möglichen Methodenaufrufe notwendig werden. Je größer der Wert von RFC, desto größer ist die Fehlerwahrscheinlichkeit.

Ein Beispiel soll zeigen, wie die RFC-Metrik angewendet wird.

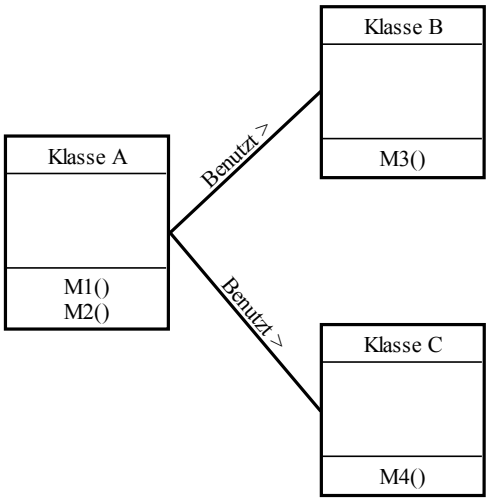


Abbildung 20: Beispiel für das Maß RFC

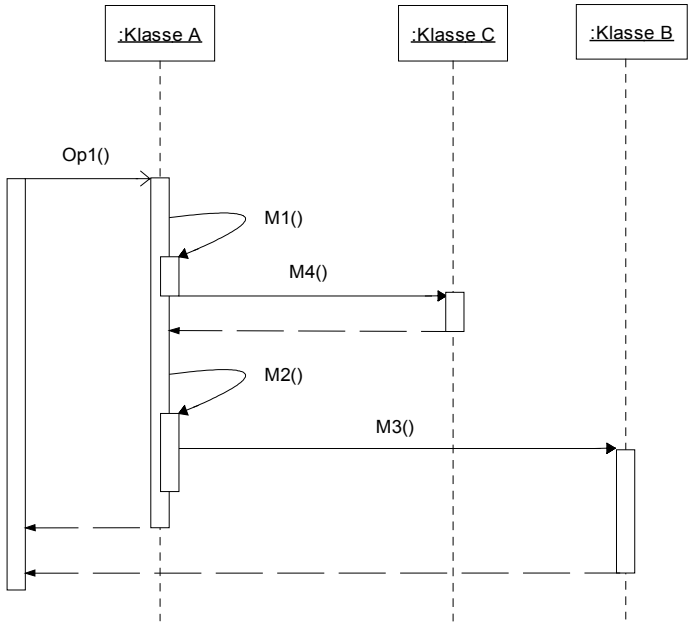


Abbildung 21: Sequenzdiagramm für das RFC-Beispiel

Nimmt man an, daß die Methode M1 der Klasse A, die Methode M5 der Klasse D aufruft und M2 die Methode M4 der Klasse C, so besteht das Response Set für die Klasse A aus:

$$RS_A = \{\text{M1, M2, M3, M4}\}.$$

Der Meßwert bestimmt sich demnach wie folgt:

$$RFC(A) = |\, RS_A \,| = 4.$$

3.1.4.5 WMC (Weighted Methods per Class)

Ebenfalls in [Chidamber 94], ist die Metrik WMC definiert.

$$WMC(C) := \sum_{i=1}^{n} g(M_i),$$

wobei $M_{i=1..n}$ die Methoden der Klasse C sind und $g(M_i)$ die Komplexität bzw. Gewichtungsfaktoren der Methode M_i darstellt. WMC steht aus diesem Grund für eine Familie von Maßen. Die Maße dieser Familie ergeben sich aus dem gewählten Gewichtungsmaß. Für die Gewichte existieren verschiedene Vorschläge. Chidamber et. al. schlagen für die Methodenkomplexität $g(M_i) = 1$ vor, womit WMC gleich der Anzahl der Methoden einer Klasse wird. Dieses Maß wird dann allgemein als NOM (Number of Methods) bezeichnet und ist unabhängig von der Implementierung, bzw. kann bereits in den Phasen der objektorientierten Analyse und dem objektorientierten Design angewendet werden. Gezählt werden nur die Methoden, die lokal in der Klasse C definiert werden. Solche Methoden, die von Oberklassen geerbt werden, werden nicht berücksichtigt. Auch bei dieser Metrik gilt, je größer der Wert von WMC, desto größer ist die Fehlerwahrscheinlichkeit.

Legt man für die Methodenkomplexität, bzw. die Gewichtungsfaktoren $g(M_i) = 1$ zugrunde, läßt sich die Anwendung von WMC beispielsweise so verdeutlichen:

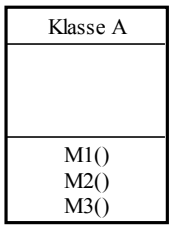


Abbildung 22: Beispiel für das Maß WMC

Eine Klasse A hat drei Methoden M1, M2 und M3. Allgemein gilt dann für die Berechnung:

$$WMC(A) = g(M_1) + g(M_2) + g(M_3),$$

bzw. für die Annahme $g(M_i) = 1$ (d.h. die einfache Form von WMC):

$$WMC\text{-}1(A) = 3$$

Die Autoren schlagen darüber hinaus, als Gewichte, die zyklomatische Komplexität nach McCabe vor. Andere Autoren erscheint dies logisch, denn je mehr Methoden eine Klasse hat, desto komplexer ist die Klasse. Je mehr Kontrollflüsse in den Methoden einer Klasse enthalten sind, um so schwerer sind diese zu verstehen und zu warten.

Folgende Standpunkte werden für die WMC-Metrik vertreten [Fetcke 95]:

- Die Anzahl und Komplexität der Methoden bestimmen den Aufwand für Entwicklung und Wartung;
- Je größer die Anzahl der Klassen, desto größer ist der Einfluß auf abgeleitete Klassen, die diese Methoden dann erben;
- Klassen mit vielen Methoden sind wahrscheinlich anwendungsspezifischer, als Klassen mit wenigen Methoden.

3.2 Metriken zur Portabilitätsanalyse

In diesem Abschnitt geht es im Speziellen um die Darstellung von Metrikvorschlägen, zur Vermessung der Portabilität von Software-Systemen. Die Einordnung von Portabilitäts-Metriken in den Rahmen von Qualitätsmodellen soll zunächst zeigen, in welchem Gesamtzusammenhang sich die vorgestellten Metriken bewegen. Im Anschluß daran werden eine Reihe von Metrikvorschlägen zur Vermessung der Portabilität von Software-Systemen vorgestellt und bewertet. Dieser Abschnitt bildet damit u.a. die Grundlage für die Berechnung einer Portabilitäts-Metrik an einem Windows-basierten Software-System.

3.2.1 Einordnung von Portabilitäts-Metriken

Wie bereits erwähnt, werden Software-Metriken primär für die Vorhersage von Wartungs- und Entwicklungsaufwand und zur Bewertung von Software-Qualität verwendet. Hierbei werden, entsprechend den Anforderungen, Prozeß-Metriken für die Vermessung des dynamischen Verhalten eines Entwicklungsprozesses und Produkt-Metriken, deren Untersuchungsgegenstand das statische Produkt der Softwareentwicklung ist, klassifiziert. Da die allgemeine Definition von Software-Qualität – nach ISO 9126, die Gesamtheit der Merkmale und Merkmalswerte, eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen – für die praktische Anwendung nicht ausreicht, setzt man für die Beschreibung der Qualität von Software-Systemen ein entsprechendes Qualitätsmodell ein. Damit wird der allgemeine Qualitätsbegriff durch Ableiten von Unterbegriffen operationalisiert. Hierbei wird die Software-Qualität allgemein oder bezogen auf einzelne Entwicklungen, durch Qualitätsmerkmale bzw. -faktoren beschrieben – meist aus Benutzersicht –und in einem weiteren Schritt in Teilmerkmale bzw. Kriterien verfeinert – meist softwareorientiert – und entsprechend definiert. Diese Teilmerkmale werden durch Metriken – in diesem Zusammenhang auch als Qualitätsindikatoren bezeichnet – meß- und bewertbar gemacht und zu den Qualitätsmerkmalen in Beziehung gesetzt. Man spricht hierbei von Software-Qualitätsmaßen, als quantitative Skala, mit einer Methode zur Bestimmung des Meßwertes, den eine Metrik bzw. ein Qualitätsindikator für ein bestimmtes Software-Produkt aufweist.

In dem Qualitätsmodell für Software-Produkte der DIN ISO 9126, werden neben den Qualitätsmerkmalen wie Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz und Änderbarkeit, auch die Übertragbarkeit als Merkmal für die Eignung eines Software-Systems, von einer Umgebung in eine andere übertragen zu werden, festgelegt. Für das Qualitätsmerkmal 'Übertragbarkeit' werden in [Balzert 98] die Teilmerkmale 'Anpaßbarkeit' (Möglichkeiten, ein Software-System an verschiedene, festgelegte Umgebungen anzupassen), Installierbarkeit (Aufwand der Installation eines Software-Systems in einer festgelegten Umgebung), Konformität (Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt) und Austauschbarkeit (Möglichkeit oder Aufwand, das Software-System, anstelle eines anderen spezifizierten Software-Systems, in einer Umgebung einzusetzen) vorgeschlagen. Andere Qualitätsmodelle beschreiben, in Bezug auf die Übertragbarkeit von Software-Systemen, ähnliche Teilmerkmale.

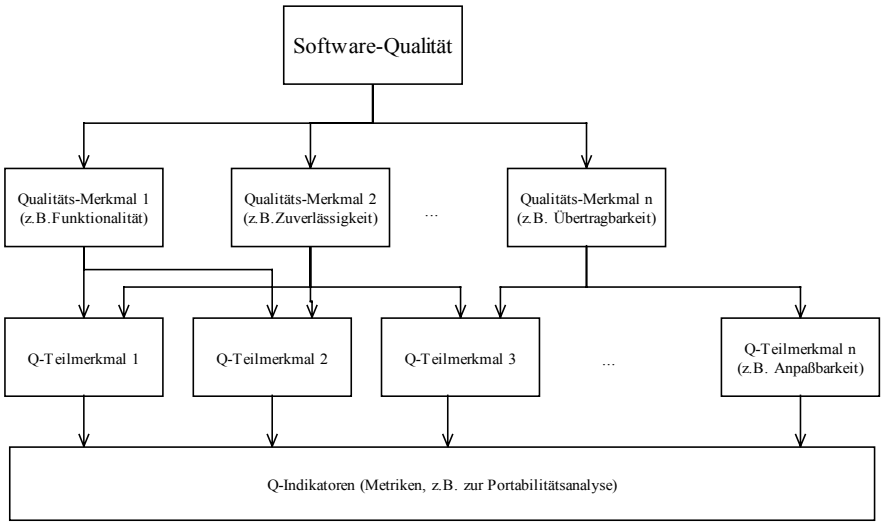


Abbildung 23: Qualitätsmodelle als Anwendungsrahmen für Metriken

Qualitätsmodelle zur Beschreibung der Software-Qualität bilden somit den allgemeinen Rahmen, für die Anwendung von Metriken zur Portabilitätsanalyse. Diese Metriken können dann als Produkt-Metriken klassifiziert werden, da sie ein bestimmtes Qualitätsmerkmal eines Software-Systems messen und damit quantitative Aussagen über die Produktqualität einer Systemkomponente, in diesem Bereich, ermöglichen sollen.

3.2.2 Metrikvorschläge zur Portabilitätsanalyse

Um die Portabilität eines Software-Systems bestimmen zu können, wurden bisher eine Reihe von Metriken vorgeschlagen, die im folgenden näher beschrieben werden sollen.

In dem Portabilitätsmaß von Hommel [Hommel 80] (siehe Kapitel 2.1.1.1) tauchte bereits das 'Maß des Aufwands einer Portierung' P auf:

$$P = \frac{AI}{AI + AP + AA}$$

mit: AI = Aufwand für die erste Implementierung
AP = Aufwand für die Portierung
AA= Aufwand für die Adaptierung.

Ein Software-Produkt galt als portabel, wenn die Bedingung $0,5 \le P \le 1$ erfüllt war.

Gewald et. al. definieren ebenfalls den Grad der Portabilität eines Programms durch verschiedene Aufwandsmaße. Hierbei wird der Begriff Konvertierungs- statt Portierungsaufwand verwendet:

$$P(i) = \frac{D(i) - C(i,m)}{D(i)} \cdot 100\%$$

mit: P = Portabilitätsgrad
D = Entwicklungsaufwand
C = Konvertierungsaufwand
i = zu übertragendes Programm
m = Konvertierungsmethode.

Ein Portabilitätsgrad von 100% bedeutet Kompatibilität. Es bleibt anzumerken, daß Gewald et. al. die ersten Autoren sind, die die Konvertierungs- bzw. Portierungsmethode in Ihrer Definition berücksichtigen. Trotzdem sind die bisher vorgestellten Vorschläge kaum mehr, als die Umsetzung einer Definition in eine mathematische Gleichung.

Ähnliche Vorschläge stammen von Gilb [Gilb 77] , der den Portierungsaufwand mit:

$$P_S = 1 - \frac{\text{Aufwand zur Übertragung von S in die Zielumgebung}}{\text{Entwicklungsaufwand von S für Hostumgebung}}$$

mit: S = Software-System,

bestimmt. Hierbei sollten nach seiner Meinung, eher die Entwicklungskosten in der Zielumgebung der Original-Software im Nenner der Gleichung stehen. Er berechnet ein Beispiel, in dem er die Entwicklungskosten mit 100000 und die Übertragungskosten mit 10000 angibt, womit sich ein Portabilitätsaufwand von $P_S = 0,9$ ergibt und eine Portabilität von 90%. Außerdem trifft er die Annahme, daß der Portierungsaufwand für nicht-triviale Software eine lineare Funktion ist. So erfordert beispielsweise ein Programm mit 100000 Statements, den 100-fachen Aufwand eines Programms mit 1000 Statements.

Alle bisher aufgeführten Vorschläge implizieren, daß der Aufwand einer Portierung im voraus bestimmbar ist und machen keine Angaben darüber, wie dieser Aufwand zustande kommt oder wie er gemessen werden kann. Hinzu kommt die Tatsache, daß der Aufwand einer Portierung mit dem Aufwand der Entwicklung in Beziehung gesetzt wird, der im nachhinein recht präzise ermittelt werden kann.

Auf der Basis eines bereits vorgestellten Prozeßmodells (siehe Abbildung 2 in Kapitel 2.1.4) für die Portierung, definiert Mooney [Mooney 93] den Grad der Portierung ähnlich wie Gilb:

$$DP(su) = 1 - \frac{Cport(su, e2)}{Crdev(req, e2)}$$

mit: $DP(su)$ = Grad der Portabilität eines Software-Systems oder Teilen davon
$Cport$ = Aufwand bzw. Kosten der Portierung eines Software-Systems oder Teilen davon su, in eine Umgebung $e2$
$Crdev$ = Aufwand bzw. Kosten der Neuentwicklung eines Software-Systems oder Teilen davon mit einer bestimmten Spezifikation req, in eine Umgebung $e2$.

Er beschreibt darüber hinaus, wie die einzelnen Maße zustande kommen, indem er für den Aufwand der Portierung eine Berechnungsvorschrift angibt, welche die Einflußgrößen: Modifikationsaufwand, Aufwand für Testen, Debugging und Dokumentation aufsummiert:

$$Cport(su, e2) = Cmod(su, e2) + Cptd(req, e2) + Cpdoc(req, e2).$$

Auch wird formal beschrieben, aus welchen Einflußgrößen sich das Maß des Aufwands für eine Neuentwicklung zusammensetzt. Hierbei geht er davon aus, daß die Phasen der Entwicklung eines Software-Systems, wie Design, Implementierung, Testen und Debugging sowie Dokumentation, zu einem Gesamtmaß aufsummiert werden können und dann in das oben aufgezeigte Verhältnis, den Einflußgrößen des Portierungsaufwands, gesetzt werden können:

$$Crdev(req, e2) = Crdes(req) + Crcod(req, e2) + Crtd(req, e2) + Crdoc(req, e2).$$

Mooney beschreibt, wie der Aufwand einer Portierung zustande kommt, bzw. welche Einflußgrößen hierbei eine Rolle spielen und zeigt, welche Einflußgrößen das Verhältnismaß ‘Grad der Portabilität’ zusätzlich bestimmen. In Beispielen (Computerspiel, Compiler) wird gezeigt, wie der Grad der Portabilität praktisch bestimmt wird. Es werden Annahmen darüber getroffen, welche Werte die Einflußgrößen haben könnten. Es fehlen jedoch konkrete Angaben darüber, wie die einzelnen Einflußgrößen gemessen werden und wie sich diese im Einzelnen zusammensetzen.

Einen praktischeren Vorschlag liefern dagegen McCall et. al. [McCall 77], die eine Reihe von Faktoren innerhalb eines Qualitätsmodells identifizieren, die das Merkmal Portabilität beeinflussen. Ihre Einflußfaktoren setzen sich zusammen aus: Modularität, Selbstbeschreibung, Maschinenunabhängigkeit sowie Software-Systemunabhängigkeit. Zu diesen Faktoren stellen sie dann einige Maße auf. So lautet beispielsweise ein Maß des Faktors ’Software-Systemunabhängigkeit’:

$$M = 1 - \frac{\text{Anzahl Module mit Betriebssystemreferenzen}}{\text{Gesamtzahl der Module}}.$$

Für den Faktor Maschinenabhängigkeit schlagen die Autoren z.B. das binäre Maß:

$$M' = \text{benutzte Programmiersprache existiert auf anderen Maschinen},$$

vor. Es wird verdeutlicht, daß die Portabilität nicht direkt bestimmbar ist, sondern aus verschiedenen meßbaren Faktoren abgeleitet wird. Eine Aggregation zu einem Gesamtmaß ‘Portabilität’ wird nicht durchgeführt.

Einen umfassenderen Ansatz findet man in einer Arbeit von Buschhorn und Kuchnowski [Buschhorn 88], die auf der Basis eines von ihnen aufgestellten Portabilitätsmodells (siehe Abbildung 4, in Kapitel 2.1.4), ebenfalls ein Maß zur Bestimmung der Portabilität eines Software-Systems aufstellen:

$$PORT = (SYS, DOK, KPL)$$
mit: SYS = Systemumgebung bzw. Software-Umgebung
DOK = Dokumentation
KPL = Komplexität.

Die einzelnen Maßfaktoren setzen sich aus meßbaren Einflußgrößen zusammen, die, neben anderen, anhand des Portierungsmodells identifiziert werden. Hierbei wird versucht, ein umfassendes Portierungsmaß aufzustellen und die einzelnen Maßfaktoren, zu einem Gesamtmaß für die Portabilität eines Software-Systems, zusammenzuführen. Es wird der evolutionäre Aspekt des Maßes betont, der dieses durch praktische Verwendung und den daraus resultierenden Erfahrungswerten, weiter anpaßt. Dieses soll dann immer genauere Aussagen bzw. Vorhersagen zum Portierungsaufwand, eines untersuchten Software-Systems, ermöglichen. Erfahrungswerte bei der Anwendung des Maßes in konkreten Portierungsprojekten, werden von den Autoren jedoch nicht angegeben.

Zusammenfassend kann gesagt werden, daß Metriken, zur Bestimmung des Aufwands einer Portierung, in Ansätzen vorhanden sind. Neben der Bestimmung des Portabilitätsgrads eines Software-Systems, die den Aufwand einer Neuentwicklung und den Portierungsaufwand in Beziehung setzen und damit eine Entscheidungshilfe in die eine oder andere Richtung zur Verfügung stellen, wurden Ansätze vorgestellt, die Einflußgrößen der Portabilität von Software-Systemen konkret vermessen und

zu einem Gesamtmaß für den Aufwand einer Portierung aggregieren. Das umfassendste Maß, von Buschhorn und Kuchnowski, wird im folgenden näher betrachtet und bewertet.

3.2.3 Ausgewählte Metriken für die Portabilitätsanalyse

In diesem Abschnitt geht es um die Beschreibung und Bewertung eines Portabilitätsmaßes, welches von Michael Buschhorn und Hans Joachim Kuchnowski in [Buschhorn 88] vorgeschlagen wird. Dieses Maß soll als Grundlage für die Portabilitätsanalyse Windows-basierter Software-Systeme dienen. Es werden Aussagen zur Maßphilosophie, den Maßobjekten, den Maßfaktoren und dem Aufbau des Portierungsmaßes gemacht. Eine abschließende Bewertung, faßt die Einhaltung von Maßanforderungen zusammen.

3.2.3.1 Maßphilosophie

Das zentrale Konzept hinter dem Portierungsmaß ist seine evolutionäre Entwicklung. Das Maß soll Aussagen über die Höhe des Portierungsaufwandes machen, der bei der Portierung eines Software-Systems, von einer Umgebung in eine andere Umgebung, entstehen wird. Hierzu werden das Software-System selbst und Unterlagen zu Unterschieden zwischen der Host- und Zielumgebung, analysiert. Es entstehen damit erste grobe Schätzungen, die jedoch keinesfalls konkrete Aussagen zum eigentlichen Portierungsaufwand beinhalten können. Eine andere Sicht ergibt sich, sobald eine Portierung erfolgreich durchgeführt wurde. Nun können die vorhandenen Informationen in Aufwandsprognosen einfließen und erlauben es mit der Zeit, immer präzisere Aussagen zum Portierungsaufwand von Software-Systemen zu machen. Wird z.B. ein Software-System in eine Umgebung portiert, die über ein anderes Betriebssystem verfügt, so läßt sich nun, da bereits dokumentierte Anpassungsaufwände für einzelne Betriebssystemroutinen vorliegen, der Aufwand, anhand der Zahl vorhandener Betriebssystemaufrufe, realistischer abschätzen. Das Portierungsmaß befindet sich somit in einer Art evolutionärem Prozeß und wird mit seinen Aussagen zum Portierungsaufwand mit wachsenden Erfahrungswerten immer präziser.

Die Autoren beschreiben als Zielgruppe eher solche Benutzer, die häufiger Software-Systeme portieren, wie z.B. Softwarefirmen. Diese können die Möglichkeiten des Maßes besser nutzen, als einmalige oder seltene Benutzer. Aber auch die einmalige Nutzung soll eine ganze Reihe von Informationen bringen, die eine Entscheidung, für oder gegen eine Portierung auf eine breitere Basis stellt.

3.2.3.2 Maßobjekte

Mit dem Maß sollen Software-Systeme oder Systemkomponenten gemessen werden, die in einer höheren Programmiersprache implementiert sind. Hardwarenahe Software-Systeme, wie z.B. Systemsoftware, müssen bei einer Übertragung in eine andere Umgebung oft zu großen Teilen neu geschrieben werden und stellen für die Autoren daher kein Maßobjekt dar. Den Schwerpunkt der Anwendung des Maßes legen sie deshalb auf Anwendungssoftware.

3.2.3.3 Maßfaktoren

Auf der Grundlage des bereits vorgestellten Portierungsmodells (siehe Abbildung 4), werden verschiedene Einflußfaktoren identifiziert, die den Aufwand einer Portierung beeinflussen können:

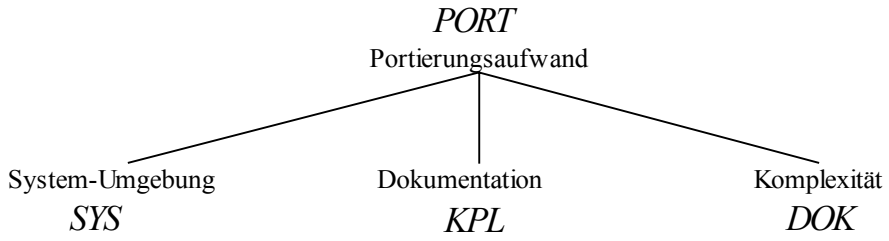
- Unterschiede in den Systemumgebungen der Host- und Zielrechner,
- Komplexität des zu portierenden Software-Systems bzw. von Systemkomponenten,
- Dokumentation,
- Portierungswerkzeuge,
- Erfahrungen und Kenntnisse des Porteurs.

Hierbei wird festgestellt, daß die Faktoren Portierungswerkzeuge sowie Erfahrungen und Kenntnisse des Porteurs, schwer zu messende Einflußgrößen darstellen. Die Güte von Portierungswerkzeugen zeigt sich nach Ansicht der Autoren erst, wenn man sich intensiv in diese eingearbeitet hat und dessen Anwendung ausreichend erprobt hat. Erst dann ist nach ihrer Ansicht eine realistische Beurteilung seiner Möglichkeiten denkbar. Aus diesem Grund wird die Güte von Portierungswerkzeugen nicht direkt gemessen, sondern geht über die Bewertung von Änderungsaufwänden, indirekt in das Maß ein. Mit Erfahrungen und Kenntnisse des Porteurs sind sowohl allgemeine Erfahrungen, wie z.B. Programmiererfahrung als auch spezielle Erfahrungen, im Hinblick auf die Portierung gemeint. Abhängig von diesen Kenntnissen kann, nach Ansicht der Autoren, der zeitliche Portierungsaufwand sehr unterschiedlich sein und stellt einen, insgesamt, subjektiven Einflußfaktor dar. Dieser ist kaum zu erfassen und geht daher nicht in das Portabilitätsmaß ein.

Die anderen Faktoren, wie Systemumgebung, Komplexität und Dokumentation sind hierbei schon von objektiverem Charakter und weitgehend automatisch bestimmbar. Zu den automatisch bestimmbaren Einflußgrößen gehören die Messung der Unterschiede in den Systemumgebungen, die durch geeignete syntaktische Analyse durchgeführt wird und die Vermessung der Komplexität des Software-Systems. Der Faktor Dokumentation ist nach Meinung der Verfasser nicht automatisch bestimmbar, da nicht automatisch ermittelt werden kann, ob die Qualität ausreichend ist, d.h. ob die nötigen Informationen vorhanden, bzw. leicht zu finden sind. Hierzu muß eine entsprechende Code-Inspektion durchgeführt werden, in der das Vorhandensein bestimmter, wichtiger Dokumentationskomponenten untersucht wird. Diese Inspektion kann sehr umfangreich sein und trotzdem nicht gewährleisten, daß die benötigten Informationen auch wirklich vorhanden waren. Dies läßt sich nur nach einer erfolgreichen Portierung feststellen. Auf der anderen Seite kann aber der Einfluß der Dokumentation so groß sein, daß er als einziger Entscheidungsfaktor, für oder gegen eine Portierung, dienen kann. So macht es beispielsweise keinen Sinn, ein Software-System zu portieren, zu dem keine oder nur unzureichende Dokumentation existiert, da es vom Porteur nur sehr schwer zu verstehen und vom Anwender kaum benutz- und erweiterbar ist. Die Autoren gehen daher einen Kompromiß ein, in dem sie die Dokumentation als stark subjektive Einflußgröße in das Maß aufnehmen und diese durch manuelle Code-Inspektion bestimmen lassen, etwa durch ein entsprechendes Code-Review, in Form von vorgegebenen Checklisten, welche die Dokumentation auf das Vorhandensein, wichtiger Dokumentationskomponenten hin überprüfen.

3.2.3.4 Maßaufbau

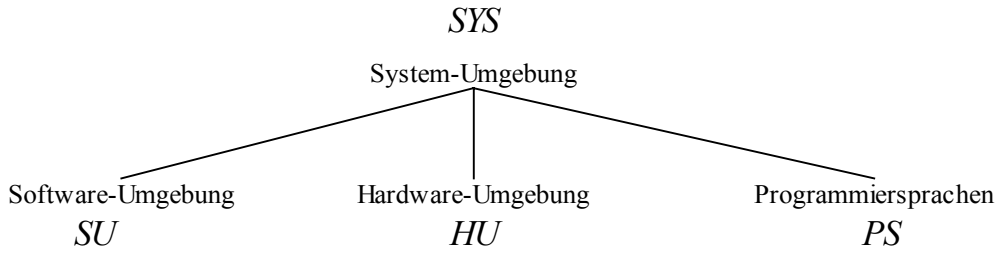
Das betrachtete Maß für die Bestimmung des Portierungsaufwands von Software-Systemen hat folgenden, grundlegenden Aufbau:



Formal:

$$PORT = (SYS,\ KPL,\ DOK)$$

Es folgt der Aufbau für das Teilmaß 'System-Umgebung':



Formal:

$SYS = (((KA_{SYS}, PA_{SYS}), SA_{SYS}, BA_{SYS}), PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}, VSG_{SYS})$ mit:

$\mathrm{KA_{SYS}} = (KA_{BS}, KA_{PS\_GES})$

Namen der BS-Routinen, und der PS-Konstrukte, die nicht angepaßt werden können

$\mathrm{PA_{SYS}} = (PA_{BS}, PA_{PS\_GES})$

Anzahl und Namen der BS-Routinen und PS-Konstrukte, über die während der Messung keine Informationen vorliegen

$\mathrm{SA_{SYS}} = SA_{SU} + SA_{PS\_GES}$

Gesamtzahl aus Aufrufen von SU-Routinen und PS-Konstrukten, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$\mathrm{BA_{SYS}} = BA_{SU} + BA_{PS\_GES}$

Summe aus bewerteten SU-Routinen und PS-Konstrukten, für die ein Aufwand vorliegt

$\text{PAN}_{\text{SYS}} = PAN_{SU} + PAN_{HU}$

Summer der potentiellen Anpassungen aus Software- und Hardware-Umgebung

$\text{KOMP}_{\text{SYS}} = KOMP_{SU} + KOMP_{HU}$

Namen der Komponenten mit Anpassungen an die Software- und Hardware-Umgebung und deren Anzahl

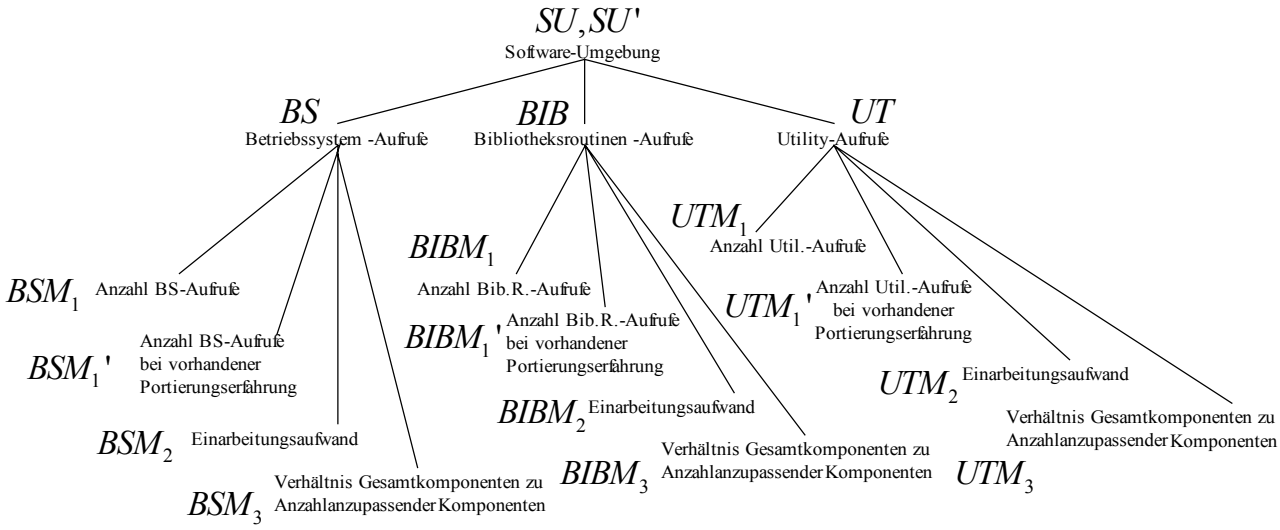
$\text{VKG}_{\text{SYS}} = VKG_{SU} + VKG_{HU}$

Verhältnis der Komponentenzahl aus $KOMP_{SYS}$ zur Gesamtzahl der Programmkomponenten

$\text{VSG}_{\text{SYS}} = PS_2'$

Verhältnis aus der Anzahl der zu ändernden Statements zur Gesamtzahl der Statements.

Aufbau Teilmaß 'Software-Umgebung':



Formal:

$SU = (PAN_{SU}, KOMP_{SU}, VKG_{SU})$ mit:

$PAN_{SU} = BSM_1 + BIBM_1 + UTM_1$

Anzahl der potentiellen Anpassungen von Aufrufen von Routinen der Software-Umgebung

$\text{KOMP}_{\text{SU}} = BSM_2 + BIBM_2 + UTM_2$

Namen der Komponenten mit Aufrufen von Routinen der Software-Umgebung und deren Anzahl

$\text{VKG}_{\text{SU}} = BSM_3 + BIBM_3 + UTM_3$

Verhältnis der Komponentenzahl aus $KOMP_{SU}$ zur Gesamtzahl der Programmkomponenten.

Mit vorhandenen Portierungserfahrungen:

$SU' = (((KA_{BS}, PA_{BS}), SA_{SU}, BA_{SU}), PAN_{SU}, KOMP_{SU}, VKG_{SU})$ mit:

$KA_{BS} = (BSR_1, ..., BSR_n)$

Namen der BS-Routinen, die nicht angepaßt werden können

$\mathrm{PA_{BS}} = ((Anz.BSR_1, BSR_1), ..., (Anz.BSR_m, BSR_m))$

Anzahl der Namen der BS-Routinen, über die während der Messung keine Informationen vorliegen

$\mathrm{SA_{SU}} = SA_{BS} + SA_{BIB} + SA_{UT}$

Gesamtzahl der Aufrufe von SU-Routinen. die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$\mathrm{BA_{SU}} = BA_{BS} + BA_{BIB} + BA_{UT}$

Summe der bewerteten SU-Routinen, für die ein Aufwand vorliegt

$PAN_{SU} = BSM_1 + BIBM_1 + UTM_1$

Anzahl der potentiellen Anpassungen von Aufrufen von Routinen der Software-Umgebung

$\mathrm{KOMP_{SU}} = BSM_2 + BIBM_2 + UTM_2$

Namen der Komponenten mit Aufrufen von Routinen der Software-Umgebung und deren Anzahl

$\mathrm{VKG_{SU}} = BSM_3 + BIBM_3 + UTM_3$

Verhältnis der Komponentenzahl aus $KOMP_{SU}$ zur Gesamtzahl der Programmkomponenten.

Betriebssystem-Aufrufe:

$BS = (BSM_1*, BSM_2, BSM_3)$

$BSM_1* = BSM_1$ oder BSM_1', abhängig von der gegebenen Situation, d.h. ob Portierungserfahrungen vorliegen oder nicht.

Anzahl BS-Aufrufe:

$BSM_1 = Anz.BSR_1 + ... + Anz.BSR_n$ mit:

$\mathrm{BSR_i}$: i-ter unterschiedlicher BS-Aufruf, $i = 1, ..., n$.

Anzahl BS-Aufrufe bei vorhandener Portierungserfahrung:

$BSM_1' = (KA_{BS}, PA_{BS}, SA_{BS}, BA_{BS})$ mit:

$KA_{BS} = (BSR_1, ..., BSR_k)$

Namen der BS-Routinen, die nicht angepaßt werden können

$\mathrm{PA_{BS}} = ((Anz.BSR_1, BSR_1), ..., (Anz.BSR_m, BSR_m)$

Namen und Anzahl der BS-Routinen, über die während der Messung keine Information vorliegt

$SA_{BS} = (Anz.BSR_1, ..., Anz.BSR_n)$

Gesamtzahl der Aufrufe von BS-Routinen, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$$\mathrm{BA_{BS}} = Anz.BSR_1 \cdot B_1 + \ldots + Anz.BSR_r \cdot B_r$$

Summe der bewerteten BS-Routinen, für die ein Aufwand vorliegt

B_i : Aufwand für die Anpassung der i – ten BS-Routine, $i = 1,\ldots,r$

KA : Keine Anpassungen
PA : Potentielle Anpassungen
SA : Sichere Anpassungen
BA : Bewertete Anpassungen.

Einarbeitungsaufwand:

$BSM_2 = ((Komp_1, Anz.BSA),\ldots,(KOMP_n, Anz.BSA))$ mit:
$KOMP_i$: i-te betroffene Komponente des Programms, $i = 1,\ldots,n$
BSA : anzupassender Aufruf einer BS-Routine.

Verhältnis Gesamtkomponenten zur Anzahl anzupassender Komponenten:

$$BSM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BS\text{-}Anpassungen}{Anzahl\ aller\ Komponenten}.$$

Bibliotheksroutinen-Aufrufe:

$BIB = (BIBM_1{*}, BIBM_2, BIBM_3)$ mit:
$BIBM* = BIBM_1$ oder $BIBM_1'$, abhängig von der gegebenen Situation.

Anzahl Bibliotheksroutinen-Aufrufe bei vorhandener Portierungserfahrung:

$BIBM_1' = (SA_{BIB}, BA_{BIB})$ mit:

$\mathrm{SA_{BIB}} = Anz.BIBR_1 + \ldots + Anz.BIBR_m$

Gesamtzahl der Aufrufe von Bibliotheksroutinen, zu denen keine Aufwände vorliegen

$\mathrm{BA_{BIB}} = Anz.BIBR_1 \cdot B_1 + \ldots + Anz.BIBR_n \cdot B_n$

Summe der bewerteten Bibliotheksroutinen für die ein Aufwand vorliegt

B_i : Aufwand für die Anpassung des Aufrufs der i-ten Bibliotheksroutine, $i = 1,\ldots,n$.

Einarbeitungsaufwand:

$BIBM_2 = ((KOMP_1, Anz.BIBA),\ldots,(KOMP_n, Anz.BIBA))$ mit:
$\mathrm{KOMP_i}$: i-te Komponente des Programms, $i = 1,\ldots,n$
$BIBA$: anzupassender Aufruf einer Bibliotheksroutine.

Verhältnis Gesamtkomponenten zur Anzahl anzupassender Komponenten:

$$BIBM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BIB\text{-}Anpassungen}{Anzahl\ aller\ Komponenten}.$$

Utility-Aufrufe:

$UT = (UTM_1{*}, UTM_2, UTM_3)$ mit:

$\mathrm{UTM_1}{*} = UTM_1$ oder UTM_1', abhängig von der gegebenen Situation.

Anzahl Utility-Aufrufe:

$UTM_1 = Anz.UTR_1 + ... + Anz.UTR_n$ mit:

$\mathrm{UTR_i}$: i-ter unterschiedlicher UT-Aufruf, $i = 1,...,n$.

Anzahl Utility-Aufrufe bei vorhandener Portierungserfahrung:

$UTM_1' = (SA_{UT}, BA_{UT})$ mit:

$\mathrm{SA_{UT}} = Anz.UTR_1 + ... + Anz.UTR_m$

Gesamtzahl der Aufrufe von UT-Routinen, zu denen keine Aufwände vorliegen

$BA_{UT} = Anz.UTR_1 \cdot B_1 + ... + Anz.UTR_n \cdot B_n$

Summe der bewerteten UT-Routinen, für die ein Aufwand vorliegt

B_i : Aufwand für die Anpassung des Aufrufs der i-ten UT-Routine, i=1,...,n.

Einarbeitungsaufwand:

$UTM_2 = ((KOMP_1, Anz.UTA),...,(KOMP_n, Anz.UTA))$ mit:

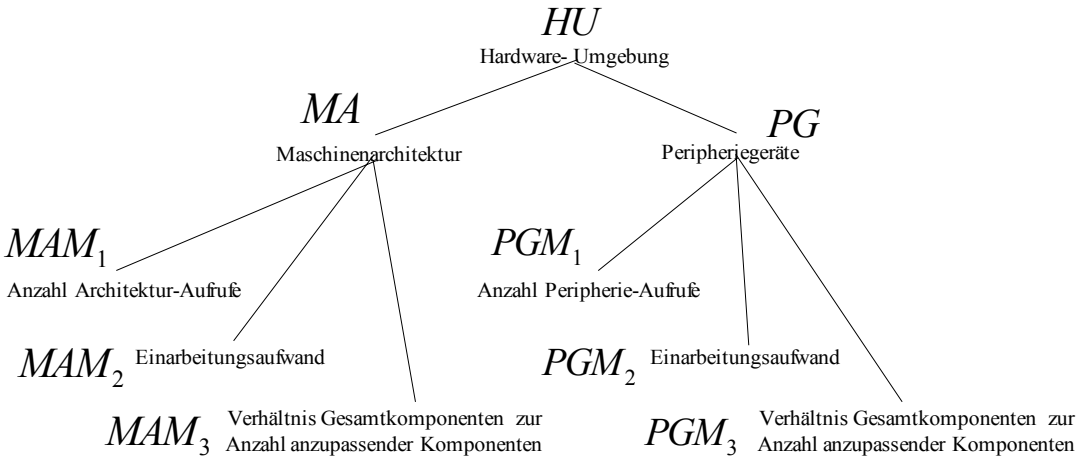
$\mathrm{KOMP_i}$: i-te betroffene Komponente des Programms, $i = 1,...,n$

UTA : anzupassender Aufruf einer UT-Routine.

Verhältnis Gesamtkomponenten zur Anzahl anzupassender Komponenten:

$$UTM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ UT\text{-}Anpassungen}{Anzahl\ aller\ Komponenten}.$$

Aufbau Teilmaß 'Hardware-Umgebung':



Formal:

$HU = (PAN_{HU}, KOMP_{HU}, VKG_{HU})$ mit:

$PAN_{HU} = MAM_1 + PGM_1$

Anzahl der potentiellen Anpassungen von Zugriffen auf die Hardware-Umgebung

$\text{KOMP}_{\text{HU}} = MAM_2 + PGM_2$

Namen der Komponenten mit Zugriffen auf die Hardware-Umgebung und deren Anzahl

$\text{VKG}_{\text{HU}} = MAM_3 + PGM_3$

Verhältnis der Komponentenzahl aus $KOMP_{HU}$ zur Gesamtzahl der Programmkomponenten.

Maschinenarchitektur:

$$MA = (MAM_1, MAM_2, MAM_3) .$$

Anzahl Architektur-Aufrufe:

$MAM_1 = Anz.MA_1 + ... + Anz.MA_n$ mit:

MA_i : i-ter unterschiedlicher MA-Zugriff, $i = 1,....,n.$

Einarbeitungsaufwand:

$MAM_2 = ((KOMP_1, Anz.MAA),...,(KOMP_n, Anz.MAA))$ mit

KOMP_i : i-te betroffene Komponente des Programms, $i = 1,...,n$

MAA : anzupassender MA-Zugriff.

Verhältnis Gesamtkomponenten zur Anzahl anzupassender Komponenten:

$$MAM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ MA\text{-}Zugriffen}{Anzahl\ der\ Komponenten}.$$

Peripheriegeräte:

$$PG = (PGM_1, PGM_2, PGM_3)\ .$$

Anzahl Peripherie-Aufrufe:

$\mathrm{PGM}_1 = Anz.PG_1 + ... + Anz.PG_n$ mit:
PG_i : i-ter unterschiedlicher PG-Zugriff, $i = 1,...,n$,

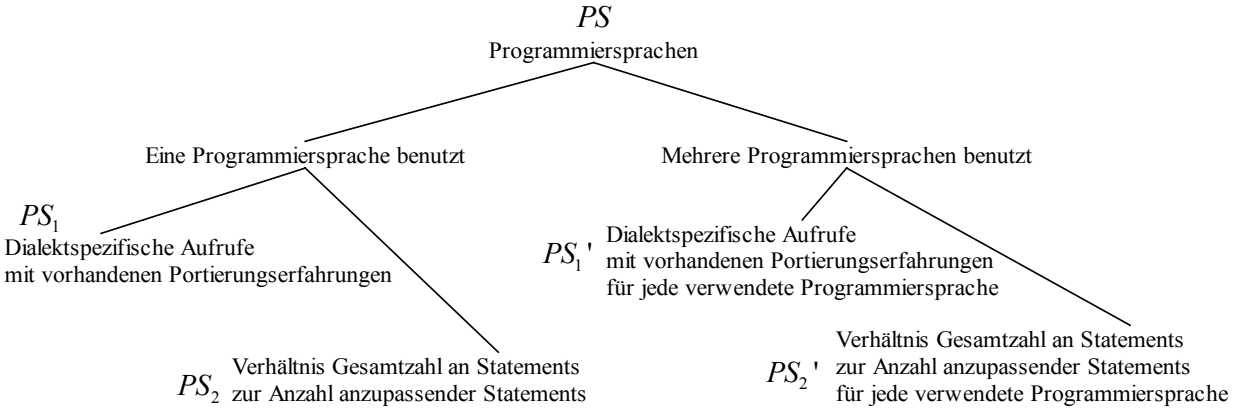
Einarbeitungsaufwand:

$PGM_2 = ((KOMP_1, Anz.PGA),...,(KOMP_n, Anz.PGA))$ mit:
KOMP_1 : i-te betroffene Komponente des Programms, i=1,...,n
PGA : anzupassender PG-Zugriff.

Verhältnis Gesamtkomponenten zur Anzahl anzupassender Komponenten:

$$PGM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ PG\text{-}Zugriffen}{Anzahl\ der\ Komponenten}.$$

Aufbau Teilmaß 'Programmiersprachen':



Formal:

$PS = (PS_1{*}, PS_2{*})$ mit:
$PS_i{*} = PS_i$ oder PS_i', $i = 1,2,$ abhängig von der gegebenen Situation, d.h.
je nach Anzahl verwendeter Programmiersprachen.

Verwendung einer Programmiersprache und Messung dialektspezifischer Aufrufe mit vorhandenen Portierungserfahrungen:

$PS_1 = (KA_{PS}, PA_{PS}, SA_{PS}, BA_{PS})$ mit:

$\mathrm{KA_{PS}} = (SK_1, ..., SK_k)$

Konstrukte, die nicht angepaßt werden können

$\mathrm{PA_{PS}} = ((Anz.SK_1, SK_1), ..., (Anz.SK_m SK_m))$

Anzahl und Namen der Konstrukte, über die während der Messung keine Informationen vorliegen

$\mathrm{SA_{PS}}$=(Anz.$\mathrm{SK_1}$,...,Anz.$\mathrm{SK_n}$)

Anzahl der Konstrukte, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$\mathrm{BA_{PS}}$=Anz.$\mathrm{SK_1} \cdot \mathrm{B_1}$+...+Anz.$\mathrm{SK_r} \cdot \mathrm{B_r}$

Bewertung der Sprachkonstrukte, für die ein Aufwand vorliegt

B_i: Aufwand für die Anpassung des i-ten Konstruktes

$i = 1, ..., r$.

Verhältnis Gesamtzahl an Statements zur Anzahl anzupassender Statements:

$$PS_2 = \frac{Anzahl\ aller\ zu\ ändernden\ Statements}{Gesamtzahl\ aller\ Statements}.$$

Verwendung mehrerer Programmiersprachen und Messung dialektspezifischer Aufrufe, mit vorhandenen Portierungserfahrungen für jede Programmiersprache:

$PS_1' = (KA_{PS\_GES}, PA_{PS\_GES}, SA_{PS\_GES}, BA_{PS\_GES})$ mit:

$\mathrm{KA_{PS\_GES}} = (KA_{PS1}, ..., KA_{PSn})$

Menge der nicht anpaßbaren Konstrukte der einzelnen Programmiersprachen P_i, $i = 1, ..., n$

$PA_{PS\_GES} = (PA_{PS1}, ..., PA_{PSn})$

Menge der Konstrukte der einzelnen Programmiersprachen zu denen keine Informationen existieren

$SA_{PS\_GES} = SA_{PS1} + ... + SA_{PSn}$

Summe der anpaßbaren Konstrukte der einzelnen Sprachen PS_i, zu denen keine Bewertungen existieren

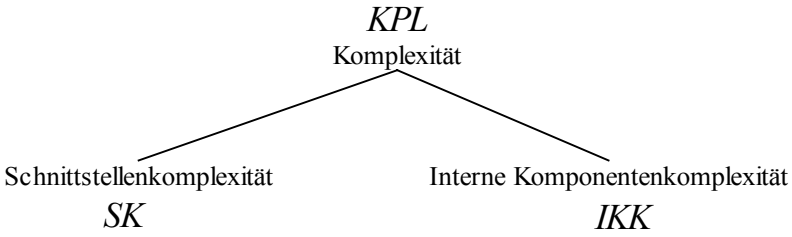
$BA_{PS\_GES} = BA_{PS1} + ... + BA_{PSn}$

Summer der bewerteten Konstrukte der einzelnen Sprachen PS_i.

Zusammenfassung der zweiten Maße der Programmiersprachen:

$$PS_2' = (PS_{21}, ..., PS_{2n}).$$

Es folgt nun der Aufbau für das Teilmaß 'Komplexität':



Formal:

$$KPL = (KPK, DKK, SK_{aggr}, DLK, IKK_{aggr})$$

mit: KPK : Kopplungskomplexität

DKK : Datenkopplungskomplexität

SK_{aggr} : aggregierte Schnittstellenkomplexität

DLK : durchschnittliche logische Komplexität

IKK_{aggr} : aggregierte interne Komplexität

wobei: $KPK = \dfrac{\text{Anzahl der Kopplungen aller Komponenten}}{KP}$

$$DKK = \frac{\text{Anzahl der Datenkopplungen aller Komponenten}}{\text{DK}}$$

$$SK_{aggr} = s_1 \cdot KP + s_2 \cdot DK$$

mit: s_1, s_2 als Gewichtungsfaktoren für die jew. Kopplungsart

$$DLK = \frac{V(P)}{k}$$

mit: $V(P)$: Zyklomatischen Komplexität des Programms

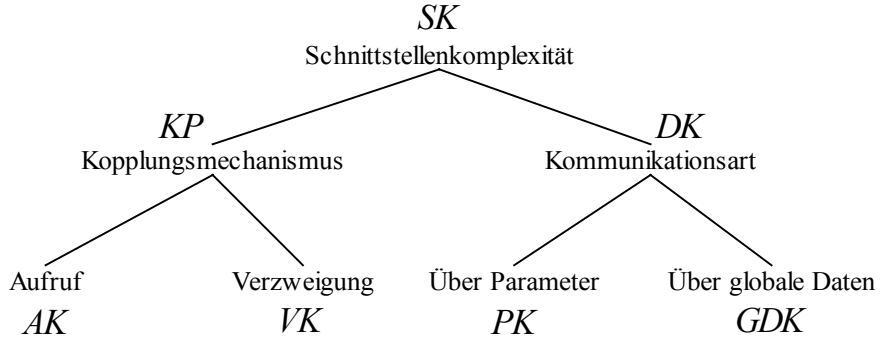
k : Anzahl der Programmkomponenten

$$IKK_{aggr} = k_1 \cdot V(K) + k_2 \cdot S(K)$$

mit: $S(K)$: Summe der Verschachtelungswerte aller Komponenten

k_1, k_2 : Gewichtungsfaktoren.

Aufbau Teilmaß 'Schnittstellenkomplexität':



Formal:

$$SK = (KP, DK).$$

Kopplungsmechanismus:

$$KP = \sum_{i=1}^{k} KP_i$$

mit: k : Anzahl der Komponenten

KP_i : Kopplung der i-ten Komponente über den Kopplungsmechanismus

Kommunikationsart:

$$DK = \sum_{i=1}^{k} DK_i$$

mit: k : Anzahl der Komponenten

DK_i : Datenkopplung der i-ten Komponente.

Über Parameter und globale Daten:

$$DK_i = d_1 \cdot PK_i + d_2 \cdot GDK_i$$

mit DK_i : Datenkopplung der i-ten Komponente $i = 1,...,k$

d_1, d_2 : Gewichtungsfaktoren der Datenkopplungsart

PK_i : Parameter der i-ten Komponente

GDK_i : globale Datenzugriffe der i-ten Komponente.

Aufruf und Verzweigung:

$$KP_i = m_1 \cdot AK_i + m_2 \cdot VK_i$$

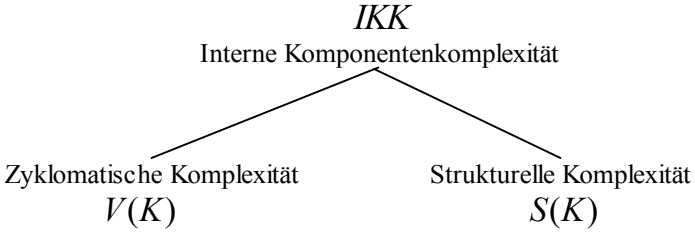
mit KP_i : Kopplung der i-ten Komponente $i = 1,...,k$

m_1, m_2 : Gewichtungsfaktoren der Kopplungsmechanismen

AK_i : Aufrufe der verschiedenen Komponenten

VP_i : Verzweigungen in verschiedene Komponenten.

Aufbau Teilmaß 'Interne Komponentenkomplexität':



Formal:

$$IKK = (V(K), S(K)).$$

Zyklomatische Komplexität:

$$V(K) = \sum_{i=1}^{k} V(K_i)$$

mit $V(K_i)$: Komplexität der i-ten Komponente

k : Anzahl der Komponenten,

$$V(K_i) = e_i - n_i + 2$$

mit e_i : Anzahl der Pfeile in einem Graphen

n_i : Anzahl der Bedingungen in Entscheidungsknoten

i : i-te Komponente

bzw. einfacher durch:

$$V(K_i) = b_i + 1$$

mit b_i : Anzahl der Bedinungen der i-ten Komponente.

Verschachtelung:

$$S(K) = \sum_{i=1}^{k} S(K_i)$$

mit $S(K_i)$: Verschachtelungswert der i-ten Komponente

k : Anzahl der Komponenten,

$$S(K_i) = \sum_{j=1}^{n} (1 - \frac{1}{d_j})$$

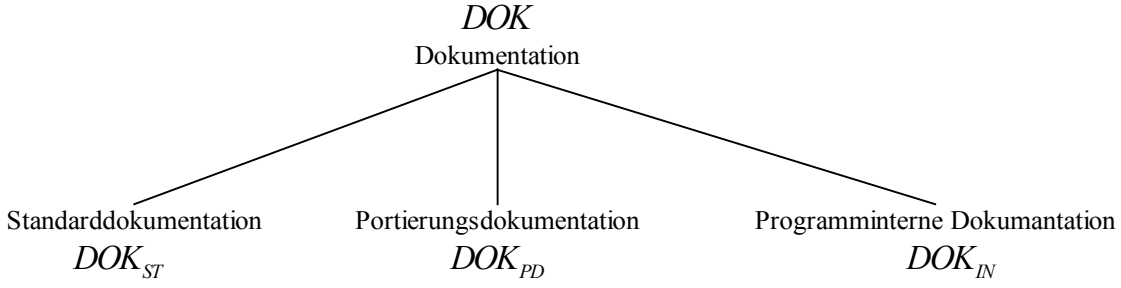
mit j : 1,2,...,n

K_i : i-te Komponente

n : Anzahl aller Verzweigungen in einem Graphen

d_j : Anzahl der vom j-ten Verzweigungsknoten dominierten weiteren Verzweigungsknoten + 1.

Es folgt der Aufbau für das Teilmaß 'Dokumentation':



Formal:

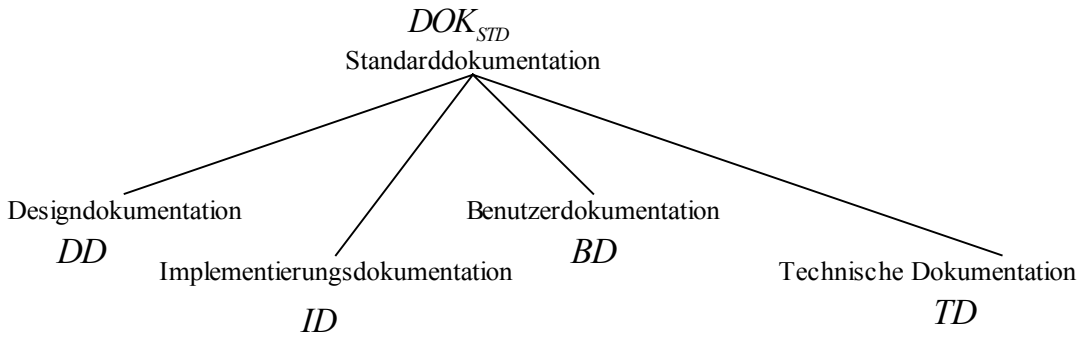
$$DOK = g_1 \cdot DOK_{ST} + g_2 \cdot DOK_{PD} + g_3 \cdot DOK_{IN}$$

mit g_1, g_2, g_3 : Gewichtungsfaktoren der einzelnen Teile der Dokumentation,

Hypothese:

$$DOK = 0{,}15 \cdot DOK_{ST} + 0{,}6 \cdot DOK_{PD} + 0{,}25 \cdot DOK_{IN} \quad \text{wobei gilt: } \sum_{i=1}^{3} g_i = 1.$$

Aufbau Teilmaß 'Standarddokumentation':



Formal:

$$DOK_{STD} = g_1 \cdot DD + g_2 \cdot ID + g_3 \cdot BD + g_4 \cdot TD \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1,$$

Hypothese:

$$DOK_{STD} = 0{,}25 \cdot DD + 0{,}25 \cdot BD + 0{,}25 \cdot TD + 0{,}25 \cdot ID.$$

Designdokumentation:

$DD =$ Designdokumentation vorhanden/nicht vorhanden.

Implementierungsdokumentation:

$ID =$ Implementierungsdokumentation vorhanden/nicht vorhanden.

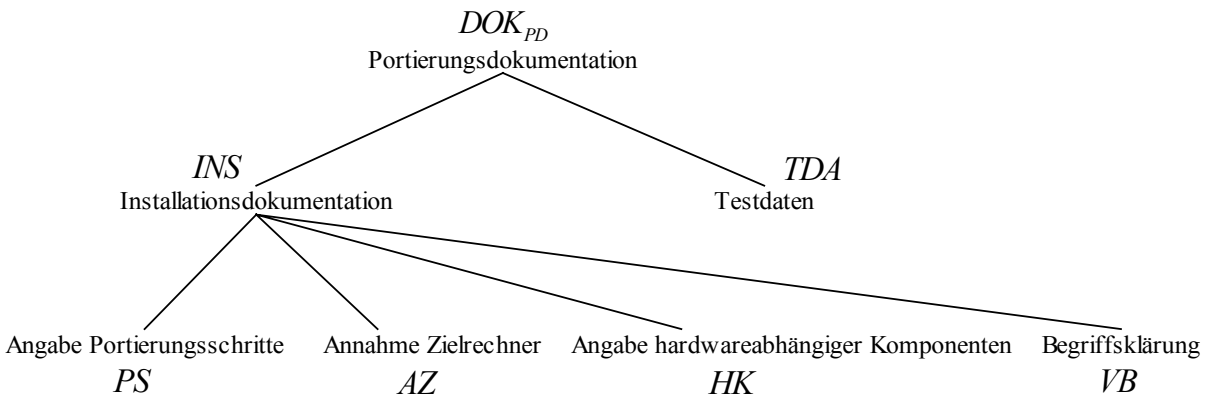
Benutzerdokumentation:

$BD =$ Benutzerdokumentation vorhanden/nicht vorhanden.

Technische Dokumentation:

$TD =$ Technische Dokumentation vorhanden/nicht vorhanden.

Aufbau Teilmaß Portierungsdokumentation:



Formal:

$$DOK_{PD} = g_1 \cdot INS + g_2 \cdot TDA \quad \text{wobei gilt: } g_1 + g_2 = 1,$$

$$Hypothese:$$
$$DOK_{PD} = 0,6 \cdot INS + 0,4 \cdot TDA.$$

Installationsdokumentation:

$$INS = g_1 \cdot PS + g_2 \cdot AZ + g_3 \cdot HK + g_4 \cdot VB \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1,$$

Hypothese:
$$INS = 0,4 \cdot PS + 0,3 \cdot AZ + 0,2 \cdot HK + 0,1 \cdot VB.$$

Testdaten:

$TDA =$ Testdaten vorhanden/nicht vorhanden.

Angabe Portierungsschritte:

$PS =$ Angabe der Portierungsschritte vorhanden/nicht vorhanden.

Annahme Zielrechner:

$AZ =$ Angabe der über den Zielrechner gemachten Annahmen vorhanden/nicht vorhanden.

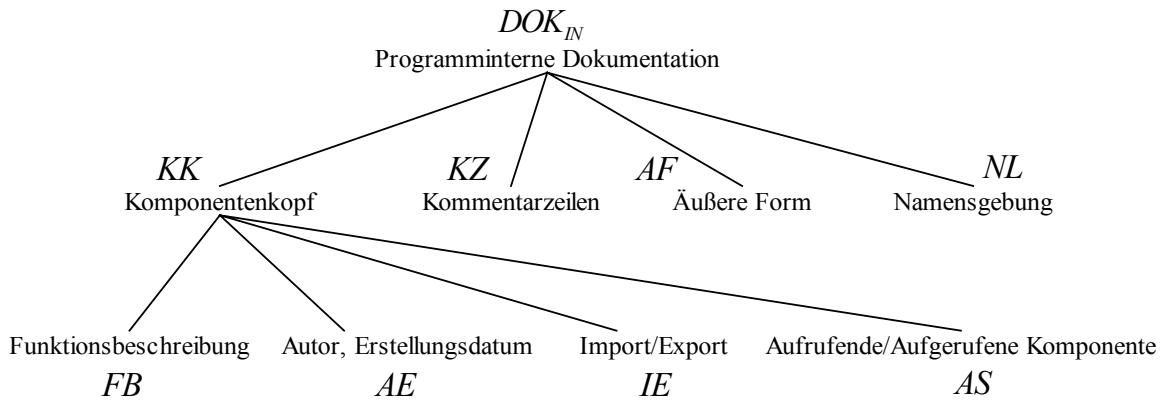
Angabe hardwareabhängiger Komponenten:

$HK =$ Angabe der hardwareabhängigen Komponenten vorhanden/nicht vorhanden.

Begriffsklärung:

$VB =$ Erklärung der verwendeten Begriffe vorhanden/nicht vorhanden.

Aufbau Teilmaß 'Programminterne Dokumentation':



Formal:

$$DOK_{IN} = g_1 \cdot KK + g_2 \cdot KZ + g_3 \cdot AF + g_4 \cdot NL \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1,$$

Hypothese:

$$DOK_{IN} = 0,3 \cdot KK + 0,3 \cdot KZ + 0,3 \cdot AF + 0,1 \cdot NL.$$

Komponentenkopf:

$$KK = g_1 \cdot FB + g_2 \cdot AE + g_3 \cdot IE + g_4 \cdot AS \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1,$$

Hypothese:

$$KK = 0,25 \cdot FB + 0,25 \cdot AE + 0,25 \cdot IE + 0,25 \cdot AS.$$

Kommentarzeilen:

$KZ =$ Benutzung von Kontrollstatements dokumentiert / nicht dokumentiert.

Äußere Form:

$AF =$ Quellcode strukturiert / nicht strukturiert.

Namensgebung:

$BL =$ durchschnittliche Bezeichnerlänge 5-9 Zeichen / nicht 5-9 Zeichen.

Funktionsbeschreibung:

FB = Beschreibung der Systemkomponente vorhanden/nicht vorhanden.

Autor und Erstellungsdatum

AE = Autor und Erstellungsdatum vorhanden/nicht vorhanden.

Import/Export:

IE = Auflistung Import/Export vorhanden/nicht vorhanden.

Aufrufende/Aufgerufene Komponente:

AS = Auflistung der abhängigen Systemkomponenten vorhanden/nicht vorhanden.

3.2.3.5 Bewertung

Das vorgestellte Maß bestimmt den Aufwand einer Portierung, anhand der Einflußfaktoren: Systemumgebung, Komplexität und Dokumentation. Um von einem Maß sprechen zu können müssen, nach der Definition von Balzert, (siehe Kapitel 3.1.2) bestimmte Gütekriterien erfüllt sein:

- Objektivität (Intersubjektivität), d.h. daß ein Maß erst dann als objektiv bezeichnet werden kann, wenn keine subjektiven Einflüsse des Messenden auf die Messung möglich sind;
- Zuverlässigkeit (Meßgenauigkeit), d.h. bei der Wiederholung der Messung unter denselben Meßbedingungen werden dieselben Meßergebnisse erzielt, womit ein Maß als stabil und präzise bzw. zuverlässig bezeichnet werden kann;
- Validität (Gültigkeit, Meßtauglichkeit), d.h. daß die Meßergebnisse einen eindeutigen und unmittelbaren Rückschluß auf die Ausprägung der Kenngröße erlauben;
- Normierung, d.h. es existiert eine Skala, auf der die Meßergebnisse eindeutig abgebildet werden und zusätzlich eine Vergleichbarkeitsskala (sozusagen als Referenzsystem zur Klassifizierung von Meßergebnissen) vorhanden ist, die das Maß entsprechend normiert;
- Vergleichbarkeit, d.h. daß das Maß in Relation mit anderen Maßen gesetzt werden kann;
- Ökonomie, d.h. daß die Messung mit geringen Kosten bzw. geringem Aufwand durchgeführt werden kann und vom Automatisierungsgrad, der Anzahl der Meßgrößen und der Anzahl der Berechnungsschritte abhängt;
- Nützlichkeit, d.h. daß mit einer Messung praktische Bedürfnisse erfüllt werden.

Da durch die lange Lebensdauer des Maßes, bedingt durch dessen Evolutionsaspekt, eine ganze Reihe verschiedener Maßanwender auftreten, spielt die Anforderung der Objektivität eine besondere Rolle. Wird den Anwendern ein zu großer Freiraum für Interpretationen gelassen, so ist keine exakte und kontinuierliche Bewertung möglich. Auch die Vergleichbarkeit von Erfahrungen verschiedener Portierungen, würde dadurch wesentlich erschwert. Da die Einflußgrößen, Systemumgebung und Komplexität, automatisch ermittelt werden, ist hierbei die Forderung nach Objektivität, im Sinne einer Ausführungsobjektivität, gewährleistet. Die Einflußgröße Dokumentation hat stark subjektiven Charakter, da durch einen Review die Kenntnisse des Porteurs in das Maß mit eingebracht werden.

Die Zuverlässigkeit eines Maßes zeigt sich, wenn das Maß unter gleichen Bedingungen, bei wiederholter Anwendung, zu den gleichen Ergebnissen kommt. Dies läßt sich u.a. durch den Einsatz formaler Beschreibungsmethoden und entsprechend homogener Methodendarstellung erreichen. Auch die Automatisierung der einzelnen Meßschritte, durch entsprechende Werkzeuge, spielt für die Zuverlässigkeit des Maßes eine ebenso starke Rolle. Da für das hier vorgestellte Aufwandsmaß noch keine praktischen Erfahrungen und somit konkret vergleichbare Ergebnisse vorliegen, kann die Einschätzung nur anhand der aufgeführten Kriterien vorgenommen werden. Die einzelnen Schritte, die zum Gesamtmaß 'Portabilität' führen, werden umfassend formalisiert und in großen Teilen automatisiert. Der stark subjektive Charakter der Dokumentationsbewertung läßt jedoch Zweifel, hinsichtlich der Zuverlässigkeit des Maßes, aufkommen. Auch läßt sich durch den Evolutionsaspekt die Einhaltung gleicher Bedingungen schwer, oder gar nicht, überprüfen.

Ein äußerst schwierig nachzuweisendes Kriterium, ist der Nachweis, daß ein Maß auch tatsächlich das mißt, was es vorgibt zu messen. Balzert [Balzert 98] geht sogar so weit zu behaupten, daß ein Maß ohne Validitätsaussagen für objektive Bewertungen unbrauchbar ist. Er unterscheidet je nach Verwendungszweck unterschiedlich hohe Anforderungen in Bezug auf die Validität eines Maßes. So muß z.B. für eine Ursache-Wirkungs-Analyse die Validität höher sein als für den, im Portabilitätsmaß auftretenden, Zeitvergleich. Konkrete Aussagen zur Validität des Aufwandsmaßes 'Portabilität' werden jedoch nicht gemacht, da bisher keine praktischen Erfahrungen mit dem Maß vorliegen, die entsprechende Validitätsaussagen, mittels empirischem Nachweis (z.B. externe Validierung gegen Portierungskosten) zulassen.

Existiert eine Funktion von der realen Welt in eine formale, numerische Welt, dann gibt es auch eine Skala, auf der sich der Zusammenhang darstellen läßt. Das hier dargestellte Maß liefert zwar einen entsprechenden Meßwert, dieser läßt sich bisher jedoch in keinen Zusammenhang mit der realen Welt bringen, da bisher keine Portierungserfahrungen vorliegen, welche eine entsprechende Zuordnung rechtfertigen. Aus diesem Grund kann das vorliegende Maß, in diesem Sinne, als nicht normiert bzw. standardisiert bezeichnet werden.

Um zwischen ähnlichen Maßen wählen zu können, müssen Informationen zu den Maßen vorhanden sein, aus denen hervorgeht, was gemessen werden soll und inwieweit diese validiert sind. Man kann diese Maße dann miteinander vergleichen bzw. in Relation zueinander setzen. Da bisher keine vergleichbaren Maße existieren und das vorhandene Maß auch nicht ausreichend validiert ist, ist das Gütekriterium Vergleichbarkeit ebenfalls nicht erfüllt.

Die Ökonomie ist ein wichtiges Kriterium für die Akzeptanz eines Maßes. Erfordert die Ermittlung der einzelnen Meßwerte einen hohen Aufwand, kann von der Anwendung des Maßes abgesehen werden. Das bedeutet, die Meßergebnisse müssen schnell und ohne großen Vorbereitungsaufwand zustande kommen. Die Ökonomie eines Maßes hängt stark von der Automatisierbarkeit der Meßwertermittlung ab. Auch hier ist der Einflußfaktor 'Dokumentation' eine entscheidende Störgröße, die sich nicht vollständig automatisiert erfassen läßt und damit die Ökonomie des Portierungsmaßes negativ beeinflußt. Alle anderen Einflußgrößen sind vollständig automatisiert erfaßbar, da hier bereits entsprechend einsetzbare Werkzeuge zur Verfügung stehen. Für die Ermittlung des Gesamtmaßes 'Portabilität' fehlt jedoch ein geeignetes Werkzeug, welches die einzelnen Teilmaße unter einer einheitlichen Oberfläche zusammenführt.

Die Nützlichkeit eines Maßes hängt davon ab, inwiefern damit praktische Bedürfnisse befriedigt werden. Es spielt also eine Rolle, wie wichtig quantitative Informationen über bestimmte Merkmale für einen Maßanwender sind und ob gleiche oder ähnliche Maße vorhanden sind, mit denen diese Informationen beschafft werden können. Es kann gesagt werden, daß der praktische Nutzen eines Portabilitätsmaßes, welches den Anspruch erhebt, den Aufwand einer Portierung vorherzusagen, sicherlich groß ist und damit ein gewisser Bedarf vorhanden ist, zumal eine Reihe verschiedenster Systemumgebungen existieren. Dieser Bedarf könnte sich mit der Zeit jedoch ändern, da der allgemeine Trend immer mehr in Richtung Vereinheitlichung von Systemumgebungen geht. Solange jedoch noch keine ausreichenden Portierungserfahrungen zur Verfügung stehen und diese in das vorgestellte Portabilitätsmaß einfließen, ist das Maß nur als eingeschränkt nützlich anzusehen und höchstens für erste grobe Portabilitätsanalysen brauchbar. Diese erlauben jedoch keinesfalls präzise Aussagen zum zeitlichen Portierungsaufwand eines dementsprechend analysierten Software-Systems.

Zusammenfassend kann gesagt werden, daß es sich bei dem vorgestellten Maß, eher um eine Software-Metrik, im Sinne der Definition von Balzert (siehe Kapitel 3.1.2) handelt, als um ein Software-Maß. Dies wird begründet durch die Untersuchung auf einzuhaltende Gütekriterien für Software-Maße, wie sie in Balzert [Balzert 98] beschrieben werden. Hierbei werden keine der geforderten Gütekriterien vollständig erfüllt. Es wird jedoch formal definiert, wie die Kenngröße ‘Portabilität’ eines Software-Produkts gemessen werden kann und fällt somit in den Rahmen der Definition einer Software-Metrik. Die Auswahl der Einflußgrößen stellt einen Kompromiß zwischen gewünschter Objektivität und gezwungener Subjektivität dar und ist nicht als vollständig anzusehen, zumal die Erfahrungen und Kenntnisse eines Porteurs, die den Aufwand einer Portierung maßgeblich beeinflussen können, nicht oder nur teilweise (Dokumentation) berücksichtigt werden. Für eine erste und grobe Portabilitätsanalyse stellt dieser Ansatz jedoch ein brauchbares Fundament dar und soll als Basis für die Untersuchung eines Windows-basierten Software-Systems dienen. Hierbei ist zu berücksichtigen, daß das vorgestellte Maß für solche Software-Systeme konzipiert ist, die den Prinzipien der strukturierten, prozeduralen Programmierung folgen. Vorschläge für die Anpassung an objektorientierte Software-Systeme werden im Folgenden unterbreitet.

3.2.3.6 Anpassung an objektorientierte Software-Systeme

Das betrachtete Maß ist speziell für die Untersuchung prozedural programmierter Software-Systeme entwickelt worden. Es soll nun geklärt werden, inwieweit das vorgestellte Maß für objektorientierte Software-Systeme verwendbar ist. Falls notwendig, werden Vorschläge unterbreitet, wie das Maß für objektorientierte Systeme anzupassen sein könnte.

In der prozeduralen Programmierung besteht ein Software-System aus Funktionen bzw. Prozeduren, die als Systemkomponenten, überwiegend sequentiell, miteinander verknüpft werden. Die einzelnen Systemkomponenten umfassen hierbei einen Anweisungsblock, mit den darin befindlichen Systemelementen und haben genau einen Ein- und Austrittspunkt. Die einzelnen Systemkomponenten und Systemelemente können nun in verschiedenen Wechselbeziehungen zueinander stehen und bilden damit ein komplexes Software-System. Solche Kontrollstrukturen werden oft als Kontrollflußgraphen dargestellt. Anweisungen in einem Programm werden durch entsprechende Knoten symbolisiert und der Übergang der Kontrolle einer Anweisung an eine andere Anweisung, wird mittels gerichteter Kanten angezeigt. Kombinationsregel auf solchen Kontrollgraphen ist überwiegend die sequentielle Verknüpfung, aber auch Schleifen oder die Auswahl kommen hierbei in Betracht. Klassische Metriken

werden auf den beschriebenen Kontrollstrukturen definiert und haben damit den hier vorgestellten Gültigkeitsbereich.

Bei objektorientierten Software-Systemen kann das System als eine Menge von Klassen mit bestimmten Eigenschaften, ergänzt um zwei Relationen auf dieser Menge, der Vererbungs- und Benutzt-Relation, verstanden werden. Die Eigenschaften einer Klasse werden durch ihre Attribute festgelegt und als Instanzvariablen dargestellt. Klassen beschreiben darüber hinaus das Verhalten von Objekten, bei der jede Instanz einer Klasse (Objekt) über den gleichen Satz solcher Variablen verfügt. Das Verhalten von Objekten wird mit sog. Methoden definiert, mit denen auf Botschaften an ein Objekt reagiert wird. Auch können Klassen solche Methoden enthalten, die durch entsprechende Nachrichten an die jeweilige Klasse angestoßen werden und analog hierzu kann es Variablen geben, die global von allen Instanzen gemeinsam benutzt werden. Die interne Beschaffenheit einer Methode, die der Systemkomponente bei der prozeduralen Programmierung entspricht, ist stark abhängig vom konkreten OOP-Modell. Der eigentliche Programmcode wird stets in den Methoden gekapselt und durch eine Nachricht an ein Objekt ausgelöst. Die Nachricht kann parametrisiert sein, was bedeutet, daß die korrespondierende Methode in der selben Form parametrisiert sein muß. Diese können Ein- oder Ausgabeparameter oder beides sein. Die Methode umfaßt, wie bei der prozeduralen Programmierung, einen Anweisungsblock, mit genau einem Eintritts- und einem Austrittspunkt. Auf dieser Abstraktionsebene im OO-Modell, d.h. auf Methodenebene, kann daher das Verhalten der Methoden mit Hilfe klassischer Software-Metriken untersucht werden. Der Unterschied zur Modulhierarchie bei der strukturierten Programmierung liegt nun darin, daß die Relationen auf den Klassen nicht notwendigerweise einen zusammenhängenden Graphen aufspannen, in dem alle Klassen als Knoten enthalten wären. Vielmehr können beide Relationen (Vererbungs- und Benutzt-Relation) jeweils mehrere gerichtete, azyklische Einzelgraphen aufspannen. Somit ist eine wesentliche Eigenschaft aller Relationen auf Klassen, daß nicht eine Beziehung alleine die Klassen eines objektorientierten Systems verbinden muß. Insbesondere ist weder in der Vererbungsrelation, noch in der Benutzt-Relation eine Wurzel gefordert, von der jede Klasse des Systems erben bzw. benutzt würde. Bei der Entwicklung von Metriken für objektorientierte Software-Systeme unterscheidet man daher die Abstraktionsebene der Methoden, der Klassen, der Aggregation, der Vererbungshierarchien und des Systems. Hierbei ist es dann sinnvoll für eine Metrik festzulegen, auf welcher Abstraktionsebene sie mißt, bzw. für welche Abstraktionsebene sie definiert wird.

Die Darstellung der prinzipiellen Unterschiede der aufgeführten Programmierparadigmen zeigt, welcher Zusammenhang zwischen den klassisch prozedural implementierten und den objektorientierten Software-Systemen besteht. Im folgenden sollen nun die Teilmaße des betrachteten Portabilitätsmaßes, auf die Verwendung im objektorientierte Bereich untersucht werden.

System-Umgebung

Bei der Maßkomponente ’System-Umgebung’ werden die Anweisungen des Software-Systems identifiziert und bewertet, die im Verlauf einer Portierung, bedingt durch Unterschiede in der Host- und Zielumgebung, angepaßt werden müssen. Bei den Maßen für die Software- und Hardware-Umgebung werden die Systemkomponenten des Software-Systems auf spezielle Zugriffe in die Systemumgebung hin vermessen, die bei der Portierung potentielle Änderungsaufwände verursachen können. Bei der Software-Umgebung sind dies Betriebssystem-Aufrufe, Aufrufe von Bibliotheksroutinen und Aufrufe von sonstigen Hilfsroutinen. Analog hierzu werden bei der Hardware-Umgebung, maschinenabhängige Zugriffe und Zugriffe auf Peripheriegeräte untersucht.

Beide Teilmaße bewegen sich damit innerhalb einer Prozedur bzw. Funktion (Einhaltung der Prinzipien der strukturierten Programmierung vorausgesetzt) und bewerten die Anweisungen innerhalb des Anweisungsblocks einer solchen Systemkomponente. Diese Maße lassen sich somit auch auf die Komponenten einer Klasse, den Methoden, anwenden und zu einem entsprechenden Umgebungsmaß zusammenfassen. Hierbei ist die Systemkomponente nun nicht mehr eine bestimmte Prozedur bzw. Funktion, sondern eine Klasse. Die Methode als Klassenkomponente kann jedoch nicht als ein Element des Systems (im Sinne der Definition) bezeichnet werden, da sie noch weiter zerlegbar sein soll, z.B. in Anweisungen, die Aufrufe von Routinen enthalten. Als Klassenkomponenten sind, bei der Anwendung des Portabilitätsmaßes in objektorientierten Software-Systemen, die Methoden und Attribute einer Klasse zu verstehen.

Bei den Programmiersprachen werden zwei Fälle unterschieden: die Sprache, in die das zu portierende Software-System übertragen werden soll, ist ein Dialekt der Sprache, in der es geschrieben wurde oder sie ist eine ganz andere Sprache. Die Probleme sind in beiden Fällen die gleichen. Es werden bestimmte Konstrukte einer Sprache als Teil des Systems oder von Systemkomponenten identifiziert und die Anzahl der Anweisungen, die sich auf das entsprechende Sprachkonstrukt beziehen, vermessen. Diese Sichtweise erzeugt somit Unabhängigkeit zum verwendeten Programmierparadigma des untersuchten Software-Systems.

Komplexität

Bei der Komplexitätskomponente des Portabilitätsmaßes werden die Aspekte der Schnittstellenkomplexität und der internen Komponentenkomplexität berücksichtigt. Die Schnittstellenkomplexität wird zum einen durch Vermessung der Kopplungsmechanismen: Aufruf und Verzweigung – wobei der Verzweigungsaspekt bei der strukturierten Programmierung (Vermeidung von goto-Anweisungen) so gut wie keine Bedeutung mehr hat – realisiert und zum anderen durch die Ermittlung der Datenkopplungen, entweder über Parameter oder globale Daten.

Bei der Untersuchung der Kopplungsmechanismen bezieht sich das Maß darauf, wie eine Komponente auf Anweisungen einer anderen Komponente zugreift und ob überhaupt eine Kopplung existiert. Hierbei werden jeweils die Häufigkeiten des Auftretens von Kopplungen einer entsprechenden Kopplungsart ermittelt und einer bestimmten Systemkomponente mit entsprechender Gewichtung zugeschrieben. Wobei der mehrfache Aufruf der gleichen Komponente oder die mehrfache Verzweigung in eine Komponente, nicht zur Erhöhung der Kopplung beitragen sollten. Die Kopplungsstärke ergibt sich durch Aufsummieren der Kopplungen aller betroffenen Systemkomponenten. Analog hierzu, wird bei der Vermessung der Datenkopplung, die Häufigkeit des Auftretens eines Datenaustauschs (Parameter, globale Daten) - wobei Mehrfachzugriffe auf das gleiche Datenelement nicht zur Erhöhung der Kopplung führen - zugrunde gelegt und ebenfalls zu einem Teilmaß ’Datenkopplung’ aufsummiert. Die beiden Teilmaße drücken somit jeweils verschiedene Kopplungsstärken aus, die dann zum Teilmaß ’Schnittstellenkomplexität’ zusammengefaßt werden.

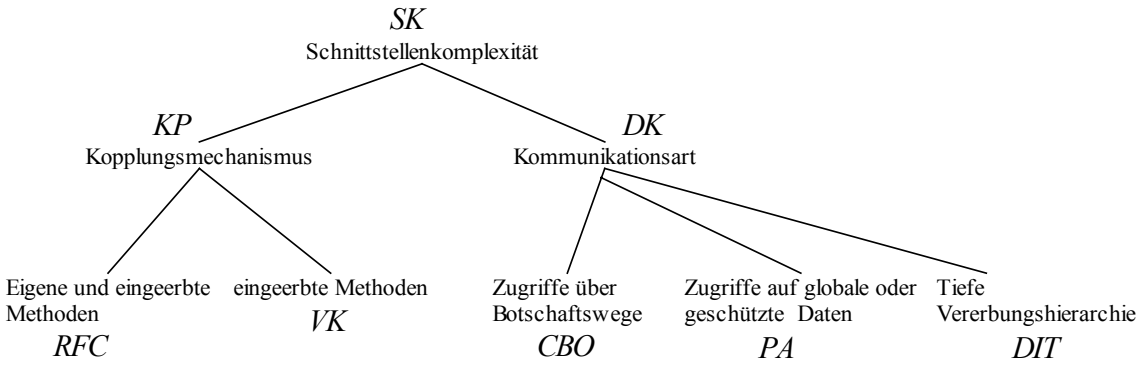
Bei objektorientierten Software-Systemen besteht die Kopplung von Systemkomponenten auf Klassenebene nicht nur aus der Berücksichtigung der Benutzt-Relation, sondern auch der Vererbungsrelation. Darüber hinaus besteht die Möglichkeit, daß Objekte als Instanzen einer Klasse, durch Beziehungen wie Assoziationen, Aggregation oder temporäre Botschaftswege gekoppelt sein können. Das Verhalten von Objekten wird über die Methoden definiert, die von Nachrichten an das

Objekt ausgelöst werden und man betrachtet daher die Methodenebene. Um diese beiden Aspekte von objektorientierten Software-Systemen zu berücksichtigen, werden für die Teilmaße ’Kopplungsmechanismus’ und ,’Kommunikationsart bzw. Datenkopplung’ folgende Vorschläge für die Verwendung objektorientierter Metriken unterbreitet.

Gemessen werden soll die Kopplung zwischen den Komponenten eines objektorientierten Software-Systems und deren Kopplungsstärke. Hierbei werden nur noch Aufrufe betrachtet, womit eine entsprechende Gewichtung auf dieser Maßebene entfallen kann und die Kopplungsstärke direkt aus der Anzahl aufgerufener Methoden bestimmbar ist.

Um bei objektorientierten Software-Systemen die Anzahl von Systemkomponenten zu bestimmen, die über den vorgestellten Kopplungsmechanismus ’Aufruf” in Verbindung stehen, eignet sich die bereits vorgestellte RFC-Metrik, welche die Anzahl der Funktionen, die durch Operationen einer Klasse aufgerufen werden (intern und extern), mißt. Hierzu zählen auch eingeerbte Methoden. Um den Aspekt der Vererbung noch stärker zu berücksichtigen, soll darüber hinaus die Anzahl der in einer Klasse implementierten Methoden *NOM* (ohne eingeerbte Methoden) in das Maß eingehen (Spezialfall der WMC-Metrik). Dazu werden die Methoden nicht gewichtet. Die Differenz aus den beiden zuvor gemessenen Werten ergibt die Zahl der eingeerbten bzw. übernommenen Methoden.

Aufbau Teilmaß ’Schnittstellenkomplexität’ für objektorientierte Software-Systeme:



Formal:

$$SK = (KP, DK)$$

Kopplungsmechanismus:

$$KP = \sum_{i=1}^{k} KP_i$$

mit: k : Anzahl der Komponenten

KP_i : Kopplung der i-ten Komponente über den Aufruf-Kopplungsmechanismus.

Eigene und Eingeerbte Methoden:

$$KP_i = m_1 \cdot RFC_i + m_2 \cdot VK_i$$

mit KP_i : Kopplung der i-ten Komponente $i = 1,...,k$

m_1, m_2 : Gewichtungsfaktoren der Maßanteile

RFC_i : Anzahl der aufrufbaren Methoden der i-ten Komponente sowie die Anzahl der internen und externen Aufrufe, die sie selbst durchführt

VK_i : Anzahl der eingeerbten bzw. übernommenen Methoden der i-ten Komponente

wobei $VK_i = RFC_i - NOM_i$

mit NOM_i : Anzahl der in der i-ten Komponente implementierten und nicht eingeerbten Methoden.

Darüber hinaus soll für objektorientierte Systeme die CBO-Metrik zugrunde gelegt werden, da diese die Anzahl der Klassen, mit denen eine Klasse gekoppelt ist, ausdrückt, wenn Methoden der einen Klasse Methoden oder Instanzvariablen einer anderen Klasse aufrufen bzw. verwenden. Das Maß berücksichtigt keine Vererbungshierarchie, deshalb soll zusätzlich die DIT-Metrik in das Maß mit einfließen. Sie gibt Aufschluß darüber, wie tief die Vererbungshierarchie der Klassen geht und zählt konkret die Anzahl der Vorfahren einer Klasse im Vererbungsbaum. Weiterhin sollen die Zugriffe auf globale oder geschützte Daten einer Klasse gemessen werden. Folgendes Maß, für die Datenkopplung der Systemteile, soll demnach für das Portierungsmaß verwendet werden:

Kommunikationsart (Datenkopplung)

$$DK = \sum_{i=1}^{k} DK_i$$

mit: k : Anzahl der Komponenten

DK_i : Datenkopplung der i-ten Komponente.

Zugriffe über Botschaftswege, globale und geschützte Daten, Tiefe des Vererbungsbaums:

$$DK_i = d_1 \cdot CBO_i + d_2 \cdot PA_i + d_3 \cdot DIT_i$$

mit DK_i : Datenkopplung der i-ten Komponente $i = 1,...,k$

d_1, d_2, d_3 : Gewichtungsfaktoren der Maßanteile mit $d_1 + d_2 + d_3 = 1$

CBO_i : Anzahl der Datenkopplungen über Botschaftswege der i-ten Komponente

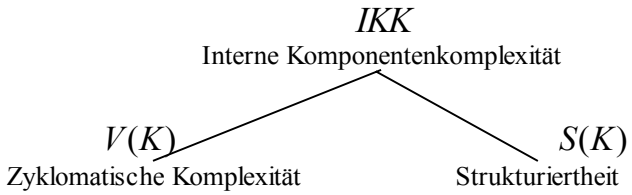
PA_i : Anzahl der Zugriffe auf public oder protected Datenelemente durch andere Klassen auf die i-te Komponente

DIT_i : Länge des maximalen Weges in der Klassenhierarchie von der i-ten Komponente bis zur Wurzel.

In dem vorliegenden Maß wird für die interne Komponentenkomplexität, zum einen die Verwendung der zyklomatische Zahl nach McCabe verwendet und zum anderen die Erweiterung der zyklomatischen Zahl um den Schachtelungswert einer jeden Komponente vorgeschlagen. Bei objektorientierten Software-Systemen herrscht, wie bereits gezeigt wurde, keine einheitliche Meinung in Bezug auf die Verwendung der zyklomatischen Zahl. Es wird daher die Verwendung der zyklomatischen Zahl nach McCabe, in Verbindung mit der WMC-Metrik, vorgeschlagen, wobei die

zyklomatische Zahl, der Methoden einer Klasse entsprechend gewichtet in die WMC-Metrik eingeht. Anstelle des Schachtelungswerts wird für jede Systemkomponente die maximale, essentielle Komplexität bestimmt, da sie einen besseren Gradmesser für die Strukturiertheit eines Software-Systems bietet und zusätzliche Aussagen zu dessen Nachvollziehbarkeit gestattet.

Aufbau Teilmaß 'Interne Komponentenkomplexität' für objektorientierte Software-Systeme:



Formal:

$$IKK = (V(K), S(K)).$$

Zyklomatische Komplexität:

$$V(K) = \sum_{i=1}^{k} VWMC_i$$

mit $VWMC_i$: Komplexität der i-ten Komponente

k : Anzahl der Komponenten,

$$VWMC_i := \sum_{j=1}^{n} g(M_j) \text{ und } g(M_j) = v(M_j)$$

mit $V(M_j)$: Komplexität der j-ten Methode einer i-ten Komponente

n : Anzahl der Methoden.

Strukturiertheit:

$$S(K) = \sum_{i=1}^{k} EWMC_i$$

mit $EWMC_i$: Maximale, essentielle Komplexität der i-ten Komponente über alle Methoden

k : Anzahl der Komponenten.

$$EWMC_i := Max_i(ev(M_{i1}),...,ev(M_{ij}))$$

mit ev(M_{ij}) : essentielle Komplexität der j-ten Methode in der i-ten Komponente

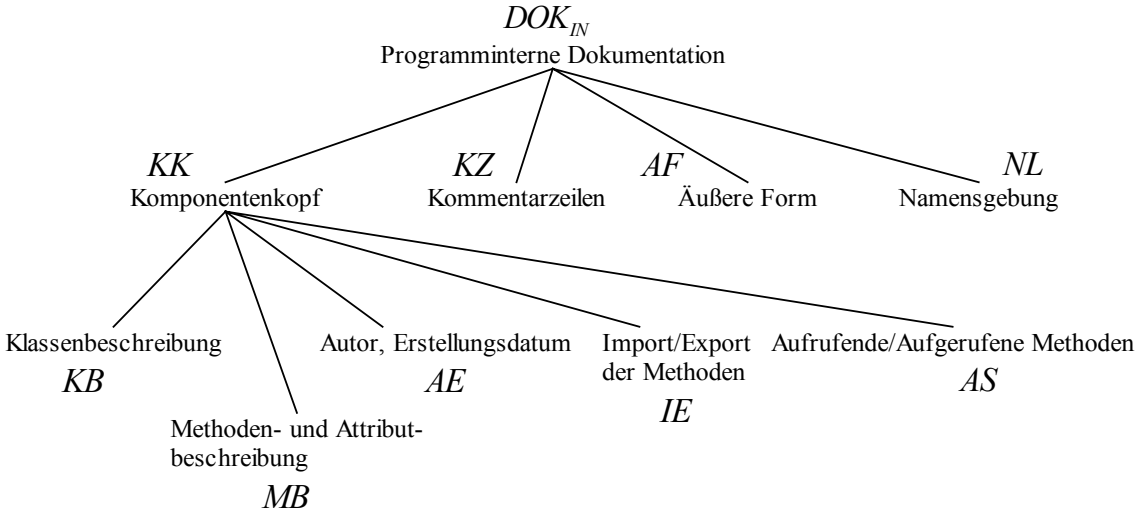
Dokumentation

Die Teilmaße zur Standart- und Portierungsdokumentation sind unabhängig vom zugrunde liegenden Programmierparadigma, da sie lediglich das Vorhandensein von Dokumentationsteilen feststellen und

keinen direkten Messungen am Quellcode vornehmen. Die programminterne Dokumentation vermißt hingegen das Vorhandensein von Dokumentationsteilen innerhalb der Quellen und muß daher an objektorientierte Software-Systeme angepaßt werden. Im vorgestellten Portabilitätsmaß werden der Komponentenkopf, Kommentarzeilen, die äußere Form und die Namensgebung geprüft.

Der Komponentenkopf sollte zusätzlich zur Methoden- und Attributbeschreibung auch eine entsprechende Klassenbeschreibung enthalten, sodaß hier ein weiteres binäres Maß dazugenommen werden sollte. Außerdem müssen Attribute und Methoden als Klassenkomponenten berücksichtigt werden. Alle anderen Maße lassen sich ebenfalls ohne Änderung auf objektorientierte Software-Systeme übertragen.

Aufbau Teilmaß ’Programminterne Dokumentation’ für objektorientierte Software-Systeme:



Formal:

$$DOK_{IN} = g_1 \cdot KK + g_2 \cdot KZ + g_3 \cdot AF + g_4 \cdot NL \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1.$$

Komponentenkopf:

$$KK = g_1 \cdot KB + g_2 \cdot MB + g_3 \cdot AE + g_4 \cdot IE + g_5 \cdot AS \quad \text{wobei gilt: } \sum_{i=1}^{5} g_i = 1.$$

Klassenbeschreibung:

$KB =$ Klassenbeschreibung vorhanden / nicht vorhanden.

Methoden- und Attributbeschreibung:

$MB =$ Beschreibung der Klassenkomponenten vorhanden/nicht vorhanden.

Autor, Erstellungsdatum:

AE = Autor und Erstellungsdatum vorhanden/nicht vorhanden.

Import/Export der Methoden:

IE = Auflistung Import/Export von Methoden vorhanden/nicht vorhanden.

Aufrufende/Aufgerufene Klassenkomponenten:

AS = Auflistung der abhängigen Methoden vorhanden/nicht vorhanden.

4 Portabilitätsanalyse am Beispiel des XCTL-Systems

Auf dem Fundament der theoretischen Ausführungen der vorangegangenen Kapitel, soll nun die Portabilitätsanalyse für ein reales Software-System durchgeführt werden. Bei dem Fallbeispiel handelt es sich um ein Windows-basiertes Software-System, zur Steuerung einer Röntgen-Topographie-Kamera. Nach kurzer Einführung in den Gegenstandsbereich und Vorstellung des Software-Systems, wird die Portabilitätsanalyse durchgeführt. Hierbei wird das Software-System auf der Grundlage des vorgestellten Portabilitätsmaßes untersucht und es werden Aussagen zur Portabilität des Systems getroffen. Den Abschluß dieses Kapitels bildet die Darstellung der Ergebnisse der Portabilitätsanalyse und deren Interpretation für das betrachtete Software-System.

4.1 Das XCTL-System

Bei dem zu analysierenden Software-System handelt es sich um ein, am Institut für Physik der Humboldt-Universität zu Berlin, entwickeltes Programm, zur Steuerung einer Röntgen-Topographie-Kamera. Mit Hilfe des Software-Systems, bezeichnet als XCTL-System (Control mittels X-Strahlung), werden Meßprozesse durchgeführt, die Rückschlüsse auf die Güte und den Aufbau eines Halbleiters zulassen. Das XCTL-System hat primär die Aufgabe, die Geräte eines Versuchsaufbaus zu steuern und zu koordinieren (Stellmotoren, Detektoren, Kollimatoren) sowie die Meßdaten der verwendeten Gerätschaften zu erfassen und auszuwerten.

Mängel in der Funktionalität und fehlerhafte Ausführung von Programmteilen, führten zur Herausbildung eines Software-Sanierungsprojektes, welches vorhandene Mängel, wie z.B. unzureichende Dokumentation des Systems und der Quellen, Fehleranfälligkeit oder schwierige Einbindung zusätzlicher Hardware-Komponenten, bzw. Anpassung an aktuelle System-Software, auf der Grundlage von Prinzipien und Methoden der modernen Software-Technik beseitigen soll. Das Ziel der Software-Sanierung des XCTL-Systems besteht darin, sämtliche Mängel aus Benutzer- und Entwicklersicht zu beseitigen und das Software-System in seinen Qualitätsmerkmalen zu optimieren. Hierzu gehört u.a. auch die Übertragbarkeit des Systems auf unterschiedliche bzw. neue System-Umgebungen. Eine detaillierte Darstellung des Gegenstandsbereiches und des Projektes finden sich in [Bothe 01a], [Bothe 01b] sowie unter [Xctl 01].

4.1.1 Quellcode und Programmiersprachen

Ausgangspunkt für die Betrachtung des Software-Systems ist das, 1998 vom Institut für Physik übergebene und wenig kommentierte, XCTL-Projekt. Der Quelltext des XCTL-Systems besteht danach aus annähernd 28000 Codezeilen, verteilt auf mehr als 50 Quelldateien und ist auf Binärebene in insgesamt 3 dynamischen Laufzeitbibliotheken, plus der ausführbaren Programmdatei, strukturiert. In erster Linie wird als Implementierungssprache C++ verwendet, einige Teilfunktionen des Systems sind in C implementiert. Anhang D gibt eine Übersicht zu den Quelldateien und deren Funktionen innerhalb des XCTL-Systems [Freund 01] und [Schützler 01]. Die Quellen wurden im Laufe der Software-Sanierung teilweise restrukturiert und erweitert, sodaß einige Quelldateien hinzugenommen bzw. umbenannt wurden. Den aktuellen Entwicklungsstand findet man unter [Xctl 01].

4.1.2 Software-Umgebung

Die Laufzeitumgebung für das XCTL-System wird durch das Betriebssystem Windows 3.1 zur Verfügung gestellt. Es bildet die Grundlage für die Implementierung des Systems und stellt somit die Hostumgebung dar. Das für die Entwicklung ursprünglich eingesetzte Software-System war das bereits vorgestellte Borland C++ in der Version 4.0. Auch diese läuft in der 16-Bit-Umgebung von Windows 3.1 und stellt dessen Funktionalität optional, in Form einer objektorientierten Systembibliothek (OWL), zur Verfügung. Bei der Implementierung des XCTL-Systems wurde jedoch auf dieses Abstraktionsniveau verzichtet und direkt auf der Schnittstelle des Betriebssystems aufgesetzt (Windows-API).

Im Prozeß der Software-Sanierung des XCTL-Systems wurde das Projekt in aktuelle Versionen der Entwicklungsumgebung (Borland C++ 4.5 und 5.02) überführt und somit für erweiterte Laufzeituntersuchungen zugänglich. Die Version 4.5 stellt hierzu anwenderfreundliche Debugging-Werkzeuge für 16-Bit-Anwendungen zur Verfügung. Die Version 5.02 hält für 32-Bit-Anwendungen zwar den gleichen Funktionsumfang bereit, ist aber nicht vollständig abwärtskompatibel zu 16-Bit-Anwendungen (Ressourcen, Debugging-Werkzeuge etc.). Grundlage für das 16-Bit-XCTL-System ist somit die Entwicklungsumgebung Borland C++, Version 4.5.

4.1.3 Hardware-Umgebung

Die Arbeitsplätze für das XCTL-System sind mit einer Reihe von PC's ausgestattet (siehe Anhang E). Die Hardwareansteuerung im XCTL-System erfolgt innerhalb der Laufzeitbibliotheken 'motors.dll' und 'counters.dll'. Hier befindet sich zum einen die Schnittstelle zum Ansteuern der Motoren und zum anderen die Einbindung und Ansteuerung der Detektoren. Jeder Meßplatzrechner verfügt über eine Motoren-Controllerkarte vom Typ C-812 und C-832, die jeweils mehrere Gleichstrommotoren steuern, welche u.a. für die Bewegungen des Probenhalters, des Kollimators und des Detektors verantwortlich sind. Der Befehlsaustausch zwischen dem Hostrechner und der Karte C-812 kann optional über die RS232-Schnittstelle, den PC-Bus (ISA) oder der IEEE488-Schnittstelle erfolgen. Die beiden letzteren Kommunikationsarten sind im XCTL-System wahlweise implementiert. Im Falle der Kommunikation über den PC-Bus (Konfiguration über XCTL-Initialisierungsdatei) wird ein gemeinsam genutzter Speicherbereich (Dual-Port-RAM) für die Befehlsübergabe eingerichtet und als Kommunikationskanal verwendet. Wird die IEEE488-Schnittstelle für Steuerbefehle eingesetzt, erfolgt die Ansteuerung durch eine programmierfreundliche C-Schnittstelle, die in Form einer 16-Bit Laufzeitbibliothek (Treiber) realisiert wird (win488.dll). C-832 bietet im Vergleich hierzu nur den Datenaustausch über I/O-Adreßbereiche, mittels Port-Befehlen. Die Funktionen für den Zugriff auf die Motoren-Hardware werden in der Datei 'motor.cpp' definiert.

Bei der vom XCTL-System angesteuerten Detektoren-Hardware handelt es sich entweder um 0-, 1- oder 2-dimensionale Detektoren. Diese werden jeweils über geeignete Schnittstellenkarten des Herstellers betrieben und über Port-Befehle in I/O-Adreßbreichen gesteuert. Folgende Detektoren werden angesteuert, bzw. sollen zusätzlich eingebunden werden:

- 0-dimensionale Detektoren:
  - RADICON SCSCS (Scintillation Counter Single-Channel Spectrometer);

Sogenannter Szintillationsdetektor mit eigener Schnittstellenkarte (im PC) und einem 32-Bit Zähler (Kanal);
  - russ. SCSCS (Scintillation Counter Single-Channel Spectrometer); Szintillationsdetektor ohne eigene Schnittstellenkarte;
  - AXIOM AX5216; Schnittstellenkarte (im PC); Fünf 16-Bit Zähler, integriert auf einem IC Am9513; Sinnvoll einsetzbar bei Messungen mit 0-dim Detektoren, wird bspw. für den russ. SCSCS benutzt;
- 1-dimensionale Detektoren:
  - Braun-PSD (Position Sensitive Detector) der Firma Braun, besitzt eigene Schnittstellenkarte (im PC);
  - Stoe-PSD, PSD der Firma Stoe;
  - Einzeiliger CCD (CCD: Charge Couple Device, Ladungsgekoppeltes Gerät); Schnittstellenkarte (im PC) ???; Halbleiterdetektor?;
- 2-dimensionale Detektoren:
  - CCD von Proscan, Schnittstellenkarte (im PC); Neues Gerät, noch keine Details;
  - CCD Eigenbau, nur offline, keine Schnittstellenkarte und geringere Auflösung als CCD von Proscan.

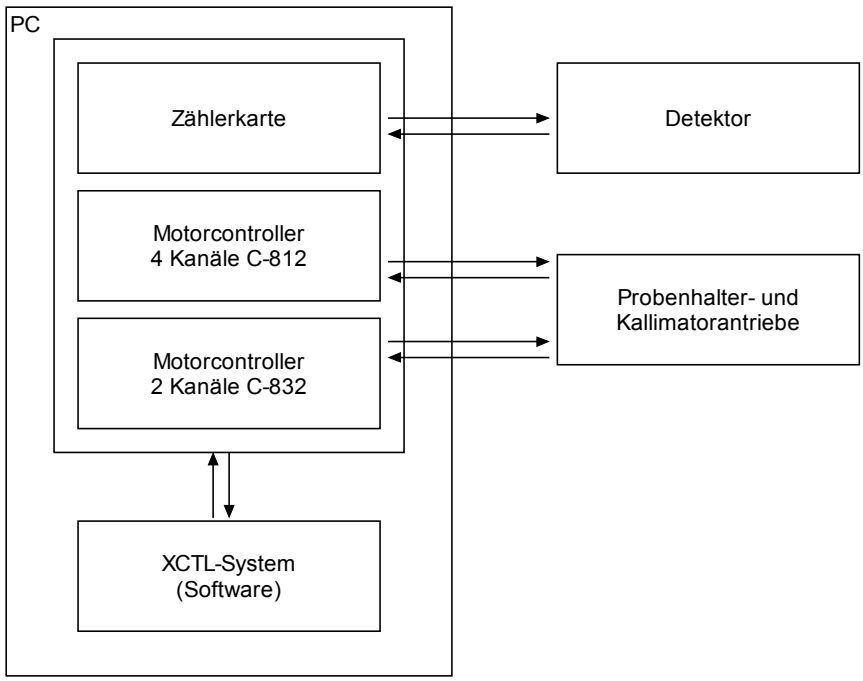


Abbildung 24: Meßplatzschema mit Arbeitsplatz-PC und Peripherie

4.2 Anwendung des Portabilitätsmaßes auf das XCTL-System

In diesem Abschnitt geht es um die Ermittlung des Qualitätsmerkmals ’Portabilität’ des XCTL-Systems. Auf der Grundlage des vorgestellten Portabilitätsmaßes wird hierbei der Versuch unternommen, aufgrund von gemessenen Kenngrößen, richtungsweisende Aussagen zur Portabilität des analysierten Software-Systems zu machen. Ausgehend von der in Abbildung 25 dargestellten

Maßbaumhierarchie, werden die einzelnen Einflußgrößen bestimmt und zu einem Aufwandsmaß für die Portierung aggregiert.

Abbildung 25: Übersicht Maßbaum für das Qualitätsmerkmal ’Portabilität’

Da der Portierungsaufwand nur indirekt über dessen Einflußfaktoren bestimmbar ist, existiert ein funktionaler Zusammenhang der Form:

$$Aufwand_{port} = f_1(F_1, F_2, F_3) \quad \text{mit:}$$

F_1 : System-Umgebung

F_2 : Dokumentation

F_3 : Komplexität.

Die einzelnen Einflußfaktoren sind in Subfaktoren zerlegbar und es ergibt sich auch hier eine funktionale Abhängigkeit der Form:

$$F_i = f_{2i}(SF_{iL},...,SF_{ik}) \quad \text{mit:}$$
$$SF_{ij} = j\text{-ter Subfaktor des Faktors } F_i \quad \text{mit:}$$
$$i = 1,...,3 \text{ und } j = 1,...,k.$$

Die Verfeinerung wird in dem Maße fortgeführt, bis die jeweiligen Subfaktoren direkt meßbar sind und damit die unterste Hierarchieebene erreicht worden ist. Für die Subfaktoren dieser Ebene werden entsprechende Maße aufgestellt:

$$SM_{ij} = \text{ Submaß für den } j\text{-ten Subfaktor } SF_{ij} \text{ des Faktors } F_i \quad \text{mit:}$$
$$i = 1,...,n \text{ und } j = 1,...,m.$$

Die Ermittlung des Portierungsmaßes erfolgt dann in umgekehrter Richtung zur obersten Ebene der Einflußfaktor-Hierarchie und geschieht durch Aggregation der Submaße zum Faktormaß, den die entsprechenden Subfaktoren bestimmen (bspw. gewichtete Addition der Meßwerte). Damit existiert zu jeder Hierarchieebene ein entsprechendes Maß.

$$M_j = g_1 \cdot SM_{i1} + ... + g_n \cdot SM_{in}.$$

Im folgenden wird diese allgemeine Vorgehensweise auf alle drei Einflußgrößen angewendet und für das XCTL-System, durch entsprechende Messungen realisiert. Bei den Messungen zur System-Umgebung werden die Messungen direkt an den Quellen durchgeführt und entsprechen den Techniken einer syntaktischen Analyse eines Compilers, d.h. sie können automatisch durchgeführt werden. Daneben kann die Vermessung des Einflußfaktors 'Komplexität' ebenfalls mit entsprechenden Werkzeugen durchgeführt werden, wobei hierbei ebenso der Quelltext des Software-Systems zugrunde gelegt wird.

Die Einflußgröße 'Dokumentation' kann nicht automatisch auf ihre Qualitätsaspekte hin untersucht werden. Diese kann aber wesentlichen Einfluß auf den Portierungsaufwand haben. Die Messung erfolgt durch manuelle Inspektion in Form eines sogenannten 'Code-Reviews'. Hierbei sind alle wesentliche Dokumentationsteile in Form einer Checkliste aufgeführt. Dieses Vorgehen setzt jedoch bestimmte Kenntnisse des Porteurs voraus, die eine gewisse Subjektivität in das Maß einbringen und deren Bestimmung, einen gewissen Aufwand erfordert. Es entsteht hierbei ein notwendiger Kompromiß zwischen geforderter Objektivität und Ökonomie des Portierungsmaßes.

Die Güte von Portierungswerkzeugen zeigt sich erst nach durchgeführter Portierung. Vor der Portierung läßt sich nur deren Existenz feststellen. Für die genaue Beurteilung von Portierungswerkzeugen ist jedoch ein intensives Einarbeiten notwendig. Aus diesem Grund ist hierfür keine direkte Messung vorgesehen. Die Portierungswerkzeuge gehen jedoch über die bewerteten Änderungsaufwände indirekt in das Portierungsmaß ein, spielen aber bei der Vermessung des XCTL-Systems keine Rolle, da hier noch keine entsprechenden Portierungserfahrungen einer Erstportierung vorliegen.

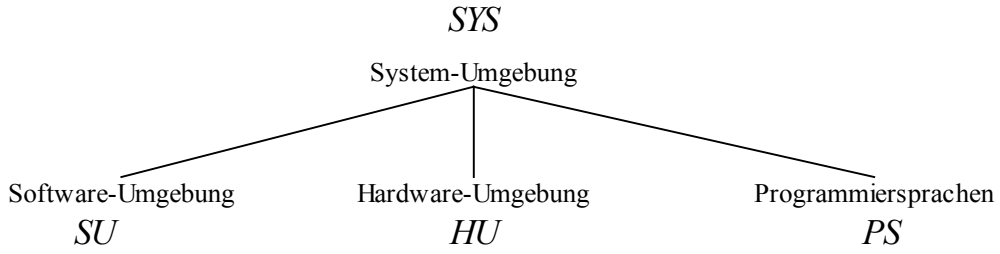
Die Messungen erfolgen statisch, sodaß hierbei Unterschiede, die sich erst zur Laufzeit auswirken(z.B. unterschiedliche Fehlerbehandlung auf Betriebssystemebene), nicht erfaßt werden.

Bei der XCTL-Analyse werden die aufgezeigten Problemfelder bei Windows-Portierungen, von 16- nach 32-Bit, wie folgt auf die Portabilitätsmaße abgebildet:

- Benutzung eines virtuellen, linearen 32-Bit-Adressraumes -> MAM_1;
- Strikte Separation aller Prozesse bzw. ihrer Adreßräume voneinander -> MAM_1:
- Globale Änderungen am Modell für Benutzereingaben über Tastatur und Maus -> $BIBM_1$;
- Erweitertes Koordinatensystem und weitere Änderungen im GDI -> $BIBM_1$;
- Wegfall MS-DOS- -> BSM_1 und 80x86-spezifischer Aufrufe -> MAM_1, PGM_1;
- Benutzung undokumentierter Win16-Eigenschaften -> $BIBM_1$;
- Probleme bei der Nutzung von Dll's, aufgrund von Architekturunterschieden-> MAM_1.

4.2.1 System-Umgebung

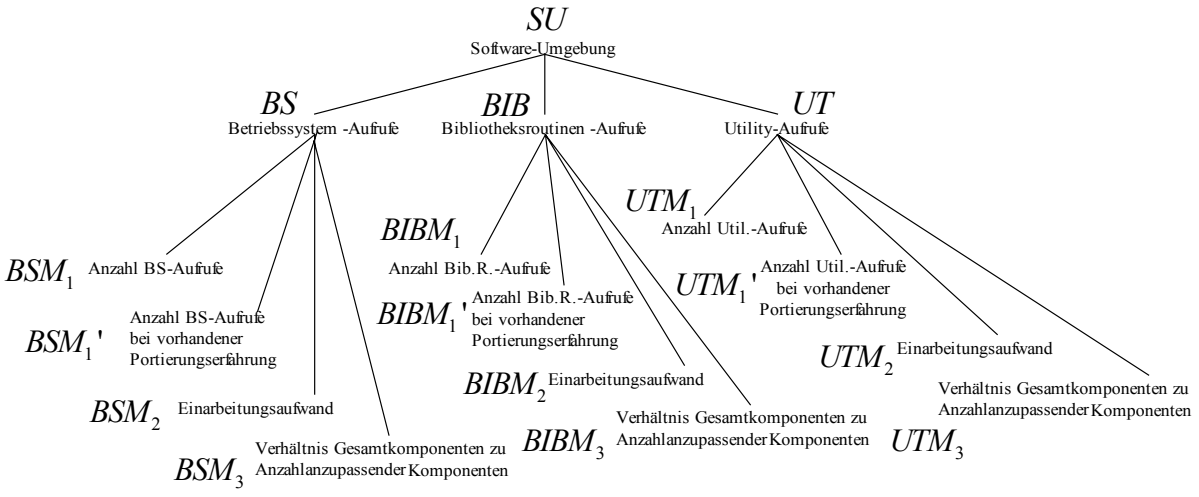
In diesem Abschnitt werden die Codeteile des XCTL-Systems ermittelt, die aufgrund von Unterschieden der Host- und Zielumgebung anzupassen sind. Hierbei wird die System-Umgebung in die Komponenten Software-Umgebung, Hardware-Umgebung und Programmiersprachen unterteilt, da sich die System-Umgebung eher in den einzelnen Komponenten ändert, statt vollständig. Dadurch lassen sich bestimmte Portierungsprobleme den Verursachern besser zuordnen.



Schritte zur Ermittlung des Teilmaßes 'System-Umgebung':

- Ermittlung der Maßkomponente 'Software-Umgebung';
- Ermittlung der Maßkomponente 'Hardware-Umgebung';
- Ermittlung der Maßkomponente 'Programmiersprachen';
- Zusammenfassung zur Maßkomponente 'System-Umgebung'.

4.2.1.1 Software-Umgebung



Schritte zur Ermittlung des Teilmaßes ’Software-Umgebung’:

- Ermittlung der Maßkomponente ‘Betriebssystem-Aufrufe’;
- Ermittlung der Maßkomponente ‚Bibliotheksroutinen-Aufrufe’;
- Ermittlung der Maßkomponente ‘Utility-Aufrufe’;
- Zusammenfassung zur Maßkomponente ’Software-Umgebung’.

Der Wechsel der Software-Umgebung führt meist zu Problemen, wenn das zu portierende Software-System, Zugriffe auf die Umgebung in Form von Systemaufrufen realisiert. Dazu zählen:

- Betriebssystem-Aufrufe, beispielsweise für die Ausgabe von Zeichen;
- Bibliotheksroutinen-Aufrufe, wie z.B. Objektbibliotheken von Funktionen zur Ansteuerung einer grafischen Benutzeroberfläche etc.;
- Hilfsprogramm-Aufrufe, z.B. zur Berechnung math. Funktionen oder Bereitstellung bestimmter Sortieralgorithmen.

Betriebssystem-Aufrufe

Schritte zur Ermittlung der Maßkomponente ’Betriebssystem-Aufrufe’:

- Ermittlung Anzahl unterschiedlicher BS-Aufrufe (BSM_1) Alternativ BSM_1';
- Porttool auswerten;
- Berechnung BSM_1;
- Ermittlung der Komponenten mit jew. Gesamtzahl (BSM_2);
- Porttool auswerten;
- Berechnung BSM_2;
- Komponentenverhältnis berechnen (BSM_3);
- Berechnung der Maßkomponente BS bzw. BS'.

Anpassungen in diesem Bereich fallen immer dann an, wenn:

- beide Betriebssysteme (Host- und Zielumgebung) von der Funktionalität her identische Funktionen zur Verfügung stellen, diese jedoch einen anderen Funktionsnamen oder eine veränderte Parameterliste besitzen, bzw. umgekehrt Funktionen mit identischem Funktionsnamen, plötzlich unterschiedliche Funktionalität aufweisen;
- auf dem Zielsystem die erforderlichen Funktionen nicht mehr zur Verfügung stehen.

In einem ersten Schritt wird zunächst die Anzahl der unterschiedlichen Betriebssystem-Aufrufe bestimmt. Hierzu braucht der Porteur spezielle Kenntnisse der Host- und Zielumgebung, die jedoch in einer Portabilitätsbibliothek für die syntaktische Analyse gesammelt und durch ein entsprechendes Werkzeug verfügbar gemacht werden können. Solch ein Werkzeug ist das bereits vorgestellte Porttool mit der zugehörigen Portierungs- bzw. Problembibliothek 'Port.ini', siehe Anhang B. Das erste zu ermittelnde Maß lautet:

$BSM_1 = Anz.BSR_1 + ... + Anz.BSR_n$ mit:
$\mathrm{BSR_i}$: i-ter unterschiedlicher BS-Aufruf, $i = 1,...,n$, (vergleiche Kapitel 3.2.3.4).

Nach der Anwendung von Porttool auf die Quellen und Auswertung der Ergebnisse, ergibt sich für die Anzahl der Betriebssystemaufrufe:

$BSM_1 = Anz.BSR_1 + ... + Anz.BSR_n = 3$, (siehe Anhang F).

Wird eine Portierung vorgenommen, fällt dieses Maß aufgrund von Erfahrungswerten, deutlich differenzierter aus. Die aus Portierungserfahrungen gewonnenen Aufwandswerte müssen dann für weitere und somit genauere Aufwandsabschätzungen, nutzbar gemacht werden. Hierbei werden nicht nur Aufwandswerte dokumentiert, sondern auch, wie konkrete Änderungen durchzuführen sind. Hierzu zählt ebenfalls die Dokumentation der verwendeten Werkzeuge. Weiterhin kann es sich zeigen, daß manche Aufrufe, nicht an das Zielsystem angepaßt werden können und eine Portierung deshalb nicht realisierbar ist. Darüber hinaus ist zu beachten, daß der Aufwand bei der ersten Anpassung wesentlich größer ist, als im Vergleich zu späteren Portierungen. Hier liegen die erforderlichen Kenntnisse von vorangegangenen Portierungen bereits vor. Die Werte für eine erste Anpassung sollten demnach im differenzierten Maß nicht erfaßt werden. Das entsprechende Maß für die Betriebssystem-Aufrufe lautet wie folgt:

$BSM_1 = (KA_{BS}, PA_{BS}, SA_{BS}, BA_{BS})$ mit:
$KA_{BS} = (BSR_1, ..., BSR_k)$
Namen der BS-Routinen, die nicht angepaßt werden können
$\mathrm{PA_{BS}} = ((Anz.BSR_1, BSR_1), ..., (Anz.BSR_m, BSR_m)$
Namen und Anzahl der BS-Routinen, über die während der Messung keine Information vorliegt
$SA_{BS} = (Anz.BSR_1, ..., Anz.BSR_n)$
Gesamtzahl der Aufrufe von BS-Routinen, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$$\mathrm{BA}_{\mathrm{BS}} = Anz.BSR_1 \cdot B_1 + \ldots + Anz.BSR_r \cdot B_r$$

Summe der bewerteten BS-Routinen, für die ein Aufwand vorliegt

B_i : Aufwand für die Anpassung der i – ten BS-Routine, $i = 1,\ldots,r$

KA : Keine Anpassungen

PA : Potentielle Anpassungen

SA : Sichere Anpassungen

BA : Bewertete Anpassungen.

Auch diese Teile des differenzierten Maßes sind mit Hilfe von geeigneten Werkzeugen zu realisieren, d.h. sollten wegen der sonst fehlenden Objektivität des Maßes, automatisch ermittelt werden. Hierzu ist eine Datenbasis der aufrufbaren Routinen des Host-Betriebssystems notwendig. Die Bewertung erfolgt dann unter Nutzung der vorhandenen Aufwandsdaten. Dieser Teil des Aufwandsmaßes ist jedoch erst für spätere Phasen der Portierung relevant und hat bei der hier durchgeführten Analyse des XCTL-Systems zunächst keine Bedeutung. Ist das XCTL-System erfolgreich portiert worden, werden die Portierungserfahrungen durch dieses Maß ausgedrückt, womit sich die Genauigkeit des Aufwandsmaßes deutlich verbessert.

Im zweiten Schritt werden die von Änderungen betroffenen Komponenten eines Software-Systems, mit der darin befindlichen Anzahl von anzupassenden Betriebssystem-Aufrufen, erfaßt. Dieses Maß ist deshalb erforderlich, da es für einen Porteur notwendig ist, sich in die betroffenen Programmkomponenten einzuarbeiten. Sind die anzupassenden Aufrufe über viele Komponenten zerstreut, ist auch der Einarbeitungsaufwand größer. Ziel des zweiten BS-Aufruf-Maßes ist es, Aussagen über den Einarbeitungsaufwand und wichtige Entscheidungen, z.B. in Richtung einer Neuimplementation. zu ermöglichen:

$$BSM_2 = ((Komp_1, Anz.BSA),\ldots,(KOMP_n, Anz.BSA)) \quad \text{mit:}$$

$KOMP_i$: i-te betroffene Komponente des Programms, $i = 1,\ldots,n$

BSA : anzupassender Aufruf einer BS-Routine.

Für die Ermittlung des zweiten BS-Aufruf-Maßes BSM_2 werden die einzelnen Systemkomponenten des XCTL-Systems identifiziert, in denen die Betriebssystem-Aufrufe stattfinden. Hiermit sollen Aussagen über den Einarbeitungsaufwand ermöglicht werden. Weiterhin kann anhand dieses Maßes die Entscheidung getroffen werden, ob eine bestimmte Systemkomponente, mit sehr vielen BS-Aufrufen, nicht besser neu implementiert, statt angepaßt werden sollte. Auch ist der Einarbeitungsaufwand höher, je mehr die anzupassenden Aufrufe über die Komponenten des Software-Systems verteilt sind. Die Summe der Änderungskandidaten ergibt sich aus den ermittelten Werten der Porttool-Analyse und kann somit automatisch bestimmt werden.

$$BSM_2 = ((KOMP_1, Anz.BSA),\ldots,(KOMP_n, Anz.BSA)) = ((M\_arscan.cpp, 1),(M\_data.cpp, 2))$$

Der dritte Schritt besteht darin, das Verhältnis der von Änderungen betroffenen Komponenten zur Gesamtkomponentenzahl, zu ermitteln. Damit werden weitere Informationen für eine Entscheidung (’Portierung oder Neuimplementierung) geliefert. Je näher sich das Verhältnis dem Wert ’1’ nähert, desto günstiger wird das Verhältnis für eine Neuimplementierung..

$$BSM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BS\text{-}Anpassungen}{Anzahl\ aller\ Komponenten}.$$

Die Anzahl aller Komponenten für das XCTL-System ergibt sich aus einer Messung mit dem McCabe-Tool [Gollnick 99], welches u.a. für die Ermittlung von Metriken für Software-Systeme dient. Die Messung ergab für die Anzahl der Komponenten den Wert 80. Da das XCTL-System nicht durchgängig objektorientiert programmiert ist, ergibt sich das Problem der Zuordnung globaler Funktionen und Daten, die in keiner Systemkomponente untergebracht sind. Solche Komponenten werden für die Berechnung in einer zusätzlichen, gedachten Systemkomponente 'Tx' untergebracht und gehen somit in das Umgebungsmaß ein. Die Anzahl aller Komponenten erhöht sich damit auf den Wert 81.

$$BSM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BS\text{-}Anpassungen}{Anzahl\ aller\ Komponenten} = \frac{2}{81} = 0,025.$$

Der vierte und letzte Schritt besteht darin, die einzelnen Maße für die Betriebssystemaufrufe zum Gesamtmaß BS bzw. BS' zu aggregieren. Dabei ist die Zusammensetzung zu einem einzigen Wert nur dann möglich, wenn die Maße gleiche Objekte messen. Diese könnten dann z.B. mittels Addition verknüpft werden. Da aber die aufgestellten Maße sehr unterschiedliche Objekte messen, ist eine Addition in diesem Falle nicht möglich, sodaß lediglich eine Gegenüberstellung der Maße in einem 3-Tupel sinnvoll ist:

$$BS = (BSM_1{*}, BSM_2, BSM_3) \quad \text{mit:}$$
$$\mathrm{BS_1}{*}:\ BS_1 \text{ oder } BS_1', \text{ abhängig von der gegebenen Situation}.$$

Für das XCTL-System hat man nun alle drei Aussagen zu den Betriebssystem-Aufrufen auf einen Blick. Es ist somit leicht automatisch zu erstellen, da lediglich vorhandene Informationen zusammengetragen werden müssen:

$$BS = (BSM_1{*}, BSM_2, BSM_3) = (3, ((\mathrm{TAreaScan}, 1), (\mathrm{TBitmapSource}, 2)), 0,025).$$

Bibliotheksroutinen-Aufrufe

Schritte zur Ermittlung der Maßkomponente 'Bibliotheksroutinen-Aufrufe':

- Ermittlung Anzahl unterschiedlicher BS-Aufrufe ($BIBM_1$) Alternativ $BIBM_1'$;
- Porttool auswerten;
- Berechnung $BIBM_1$;
- Ermittlung der Komponenten mit jew. Gesamtzahl ($BIBM_2$);
- Porttool auswerten;
- Berechnung $BIBM_2$;
- Komponentenverhältnis berechnen ($BIBM_3$);
- Berechnung der Maßkomponente BIB bzw. BIB'.

Bei Anpassungen in Hinblick auf Bibliotheksroutinen, herrscht in etwa die gleiche Situation, wie bei den Betriebssystem-Aufrufen. Auch hier werden zunächst die Anzahl der anzupassenden Aufrufe von Bibliotheksroutinen ermittelt. Danach wird, wie zuvor bei den Betriebssystem-Aufrufen, die Verteilung auf die Komponenten berechnet. Zum Schluß wird dann das Verhältnis zur Gesamtkomponentenzahl gebildet.

Stehen außer dem Verzeichnis der Bibliotheksroutinen, die an das Zielsystem anzupassen sind (Inhalt der zentralen Problembibliothek), keine weiteren Informationen zur Verfügung, weil etwa noch keine Portierung stattgefunden hat, so wird in einem ersten Schritt, nur die Anzahl potentieller Änderungen ermittelt:

$$BIBM_1 = Anz.BIBR_1 + ... + Anz.BIBR_n \quad \text{mit:}$$
$$BIBR_i : i\text{-ter unterschiedlicher BIB-Aufruf, } i = 1,...,n.$$

Die anzupassenden Bibliotheksroutinen werden, sofern sie in der zentralen Problembibliothek erfaßt sind, automatisch durch das Porttool-Werkzeug ermittelt und manuell in das Maß $BIBM_1$ eingebracht. Der ermittelte Wert für das XCTL-System beträgt:

$$BIBM_1 = Anz.BIBR_1 + ... + Anz.BIBR_n = 353 \text{ , (siehe Anhang F).}$$

Nach erfolgter Portierung sind die durchgeführten bzw. notwendigen Anpassungen bekannt und werden in einem differenzierten Maß $BIBM_1'$ erfaßt. Hierbei wird die Annahme zugrunde gelegt, daß eine Bibliotheksroutine, über die keine Informationen existiert, grundsätzlich entwickelt werden kann und unterscheidet sich somit von den Betriebssystem-Aufrufen, die zumeist nicht modifiziert oder selbst entwickelt werden können bzw. nicht modifiziert werden sollten. Bibliotheksroutinen, die im Laufe der Portierung entwickelt worden sind, können bei einer weiteren Portierung eingesetzt werden und müssen im Aufwandsmaß deshalb nicht mehr berücksichtigt werden. Das differenzierte Maß $BIBM_1'$ für Bibliotheksroutinen hat folgenden Aufbau:

$$BIBM_1' = (SA_{BIB}, BA_{BIB}) \quad \text{mit:}$$
$$\mathrm{SA_{BIB}} = Anz.BIBR_1 + ... + Anz.BIBR_m$$
Gesamtzahl der Aufrufe von Bibliotheksroutinen,
zu denen keine Aufwände vorliegen
$$\mathrm{BA_{BIB}} = Anz.BIBR_1 \cdot B_1 + ... + Anz.BIBR_n \cdot B_n$$
Summe der bewerteten Bibliotheksroutinen für die
ein Aufwand vorliegt
B_i : Aufwand für die Anpassung des Aufrufs der i-ten
Bibliotheksroutine, $i = 1,...,n.$

Da für das XCTL-System, in dieser Phase, noch keine konkreten Portierungserfahrungen zugrunde gelegt werden können, ist dieses Maß noch nicht realisierbar.

Der zweite Schritt bei der Erfassung der Bibliotheksroutinen besteht in der Ermittlung der Anzahl der anzupassenden Aufrufe. Diese verursachen ebenfalls einen höheren Portierungsaufwand, wenn sie in vielen Komponenten verstreut sind. Deshalb wird auch hier die Häufigkeit des Auftretens von

potentiell anzupassenden Aufrufen einer Komponente gemessen, wobei jede Komponente einmal gezählt wird. Das Maß hierfür lautet:

$$BIBM_2 = ((KOMP_1, Anz.BIBA),...,(KOMP_n, Anz.BIBA)) \quad \text{mit:}$$

KOMP_i : i-te Komponente des Programms, $i = 1,...,n$

$BIBA$: anzupassender Aufruf einer Bibliotheksroutine.

Die Realisierung für das Maß $BIBM_2$ sieht damit wie folgt aus (siehe Anhang F):

$$BIBM_2 = ((KOMP_1, Anz.BIBA),...,(KOMP_n, Anz.BIBA)) =$$

((*TAbout* , 2),
(*TAdjustmentExecute* , 2),
(*TAdjustmentWindow* , 1),
(*TAm9513a* , 2),
(*TAngleControl* , 32),
(*TAquisition* , 2),
(*TAreaScan* , 13),
(*TAreaScanCmd* , 1),
(*TAreaScanParameters* , 1),
(*TBitmapSource* , 5),
(*TBraun_Psd* , 8),
(*TC_812* , 2),
(*TC_812GPIB* , 11),
(*TC_832* , 2),
(*TCalibrate* , 5),
(*TCalibratePsd* , 8),
(*TCmd* , 2),
(*TCommonDevParam* , 4),
(*TCounterShowParam* , 1),
(*TCounterWindow* , 2),
(*TCurve* , 2),
(*TDC_Drive* , 1),
(*TDevice* , 8),
(*TDList* , 1),
(*TEditWindow* , 3),
(*TEncoder* , 3),
(*TExecuteCmd* , 2),
(*TGenericDevice* , 3),
(*TMacroExecute* , 5),
(*TMain* , 3),
(*TMDIWindow* , 4),
(*TMList* , 2),
(*TModalDlg* , 3),
(*TModelessDlg* , 2),
(*TMotor* , 1),
(*TMotorParam* , 1),
(*TPlotData* , 2),
(*TPosControl* , 7),
(*TPsd* , 1),
(*TPsdRemoteSync* , 4),
(*TRadicon* , 6),
(*TScan* , 13),
(*TScanCmd* , 5),

```
(TScsParameters        ,   1),
(TSetAdjustmentParam   ,   1),
(TSetupAreaScan        ,   3),
(TSetupContinuousScan  ,   2),
(TSetupScanCmd         ,   1),
(TSetupStepScan        ,   6),
(TShowValueCmd         ,   3),
(TSteering             ,   9),
(TStoe_Psd             ,   2),
(TTopographyExecute    ,   8),
(Tx                    , 129)).
```

Auch im dritten Schritt stimmt der Maßaufbau mit dem der Betriebssystem-Aufrufe überein und definiert sich wie folgt:

$$BIBM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BIB\text{-}Anpassungen}{Anzahl\ aller\ Komponenten}.$$

Das Verhältnismaß berechnet sich für das XCTL-System mit:

$$BIBM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BIB\text{-}Anpassungen}{Anzahl\ aller\ Komponenten} = \frac{54}{81} = 0,667.$$

Im vierten Schritt erfolgt wieder die Aggregation des Maßes für die Bibliotheksroutinen. Dabei wird analog dem Maß für die Betriebssystem-Aufrufe zusammengefaßt, weil auch hier die Maße ähnlich (1.Maß) oder sogar gleich (2. und 3.Maß) sind.

$BIB = (BIBM_1{*}, BIBM_2, BIBM_3)$ mit:
$BIBM{*} = BIBM_1$ oder $BIBM_1'$, abhängig von der gegebenen Situation.

Damit ergibt sich für das Teilmaß der Bibliotheksroutinen im XCTL-Systems folgendes Ergebnis:

$$BIB = (BIBM_1{*}, BIBM_2, BIBM_3) = (353, BIBM_2, 0,667).$$

Utility-Aufrufe

Für das XCTL-System wurden keine Utility-Aufrufe identifiziert, die im Sinne des Zielsystems anzupassen wären. Die Probleme bei Utility-Aufrufen sind prinzipiell die gleichen wie bei den Bibliotheksroutinen, deshalb können die Ausführungen der zuvor vorgestellten Maße, auf die Utility-Aufrufe übertragen werden. Die Definition der Utiltity-Maße wurde bereits im Kapitel 3.2.3.4 dargestellt und soll hier nicht noch einmal wiederholt werden.

Aggregation zum Gesamtmaß 'Software-Umgebung'

Bei der Aggregation zum Gesamtmaß für die Software-Umgebung werden jeweils die zweite und dritte Stelle der drei Maße BS, BIB und UT zusammengefaßt, da sie für alle drei Maße gleich sind. Hierbei besteht die Möglichkeit, daß in einer Komponente unterschiedliche Aufrufarten vorkommen, so etwa von Bibliotheks- und Utility-Routinen. In diesem Fall werden die jeweiligen Anzahlen für

jede Komponente addiert. Der hierbei entstandene Informationsverlust besteht dann darin, daß nicht mehr einzelne Aufrufarten in das Maß eingehen, sondern nur noch pauschal die Aufrufe von Software-Umgebungs-Routinen.

In der ersten Position stimmen die Maße dann überein, wenn keine Portierungsinformationen vorliegen und zunächst nur die potentiellen Änderungen gezählt werden. Die Änderungen bzgl. der verschiedenen Aufrufarten sind disjunkt und lassen sich deshalb addieren. Im differenzierten Fall erfolgt die Aggregation, indem die Positionen der nicht-anpaßbaren Routinen und die Routinen, über die nichts bekannt ist, addiert werden und die ersten beiden Positionen des BS-Aufrufmaßes unverändert übernommen werden.

Falls noch keine Portierungsinformationen in das Maß für die Software-Umgebung einfließen, lautet dieses wie folgt:

$SU = (PAN_{SU}, KOMP_{SU}, VKG_{SU})$ mit:

$PAN_{SU} = BSM_1 + BIBM_1 + UTM_1$

Anzahl der potentiellen Anpassungen von Aufrufen von Routinen der Software-Umgebung

$\mathrm{KOMP_{SU}} = BSM_2 + BIBM_2 + UTM_2$

Namen der Komponenten mit Aufrufen von Routinen der Software-Umgebung und deren Anzahl

$\mathrm{VKG_{SU}} = BSM_3 + BIBM_3 + UTM_3$

Verhältnis der Komponentenzahl aus $KOMP_{SU}$ zur Gesamtzahl der Programmkomponenten.

Für das XCTL-System ergibt sich damit für die Software-Umgebung das Aufwandsmaß (siehe Anhang F):

$$SU = (PAN_{SU}, KOMP_{SU}, VKG_{SU}) = (356,\ KOMP_{SU},\ 0{,}691)$$

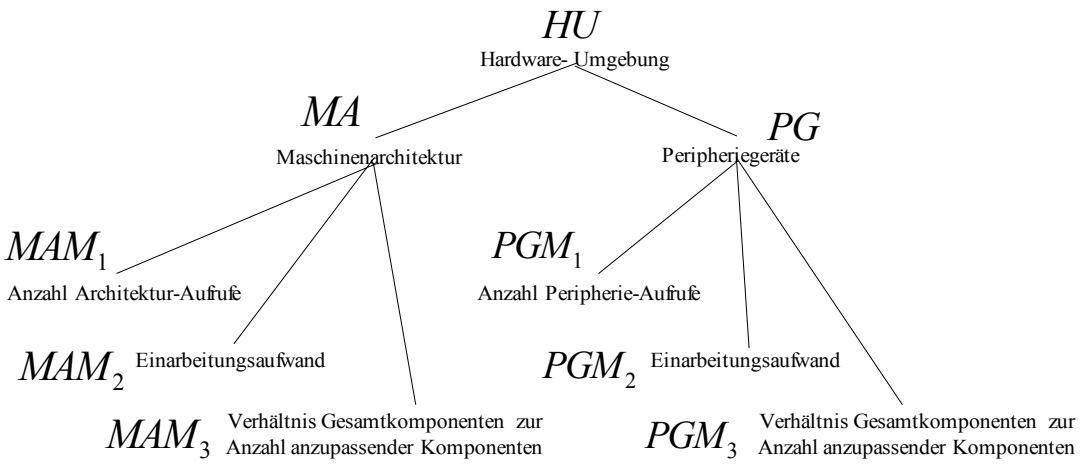
$$\begin{aligned}
\mathrm{KOMP_{SU}} = BSM_2 + BIBM_2 + UTM_2 = \\
((&TAbout &, &\ 2),\\
(&TAdjustmentExecute &, &\ 2),\\
(&TAdjustmentWindow &, &\ 1),\\
(&TAm9513a &, &\ 2),\\
(&TAngleControl &, &\ 32),\\
(&TAquisition &, &\ 2),\\
(&TAreaScan &, &\ 14),\\
(&TAreaScanCmd &, &\ 1),\\
(&TAreaScanParameters &, &\ 1),\\
(&TBitmapSource &, &\ 7),\\
(&TBraun_Psd &, &\ 8),\\
(&TC_812 &, &\ 2),\\
(&TC_812GPIB &, &\ 11),\\
(&TC_832 &, &\ 2),\\
(&TCalibrate &, &\ 5),\\
(&TCalibratePsd &, &\ 8),\\
(&TCmd &, &\ 2),\\
(&TCommonDevParam &, &\ 4),
\end{aligned}$$

```
(TCounterShowParam   ,   1),
(TCounterWindow      ,   2),
(TCurve              ,   2),
(TDC_Drive           ,   1),
(TDevice             ,   8),
(TDList              ,   1),
(TEditWindow         ,   3),
(TEncoder            ,   3),
(TExecuteCmd         ,   2),
(TGenericDevice      ,   3),
(TMacroExecute       ,   5),
(TMain               ,   3),
(TMDIWindow          ,   4),
(TMList              ,   2),
(TModalDlg           ,   3),
(TModelessDlg        ,   2),
(TMotor              ,   1),
(TMotorParam         ,   1),
(TPlotData           ,   2),
(TPosControl         ,   7),
(TPsd                ,   1),
(TPsdRemoteSync      ,   4),
(TRadicon            ,   6),
(TScan               ,  13),
(TScanCmd            ,   5),
(TScsParameters      ,   1),
(TSetAdjustmentParam ,   1),
(TSetupAreaScan      ,   3),
(TSetupContinuousScan,   2),
(TSetupScanCmd       ,   1),
(TSetupStepScan      ,   6),
(TShowValueCmd       ,   3),
(TSteering           ,   9),
(TStoe_Psd           ,   2),
(TTopographyExecute  ,   8),
(Tx                  , 129)).
```

Da noch keine Aufwände zu einer oder mehreren Komponenten der Software-Umgebung vorliegen, ist die Analyse für die Software-Umgebung des XCTL-Systems damit abgeschlossen. Bei vorhandenen Aufwänden wird das umfangreichere Maß SU' (siehe 3.2.3.4) für die Berechnungen herangezogen.

4.2.1.2 Hardware-Umgebung

In diesem Maß soll der Aufwand eingehen, der entsteht, wenn sich im Rahmen einer Portierung die Hardware-Umgebung für das zu portierende Software-System ändert. Dazu gehören in erster Linie die Rechner auf denen die Software läuft, aber auch die angeschlossenen Peripheriegeräte. Beide Möglichkeiten werden im Maß der Hardware-Umgebung getrennt erfaßt, da die Möglichkeit besteht, daß Host- und Zielrechner bei einer Portierung übereinstimmen, sich die Peripheriegeräte jedoch ändern. Weiterhin kann es auch sein, daß sich auf dem Zielrechner die Aufrufe für Zugriffe auf Peripheriegeräte ändern und somit anzupassen sind.



Schritte zur Ermittlung des Teilmaßes ’Hardware-Umgebung’:

- Ermittlung der Maßkomponente ‘Maschinenarchitektur’;
- Ermittlung der Maßkomponente ’Peripheriegeräte’;
- Zusammenfassung zur Maßkomponente ’Hardware-Umgebung’.

Maschinenarchitektur

Schritte zur Ermittlung der Maßkomponente ’Maschinenarchitektur’:

- Ermittlung Anzahl unterschiedlicher BS-Aufrufe (MAM_1) Alternativ MAM_1';
- Porttool auswerten;
- Berechnung MAM_1;
- Ermittlung der Komponenten mit jew. Gesamtzahl (MAM_2);
- Porttool auswerten;
- Berechnung MAM_2;
- Komponentenverhältnis berechnen (MAM_3);
- Berechnung der Maßkomponente MA .

Zu den entscheidenden Abhängigkeiten bei der Maschinenarchitektur gehören:

- Zugriffe auf Register, wie z.B. Steuer- oder Statusregister;
- Zugriffe auf Speicherstellen, z.B. in gemeinsam genutzten Speicherbereichen oder Bildschirmspeicher;
- Zugriffe auf Ports bzw. Kommunikation über Ports, z.B. um Mausabfragen zu realisieren;
- Aufrufe von Assemblerroutinen.

Hardwarezugriffe lassen sich bei einer statischen Untersuchung des Quellcodes, nicht immer eindeutig als notwendig zu ändernde Ausdrücke identifizieren. So ist bei statischen Untersuchungen oft nicht zu erkennen, auf welche Speicherstellen zugegriffen wird und ob hier Änderungen überhaupt erforderlich sind. Dieses darf als gewisse Eigenheit von Programmiersprachen angesehen werden und eine Aufwandsbewertung erscheint hier recht problematisch. Das Portierungsmaß ist von den Autoren des vorgestellten Maßes, im Kapitel 3.2.3, jedoch als ein erster allgemeiner Ansatz konzipiert worden, der zunächst nicht auf die Besonderheiten einzelner Programmiersprachen abzielt und damit nur die

Aufrufe ermittelt, die Aufwände verursachen können. Die Ermittlung der Aufrufe, die aufgrund von Maschinenabhängigkeiten zu Portierungsproblemen führen können, werden nun in einem ersten Schritt ermittelt. Die Vorgehensweise entspricht dabei dem, für die System-Umgebung. Als erstes wird Zahl der MA-Zugriffe für das XCTL-System ermittelt. Hierbei, wie auch bei den folgenden Maßen, gelten die zuvor festgelegten Zuordnungskriterien zwischen den ermittelten Problemklassen und den Aufwandsmaßen.

$$MAM_1 = Anz.MA_1 + ... + Anz.MA_n \quad \text{mit:}$$
$$MA_i: \ i\text{-ter unterschiedlicher MA-Zugriff, } i = 1,....,n.$$

Die Zahl der identifizierten Maschinenabhängigkeiten für das XCTL-System beträgt:

$$MAM_1 = Anz.MA_1 + ... + Anz.MA_n = 563 \text{, (siehe Anhang F).}$$

Da auch die Hardwareabhängigkeiten- bzw. zugriffe ein Einarbeiten in die Quellen erforderlich machen, sind hier die Komponentenmaße ebenfalls sinnvoll und lauten wie folgt:

$$MAM_2 = ((KOMP_1, Anz.MAA),...,(KOMP_n, Anz.MAA)) \quad \text{mit}$$
$$\text{KOMP}_i: \ i\text{-te betroffene Komponente des Programms, } i = 1,...,n$$
$$MAA: \ \text{anzupassender MA-Zugriff.}$$

Ermittelt für das XCTL-System:

$$MAM_2 = ((KOMP_1, Anz.MAA),...,(KOMP_n, Anz.MAA)) =$$

```
((TAdjustmentWindow   ,  3),
 (TAm9513a            ,  39),
 (TAreaScan           ,  4),
 (TBitmapSource       ,  17),
 (TBraun_Psd          ,  39),
 (TC_812              ,  20),
 (TC_812GPIB          ,  6),
 (TC_832              ,  18),
 (TChooseDeviceCmd    ,  1),
 (TCounterShowParam   ,  1),
 (TCounterWindow      ,  1),
 (TDC_Drive           ,  4),
 (TDevice             ,  9),
 (TEditWindow         ,  4),
 (TEncoder            ,  1),
 (TGenericDevice      ,  6),
 (TMain               ,  19),
 (TMDIWindow          ,  6),
 (TModalDlg           ,  1),
 (TModelessDlg        ,  1),
 (TMotor              ,  9),
 (TMotorParam         ,  4),
 (TOptimizeDC_812     ,  7),
 (TOptimizeDC_832     ,  5),
 (TPlotData           ,  6),
 (TPsd                ,  10),
 (TRadicon            ,  10),
```

```
(TScan                 ,   4),
(TScanCmd              ,   1),
(TSetupContinuousScan  ,   2),
(TStoe_Psd             ,   7)
(Tx                    , 298)).
```

Der dritte Schritt, die Ermittlung des Komponenentverhältnisses:

$$MAM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ MA\text{-}Zugriffen}{Anzahl\ der\ Komponenten},$$

ergibt für das XCTL-System:

$$MAM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ MA\text{-}Zugriffen}{Anzahl\ der\ Komponenten} = \frac{32}{81} = 0,395\ .$$

Der vierte Schritt besteht dann wieder aus der Zusammenführung der Maße für die Maschinenarchitektur. Hierbei wird das Maß MA wie folgt definiert:

$$MA = (MAM_1, MAM_2, MAM_3)$$

und sieht für das XCTL-System folgendermaßen aus:

$$MA = (MAM_1, MAM_2, MAM_3) = (563, MAM_2, 0,395)\ .$$

Peripheriegeräte

Schritte zur Ermittlung der Maßkomponente **'**Peripheriegeräte':

- Ermittlung Anzahl unterschiedlicher BS-Aufrufe (PGM_1) Alternativ PGM_1 ';
- Porttool auswerten;
- Berechnung PGM_1;
- Ermittlung der Komponenten mit jew. Gesamtzahl (PGM_2);
- Porttool auswerten;
- Berechnung PGM_2;
- Komponentenverhältnis berechnen (PGM_3);
- Berechnung der Maßkomponente PG .

Zugriffe auf Peripheriegeräte erfolgen innerhalb von Software-Systemen über Ein/Ausgabebefehle, wobei z.B. Steuerzeichen etc. anzupassen sind. Die Maße für Peripheriegeräte sind dem vorangegangenen Maß für die Maschinenarchitektur, vom Aufbau her gleich und die Ausführungen von oben gelten hier im Prinzip ebenso. Die drei Maße für die Peripheriegeräte sehen wie folgt aus:

$$\mathrm{PGM}_1 = Anz.PG_1 + ... + Anz.PG_n \quad \text{mit:}$$

PG_i : i-ter unterschiedlicher PG-Zugriff, $i = 1,...,n$,

$PGM_2 = ((KOMP_1, Anz.PGA),...,(KOMP_n, Anz.PGA))$ mit:

$KOMP_1$: *i*-te betroffene Komponente des Programms, i=1,...,n

PGA : anzupassender PG-Zugriff,

$$PGM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ PG\text{-}Zugriffen}{Anzahl\ der\ Komponenten}.$$

Die Realisierung dieser Maße ergibt sich zu (siehe Anhang F):

$$PGM_1 = Anz.PG_1 + ... + Anz.PG_n = 37\ ,$$

$$PGM_2 = ((KOMP_1, Anz.PGA),...,(KOMP_n, Anz.PGA)) =$$
$$((TAm9513a\ ,\ 8),\ (TBraun\_Psd\ ,\ 1),\ (TC\_812ISA\ ,\ 2),\ (TC\_832\ ,\ 5),\ (TStoe\_Psd\ ,\ 7),\ (Tx\ ,\ 14)),$$

$$PGM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ PG\text{-}Zugriffen}{Anzahl\ der\ Komponenten} = \frac{6}{81} = 0,074\ .$$

Damit läßt sich nun das Gesamtmaß $PG = (PGM_1, PGM_2, PGM_3)$ für das XCTL-System zusammensetzen:

$$PG = (PGM_1, PGM_2, PGM_3) = (37,\ PGM_2,\ 0,074)\ .$$

Aggregation zum Gesamtmaß 'Hardware-Umgebung'

Die Teilmaße der Hardware-Umgebung (Maschinenarchitektur und Peripheriegeräte) stimmen in allen drei Positionen überein und können deshalb leicht zusammengeführt werden. Die Aggregation zum Gesamtmaß erfolgt analog zum Maß 'Software-Umgebung':

$HU = (PAN_{HU}, KOMP_{HU}, VKG_{HU})$ mit:

$PAN_{HU} = MAM_1 + PGM_1$

Anzahl der potentiellen Anpassungen von Zugriffen auf die Hardware-Umgebung

$KOMP_{HU} = MAM_2 + PGM_2$

Namen der Komponenten mit Zugriffen auf die Hardware-Umgebung und deren Anzahl

$VKG_{HU} = MAM_3 + PGM_3$

Verhältnis der Komponentenzahl aus $KOMP_{HU}$ zur Gesamtzahl der Programmkomponenten.

Dabei ist anzumerken, daß der Wert von VKG_{HU} hierbei eine wichtige Rolle spielt. Dieser drückt aus, inwieweit ein Software-System portabilitätsfördernde Prinzipen wie beispielsweise 'Lokalität' einhält, nach dem die Hardwareabhängigkeiten in einigen wenigen Komponenten zusammengefaßt werden sollten. Für das XCTL-System ergibt sich, bezüglich des Gesamtmaßes der Hardware-Umgebung, folgendes Tupel:

$$HU = (PAN_{HU}, KOMP_{HU}, VKG_{HU}) = (600,\ KOMP_{HU},\ 0{,}395),$$

$$\begin{aligned}
\mathrm{KOMP_{HU}} = MAM_2 + PGM_2 = \\
((&\mathit{TAdjustmentWindow} &,\ 3),\\
(&\mathit{TAm9513a} &,\ 47),\\
(&\mathit{TareaScan} &,\ 4),\\
(&\mathit{TBitmapSource} &,\ 17),\\
(&\mathit{TBraun_Psd} &,\ 40),\\
(&\mathit{TC_812} &,\ 20),\\
(&\mathit{TC_812GPIB} &,\ 6),\\
(&\mathit{TC_812ISA} &,\ 2),\\
(&\mathit{TC_832} &,\ 23),\\
(&\mathit{TChooseDeviceCmd} &,\ 1),\\
(&\mathit{TCounterShowParam} &,\ 1),\\
(&\mathit{TCounterWindow} &,\ 1),\\
(&\mathit{TDC_Drive} &,\ 4),\\
(&\mathit{Tdevice} &,\ 9),\\
(&\mathit{TeditWindow} &,\ 4),\\
(&\mathit{Tencoder} &,\ 1),\\
(&\mathit{TGenericDevice} &,\ 6),\\
(&\mathit{Tmain} &,\ 19),\\
(&\mathit{TMDIWindow} &,\ 6),\\
(&\mathit{TmodalDlg} &,\ 1),\\
(&\mathit{TmodelessDlg} &,\ 1),\\
(&\mathit{Tmotor} &,\ 9),\\
(&\mathit{TmotorParam} &,\ 4),\\
(&\mathit{TOptimizeDC_812} &,\ 12),\\
(&\mathit{TplotData} &,\ 6),\\
(&\mathit{TPsd} &,\ 10),\\
(&\mathit{Tradicon} &,\ 10),\\
(&\mathit{Tscan} &,\ 4),\\
(&\mathit{TscanCmd} &,\ 1),\\
(&\mathit{TSetupContinuousScan} &,\ 2),\\
(&\mathit{TStoe_Psd} &,\ 14),\\
(&\mathit{Tx} &\ 312)).
\end{aligned}$$

4.2.1.3 Programmiersprachen

Bei diesem Maß soll bestimmt werden, wie hoch die Änderungsaufwände sind, die durch Unterschiede in der verwendeten Programmiersprache, auf Host- und Zielrechner entstehen. Diese unterscheidet sich entweder in ihrem Dialekt oder ist eine völlig andere, sodaß die Konstrukte, die in der jeweiligen Zielsprache nicht vorhanden sind, an diese angepaßt werden müssen. Um dieses Maß berechnen zu können, müssen jedoch Informationen der Zielsprache in Form geeigneter Beschreibungen vorliegen. Fehlen solche Beschreibungen, so macht es, anders als z.B. bei den BS-Aufrufen, wenig Sinn, Änderungskandidaten zu ermitteln, da diese ja die Menge aller Ausdrücke sein

kann. Aus diesem Grunde existiert kein Maß für die Anzahl potentieller Änderungskandidaten. Nach erfolgter Portierung können wiederum genauere Informationen in das Maß einfließen und darüber informieren, ob bestimmte programmiersprachliche Konstrukte anpaßbar sind und welche ermittelten Aufwände vorliegen (vergleiche Kapitel 3.2.3.4):

$PS_1 = (KA_{PS}, PA_{PS}, SA_{PS}, BA_{PS})$ mit:

$\mathrm{KA_{PS}} = (SK_1, ..., SK_k)$

Konstrukte, die nicht angepaßt werden können

$\mathrm{PA_{PS}} = ((Anz.SK_1, SK_1), ..., (Anz.SK_m SK_m))$

Anzahl und Namen der Konstrukte, über die während der Messung keine Informationen vorliegen

$\mathrm{SA_{PS}} = (\mathrm{Anz.SK_1}, ..., \mathrm{Anz.SK_n})$

Anzahl der Konstrukte, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$\mathrm{BA_{PS}} = \mathrm{Anz.SK_1} \cdot \mathrm{B_1} + ... + \mathrm{Anz.SK_r} \cdot \mathrm{B_r}$

Bewertung der Sprachkonstrukte, für die ein Aufwand vorliegt

B_i: Aufwand für die Anpassung des i-ten Konstruktes

$i = 1, ..., r.$

Überwiegt der Teil von zu ändernden Codekonstrukten, so spricht dies eventuell für eine Neuimplementation. Um diesen Sachverhalt geeignet auszudrücken, existiert bei den Programmiersprachen ein zweites Maß, in welchem die Anzahl der zu ändernden Konstrukte, in Beziehung zur Gesamtzahl aller Ausdrücke gesetzt wird und sich wie folgt ausdrückt:

$$PS_2 = \frac{Anzahl\ aller\ zu\ ändernden\ Statements}{Gesamtzahl\ aller\ Statements}.$$

Bei der Analyse des XCTL-Systems spielt das Programmiersprachenmaß u.U. dann eine Rolle, wenn die geplante ‘Quer-Portierung’ in eine andere Entwicklungsumgebung untersucht wird und somit den Gegenstand weiterer Analysen bildet.

Mehrere Programmiersprachen

Wird bei der Implementierung eines Software-Systems mehr als eine Programmiersprache eingesetzt, so muß das Maß PS_1 für jede eingesetzte Sprache aufgestellt werden. Alle nicht anpaßbaren Konstrukte und solche, über die sonst keine Informationen vorliegen, werden getrennt aufgeführt und können zu einer Entscheidung, für oder gegen eine Portierung führen. Alle unbewerteten Konstrukte und die ermittelten Aufwände können dann zu einem Gesamtmaß addiert werden. Die Addition für PS_2 macht in diesem Fall wenig Sinn, da sich alle durchzuführenden Änderungen u.U. nur auf eine der benutzten Programmiersprachen beschränkt. Für diese Komponenten wäre dann eventuell eine Neuimplementation einer Portierung vorzuziehen, wobei diese Information durch eine Addition der Maße verloren gehen würde. Aus diesem Grund existieren für mehrere Programmiersprachen folgende Maße, wobei die zweiten Maße der Programmiersprachen, in einem Tupel zusammengefaßt werden:

$PS_1' = (KA_{PS\_GES}, PA_{PS\_GES}, SA_{PS\_GES}, BA_{PS\_GES})$ mit:

$\mathrm{KA}_{\mathrm{PS\_GES}} = (KA_{PS1}, ..., KA_{PSn})$

Menge der nicht anpaßbaren Konstrukte der einzelnen Programmiersprachen P_i, $i = 1, ..., n$

$PA_{PS\_GES} = (PA_{PS1}, ..., PA_{PSn})$

Menge der Konstrukte der einzelnen Programmiersprachen zu denen keine Informationen existieren

$SA_{PS\_GES} = SA_{PS1} + ... + SA_{PSn}$

Summe der anpaßbaren Konstrukte der einzelnen Sprachen PS_i, zu denen keine Bewertungen existieren

$BA_{PS\_GES} = BA_{PS1} + ... + BA_{PSn}$

Summer der bewerteten Konstrukte der einzelnen Sprachen PS_i,

$$PS_2' = (PS_{21}, ..., PS_{2n}).$$

Aggregation zum Gesamtmaß 'Programmiersprachen'

Das aggregierte Programmiersprachenmaß für die System-Umgebung, z.B. für den Fall der Analyse einer Quer-Portierung, lautet:

$PS = (PS_1^*, PS_2^*)$ mit:

$PS_i^* = PS_i$ oder PS_i', $i = 1, 2,$ abhängig von der gegebenen Situation.

4.2.1.4 Aggregation zum Gesamtmaß 'System-Umgebung'

Der letzte Schritt bei der Ermittlung des Maßes für die System-Umgebung besteht darin, die bisher aufgezeigten und ermittelten Maße zusammenzuführen. Dabei lassen sich bei den einzelnen Faktoren eine Reihe von Übereinstimmungen feststellen. An diesen Positionen werden sie dann geeignet zusammengefaßt. So existieren z.B. bei dem Software-Umgebungsmaß Betriebssystemroutinen bzw. bei dem Programmiersprachenmaß Sprachkonstrukte, die nicht an die Zielumgebung angepaßt werden können. Darüber hinaus enthalten z.B. Software- und Hardware-Umgebungsmaß das gleiche Verhältnismaß etc. Bei der Aggregation werden die Maße SU' und PS' als jeweils umfassendere Maße zugrundegelegt. Zur Orientierung seien hier noch einmal kurz die entsprechenden Teilmaße aufgeführt:

$SU = (PAN_{SU}, KOMP_{SU}, VKG_{SU})$ bzw.

$SU' = (((KA_{BS}, PA_{BS}), SA_{SU}, BA_{SU}), PAN_{SU}, KOMP_{SU}, VKG_{SU})$

$HU = (PAN_{HU}, KOMP_{HU}, VKG_{HU})$

$PS = ((KA_{PS}, PA_{PS}, SA_{PS}, BA_{PS}), PS_2)$ bzw.

$PS' = (KA_{PS\_GES}, PA_{PS\_GES}, SA_{PS\_GES}, BA_{PS\_GES}), PS_2').$

Das Gesamtmaß 'System-Umgebung' setzt sich wie folgt zusammen:

$SYS = (((KA_{SYS}, PA_{SYS}), SA_{SYS}, BA_{SYS}), PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}, VSG_{SYS})$ mit:

$\mathrm{KA_{SYS}} = (KA_{BS}, KA_{PS\_GES})$

Namen der BS-Routinen, und der PS-Konstrukte, die nicht angepaßt werden können

$\mathrm{PA_{SYS}} = (PA_{BS}, PA_{PS\_GES})$

Anzahl und Namen der BS-Routinen und PS-Konstrukte, über die während der Messung keine Informationen vorliegen

$\mathrm{SA_{SYS}} = SA_{SU} + SA_{PS\_GES}$

Gesamtzahl aus Aufrufen von SU-Routinen und PS-Konstrukten, die anpaßbar sind, für die aber keine Aufwandswerte vorliegen

$\mathrm{BA_{SYS}} = BA_{SU} + BA_{PS\_GES}$

Summe aus bewerteten SU-Routinen und PS-Konstrukten, für die ein Aufwand vorliegt

$\mathrm{PAN_{SYS}} = PAN_{SU} + PAN_{HU}$

Summer der potentiellen Anpassungen aus Software- und Hardware-Umgebung

$\mathrm{KOMP_{SYS}} = KOMP_{SU} + KOMP_{HU}$

Namen der Komponenten mit Anpassungen an die Software- und Hardware-Umgebung und deren Anzahl

$\mathrm{VKG_{SYS}} = VKG_{SU} + VKG_{HU}$

Verhältnis der Komponentenzahl aus $KOMP_{SYS}$ zur Gesamtzahl der Programmkomponenten

$\mathrm{VSG_{SYS}} = PS_2'$

Verhältnis aus der Anzahl der zu ändernden Statements zur Gesamtzahl der Statements.

Damit ergibt sich für das XCTL-System folgendes Maß für die System-Umgebung, wobei die einzelnen Maße automatisch berechnet werden können, da sie auf bereits ermittelte Meßwerte zurückgreifen:

$$SYS = (PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}) = (956, \quad KOMP_{SYS}, \quad 0,703),$$

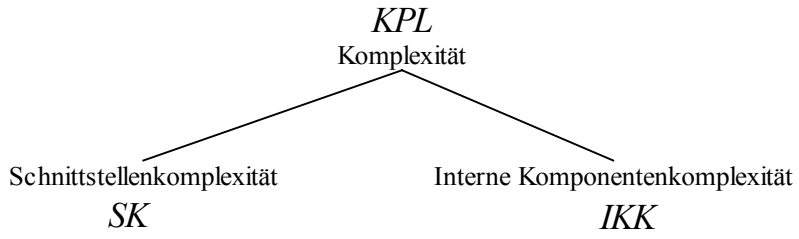
$$\begin{aligned}
&KOMP_{SYS} = KOMP_{SU} + KOMP_{HU} = \\
&((TAbout &&, \ 2),\\
&(TAdjustmentExecute &&, \ 2),\\
&(TAdjustmentWindow &&, \ 4),\\
&(TAm9513a &&, \ 49),\\
&(TAngleControl &&, \ 32),\\
&(TAquisition &&, \ 2),\\
&(TAreaScan &&, \ 18),\\
&(TAreaScanCmd &&, \ 1),\\
&(TAreaScanParameters &&, \ 1),\\
&(TBitmapSource &&, \ 24),\\
&(TBraun_Psd &&, \ 48),\\
&(TC_812 &&, \ 22),\\
&(TC_812GPIB &&, \ 17),
\end{aligned}$$

```
(TC_812ISA              ,   2),
(TC_832                 ,  25),
(TCalibrate             ,   5),
(TCalibratePsd          ,   8),
(TChooseDeviceCmd       ,   1),
(TCmd                   ,   2),
(TCommonDevParam        ,   4),
(TCounterShowParam      ,   2),
(TCounterWindow         ,   3),
(TCurve                 ,   2),
(TDC_Drive              ,   5),
(TDevice                ,  17),
(TDList                 ,   1),
(TEditWindow            ,   7),
(TEncoder               ,   4),
(TExecuteCmd            ,   2),
(TGenericDevice         ,   9),
(TMacroExecute          ,   5),
(TMain                     22),
(TMDIWindow                10),
(TMList                     2),
(TModalDlg                  4),
(TModelessDlg               3),
(TMotor                    10),
(TMotorParam                5),
(TOptimizeDC_812           12),
(TPlotData                  8),
(TPosControl                7),
(TPsd                      11),
(TPsdRemoteSync             4),
(TRadicon                  16),
(TScan                     17),
(TScanCmd                   6),
(TScsParameters             1),
(TSetAdjustmentParam        1),
(TSetupAreaScan             3),
(TSetupContinuousScan       4),
(TSetupScanCmd              1),
(TSetupStepScan             6),
(TShowValueCmd              3),
(TSteering                  9),
(TStoe_Psd                 16),
(TTopographyExecute         8),
(Tx                       441)).
```

4.2.2 Komplexität

Die notwendigen Anpassungen eines Software-Systems bei einer Portierung setzen einige Kenntnisse über dessen Aufbau und die Funktionsweise voraus. Damit Fehler gefunden und korrigiert werden können, ist eine gewisse Einarbeitung in das entsprechende System notwendig. Hierbei spielen die Schnittstellen zwischen den Systemkomponenten eine tragende Rolle, da Änderungen an bestimmten Komponenten nur geringe Auswirkungen auf andere Komponenten haben sollten. Aus diesem Grund wird zunächst das Maß für die Schnittstellenkomplexität von Systemkomponenten vorgestellt und für

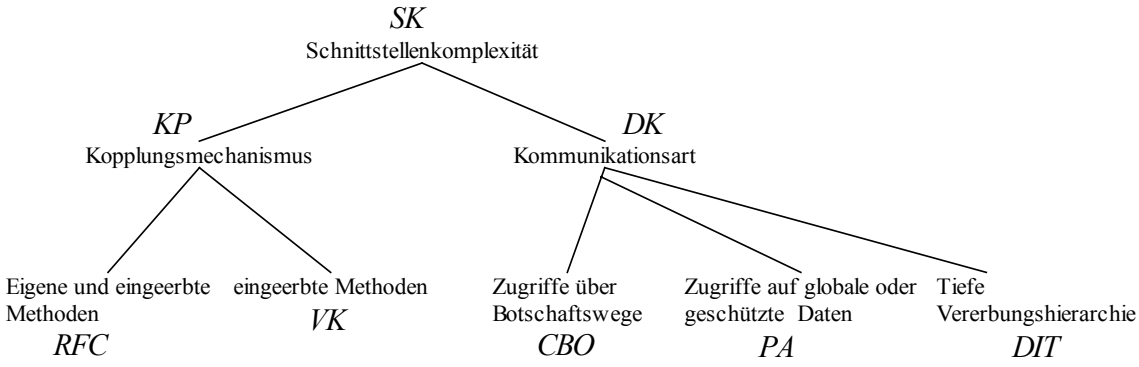
das XCTL-System berechnet. Ein weiterer wichtiger Punkt bei der Bestimmung der Komplexität eines Systems ist die Kenntnis über die logische und strukturelle Komplexität, die sich entscheidend auf den Testumfang auswirkt. Aus diesem Grund wird in einem weiteren Schritt das Maß für die interne Komponentenkomplexität, zunächst noch einmal vorgestellt und ebenfalls für das XCTL-System bestimmt. Beide Komplexitätsmaße werden im Anschluß daran zu einem Gesamtmaß für die Komplexität zusammengefaßt.



Schritte zur Ermittlung des Teilmaßes ’Komplexität’:

- Ermittlung der Maßkomponente ‘Schnittstellenkomplexität’;
- Ermittlung der Maßkomponente ‘Interne Komponentenkomplexität’;
- Zusammenfassung zur Maßkomponente ‘Komplexität’.

4.2.2.1 Schnittstellenkomplexität



Schritte zur Ermittlung des Teilmaßes ’Schnittstellenkomplexität’:

- Ermittlung der Maßkomponente ‘Kopplungsmechanismus’;
- Ermittlung der Maßkomponente ’Kommunikationsart’ bzw. ’Datenkopplung’;
- Zusammenfassung zur Maßkomponente ’Schnittstellenkomplexität’.

Bei der Betrachtung der traditionellen Metriken für prozedural strukturierte Software-Systeme stellten u.a. die Metriken von Myers und Stevens einen Ansatz dar, um über Kopplungsstärken die Schnittstellenkomplexität von Software-Komponenten zu bestimmen. Hierbei waren, mit Ausnahme der Hybrid-Kopplung, die nur für Assembler-Programme von Bedeutung ist, alle Kopplungen in höheren Programmiersprachen realisierbar. Hierbei wurde auch festgestellt, daß nicht jede Programmiersprache auch alle Formen der Kopplung zuläßt. Ein wichtiger Punkt dabei ist die automatische Ermittlung der Kopplungsarten, wobei dies im Fall der reinen Datenkopplung und der

Kopplung über Kontrollparameter nicht immer möglich ist und darüber hinaus auch keine automatische Unterscheidung zwischen Zustands- und Kontrolldaten zu treffen ist. Aus diesem Grund wurden Kontrollparameter in der Variante für Software-Systeme mit prozedural modularem Aufbau in dem vorgestellten Maß, im Kapitel 3.2.3, nicht berücksichtigt. Es wurde bei der Bestimmung der Schnittstellenkomplexität, der Kopplungsmechanismus und die Datenkopplung (Kommunikationsart) einer Komponente unterschieden. Die Kopplungsstärke bzgl. des Mechanismus wurde über die Summation der Anzahl der Kopplungen aller Komponenten ermittelt. Analog hierzu erfolgte die Ermittlung der Kopplungsstärke bzgl. der Datenkopplung durch Aufsummierung der Datenkopplungen einer jeden Komponente.

Kopplungsmechanismus

Schritte zur Ermittlung der Maßkomponente ’Kopplungsmechanismus’:

- Ermittlung RFC ;
- Ermittlung WMC;
- Berechnung VK;
- McCabe-Toolset auswerten;
- Berechnung der Maßkomponente ’Kopplungsmechanismus’ (KP).

Bei einem Portierungsmaß für objektorientierte Software-Systeme kam hinzu, das zusätzliche Konzepte, wie z.B. Klassen, durch Vererbungsgraphen, oder Objekte, durch Assoziationen und Aggregationen, sowie temporäre Botschaftswege, gekoppelt sein können. Für objektorientierte Systeme waren deshalb die vorhanden Maße entsprechend anzupassen. Um bei objektorientierten Software-Systemen die Anzahl von Systemkomponenten zu bestimmen, die über die vorgestellten Kopplungsmechanismen in Verbindung stehen, eignete sich die RFC-Metrik, welche die Anzahl der Funktionen, die durch Operationen einer Klasse aufgerufen werden (intern und extern), mißt. Um den Aspekt der Vererbung zu berücksichtigen, wurde darüber hinaus die Anzahl der in einer Klasse implementierten Methoden, ohne Eingeerbte, in das Maß mit aufgenommen (WMC). Die Differenz aus den beiden zuvor gemessenen Werten ergibt die Zahl der eingeerbten bzw. übernommenen Methoden und hebt damit den Vererbungsaspekt hervor.

Das angepaßte Maß für die Kopplungsstärke, in Bezug auf den Kopplungsmechanismus zwischen den Systemkomponenten in objektorientierten Software-Systemen, lautete (vergleiche Kapitel 3.2.3.6):

$$KP = \sum_{i=1}^{k} KP_i$$

mit: k : Anzahl der Komponenten

KP_i : Kopplung der i-ten Komponente über den Aufruf-Kopplungsmechanismus ,

$$KP_i = m_1 \cdot RFC_i + m_2 \cdot VK_i$$

mit KP_i : Kopplung der i-ten Komponente $i = 1,...,k$

m_1, m_2 : Gewichtungsfaktoren der Maßanteile

RFC_i : Anzahl der aufrufbaren Methoden der i-ten Komponente sowie die Anzahl der internen und externen Aufrufe, die sie selbst durchführt

VK_i : Anzahl der eingeerbten bzw. übernommenen Methoden der i-ten Komponente

wobei $VK_i = RFC_i - NOM_i$

mit NOM_i : Anzahl der in der i-ten Komponente implementierten und nicht eingeerbten Methoden.

Mit Hilfe des McCabe Toolsets werden die Metrikwerte automatisch für das XCTL-System ermittelt [Gollnick 99] (siehe Anhang F):

$$KP = \sum_{i=1}^{k} KP_i = 1622{,}5 .$$

Hierbei ergibt sich das Problem, daß eine Reihe von identifizierten, potentiell anzupassenden Aufrufen, einer gedachten Systemkomponente ’Tx’ zugeordnet werden. Diese Hilfestellung resultiert aus der Tatsache, daß das XCTL-System nicht durchgängig objektorientiert implementiert wurde. Die Teile des Software-Systems, die keiner Klasse, außer Tx, zugeordnet werden konnten, werden deshalb zunächst nicht berücksichtigt. Da das XCTL-System innerhalb eines Reengineering-Projekts bearbeitet wird, ist es jedoch sehr wahrscheinlich, daß in einer weiteren Version der Portabilitätsanalyse, in diesem Teil, genauere Meßdaten zur Verfügung stehen. Diese Aussage gilt für den gesamten Teilbereich der Komplexitätsanalyse.

Kommunikationsart (Datenkopplung)

Schritte zur Ermittlung der Maßkomponente ’Kopplungsmechanismus’:

- Ermittlung CBO;
- Ermittlung PA;
- Ermittlung DIT;
- McCabe-Toolset auswerten;
- Berechnung der Maßkomponente ’Kommunikationsart’ (DK).

Bei der Datenkopplung für objektorientierte Systeme wird die CBO-Metrik zugrunde gelegt und für das XCTL-System bestimmt. Die CBO-Metrik mißt die Anzahl der Klassen, mit denen eine Klasse gekoppelt ist, wenn Methoden der einen Klasse, Methoden oder Instanzvariablen einer anderen Klasse, aufrufen bzw. verwenden. Das Maß berücksichtigt keine Vererbungshierarchie, deshalb soll zusätzlich die DIT-Metrik in das Maß mit einfließen. Sie gibt Aufschluß darüber, wie tief die Vererbungshierarchie der Klassen geht und zählt konkret die Anzahl der Vorfahren einer Klasse im Vererbungsbaum. Der Datenaustausch zwischen den Komponenten über globale Daten wird durch das Maß PA ausgedrückt:

$$DK = \sum_{i=1}^{k} DK_i$$

mit: k : Anzahl der Komponenten

DK_i : Datenkopplung der i-ten Komponente,

$$DK_i = d_1 \cdot CBO_i + d_2 \cdot PA_i + d_3 \cdot DIT_i$$

mit DK_i : Datenkopplung der i-ten Komponente $i = 1,...,k$

d_1, d_2, d_3 : Gewichtungsfaktoren der Datenkopplungsart

CBO_i : Anzahl der Datenkopplungen über Botschaftswege der i-ten Komponente

PA_i : Anzahl der Zugriffe auf public oder protected Datenelemente durch andere Klassen auf die i-te Komponente

DIT_i : Länge des maximalen Weges in der Klassenhierarchie von der i-ten Komponente bis zur Wurzel.

Für das XCTL-System ergeben sich hierbei folgende Meßwerte (ermittelt mit McCabe Toolset):

$$DK = \sum_{i=1}^{k} DK_i = 11{,}95 .$$

Hierbei ist anzumerken, daß bei der Vermessung des XCTL-Systems keine ausreichenden Meßwerte für den Zugriff auf globale Daten ermittelt werden konnten (Vergleich Anhang F). Der ermittelte Meßwert beträgt für jede Systemkomponente 0, d.h. daß man theoretisch alle momentan als global deklarierten Klassenkomponenten, auch als private deklarieren könnte, da auf keines von außen zugegriffen wird. Damit geht diese Art der Datenkopplung bei der Portabilitätsanalyse des XCTL-Systems zunächst nicht mit ein.

Aggregation zum Gesamtmaß 'Schnittstellenkomplexität'

Das Gesamtmaß besteht aus den Maßzahlen für die Bewertung der Kopplungsmechanismen und der Datenkopplungen. Es wird wie folgt durch einen 2er-Tupel gebildet:

$$SK = (KP,\ DK).$$

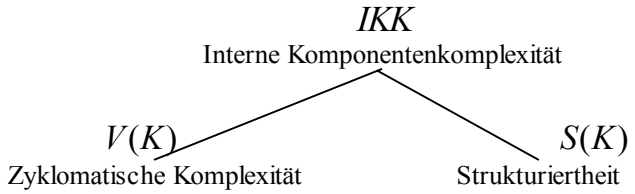
Für das XCTL-System konkretisiert heißt das:

$$SK = (KP, DK) = (1622{,}5\ \ ,\ \ 11{,}95).$$

4.2.2.2 Interne Komponentenkomplexität

Wie bereits diskutiert, unterscheidet man eine Vielzahl von Bindungsarten zwischen den Elementen einer Komponente. Da jedoch die automatische Bestimmung der Bindungsart, aufgrund fehlender, objektiv meßbarer Kriterien, nicht möglich ist und sie nur mittels eines 'Code-Review' bestimmbar ist, kann dieser Ansatz nicht in das Portierungsmaß eingehen. Es ist jedoch möglich die interne Komplexität einer Komponente durch die Anzahl Tests zu bestimmen, die notwendig sind, um deren Korrektheit zu überprüfen. Hierfür hat McCabe zur Ermittlung der Testbarkeit einer Komponente oder

Programms, die zyklomatische Zahl vorgeschlagen. Der Vorteil hierbei ist die einfache Berechnung dieser Maßzahl für die logische Komplexität. Nachteil ist, wie bereits oben erwähnt worden ist, daß nur das Programmgerüst und nicht die Komplexität einzelner und verschachtelter Anweisungen berücksichtigt wird. Um auch die Strukturiertheit eines Software-Systems in das Portierungsmaß mit eingehen zu lassen, kann u.a. die Schachtelungstiefe berechnet werden und als Gradmesser für die Strukturiertheit eines Systems gelten. Neben der Schachtelungstiefe kann auch die essentielle zyklomatische Zahl als Maß für die Strukturiertheit eines Software-Systems herangezogen werden.



Schritte zur Ermittlung des Teilmaßes 'Interne Komponentenkomplexität'

- Ermittlung der Maßkomponente 'Zyklomatische Komplexität';
- Ermittlung der Maßkomponente 'Strukturiertheit';
- Zusammenfassung zur Maßkomponente 'Interne Komponentenkomplexität'.

Zyklomatische Komplexität

Schritte zur Ermittlung der Maßkomponente 'Kopplungsmechanismus':

- Ermittlung $VWMC$;
- McCabe-Toolset auswerten;
- Berechnung der zyklomatischen Komplexität $V(K)$.

Die Diskussion zur Anwendung der zyklomatischen Zahl auf objektorientierte Systeme, ergab die Reduzierung auf Methodenebene, da nur dort ein Kontrollflußgraph im McCabeschen Sinne konstruierbar ist. Andere Autoren schlagen vor, die zyklomatische Komplexität auch auf Klassenhierarchien anzuwenden (AC). Die Verwendung der zyklomatischen Zahl auf Klassen- oder Systemniveau ist durch Akkumulation möglich. Chidamber et. al. haben das bei der Vorstellung der WMC-Metrik vorgeschlagen, wobei dann die Gewichtung der Methodenkomplexität durch die, auf Methodenebene ermittelte, zyklomatische Zahl vorgenommen wird.

Ausgehend von den bisherigen Überlegungen, soll die zyklomatische Zahl des zu portierenden Software-Systems, über die entsprechend gewichtete WMC-Metrik, in das Portierungsmaß für objektorientierte Systeme eingehen. Es hatte folgenden Aufbau:

$$V(K) = \sum_{i=1}^{k} VWMC_i$$

mit $VWMC_i$: Komplexität der i-ten Komponente

k : Anzahl der Komponenten,

$$VWMC_i := \sum_{j=1}^{n} g(M_j) \text{ und } g(M_j) = v(M_j)$$

mit $V(M_j)$: Komplexität der j-ten Methode einer i-ten Komponente

n : Anzahl der Methoden.

Für das XCTL-System ergeben sich hierfür folgender Werte(ermittelt mit dem McCabe Toolset):

$$V(K) = \sum_{i=1}^{k} VWMC_i = 3285.$$

Strukturiertheit

Schritte zur Ermittlung der Maßkomponente 'Strukturiertheit':

- Ermittlung $EWMC$;
- McCabe-Toolset auswerten;
- Berechnung der Strukturiertheit des Software-Systems $S(K)$.

Für das Maß der Strukturiertheit eines Software-Systems wird die essentielle zyklomatische Zahl als Grundlage genommen. Das Maß hatte folgenden Aufbau:

$$S(K) = \sum_{i=1}^{k} EWMC_i$$

mit $EWMC_i$: Maximale, essentielle Komplexität der i-ten Komponente über alle Methoden

k : Anzahl der Komponenten,

$$EWMC_i := Max_i(ev(M_{i1}),...,ev(M_{ij}))$$

mit ev(M_{ij}): essentielle Komplexität der j-ten Methode in der i-ten Komponente.

Das XCTL-System weist für das Maß der Strukturiertheit $S(K)$ folgenden Wert auf:

$$S(K) = \sum_{i=1}^{k} EWMC_i = 398.$$

Aggregation zum Gesamtmaß 'Interne Komponentenkomplexität'

Das Maß für die interne Komponentenkomplexität setzt sich aus zwei Maßzahlen zusammen:

$$IKK = (V(K), S(K))$$

Eine Addition der beiden Komplexitätswerte wird auf dieser Ebene nicht durchgeführt, da beide Maße unterschiedliche Komplexitätsarten messen. So mißt die zyklomatische Zahl die logische- und die essentielle Komplexität die strukturelle Komplexität des Software-Systems. Für das XCTL-System ergibt sich hierfür folgendes:

$$IKK = (V(K), S(K)) = (3285\ ,\ \ 398).$$

4.2.2.3 Aggregation zum Gesamtmaß 'Komplexität'

Ausgehend von den bisher ermittelten Komplexitätsmaßen wird nun das Gesamtmaß zusammengesetzt. Diese werden jedoch nicht einfach zusammengefaßt, sondern noch weiter verrechnet, um detailliertere Aussagen über die einzelnen Einflußfaktoren machen zu können. Das Gesamtmaß für die Komplexität hat folgenden Aufbau:

$$KPL = (KPK, DKK, SK_{aggr}, DLK, IKK_{aggr})$$

mit: KPK : Kopplungskomplexität
DKK : Datenkopplungskomplexität
SK_{aggr} : aggregierte Schnittstellenkomplexität
DLK : durchschnittliche logische Komplexität
IKK_{aggr} : aggregierte interne Komplexität .

Um Portierungserfahrungen in das Maß mit einfließen zu lassen, werden die einzelnen Maße mit Gewichtungsfaktoren versehen und anschließend aufsummiert. Somit ist es möglich, den Einfluß der Kopplungswerte nach erfolgter Portierung erfahrungsmäßig zu gewichten. Um in das Maß für die Schnittstellkomplexität noch Portierungserfahrungen einfließen zu lassen, werden wie folgt Gewichtungsfaktoren (zunächst rein hypothetisch) eingefügt und zu einer Maßzahl SK_{aggr} für das XCTL-System aufsummiert:

$$SK_{aggr} = s_1 \cdot KP, s_2 \cdot DK = 0,5 \cdot 1622,5 + 0,5 \cdot 11,95 = 817,23\ .$$

Analog hierzu, für das Maß der internen Komponentenkomplexität (wiederum mit rein hypothetischen Gewichten, da noch keine Portierungserfahrungen vorliegen):

$$IKK_{aggr} = k_1 \cdot V(K) + k_2 \cdot S(K) = 0,5 \cdot 3285 + 0,5 \cdot 398 = 1841,5.$$

Das Maß KPK wird durch das Verhältnis der Anzahl der Kopplungen des Software-Systems und der gewichteten Kopplungsmechanismen KP aller Komponenten gebildet. Das Verhältnis soll den Grad der Kopplungen unter den Komponenten verdeutlichen. Je höher der Wert gegen 1 geht, desto schwieriger wird die Einarbeitung und um so umfangreicher ist der Änderungsaufwand.

$$KPK = \frac{KP}{Anzahl\ der\ Kopplungen\ aller\ Komponenten} = 0,506.$$

Das gleiche Verhältnis wird für die Datenkopplungen des Software-Systems gebildet Je höher der Datenkopplungsgrad, um so schwieriger die Einarbeitung und desto höher die Zahl der durchzuführenden Änderungen:

$$DKK = \frac{DK}{Anzahl\ aller\ Datenkopplungen} = 0,027.$$

Auch hier bleibt anzumerken, daß bei der Vermessung des XCTL-Systems keine ausreichenden Meßwerte für den Zugriff auf globale Daten ermittelt werden konnten. Dieses Maß ist vom McCabe-Toolset offensichtlich unkorrekt bestimmt worden. Deswegen an diese Stelle noch einmal der Hinweis, daß die Datenkopplung über globale Daten, in diese Version der Portabilitätsanalyse, nicht mit eingeht.

Neben den bereits berücksichtigten Maßen, soll auch die durchschnittliche zyklomatische Komplexität noch mit einbezogen werden, da diese im Allgemeinen einen guten Indikator für die Gesamtkomplexität des Software-Systems darstellt. Überschreitet der Meßwert eine vorgeschlagene Grenze von 10 [McCabe 76], sollte das Software-System, entweder weiter zerlegt oder neuimplementiert werden. Dieses Maß wird für das XCTL-System wie folgt automatisch durch das McCabe-Toolset ermittelt:

$$DLK = \overline{V(K)} = 2,87\ .$$

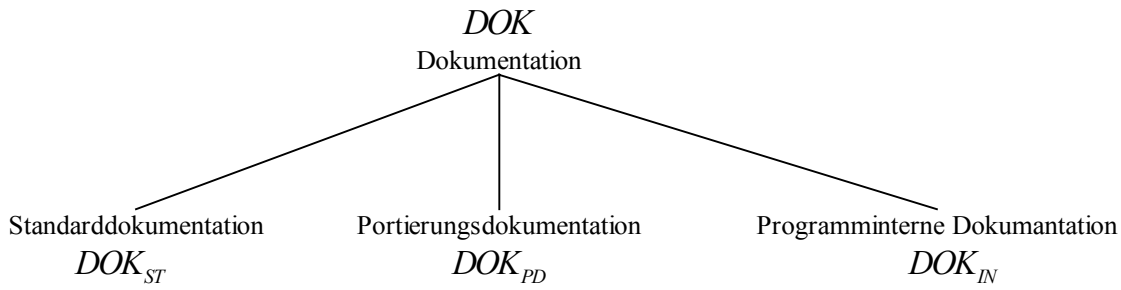
Das zusammengesetzte Gesamtmaß für die Komplexität des XCTL-Systems lautet damit:

$$KPL = (KPK, DKK, SK_{aggr}, DLK, IKK_{aggr}) =$$
$$(0,506\ \ ,\ \ 0,027\ \ ,\ \ 817,23\ \ ,\ \ 2,87\ \ ,\ \ 1841,5\ ).$$

4.2.3 Dokumentation

Die Dokumentation eines Software-Systems hat großen Einfluß auf das Verständnis und die schnelle Einarbeitung in ein bestehendes System. Durch geeignete Dokumentation kann der Portierungsaufwand maßgeblich beeinflußt werden. Aufgrund der Relevanz von Dokumentationsinformationen werden diese im Portierungsmaß mitgeführt. Hierbei wird das Vorhandensein wichtiger Dokumentationsteile manuell überprüft und dem Portierungsmaß, als subjektive Bewertung, zugeführt. Somit geht die Dokumentation als subjektive Maßkomponente in das Gesamtmaß mit ein, wodurch die Objektivität des Maßes nicht vollständig gegeben ist. Da die Dokumentation den Portierungsaufwand aber wesentlich beeinflußt, wird der Nachteil der reduzierten Objektivität des Portierungsmaßes hingenommen.

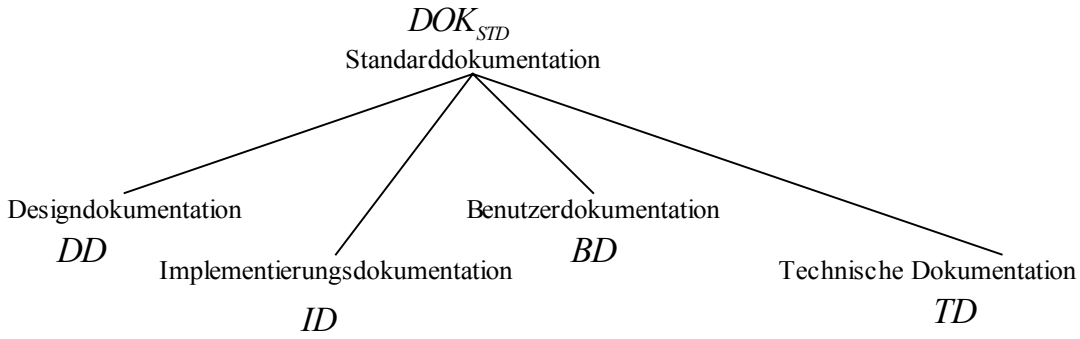
Bei dem Dokumentationsmaß werden die relevanten Dokumente mit brauchbaren Informationen auf ihr Vorhandensein hin überprüft und folglich binär bewertet (vorhanden/nicht vorhanden). Die wesentliche Teile werden wie folgt klassifiziert:



Schritte zur Ermittlung des Teilmaßes 'Dokumentation':

- Ermittlung der Maßkomponente 'Standarddokumentation';
- Ermittlung der Maßkomponente 'Portierungsdokumentation';
- Ermittlung der Maßkomponente 'Programminterne Dokumentation';
- Zusammenfassung zur Maßkomponente 'Dokumentation'.

4.2.3.1 Standarddokumentation



Schritte zur Ermittlung des Teilmaßes ' Standarddokumentation':

- Ermittlung der Maßkomponente 'Designdokumentation';
- Ermittlung der Maßkomponente 'Benutzerdokumentation';
- Ermittlung der Maßkomponente' Technische Dokumentation';
- Ermittlung der Maßkomponente' Implementierungsdokumentation';
- Auswertung der manuellen Inspektion;
- Zusammenfassung zur Maßkomponente ' Standarddokumentation'.

Mit der Standarddokumentation ist die Dokumentation gemeint, die bei der Entwicklung und dem Betrieb von Software-Systemen entsteht. Dazu gehören sowohl Entwickler-, als auch Nutzerdokumente. Ohne diese Dokumente ist es für einen Entwickler sehr schwierig ein Software-System zu warten oder zu erweitern, bzw. für einen Benutzer das System vernünftig einzusetzen. Ein Porteur muß sich während einer Portierung u.a. die Aufrufstruktur und bestimmte Implementierungsentscheidungen klar machen, wobei ein Fehlen der entsprechenden Dokumentation, einen Portierungsversuch deutlich erschwert.

Für jedes der oben angeführten Dokumente wird ein binäres Maß vorgeschlagen. Das Maß für die Designdokumentation lautet wie folgt:

$DD =$ Designdokumentation vorhanden/nicht vorhanden.

Bei vorhandener Dokumentation erhält das Maß den Wert 1, ansonsten den Wert 0. Die Bewertung erfolgt durch eine rein subjektive Bewertung, z.B. Befragung des Porteurs. Dies kann in Form einer Checkliste durchgeführt werden und ist von den Erfahrungen des Porteurs abhängig. Zu den wichtigen Designdokumenten zählen z.B. das Pflichtenheft, ein Produkt-Modell oder das Konzept für die Benutzeroberfläche.

Analog hierzu wird für die anderen Teile der Standartdokumentation ein Maß angegeben:

$BD =$ Benutzerdokumentation vorhanden/nicht vorhanden,

$TD =$ Technische Dokumentation vorhanden/nicht vorhanden,

$ID =$ Implementierungsdokumentation vorhanden/nicht vorhanden.

Zur Benutzerdokumentation gehört in erster ein adäquates Benutzerhandbuch oder ein entsprechender Benutzerleitfaden. Die technische Dokumentation beinhaltet u.a. Angaben zur Software-Architektur und zur Spezifikation der Systemkomponenten. Bei der Implementierungsdokumentation wird in erster Linie die Verifikationsdokumentation und die externe Dokumentation der Quellen bewertet.

Die vier Maße werden zu einem Maß aggregiert, indem eine gewichtete Summe gebildet wird. Diese sollen Erfahrungen der Relevanz bestimmter Dokumentationsteile, bezüglich einer Portierung auszudrücken. Die Faktoren werden entsprechend normiert, um die Aggregation zum Gesamtmaß ’Dokumentation’ durchführen zu können. Die Normierung erfolgt durch Summenbildung zum Wert 1. Das Maß für die Standarddokumentation hat folgenden Aufbau:

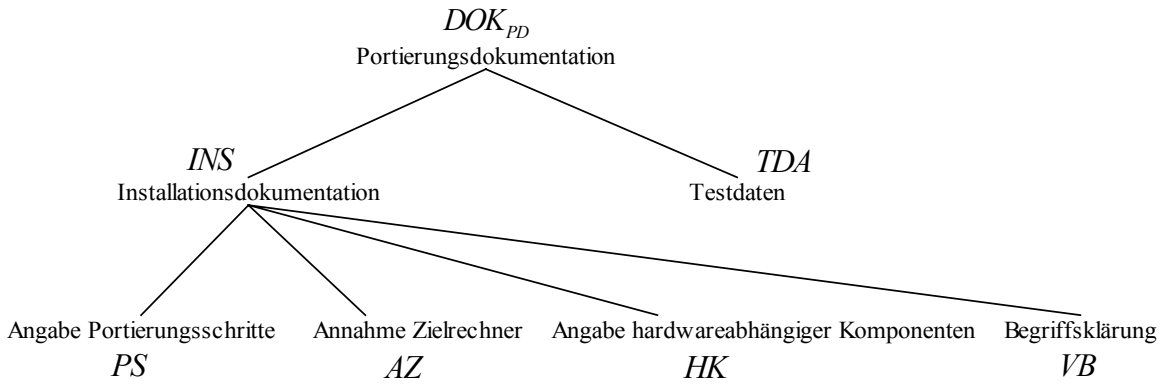
$$DOK_{STD} = g_1 \cdot DD + g_2 \cdot BD + g_3 \cdot TD + g_4 \cdot ID \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1.$$

Liegen noch keine konkreten Portierungserfahrungen vor, werden die Gewichtungsfaktoren mit 0,25 belegt, was in einem ersten Ansatz bedeutet, daß alle Teile gleich wichtig sind. Diese Annahme ist ggf. durch Portierungserfahrungen zu bestätigen.

Für das XCTL-System lag nur sehr wenig Dokumentation vor. Hierzu gehörten die Schnittstellenbeschreibungen einiger Peripheriegeräte und eine Dokumentation von Teilen der Initialisierungsdateien des Systems. Im Laufe des Reengineering-Projekts entstanden jedoch Pflichtenhefte zu bestimmten Aufgabenbereichen sowie ein entsprechender Benutzerleitfaden. Auch wurden die Oberfläche und die Dialogelemente untersucht und ein Dokument für die Beschreibung der Oberfläche entwickelt. Die Quelldateien wurden untersucht und jeweils mit Funktionalität und Headerstruktur beschrieben. Daraus entstand eine erste Übersicht zu den Quelldateien und deren Abhängigkeiten. Für die Bewertung der Standarddokumentation ergibt sich somit folgende, rein subjektive, Einschätzung:

$$DOK_{STD} = 0,25 \cdot 1 + 0,25 \cdot 1 + 0,25 \cdot 0 + 0,25 \cdot 0 = 0,5.$$

4.2.3.2 Portierungsdokumentation



Schritte zur Ermittlung des Teilmaßes ' Portierungsdokumentation':

- Ermittlung der Maßkomponente 'Installationsdokumentation';
- Ermittlung der Maßkomponente 'Testdaten';
- Auswertung der manuellen Inspektion;
- Zusammenfassung zur Maßkomponente 'Portierungsdokumentation'.

Die Teile der Dokumentation, die Erläuterungen und Anweisungen zur z.B. physischen Übertragung eines Software-Systems, Anpassung an die neue System-Umgebung oder den Nachweis der Korrektheit der Portierung geben, werden in diesem Maß zusammengefaßt. Hierbei wird das Maß in die Submaße für die Installationsdokumentation und der Testdaten unterteilt.

Installationsdokumentation

Schritte zur Ermittlung der Maßkomponente 'Installationsdokumentation':

- Ermittlung, ob Angaben zu den Portierungsschritten vorhanden sind (PS);
- Ermittlung, ob Annahmen über den Zielrechner vorhanden sind (AZ);
- Ermittlung, ob hardware-abhängige Komponenten dokumentiert werden (HK);
- Ermittlung, ob die verwendeten Begriffe erklärt werden (VB);
- Auswertung der manuellen Inspektion;
- Zusammenfassung zur Maßkomponente ' Installationsdokumentation'.

Eine gute Installationsdokumentation sollte auf jeden Fall alle der aufgeführten Elemente aufweisen. Analog zum Maß für die Standarddokumentation, setzt sich das Maß für die Installationsdokumentation wie folgt zusammen:

$$INS = g_1 \cdot PS + g_2 \cdot AZ + g_3 \cdot HK + g_4 \cdot VB \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1.$$

Hierbei wird die Hypothese aufgestellt, daß Angaben zu den Portierungsschritten sowie die Annahmen über den Zielrechner stärkeren Einfluß auf den Portierungsaufwand haben, als die restlichen Elemente. Aus diesem Grund kann hier bereits eine erste hypothetische Annahme wie folgt getroffen werden:

$$INS = 0,4 \cdot PS + 0,3 \cdot AZ + 0,2 \cdot HK + 0,1 \cdot VB.$$

Bei dem XCTL-System liegen, in diesem Sinne, bisher keine Portierungsdokumente vor. Es liegen aber Annahmen zu den Zielrechnern vor und es ist auch schon ausreichend bekannt, welche Teile des Systems hardwareabhängig sind. Die verwendeten Begriffe sind bisher sehr eingeschränkt dokumentiert. Für das XCTL-System ergibt sich für die Installationsdokumentation somit folgendes:

$$INS = 0,4 \cdot 0 + 0,3 \cdot 1 + 0,2 \cdot 1 + 0,1 \cdot 0 = 0,5.$$

Testdaten

Nach erfolgter Portierung muß die Korrektheit eines Software-Systems überprüft werden und festgestellt werden, daß es der ursprünglichen Spezifikation gemäß funktioniert. Dieser Nachweis kann über Testprotokolle und geeigneten Testdaten erfolgen. Da der Korrektheitsnachweis von Software-Systemen notwendig ist und vorhandene Testdaten den Portierungsprozeß beeinflussen, wird dieses Maß in das Gesamtmaß mit aufgenommen:

$$TDA = \text{ Testdaten vorhanden/nicht vorhanden.}$$

Sind die Angaben zu den Testdaten vorhanden (z.B. Grad der Pfadüberdeckung), so ist die Beurteilung relativ einfach. Falls die Testdaten fehlen, geht wieder eine stark subjektive Meinung des Porteurs in das Maß ein.

Im Verlauf der Restrukturierung des XCTL-Systems wurden eine Reihe von Testdaten ermittelt und ein entsprechendes Testprotokoll angelegt. Daher soll dieser Teil mit dem Wert 1 belegt werden. Somit gilt für das XCTL-System:

$$TDA = 1.$$

Aggregation zum Gesamtmaß 'Portierungsdokumentation'

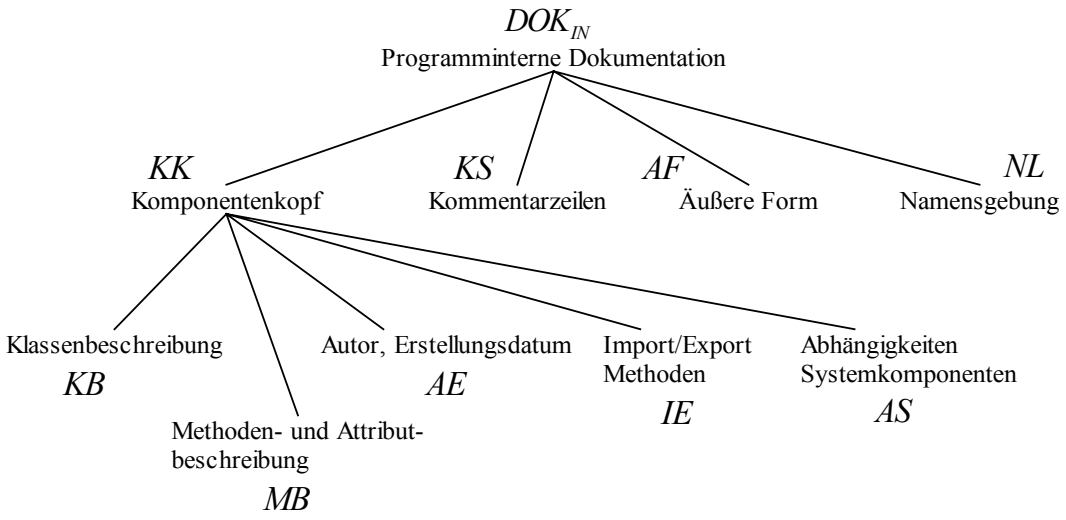
Das Maß wird aus den Submaßen der Installationsdokumentation und den Testdaten zusammengesetzt. Auch hier wird, wie bei den Maßen zuvor, eine entsprechende Gewichtung durch Faktoren vorgenommen. Die Maße mit den Gewichtungsfaktoren werden analog zu den anderen Maßen summiert und mit der Normierungsbedingung versehen. Das Maß für die Portierungsdokumentation hat folgenden Aufbau:

$$DOK_{PD} = g_1 \cdot INS + g_2 \cdot TDA \quad \text{wobei gilt: } g_1 + g_2 = 1.$$

Für das XCTL-System ergibt sich, mit hypothetischer Gewichtung der Einzelmaße:

$$DOK_{PD} = 0,6 \cdot 0,5 + 0,4 \cdot 1 = 0,7.$$

4.2.3.3 Programminterne Dokumentation



Neben der externen Dokumentation der Quellen, wie z.B. Verzeichnisse der Quelldateien des Software-Systems etc., wird bei der programminternen Dokumentation der, über die Codezeilen hinausgehende, dokumentierende Informationsgehalt in den Quelltexten bewertet. Eine gute programminterne Dokumentation ermöglicht dem Porteur eine schnelle Einarbeitung in die Quellen und deren einfache Bearbeitung. Außerdem stellt sie dem Porteur alle wichtigen Informationen, der zu bearbeitenden Softwarekomponente, u.a. nach den Prinzipien der integrierten Dokumentation, direkt in den Quellen zur Verfügung.

Zu den wichtigsten programminternen Informationen zählen:

- Komponentenkopf,
- Kommentarzeilen,
- Äußere Form,
- Namensgebung,

Diese werden nacheinander manuell ermittelt und zur Maßkomponente 'Programminterne Dokumentation' zusammengeführt.

Komponentenkopf

Der Kopf einer Systemkomponente sollte deren Beschreibung beinhalten. Also die Beschreibung der Aufgabe der Komponente, um nicht erst den Quelltext analysieren oder umständlich in der externen Systemdokumentation nachschlagen zu müssen, um den Zweck einer Systemkomponente zu bestimmen. Darüber hinaus ist der Autor der Softwarekomponenten und deren Erstellungsdatum bzw. Versionsnummer eine wichtige Information, um evtl. Rücksprachen mit dem Entwickler halten zu können und um Informationen über den aktuellen Stand der jeweiligen Komponenten zu erhalten. Zu einer guten Komponenteninformation gehört auch die Angabe von importierten und exportierten Datenstrukturen sowie die Auflistung der aufgerufenen Systemkomponenten bzw. der Verbindungen zu anderen Systemkomponenten.

Die Einzelmaße, die diese Informationen aufnehmen, haben folgenden Aufbau und werden schrittweise manuell ermittelt:

KB = Beschreibung der Systemkomponenten (Klassen) vorhanden/nicht vorhanden;
SB = Beschreibung der Klassenkomponenten (Methoden, Attribute) vorhanden/nicht vorhanden;
AE = Autor und Erstellungsdatum vorhanden/nicht vorhanden;
IE = Auflistung Import/Export der Methoden vorhanden/nicht vorhanden;
AS = Auflistung der abhängigen Systemkomponenten vorhanden/nicht vorhanden.

Da die Handhabung von Komponentenköpfen nicht standardisiert ist und somit eine automatische Ermittlung des Maßes nicht möglich ist, werden Stichproben durch den Porteur ermittelt und subjektiv bewertet. Hierbei wird die Maßzahl durch die absolute Mehrheit festgelegt, d.h. wenn mehr als der Hälfte aller Systemkomponenten ohne Komponentenkopf mit entsprechenden Einzelinformationen sind, so wird dem jeweiligen Maß der Binärwert 1 zugeteilt, ansonsten der Wert 0.

Das aggregierte Maß für den Komponentenkopf mit den jeweiligen Gewichtungsfaktoren sieht wie folgt aus:

$$KK = g_1 \cdot KB + g_2 \cdot MB + g_3 \cdot AE + g_4 \cdot IE + g_5 \cdot AS \quad \text{wobei gilt: } \sum_{i=1}^{5} g_i = 1.$$

Eine Code-Inspektion im XCTL-System ergab, daß lediglich der Autorenname und das Erstellungsdatum einer Quelldatei, teilweise aufgeführt waren. Alle anderen Teile waren nicht oder nur auszugsweise vorhanden. Deshalb folgende Bewertung (wiederum mit angenommener Gewichtung, da noch keine Portierungserfahrungen vorliegen):

$$KK = 0,2 \cdot 0 + 0,2 \cdot 0 + 0,2 \cdot 1 + 0,2 \cdot 0 + 0,2 \cdot 0 = 0,2.$$

Kommentarzeilen

Hierbei könnte man das Verhältnis von Kommentarzeilen zur Gesamtzeilenzahl des Software-Systems zugrunde legen. Dadurch würde aber die Fehlannahme impliziert, je mehr Kommentarzeilen ein Software-System in seinen Quellen aufweist, desto besser. Was natürlich so nicht der Fall sein kann. Um das Maß der Kommentarzeilen zu bestimmen, wird davon ausgegangen, daß es sinnvoller ist zu überprüfen, ob mindestens die Benutzung von Kontrollstatements ausreichend dokumentiert ist. Dabei wird durch eine subjektive Einschätzung, der Einhaltung des Prinzips der Verbalisierung – Gedanken und Vorstellungen in Worten ausdrücken und damit ins Bewußtsein bringen; Hier die Ideen und Konzepte des Entwicklers –, Rechnung getragen. Das folgende Maß erhält den Wert 1, falls mehr als die Hälfte aller Kontrollstatements nicht ausreichend kommentiert sind. Es wird anhand von Stichproben ermittelt:

KS = Benutzung von Kontrollstatements dokumentiert / nicht dokumentiert.

Die Codeinspektion ergab, daß nur wenig Kontrollstatements dokumentiert sind und auch die restrukturierten Teile, den Gesamtdurchschnitt nicht ausreichend beeinflussen. Daher hier der Wert:

$$KS = 0\ .$$

Äußere Form

Die Übersichtlichkeit eines Quellcodes wird durch seine äußere Form bestimmt. Entscheidend hierbei ist z.B. die Einhaltung von Einrückungsprinzipien, welche man seiner Quellcodestruktur zugrunde legt. Die Lesbarkeit bzw. Strukturiertheit des Quellcodes wird anhand eines kurzen Codeinspektion durchgeführt. Die Bewertung des Maßes erfolgt mit 1, falls der Code strukturiert ist.

$$AF = \text{ Quellcode strukturiert / nicht strukturiert.}$$

Die Überprüfung der Quellen des XCTL-Systems ergab zwar, daß diese strukturiert sind, jedoch läßt diese Strukturierung noch Raum für viele Verbesserungen, auch in Hinblick auf die Lesbarkeit. Dieser Teil wird wie folgt bewertet:

$$AF = 1.$$

Namensgebung

Für eine problemadäquate Charakterisierung von Konstanten, Variablen, Prozeduren usw. sind Bezeichner zu wählen, die deren Funktion bzw. ihre Aufgabe zum Ausdruck bringen. Hierbei sind kurze Bezeichner wenig aussagekräftig und wegen der geringen Redundanz anfälliger gegen Tippfehler und Verwechslungen. Zu lange Bezeichner haben den Nachteil, daß sie schwer zu merken sind. Eine Untersuchung ergab eine optimale Bezeichnerlänge zwischen 5 und 9 Zeichen. Dieses Kriterium wird dem Maß für die Bezeichnerlänge zugrunde gelegt, wobei die durchschnittliche Bezeichnerlänge automatisch ermittelt werden kann.

$$BL = \text{ durchschnittliche Bezeichnerlänge 5-9 Zeichen / nicht 5-9 Zeichen.}$$

Eine manuelle Überprüfung des XCTL-Systems ergab, daß die durchschnittliche Bezeichnerlänge den Grenzwert für das Optimum übersteigt und somit folgendermaßen bewertet wird:

$$BL = 0 .$$

Aggregation zum Gesamtmaß 'Programminterne Dokumentation'

Die vier Maße für die programminterne Dokumentation sind unabhängig voneinander und werden deshalb zu einem Gesamtmaß gewichtet und normiert aufsummiert:

$$DOK_{IN} = g_1 \cdot KK + g_2 \cdot KS + g_3 \cdot AF + g_4 \cdot BL \quad \text{wobei gilt: } \sum_{i=1}^{4} g_i = 1.$$

Für das XCTL-System ergibt sich, mit hypothetischen Annahmen zur Gewichtung der einzelnen Teilmaße, folgendes Gesamtmaß für die programminterne Dokumentation:

$$DOK_{IN} = 0,3 \cdot 0,2 + 0,3 \cdot 0 + 0,3 \cdot 1 + 0,1 \cdot 0 = 0,36.$$

4.2.3.4 Aggregation zum Gesamtmaß 'Dokumentation'

Die Teilmaße werden zu einem Gesamtmaß zusammengesetzt, in dem man auch hier wieder eine Gewichtung vornimmt. Das macht Sinn, da man annehmen kann, daß die Portierungsdokumentation den Aufwand am stärksten beeinflußt, während der Einfluß der internen Dokumentation geringer ist. Man trifft hierbei eine erste Annahme, die wie folgt lauten könnte und durch konkrete Portierungserfahrungen zu bestätigen ist:

$$DOK = g_1 \cdot DOK_{ST} + g_2 \cdot DOK_{PD} + g_3 \cdot DOK_{IN} \quad \text{wobei gilt: } \sum_{i=1}^{3} g_i = 1.$$

Annahme:

$$DOK = 0{,}15 \cdot DOK_{ST} + 0{,}6 \cdot DOK_{PD} + 0{,}25 \cdot DOK_{IN} \quad \text{wobei gilt: } \sum_{i=1}^{3} g_i = 1.$$

Bei dem XCTL-System ergibt sich für das Dokumentationsmaß folgender Wert:

$$DOK = 0{,}15 \cdot 0{,}5 + 0{,}6 \cdot 0{,}7 + 0{,}25 \cdot 0{,}36 = 0{,}59.$$

4.3 Zur Portabilität des XCTL-Systems

Alle Maßkomponenten werden wie folgt zum Gesamtmaß zusammengefaßt, wobei ein Tupel gebildet wird, da die Maße sehr unterschiedliche Systemaspekte messen:

$$PORT = (SYS,\ KPL,\ DOK) \quad \text{mit:}$$
$$SYS = (((KA_{SYS}, PA_{SYS}), SA_{SYS}, BA_{SYS}), PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}, VSG_{SYS})$$
$$KPL = (KPK, DKK, SK_{aggr}, DLK,\ IKK_{aggr})$$
$$DOK = g_1 \cdot DOK_{ST} + g_2 \cdot DOK_{PD} + g_3 \cdot DOK_{IN}.$$

Das Gesamtmaß der Portabilität läßt sich aus den bisher ermittelten Daten für das XCTL-System, wie folgt berechnen (Tabellenführung $\downarrow\nearrow\downarrow$):

$$SYS = (PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}) =$$

(956,
((*TAbout* , 2),
(*TAdjustmentExecute* , 2),
(*TAdjustmentWindow* , 4),
(*TAm9513a* , 49),
(*TAngleControl* , 32),
(*TAquisition* , 2),
(*TAreaScan* , 18),
(*TAreaScanCmd* , 1),
(*TAreaScanParameters* , 1),
(*TBitmapSource* , 24),
(*TBraun_Psd* , 48),
(*TC_812* , 22),
(*TC_812GPIB* , 17),
(*TC_812ISA* , 2),
(*TC_832* , 25),
(*TCalibrate* , 5),
(*TCalibratePsd* , 8),
(*TChooseDeviceCmd* , 1),
(*TCmd* , 2),
(*TCommonDevParam* , 4),
(*TCounterShowParam* , 2),
(*TCounterWindow* , 3),
(*TCurve* , 2),
(*TDC_Drive* , 5),
(*TDevice* , 17),
(*TDList* , 1),
(*TEditWindow* , 7),
(*TEncoder* , 4),
(*TExecuteCmd* , 2),
(*TGenericDevice* , 9),
(*TMacroExecute* , 5),
(*TMain* 22),
(*TMDIWindow* 10),
(*TMList* 2),
(*TModalDlg* 4),
(*TModelessDlg* 3),
(*TMotor* 10),
(*TMotorParam* 5),
(*TOptimizeDC_812* 12),
(*TPlotData* 8),
(*TPosControl* 7),
(*TPsd* 11),
(*TPsdRemoteSync* 4),
(*TRadicon* 16),
(*TScan* 17),
(*TScanCmd* 6),
(*TScsParameters* 1),
(*TSetAdjustmentParam* 1),
(*TSetupAreaScan* 3),
(*TSetupContinuousScan* 4),
(*TSetupScanCmd* 1),
(*TSetupStepScan* 6),
(*TShowValueCmd* 3),
(*TSteering* 9),
(*TStoe_Psd* 16),
(*TTopographyExecute* 8),
(*Tx* 441)),
0,703).

$$KPL = (KPK, DKK, SK_{aggr}, DLK, IKK_{aggr}) =$$
$$(0,506 \quad , \quad 0,027 \quad , \quad 817,23 \quad , \quad 2,87 \quad , \quad 1841,5).$$

$$DOK = 0,15 \cdot 0,5 + 0,6 \cdot 0,7 + 0,25 \cdot 0,36 = 0,59.$$

$$PORT = (SYS,\ KPL,\ DOK) =$$

```
(956,
((TAbout                 ,   2),
 (TAdjustmentExecute     ,   2),
 (TAdjustmentWindow      ,   4),
 (TAm9513a               ,  49),
 (TAngleControl          ,  32),
 (TAquisition            ,   2),
 (TAreaScan              ,  18),
 (TAreaScanCmd           ,   1),
 (TAreaScanParameters    ,   1),
 (TBitmapSource          ,  24),
 (TBraun_Psd             ,  48),
 (TC_812                 ,  22),
 (TC_812GPIB             ,  17),
 (TC_812ISA              ,   2),
 (TC_832                 ,  25),
 (TCalibrate             ,   5),
 (TCalibratePsd          ,   8),
 (TChooseDeviceCmd       ,   1),
 (TCmd                   ,   2),
 (TCommonDevParam        ,   4),
 (TCounterShowParam      ,   2),
 (TCounterWindow         ,   3),
 (TCurve                 ,   2),
 (TDC_Drive              ,   5),
 (TDevice                ,  17),
 (TDList                 ,   1),
 (TEditWindow            ,   7),
 (TEncoder               ,   4),
 (TExecuteCmd            ,   2),
 (TGenericDevice         ,   9),
 (TMacroExecute          ,   5),
 (TMain                  ,  22),
 (TMDIWindow             ,  10),
 (TMList                 ,   2),
 (TModalDlg              ,   4),
 (TModelessDlg           ,   3),
 (TMotor                 ,  10),
 (TMotorParam            ,   5),
 (TOptimizeDC_812        ,  12),
 (TPlotData              ,   8),
 (TPosControl            ,   7),
 (TPsd                   ,  11),
 (TPsdRemoteSync         ,   4),
 (TRadicon               ,  16),
 (TScan                  ,  17),
 (TScanCmd               ,   6),
 (TScsParameters         ,   1),
 (TSetAdjustmentParam    ,   1),
 (TSetupAreaScan         ,   3),
 (TSetupContinuousScan   ,   4),
 (TSetupScanCmd          ,   1),
 (TSetupStepScan         ,   6),
 (TShowValueCmd          ,   3),
 (TSteering              ,   9),
 (TStoe_Psd              ,  16),
 (TTopographyExecute     ,   8),
 (Tx                     , 441)),
 0,703                   ),
(0,506                   ,
 0,027                   ,
 817,23                  ,
 2,87                    ,
 1841,5                  ),
 0,59                    ).
```

Das Maß für die Systemumgebung SYS ist ein Maß, das konkrete Aufwandsdaten liefern kann. Im Falle des XCTL-Systems liegen noch keine Portierungserfahrungen vor, sodaß hier nur die Anzahl potentieller Anpassungen in das Maß einfließen. Die ermittelten Daten geben einen ersten Überblick über die eventuell anstehende Portierungsarbeit. Es hängt hierbei jeweils von den Erfahrungen des Porteurs ab, der die Daten interpretiert, wie aufschlußreich die Daten für ihn sind.

Das erste, für das XCTL-System, ermittelte Maß PAN_{SYS} gibt Auskunft darüber, wie hoch die Gesamtzahl der potentiell anzupassenden Systemelemente ist. Diese ergeben sich aus den Unterschieden von Host- und Zielumgebung und vermitteln einen ersten Eindruck über den zu erwartenden Umfang der Portierungsarbeiten. Im Falle des XCTL-Systems war hierbei die Hostumgebung Windows 3.1 (16 Bit) und die Zielumgebung Windows 98 (32 Bit).

$$PAN_{SYS} = 956 .$$

Das zweite Maß $KOMP_{SYS}$ liefert die Namen der potentiellen Änderungskandidaten mit der jeweiligen Anzahl anzupassender Systemelemente (siehe Abbildung 26). Für das XCTL-System wird damit ausgesagt, in welchen Systemkomponenten der größte Portierungsaufwand zu erwarten ist und es kann daraus gefolgert werden, welche Komponenten des Software-Systems mit relativ geringem Aufwand portiert werden können. Hierbei entstand das Problem der Zuordnung von Systemelementen, die in keiner Systemkomponente klassifiziert wurden, da das XCTL-System bisher nicht durchgängig objektorientiert implementiert ist. Diese Elemente wurden für die durchgeführte Portabilitätsanalyse in einer gedachten Systemkomponente 'Tx' gekapselt und so der Vermessung zugänglich gemacht. Somit können die hier zur Verfügung stehenden Informationen als erste Entscheidungsträger für oder gegen eine Portierung herangezogen werden. Sind die Umgebungszugriffe breit über das gesamte Software-System gestreut, muß sich der Porteur u.U. in viele Systemteile einarbeiten, sodaß dies wiederum einen höheren Einarbeitungsaufwand zur Folge hat. Damit ist das Maß auch ein Indiz dafür, in welchem Umfang Umgebungsabhängigkeiten in einigen wenigen Teilen des Software-Systems zusammengefaßt werden und somit den Prinzipien portabler Software (insbesondere der Lokalität) folgen. Die sinnvolle Aufteilung in entsprechende Systemkomponenten sollte sich auch in der Quelltextstruktur widerspiegeln. Aus der Analyse der Systemkomponenten kann man für jede dieser Komponenten angeben, in welcher Quelldatei sie definiert sind, bzw. in welcher Quelldatei potentiell die meisten Änderungen vorgenommen werden müßten (siehe Abbildung 27).

Das dritte Maß der Systemumgebung (VKG_{SYS}) bildet das Verhältnis von anzupassenden Systemkomponenten zur Gesamtzahl vorhandener Systemkomponenten. Damit wird, im Vergleich zum zweiten Maß, eine allgemeinere Aussage zur Portabilität des XCTL-Systems ermöglicht. Dieses Maß kann ebenfalls als Grundlage für eine Portierungsentscheidung dienen. Kommen noch Unterschiede in den Programmiersprachen hinzu, wie das z.B. bei einem Wechsel von einer Entwicklungsumgebung in eine andere der Fall ist (Querportierung), so kann dieser Aufwand u.a. durch das Maß VSG_{SYS} ausgedrückt werden.

$$VKG_{SYS} = 0,703.$$

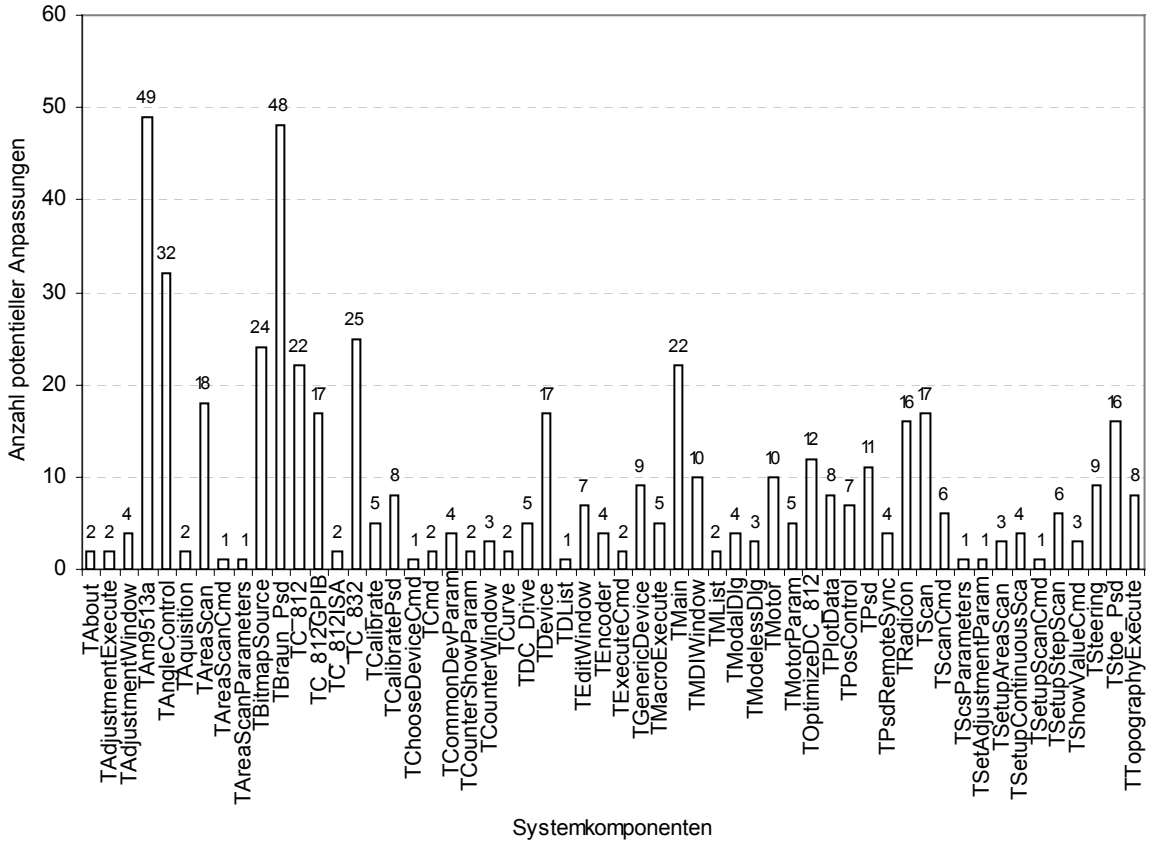


Abbildung 26: Ergebnis der Maßkomponente *KOMPsys* (ohne Komponente 'Tx')

Sobald das XCTL-System portiert worden ist, existieren weitergehende Informationen, die dem Porteur durch das System-Umgebungsmaß zur Verfügung gestellt werden können. Die konkretesten Ergebnisse liefert dabei das Maß BA_{SYS}. Bewertete Änderungsaufwände aus dem Systemumgebungsmaß und dem Programmiersprachenmaß, soweit sie durch frühere Portierungen in Erfahrung gebracht werden konnten, werden hier aufsummiert und liefern somit einen konkreten Anhaltspunkt für die Ermittlung einer einzuplanenden Portierungsdauer. Die Elemente und Konstrukte (Programmiersprachen), die bereits als anpaßbar identifiziert werden konnten, für die aber während der Portierung keine Aufwandsdaten zu ermitteln sind, werden durch das Maß SA_{SYS} berücksichtigt. Die nächsten Maße beinhalten die Anzahl und Namen von Betriebssystem-Routinen und eine Teilmenge von Programmiersprachenkonstrukten, über die während der Messung überhaupt keine Informationen vorliegen (PA_{SYS}). Sie werden separat ausgewiesen, weil diese Elemente als fester Bestandteil eines Betriebssystems oder einer bestimmten Programmiersprache integriert sind und somit keine Anpassungen zulassen sollten (KA_{SYS}, PA_{SYS}). Sie können sich, neben anderen BS-Routinen und PS-Konstrukten, über die solche Informationen vorliegen, bei einer Portierung als nicht anpaßbar erweisen (KA_{SYS}) und stellen somit einen besonderen Problembereich dar. Das Fehlen solcher Informationen kann sich stark auf die Funktionalität des Software-Systems auswirken und muß daher im weiteren Verlauf der Portierungsarbeiten genauer analysiert werden, sodaß evtl. sogar von einer Portierung abzusehen ist. Alle übrigen BS-Aufrufe und PS-Konstrukte sind zwar anpaßbar und gehen damit in das Maß SA_{SYS} ein, aber es ist, wie bei der Gesamtzahl potentieller Änderungskandidaten PAN_{SYS}, nicht sicher, ob Anpassungen auch wirklich erforderlich werden. Diese Aussagen sind somit eher als vage zu bezeichnen. Das Maß für die Systemumgebung liefert bei einer

ersten Analyse zwar konkrete Ergebnisse $(PAN_{SYS}, KOMP_{SYS}, VKG_{SYS})$, muß aber trotzdem mit konkreten Portierungserfahrungen angereichert werden $(KA_{SYS}, PA_{SYS}, SA_{SYS}, BA_{SYS}, VSG_{SYS})$, um die Ergebnisse korrekt interpretieren zu können und um zeitliche Aufwandsprognosen zu ermöglichen.

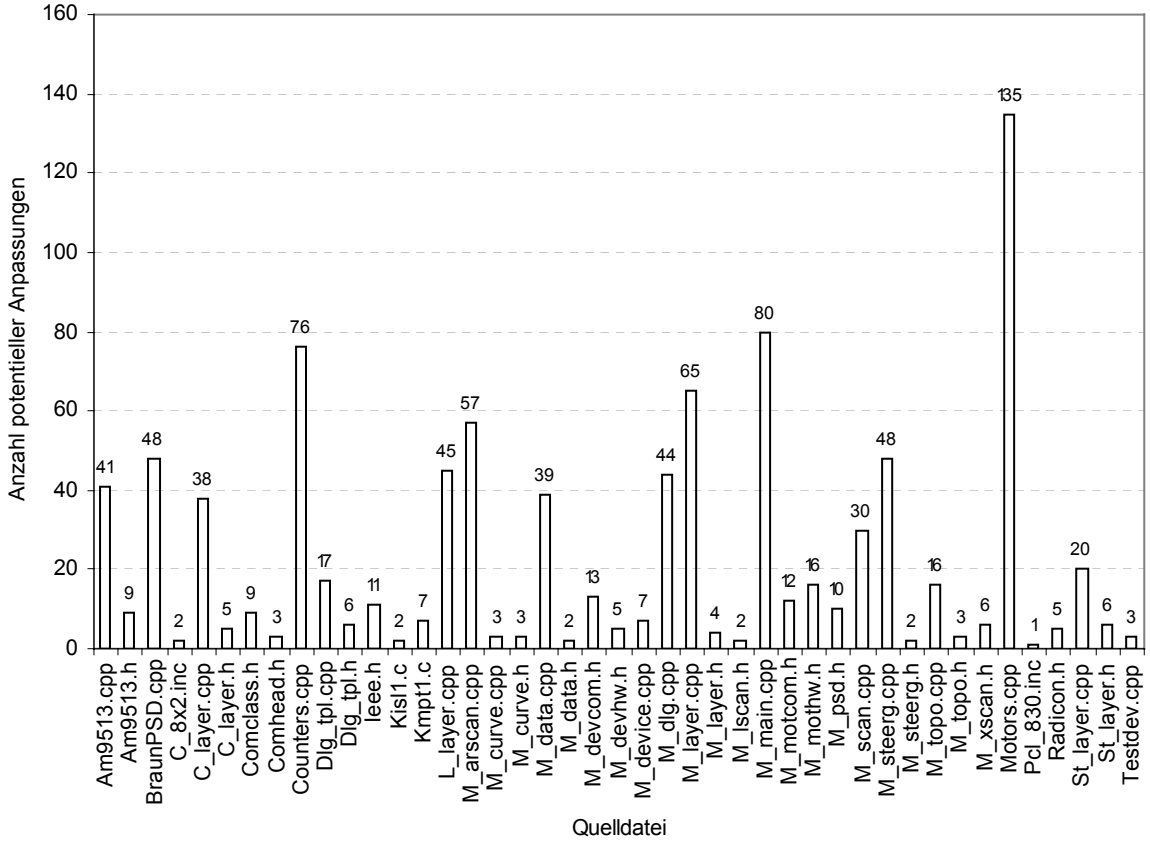


Abbildung 27: Anzahl potentieller Anpassungen, bzgl. der Struktur der Quelldateien

Bei dem darauf folgenden Komplexitätsmaß *KPL* werden keine konkreten Aufwände gemessen. Vielmehr wird über die Vermessung der Systemstruktur versucht, Aussagen zur Höhe des Änderungsaufwands zu ermöglichen. Darüber hinaus wird damit auch die Testbarkeit eines Software-Systems untersucht, da eine Portierung immer im abschließenden Systemtest endet. Die Vielschichtigkeit des Problems der Komplexität von Software-Systemen läßt jedoch nur eine gewisse Auswahl an wichtigen Einflußfaktoren zu, die im Hinblick auf die Portierung von Software, zwar erfaßt werden können, darüber hinaus aber eine weitergehende Interpretation, aufgrund fehlender Erfahrungsdaten, nicht zulassen. Auch hierbei entsteht durch die Aggregation zwar wieder eine kompaktere Aussage, aber auch diese ist mit Informationsverlust behaftet. Für genauere Analysen muß deshalb auf die Submaße zurückgegriffen werden. Auch ergibt sich hier wieder das Problem der Zuordnung 'verwaister' Systemelemente. Die Teile des Software-Systems, die keiner Systemkomponente, außer Tx, zugeordnet werden konnten, werden deshalb zunächst nicht berücksichtigt. Da das XCTL-System innerhalb eines Reengineering-Projekts bearbeitet wird, ist es jedoch wahrscheinlich, daß in einer weiteren Version der Portabilitätsanalyse, in diesem Teil, genauere Meßdaten zur Verfügung stehen.

Das Maß SK_{aggr}, als gewichtete Summe der Kopplungen zwischen den Systemkomponenten, innerhalb eines untersuchten Software-Systems, ist hierbei ein Indikator für die Kopplungsstärke zwischen den Systemkomponenten. Die Summe beinhaltet Portierungserfahrungen über die Gewichtung der Submaße und erlaubt Aussagen bezüglich des potentiellen Änderungsaufwands oder der Testbarkeit eines Software-Systems:

$$SK_{aggr} = 817,23.$$

Bei diesem Maß muß noch einmal darauf hingewiesen werden, daß bei der Vermessung des XCTL-Systems keine ausreichenden Meßwerte für den Zugriff auf globale Daten ermittelt werden konnten (Vergleich Anhang F). Der ermittelte Meßwert beträgt für jede Systemkomponente 0, d.h. daß man theoretisch alle momentan, als global deklarierten Klassenkomponenten, auch als 'private' deklarieren könnte, da auf keines von außen zugegriffen wird. Damit geht diese Art der Datenkopplung, bei der Portabilitätsanalyse des XCTL-Systems, zunächst nicht mit ein. Die Maße KPK und DKK bilden dann den jeweiligen Kopplungsgrad der Submaße:

$$KPK = 0,506 \text{ und } DKK = 0,027.$$

Analog zum Maß SK_{aggr} wird das Maß IKK_{aggr} aufgestellt, was die Interne Komplexität, angereichert durch Portierungserfahrungen, ausdrückt:

$$IKK_{aggr} = 1841,5 .$$

Neben diesen Maßen wird auch noch die durchschnittliche zyklomatische Komplexität DLK mit einbezogen, da diese einen guten Indikator für die Gesamtkomplexität des Software-Systems darstellt. Überschreitet der Meßwert eine vorgeschlagene Grenze von 10, wird der Portierungsaufwand für das Software-System mit großer Wahrscheinlichkeit so groß, daß vor der Portierung zunächst eine Reimplementierung (u.a. die Zerlegung in weitere Systemkomponenten) durchgeführt werden sollte oder sogar eine Neuimplementierung sinnvoll erscheint.

$$DLK = 2,87 .$$

Das Dokumentationsmaß für das XCTL-System liefert in der letzten Aggregation einen einzelnen Meßwert, der durch die sehr einheitliche Maßstruktur ermöglicht wird. Bei jeder Aggregation tritt jedoch wieder ein gewisser Informationsverlust auf und es stellt sich somit die Frage, was mit einer einzigen Maßzahl zur Dokumentation ausgesagt wird. Ein Problem hierbei ist die Granularität der einzelnen Maße, da diese in einem Binärwert das Vorhandensein von Dokumentationsteilen erfassen und somit eine sehr grobe, subjektive Messung vornehmen. Um den Einfluß der Dokumentation auf die Höhe des Portierungsaufwands zu bewerten, sind eine Reihe von Portierungen mit entsprechenden Untersuchungen notwendig. Da die Dokumentation in erster Linie einen Indikator für den Einarbeitungsaufwand in das entsprechende Software-System darstellt, lassen sich dann auch zeitliche Aufwandsprognosen für diesen Aspekt der Portierung ableiten.

$$DOK = 0,59 .$$

Im Zusammenhang mit zeitlichen Aufwandsprognosen kann es sinnvoll sein, die allgemein anerkannte Einheit von Mitarbeiter-Monaten zu verwenden. Hierzu muß dann ein Gesamtmaß 'Portabilität' gebildet werden, welches dann auf die Anzahl der benötigten Mitarbeiter-Monate, als zeitlicher Aufwand für die Portierung des Software-Systems, abgebildet wird. Dies kann z.B. durch die Summation der ermittelten Maßkomponenten erfolgen. Dafür sind jedoch eine Reihe empirischer Portierungsuntersuchungen notwendig, die dann einen entsprechend funktionalen Zusammenhang herausstellen, wie das z.B. bei der Function-Point-Methode der Fall ist.

Zusammenfassung – Anmerkungen – Ausblick

Ausgehend von der Fragestellung, ob der Aufwand für die Portierung von Software-Systemen im voraus bestimmbar ist, sollte ein Windows-basiertes Software-System zur Steuerung einer Röntgen-Topographie-Kamera (XCTL-System) im Rahmen eines Reengineering-Projekts untersucht werden. Hierbei ist sowohl die Portierung von einer 16-Bit-Plattform (Windows 3.1) in eine 32-Bit-Windows-Umgebung (Windows 98), als auch ein Wechsel in eine andere Entwicklungsumgebung (von Borland C++ nach Visual C++) geplant.

Zur Beantwortung der Fragestellung, bzgl. der Portierung auf die 32-Bit-Windows-Plattform, wurden zunächst alle wichtigen Aspekte zum Thema Portabilität systematisch zusammengetragen. Hierbei hat sich gezeigt, daß die Probleme sehr vielschichtig sind. Um die Handhabung von Portabilitätsuntersuchungen zu vereinfachen, wurden bestehende Portierungsmodelle vorgestellt und diskutiert. Auf der Grundlage dieser allgemeinen Überlegungen wurden dann im Speziellen die Probleme bei der Portierung Windows-basierter Software-Systeme behandelt und entsprechend der Fragestellung klassifiziert. Die systematische Darstellung der Portabilitätsprobleme, Strategien, Konzepte und Werkzeuge im Allgemeinen, wie auch im Speziellen, bildeten das Fundament für weitere Überlegungen, bzgl. der Aufwandsanalyse.

Um Software-Systeme zu vermessen, existieren eine Reihe von Metriken, die jeweils unterschiedliche Aspekte von Software-Systemen quantitativ erfassen. Hierzu gehören neben den klassischen Metriken auch objektorientierte Metriken, die auf jeweils unterschiedlich strukturierten Software-Systemen arbeiten. Die Frage nach speziellen Metriken zur Messung der Portabilität von Software-Systemen, führte in dieser Arbeit zur Ermittlung einer Reihe von Metrikvorschlägen, die diesen Bereich näher beleuchten. Hierbei stellte sich heraus, daß die meisten Vorschläge nur ein Verhältnismaß enthalten, daß den Portierungsaufwand eines Software-Systems in Beziehung zum Aufwand einer Neuimplementierung setzt, ohne jedoch darauf einzugehen, wie der Aufwand einer Portierung gemessen werden kann. Einige Vorschläge konkretisieren die Ermittlung des Portierungsaufwands und wurden als Basis für weitere Überlegungen herangezogen.

Auf der Grundlage des zuvor bewerteten Metrikvorschlags, zur Vermessung der Portabilität von prozedural strukturierten Software-Systemen, konnte der ursprünglichen Fragestellung, nach der Bestimmbarkeit des Portierungsaufwands, weiter nachgegangen werden. Da das zu untersuchende XCTL-System zum größten Teil dem objektorientierten Paradigma folgt, mußte eine entsprechende Anpassung des Metrikvorschlags für objektorientierte Software-Systeme vorgenommen werden. Dabei entstand die Frage nach der Übertragbarkeit klassischer Metriken auf objektorientierte Software-Systeme, wobei sich herausstellte, daß dies nur eingeschränkt der Fall ist, da die empirischen Objekte im objektorientierten Modell, mit Ausnahme der Methoden, konstitutiv anderer Art sind.

Auf der Grundlage der theoretischen Überlegungen zur Portabilität und der Vermessung von Software-Systemen, konnte daraufhin eine erste Portabilitätsanalyse des XCTL-Systems, im Hinblick auf die Portierung in eine 32-Bit-Windows-Umgebung, durchgeführt werden. Hierbei wurden die Aspekte Systemanpassung, Komplexität und Dokumentation berücksichtigt. Bei der Anpassung an die Systemumgebung ging es in erster Linie um die Ermittlung von potentiell anzupassenden Teilen des untersuchten Software-Systems. Somit entstehende Änderungsaufwände können durch komplexe Software-Systeme weiter erhöht werden, wodurch auch der Einarbeitungsaufwand beeinflußt wird.

Des weiteren wird der abschließend anfallende Systemtest deutlich aufwendiger als bei weniger komplexen Systemen. Der Einarbeitungs- und Änderungsaufwand wird darüber hinaus maßgeblich durch die vorhandene Dokumentation des untersuchten Software-Systems beeinflußt. Diese wurde durch ein manuelles Code-Review subjektiv bewertet.

Die so entstandenen Meßergebnisse liefern einen ersten Eindruck von der Beschaffenheit des XCTL-Systems im betrachteten Portierungsfeld. Hierbei wurde sowohl die Gesamtzahl potentieller Anpassungen, als auch jeder Systemkomponente, objektiv ermittelt und in Verhältnismaßen ausgedrückt. Die so ermittelten Daten geben damit einen Überblick über die eventuell anstehende Portierungsarbeit. Die Meßwerte der Komplexitätskomponente unterstützen zwar diesen objektiven Eindruck, lassen darüber hinaus aber, aufgrund fehlender Erfahrungsdaten, eine weitergehende Interpretation der Daten nicht zu. Der Meßwert für die Dokumentation hat ähnlichen Charakter. Um den Einfluß der Dokumentation und des ermittelten Meßwertes auf die Höhe des Portierungsaufwands bewerten zu können, sind auch hier eine Reihe von Portierungen, mit entsprechenden Untersuchungen, notwendig.

Das volle Potential der ermittelten Meßwerte erschließt sich somit erst, wenn konkrete Portierungserfahrungen vorliegen. Dann können die Maßzahlen mit konkreten Erfahrungsdaten aufgewertet werden und lassen mit der Zeit immer präzisere Aufwandsaussagen zu. Sie bilden dann eine solide Grundlage für bevorstehende Portierungsentscheidungen. Dieser evolutionäre Aspekt wird durch das Maß unterstützt und ist in allen Maßkomponenten integriert. Somit ist die geplante Portierung des XCTL-Systems eine gute Möglichkeit, konkrete Portierungserfahrungen in das Maß einfließen zu lassen. Erste Portierungsversuche im Portierungsfeld ’Win16-Borland C++ nach Win32-VisualC++’ wurden bereits durchgeführt [Bojic 02] (siehe Anhang G).

Der geplanten Querportierung von Borland C++ nach Visual C++ sollte jedoch zunächst eine entsprechende Portabilitätsanalyse vorausgehen, da diese im betrachteten Portierungsfeld noch nicht berücksichtigt wurde. Dies betrifft den Teil der Programmiersprachen und ist im Maß entsprechend berücksichtigt. Anschließend kann das so erweiterte Portierungsfeld mit konkreten Portierungserfahrungen angereichert werden. Es existiert somit eine umfassende theoretische Grundlage, den Aufwand der Portierung von Software-Systemen zu bestimmen. Die geplante Portierung des XCTL-Systems kann hierbei wichtige Aufwandsdaten liefern, die für zukünftige Portierungen, evtl. nicht nur im Windows-Umfeld, genutzt werden kann.

Ein weiterer Gedanke ist die Realisierung der Aufwandsbestimmung von Portierungen durch ein entsprechendes Metrik-Werkzeug. Solch ein Werkzeug hätte nicht nur den großen Vorteil, die Meßwerte automatisch zu ermitteln, sondern könnte darüber hinaus auch den Prozeß der Datenerfassung und Berechnung standardisieren. Auch könnte man die Aufwandsbestimmung in einen umfassenderen Werkzeugverbund integrieren. Damit wären dann entscheidende Schritte auf dem Weg zur ’Portierung auf Knopfdruck’ absolviert.

Literaturverzeichnis

[Abreu 94] Abreu, F.B.: *Object-Oriented Software-Engineering: Measuring and Controlling the Development Process*. Proceedings of the 4th Int. Conf. on Software-Quality McLean, VA, USA, 1994

[Adamov 85] Adamov, R.: *Structural Metric Proposal for Complex Software*. Dortmund, Universität, Dissertation, 1985

[Balzert 96] Balzert, H.: *Lehrbuch der Software-Technik: Software-Entwicklung*. Heidelberg, Berlin, Oxford: Spektrum, Akad., Verl., 1996

[Balzert 98] Balzert, H.: *Lehrbuch der Software-Technik: Software-Managment, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg, Berlin: Spektrum, Akad. Verl., 1998

[Basili 80] Basili, V.R.: *Product Metrics*. In: *Tutorial on Models and Metrics for Software-Managment and Engineering*. COMPSAC80, Computer Society Press, 1980

[Bojic 02] Bojic, D.: *Porting XCTL From Borland (16-Bit) to Visual C++ 6.0 (32 Bit)*. URL https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98/produktdoku/porting_vc++.txt - Aktualisierungsdatum: 01.03.2002. Humboldt-Universität zu Berlin

[Borland 96] Borland, Inc.: *Borland C++ 5.02 Programmierhandbuch*. Borland International Inc., 1996

[Bothe 01a] Bothe, K.: *Reverse Engineering: the Challenge of Large-Scale Real-World Educational Projects.* 14th Conference on Software Engineering Education and Training, Charlotte, USA, 2001

[Bothe 01b] Bothe, K.; Sacklowski, U.: *Praxisnähe durch Reverse Engineering-Projekte: Erfahrungen und Verallgemeinerungen*. 7. Workshop SEUH, Zürich, Schweiz, 2001

[Boyle 80] Boyle, J.M.: *Programm Adaption and Programm Transformation*. In: Ebert, R.; Lügger, J.; Göcke, L.: *Practice in Software Adaption and Maintenance (Proceedings of the SAM Workshop Berlin 1979)*. Amsterdam, New York, Oxford: North-Holland Publishing Company, 1980

[Brown 77] Brown, P.J.: *Software Portability : An Advanced Course*. Cambridge, England: Cambridge University Press, 1977

[Buchheit 92] Buchheit, M.: *Windows Programmierbuch*. Düsseldorf: Sybex, 1992

[Buschhorn 88] Buschhorn, M; Kuchnowski, H.J.: *Entwicklung eines Portabilitätsmaßes und Entwurf eines Tools zur Unterstützung der Portierung*. Dortmund, Universität, Dipl.-Arb., 1988

[Catanzaro 91] Catanzaro, B.J.: *The SPARC Technical Papers*. Springer, 1991

[Chidamber 94] Chidamber, S.R.; Kemerer, C.F.: *Towards a Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software-Engineering, Vol. 20, No.6, pp.476-493, 1994

[Conger 92] Conger, J. L.: *Windows API Bible*. Corte Madera, Calif.: Waite Group Press, 1992

[Doyle 84] Doyle, J.K.; Mandelberg K.I.: *A Portable PDP-11*. In: Simulator. Software, Practice and Expirience, Vol. 14 (11), Nov. 1984, pp. 1047-1059

[Fenton 92] Fenton, N.: *Software Metrics: A Rigoros Approach*. London: Chapman & Hall, 1998

[Fetcke 95] Fetcke, T.: *Softwaremetriken bei objektorientierter Porgrammierung*. Berlin, Techn. Univ., Dipl.-Arb., 1995

[Ford 77] Ford, B.: *Preparing Conventions For Parameters For Transportable Numerical Software*. In: Cowell, H. (Hrsg.): *Portability of Numerical Software (Workshop Oak Brook Illinois 1976)*. Lecture Notes in Computer Science 57, Berlin, Heidelberg, New York: Springer, 1977

[Frank 79] Frank, G.R.; Theaker, C.J.: *The Design of the MUSS Operating System*. SPE, Vol. 9, No. 8, Aug. 1979, pp. 599-620

[Freund 01] Freund, S.; Hepp, D.: *Vom Reverse Engineering zur Programmerweiterung: Automatische Justage für ein Röntgentopographie-Steuerprogramm*. Berlin, Humboldt-Universität, Dipl.-Arb., 2001

[Gewald 79] Gewald, K.; Haake G.; Pfadler, W.: *Software Engineering: Grundlagen und Technik rationaler Programmentwicklung*. 2. Auflage München: R. Oldenbourg Verlag, 1979

[Gilb 77] Gilb, T.: *Software-Metrics*. Cambridge, Massachusetts: Winthrop Publishers, 1977

[Gollnick 99] Gollnick, M; Lützkendorf, S: *Softwaremetriken*: McCabe-Report für das XCTL-System. URL https://www.informatik.hu-berlin.de/swt/lehre/PROJEKT98/produktdoku/mccabe_report/index.htm - Aktualisierungsdatum: 25.5.1999. Humboldt-Universität zu Berlin

[Großwendt 98] Grosswendt, V.: *API-Programmierung mit Windows 98*. Poing: Franzis, 1998

[Halstead 77] Halstead, M.H.: *Elements of Software Science*. New York: Elsevier North-Holland, 1977

[Hansen 78] Hansen, W.J.: *Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)*. SIGPLAN Notices, Vol. 13, 1978

[Hommel 80] Hommel, G.: *Portabilität von Software*. In: Schneider, H.J. (Hrsg.): *Portable Software*. Stuttgart: Teubner, 1980

[IEEE 93] IEEE Computer Society: *IEEE Standard for a Software Quality Metrics Methodology*. IEEE Standard 1061, 1993

[Infotech 80] *Use of Standard High-Level Languages for Portability*, Info State of the Art Report 63, Life Cycle Management 1 & 2, Maidenhead. Berkshire: Infotech 1980, p. 117-121

[ISO 91] ISO/IEC Standard: *ISO 9126 Software Product Evaluation – Quality Characteristics and Guidelines for Their Use*. 1991

[Kaindl 88] Kaindl, H.: *Portability of Software*. SIGPLAN Notices, Vol. 23, No. 6, 1988

[Kernighan 99] Kernighan, B. W.; Pike, R.: *The Practice of Programming*. Boston: Addison-Wesley, 1999

[Kipping 95] Kipping, D.: *Migrating to Windows 95*. Rocklin, CA: Prima Pub, 1995

[Kithara 01] Kithara Software GmbH.: *Homepage*. URL http://www.kithara.de. - Aktualisierungsdatum: 01.03.2002

[Lake 94] Lake, A.; Cook, C.: *Use of Factor Analysis to Develop OOP Software Complexity Metrics.* Annual Oregon Workshop on Software-Metrics, Oregon, USA, 1994

[Lauer 96] Lauer, T.: *Porting to Win32: A Guide to making your Applications ready for the 32-Bit Future of Windows*. New York: Springer, 1996

[LeCarme 89] LeCarme, O.; Gart, M.P: *Software Portability*. 2nd ed. McGraw-Hill, 1989

[Leong 83] Leong, S. et al.: *Methods for Transporting Programs*. Task Force Group, COMPSAC, 1983

[McCabe 76] McCabe, T.: *A Complexity Measure*. IEEE Transactions of Software Engingeering Vol. SE-1, No.3 pp. 312-327, 1976

[McCall 77] McCall, J.A.; Richards P.K.; Walters G.F.: *Factors in Software Qualitity*. Rome Air Development Center, 1977

[Meyers 77] Meyers, G.J.: *An Extension to the Cyclomatic Measure of Programm Complexity*. SIGPLAN Notices, 1977

[Microsoft 95] Microsoft Corporation: *Microsoft Windows 95: Die technische Referenz*. Unterschleißheim: Microsoft Press, 1995

[Microsoft 98] Microsoft Corporation: *Microsoft Windows 98: Die technische Referenz. Unterschleißheim*: Microsoft Press, 1998

[Microsoft 98a] Microsoft Corporation: *Windows NT Workstation Version 4.0: Die technische Referenz*. Unterschleißheim: Microsoft Press, 1998

[Microsoft 98b] Microsoft Corporation (Hg.): *MSDN Library Visual Studio 6.0-Release*. 1998

[Mooney 90] Mooney, J.D.: *Strategies for Supporting Application Portability*. IEEE Computer, V23 N11 pp. 59-70, Nov. 1990

[Mooney 93] Mooney, J.D.: *Issues in the Specification an Measurement of Software Portability*. Technical Report TR 93-6, Dept. of Statistics and Computer-Science, West Virginia University, Morgantown WV, 1993

[Mooney 94] Mooney, J.D.: *Portability and Reusability: Common Issues and Differences*. Technical Report TR 94-2, Dept. of Statistics and Computer-Science, West Virginia University, Morgantown WV, 1994

[Mooney 97] Mooney, J.D.: *Bringing Portability to the Software Process*. Technical Report TR 97-1, Dept. of Statistics and Computer-Science, West Virginia University, Morgantown WV, 1997

[Moreau 89] Moreau, D.R.; Dominick, W.D.: *Object-Oriented Graphical Information Systems: Research Plan and Evolution Metrics*. The Journal of Systems and Software, Vol. 10 pp.23-28, 1989

[Myers 75] Myers, G.J.: *Reliable Software Through Composite Design*. New York: Van Nostrand Reinholt Company, 1974

[Orth 74] Orth, B.: *Einführung in die Theorie des Messens*. Stuttgart: Verlag W. Kohlhammer, 1974

[Peck 78] Peck, J.L.; Schrack, G.F.: *The portability of quality software*: *experiences with BCPL*. In: Hibbard, W.G; Schuman, S.A. (Hrsg.): *Construction Quality Software (Proceedings of the IFIP Working Conference on Construction Quality Software)*. Amsterdam, New York, Oxford: North-Holland Publishing Company, 1978

[Perlis 68] NATO Science Committee: *Software Engineering Concepts and Techniques (Proceedings of the NATO Conferences Garmisch 1968)*. New York: Petrocelli/Charter, 1976

[Petzold 98] Petzold, C.: *Programming Windows - 5th ed*. Microsoft Press, 1998

[Pivaldi 93] Pirvali, B.: P*ortable Window Programming*. Graz: Techn. Univ., Dipl.-Arb., 1993

[Poole 73] Poole, P. C.; Waite, W.M.: *Portability and Adaptibility*. In: Goos, J.; Hartmanis, J.: *Advanced Course in Software Engineering.* Lecture Notes in Computer Science 30, Berlin, Heidelberg, New York: Springer, 1977

[Richter 99] Richter, J.: *Programming Applications for Microsoft Windows - 4th ed*. Microsoft Press 1999

[Roth 87] Roth, C.: *Ein Verfahren zur Quantifizierung der Strukturiertheit von Software*. In: Fromm, H.; Steinhoff A.. (Hrsg.): *Software-Metriken (Arbeitsgespräch der Fachgruppe Software-Engineering)*. Gesellschaft für Informatik, IBM Bildungszentrum, März 1987

[Schmidt 87] Schmidt, M.: *Über das Messen und Bewerten von Software-Qualität mit Maß und Metrik*. In: In: Fromm, H.; Steinhoff A.. (Hrsg.): *Software-Metriken (Arbeitsgespräch der Fachgruppe Software-Engineering)*. Gesellschaft für Informatik, IBM Bildungszentrum, März 1987

[Schützler 01] Schützler, K.: *Wiedergewinnung von Subsystemen durch Use-Case-Analyse und Dateirestrukturierung am Beispiel des XCTL-Systems*. Berlin, Humboldt-Universität, Dipl.-Arb., 2001

[SHARE 68] SHARE, Ad-Hoc Commitee on Universal Languages: *The Problem of Programming Communication with Changing Machines: A Proposed Solution*. CACM, 1, 1968, pp.12-15

[Sibley 61] Sibley, R. A.: *The SLANG-System*. CACM, 4, Jan. 1961, S.75-84

[Soltendick 96] Soltendick, W.: *Borland C++ Version 5 - Das Buch*. München: tewi-Verl., 1996

[Soltendick 99] Soltendick, W. (Hg.): *Das Win32 API - Band 1 LZ32, ComCtl32, Kernel32*. Vaterstetten: Computer und Literaturverl., 1999

[Soltendick 99a] Soltendick, W. (Hg.): *Das Win32 API - Band 2 Shell32, ComDlg32, OLE32, OLE-Interfaces, MS-DOS-Extensions in Windows 95*. Vaterstetten: Computer und Literaturverl., 1999

[Soltendick 99b] Soltendick, W. (Hg.): *Das Win32 API - Band 3 User32, advanced API*. Vaterstetten: Computer und Literaturverl., 1999

[Soltendick 99c] Soltendick, W. (Hg.): *Das Win32 API - Band 4 GDI32, Diskcopy, Multimedia-System*. Vaterstetten: Computer und Literaturverl., 1999

[Stevens 81] Stevens, W.P.: *Using Structured Design*. New York: John Wiley & Sons, 1981

[Stroustrup 98] Stroustrup, B.: *Die C++ Programmiersprache*. Bonn: Addison-Wesley, 1998

[Swan 95] Swan, T.: *Mastering Borland C++ 4.5 - 2. ed.* Indianapolis, Ind. : Sams, 1995

[Tegarden 92] Tegarden, D.P.; Sheetz, S.D.; Monarchi, D.E.: *Effectivness of traditional Software-Metrics for Object-oriented Systems*. 25th Hawaii International Conference on System Sciences, Proc. HICSS-92, San Diaego, 1992

[VMWare 01] VMWare, Inc.: *Homepage*. URL http://www.vmware.com. - Aktualisierungsdatum: 01.03.2002

[Weiskamp 93] Weiskamp, K.; Pronk, R.: *Windows 3.1 Insider: The Guide to hard-to-find and undocumented features*. New York: John Wiley & Sons, Inc., 1993

[Xctl 01] Schützler, K: *Projekt: Software-Sanierung*. URL https://www.informatik.hu-berlin.de/Institut/struktur/softwaretechnikII/lehre/PROJEKT98/index.html - Aktualisierungsdatum: 01.03.2002. Humboldt-Universität zu Berlin

[Zuse 91] Zuse, H.: *Software Complexity: Measures and Methods.* Berlin, New York: de Gruyter, 1991

[Zuse 98] Zuse, H.: *A framework of software measurement*. Berlin, New York: de Gruyter, 1998

Anhang A - Wesentliche Systemunterschiede des betrachteten Portierungsfelds

Windows NT	Windows 9x	Windows 3.x (Enhanced-Mode)
Echtes 32-Bit-System (CPU-Register, lineares Speichermodell etc.)	X	16-Bit-Verarbeitung, nur durch nicht standardisierte und daher wenig portable Verfahren wird eine eingeschränkte 32-Bit Verarbeitung ermöglicht.
Unterstützung von Multi-Prozessorsystemen (’symmetric multiprocessing’, SMP).		Keine Unterstützung von Multiprozessorsystemen.
Portable Implementation, Schichtenmodell.		Extrem prozessorabhängig und monolithisch implementiert. Win16 setzt auf einem anderen Betriebssystem (MSDOS) auf.
4 GB virtueller Adreßraum pro Applikation (wobei die obere Hälfte, also 2 GB für Betriebssystem-Code und -Daten reserviert sind).	X	Insgesamt 4 GB Adreßraum, der global von allen Applikationen und dem BS genutzt wird.
Vollständige Trennung aller Adreßräume voneinander und vom Betriebssystem.	X	Keine sichere Trennung der Speicherbereiche voneinander.
Der gesamte virtuelle Adreßraum (4 GB) einer Anwendung kann durch 32-Bit-Zugriffe als ein Segment behandelt werden, daher keine Segmentarithmetik notwendig.	X	Der Adreßraum teilt sich auf in eine Reihe von 64 KB-Segmenten.
Das ’demand paging’, mit dem die virtuelle Speicherverwaltung arbeitet, erlaubt die Kontrolle über die Attribute einer Speicherseite.	X	Zwar ist der Paging-Mechanismus des 386 im Enhanced-Mode aktiv, dieser stellt jedoch nur eine vereinfachte Form der virtuellen Speicherverwaltung dar.

Tabelle A1: Vergleich Win16 und Win32: Prozessor und Speicherverwaltung

Windows NT	Windows 9x	Windows 3.x (Enhanced-Mode)
Objektorientierter Entwurf: Betriebssystem-Objekte wie Dateien; Prozesse, Speicherbereiche etc. werden vom Kernel als Objekte behandelt.		Strukturierter Entwurf: Sammlung von Funktionen und Daten.
Volles ’preemptive multitasking’ und multiple Threads, inklusive der notwendigen IPC-Mechanismen.	X	Kooperatives Multitasking, keine multiplen Threads; IPC-Unterstützung nur durch DDE.
Portable und strukturierte Ausnahmen- und Fehlerbehandlung.	X	Keinerlei Fehlerbehandlungs-mechanismen vorhanden; Ausnahmen und Fehler müssen komplett manuell verfolgt

		und bearbeitet werden.
Der Kernel benutzt als nativen Zeichensatz Unicode. Alle wichtigen Alphabete, Silben oder-Symbolschriften sowie Sonderzeichen sind einheitlich verfügbar. Die Übersetzung in das ANSI-Format wird völlig transparent vorgenommen.		Es wird der ANSI-Zeichensatz benutzt, der mit dem OEM-Zeichensatz der Hardware (meist IBM-ASCII) nicht vollständig kompatibel ist. Konvertierungen von und nach ANSI müssen manuell vorgenommen werden.
Jedem Prozeß wird ein Limit für seinen Ressourcenverbrauch gesetzt, das sowohl den Gebrauch von Kernel-Objekten (Threads, Semaphoren etc.) als auch die Auslastung des Speichers berücksichtigt.		Anwendungen können, was Rechenzeit und Ressourcenverbrauch angeht; vom System nicht kontrolliert werden. Nichtkooperative Anwendungen können andere beeinträchtigen bzw. deren Ablauf völlig verhindern.

Tabelle A2: Vergleich Win16- und Win32-Systemkern

Windows NT	Windows 9x	Windows 3.x (Enhanced-Mode)
Gerätetreiber und diverse Dateisysteme (FAT, HPFS, NTFS etc.) können zur Laufzeit geladen und entfernt werden und weitgehend portabel in einer Hochsprache implementiert werden.	X	Treibermodell, das bimodale Implementation (DOS/Real-Mode und Windows/ Protected-Mode) erfordert. Treiber müssen als x86-Assembler-Code geschrieben werden. Es wird nur das FAT-Dateisystem von MS-DOS unterstützt.
Dateien können als ’memory, mapped files’ behandelt werden, indem die gesamte. Datei über Paging in den virtuellen Adreßraum der Anwendung gemappt wird (z.B. für ’shared memory’ oder zur Performance-Steigerung).	X	Kein vergleichbarer Mechanismus. Der gesamte globale Adreßraum kann als ein großer ’shared memory’-Bereich betrachtet werden.
Integrierte Unterstützung von CD-ROM-Laufwerken über ein eigenes Dateisystem (CDFS).	X	Gerätetreiber und residente Hi1fsprogramme simulieren ein FAT-ähnliches System
Asynchrone I/O-Operationen sind möglich. Prozesse können daher effizienter arbeiten..	X	Synchrones I/O-Modell, bei dem der Prozess bis zur Beendigung des I/O-Transfers warten muß
NTFS-Dateien können bis zu 2^64 Bytes umfassen. Damit sind auch extrem große, Festplatten und weitere, heute nur experimentell verfügbare Massenspeicher unter NT als ein Volume einsetzbar.	X (VFAT)	MS-DOS (und damit Windows) kann maximal 2^32 Byte lange Dateien bearbeiten.
NTFS arbeitet transaktionsbasiert und speichert Systeminformationen redundant ab.	X (VFAT)	MS-DOS speichert zwar die FAT zweimal ab, viele weitere Informationen zur sicheren Wiederherstellung eines korrumpierten Dateisystems fehlen.

		Transaktionen sind hier unbekannt.
NTFS implementiert ein gewisses Maß an Fehlertoleranz (z.B. durch 'disk striping') und stellt die Basis für weitere Maßnahmen ('mirroring', 'striping with parity') zur Verfügung.	X (VFAT zum Teil)	Ein vergleichbarer Mechanismus ist nicht vorhanden. Erst durch dedizierte Netzwerk-Software kann ein entsprechendes Verhalten erreicht werden.
NTFS-Dateien können durch die NT-Sicherheitsmechanismen vor dem unberechtigten Zugriff durch andere Benutzer geschützt werden.		MS-DOS kennt nur sehr eingeschränkte Schutzmechanismen.
Peer-to-Peer-Netzwerke werden von NT direkt unterstützt. Die notwendigen Hilfsprogramme sind integraler Bestandteil des Systems.	X	MS-DOS stellt keine Netzwerkunterstützung zur Verfügung. Mit Erweiterungen, wie Windows for Workgroups, ist jedoch ein Peer-to-Peer-Netzwerk möglich.

Tabelle A3: Vergleich Win16 und Win32: Ein/Ausgabe- und Dateisystem

X: Eigenschaft ähnlich zu WindowsNT unterstützt

Anhang B – Inhalt der zentralen Problembibliothek

Der Inhalt der zentralen Problembibliothek ’Port.ini’ ist in sechs Bereiche unterteilt: APIS für alle Windows-Funktionen, MESSAGES behandelt Nachrichten und SRUCTURES Strukturen, TYPES ist für die einfachen Datentypen, CONSTANTS für die Konstanten und MACROS für Makrodefinitionen zuständig. Im Bereich CUSTOM werden alle Portierungsprobleme hinterlegt, die sich keinem der oberen Bereiche zuordnen lassen.

```
[PORTTOOL]
; Um zu den einzelnen Punkten aus PORTTOOL heraus Hilfe zu erhalten, wird hier der Pfad
; eingetragen, unter dem WinHelp die API-Hilfedatei findet.
WinHelp=z:\hlp\api32wh.hlp
WinHelp16=d:\bc\bin\tcwhelp.hlp

[APIS]
; Das Format fuer die weiteren Eintraege, ist wie folgt (KEINE Umlaute!):
;   SuchSchluessel=Win32APIHilfeBegriff;Grund der Aenderung;Vorgeschlagene Massnahme;

; Optionale Varianten:
;   SuchSchluessel=APIHilfeBegriff;Grund der Aenderung; ;
;   SuchSchluessel=APIHilfeBegriff; ; ;
;   SuchSchluessel=APIHilfeBegriff; ;Vorgeschlagene Massnahme;
;   SuchSchluessel= ;Grund der Aenderung;Vorgeschlagene Massnahme;
;   SuchSchluessel= ;Grund der Aenderung;Vorgeschlagene Massnahme;
;   SuchSchluessel= ; ;Vorgeschlagene Massnahme;
; Abschliessende Semikola sind optional.

;Zuordnung zu den Problemklassen
; [LA] (109) Benutzung eines virtuellen linearen 32-Bit-Adressraumes
; [SEP] (34) Separation aller Prozesse bzw. ihrer Adressraueme
; [MBE] (46) Globale Aenderungen am Modell fuer Benutzereingaben
; [GDI] (71) Erweitertes Koordinatensystem und weitere Aenderungen im GDI
; [DOS] (21) Wegfall MS-DOS- und 80x86-spezifischer Aufrufe
; [DOK] (4) Benutzung undokumentierter WIN-16-Eigenschaften
; ()=Aktuelle Gesamtzahl der potentiellen Aenderungen in zugeordneter Problemklasse

; Die erste Zeile muss zweimal erscheinen, damit sie einmal geparst wird...
AccessResource=AccessResource;Kein Win32-Aequivalent;[LA] Nicht erforderlich, loeschen;
AccessResource=AccessResource;Kein Win32-Aequivalent;[LA] Nicht erforderlich, loeschen;
AddFontResource=AddFontResource;Nur Dateinamen (String), keine Handles verwenden;[GDI];
AllocDSToCSAlias=AllocDSToCSAlias;Kein Win32-Aequivalent;[LA];
AllocResource=AllocResource;Kein direktes Win32-Aequivalent;[LA] Ersetzen durch Load/Find/LockResource;
AllocSelector=AllocSelector;Kein Win32-Aequivalent;[LA];
AnsiLower=AnsiLower;Makro um CharLower;[MBE];
AnsiLowerBuff=AnsiLowerBuff;Makro um CharLowerBuff;[MBE];
AnsiNext=AnsiNext;Makro um CharNext;[MBE];
AnsiPrev=AnsiPrev;Makro um CharPrev;[MBE];
AnsiToOem=AnsiToOem;Makro um CharToOem;[MBE];
AnsiToOemBuff=AnsiToOemBuff;Makro um CharToOemBuff;[MBE];
AnsiUpper=AnsiUpper;Makro um CharUpper;[MBE];
AnsiUpperBuff=AnsiUpperBuff;Makro um CharUpperBuff;[MBE];
Catch=Catch;Kein Win32-Aequivalent;[SEP] Ersetzen durch Structured Exception Handling (SEH);
ChangeMenu=ChangeMenu;Neue Funktionen verfuegbar;[GDI] Ersetzen durch portable Funktionen;
ChangeSelector=ChangeSelector;Kein Win32-Aequivalent;[LA];
CloseComm=CloseComm;COMM-Funktionen auf File-I/O gemappt;[DOS] Ersetzen durch CloseHandle;
CloseSound=CloseSound;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
CountVoiceNotes=CountVoiceNotes;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
DefHookProc=DefHookProc;Veraltete Hook-API, erzeugt nur Thread-lokale Hooks;[SEP] Neue CallNextHookEx verwenden;
DefineHandleTable=DefineHandleTable;Kein Win32-Aequivalent;[LA] Nicht erforderlich, loeschen;
DeviceCapabilities=DeviceCapabilities;Kein Win32-Aequivalent;[DOS] Ersetzen durch portable DeviceCapabilitiesEx;
DeviceMode=DeviceMode;Kein Win32-Aequivalent;[DOS] Ersetzen durch portable DeviceModeEx;
DialogProc=DialogProc;Dialog-Prozeduren sollten portabel definiert werden;[GDI] BOOL CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam);
DirectedYield=DirectedYield;Kein Win32-Aequivalent;[GDI];
DlgDirSelect=DlgDirSelect;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable DlgDirSelectEx;
DlgDirSelectComboBox=DlgDirSelectComboBox;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable DlgDirSelectComboBoxEx;
DlgProc=DialogProc;Dialog-Prozeduren sollten portabel definiert werden;[GDI] BOOL CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM
lParam);
DOS3Call=DOS3Call;Kein Win32-Aequivalent;[DOS] Ersetzen durch benanntes, portables Win32-API;
EnumTaskWindows=EnumTaskWindows;Makro um EnumThreadWindows;[SEP];
ExitWindows=ExitWindows;EW_*-Konstanten nicht mehr unterstuetzt;[GDI] Siehe evtl. ExitWindowsEx;
ExitWindowsExec=ExitWindowsExec;Kein Win32-Aequivalent;[GDI] Evtl. ersetzen durch ExitWindowsEx;
ExtDeviceMode=ExtDeviceMode;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable ExtDeviceModeEx;
ffree=free;NEAR/FAR-Funktionen nicht mehr definiert;[SEP] Entweder Makros in WINDOWSX.H benutzen oder free;
FlushComm=FlushComm;Kein Win32-Aequivalent;[DOS] Ersetzen durch PurgeComm;
fmalloc=malloc;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder malloc;
fmemccpy=memccpy;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memccpy;
fmemchr=memchr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memchr;
fmemcmp=memcmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memcmp;
fmemcpy=memcpy;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memcpy;
fmemicmp=memicmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memicmp;
fmemmove=memmove;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memmove;
fmemset=memset;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder memset;
fmsize=_msize;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder _msize;
frealloc=realloc;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder realloc;
FreeModule=FreeModule;Makro um FreeLibrary;[SEP] Ersetzen durch FreeLibrary;
FreeProcInstance=FreeProcInstance;Leeres Makro;[SEP] Nicht erforderlich, loeschen;
FreeResource=FreeResource;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
FreeSelector=FreeSelector;Kein Win32-Aequivalent;[LA];
fstrcat=strcat;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strcat;
fstrchr=strchr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strchr;
```

fstrcmp=strcmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strcmp;
fstrcpy=strcpy;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strcpy;
fstrcspn=strcspn;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strcspn;
fstrdup=strdup;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strdup;
fstricmp=stricmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder stricmp;
fstrlen=strlen;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strlen;
fstrlwr=strlwr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strlwr;
fstrncat=strncat;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strncat;
fstrncmp=strncmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strncmp;
fstrncpy=strncpy;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strncpy;
fstrnicmp=strnicmp;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strnicmp;
fstrnset=strnset;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strnset;
fstrpbrk=strpbrk;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strpbrk;
fstrrchr=strrchr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strrchr;
fstrrev=strrev;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strrev;
fstrset=strset;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strset;
fstrspn=strspn;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strspn;
fstrstr=strstr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strstr;
fstrtok=strtok;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strtok;
fstrupr=strupr;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strupr;
GetActiveWindow=GetActiveWindow;Rueckgabe kann gleich 0 sein;[GDI] Lokalen Eingabestatus beruecksichtigen;
GetAspectRatioFilter=GetAspectRatioFilter;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetAspectRatioFilterEx;
GetAtomHandle=GetAtomHandle;Kein Win32-Aequivalent;[MBE];
GetBitmapDimension=GetBitmapDimension;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetBitmapDimensionEx;
GetBrushOrg=GetBrushOrg;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetBrushOrgEx;
GetCapture=GetCapture;Rueckgabe kann gleich 0 sein;[MBE] Lokalen Eingabestatus beruecksichtigen;
GetClassWord=GetClassWord;Verbreiterte Datentypen beruecksichtigen;[GDI] GetClassLong fuer Werte, die auf 32 Bit gewachsen sind;
GetCodeHandle=GetCodeHandle;Kein Win32-Aequivalent;[GDI];
GetCodeInfo=GetCodeInfo;Kein Win32-Aequivalent;[GDI];
GetCommError=GetCommError;Kein Win32-Aequivalent;[GDI] Ersetzen durch ClearCommError;
GetCurrentPDB=GetCurrentPDB;Kein Win32-Aequivalent;[MBE] Evtl. GetCommandLine benutzen;
GetCurrentPosition=GetCurrentPosition;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetCurrentPositionEx;
GetCurrentTask=GetCurrentTask;Kein Win32-Aequivalent;[SEP] GetCurrentThread/Process benutzen;
GetDOSEnvironment=GetDOSEnvironment;[Kein Win32-Aequivalent;[DOS] Evtl. GetEnvironmentStrings benutzen;
GetEnvironment=GetEnvironment;Kein Win32-Aequivalent;[DOS];
GetFileResource=GetFileResource;Kein Win32-Aequivalent;[DOS];
GetFileResourceSize=GetFileResourceSize;Kein Win32-Aequivalent;[DOS];
GetFocus=GetFocus;Rueckgabe kann gleich 0 sein;[GDI] Lokalen Eingabestatus beruecksichtigen;
GetFreeSpace=GetFreeSpace;Kein Win32-Aequivalent;[LA] Ersetzen durch GlobalMemoryStatus;
GetFreeSystemResources=GetFreeSystemResources;Kein Win32-Aequivalent;[LA];
GetInstanceData=GetInstanceData;Kein Win32-Aequivalent;[SEP] Ersetzen durch IPC-Mechanismen;
GetKBCodePage=GetKBCodePage;Kein Win32-Aequivalent;[SEP];
GetMetaFileBits=GetMetaFileBits;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetMetaFileBitsEx;
GetModuleUsage=GetModuleUsage;Kein Win32-Aequivalent;[SEP];
GetNumTask=GetNumTask;Kein Win32-Aequivalent;[LA];
GetSelectorBase=GetSelectorBase;Kein Win32-Aequivalent;[LA];
GetSelectorLimit=GetSelectorLimit;Kein Win32-Aequivalent;[LA];
GetSysModalWindow=GetSysModalWindow;Kein Win32-Aequivalent;[SEP];
GetTempDrive=GetTempDrive;Kein Win32-Aequivalent;[LA] Siehe GetTempPath;
GetTextExtent=GetTextExtent;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetTextExtentPoint;
GetTextExtentEx=GetTextExtentEx;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable GetTextExtentExPoint;
GetThresholdEvent=GetThresholdEvent;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
GetThresholdStatus=GetThresholdStatus;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
GetTimerResolution=GetTimerResolution;Kein Win32-Aequivalent;[LA];
GetViewportExt=GetViewportExt;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable GetViewportExtEx;
GetViewportOrg=GetViewportOrg;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable GetViewportOrgEx;
GetWindowExt=GetWindowExt;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable GetWindowExtEx;
GetWindowOrg=GetWindowOrg;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable GetWindowOrgEx;
GetWindowTask=GetWindowTask;Makro um GetWindowThreadProcessId;[SEP];
GetWindowWord=GetWindowWord;Verbreiterte Datentypen beruecksichtigen;[SEP] GetWindowLong fuer Werte, die auf 32 Bit gewachsen sind;
GetWinFlags=GetWinFlags;Kein Win32-Aequivalent;[DOS] Ersetzen durch GetSystemInfo;
GlobalCompact=GlobalCompact;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
GlobalDosAlloc=GlobalDosAlloc;Kein Win32-Aequivalent;[DOS];
GlobalDosFree=GlobalDosFree;Kein Win32-Aequivalent;[DOS];
GlobalFix=GlobalFix;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
GlobalLRUNewest=GlobalLRUNewest;Leeres Makro;[LA] Nicht erforderlich, loeschen;
GlobalLRUOldest=GlobalLRUOldest;Leeres Makro;[LA] Nicht erforderlich, loeschen;
GlobalNotify=GlobalNotify;Kein Win32-Aequivalent;[LA];
GlobalPageLock=GlobalPageLock;Kein Win32-Aequivalent;[LA] Evtl. VirtualLock benutzen;
GlobalPageUnlock=GlobalPageUnlock;Kein Win32-Aequivalent;[LA] Evtl. VirtualUnLock benutzen;
GlobalUnfix=GlobalUnfix;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
GlobalUnwire=GlobalUnwire;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
GlobalWire=GlobalWire;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
int86=int86;Kein Win32-Aequivalent;[DOS] Ersetzen durch benanntes, portables Win32-API;
intdos=intdos;Kein Win32-Aequivalent;[DOS] Ersetzen durch benanntes, portables Win32-API;
IsGDIObject=IsGDIObject;Kein Win32-Aequivalent;[GDI];
IsTask=IsTask;Kein Win32-Aequivalent;[SEP];
LibMain=DllEntryPoint;DLL-Initialisierung geaendert;[SEP] Anpassen an Win32;
LimitEmsPages=LimitEmsPages;Kein Win32-Aequivalent;[LA];
LocalCompact=LocalCompact;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
LocalInit=LocalInit;Kein Win32-Aequivalent;[LA];
LocalNotify=LocalNotify;Kein Win32-Aequivalent;[LA];
LocalShrink=LocalShrink;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
LockData=LockData;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
LockSegment=LockSegment;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
MakeProcInstance=MakeProcInstance;Leeres Makro;[SEP] Nicht erforderlich, loeschen;
MoveTo=MoveTo;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable MoveToEx;
ncalloc=calloc;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder calloc;
NetBIOSCall=NetBIOSCall;Kein Win32-Aequivalent;[DOS] Ersetzen durch benanntes, portables Win32-API;
nexpand=expand;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder expand;
nfree=free;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder free;
nmalloc=malloc;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder malloc;
nmsize=msize;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder msize;
nrealloc=realloc;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder realloc;
nstrdup=strdup;NEAR/FAR-Funktionen nicht mehr definiert;[LA] Entweder Makros in WINDOWSX.H benutzen oder strdup;
OemToAnsi=OemToAnsi;Makro um OemToChar;[MBE];
OemToAnsiBuff=OemToAnsiBuff;Makro um OemToCharBuff;[MBE];
OffsetViewportOrg=OffsetViewportOrg;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable OffsetViewportOrgEx;
OffsetWindowOrg=OffsetWindowOrg;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable OffsetWindowOrgEx;
OpenComm=OpenComm;COMM-Funktionen auf File-I/O gemappt;[DOS] Ersetzen durch CreateFile;
OpenSound=OpenSound;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
PostAppMessage=PostAppMessage;Makro um PostThreadMessage;[GDI] Ersetzen durch PostThreadMessage;
PrestoChangoSelector=PrestoChangoSelector;Kein Win32-Aequivalent;[LA];

ProfClear=ProfClear;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfFinish=ProfFinish;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfFlush=ProfFlush;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfInsChk=ProfInsChk;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfSampRate=ProfSampRate;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfSetup=ProfSetup;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfStart=ProfStart;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
ProfStop=ProfStop;Profiling API gestrichen, siehe Tools-Dokumentation;[GDI];
QuerySendMessage=QuerySendMessage;Kein Win32-Aequivalent;[MBE];
ReadComm=ReadComm;COMM-Funktionen auf File-I/O gemappt;[DOS] Ersetzen durch ReadFile;
RemoveFontResource=RemoveFontResource;Nur Dateinamen (String),keine Handles verwenden;[GDI];
ScaleViewportExt=ScaleViewportExt;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable ScaleViewportExtEx;
ScaleWindowExt=ScaleWindowExt;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable ScaleWindowExtEx;
SetActiveWindow=SetActiveWindow;;[GDI] Lokalen Eingabestatus beruecksichtigen;
SetBitmapDimension=SetBitmapDimension;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable SetBitmapDimensionEx;
SetBrushOrg=SetBrushOrg;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable SetBrushOrgEx;
SetCapture=SetCapture;;[GDI] Lokalen Eingabestatus beruecksichtigen;
SetClassWord=SetClassWord;Verbreiterte Datentypen beruecksichtigen;[LA] SetClassLong fuer Werte, die auf 32 Bit gewachsen sind;
SetCommEventMask=SetCommEventMask;Kein Win32-Aequivalent;[DOS] Ersetzen durch SetCommMask;
SetEnvironment=SetEnvironment;Kein Win32-Aequivalent;[SEP];
SetFocus=SetFocus;;[GDI] Lokalen Eingabestatus beruecksichtigen;
SetMessageQueue=SetMessageQueue;Unter Win32 nicht mehr erforderlich;[MBE] Einfach loeschen;
SetMetaFileBits=SetMetaFileBits;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable SetMetaFileBitsEx;
SetResourceHandler=SetResourceHandler;Kein Win32-Aequivalent;[GDI];
SetSelectorBase=SetSelectorBase;Kein Win32-Aequivalent;[LA];
SetSelectorLimit=SetSelectorLimit;Kein Win32-Aequivalent;[LA];
SetSoundNoise=SetSoundNoise;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetSwapAreaSize=SetSwapAreaSize;Kein Win32-Aequivalent;[LA];
SetSysModalWindow=SetSysModalWindow;Kein Win32-Aequivalent;[GDI];
SetViewportExt=SetViewportExt;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable SetViewportExtEx;
SetViewportOrg=SetViewportOrg;Kein Win32-Aequivalent;[SEP] Ersetzen durch portable SetViewportOrgEx;
SetVoiceAccent=SetVoiceAccent;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetVoiceEnvelope=SetVoiceEnvelope;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetVoiceNote=SetVoiceNote;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetVoiceQueueSize=SetVoiceQueueSize;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetVoiceSound=SetVoiceSound;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetVoiceThreshold=SetVoiceThreshold;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SetWindowExt=SetWindowExt;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable SetWindowExtEx;
SetWindowOrg=SetWindowOrg;Kein Win32-Aequivalent;[GDI] Ersetzen durch portable SetWindowOrgEx;
SetWindowsHook=SetWindowsHook;Veraltete Hook-API, erzeugt nur Thread-lokale Hooks;[GDI] Neue SetWindowsHookEx verwenden;
SetWindowWord=SetWindowWord;Verbreiterte Datentypen beruecksichtigen;[GDI] SetWindowLong fuer Werte, die auf 32 Bit gewachsen sind;
StartSound=StartSound;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
StopSound=StopSound;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
SwitchStackBack=SwitchStackBack;[Kein Win32-Aequivalent;[LA];
SwitchStackTo=SwitchStackTo;Kein Win32-Aequivalent;[LA];
SyncAllVoices=SyncAllVoices;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
Throw=Throw;Kein Win32-Aequivalen;[SEP] Ersetzen durch Structured Exception Handling (SEH);
UngetCommChar=UngetCommChar;Kein Win32-Aequivalent;[DOS];
UnhookWindowsHook=UnhookWindowsHook;Veraltete Hook-API, erzeugt nur Thread-lokale Hooks;[SEP] Neue UnhookWindowsHookEx verwenden;
UnlockData=UnlockData;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
UnlockResource=UnlockResource;Leeres Makro;[LA] Nicht erforderlich, loeschen;
UnlockSegment=UnlockSegment;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
UnrealizeObject=UnrealizeObject;Unter Win32 nicht mehr erforderlich;[LA] Einfach loeschen;
ValidateCodeSegments=ValidateCodeSegments;Kein Win32-Aequivalent;[LA] ;
ValidateFreeSpaces=ValidateFreeSpaces;Kein Win32-Aequivalent;[LA];
WaitSoundState=WaitSoundState;Kein Win32-Aequivalent;[GDI] Ersetzen durch multimedia sound support oder PlaySound/Beep;
WEP=DllEntryPoint;DLL-Terminierung geaendert;[SEP] Anpassen an Win32;
WindowProc=WindowProc;Window-Prozeduren sollten portabel definiert werden;[SEP] LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
WndProc=WindowProc;Window-Prozeduren sollten portabel definiert werden;[SEP] LRESULT CALLBACK WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
WriteComm=WriteComm;COMM-Funktionen auf File-I/O gemappt;[DOS] Ersetzen durch WriteFile;
Yield=Yield;Kein Win32-Aequivalent;[SEP] Ersetzen durch PeekMessage oder Sleep;

[MESSAGES]
EM_GETSEL=EM_GETSEL;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
EM_LINESCROLL=EM_LINESCROLL;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
EM_SETSEL=EM_SETSEL;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_ACTIVATE=WM_ACTIVATE;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_CHANGECBCHAIN=WM_CHANGECBCHAIN;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_CHARTOITEM=WM_CHARTOITEM;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_COMMAND=WM_COMMAND;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_COMMNOTIFY=WM_COMMNOTIFY;Nachricht unter Win32 gestrichen;[MBE] Siehe □berlappende File-I/O-Funktion;
WM_CTLCOLOR=WM_CTLCOLOR;Ersetzt durch 7 neue Nachrichten, Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_DDE_ACK=WM_DDE_ACK;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_ADVISE=WM_DDE_ADVISE;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_DATA=WM_DDE_DATA;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_EXECUTE=WM_DDE_EXECUTE;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_POKE=WM_DDE_POKE;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_REQUEST=WM_DDE_REQUEST;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_TERMINATE=WM_DDE_TERMINATE;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_DDE_UNADVISE=WM_DDE_UNADVISE;lParam fuer Informationen nicht ausreichend;[MBE] PackDDElParam etc. benutzen;
WM_HSCROLL=WM_HSCROLL;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_MDIACTIVATE=WM_MDIACTIVATE;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_MDISETMENU=WM_MDISETMENU;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_MENUCHAR=WM_MENUCHAR;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_MENUSELECT=WM_MENUSELECT;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_PARENTNOTIFY=WM_PARENTNOTIFY;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_QUIT=WM_QUIT;Falls in einem PostMessage-Aufruf, Fehlfunktionen unter Win32 moeglich;[MBE] Ersetzen durch PostQuitMessage;
WM_VKEYTOITEM=WM_VKEYTOITEM;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;
WM_VSCROLL=WM_VSCROLL;Information in wParam/lParam anders verpackt;[MBE] message cracker oder alternative Makroschale benutzen;

[STRUCTURES]
cbClsExtra=WNDCLASS;Falls != 0, verbreiterte Datentypen beruecksichtigen;[LA] Evtl. Groesse anpassen;
cbWndExtra=WNDCLASS;Falls != 0, verbreiterte Datentypen beruecksichtigen;[LA] Evtl. Groesse anpassen;
DCB=DCB;Bitfelder geaendert, neue Komponenten;[DOS];

[TYPES]
HTASK=HTASK;Datentyp gestrichen;[LA] Ersetzen durch Thread-Id (DWORD);
LONG=LONG;LONG-Variable oder -Parameter pruefen;[LA] Ggf. durch LPARAM / LRESULT ersetzen;
(short)=short;Cast auf 16 oder 32 Bit pruefen;[LA] 16 Bit Datentypen mit ihren 32-Bit-Aequivalenten ersetzen;
(WORD)=WORD;Cast auf 16 oder 32 Bit pruefen;[LA] 16 Bit Datentypen mit ihren 32-Bit-Aequivalenten ersetzen;

```
WORD=WORD;WORD-Variable oder -Parameter pruefen;[LA] Ggf. durch UINT oder WPARAM ersetzen;

[CONSTANTS]
CS_GLOBALCLASS=RegisterClass;Windows-Klassen sind nicht nicht mehr global;[SEP] Explizites Laden der betreffenden DLL;
GCW_ATOM=GetClassLong;Kein Win32-Aequivalent;[SEP];
GCW_CBWNDEXTRA=GetClassLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GCL_CBWNDEXTRA;
GCW_CBCLSEXTRA=GetClassLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GCL_CBCLSEXTRA;
GCW_HCURSOR=GetClassLong;Datentyp-Verbreiterung;[GDI] Ersetzen durch GCL_HCURSOR;
GCW_HBRBACKGROUND=GetClassLong;Datentyp-Verbreiterung;[GDI] Ersetzen durch GCL_HBRBACKGROUND;
GCW_HICON=GetClassLong;Datentyp-Verbreiterung;[GDI] Ersetzen durch GCL_HICON;
GCW_HMODULE=GetClassLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GCL_HMODULE;
GCW_STYLE=GetClassLong;Datentyp-Verbreiterung;[GDI] Ersetzen durch GCL_STYLE;
GMEM_DDESHARE=GMEM_DDESHARE;Unter Win32 wirkungslos;[SEP] shared memory ersetzen durch IPC-Mechanismen;
GMEM_SHARE=GMEM_SHARE;Unter Win32 wirkungslos;[SEP] shared memory ersetzen durch IPC-Mechanismen;
GWW_HINSTANCE=GetWindowLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GWL_HINSTANCE;
GWW_HWNDPARENT=GetWindowLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GWL_HWNDPARENT;
GWW_ID=GetWindowLong;Datentyp-Verbreiterung;[LA] Ersetzen durch GWL_ID;
GWW_USERDATA=GetWindowLong;Datentyp-Verbreiterung;[MBE] Ersetzen durch GWL_USERDATA;

[MACROS]
HIWORD=HIWORD;HIWORD-Ziel 16 oder 32 Bit? Bei Nachrichtenparametern: geaenderte Informationspackung?;[MBE] Evtl. message cracker verwenden;
LOWORD=LOWORD;LOWORD-Ziel 16 oder 32 Bit? Bei Nachrichtenparametern: geaenderte Informationspackung?;[MBE] Evtl. message cracker verwenden;
MAKELONG=MAKELONG;Wenn Ziel lParam: geaenderte Informationspackung?;[MBE] Evtl. message cracker verwenden;
MAKELP=MAKELP;Keine FAR-Pointer unter Win32;[LA] Ersetzen durch flat memory model Code;
MAKELPARAM=MAKELPARAM;Geaenderte Informationspackung?;[MBE] Evtl. message cracker verwenden;
MAKEPOINT=MAKEPOINT;sizeof(POINT) != sizeof(DWORD);[LA] Entweder ersetzen durch MAKEPOINTS und POINTSTOPOINT oder eigene Konversionsfunktion;
OFFSETOF=OFFSETOF;Keine FAR-Pointer unter Win32;[LA] Ersetzen durch flat memory model Code;
SELECTOROF=SELECTOROF;Keine FAR-Pointer unter Win32;[LA] Ersetzen durch flat memory model Code;

[CUSTOM]
export=export;Keine export-Aufrufkonvention unter Win32;[SEP] CALLBACK oder WINAPI benutzen;
far=far;Win32 kennt keine Segmente, daher FAR == NEAR == Nichts!;[LA];
FAR=far;Win32 kennt keine Segmente, daher FAR == NEAR == Nichts!;[LA];
huge=huge;huge-Bereiche ueberfluessig;[LA];
HUGE=huge;huge-Bereiche ueberfluessig;[LA];
near=near;Win32 kennt keine Segmente, daher FAR == NEAR == Nichts!;[LA];
NEAR=near;Win32 kennt keine Segmente, daher FAR == NEAR == Nichts!;[LA];
pascal=pascal;Keine pascal-Aufrufkonvention unter Win32;[LA] CALLBACK oder WINAPI benutzen;
PASCAL=pascal;Keine pascal-Aufrufkonvention unter Win32;[LA] CALLBACK oder WINAPI benutzen;
stress.h=STRESS.H;Stress-Funktionen bis auf weiteres nicht verfuegbar;[DOK] Durch entsprechendes Win32-API ersetzen oder streichen;
STRESS.H=STRESS.H;Stress-Funktionen bis auf weiteres nicht verfuegbar;[DOK] Durch entsprechendes Win32-API ersetzen oder streichen;
toolhelp.h=TOOLHELP.H;ToolHelper-Funktionen bis auf weiteres nicht verfuegbar;[DOK] Durch entsprechendes Win32-API ersetzen oder streichen;
TOOLHELP.H=TOOLHELP.H;ToolHelper-Funktionen bis auf weiteres nicht verfuegbar;[DOK] Durch entsprechendes Win32-API ersetzen oder streichen;
```

Anhang C – Wichtige Datentypen im Vergleich

Datentyp	Win16	Win32	Bemerkung
int	int (2)	int INT (4)	[1]
char	char (1)	char, CHAR (1)	
short	short (2)	short, SHORT (2)	
BOOL	int (2)	int (4)	[1]
BYTE	unsigned char (1)	unsigned char 1)	
WORD	unsigned short (2)	unsigned short (2)	
DWORD	unsigned long (4)	unsigned long (4)	
UINT	unsiged int (2)	unsiged int (4)	[1]
LONG	signed long (4)	signed long (4)	
WPARAM	UINT (2)	UINT (4)	[1]
LPARAM	LONG (4)	LONG (4)	
LRESULT	LONG (4)	LONG (4)	
PSTR	char NEAR * (2)	char * (4)	[1]
NPSTR	char NEAR * (2)	char *. (4)	[1]
LPSTR	char FAR * (4)	char * (4)	[2]
LPCSTR	const char FAR * (4)	const char * (4)	[2]
PBYTE	BYTE NEAR * (2)	BYTE * (4)	[1]
LPBYTE	BYTE FAR * (4)	BYTE " (4)	[2]
PINT	int NEAR * (2)	INT * (4)	[1]
LPINT	int FAR * (4)	INT * (4)	[2]
PWORD	WORD NEAR * (2)	WORD * (4)	[1]
LPWORD	WORD FAR * (4)	WORD * (4)	[2]
PLONG	long NEAR * (2)	long * (4)	[1]
LPLONG	long FAR * (4)	long * (4)	[2]
PDWORD	DWORD NEAR * (2)	DWORD * (4) -	[1]
LPDWORD	DWORD FAR * (4)	DWORD * (4)	[2]
LPVOID	void FAR * (4)	void * (4)	[2]
HANDLE (STRICT)	const void NEAR *(2)	void * (4)	[1]
HANDLE (normal)	UINT (2)	void * (4)	[1]
HWND (STRICT)	const struct HWND_NEAR * (2)	const struct HWND_ * (4)	[1,3]
HWND (normal)	UINT	HANDLE.	[1,3]
PHANDLE	HANDLE * (2)	HANDLE * (4)	[1,4]
SPHANDLE	HANDLE NEAR * (2)	HANDLE'* (4)	[1]
LPHANDLE	HANDLE FAR * (4)	HANDLE * (4)	[2]
ATOM	UINT (2)	WORD (2)	[5[
HFILE	int (2)	int (4)	[1]
FARPROC	void (CALLBACK *) (void) (4)	int (WINAPI *) () (4)	[2,6]
WNDPROC	LRESULT (CALLBACK *) (HWND, UINT, WPARAM, LPARAM) (4)	LRESULT (CALLBACK *) (HWND, UINT, WPARAM, LPARAM) (4)	[2,7]

Tabelle C4: Die wichtigsten Datentypen von Win16 und Win32 im Vergleich

[1] Die Größe des Datentyps hat sich geändert..

[2] Win32-Zeiger bestehen nicht mehr aus Segment und Offset sondern nur noch aus (einem vergrößerten) Offset, daher ist keine Segmentarithmetik möglich. Es gibt keine Unterscheidung zwischen NEAR- und FAR-Zeigern.

[3] ...ebenso weitere mit DECLARE_HANDLE definierte Typen, wie z.B. HDC, HMENU etc.

[4] Zeigertyp, der unter Win 16 vom Speichermodell (small, mdium etc.) abhängig ist.

[5] Hier handelt es sich um eine Ausnahme: aus einem Win16-UINT wird ein Win32-WORD (die Größenverhältnisse ändern sich nicht).

[6] Einer der Fälle, in denen die beiden entsprechenden Header-Dateien inkonsistent sind.

[7] Gilt bis auf die Parameterliste auch für alle anderen Callback-Funktionen (wie DLGPROC etc.)

Anhang D – Quellenübersicht des XCTL-Systems

Quelldateien	Beschreibung	inkludierte Headerdateien	Use-Case
Für die Ausführung des XCTL-Systems benötigte Programmdateien			
steerng.exe	ausführbares Steuerprogramm		
steerng.ini	Initialisierungsdatei		
motors.dll	Dynamische Laufzeitbibliothek zur Ansteuerung und Verwaltung der einzelnen Antriebe		
counters.dll	Dynamische Laufzeitbibliothek zur Ansteuerung der unterschiedlichen Detektoren		
splib.dll	Dynamische Laufzeitbibliothek zum Verwalten von Intensitäts- und Diffraktometriekurven		
win488.dll	16-Bit Treiber für 6812 Motorsteuerkarte		
sphelp.hlp	Hilfebibliothek		
asa.dll	16-Bit Treiber für die Ansteuerung des BTaunPSD Detektors		
bc450rtl.dll	Dynamische Laufzeitbibliothek der Borland Entwicklungsumgebung		
standard.mak	Makroscript für die Steuerung von Basisaufgaben		
splib.dll			
dlg tpl.cpp	selbstdefinierte Templateklassen für Dialogobjekte inklusive der Methoden		Interaktion mit der Software
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
l_layer.cpp	Allgemeine Hilfsfunktionen		Interne Funktionaliät
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
m_curve.cpp	Klassen und Instanzen für die Datenhaltung		Repräsentation und Darstellung der Meßdaten
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
splib.rc	Dialogressource für About-Dialog		Windows-Ressourcen
		l_layer.h	Interne Funktionalität
motors.dll			
motors.cpp	Klassen zur Motorenansteuerung		Motorsteuerung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_motcom.h	Motorsteuerung
		m_mothw.h	Motorsteuerung
		m_layer.h	Motorsteuerung
		ieee.h	Motorsteuerung
m_layer.cpp	C-Schnittstelle für die Motorenansteuerung		Motorsteuerung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_motcom.h	Motorsteuerung

		m_mothw.h	Motorsteuerung
		m_layer.h	Motorsteuerung
dlg_tpl.cpp	selbstdefinierte Templateklassen für Dialogobjekte, inklusive der Methoden		Interaktion mit der Software
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		dgl_tpl.h	Interaktion mit der Software
motors.rc	Dialogressourcen für Motoren		Windows-Ressourcen
		rc_def.h	Windows-Ressourcen
counters.dll			
counters.cpp	Klassen für die Ansteuerung der Detektoren		Detektornutzung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_devhw.h	Detektornutzung
		c_layer.h	Detektornutzung
		m_layer.h	Motorsteuerung
c_layer.cpp	C-Schnittstelle für die Ansteuerung der Detektoren		Detektornutzung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_devhw.h	Detektornutzung
		c_layer.h	Detektornutzung
		m_layer.h	Motorsteuerung
		dfkisl.h	Detektornutzung
am9513.cpp	Klassen für die Ansteuerung der Zählerkarte Am9513A		Detektornutzung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		am9513a.h	Detektornutzung
braunpsd.cpp	Klassen zur Ansteuerung des BraunPSD		Detektornutzung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_psd.h	Detektornutzung
kisl1. c	Radicon SCSCS Treiber		Detektornutzung
		dfkisl.h	Detektornutzung
		prkmpt1.h	Detektornutzung
		radicon.h	Detektornutzung
kmpt1.c	Funktionen für Kommunikation mit der Radicon Controllerkarte		Detektornutzung
		dfkisl.h	Detektornutzung
		prkmpt1.h	Detektornutzung
dlg_tpl.cpp	selbstdefinierte Templateklassen für Dialogobjekte inklusive der Methoden		Interaktion mit der Software

		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
counters.rc	Dialogressourcen für die Dialoge „Einstellungen für SCS" und „Zählerkonfiguration"		Windows-Ressourcen
		rc_def.h	Windows-Ressourcen
develop.exe			
m_data. cpp	Modul für die Präsentation der Daten		Repräsentation und Darstellung der Meßdaten
		rc_def .h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_steerg.h	Ablaufsteuerung
		m_data.h	Repräsentation und Darstellung der Meßdaten
arscan.cpp	Modul für die Ausführung des AreaScans mit dem PSD und SLD		Diffraktometrie/Reflektrometrie
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_layer.h,	Motorsteuerung
		m_steerg.h	Ablaufsteuerung
		m_dlg.h	Interaktion mit der Software
		m_xscan.h	Diffraktometrie/Reflektrometrie
		c_layer.h	Detektornutzung
m_scan.cpp	Klassen für die Diffraktometrie und Dialoge für die Scanparameter		Diffraktometrie/Reflektrometrie
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_steerg.h	Ablaufsteuerung
		m_xscan.h	Diffraktometrie/Reflektrometrie
		c_layer.h	Detektornutzung
		m_layer.h,	Motorsteuerung
		m_devcom.h	Detektornutzung
m_steerg.cpp	Klassen für die Ablaufsteuerung (Makros)		Ablaufsteuerung
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_layer.h,	Motorsteuerung
		m_steerg.h	Ablaufsteuerung
		m_dlg.h	Interaktion mit der Software
		m_xscan.h	Diffraktometrie/Reflektrometrie
		c_layer.h	Detektornutzung
		m_layer.h	Motorsteuerung
		l_layer.h	Interne Funktionalität
m_topo. cpp	Klassen für die Topographie		Topographie
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung

		m_layer.h,	Motorsteuerung
		m_steerg.h	Ablaufsteuerung
		m_topo.h	Topographie
m_device.cpp	Klasse bzw. Methoden für das Zählerfenster, Klasse: TCounterWindow		Interaktion mit der Software
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
m_main.cpp	Hauptfunktion für das Programm, Initialisierung der Bibliotheken, Nachrichtenschleife, Handler)		Interne Funktionalität
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_steerg.h	Ablaufsteuerung
		m_topo.h,	Topographie
		m_xscan.h	Diffraktometrie/Reflektrometrie
		m_dlg.h	Interaktion mit der Software
		help_def.h	Online-Hilfe
		st_layer.h	Diffraktometrie/Reflektrometrie
		l_layer.h	Interne Funktionalität
		m_layer.h	Motorsteuerung
		c_layer.h	Detektornutzung
m_dlg.cpp	Implementation verschiedener Dialogklassen, unter anderem Manuelle Justage		Interaktion mit der Software
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
		m_devcom.h	Detektornutzung
		m_steerg.h	Ablaufsteuerung
		m_layer.h	Motorsteuerung
		c_layer.h	Detektornutzung
		m_dlg.h	Interaktion mit der Software
dlg tpl.cpp	selbstdefinierte Templateklassen für Dialogobjekte, inklusive der Methoden		Interaktion mit der Software
		rc_def.h	Windows-Ressourcen
		comhead.h	Nicht zugeordnet
main.rc	Ressourcen für die Dialoge Steuerprogrammes		Windows-Ressourcen
		rc_def.h	Windows-Ressourcen

Tabelle D5: Quellenübersicht des XCTL-Systems

Anhang E – Hardwareausstattung der Arbeitsplätze für das XCTL-System

PC	Nutzer/Projekt.....	Prozessor.....	Memory	HDisk	Controller	Motor-. Karte
x-ray7	PC RTK7 (Mhling)	i 486 DX 33MHz	4MB	152MB	SCSI	?
x-ray8	PC RTK3(Mini2)	i 486 DX 33MHz	8MB	203MB	AT	?
x-ray6	PC RTK4 (Frank)	i Pentium 66MHz	32MB	2100MB	SCSI NCR (on Bord)	?
Grenoble	PC RTK Grenoble	i 486 DX2 66MHz	8MB	406MB	EIDE/VL DC 2000 PromiseTech	?
HRM1	.	486 DX2	20MB	1000MB	.	1x C832 1x C812
HRM2	.	486 DX	8MB	200 MB	.	1x C832 1x C812
ESRF	.	486 DX	8MB	??	.	1x C832 1x C812
Meßplatz AFM(Atomkraftmikroskop): Pentium 100 (kein MMX) 32 MB RAM 1000 MB IDE HD S3 Grafik on Board Motorsteuerungskarte 1x C832 XY-Tisch mit Mikedriveantrieben Betriebssystem IBM PCDOS 7.0 + Windows 3.11						
Offline Meßplatz (Digitalisierung von Bilddaten): mit Kugelumlaufspindeln und Antrieb über Faulhaber DC-Motorgetriebekombination Motorsteuerungskarte 1x C832 AMD K5 PR100 2GB IDE HD 48MB RAM PCI-Framegrabber Matrox Pulsar (monochromer Universalframegrabber mit MGA Grafik) mit RS422 Digitalinterface Betriebssystem Win NT4.0SP4 (notwendig wegen 32-bit Framegrabbersoftware)						

Tabelle E6: Hardwareausstattung der Arbeitsplätze für das XCTL-System

Anhang F – Daten der Quelltextanalyse

F 1 Maßkomponente ’Software-Umgebung’

F 1.1 Ermittlung der Maßkomponente ’Betriebssystem-Aufrufe’

zum Maß *BS* , siehe Kapitel 4.2.1.1

$BSR_1 = memcpy(),\ Anz.BSR_1 = 1$
$BSR_2 = hmemcpy(),\ Anz.BSR_2 = 2$

M_arscan.cpp 2500 memcpy((HPSTR) buf, (HPSTR) daten, cnt); TAreaScan::LoadOldData(int)

$KOMP_1 = \text{TAreaScan},\ Anz.BSA = 1$

M_data.cpp 865 hmemcpy((HPBYTE)&(hpData[addr]),(LPBYTE)lpBuf,Screen.dx); TBitmapSource::GenerateAngleSpaceBitmap(int,int,int,unsigned

M_data.cpp 1324 hmemcpy(hpNewData+szInfo,hpData,GlobalSize(hDIBData)); TBitmapSource::RenderDIB()

$KOMP_2 = \text{TBitmapSource},\ Anz.BSA = 2$

Anzahl - GROUP	
GROUP	Ergebnis
DOS	21
Gesamtergebnis	21

$BSM_1 = Anz.BSR_1 + ... + Anz.BSR_n = 3$

$BSM_2 = ((KOMP_1, Anz.BSA), ..., (KOMP_n, Anz.BSA)) =$
$((\text{TAreaScan}, 1), (\text{TBitmapSource}, 2))$

$$BSM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BS\text{-}Anpassungen}{Anzahl\ aller\ Komponenten} = \frac{2}{81} = 0,025$$

$BS = (BSM_1*, BSM_2, BSM_3) = (3, ((\text{TAreaScan}, 1), (\text{TBitmapSource}, 2)), 0,025)$

F 1.2 Ermittlung der Maßkomponente ’Bibliotheksroutinen-Aufrufe’

zum Maß *BIB* , siehe Kapitel 4.2.1.1

$BSR_1 = DialogProc(),\ Anz.BSR_1 = 4$
$BSR_2 = FreeModule(),\ Anz.BSR_2 = 2$
$BSR_3 = FreeProcInstance(),\ Anz.BSR_3 = 1$
$BSR_4 = FreeResource(),\ Anz.BSR_4 = 1$
$BSR_5 = GetFocus(),\ Anz.BSR_5 = 112$
$BSR_6 = GetModuleUsage(),\ Anz.BSR_6 = 1$
$BSR_6 = GlobalCompact(),\ Anz.BSR_6 = 3$
$BSR_7 = MakeProcInstance(),\ Anz.BSR_7 = 2$

$BSR_8 = MoveTo(),\ Anz.BSR_8 = 12$
$BSR_9 = SetCapture(),\ Anz.BSR_9 = 1$
$BSR_{10} = SetFocus(),\ Anz.BSR_{10} = 41$
$BSR_{11} = SetMessageQueue(),\ Anz.BSR_{11} = 1$
$BSR_{12} = UnlockData(),\ Anz.BSR_{12} = 4$
$BSR_{13} = WM\_COMMAND,\ Anz.BSR_{13} = 162$
$BSR_{14} = WM\_HSCROLL,\ Anz.BSR_{14} = 2$
$BSR_{15} = WM\_MENUSELECT,\ Anz.BSR_{15} = 1$
$BSR_{16} = WM\_QUIT,\ Anz.BSR_{16} = 1$
$BSR_{17} = WM\_VSCROLL,\ Anz.BSR_{17} = 2$

L_layer.cpp 119 MessageBox(GetFocus(),AuthorAddress,"Adresse",MBINFO); TAbout::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int)

L_layer.cpp 115 MessageBox(GetFocus(),WorkAddress,"Adresse",MBINFO); TAbout::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int)

$KOMP_1 = \text{TAbout},\ Anz.BIBA = 2$

M_steerg.cpp 3164 SetFocus(GetDlgItem(GetHandle(),cm_Start)); TAdjustmentExecute::Dlg_OnInit(HWND___*,HWND___*,long)

M_steerg.cpp 3129 FORWARD_WM_COMMAND(GetHandle(),cm_ParamSet,0,0,PostMessage); TAdjustmentExecute::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_2 = \text{TAdjustmentExecute},\ Anz.BIBA = 2$

M_steerg.cpp 3348 MessageBox(GetFocus(),"ContinuousScan wurde ausgeführt !","Justagen",MBINFO); TAdjustmentWindow::CounterSetRequest(long)

$KOMP_3 = \text{TAdjustmentWindow},\ Anz.BIBA = 1$

Am9513.cpp 274 MessageBox(GetFocus(),"Fehler beim Zerlegen !","Am9513a",MBINFO); TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne

Am9513.cpp 317 MessageBox(GetFocus(),buf,"Device-Fehler Am9513a",MBINFO); TAm9513a::SelectChip(unsigned_char)

$KOMP_4 = \text{TAm9513a},\ Anz.BIBA = 2$

M_dlg.cpp 767 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::GetBarEgde(int)

M_dlg.cpp 396 SetFocus(BarHandle); TAngleControl::Dlg_OnTimer(HWND___*,unsigned_int)

M_dlg.cpp 757 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 745 FORWARD_WM_COMMAND(hwnd,cm_MoveButton,0,0,SendMessage); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 742 mMoveToDistance(mGetValue(MaxDistance)); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 740 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 732 FORWARD_WM_COMMAND(hwnd,cm_MoveButton,0,0,SendMessage); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 729 mMoveToDistance(mGetValue(MinDistance)); TAngleControl::Dlg_OnHScrollBar(HWND___*,HWND___*,unsigned_i

M_dlg.cpp 695 SetFocus(BarHandle); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 680 FORWARD_WM_COMMAND(hwnd,cm_MoveButton,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 679 FORWARD_WM_COMMAND(hwnd,cm_MoveButton,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 637 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 636 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 628 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 627 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 619 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 618 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 610 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 609 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 597 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,PostMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 570 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,SendMessage); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 543 SendMessage(hwnd,WM_COMMAND,cm_ParamSet,0); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 542 SendMessage(hwnd,WM_COMMAND,cm_MotorInit,0); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 526 SetFocus(GetDlgItem(hwnd,id_SpeedValue)); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp 519 SetFocus(BarHandle); TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

M_dlg.cpp	518	FORWARD_WM_COMMAND(hwnd,cm_MotorInit,0,0,SendMessage);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	509	SetFocus(GetDlgItem(hwnd,id_AngleWidth));	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	504	SetFocus(BarHandle);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	503	SendMessage(hwnd,WM_COMMAND,cm_MotorInit,0);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	458	SetFocus(BarHandle);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	448	SetFocus(BarHandle);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_
M_dlg.cpp	434	SetFocus(BarHandle);	TAngleControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_

$KOMP_5 = \text{TAngleControl},\ Anz.BIBA = 32$

M_arscan.cpp	3770	MessageBox(GetFocus(),buf,"Meldung",MBINFO);	TAquisition::CanClose()
M_arscan.cpp	3761	if(IDYES == MessageBox(GetFocus(),buf,"Meldung",MBASK))	TAquisition::CanClose()

$KOMP_6 = \text{TAquisition},\ Anz.BIBA = 2$

M_arscan.cpp	344	switch(GET_WM_COMMAND_ID(wParam,lParam)){	TAreaScan::~TAreaScan()
M_arscan.cpp	495	MessageBox(GetFocus(),"Eingabe wird verworfen !","Fehler",MBINFO);	TAreaScan::InitializeDlg(unsigned_int,long)
M_arscan.cpp	650	MessageBox(GetFocus(),buf,"Meldung",MBINFO);	TAreaScan::InitializeDlg(unsigned_int,long)
M_arscan.cpp	671	MessageBox(GetFocus(),buf,"Daten-Erhebung",MBINFO);	TAreaScan::InitializeDlg(unsigned_int,long)
M_arscan.cpp	1062	if(IDNO == MessageBox(GetFocus(),msg,HeadLine,MBASK))	TAreaScan::InitializeTask(unsigned_int,long)
M_arscan.cpp	1080	MessageBox(GetFocus(),buf,szMsgFailure,MBSTOP);	TAreaScan::InitializeTask(unsigned_int,long)
M_arscan.cpp	2396	MessageBox(GetFocus(),szMsgLine014,szMsgFailure,MBINFO);	TAreaScan::LoadMeasurementInfo(int)
M_arscan.cpp	2457	MessageBox(GetFocus(),szMsgLine014,szMsgFailure,MBINFO);	TAreaScan::LoadOldData(int)
M_arscan.cpp	1905	MessageBox(GetFocus(),szMsgLine014,szMsgFailure,MBINFO);	TAreaScan::SaveFile(int)
M_arscan.cpp	1792	MessageBox(GetFocus(),buf,szMsgFailure, MB_OK | MB_ICONSTOP);	TAreaScan::SaveMeasurementInfo(int)
M_arscan.cpp	877	MessageBox(GetFocus(),"Kein gültiger Darstellungstyp !","Meldung",MBINFO);	TAreaScan::SetRanges()
M_arscan.cpp	929	MessageBox(GetFocus(),"Kein gültiger Darstellungstyp !","Meldung",MBINFO);	TAreaScan::SetRanges()
M_arscan.cpp	1557	MessageBox(GetFocus(),"Kein Psd verfügbar.",szMessage,MBINFO);	TAreaScan::ShowSensorContinuous(int)

$KOMP_7 = \text{TAreaScan},\ Anz.BIBA = 13$

M_steerg.cpp	641	MessageBox(GetFocus(),"Start Scan failed.","Steering",MBSTOP);	TAreaScanCmd::TAreaScanCmd(TCmdTag)

$KOMP_8 = \text{TAreaScanCmd},\ Anz.BIBA = 1$

M_arscan.cpp	187	MessageBox(GetFocus(),szMsgLine016,szMessage,MBINFO);	TAreaScanParameters::TAreaScanParameters()

$KOMP_9 = \text{TAreaScanParameters},\ Anz.BIBA = 1$

M_data.cpp	1240	MessageBox(GetFocus(),"Cannot create memory object (1)","Message",MBINFO);	TBitmapSource::FormatDBaseToBitmapSource()
M_data.cpp	1283	MessageBox(GetFocus(),"Unknown OutputType","Message",MBINFO);	TBitmapSource::FormatDBaseToBitmapSource()
M_data.cpp	783	MessageBox(GetFocus(),"Cannot create memory object (2)","Message",MBINFO);	TBitmapSource::GenerateAngleSpaceBitmap(int,int,int,unsigned
M_data.cpp	1035	MessageBox(GetFocus(),"Cannot create memory object (2)","Message",MBINFO);	TBitmapSource::GenerateRLBitmap()
M_data.cpp	600	MessageBox(GetFocus(),"Screen (1)","Message",MBINFO);	TBitmapSource::SetScreen(HDC___*,TScreen_&)

$KOMP_{10} = \text{TBitmapSource},\ Anz.BIBA = 5$

BraunPSD.cpp	242	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::~TBraun_Psd()
BraunPSD.cpp	472	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::PsdInit()
BraunPSD.cpp	353	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	453	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	505	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::PsdStart()
BraunPSD.cpp	576	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::PsdStop()
BraunPSD.cpp	281	MessageBox(GetFocus(),"Kein gültiger Typ.","TPsdDataType",MBINFO);	TBraun_Psd::SetDataType(TPsdDataType)
BraunPSD.cpp	544	MessageBox(GetFocus(),TBraunError[nErrorCode][0],HeadLine,MBINFO);	TBraun_Psd::SetEnergyRange(unsigned_int,unsigned_int)

$KOMP_{11} = \text{TBraun\_Psd},\ Anz.BIBA = 8$

Motors.cpp	2631	MessageBox(GetFocus(),msg,"TimeOut MS",MBFAILURE);	TC_812::GetStatus()
Motors.cpp	2636	MessageBox(GetFocus(),msg,"Fehler",MBFAILURE);	TC_812::GetStatus()

$KOMP_{12} = \text{TC\_812},\ Anz.BIBA = 2$

Motors.cpp	2829	MessageBox(GetFocus(),buf,"Failure",MBSTOP);	TC_812GPIB::CheckBoardOk()
Motors.cpp	2998	MessageBox(GetFocus(),buf,"Fehler MS",MBSTOP);	TC_812GPIB::CheckBoardOk()
Motors.cpp	3023	MessageBox(GetFocus(),buf,"Fehler MS",MBFAILURE);	TC_812GPIB::CheckBoardOk()
Motors.cpp	2907	MessageBox(GetFocus(),(LPSTR)pString,"Command Failure",MBSTOP);	TC_812GPIB::ExecuteCmd(char_*)
Motors.cpp	2909	MessageBox(GetFocus(),(LPSTR)pString,"Command Failure",MBSTOP);	TC_812GPIB::ExecuteCmd(char_*)
Motors.cpp	3156	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	TC_812GPIB::ExecuteCmd(char_*)
Motors.cpp	3175	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	TC_812GPIB::ExecuteCmd(char_*)
Motors.cpp	3198	MessageBox(GetFocus(),"TimeOut read","C-812",MBINFO);	TC_812GPIB::ExecuteCmd(char_*)
Motors.cpp	2805	MessageBox(GetFocus(),"Can't find win488.dll","Motor Library",MBSTOP);	TC_812GPIB::Initialize()
Motors.cpp	2810	MessageBox(GetFocus(),"No GPIB-Controller present !","Failure",MBSTOP);	TC_812GPIB::Initialize()
Motors.cpp	2783	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	TC_812GPIB::TC_812GPIB()

$KOMP_{13} = \text{TC\_812GPIB},\ Anz.BIBA = 8$

Motors.cpp	3471	PostMessage(GetFrameHandle(),WM_COMMAND,cm_SetWatchIndicator,(LP…	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u
Motors.cpp	3487	PostMessage(GetFrameHandle(),WM_COMMAND,cm_LimitHitOccure,(LPARAM)…	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u

$KOMP_{14} = \text{TC\_832},\ Anz.BIBA = 2$

M_arscan.cpp	2902	MessageBox(GetFocus(),"Vorgang bitte wiederholen !",szMsgFailure,MBINFO);	TCalibrate::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int
M_arscan.cpp	2941	MessageBox(GetFocus(),"Vorgang bitte wiederholen !","Fehler",MBINFO);	TCalibrate::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int
Motors.cpp	610	if(IDYES != MessageBox(GetFocus(),buf,"Meldung",MBASK))	TCalibrate::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int
Motors.cpp	652	FORWARD_WM_COMMAND(hwnd,IDCANCEL,hwndCtl,0,PostMessage);	TCalibrate::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int
Motors.cpp	731	FORWARD_WM_COMMAND(hwnd,IDCANCEL,hwndCtl,0,PostMessage);	TCalibrate::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_int

$KOMP_{15} = \text{TCalibrate},\ Anz.BIBA = 5$

M_arscan.cpp	2857	SetFocus(hBar);	TCalibratePsd::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_arscan.cpp	2968	SetFocus(hBar);	TCalibratePsd::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_arscan.cpp	2983	SetFocus(hBar);	TCalibratePsd::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_arscan.cpp	3015	mMoveToDistance(atof(buf));	TCalibratePsd::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_arscan.cpp	3021	SetFocus(hBar);	TCalibratePsd::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
Motors.cpp	538	FORWARD_WM_COMMAND(hwnd,cm_CarryOnZero,hwndCtl,0,PostMe…;	TCalibratePsd::Dlg_OnInit(HWND___*,HWND___*,long)
Motors.cpp	566	FORWARD_WM_COMMAND(hwnd,cm_IndexArrived,0,idx,SendMessage);	TCalibratePsd::Dlg_OnTimer(HWND___*,unsigned_int)
Motors.cpp	576	FORWARD_WM_COMMAND(hwnd,cm_DistanceSet,0,idx,SendMessage);	TCalibratePsd::Dlg_OnTimer(HWND___*,unsigned_int)

$KOMP_{16} = \text{TCalibratePsd},\ Anz.BIBA = 8$

M_steerg.cpp	652	MessageBox(GetFocus(),"Kein Scan-Fenster offen","Fehler",MBSTOP);	TCmd::DoAction()
M_steerg.cpp	128	if(mlMoveToDistance(mid,dDistance))	TCmd::StartMove(const_int,double)

$KOMP_{17} = \text{TCmd},\ Anz.BIBA = 2$

Counters.cpp	2052	FORWARD_WM_COMMAND(GetFrameHandle(),cm_CallExecute…;	TCommonDevParam::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Counters.cpp	2070	FORWARD_WM_COMMAND(GetHandle(),cm_ActivateChanges,0,0,…	TCommonDevParam::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Counters.cpp	2010	SetFocus(GetDlgItem(hwnd,id_Level));	TCommonDevParam::Dlg_OnInit(HWND___*,HWND___*,long)
Counters.cpp	2015	FORWARD_WM_COMMAND(hwnd,cm_Initialization,hwndCtl,0,Sen;	TCommonDevParam::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{18} = \text{TCommonDevParam},\ Anz.BIBA = 4$

M_device.cpp	332	SetFocus(GetDlgItem(GetHandle(),IDOK));	TCounterShowParam::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{19} = \text{TCounterShowParam},\ Anz.BIBA = 1$

M_device.cpp	82	MessageBox(GetFocus(),(LPSTR)buf,"Fehler",MB_OK);	TCounterWindow::CanOpen()
M_device.cpp	202	void TCounterWindow::SetFocus(void)	TCounterWindow::SetFocus()

$KOMP_{20} = \text{TCounterWindow},\ Anz.BIBA = 2$

M_curve.cpp	143	GlobalCompact(MaxMemIdx * sizeof(CPoint));	TCurve::TCurve(int)
M_curve.cpp	151	MessageBox(GetFocus(),"Kein Speicher TCurve","Fehler",MB_OK | MB_ICONSTOP);	TCurve::TCurve(int)

$KOMP_{21} = \text{TCurve},\ Anz.BIBA = 2$

Motors.cpp	2158	MessageBox(GetFocus(),buf,szMsgFailure,MBSTOP);	TDC_Drive::Initialize()

$KOMP_{22} = \text{TDC\_Drive},\ Anz.BIBA = 1$

Counters.cpp	352	PostMessage(hEventControlWnd,WM_COMMAND,cm_SteeringReady,0l);	TDevice::EventHandler(unsigned_int,unsigned_int,unsigned_lon
Counters.cpp	356	PostMessage(hEventControlWnd,WM_COMMAND,cm_CounterSet,nCalledEvents);	TDevice::EventHandler(unsigned_int,unsigned_int,unsigned_lon
Counters.cpp	370	MessageBox(GetFocus(),"Meßzeit wurde zu klein gewählt !","InitializeEvent",MBINFO);	TDevice::InitializeEvent(HWND___*,int)
Counters.cpp	1044	PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId());	TDevice::PollDevice()
Counters.cpp	1047	PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId());	TDevice::PollDevice()
Counters.cpp	484	PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId());	TDevice::PollDevice()
Counters.cpp	487	PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId());	TDevice::PollDevice()
Counters.cpp	383	FORWARD_WM_COMMAND(hDisplayWnd,cm_CounterSet,0,0,SendMessage);	TDevice::UpdateDisplay()

$KOMP_{23} = \text{TDevice},\ Anz.BIBA = 8$

Counters.cpp	171	MessageBox(GetFocus(),buf,"Device-Fehler",MBSTOP);	TDList::InitializeModule()

$KOMP_{24} = \text{TDList},\ Anz.BIBA = 1$

M_steerg.cpp	2835	void TEditWindow::SetFocus()	TEditWindow::SetFocus()
M_steerg.cpp	2837	SetFocus();	TEditWindow::SetFocus()
M_steerg.h	583	void SetFocus(void);	TEditWindow::SetFocus()

$KOMP_{25} = \text{TEditWindow},\ Anz.BIBA = 3$

Counters.cpp	700	MessageBox(GetFocus(),"Encoder fehlt","Fehler",MBINFO);	TEncoder::PollDevice()
Counters.cpp	705	PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId());	TEncoder::PollDevice()
Counters.cpp	708	PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId());	TEncoder::PollDevice()

$KOMP_{26} = \text{TEncoder},\ Anz.BIBA = 3$

M_dlg.cpp	182	SetFocus(GetDlgItem(hwnd,id_CommandLine));	TExecuteCmd::Dlg_OnInit(HWND___*,HWND___*,long)
M_dlg.cpp	213	SetFocus(GetDlgItem(hwnd,id_CommandLine));	TExecuteCmd::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{27} = \text{TExecuteCmd},\ Anz.BIBA = 2$

Counters.cpp	624	if(IDYES == MessageBox(GetFocus(),"Time = 0.0 \n Neustart ?","Generic Device",MBASK))	TGenericDevice::PollDevice()
Counters.cpp	633	PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId());	TGenericDevice::PollDevice()
Counters.cpp	637	PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId());	TGenericDevice::PollDevice()

$KOMP_{28} = \text{TGenericDevice},\ Anz.BIBA = 3$

M_steerg.cpp	2634	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	TMacroExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_steerg.cpp	2668	PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0);	TMacroExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_steerg.cpp	2669	PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0);	TMacroExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_steerg.cpp	2681	PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0);	TMacroExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_steerg.cpp	2696	PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0);	TMacroExecute::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{29} = \text{TMacroExecute},\ Anz.BIBA = 5$

M_main.cpp	150	FreeResource(hVersion);	TMain::GetVersion(char_*)
M_main.cpp	125	if(msg.message == WM_QUIT)	TMain::MessageLoop()
M_main.cpp	1616	MessageBox(GetFocus(),strMessage13,strMessage,MBINFO);	TMain::TellMessage(int)

$KOMP_{30} = \text{TMain},\ Anz.BIBA = 3$

Comclass.h	79	virtual void SetFocus(void);	TMDIWindow::SetFocus()
M_devcom.h	220	void SetFocus(void);	TMDIWindow::SetFocus()
M_main.cpp	1862	MessageBox(GetFocus(),strMessage02,strFailure,MB_OK);	TMDIWindow::CanOpen()
M_main.cpp	1916	void TMDIWindow::SetFocus(void)	TMDIWindow::SetFocus()

$KOMP_{31} = \text{TMDIWindow},\ Anz.BIBA = 4$

Motors.cpp 308 if(IDYES == MessageBox(GetFocus(),szMsgLine001,szMsgFailure,MBASK)) TMList::InitializeModule()

Motors.cpp 441 FORWARD_WM_COMMAND(GetHandle(),cm_InquireExeption,0,0,PostMessage); TMList::MP()

$KOMP_{32} = \text{TMList},\ Anz.BIBA = 2$

Dlg_tpl.cpp 67 FreeProcInstance(dialogFunc); TModalDlg::~TModalDlg()

Dlg_tpl.cpp 16 BOOL CALLBACK _export DialogProc(HWND hDlg,UINT msg,WPARAM wParam,LPARAM lParam) TModalDlg::TModalDlg(char_*)

Dlg_tpl.cpp 61 dialogFunc = MakeProcInstance((FARPROC)DialogProc,hInstance); TModalDlg::TModalDlg(char_*)

$KOMP_{33} = \text{TModalDlg},\ Anz.BIBA = 3$

Dlg_tpl.cpp 141 dialogFunc = (DLGPROC)MakeProcInstance((FARPROC)ModelessProc,GetMainInstance()); TModelessDlg::TModelessDlg(char_*)

Dlg_tpl.h 59 virtual void Dlg_OnSetFocus (HWND,HWND) {}; TModelessDlg::Dlg_OnSetFocus(HWND___*,HWND___*)

$KOMP_{34} = \text{TModelessDlg},\ Anz.BIBA = 2$

Motors.cpp 1870 MessageBox(GetFocus(),buffer,szMsgFailure,MBINFO); TMotor::Translate(long_&,double)

$KOMP_{35} = \text{TMotor},\ Anz.BIBA = 1$

Motors.cpp 1051 SetFocus(GetDlgItem(GetHandle(),id_Velocity)); TMotorParam::TMotorParam()

$KOMP_{36} = \text{TMotorParam},\ Anz.BIBA = 1$

M_data.cpp 1507 SetCapture(hWndChild); TPlotData::lButtonDown(unsigned_int,long)

M_data.cpp 2337 MessageBox(GetFocus(),msg,"Fehler",MB_OK | MB_ICONHAND); TPlotData::SaveSecondCurve()

$KOMP_{37} = \text{TPlotData},\ Anz.BIBA = 2$

Motors.cpp 923 FORWARD_WM_COMMAND(hwnd,cm_SetupDlgItem,hwndCtl,0,SendMessage); TPosControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_in

Motors.cpp 936 FORWARD_WM_COMMAND(hwnd,cm_MotorInit,hwndCtl,0,PostMessage); TPosControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_in

Motors.cpp 958 FORWARD_WM_COMMAND(hwnd,cm_MoveButton,hwndCtl,0,PostMessage); TPosControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_in

Motors.cpp 959 SetFocus(GetDlgItem(GetHandle(),id_Bar)); TPosControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_in

Motors.cpp 966 FORWARD_WM_COMMAND(hwnd,cm_SetupDlgItem,hwndCtl,0,PostMessage); TPosControl::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned_in

Motors.cpp 875 SetFocus(BarHandle); TPosControl::Dlg_OnInit(HWND___*,HWND___*,long)

Motors.cpp 895 FORWARD_WM_COMMAND(hwnd,cm_SetupDlgItem,0,0,SendMessage); TPosControl::Dlg_OnTimer(HWND___*,unsigned_int)

$KOMP_{38} = \text{TPosControl},\ Anz.BIBA = 7$

Counters.cpp 785 MessageBox(GetFocus(),"Psd ist nicht kalibriert !","Meldung", TPsd::Initialize()

$KOMP_{39} = \text{TPsd},\ Anz.BIBA = 1$

M_dlg.cpp 70 FORWARD_WM_COMMAND(hwnd,IDCANCEL,hwndCtl,0,PostMessage); TPsdRemoteSync::Dlg_OnInit(HWND___*,HWND___*,long)

M_dlg.cpp 79 SetFocus(GetDlgItem(hwnd,id_AddedChannels)); TPsdRemoteSync::Dlg_OnInit(HWND___*,HWND___*,long)

M_dlg.cpp 118 MessageBox(GetFocus(),"Zur Zeit nicht unterstützt !","Message",MBINFO); TPsdRemoteSync::Dlg_OnInit(HWND___*,HWND___*,long)

M_dlg.cpp 128 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,PostMessage); TPsdRemoteSync::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{40} = \text{TPsdRemoteSync},\ Anz.BIBA = 4$

Counters.cpp 1835 PostMessage(hEventControlWnd,WM_COMMAND,cm_SteeringReady,0l); TRadicon::EventHandler(unsigned_int,unsigned_int,unsigned_lo

Counters.cpp 1893 MessageBox(GetFocus(),buf,"Geräte-Fehler",MBINFO); TRadicon::FailureOccured(int,int)

Counters.cpp 1915 MessageBox(GetFocus(),"Kein scs !","Fehler",MBSTOP); TRadicon::FailureOccured(int,int)

Counters.cpp 1851 MessageBox(GetFocus(),"Meßzeit wurde zu klein gewählt !","InitializeEvent",MBINFO); TRadicon::InitializeEvent(HWND___*,int)

Counters.cpp 1781 PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId()) TRadicon::PollDevice()

Counters.cpp 1785 PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId()); TRadicon::PollDevice()

$KOMP_{41} = \text{TRadicon},\ Anz.BIBA = 6$

M_scan.cpp 521 SendMessage(hWndChild,WM_COMMAND,cm_UpdateFile,0); TScan::CounterSetRequest(long)

M_scan.cpp 555 SendMessage(hWndChild,WM_COMMAND,cm_UpdateFile,0); TScan::CounterSetRequest(long)

M_scan.cpp 359 if(IDNO == MessageBox(GetFocus(),buf,HeadLine,MBASK)) TScan::InitializeTask(unsigned_int,long)

M_scan.cpp 403 if(IDNO == MessageBox(GetFocus(),buf,HeadLine,MBASK)) TScan::InitializeTask(unsigned_int,long)

M_scan.cpp 440 if(IDNO == MessageBox(GetFocus(),buf,HeadLine,MBASK)) TScan::InitializeTask(unsigned_int,long)
M_scan.cpp 1001 FORWARD_WM_COMMAND(GetHandle(),cm_MoveButton,0,0,PostMessage); TScan::LoadOldData(int)
M_scan.cpp 1016 MessageBox(GetFocus()," File wurde nicht geöffnet !", "Fehler", MBINFO); TScan::LoadOldData(int)
M_scan.cpp 1029 MessageBox(GetFocus(),"Header ist nicht korrekt","Meldung",MBINFO); TScan::LoadOldData(int)
M_scan.cpp 1031 MessageBox(GetFocus(),"Incorrect Header.","Message",MBINFO); TScan::LoadOldData(int)
M_scan.cpp 913 MessageBox(GetFocus(),"Header ist nicht korrekt","Meldung",MBINFO); TScan::SaveFile(int)
M_scan.cpp 915 MessageBox(GetFocus(),"Incorrect Header.","Message",MBINFO); TScan::SaveFile(int)
M_scan.cpp 656 MessageBox(GetFocus(),"ContinuousScan wurde ausgeführt !","Justagen",MBINFO); TScan::SteeringReady(long)
M_scan.cpp 969 MessageBox(GetFocus()," File existiert nicht !", "Fehler", MB_OK); TScan::UpdateFile()

$KOMP_{42} = \text{TScan},\ Anz.BIBA = 13$

M_steerg.cpp 212 MessageBox(GetFocus(),buf,"Information",MBINFO); TScanCmd::TScanCmd(TCmdTag)
M_steerg.cpp 216 MessageBox(GetFocus(),buf,"Information",MBINFO); TScanCmd::TScanCmd(TCmdTag)
M_steerg.cpp 220 MessageBox(GetFocus(),buf,"Information",MBINFO); TScanCmd::TScanCmd(TCmdTag)
M_steerg.cpp 539 MessageBox(GetFocus(),"Kein Scan-Fenster offen","Fehler",MBSTOP); TScanCmd::TScanCmd(TCmdTag)
M_steerg.cpp 601 MessageBox(GetFocus(),"Kein AreaScan-Fenster offen","Fehler",MBSTOP); TScanCmd::TScanCmd(TCmdTag)

$KOMP_{43} = \text{TScanCmd},\ Anz.BIBA = 5$

Counters.cpp 1923 SetFocus(GetDlgItem(GetHandle(),id_AngleRange)); TScsParameters::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{44} = \text{TScsParameters},\ Anz.BIBA = 1$

M_steerg.cpp 2973 SetFocus(GetDlgItem(TModalDlg::GetHandle(), id_Level)); TSetAdjustmentParam::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{45} = \text{TSetAdjustmentParam},\ Anz.BIBA = 1$

M_arscan.cpp 3656 PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0l); TSetupAreaScan::CanClose()
M_arscan.cpp 3403 MessageBox(GetFocus(),szMsgLine015,szMsgFailure,MBSTOP); TSetupAreaScan::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_arscan.cpp 3466 MessageBox(GetFocus(),szMsgLine015,szMsgFailure,MBSTOP); TSetupAreaScan::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned

$KOMP_{46} = \text{TSetupAreaScan},\ Anz.BIBA = 3$

M_scan.cpp 1516 SetFocus(GetDlgItem(GetHandle(),id_Actor)); TSetupContinuousScan::TSetupContinuousScan(TScan_*)
M_scan.cpp 1535 FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage); TSetupContinuousScan::TSetupContinuousScan(TScan_*)

$KOMP_{47} = \text{TSetupContinuousScan},\ Anz.BIBA = 2$

M_steerg.cpp 151 MessageBox(GetFocus(),"Unbekannter Schritt","Meldung",MBINFO); TSetupScanCmd::TSetupScanCmd(TCmdTag)

$KOMP_{48} = \text{TSetupScanCmd},\ Anz.BIBA = 1$

M_scan.cpp 1465 PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0); TSetupStepScan::CanClose()
M_scan.cpp 1559 SetFocus(GetDlgItem(GetHandle(),cm_InitializeScan)); TSetupStepScan::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_scan.cpp 1577 PostMessage(GetFrameHandle(),WM_COMMAND,cm_InitializeScan,0l); TSetupStepScan::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_scan.cpp 1619 PostMessage(GetHandle(),WM_COMMAND,(WPARAM)cm_ParamSet,0); TSetupStepScan::Dlg_OnCommand(HWND___*,int,HWND___*,unsigned
M_scan.cpp 1192 SetFocus(GetDlgItem(GetHandle(),id_ChooseMotor)); TSetupStepScan::Dlg_OnInit(HWND___*,HWND___*,long)
M_scan.cpp 1234 SetFocus(GetDlgItem(GetHandle(),IDOK)); TSetupStepScan::Dlg_OnInit(HWND___*,HWND___*,long)

$KOMP_{49} = \text{TSetupStepScan},\ Anz.BIBA = 6$

M_steerg.cpp 701 MessageBox(GetFocus(),"Start Scan failed.","Steering",MBSTOP); TShowValueCmd::TShowValueCmd(TCmdTag)
M_steerg.cpp 707 MessageBox(GetFocus(),"Start Scan","Fehler",MBSTOP); TShowValueCmd::TShowValueCmd(TCmdTag)
M_steerg.cpp 1795 MessageBox(GetFocus(),"Das laufende Macro erst unterbrechen !","Meldung",MBINFO); TShowValueCmd::TShowValueCmd(TCmdTag)

$KOMP_{50} = \text{TShowValueCmd},\ Anz.BIBA = 3$

M_steerg.cpp 1812 MessageBox(GetFocus(),"Makroausführung ist fehlgeschlagen !","Steuerung",...; TSteering::ExecuteNextCmd()
M_steerg.cpp 2431 MessageBox(GetFocus(), "Makrodatei fehlt!", fn, MBSTOP); TSteering::LoadMacro(char_*,char_*)
M_steerg.cpp 2472 MessageBox(GetFocus(), "Kommandoliste unvollstaendig!", makname, MBSTOP); TSteering::LoadMacro(char_*,char_*)
M_steerg.cpp 2490 MessageBox(GetFocus(), "Makro ohne Kommandos!", makname, MBSTOP); TSteering::LoadMacro(char_*,char_*)
M_steerg.cpp 2526 MessageBox(GetFocus(), "Makro fehlt!", makname, MBSTOP); TSteering::LoadMacro(char_*,char_*)

M_steerg.cpp	1859	PostMessage(hHostWindow,WM_COMMAND,cm_SteeringReady,0l);	TSteering::NotifyCmdReady()
M_steerg.cpp	1870	PostMessage(hHostWindow,WM_COMMAND,cm_SteeringReady,0l);	TSteering::NotifyMacroReady()
M_steerg.cpp	2336	MessageBox(GetFocus(),buf,"Macro einlesen",MBINFO);	TSteering::ParsingCmd(TCmdTag_&,char_*,char_*,char_*,char_*)
M_steerg.cpp	1377	MessageBox(GetFocus(),"Kein AreaScan-Fenster offen","Fehler",MBSTOP);	TSteering::StartMacroExecution(TMacroTag_*,HWND___*)

$KOMP_{51} = \text{TSteering},\ Anz.BIBA = 9$

Counters.cpp	1207	PostMessage(hControlWnd,WM_COMMAND,wpJumpTo,GetId());	TStoe_Psd::PollDevice()
Counters.cpp	1210	PostMessage(hDisplayWnd,WM_COMMAND,wpJumpTo,GetId());	TStoe_Psd::PollDevice()

$KOMP_{52} = \text{TStoe\_Psd},\ Anz.BIBA = 2$

M_topo.cpp	265	SendMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	TTopographyExecute::Dlg_OnCommand(HWND___*,int,HWND___*,unsi
M_topo.cpp	403	SendMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	TTopographyExecute::Dlg_OnCommand(HWND___*,int,HWND___*,unsi
M_topo.cpp	421	FORWARD_WM_COMMAND(GetHandle(),cm_SetupPosition,0,0,PostMessage);	TTopographyExecute::Dlg_OnCommand(HWND___*,int,HWND___*,unsi
M_topo.cpp	479	SetFocus(GetDlgItem(GetHandle(),cm_SwitchControl));	TTopographyExecute::Dlg_OnCommand(HWND___*,int,HWND___*,unsi
M_topo.cpp	570	FORWARD_WM_COMMAND(GetHandle(),cm_SwitchControl,0,0,PostMessage);	TTopographyExecute::Dlg_OnCommand(HWND___*,int,HWND___*,unsi
M_topo.cpp	163	SendMessage(GetHandle(),WM_COMMAND,cm_Initialize,0);	TTopographyExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_topo.cpp	164	SendMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	TTopographyExecute::Dlg_OnInit(HWND___*,HWND___*,long)
M_topo.cpp	217	FORWARD_WM_COMMAND(hwnd,cm_SwitchControl,0,0,PostMessage);	TTopographyExecute::Dlg_OnTimer(HWND___*,unsigned_int)

$KOMP_{53} = \text{TTopographyExecute},\ Anz.BIBA = 8$

C_layer.cpp	259	PostMessage(hControlWnd,WM_COMMAND,cm_CounterSet,77);	InquireIntensity_A913(unsigned_int,unsigned_int,unsigned_lon
C_layer.cpp	195	PostMessage(hControlWnd,WM_COMMAND,cm_CounterSet,77);	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	213	PostMessage(hControlWnd,WM_COMMAND,cm_SteeringReady,77);	InquireIntensity_TDC(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	227	PostMessage(hControlWnd,WM_COMMAND,cm_CounterSet,77);	InquireIntensity_TDC(unsigned_int,unsigned_int,unsigned_long
Counters.cpp	78	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
Counters.cpp	114	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
Counters.cpp	187	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
Counters.cpp	337	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
Counters.cpp	440	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
Counters.cpp	1610	PostMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	
Counters.cpp	1838	PostMessage(hEventControlWnd,WM_COMMAND,cm_CounterSet,nCalledEvents);	
Counters.cpp	1985	PostMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	
Dlg_tpl.cpp	23	HANDLE_MSG(hDlg, WM_COMMAND, TheDialog->Dlg_OnCommand);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	24	HANDLE_MSG(hDlg, WM_HSCROLL, TheDialog->Dlg_OnHScrollBar);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	25	HANDLE_MSG(hDlg, WM_VSCROLL, TheDialog->Dlg_OnVScrollBar);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	61	dialogFunc = MakeProcInstance((FARPROC)DialogProc,hInstance);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	115	HANDLE_MSG(hDlg, WM_COMMAND, TheModeless->Dlg_OnComma	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	116	HANDLE_MSG(hDlg, WM_HSCROLL, TheModeless->Dlg_OnHScrollB	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	117	HANDLE_MSG(hDlg, WM_VSCROLL, TheModeless->Dlg_OnVScrollB	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	120	HANDLE_MSG(hDlg, WM_SETFOCUS, TheModeless->Dlg_OnSetF	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	409	GlobalCompact(MaxMemIdx * (3*sizeof(float) + sizeof(BOOL)));	
Dlg_tpl.h	7	BOOL CALLBACK _export DialogProc (HWND,UINT,WPARAM,LPARAM);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.h	16	friend BOOL CALLBACK DialogProc(HWND,UINT,WPARAM,LPARAM);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
L_layer.cpp	45	UnlockData(0);	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
M_arscan.cpp	151	FORWARD_WM_COMMAND(GetHandle(),cm_SetPositionRange,0,0,SendMessage);	
M_arscan.cpp	237	SetFocus(GetDlgItem(hwnd,IDOK));	
M_arscan.cpp	336	FORWARD_WM_COMMAND(hwnd,cm_MotorInit,hwndCtl,0,SendMessage);	
M_arscan.cpp	433	PostMessage(hWndChild,WM_COMMAND,cm_DataAquisition,0l);	
M_arscan.cpp	571	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_arscan.cpp	596	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_arscan.cpp	662	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_arscan.cpp	726	mMoveToDistance(mGetValue(Distance) - mGetValue(Width));	
M_arscan.cpp	739	mMoveToDistance(mGetValue(Distance) + mGetValue(Width));	
M_arscan.cpp	874	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_arscan.cpp	935	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	

M_arscan.cpp	946	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_arscan.cpp	1050	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_arscan.cpp	1206	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_arscan.cpp	3017	FORWARD_WM_COMMAND(hwnd,cm_MoveButton,hwndCtl,0,SendMessage);	
M_arscan.cpp	3051	FORWARD_WM_COMMAND(hwnd,cm_MoveButton,hwndCtl,0,SendMessage);	
M_arscan.cpp	3059	FORWARD_WM_COMMAND(hwnd,cm_MoveButton,hwndCtl,0,SendMessage);	
M_arscan.cpp	3064	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,PostMessage);	
M_arscan.cpp	3337	FORWARD_WM_COMMAND(hwnd,cm_SetupParameters,hwndCtl,0,SendMessage);	
M_arscan.cpp	3412	FORWARD_WM_COMMAND(hwnd,cm_InquireRelevantData,hwndCtl,0,SendMessage);	
M_arscan.cpp	3507	FORWARD_WM_COMMAND(GetHandle(),cm_SetupParameters,0,0,SendMessage);	
M_arscan.cpp	3514	FORWARD_WM_COMMAND(GetHandle(),cm_SetupParameters,0,0,PostMessage);	
M_curve.cpp	236	if(lSize > GlobalCompact(lSize))	
M_data.cpp	2473	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	2486	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	2496	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	2512	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	2525	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	2535	MessageBox(GetFocus(),Msg,"Message",MBINFO);	
M_data.cpp	3407	SetFocus(GetDlgItem(hwnd,IDOK));	
M_dlg.cpp	1233	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_dlg.cpp	1346	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_dlg.cpp	2782	FORWARD_WM_COMMAND(hwnd,cm_MotorInit,hwndCtl,0,SendMessage);	
M_dlg.cpp	3050	mMoveToDistance(mGetValue(Distance) - mGetValue(Width));	
M_dlg.cpp	3058	mMoveToDistance(mGetValue(Distance) + mGetValue(Width));	
M_dlg.cpp	3141	FORWARD_WM_COMMAND(GetHandle(),cm_ParamSet,0,0,PostMessage);	
M_layer.cpp	509	MFT.MoveToDistance = (TMoveToDistance) mMoveToDistance;	DllEntryPoint(void_*,unsigned_long,void_*)
M_layer.cpp	519	FreeModule(hGPIBModule);	DllEntryPoint(void_*,unsigned_long,void_*)
M_layer.cpp	270	lpMList->MP(mid)->MoveToAngle(distance);	mlMoveToDistance(int,double)
M_layer.cpp	239	PostMessage(GetFrameHandle(),WM_COMMAND,cm_CallExecuteScan,0l);	mlOptimizingDlg()
M_layer.cpp	352	BOOL _export WINAPI mMoveToDistance(double angle)	mMoveToDistance(double)
M_layer.cpp	364	lpMList->MP()->MoveToAngle(angle);	mMoveToDistance(double)
M_layer.cpp	120	PostMessage(GetFrameHandle(),WM_COMMAND,cm_MoveScanReady,8	mSavePosition(unsigned_int,unsigned_int,unsigned_long,unsign
M_layer.cpp	126	PostMessage(GetFrameHandle(),WM_COMMAND,cm_MoveScanReady,8	mSavePosition(unsigned_int,unsigned_int,unsigned_long,unsign
M_layer.cpp	539	FreeModule(hGPIBModule);	WEP(int)
M_main.cpp	547	Cmd = GET_WM_COMMAND_ID(wParam,lParam);	DoCommandsChild(TMDIWindow_*,HWND___*,unsigned_int,long)
M_main.cpp	601	MessageBox(GetFocus(),"Trial to stop all drives !","Message",MBINFO);	DoCommandsChild(TMDIWindow_*,HWND___*,unsigned_int,long)
M_main.cpp	249	switch(GET_WM_COMMAND_ID(wParam,lParam))	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	279	MessageBox(GetFocus(),"Bibliothek counters.dll nicht geladen !","Fehler",...	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	315	PostMessage(hwnd,WM_COMMAND,cm_AngleControl,0);	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	325	PostMessage(hwnd,WM_COMMAND,cm_CallExecuteMacro,0);	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	501	SendMessage(hChildWnd,WM_COMMAND,wParam,lParam);	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	1008	nDevice = GET_WM_COMMAND_ID(wParam,lParam) - DeviceTimerIdStart;	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1033	case WM_COMMAND:	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1119	case WM_MENUSELECT:	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1124	if((GET_WM_COMMAND_ID(wParam,lParam) == MSGF_DIALOGBOX)	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1127	PostMessage((HWND)LOWORD(lParam),WM_COMMAND,cm_Index,0);	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1129	if((GET_WM_COMMAND_ID(wParam,lParam) == MSGF_MENU) &&	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1140	Main.DrawStatus(hWindow,(ATOM)lParam,GET_WM_COMMAND_ID(wPar…	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1056, 1084	if(2 < GetModuleUsage(hMDLL))	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	910	if(0xFFFF == (WORD)GET_WM_COMMAND_HWND(wParam,lParam))	MenuSelect(HWND___*,unsigned_int,long)
M_main.cpp	916	if(MF_POPUP & (UINT)GET_WM_COMMAND_HWND(wParam,lParam))	MenuSelect(HWND___*,unsigned_int,long)
M_main.cpp	918	Main.CurrentPopup = (HMENU)GET_WM_COMMAND_ID(wParam,lParam);	MenuSelect(HWND___*,unsigned_int,long)
M_main.cpp	934	Main.CurrentID = GET_WM_COMMAND_ID(wParam,lParam);	MenuSelect(HWND___*,unsigned_int,long)
M_main.cpp	167	MessageBox(GetFocus(),"Kein Konfigurationsfile !","Fehler",MBSTOP);	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	169	MessageBox(GetFocus(),"Configuration file not founded !","Failure",MBS	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	199	while(!SetMessageQueue(cnt--));	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)

M_main.cpp	215	MessageBox(GetFocus(),"Keine Timer verfügbar !","System-Fehler",MBST	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	217	MessageBox(GetFocus(),"No Timer available !","System-Failure",MBSTOP);	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	221	FORWARD_WM_COMMAND(Main.hWndFrame,cm_Initialization,0,0,Se	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	1189	case WM_COMMAND:	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1195	switch(GET_WM_COMMAND_ID(wParam,lParam)) {	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1197	window->TimerRequest(GET_WM_COMMAND_ID(wParam,lParam));	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1213	if(1 < (int)GET_WM_COMMAND_HWND(wParam,lParam))	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1243	window->SetFocus();	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	24	MessageBox(GetFocus(),"Bibliothek motors.dll nicht geladen !","Fehler",MBSTOP);	
M_scan.cpp	1922	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_scan.cpp	2048	PostMessage(GetHandle(),WM_COMMAND,cm_ParamSet,0);	
M_steerg.cpp	1593	FORWARD_WM_COMMAND(GetHandle(),cm_ParamSet,0,0,PostMessage);	
M_steerg.cpp	2014	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_steerg.cpp	3489	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,PostMessage);	
M_topo.cpp	1106	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_topo.cpp	1119	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_topo.cpp	2055	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_topo.cpp	2076	FORWARD_WM_COMMAND(GetHandle(),cm_ActivateChanges,0,0,PostMessage);	
M_topo.cpp	2092	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_topo.cpp	2102	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
M_topo.cpp	2639	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
M_topo.cpp	3538	FORWARD_WM_COMMAND(GetHandle(),cm_SetupParameters,0,0,SendMessage);	
Motors.cpp	2667	MessageBox(GetFocus(),msg,"TimeOut MS",MBFAILURE);	
Motors.cpp	2672	MessageBox(GetFocus(),msg,"Fehler",MBFAILURE);	
Motors.cpp	2708	MessageBox(GetFocus(),msg,"TimeOut MS",MBFAILURE);	
Motors.cpp	2713	MessageBox(GetFocus(),msg,"Fehler",MBFAILURE);	
Motors.cpp	2974	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
Motors.cpp	3217	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
Motors.cpp	3244	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,0,0,SendMessage);	
Motors.cpp	3336	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
Motors.cpp	3414	FORWARD_WM_COMMAND(hwnd,cm_ParamSet,hwndCtl,0,SendMessage);	
St_layer.cpp	38	MessageBox(GetFocus(),"Die Antriebe sind nicht Ok !","Fehler",MBINFO);	
St_layer.cpp	261	MessageBox(GetFocus(),"Bibliothek motors.dll nicht geladen !","Fehler",MBSTOP);	
St_layer.cpp	145, 147	BOOL WINAPI _export UnlockDataBase(void)	
St_layer.h	16	BOOL WINAPI UnlockDataBase	
St_layer.h	34	typedef BOOL (WINAPI *TUnlockDataBase)	
Testdev.cpp	33	PostMessage(hControlWnd, WM_COMMAND, wpJumpTo, GetId());	
Testdev.cpp	36	PostMessage(hDisplayWnd, WM_COMMAND, wpJumpTo, GetId());	

$KOMP_{54} = \mathrm{Tx},\ Anz.BIBA = 129$

Anzahl – GROUP	
GROUP	Ergebnis
GDI	71
MBE	46
DOK	4
Gesamtergebnis	121

$$BIBM_1 = Anz.BIBR_1 + ... + Anz.BIBR_n = 353$$

$$BIBM_2 = ((KOMP_1, Anz.BIBA),...,(KOMP_n, Anz.BIBA)) =$$

((*TAbout* , 2),
(*TAdjustmentExecute* , 2),
(*TAdjustmentWindow* , 1),
(*TAm9513a* , 2),
(*TAngleControl* , 32),

(*TAquisition* , 2),
(*TAreaScan* , 13),
(*TAreaScanCmd* , 1),
(*TAreaScanParameters* , 1),
(*TBitmapSource* , 5),
(*TBraun_Psd* , 8),
(*TC_812* , 2),
(*TC_812GPIB* , 11),
(*TC_832* , 2),
(*TCalibrate* , 5),
(*TCalibratePsd* , 8),
(*TCmd* , 2),
(*TCommonDevParam* , 4),
(*TCounterShowParam* , 1),
(*TCounterWindow* , 2),
(*TCurve* , 2),
(*TDC_Drive* , 1),
(*TDevice* , 8),
(*TDList* , 1),
(*TEditWindow* , 3),
(*TEncoder* , 3),
(*TExecuteCmd* , 2),
(*TGenericDevice* , 3),
(*TMacroExecute* , 5),
(*TMain* , 3),
(*TMDIWindow* , 4),
(*TMList* , 2),
(*TModalDlg* , 3),
(*TModelessDlg* , 2),
(*TMotor* , 1),
(*TMotorParam* , 1),
(*TPlotData* , 2),
(*TPosControl* , 7),
(*TPsd* , 1),
(*TPsdRemoteSync* , 4),
(*TRadicon* , 6),
(*TScan* , 13),
(*TScanCmd* , 5),
(*TScsParameters* , 1),
(*TSetAdjustmentParam* , 1),
(*TSetupAreaScan* , 3),
(*TSetupContinuousScan* , 2),
(*TSetupScanCmd* , 1),
(*TSetupStepScan* , 6),
(*TShowValueCmd* , 3),
(*TSteering* , 9),
(*TStoe_Psd* , 2),
(*TTopographyExecute* , 8),
(*Tx* , 129))

$$BIBM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ BIB\text{-}Anpassungen}{Anzahl\ aller\ Komponenten} = \frac{54}{81} = 0,667$$

$$BIB = (BIBM_1{*}, BIBM_2, BIBM_3) = (353, BIBM_2, 0,667)$$

F 1.3 Ermittlung der Maßkomponente 'Software-Umgebung'

zum Maß SU , siehe Kapitel 4.2.1.1

$$PAN_{SU} = BSM_1 + BIBM_1 + UTM_1 = 3 + 353 + 0 = 356$$

$$\mathrm{KOMP}_{SU} = BSM_2 + BIBM_2 + UTM_2 =$$

((*TAbout* , 2),
(*TAdjustmentExecute* , 2),
(*TAdjustmentWindow* , 1),
(*TAm9513a* , 2),
(*TAngleControl* , 32),
(*TAquisition* , 2),
(*TAreaScan* , 14),
(*TAreaScanCmd* , 1),
(*TAreaScanParameters* , 1),
(*TBitmapSource* , 7),
(*TBraun_Psd* , 8),
(*TC_812* , 2),
(*TC_812GPIB* , 11),
(*TC_832* , 2),
(*TCalibrate* , 5),
(*TCalibratePsd* , 8),
(*TCmd* , 2),
(*TCommonDevParam* , 4),
(*TCounterShowParam* , 1),
(*TCounterWindow* , 2),
(*TCurve* , 2),
(*TDC_Drive* , 1),
(*TDevice* , 8),
(*TDList* , 1),
(*TEditWindow* , 3),
(*TEncoder* , 3),
(*TExecuteCmd* , 2),
(*TGenericDevice* , 3),
(*TMacroExecute* , 5),
(*TMain* , 3),
(*TMDIWindow* , 4),
(*TMList* , 2),
(*TModalDlg* , 3),
(*TModelessDlg* , 2),
(*TMotor* , 1),
(*TMotorParam* , 1),
(*TPlotData* , 2),
(*TPosControl* , 7),
(*TPsd* , 1),
(*TPsdRemoteSync* , 4),
(*TRadicon* , 6),
(*TScan* , 13),
(*TScanCmd* , 5),
(*TScsParameters* , 1),
(*TSetAdjustmentParam* , 1),

(*TSetupAreaScan* , 3),
(*TSetupContinuousScan* , 2),
(*TSetupScanCmd* , 1),
(*TSetupStepScan* , 6),
(*TShowValueCmd* , 3),
(*TSteering* , 9),
(*TStoe_Psd* , 2),
(*TTopographyExecute* , 8),
(*Tx* , 129))

$$\mathrm{VKG}_{\mathrm{SU}} = BSM_3 + BIBM_3 + UTM_3 = 0,025 + 0,667 + 0 = 0,691$$

$$SU = (PAN_{SU}, KOMP_{SU}, VKG_{SU}) = (356,\ KOMP_{SU},\ 0,691)$$

F 2 Maßkomponente ’Hardware-Umgebung’

F 2.1 Ermittlung der Maßkomponente ’Maschinenarchitektur’

zum Maß *MA* , siehe Kapitel 4.2.1.2

$MA_1 = LibMain(),\ Anz.MA_1 = 4$
$MA_2 = GetModuleHandle(),\ Anz.MA_2 = 2$
$MA_3 = GetModuleUsage(),\ Anz.MA_3 = 2$
$MA_4 = (short),\ Anz.MA_4 = 3$
$MA_5 = (WORD),\ Anz.MA_5 = 102$
$MA_6 = cbClsExtra,\ Anz.MA_6 = 2$
$MA_7 = cbWndExtra,\ Anz.MA_7 = 2$
$MA_8 = export,\ Anz.MA_8 = 131$
$MA_9 = FAR,\ Anz.MA_9 = 19$
$MA_{10} = HIWORD,\ Anz.MA_{10} = 10$
$MA_{11} = huge,\ Anz.MA_{11} = 3$
$MA_{12} = LONG,\ Anz.MA_{12} = 26$
$MA_{13} = LOWORD,\ Anz.MA_{13} = 14$
$MA_{15} = pascal,\ Anz.MA_{15} = 4$
$MA_{16} = WEP,\ Anz.MA_{16} = 4$
$MA_{17} = Wnd\,\mathrm{Pr}oc(),\ Anz.MA_{17} = 9$
$MA_{18} = WORD,\ Anz.MA_{18} = 224$
$MA_{19} = \mathrm{WORD\ FAR\ \ PASCAL},\ Anz.MA_{19} = 2$

M_steerg.cpp 3353 void TAdjustmentWindow::rButtonDown(LONG lParam) TAdjustmentWindow::rButtonDown(long)
M_steerg.cpp 3357 pp.x = LOWORD(lParam); TAdjustmentWindow::rButtonDown(long)
M_steerg.cpp 3358 pp.y = HIWORD(lParam); TAdjustmentWindow::rButtonDown(long)

$KOMP_1 = \mathrm{TAdjustmentWindow},\ Anz.MAA = 3$

Am9513.cpp 33 WORD wValue; TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne

Am9513.cpp	72	DWORD dwData = 0;	TAm9513a::IOCTL(TIOCCmd)
Am9513.cpp	76	int TAm9513a::IOCTL(TIOCCmd cmd,DWORD& param)	TAm9513a::IOCTL(TIOCCmd)
Am9513.cpp	78	WORD regH,regL;	TAm9513a::IOCTL(TIOCCmd)
Am9513.cpp	79	WORD wValue;	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	85	param = (DWORD)((double)param * fTimeCorrection);	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	91	param = (DWORD)wLowTicks * wHighTicks;	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	92	param = (DWORD)((double)param / fTimeCorrection);	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	97	param = (DWORD)wLowCounts * wHighCounts;	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	158	param = (DWORD)(wHighTicks - regH) * (DWORD)wLowTicks;	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	159	param += (DWORD)(wLowTicks - regL);	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	160	param = (DWORD)((double)param / fTimeCorrection);	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	175	param = (DWORD)(wHighCounts - regH) * (DWORD)wLowCounts;	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	176	param += (DWORD)(wLowCounts - regL);	TAm9513a::IOCTL(TIOCCmd,unsigned_long_&)
Am9513.cpp	181	WriteData((WORD)param);	TAm9513a::IOCTL(TIOCCmd)
Am9513.cpp	218	BOOL TAm9513a::SplitNumber(DWORD value,WORD& low,WORD& high)	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	220	LONG f1,f2,f,fail;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	227	low = (WORD)value;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	263	high = (WORD)f1;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	264	low = (WORD)f2;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	270	high = (WORD)f2;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	271	low = (WORD)f1;	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	278	DWORD TAm9513a::GetTicksPerSecond(void)	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.cpp	299	WORD value;	TAm9513a::SelectChip(unsigned_char)
Am9513.cpp	395	WORD TAm9513a::ReadStatus(void)	TAm9513a::ChooseDataPtr
Am9513.cpp	430	void TAm9513a::WriteData(WORD daten)	TAm9513a::WriteData(unsigned_short)
Am9513.cpp	446	test = (BYTE)(((WORD)daten >> 8) & 0x00FF);	TAm9513a::WriteData(unsigned_short)
Am9513.cpp	461	WORD TAm9513a::ReadData(void)	TAm9513a::ReadData()
Am9513.cpp	463	WORD value;	TAm9513a::ReadData()
Am9513.cpp	468	value = (WORD)inp(addr);	TAm9513a::ReadData()
Am9513.cpp	480	value += (WORD)((WORD)inp(addr) << 8);	TAm9513a::ReadData()
Am9513.h	109	int IOCTL(TIOCCmd,DWORD&);	TAm9513a::Init()
Am9513.h	113	DWORD GetTicksPerSecond(void);	TAm9513a::GetTicksPerSecond()
Am9513.h	121	void WriteData(WORD);	TAm9513a::WriteData(unsigned_short)
Am9513.h	122	WORD ReadData(void);	TAm9513a::ReadData()
Am9513.h	123	WORD ReadStatus(void);	TAm9513a::ReadStatus()
Am9513.h	134	BOOL SplitNumber(DWORD,WORD&,WORD&);	TAm9513a::SplitNumber(unsigned_long,unsigned_short_&,unsigne
Am9513.h	136	WORD wLowTicks,wHighTicks;	TAm9513a
Am9513.h	137	WORD wLowCounts,wHighCounts;	TAm9513a

$KOMP_2 = \text{TAm9513a},\ Anz.MAA = 39$

M_arscan.cpp	384	void TAreaScan::rButtonDown(LONG lParam)	TAreaScan::rButtonDown(long)
M_arscan.cpp	2420	DWORD dwReadCnt;	TAreaScan::LoadOldData(int)
M_data.cpp	440	nColors = min((WORD)lpBMPInfo->bmiHeader.biClrUsed,(WORD)pPal->palNumEntries);	TAreaScan::TAreaScan()
M_data.cpp	902	Screen.dy = ::RLdy = max((WORD)0,min((WORD)700,::RLdy));	TAreaScan::~TAreaScan(

$KOMP_3 = \text{TAreaScan},\ Anz.MAA = 4$

M_arscan.cpp	266	AppendMenu(TheMenu,MF_POPUP,(WORD)hMAdd,"&Scan");	TBitmapSource::FormatDBaseToBitmapSource()
M_data.cpp	230	DWORD memsize;	TBitmapSource::ProcessBitmapFile(int)
M_data.cpp	290	DWORD TBitmapSource::GetImageSize(void)	TBitmapSource::ProcessBitmapFile(int)
M_data.cpp	292	DWORD nBits;	TBitmapSource::GetImageSize()
M_data.cpp	301	nBits *= (DWORD)lpBMPInfoHdr->biHeight;	TBitmapSource::GetImageSize()
M_data.cpp	403	nColors = (short)lpBMPInfo->bmiHeader.biClrUsed;	TBitmapSource::CreatePaletteFromDIB()
M_data.cpp	758	void TBitmapSource::GenerateAngleSpaceBitmap(BOOL bStretch,int nAddedPoints, int nEnlargedLines,DWORD nBits)	TBitmapSource::GenerateAngleSpaceBitmap(int,int,int,unsigned
M_data.cpp	768	DWORD addr;	TBitmapSource::GenerateAngleSpaceBitmap(int,int,int,unsigned
M_data.cpp	1014	DWORD addr,MaxAddr = GetImageSize();	TBitmapSource::GenerateRLBitmap()

M_data.cpp	1132	nColor = (short)GetColor(fIntens);	TBitmapSource::GenerateRLBitmap()
M_data.cpp	1142	addr = rlx + (DWORD)rly * Screen.dx;	TBitmapSource::GenerateRLBitmap()
M_data.cpp	1191	DWORD nBits,dwSize;	TBitmapSource::FormatDBaseToBitmapSource()
M_data.cpp	1261	Screen.dy = ::RLdy = max((WORD)0,min((WORD)700,::RLdy));	TBitmapSource::FillBMInfoFromPalette(tagLOGPALETTE_*)
M_data.cpp	1274	nEnlargedLines = min((WORD)6,max((WORD)1,(WORD)(440/nScanNumber)));	TBitmapSource::DrawMeasurementArea(HDC___*)
M_data.h	81	DWORD GetImageSize(void);	TBitmapSource::GetImageSize()
M_data.h	146	void GenerateAngleSpaceBitmap(BOOL,int,int,DWORD);	TBitmapSource::GenerateAngleSpaceBitmap(int,int,int,unsigned
M_scan.cpp	154	AppendMenu(TheMenu,MF_POPUP,(WORD)hM2,"Sca&n");	TBitmapSource::FormatDBaseToBitmapSource()

$KOMP_4 = \text{TBitmapSource},\ Anz.MAA = 17$

BraunPSD.cpp	125	void (WINAPI *lpfnSetPort) (WORD,WORD);	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	126	BYTE (WINAPI *lpfnGetPort) (WORD);	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	127	void (WINAPI *lpfnSetTimeout) (DWORD);	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	128	int (WINAPI *lpfnGetData) (WORD,WORD,WORD,LPLONG,WORD,long&);	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	128	int (WINAPI *lpfnGetData) (WORD,WORD,WORD,LPLONG,WORD,long&);	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	141	(FARPROC)lpfnSetPort = GetProcAddress(AsaDllInstance,"SetPort");	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	142	(FARPROC)lpfnGetPort = GetProcAddress(AsaDllInstance,"GetPort");	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	143	(FARPROC)lpfnSetTimeout = GetProcAddress(AsaDllInstance,"SetTimeout");	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	144	(FARPROC)lpfnGetData = GetProcAddress(AsaDllInstance,"GetData");	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	166	transform.lWert[0] = (WORD)uEnergyHigh;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	174	transform.lWert[0] = (WORD)uEnergyLow;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	184	transform.lWert[0] = (WORD)uMuxTimeDet1;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	192	befehl[11][3] = (WORD)bRatemeter;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	193	echo[11][4] = (WORD)bRatemeter;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	197	befehl[15][3] = (WORD)bRealLifeTime;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	198	echo[15][4] = (WORD)bRealLifeTime;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	204	befehl[10][3] = (WORD)nDeathTime;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	205	echo[10][4] = (WORD)nDeathTime;	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	218	hReadBuf = GlobalAlloc(GHND,GetBufferSize() * sizeof(DWORD));	TBraun_Psd::TBraun_Psd()
BraunPSD.cpp	294	LPDWORD lpdwCountBuf;	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	295	LPDWORD lpdwReadBuf;	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	334	lpdwReadBuf = (LPDWORD)GlobalLock(hReadBuf);	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	337	Words = (UINT) lpfnGetData((WORD)...	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	348	memcpy((LPLONG)&Header,(LPLONG)lpdwReadBuf,16 * sizeof(long));	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	371	lpdwCountBuf = (LPDWORD)GlobalLock(hCountBuf);	TBraun_Psd::PsdReadOut(THowReadOutPsd)
BraunPSD.cpp	524	transform.lWert[0] = (WORD)uEnergyHigh;	TBraun_Psd::SetEnergyRange(unsigned_int,unsigned_int)
BraunPSD.cpp	531	transform.lWert[0] = (WORD)uEnergyLow;	TBraun_Psd::SetEnergyRange(unsigned_int,unsigned_int)
BraunPSD.cpp	596	lpfnSetPort((WORD) nBaseAddr,zgrbefehl[i]);	TBraun_Psd::BuildOperation(unsigned_char_*,unsigned_char_*,i
BraunPSD.cpp	603	EchoRead[i+2] = lpfnGetPort((WORD)(nBaseAddr+2));	TBraun_Psd::BuildOperation(unsigned_char_*,unsigned_char_*,i
BraunPSD.cpp	604	EchoRead[i+1] = lpfnGetPort((WORD)(nBaseAddr+3));	TBraun_Psd::BuildOperation(unsigned_char_*,unsigned_char_*,i
BraunPSD.cpp	629	lpfnSetPort((WORD)(nBaseAddr+1),0x00);	TBraun_Psd::ResetDelayTime()
BraunPSD.cpp	646	while((lpfnGetPort((WORD) (nBaseAddr+1))) > Bit7)	TBraun_Psd::Look_till_BaseAddr1(int)
BraunPSD.cpp	667	lpfnSetPort((WORD)nBaseAddr,0x3A);	TBraun_Psd::SynchronHexFile(unsigned_char_&,unsigned_char_&)
BraunPSD.cpp	669	lpfnGetPort((WORD)(nBaseAddr+3));	TBraun_Psd::SynchronHexFile(unsigned_char_&,unsigned_char_&)
BraunPSD.cpp	670	R_String = lpfnGetPort((WORD)(nBaseAddr+2));	TBraun_Psd::SynchronHexFile(unsigned_char_&,unsigned_char_&)
BraunPSD.cpp	758	lpfnSetPort((WORD)nBaseAddr,zeichen);	TBraun_Psd::LoadHexFile()
BraunPSD.cpp	765	lpfnGetPort((WORD)(nBaseAddr+3));	TBraun_Psd::LoadHexFile()
BraunPSD.cpp	766	ReadString[i+1] = lpfnGetPort((WORD)(nBaseAddr+2));	TBraun_Psd::LoadHexFile()
BraunPSD.cpp	336-338	Words = (UINT) lpfnGetData((WORD)...	TBraun_Psd::PsdReadOut(THowReadOutPsd)

$KOMP_5 = \text{TBraun\_Psd},\ Anz.MAA = 39$

m_mothw.h	98	WORD GetStatus (void);	TC_812::GetStatus()
Motors.cpp	2292	wStaticGain = (WORD)GetPrivateProfileInt(Ident,"Gain",100,GetCFile());	TC_812::TC_812()
Motors.cpp	2293	wDynamicGain = (WORD)GetPrivateProfileInt(Ident,"DynamicGain",37,GetCFile());	TC_812::TC_812()
Motors.cpp	2296	wTorque = (WORD)GetPrivateProfileInt(Ident,"Torque",110,GetCFile());	TC_812::TC_812()

Motors.cpp	2370	WORD torque;	TC_812::GetMoment()
Motors.cpp	2495	BOOL TC_812::SetDynamicGain(WORD dygain)	TC_812::SetDynamicGain(unsigned_short)
Motors.cpp	2499	dygain = min(dygain,(WORD)255);	TC_812::SetDynamicGain(unsigned_short)
Motors.cpp	2500	dygain = max(dygain,(WORD)1);	TC_812::SetDynamicGain(unsigned_short)
Motors.cpp	2510	WORD TC_812::GetDynamicGain(void)	TC_812::GetDynamicGain()
Motors.cpp	2516	BOOL TC_812::SetTorque(WORD torque)	TC_812::SetTorque(unsigned_short)
Motors.cpp	2520	BOOL TC_812::SetTorque(WORD torque)	TC_812::SetTorque(unsigned_short)
Motors.cpp	2528	WORD TC_812::GetTorque(void)	TC_812::GetTorque()
Motors.cpp	2530	wTorque = min(torque,(WORD)127);	TC_812::GetTorque()
Motors.cpp	2535	BOOL TC_812::SetVelocity(DWORD velocity)	TC_812::SetVelocity(unsigned_long
Motors.cpp	2550	DWORD TC_812::GetVelocity(void)	TC_812::GetVelocity()
Motors.cpp	2557	BOOL TC_812::SetAcceleration(DWORD acceleration)	TC_812::SetAcceleration(unsigned_long)
Motors.cpp	2575	DWORD TC_812::GetAcceleration(void)	TC_812::GetAcceleration()
Motors.cpp	2581	BOOL TC_812::SetLimit(DWORD limit)	TC_812::SetLimit(unsigned_long)
Motors.cpp	2606	WORD TC_812::GetStatus(void)	TC_812::GetStatus()
Motors.cpp	2611	WORD _status;	TC_812::GetStatus()

$KOMP_6 = \text{TC\_812},\ Anz.MAA = 20$

Motors.cpp	2757	WORD hostio,hostaddr;	TC_812GPIB::TC_812GPIB()
Motors.cpp	2767	hostio = (WORD)GetPrivateProfileInt(Ident,"IOAddr",0x200,GetCFile());	TC_812GPIB::TC_812GPIB()
Motors.cpp	2768	hostaddr = (WORD)GetPrivateProfileInt(Ident,"GPIBAddr",1,GetCFile());	TC_812GPIB::TC_812GPIB()
Motors.cpp	2791	wGPIBAddr = (WORD)GetPrivateProfileInt(Ident,"GPIBAddr",15,GetCFile());	TC_812GPIB::TC_812GPIB()
Motors.cpp	2834	WORD length;	TC_812GPIB::CheckBoardOk()
Motors.cpp	2770-2785	(FARPROC)gBoardPresent= GetProcAddress(hGPIBModule,"IEEE488_BOARD_PRESENT");	TC_812GPIB::TC_812GPIB()

$KOMP_7 = \text{TC\_812GPIB},\ Anz.MAA = 6$

m_mothw.h	156	friend long Drive628c(BYTE,WORD,long,WORD,WORD);	TC_832::Drive628(unsigned_char,unsigned_short,long)
m_mothw.h	161	void CALLBACK (*lpfnLimitWatch) (UINT,UINT,DWORD,DWORD,DWORD);	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u
m_mothw.h	194	static void CALLBACK LimitWatch (UINT,UINT,DWORD,DWORD,DWORD);	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u
m_mothw.h	105-115	BOOL SetLimit (DWORD);	TC_832::SetLimit(unsigned_long)
m_mothw.h	175-188	BOOL SetLimit (DWORD);	TC_832::SetLimit(unsigned_long)
m_mothw.h	31-47	virtual BOOL SetVelocity(DWORD) {return TRUE;};	TC_832::SetVelocity(unsigned_long)
Motors.cpp	3415	WORD ctr = 0x0400; // Stop smoothly --> programming manual (page 29)	TC_832::StopDrive(int)
Motors.cpp	3450	void CALLBACK TC_832::LimitWatch(UINT /*IDEvent*/,UINT,DWORD /*dwUse…	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u
Motors.cpp	3607	BOOL TC_832::SetVelocity(DWORD velocity)	TC_832::SetVelocity(unsigned_long)
Motors.cpp	3616	DWORD TC_832::GetVelocity(void)	TC_832::GetVelocity()
Motors.cpp	3623	BOOL TC_832::SetAcceleration(DWORD acceleration)	TC_832::SetAcceleration(unsigned_long)
Motors.cpp	3632	DWORD TC_832::GetAcceleration(void)	TC_832::GetAcceleration()
Motors.cpp	3638	BOOL TC_832::SetLimit(DWORD limit)	TC_832::SetLimit(unsigned_long)
Motors.cpp	3645	WORD TC_832::GetStatus(void)	TC_832::GetStatus()
Motors.cpp	3647	return (WORD)Drive628(RDSTAT,0,0);	TC_832::GetStatus()
Motors.cpp	3723	long TC_832::Drive628(BYTE cmd,WORD ctrl_word,long param)	TC_832::Drive628(unsigned_char,unsigned_short,long)
Motors.cpp	3817	int GetWord(WORD base,WORD regaddr)	TC_832::Drive628(unsigned_char,unsigned_short,long)
Motors.cpp	3308-3311	wKI = (WORD)GetPrivateProfileInt(Ident,"IntegralGain",10,GetCFile());	TC_832::TC_832()

$KOMP_8 = TC\_832,\ Anz.MAA = 18$

M_steerg.cpp	43	static DWORD dwExposureCounts;	TChooseDeviceCmd::TChooseDeviceCmd(TCmdTag)

$KOMP_9 = \text{TChooseDeviceCmd},\ Anz.MAA = 1$

M_device.cpp	363	value_s = (short)atoi(buf);	TCounterShowParam::CanClose()

$KOMP_{10} = \text{TCounterShowParam},\ Anz.MAA = 1$

M_device.cpp	180	void TCounterWindow::rButtonDown(LONG lParam)	TCounterWindow::rButtonDown(long)

$KOMP_{11} = \text{TCounterWindow},\ Anz.MAA = 1$

Motors.cpp	2128	wDeathBand = (WORD)GetPrivateProfileInt(Ident,"DeathBand",1,GetCFile());	TDC_Drive::TDC_Drive()
Motors.cpp	2129	wDeccelerationPoint = (WORD)GetPrivateProfileInt(Ident,"DeccelerationPoint",20,GetCFile());	TDC_Drive::TDC_Drive()
Motors.cpp	2221	DWORD velocity;	TDC_Drive::SetSpeed(double)
Motors.cpp	2232	DWORD vel;	TDC_Drive::GetSpeed()

$KOMP_{12} = \text{TDC\_Drive},\ Anz.MAA = 4$

Counters.cpp	281	int TDevice::SetExposureValues(float et,DWORD ec,float ef)	TDevice::SetExposureValues(float,unsigned_long,float)
Counters.cpp	295	void TDevice::GetExposureValues(float& et,DWORD& ec,float& ef)	TDevice::GetExposureValues(float_&,unsigned_long_&,float_&)
Counters.cpp	302	double TDevice::CalcTime(DWORD mssum)	TDevice::CalcTime(unsigned_long)
Counters.cpp	334	CALLBACK TDevice::EventHandler(UINT,UINT,DWORD,DWORD,DWORD)	TDevice::EventHandler(unsigned_int,unsigned_int,unsigned_lon
M_devcom.h	60	int SetExposureValues(float,DWORD,float);	TDevice::SetExposureValues(float,unsigned_long,float)
M_devcom.h	71	void GetExposureValues(float&,DWORD&,float&);	TDevice::GetExposureValues(float_&,unsigned_long_&,float_&)
M_devcom.h	80	virtual int GetData(WORD*,WORD,WORD) {return 0;};	TDevice::GetData(TCurve_&)
M_devcom.h	119	static CALLBACK EventHandler(UINT,UINT,DWORD,DWORD,DWORD);	TDevice::EventHandler(unsigned_int,unsigned_int,unsigned_lon
M_devcom.h	151	double CalcTime(DWORD);	TDevice::CalcTime(unsigned_long)

$KOMP_{13} = \text{TDevice},\ Anz.MAA = 9$

M_steerg.cpp	2952	void TEditWindow::rButtonDown(LONG lParam)	TEditWindow::rButtonDown(long)
M_steerg.cpp	2958	pp.x = LOWORD(lParam);	TEditWindow::rButtonDown(long)
M_steerg.cpp	2959	pp.y = HIWORD(lParam);	TEditWindow::rButtonDown(long)
M_steerg.h	576	void rButtonDown(LONG lParam);	TEditWindow::rButtonDown(long)

$KOMP_{14} = \text{TEditWindow},\ Anz.MAA = 4$

M_devhw.h	41	int GetData(WORD *, WORD, WORD)	TEncoder::GetData(TCurve_&)

$KOMP_{15} = \text{TEncoder},\ Anz.MAA = 1$

Counters.cpp	534	DWORD param;	TGenericDevice::Initialize()
Counters.cpp	550	DWORD value;	TGenericDevice::SetParameters()
Counters.cpp	552	value = fExposureTime * (DWORD)GetTicksPerSecond();	TGenericDevice::SetParameters()
Counters.cpp	589	DWORD counts,ticks;	TGenericDevice::PollDevice()
Counters.cpp	646	CALLBACK TGenericDevice::EventHandler(UINT,UINT,DWORD,DWORD,DWORD)	TGenericDevice::EventHandler(unsigned_int,unsigned_int,unsig
M_devhw.h	22	static CALLBACK TGenericDevice::EventHandler(UINT, UINT, DWORD…	TGenericDevice::EventHandler(unsigned_int,unsigned_int,unsig

$KOMP_{16} = \text{TGenericDevice},\ Anz.MAA = 6$

M_main.cpp	910	if(0xFFFF == (WORD)GET_WM_COMMAND_HWND(wParam,lParam))	TMain::TMain()
M_main.cpp	1429	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Datei");	TMain::LoadMenuBar()
M_main.cpp	1434	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Bearbeiten");	TMain::LoadMenuBar()
M_main.cpp	1449	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Öffnen");	TMain::LoadMenuBar()
M_main.cpp	1461	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Ausführen");	TMain::LoadMenuBar()
M_main.cpp	1475	AppendMenu(hmPL1,MF_POPUP,(WORD)hmPL2,"&Antriebe");	TMain::LoadMenuBar()
M_main.cpp	1488	AppendMenu(hmPL1,MF_POPUP,(WORD)hmPL2,"&Detektoren");	TMain::LoadMenuBar()
M_main.cpp	1494	AppendMenu(hmPL1,MF_POPUP,(WORD)hmPL2,"Matro&x");	TMain::LoadMenuBar()
M_main.cpp	1498	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Einstellungen");	TMain::LoadMenuBar()
M_main.cpp	1505	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,(LPSTR)"&Fenster");	TMain::LoadMenuBar()
M_main.cpp	1511	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,(LPSTR)"&Hilfe");	TMain::LoadMenuBar()
M_main.cpp	1533	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&File");	TMain::LoadMenuBar()
M_main.cpp	1538	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Edit");	TMain::LoadMenuBar()
M_main.cpp	1550	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Open");	TMain::LoadMenuBar()
M_main.cpp	1562	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Execute");	TMain::LoadMenuBar()
M_main.cpp	1575	AppendMenu(hmPL1,MF_POPUP,(WORD)hmPL2,"&Motors");	TMain::LoadMenuBar()
M_main.cpp	1583	AppendMenu(hmPL1,MF_POPUP,(WORD)hmPL2,"&Detectors");	TMain::LoadMenuBar()
M_main.cpp	1585	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,"&Settings");	TMain::LoadMenuBar()
M_main.cpp	1592	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,(LPSTR)"&Window");	TMain::LoadMenuBar()

$KOMP_{17} = \text{TMain},\ Anz.MAA = 19$

Comclass.h	65	virtual void rButtonDown(LONG) {};	TMDIWindow::rButtonDown(long)
Comclass.h	147	WORD RegisterWnd(void);	TMDIWindow::RegisterWnd()
M_main.cpp	180	wndClass.cbWndExtra = 0;	TMDIWindow::GetWindowClass(tagWNDCLASSA_*)
M_main.cpp	1146	LRESULT CALLBACK _export WndProc(HWND hWindow,UINT message…	TMDIWindow::GetWindowClass(tagWNDCLASSA_*)
M_main.cpp	1747	WORD TMDIWindow::RegisterWnd(void)	TMDIWindow::RegisterWnd()
M_main.cpp	1761	wndClass->cbClsExtra = 0;	TMDIWindow::GetWindowClass(tagWNDCLASSA_*)

$KOMP_{18} = \text{TMDIWindow},\ Anz.MAA = 6$

Dlg_tpl.cpp	61	dialogFunc = MakeProcInstance((FARPROC)DialogProc,hInstance);	TModalDlg::TModalDlg(char_*)

$KOMP_{19} = \text{TModalDlg},\ Anz.MAA = 1$

Dlg_tpl.cpp	141	dialogFunc = (DLGPROC)MakeProcInstance((FARPROC)ModelessProc,GetMainInstance());	TModelessDlg::TModelessDlg(char_*)

$KOMP_{20} = \text{TModelessDlg},\ Anz.MAA = 1$

M_motcom.h	54	DWORD GetLimit (void) {return dwRemoveLimit;};	TMotor::GetLimit()
M_motcom.h	60	virtual BOOL SetLimit (DWORD) {return TRUE;};	TMotor::SetLimit(unsigned_long)
M_motcom.h	104	virtual WORD GetStatus (void) {return 0xFF;};	TMotor::GetStatus()
Motors.cpp	1467	dwHysteresis = (DWORD)GetPrivateProfileInt(Ident,"Hysteresis",0,GetCFile());	TMotor::TMotor()
Motors.cpp	1567	wPositionMinWidth = (WORD)atoi(buf);	TMotor::Initialize()
Motors.cpp	1572	wPositionMaxWidth = (WORD)atoi(buf);	TMotor::Initialize()
Motors.cpp	1589	wPositionWidth = (WORD)atoi(buf);	TMotor::Initialize()
Motors.cpp	1623	dwRemoveLimit = (DWORD)atol(buf);	TMotor::Initialize()
Motors.cpp	1682	dwInterval = (DWORD)(3.0 * MaxFailure / dKoeff_1);	TMotor::Initialize()

$KOMP_{21} = \text{TMotor},\ Anz.MAA = 9$

Motors.cpp	1139	WORD valueW;	TMotorParam::CanClose()
Motors.cpp	1140	DWORD valueDW;	TMotorParam::CanClose()
Motors.cpp	1165	valueDW = (DWORD)atol(buf);	TMotorParam::CanClose()
Motors.cpp	1175	valueW = (WORD)atoi(buf);	TMotorParam::CanClose()

$KOMP_{22} = \text{TMotorParam},\ Anz.MAA = 4$

Motors.cpp	1380	Drive->wStaticGain = (WORD)atoi(buf);	TOptimizeDC_812::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1382	Drive->wDynamicGain = (WORD)atoi(buf);	TOptimizeDC_812::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1384	Drive->wTorque = (WORD)atoi(buf);	TOptimizeDC_812::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1386	Drive->wPositionWidth = (WORD)atoi(buf);	TOptimizeDC_812::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1407	Drive->wStaticGain = (WORD)atoi(buf);	TOptimizeDC_812::CanClose()
Motors.cpp	1409	Drive->wDynamicGain = (WORD)atoi(buf);	TOptimizeDC_812::CanClose()
Motors.cpp	1411	Drive->wTorque = (WORD)atoi(buf);	TOptimizeDC_812::CanClose()

$KOMP_{23} = \text{TOptimizeDC\_812},\ Anz.MAA = 7$

Motors.cpp	1273	Drive->wKD = (WORD)atoi(buf);	TOptimizeDC_832::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1275	Drive->wKP = (WORD)atoi(buf);	TOptimizeDC_832::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1277	Drive->wKI = (WORD)atoi(buf);	TOptimizeDC_832::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1279	Drive->wKL = (WORD)atoi(buf);	TOptimizeDC_832::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne
Motors.cpp	1281	Drive->wPositionWidth = (WORD)atoi(buf);	TOptimizeDC_832::Dlg_OnCommand(HWND___*,int,HWND___*,unsigne

$KOMP_{24} = \text{TOptimizeDC\_832},\ Anz.MAA = 5$

M_data.cpp	1514	xi = LOWORD(lParam) - xBorder;	TPlotData::lButtonDown(unsigned_int,long)
M_data.cpp	1515	yi = Screen.y1 - HIWORD(lParam) - yBorder;	TPlotData::lButtonDown(unsigned_int,long)
M_data.cpp	1520	xi = LOWORD(lParam) - xBorder;	TPlotData::lButtonDown(unsigned_int,long)
M_data.cpp	1521	yi = Screen.y1 - HIWORD(lParam);	TPlotData::lButtonDown(unsigned_int,long)
M_data.cpp	1640	xi = LOWORD(lParam) - xBorder;	TPlotData::MouseMove(unsigned_int,long)
M_data.cpp	1641	yi = Screen.y1 - HIWORD(lParam);	TPlotData::MouseMove(unsigned_int,long)

$KOMP_{25} = \text{TPlotData},\ Anz.MAA = 6$

Counters.cpp	806	hCountBuf = GlobalAlloc(GHND,GetBufferSize() * sizeof(DWORD));	TPsd::Initialize()
Counters.cpp	858	LPDWORD lpdwData;	TPsd::GetData(float_&)
Counters.cpp	869	lpdwData = (LPDWORD)GlobalLock(hCountBuf);	TPsd::GetData(float_&)
Counters.cpp	1097	LPDWORD lpdwCountBuf;	TPsd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1098	DWORD counts;	TPsd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1103	lpdwCountBuf = (LPDWORD)GlobalLock(hCountBuf);	TPsd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1111	counts = (DWORD)(argum * 300.0 + (rand() % 7));	TPsd::PsdReadOut(THowReadOutPsd)
M_psd.h	56	int GetData(WORD*,WORD,WORD) {return 0;};	TPsd::GetData(unsigned_short_*,unsigned_short,unsigned_short
M_psd.h	99	virtual int PsdRead(int,int,int,LPWORD) {return R_OK;};	TPsd::PsdRead(int,int,int,unsigned_short_*)
M_psd.h	119	int PsdRead(int,int,int,LPWORD);	TPsd::PsdRead(int,int,int,unsigned_short_*)

$KOMP_{26} = \text{TPsd},\ Anz.MAA = 10$

Counters.cpp	1572	LONG FileSize;	TRadicon::Initialize()
Counters.cpp	1648	DWORD counts;	TRadicon::MeasureStart()
Counters.cpp	1724	DWORD counts;	TRadicon::PollDevice()
Counters.cpp	1792	DWORD counts = dwExposureCounts;	TRadicon::SetParameters()
Counters.cpp	1793	WORD lowthresh,highthresh,voltage = wHighVoltage;	TRadicon::SetParameters()
Counters.cpp	1795	wDacLowerThresh = min((WORD)1022,max(wDacLowerThresh,(WORD)0));	TRadicon::SetParameters()
Counters.cpp	1796	wDacUpperThresh = max((WORD)1,min(wDacUpperThresh,(WORD)1023));	TRadicon::SetParameters()
Counters.cpp	1811	void CALLBACK TRadicon::EventHandler(UINT,UINT,DWORD,DWORD,DWORD)	TRadicon::EventHandler(unsigned_int,unsigned_int,unsigned_lo
Counters.cpp	1813	DWORD counts;	TRadicon::EventHandler(unsigned_int,unsigned_int,unsigned_lo
M_devhw.h	70	static void CALLBACK TRadicon::EventHandler(UINT, UINT, DWORD…	TRadicon::EventHandler(unsigned_int,unsigned_int,unsigned_lo

$KOMP_{27} = \text{TRadicon},\ Anz.MAA = 10$

M_main.cpp	1598	AppendMenu(hTheMenu,MF_POPUP,(WORD)hmPL1,(LPSTR)"&Help");	TScan::TScan()
M_scan.cpp	1085	void TScan::rButtonDown(LONG lParam)	TScan::rButtonDown(long)
M_scan.cpp	1094	pp.x = LOWORD(lParam);	TScan::rButtonDown(long)
M_scan.cpp	1095	pp.y = HIWORD(lParam);	TScan::rButtonDown(long)

$KOMP_{28} = \text{TScan},\ Anz.MAA = 4$

M_steerg.cpp	325	DWORD ocounts,counts;	TScanCmd::GetShowData(char_*)

$KOMP_{29} = \text{TScanCmd},\ Anz.MAA = 1$

M_scan.cpp	1495	switch(HIWORD(lParam)){	TSetupContinuousScan::TSetupContinuousScan(TScan_*)
M_scan.cpp	1498	sprintf(buf,"Es ist ein Fehler aufgetreten. %d",LOWORD(lParam));	TSetupContinuousScan::TSetupContinuousScan(TScan_*)

$KOMP_{30} = \text{TSetupContinuousScan},\ Anz.MAA = 2$

Counters.cpp	1143	hReadBuf = GlobalAlloc(GHND,(GetChannelNumber()+1)*sizeof(DWORD));	TStoe_Psd::PollDevice()
Counters.cpp	1218	LPDWORD lpdwCountBuf;	TStoe_Psd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1219	LPWORD lpwReadBuf;	TStoe_Psd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1221	lpdwCountBuf = (LPDWORD)GlobalLock(hCountBuf);	TStoe_Psd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1222	lpwReadBuf = (LPWORD)GlobalLock(hReadBuf);	TStoe_Psd::PsdReadOut(THowReadOutPsd)
Counters.cpp	1392	int TStoe_Psd::PsdRead(int FirstCh,int LastCh,int bIsLong, LPWORD buf)	TStoe_Psd::PsdRead(int,int,int,unsigned_short_*)
Counters.cpp	1503	int TStoe_Psd::PsdRead(int,int,int,LPWORD)	TStoe_Psd::PsdRead(int,int,int,unsigned_short_*)

$KOMP_{31} = \text{TStoe\_Psd},\ Anz.MAA = 7$

Am9513.h	40-86	const WORD CB_UP = 3; // aufwärts zählen	
C_8x2.inc	0	extern "C" WORD FAR PASCAL __D000h(void);	
C_8x2.inc	1	extern "C" WORD FAR PASCAL __D000h(void);	
C_layer.cpp	35	static DWORD expcounts;	
C_layer.cpp	42	int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
C_layer.cpp	51	int CALLBACK WEP(int)	WEP(int)
C_layer.cpp	58	LPCSTR _export WINAPI dlGetVersion(void)	dlGetVersion()
C_layer.cpp	63	HINSTANCE _export WINAPI dlGetInstance(void)	dlGetInstance()

C_layer.cpp	68	BOOL _export WINAPI InitializeCountersDLL(void)	InitializeCountersDLL()
C_layer.cpp	74	LPDList _export WINAPI GetCounterListPtr(void)	GetCounterListPtr()
C_layer.cpp	80	BOOL _export WINAPI dlSetDevice(int device)	dlSetDevice(int)
C_layer.cpp	88	int _export WINAPI dMeasureStart(void)	dMeasureStart()
C_layer.cpp	93	int _export WINAPI dMeasureStop(void)	dMeasureStop()
C_layer.cpp	98	BOOL _export WINAPI dSetExposureValues(float time,DWORD counts,float fl)	dSetExposureValues(float,unsigned_long,float)
C_layer.cpp	98	BOOL _export WINAPI dSetExposureValues(float time,DWORD counts,float fl)	dSetExposureValues(float,unsigned_long,float)
C_layer.cpp	103	BOOL _export WINAPI dGetExposureValues(float& time,DWORD&…)	dGetExposureValues(float_&,unsigned_long_&,float_&)
C_layer.cpp	103	BOOL _export WINAPI dGetExposureValues(float& time,DWORD&…)	dSetExposureValues(float,unsigned_long,float)
C_layer.cpp	109	int _export WINAPI dlGetDevice(void)	dlGetDevice()
C_layer.cpp	130	int _export WINAPI dlGetIdByName(TDeviceType at)	dlGetIdByName(TDeviceType)
C_layer.cpp	135	TDeviceType _export WINAPI dlParsingDevice(LPSTR devicename)	dlParsingDevice(char_*)
C_layer.cpp	152	BOOL _export WINAPI dlIsDeviceValid(TDeviceType type)	dlIsDeviceValid(TDeviceType)
C_layer.cpp	179	void CALLBACK _export InquireIntensity_SCS(…,DWORD,DWORD,DWORD)	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	179	void CALLBACK _export InquireIntensity_SCS(…,DWORD,DWORD,DWORD)	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	182	DWORD counts = 1000;	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	198	float WINAPI _export GetIntensity_SCS(void)	GetIntensity_SCS()
C_layer.cpp	206	void CALLBACK _export InquireIntensity_TDC(…,DWORD,DWORD,DWORD)	InquireIntensity_TDC(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	206	void CALLBACK _export InquireIntensity_TDC(…,DWORD,DWORD,DWORD)	InquireIntensity_TDC(unsigned_int,unsigned_int,unsigned_long
C_layer.cpp	230	BOOL WINAPI _export InitializeTDC_Event(float time,HWND contolwnd)	InitializeTDC_Event(float,HWND___*)
C_layer.cpp	242	float WINAPI _export GetIntensity_TDC(void)	GetIntensity_TDC()
C_layer.cpp	250	void CALLBACK _export InquireIntensity_A913(…,DWORD,DWORD,DWORD)	InquireIntensity_A913(unsigned_int,unsigned_int,unsigned_lon
C_layer.cpp	250	void CALLBACK _export InquireIntensity_A913(…,DWORD,DWORD,DWORD)	InquireIntensity_A913(unsigned_int,unsigned_int,unsigned_lon
C_layer.cpp	253	DWORD counts = 1000;	InquireIntensity_A913(unsigned_int,unsigned_int,unsigned_lon
C_layer.cpp	262	float WINAPI _export GetIntensity_A913(void)	GetIntensity_A913()
C_layer.cpp	507	float WINAPI _export maxf(float value1, float value2)	maxf(float,float)
C_layer.cpp	511	float WINAPI _export minf(float value1, float value2)	minf(float,float)
C_layer.h	18	BOOL WINAPI dSetExposureValues (float,DWORD,float);	dSetExposureValues(float,unsigned_long,float)
C_layer.h	19	BOOL WINAPI dGetExposureValues (float&,DWORD&,float&);	dGetExposureValues(float_&,unsigned_long_&,float_&)
C_layer.h	21	void CALLBACK InquireIntensity_SCS (…,DWORD,DWORD,DWORD);	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
C_layer.h	23	void CALLBACK InquireIntensity_TDC (…,DWORD,DWORD,DWORD);	InquireIntensity_TDC(unsigned_int,unsigned_int,unsigned_long
C_layer.h	26	void CALLBACK InquireIntensity_TDC (…,DWORD,DWORD,DWORD);	InquireIntensity_SCS(unsigned_int,unsigned_int,unsigned_long
Comclass.h	32	LRESULT CALLBACK _export WndProc (HWND,UINT,WPARAM,LPARAM);	WndProc(HWND___*,unsigned_int,unsigned_int,long)
Comclass.h	32	LRESULT CALLBACK _export WndProc (HWND,UINT,WPARAM,LPARAM);	WndProc(HWND___*,unsigned_int,unsigned_int,long)
Comclass.h	33	LRESULT CALLBACK _export FrameWndProc (HWND,UINT,…);	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
Comclass.h	33	LRESULT CALLBACK _export FrameWndProc (HWND,UINT,…);	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
Comclass.h	53	friend LRESULT CALLBACK FrameWndProc (HWND,UINT,WPARAM,LPARAM);	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
Comclass.h	54	friend LRESULT CALLBACK WndProc (HWND,UINT,WPARAM,LPARAM);	WndProc(HWND___*,unsigned_int,unsigned_int,long)
Comhead.h	282	typedef WORD* HPWORD;	
Comhead.h	287	typedef WORD _huge * HPWORD;	
Comhead.h	284-287	typedef char _huge * HPSTR;	
Dlg_tpl.cpp	16	BOOL CALLBACK _export DialogProc(HWND hDlg,UINT msg,…)	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.cpp	109	BOOL CALLBACK _export ModelessProc(HWND hDlg,UINT msg,…)	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.h	7	BOOL CALLBACK _export DialogProc (HWND,UINT,WPARAM,LPARAM);	DialogProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.h	8	BOOL CALLBACK _export ModelessProc (HWND,UINT,WPARAM,LPARAM)	ModelessProc(HWND___*,unsigned_int,unsigned_int,long)
Dlg_tpl.h	40	FARPROC dialogFunc;	
Ieee.h	45	typedef int (WINAPI *TSend) (int,LPSTR,WORD,LPINT);	
Ieee.h	46	typedef int (WINAPI *TEnter) (LPSTR,WORD,LPWORD,int,LPINT);	
Ieee.h	47	typedef int (WINAPI *TTransmit) (LPSTR,WORD,LPINT);	
Ieee.h	48	typedef int (WINAPI *TReceive) (LPSTR,WORD,LPWORD,LPINT);	
Ieee.h	51	typedef void (WINAPI *TSetPort) (int,WORD);	
Ieee.h	52	typedef void (WINAPI *TSetTimeout) (WORD);	
Ieee.h	69	void far pascal ieee488_protcheck(int);	
Ieee.h	69	void far pascal ieee488_protcheck(int);	
Ieee.h	13-39	void far pascal ieee488_initialize (int,int);	
Ieee.h	13-39	void far pascal ieee488_initialize (int,int);	

Ieee.h	75-82	#define transmit(cmd,status) ieee488_transmit((char far *) (cmd),0xFFFF,(int far *) status)	
Kmpt1.c	681	DWORD crnt_time()	crnt_time()
L_layer.cpp	27	BOOL DllEntryPoint(HANDLE hModule,DWORD dwFunction,LPVOID)	DllEntryPoint(void_*,unsigned_long,void_*)
L_layer.cpp	42	int FAR PASCAL LibMain(HINSTANCE hInstance,WORD,WORD…)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
L_layer.cpp	42	int FAR PASCAL LibMain(HINSTANCE hInstance,WORD,WORD…)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
L_layer.cpp	42	int FAR PASCAL LibMain(HINSTANCE hInstance,WORD,WORD…)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
L_layer.cpp	42	int FAR PASCAL LibMain(HINSTANCE hInstance,WORD,WORD…)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
L_layer.cpp	53	int CALLBACK WEP(int bSystemExit)	WEP(int)
L_layer.cpp	60	LPCSTR _export WINAPI spGetVersion(void)	spGetVersion()
L_layer.cpp	65	HINSTANCE _export WINAPI spGetInstance(void)	spGetInstance()
L_layer.cpp	87	void _export WINAPI AboutTheMaker(void)	AboutTheMaker()
L_layer.cpp	129	LPCSTR _export WINAPI GetCFile(void)	GetCFile()
L_layer.cpp	134	HWND _export WINAPI GetFrameHandle(void)	GetFrameHandle()
L_layer.cpp	139	HWND _export WINAPI GetClientHandle(void)	GetClientHandle()
L_layer.cpp	144	HINSTANCE _export WINAPI GetMainInstance(void)	GetMainInstance()
L_layer.cpp	149	LPDataBase _export WINAPI GetDataBasePtr(void)	GetDataBasePtr()
L_layer.cpp	155	BOOL _export WINAPI InitializeSPLibrary(TMainParameters* MainParam)	InitializeSPLibrary(TMainParameters_*)
L_layer.cpp	162	TxScanType _export WINAPI ParsingXScanType(LPSTR xst)	ParsingXScanType(char_*)
L_layer.cpp	174	BOOL _export WINAPI CreateDefaults(void)	CreateDefaults()
L_layer.cpp	179	void _export WINAPI SetInfo(LPCSTR status_txt)	SetInfo(char_*)
L_layer.cpp	185	SendMessage(MainP.hWndFrame,MainP.wm_DrawStatus,1002,(LONG)hAtom);	SetInfo(char_*)
L_layer.cpp	188	void _export WINAPI SetStaticInfo(LPCSTR status_txt)	SetStaticInfo(char_*)
L_layer.cpp	194	PostMessage(MainP.hWndFrame,MainP.wm_DrawStatus,1000,(LONG)hAtom);	SetStaticInfo(char_*)
L_layer.cpp	198	void _export WINAPI SetStatus(LPCSTR status_txt)	SetStatus(char_*)
L_layer.cpp	204	PostMessage(MainP.hWndFrame,MainP.wm_DrawStatus,1001,(LONG)hAtom);	SetStatus(char_*)
L_layer.cpp	207	BOOL _export WINAPI FileOpen(LPCSTR HeadLine,LPSTR szFilter,LPSTR…	FileOpen(char_*,char_*,char_*,char_*)
L_layer.cpp	210	DWORD Errval; // Error value	FileOpen(char_*,char_*,char_*,char_*)
L_layer.cpp	258	BOOL _export WINAPI FileSave(LPCSTR HeadLine,LPSTR szFilter,LPSTR…	FileSave(char_*,char_*,char_*,char_*)
L_layer.cpp	261	DWORD Errval; // Error value	FileSave(char_*,char_*,char_*,char_*)
L_layer.cpp	311	BOOL _export WINAPI SetFPOnData(HFILE hfile)	SetFPOnData(int)
L_layer.cpp	364	int WINAPI _export GetBufferLine(LPSTR src,LPSTR dst,int dstsize)	
L_layer.cpp	367	static WORD cnt_s;	
L_layer.cpp	402	int WINAPI _export GetFileLine(int hFile,LPSTR dst,int dstsize)	GetFileLine(int,char_*,int)
L_layer.cpp	443	TUnitType WINAPI _export UnitEnum(LPSTR sz_unit)	UnitEnum(char_*)
L_layer.cpp	459	LPSTR WINAPI _export UnitStr(TUnitType n_unit)	UnitStr(TUnitType)
L_layer.cpp	472	void WINAPI _export Delay(long count)	Delay(long)
L_layer.cpp	483	void WINAPI _export DelayTime(int MilliSec)	DelayTime(int)
L_layer.cpp	485	DWORD StartTime = timeGetTime();	DelayTime(int)
L_layer.cpp	491	int WINAPI _export maxi(int value1, int value2)	maxi(int,int)
L_layer.cpp	495	int WINAPI _export mini(int value1, int value2)	mini(int,int)
L_layer.cpp	499	long WINAPI _export maxl(long value1, long value2)	maxl(long,long)
L_layer.cpp	503	long WINAPI _export minl(long value1, long value2)	minl(long,long)
L_layer.cpp	515	double WINAPI _export maxd(double value1, double value2)	maxd(double,double)
L_layer.cpp	519	double WINAPI _export mind(double value1, double value2)	mind(double,double)
M_arscan.cpp	195	pp.x = LOWORD(lParam);	
M_arscan.cpp	196	pp.y = HIWORD(lParam);	
M_arscan.cpp	271	AppendMenu(TheMenu,MF_POPUP,(WORD)hMAdd,"&Psd");	MenuSelect(HWND___*,unsigned_int,long)
M_arscan.cpp	595	switch(HIWORD(lParam)){	
M_arscan.cpp	599	sprintf(buf,"Es ist ein Fehler aufgetreten. %d",LOWORD(lParam));	
M_curve.h	7	#define _CURVECLASS _export	
M_curve.h	243	typedef TCurve FAR * LPCurve;	
M_curve.h	315	typedef TDataBase FAR * LPDataBase;	
M_data.cpp	47	static WORD RLdx = 350;	
M_data.cpp	50	static WORD RLdy = 500;	
M_devcom.h	21	#define _COUNTERCLASS _export	
M_devcom.h	32	DWORD dwExposureCounts;	

M_devcom.h	155	DWORD dwExposureCounts;	
M_devcom.h	230	float _huge *IField;	
M_devcom.h	259	typedef TDList FAR * LPDList;	
M_devcom.h	260	typedef TDevice FAR * LPDevice;	
M_devcom.h	144-146	DWORD dwElapsedTime,dwStartTime;	
M_devhw.h	145	BOOL WriteTag(WORD,WORD,DWORD,DWORD,HFILE);	
M_devhw.h	84-86	WORD wDacUpperThresh;	
M_device.cpp	416	pp.x = LOWORD(lParam);	
M_device.cpp	417	pp.y = HIWORD(lParam);	
M_layer.cpp	19	int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
M_layer.cpp	33	LPCSTR _export WINAPI mlGetVersion(void)	mlGetVersion()
M_layer.cpp	41	HINSTANCE _export WINAPI mlGetInstance(void)	mlGetInstance()
M_layer.cpp	49	int _export WINAPI mGetScanSize(void)	mGetScanSize(
M_layer.cpp	55	void _export WINAPI mStartMoveScan(int tic,int)	mStartMoveScan(int,int)
M_layer.cpp	92	LPLONG _export WINAPI mGetMoveScan(void)	mGetMoveScan()
M_layer.cpp	92	LPLONG _export WINAPI mGetMoveScan(void)	mGetMoveScan()
M_layer.cpp	95	return (LPLONG)Scan;	mGetMoveScan()
M_layer.cpp	98	int _export WINAPI mGetMoveFinishIdx(void)	mGetMoveFinishIdx()
M_layer.cpp	103	void _export CALLBACK mSavePosition(…,DWORD,DWORD,DWORD)	mSavePosition(unsigned_int,unsigned_int,unsigned_long,unsign
M_layer.cpp	103	void _export CALLBACK mSavePosition(…,DWORD,DWORD,DWORD)	mSavePosition(unsigned_int,unsigned_int,unsigned_long,unsign
M_layer.cpp	136	BOOL _export WINAPI mlInitializeMotorsDLL(void)	mlInitializeMotorsDLL()
M_layer.cpp	148	BOOL _export WINAPI mlSetAxis(int axis)	mlSetAxis(int)
M_layer.cpp	157	int _export WINAPI mlGetAxis(void)	mlGetAxis()
M_layer.cpp	164	int _export WINAPI mlGetIdByName(TAxisType at)	mlGetIdByName(TAxisType)
M_layer.cpp	189	TAxisType _export WINAPI mlParsingAxis(LPSTR axisname)	mlParsingAxis(char_*)
M_layer.cpp	194	BOOL _export WINAPI mlIsAxisValid(TAxisType type)	mlIsAxisValid(TAxisType)
M_layer.cpp	217	BOOL _export WINAPI mlIsServerOK(void)	mlIsServerOK()
M_layer.cpp	222	int _export WINAPI mlGetAxisNumber(void)	mlGetAxisNumber()
M_layer.cpp	227	double _export WINAPI mlGetOffset(int mid)	mlGetOffset(int)
M_layer.cpp	232	void _export WINAPI mlSetAngleDefault(void)	mlSetAngleDefault()
M_layer.cpp	237	void _export WINAPI mlOptimizingDlg(void)	mlOptimizingDlg()
M_layer.cpp	243	BOOL _export WINAPI mlGetDistance(int mid,double& distance)	mlGetDistance(int,double_&)
M_layer.cpp	257	double _export WINAPI mlGetValue(int mid,TValueType vtype)	mlGetValue(int,TValueType)
M_layer.cpp	270	BOOL _export WINAPI mlMoveToDistance(int mid,double distance)	mlMoveToDistance(int,double)
M_layer.cpp	287	BOOL _export WINAPI mlIsMoveFinish(int mid)	mIsMoveFinish()
M_layer.cpp	292	void _export WINAPI mlSetParametersDlg(void)	mlSetParametersDlg()
M_layer.cpp	297	void _export WINAPI mlPositionControlDlg(void)	mlPositionControlDlg()
M_layer.cpp	302	void _export WINAPI mlSaveModuleSettings(void)	mlSaveModuleSettings()
M_layer.cpp	307	void _export WINAPI mlInquireReferencePointDlg(int task)	mlInquireReferencePointDlg(int)
M_layer.cpp	315	BOOL _export WINAPI mSetLine(int number,BOOL state)	mSetLine(int,int)
M_layer.cpp	323	void _export WINAPI mSetRelativeZero(BOOL yes,double value)	mSetRelativeZero(int,double)
M_layer.cpp	331	BOOL _export WINAPI mIsDistanceRelative(void)	mIsDistanceRelative()
M_layer.cpp	336	BOOL _export WINAPI mIsRangeHit(void)	mIsRangeHit()
M_layer.cpp	341	BOOL _export WINAPI mMoveByDistance(double angle)	mMoveByDistance(double)
M_layer.cpp	352	BOOL _export WINAPI mMoveToDistance(double angle)	mMoveToDistance(double)
M_layer.cpp	369	BOOL _export WINAPI mIsMoveFinish(void)	mIsMoveFinish()
M_layer.cpp	378	BOOL _export WINAPI mGetDistance(double &ang)	mGetDistance(double_&)
M_layer.cpp	396	double _export WINAPI mGetDistanceProcess(void)	mGetDistanceProcess()
M_layer.cpp	402	void _export WINAPI mStopDrive(BOOL restart)	mStopDrive(int)
M_layer.cpp	407	double _export WINAPI mGetValue(TValueType vtype)	mGetValue(TValueType)
M_layer.cpp	420	BOOL _export WINAPI mSetValue(TValueType vtype,double value)	mSetValue(TValueType,double)
M_layer.cpp	434	TUnitType _export WINAPI mGetUnitType(void)	mGetUnitType()
M_layer.cpp	439	BOOL _export WINAPI mIsCalibrated(void)	mIsCalibrated()
M_layer.cpp	444	void _export WINAPI mActivateDrive(void)	mActivateDrive()
M_layer.cpp	449	void _export WINAPI mSetCorrectionState(BOOL onoff)	mSetCorrectionState(int)
M_layer.cpp	454	LPCSTR _export WINAPI mGetAxisName(void)	mGetAxisName()

M_layer.cpp	459	LPCSTR _export WINAPI mGetAxisUnit(void)	mGetAxisUnit()
M_layer.cpp	464	LPCSTR _export WINAPI mGetSF(void)	mGetSF()
M_layer.cpp	469	LPCSTR _export WINAPI mGetDF(void)	mGetDF()
M_layer.cpp	474	int _export WINAPI mExecuteCmd(LPSTR cmd)	mExecuteCmd(char_*)
M_layer.cpp	479	void _export WINAPI mPushSettings(void)	mPushSettings()
M_layer.cpp	484	void _export WINAPI mPopSettings(TMParameter mp)	mPopSettings(TMParameter)
M_layer.cpp	492	int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
M_layer.cpp	496	BOOL DllEntryPoint(HANDLE hModule,DWORD dwFunction,LPVOID)	DllEntryPoint(void_*,unsigned_long,void_*)
M_layer.cpp	527	int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
M_layer.cpp	535	int CALLBACK WEP(int)	WEP(int)
M_layer.h	233	void CALLBACK mSavePosition (…,DWORD,DWORD,DWORD);	mSavePosition(unsigned_int,unsigned_int,unsigned_long,unsign
M_layer.h	240	LPLONG WINAPI mGetMoveScan	mGetMoveScan()
M_layer.h	304	typedef LPLONG WINAPI (*TGetMoveScan) (void);	
M_layer.h	307	typedef void CALLBACK (*TSavePosition) (UINT,UINT,DWORD,DWORD,DWORD);	
m_lscan.h	19	DWORD GetImageSize(BITMAPINFOHEADER &);	
m_lscan.h	73	void rButtonDown(LONG lParam);	
M_main.cpp	154	int PASCAL WinMain(HINSTANCE hInstance,HINSTANCE…)	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	178	wndClass.lpfnWndProc = FrameWndProc;	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	179	wndClass.cbClsExtra = 0;	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	274	CountersDLLInstance = GetModuleHandle("counters.dll");	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	369	NewWindow((TCounterWindow *)new TCounterWindow(LOWORD(lParam)));	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	499	HWND hChildWnd = (HWND)LOWORD(SendMessage(Main.hWndClient,WM_…	DoCommandsFrame(HWND___*,unsigned_int,long)
M_main.cpp	664	LPLONG data = mGetMoveScan();	DoCommandsChild(TMDIWindow_*,HWND___*,unsigned_int,long)
M_main.cpp	746	window->LoadOldData(LOWORD(lParam));	DoCommandsChild(TMDIWindow_*,HWND___*,unsigned_int,long)
M_main.cpp	991	LRESULT CALLBACK _export FrameWndProc(HWND …	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	991	LRESULT CALLBACK _export FrameWndProc(HWND …	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	998	char buf[MaxString];	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1127	PostMessage((HWND)LOWORD(lParam),WM_COMMAND,cm_Index,0);	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1146	LRESULT CALLBACK _export WndProc(HWND…	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1161	SetWindowLong(hWindow,1,DWORD(window));	WndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1715	(LONG)(LPMDICREATESTRUCT)&mdiCreate);	NewWindow(TMDIWindow_*)
M_main.cpp	1762	wndClass->cbWndExtra = 8;	WinMain(HINSTANCE___*,HINSTANCE___*,char_*,int)
M_main.cpp	1771	wndClass->lpfnWndProc = WndProc;	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1056...	if(1 < GetModuleUsage(hMDLL))...	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	1083...	bShutdown = TRUE; if(2 < GetModuleUsage(hMDLL))...	FrameWndProc(HWND___*,unsigned_int,unsigned_int,long)
M_main.cpp	259...	hMDLL = GetModuleHandle("motors.dll");...	DoCommandsFrame(HWND___*,unsigned_int,long)
M_motcom.h	106	WORD wPositionMinWidth;	
M_motcom.h	107	WORD wPositionMaxWidth;	
M_motcom.h	112	WORD wPositionWidth;	
M_motcom.h	133	DWORD dwRemoveLimit;	
M_motcom.h	137	DWORD dwHysteresis;	
M_motcom.h	156	DWORD dwInterval;	
M_motcom.h	204	typedef TMList FAR * LPMList;	
M_motcom.h	205	typedef TMotor FAR * LPMotor;	
M_motcom.h	141-143	DWORD dwVelocity;	
m_mothw.h	134	WORD wBaseAddr;	
m_mothw.h	149	WORD wGPIBAddr;	
m_mothw.h	198	WORD wBaseAddr;	
m_mothw.h	201	WORD wSample;	
m_mothw.h	100-103	WORD wTorque;	
m_mothw.h	170-173	WORD wKI; // IntegralGain	
m_mothw.h	272-74	DWORD old_vel,old_accel;	
m_mothw.h	288-290	DWORD old_vel,old_accel;	
m_mothw.h	47-50	WORD wDeathBand;	
M_psd.h	9	#define _COUNTERCLASS _export	
M_psd.h	77	DWORD dwMaxCounts,dwIntegratedCounts;	

M_psd.h	159	LONG LWert;	
M_psd.h	160	WORD IWert[2];	
M_psd.h	187	LONG PositionsDatenHeader[16];	
M_psd.h	192	LONG lPositionStop; // Countstop fuer Positionskanaele 0..EFFFFFFFFFFF hex	
M_psd.h	199	LONG lMeasTime; // Messzeit 1..9999999 dec [sec]	
M_scan.cpp	3277	AppendMenu(TheMenu,MF_POPUP,(WORD)hM2,"&Vergleichs-Scan");	
M_steerg.cpp	38	static DWORD dwStartTimeTicks;	
M_steerg.cpp	121	friend LRESULT CALLBACK FrameWndProc(HWND,UINT,WPARAM,LPARAM);	
M_steerg.cpp	755	DWORD dwElapsedTicks;	
M_steerg.cpp	1885	void CALLBACK _export RecallSteering(UINT,UINT,DWORD,DWORD,DWORD)	RecallSteering(unsigned_int,unsigned_int,unsigned_long,unsig
M_steerg.cpp	1885	void CALLBACK _export RecallSteering(UINT,UINT,DWORD,DWORD,DWORD)	RecallSteering(unsigned_int,unsigned_int,unsigned_long,unsig
M_topo.h	23	DWORD CurrentTime,StartTime;	
M_topo.h	55	WORD nNumberCycle; // Anzahl der Meß-Zyklen	
M_topo.h	66	DWORD dwMaxCounts; // Maximale Anzahl Impulse	
M_xscan.h	58	WORD nNumberCycle; // Anzahl der Meß-Zyklen	
M_xscan.h	61	DWORD dwMaxCounts; // Maximale Anzahl Impulse	
M_xscan.h	254	DWORD dwMaxCounts; // Maximale Anzahl Impulse	
M_xscan.h	669	DWORD dwMaxCounts;	
M_xscan.h	745	DWORD dwMaxCounts;	
M_xscan.h	891	DWORD dwExposureCounts;	
Motors.cpp	11	static long Drive628c(BYTE,WORD,long,WORD,WORD);	Drive628c(unsigned_char,unsigned_short,long,unsigned_short,u
Motors.cpp	90	static WORD drive;	
Motors.cpp	91	static WORD baddr;	
Motors.cpp	94	static int GetWord(WORD,WORD);	GetWord(unsigned_short,unsigned_short)
Motors.cpp	95	static long GetDWord(WORD,WORD);	GetDWord(unsigned_short,unsigned_short)
Motors.cpp	96	static void PutWord(int,WORD,WORD);	PutWord(int,unsigned_short,unsigned_short)
Motors.cpp	97	static void PutDWord(long,WORD,WORD);	PutDWord(long,unsigned_short,unsigned_short)
Motors.cpp	98	static BOOL LM628Ready(WORD);	LM628Ready(unsigned_short)
Motors.cpp	111	void _export WINAPI msSetSimulationType(TSimulationType t)	
Motors.cpp	115	void _export WINAPI msRegister_C812ISA_Get(T812ISA_GET_CALLBACK cb)	
Motors.cpp	119	void _export WINAPI msRegister_C812ISA_Put(T812ISA_PUT_CALLBACK cb)	
Motors.cpp	123	void _export WINAPI msRegister_C832_Get(T832_GET_CALLBACK cb)	
Motors.cpp	127	void _export WINAPI msRegister_C832_Put(T832_PUT_CALLBACK cb)	
Motors.cpp	3751	long Drive628c(BYTE cmd,WORD ctrl_word,long param,WORD base,…)	Drive628c(unsigned_char,unsigned_short,long,unsigned_short,u
Motors.cpp	3837	long GetDWord(WORD base,WORD regaddr)	GetDWord(unsigned_short,unsigned_short)
Motors.cpp	3861	void PutWord(int data,WORD base,WORD regaddr)	PutWord(int,unsigned_short,unsigned_short)
Motors.cpp	3884	void PutDWord(long data,WORD base,WORD regaddr)	PutWPutDWord(long,unsigned_short,unsigned_short)
Motors.cpp	3910	BOOL LM628Ready(WORD base)	LM628Ready(unsigned_short)
Pcl_830.inc	37-84	const WORD CB_UP = 3; // aufwärts zählen	
Radicon.h	36	#define BiosDataSeg ((WORD)(__0040h))	
Radicon.h	43	#define CALLTYPE _far _Cdecl _export	
Radicon.h	43	#define CALLTYPE _far _Cdecl _export	
Radicon.h	54	int CALLTYPE setprm (int,int,WORD,WORD,int,double,DWORD,int,int,int);	setprm(int,int,unsigned_short,unsigned_short,int,double,unsi
Radicon.h	56	int CALLTYPE getinf (int,int,int,double*,DWORD*);	getinf(int,int,int,double_*,unsigned_long_*)
St_layer.cpp	19	int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
St_layer.cpp	28	int CALLBACK WEP(int)	WEP(int)
St_layer.cpp	33	BOOL WINAPI _export InitializeLineScanModule(HWND hcwnd)	
St_layer.cpp	57	BOOL WINAPI _export AquireOptimalExTime(void)	
St_layer.cpp	62	BOOL WINAPI _export SetLineScanParameters(void)	
St_layer.cpp	72	BOOL _export WINAPI GetScanRanges(TFrameDimension* fd)	
St_layer.cpp	77	BOOL _export WINAPI SetImageSaveOptions(void)	
St_layer.cpp	87	BOOL _export WINAPI GetHistogram(LPWORD* lpHist,int* len)	
St_layer.cpp	87	BOOL _export WINAPI GetHistogram(LPWORD* lpHist,int* len)	
St_layer.cpp	95	BOOL _export WINAPI InitializeAxis(int,LPSTR,int)	
St_layer.cpp	100	BOOL WINAPI _export InitializeTask(TScannerTask task)	
St_layer.cpp	115	BOOL WINAPI _export ExecuteNextSubTask(TScannerTask task)	

```
St_layer.cpp   134     BOOL WINAPI _export ReadOutCCD(int fkt)
St_layer.cpp   140     BOOL WINAPI _export SaveImage(LPSTR fname)
St_layer.cpp   145     BOOL WINAPI _export UnlockDataBase(void)
St_layer.cpp   150     BOOL WINAPI _export GetDataBaseAccess(HPBYTE* hpDB,SIZE* size)
St_layer.cpp   527     int WINAPI LibMain(HINSTANCE inst,WORD,WORD,LPSTR)
St_layer.h     14      BOOL    WINAPI  GetHistogram          (LPWORD*,int*);
St_layer.h     30      typedef BOOL (WINAPI *TInitializeTask)      (WORD);
St_layer.h     31      typedef BOOL  (WINAPI *TExecuteNextSubTask)     (WORD);
St_layer.h     33      typedef BOOL (WINAPI *TGetHistogram)          (LPWORD*,int*);
Testdev.cpp    2       long _huge testDevData[9][17][601];
```

$KOMP_{32} = Tx,\ \ Anz.MAA = 298$

Anzahl - GROUP	
GROUP	Ergebnis
LA	109
SEP	34
Gesamtergebnis	143

$MAM_1 = Anz.MA_1 + ... + Anz.MA_n = 563$

$MAM_2 = ((KOMP_1, Anz.MAA),...,(KOMP_n, Anz.MAA)) =$

```
((TAdjustmentWindow    ,   3),
 (TAm9513a             ,  39),
 (TAreaScan            ,   4),
 (TBitmapSource        ,  17),
 (TBraun_Psd           ,  39),
 (TC_812               ,  20),
 (TC_812GPIB           ,   6),
 (TC_832               ,  18),
 (TChooseDeviceCmd     ,   1),
 (TCounterShowParam    ,   1),
 (TCounterWindow       ,   1),
 (TDC_Drive            ,   4),
 (TDevice              ,   9),
 (TEditWindow          ,   4),
 (TEncoder             ,   1),
 (TGenericDevice       ,   6),
 (TMain                ,  19),
 (TMDIWindow           ,   6),
 (TModalDlg            ,   1),
 (TModelessDlg         ,   1),
 (TMotor               ,   9),
 (TMotorParam          ,   4),
 (TOptimizeDC_812      ,   7),
 (TOptimizeDC_832      ,   5),
 (TPlotData            ,   6),
 (TPsd                 ,  10),
 (TRadicon             ,  10),
 (TScan                ,   4),
 (TScanCmd             ,   1),
 (TSetupContinuousScan ,   2),
 (TStoe_Psd            ,   7)
 (Tx                   , 298))
```

$$MAM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ MA\text{-}Zugriffen}{Anzahl\ der\ Komponenten} = \frac{32}{81} = 0,395$$

$$MA = (MAM_1, MAM_2, MAM_3) = (563, MAM_2, 0,395)$$

F 2.2 Ermittlung der Maßkomponente ’Peripheriegeräte’

zu Maß PG, siehe Kapitel 4.2.1.2

$PG_1 = outp(),\ Anz.PG_1 = 4$
$PG_2 = inp(),\ Anz.PG_2 = 5$
$PG_3 = lpfnSetPort(),\ Anz.PG_3 = 1$
$PG_4 = MK\_FP,\ Anz.PG_4 = 3$
$PG_5 = C832\_Put,\ Anz.PG_5 = 10$
$PG_6 = asm,\ Anz.PG_6 = 14$

Am9513.cpp	286	outp(nBaseAddr+3,data);	TAm9513a::DigitalOut(unsigned_char)
Am9513.cpp	293	*data = inp(nBaseAddr + 2);	TAm9513a::DigitalIn(unsigned_char_*)
Am9513.cpp	400	return inp(addr);	TAm9513a::ReadStatus()
Am9513.cpp	418	outp(addr,cmd);	TAm9513a::WriteCmd(unsigned_char)
Am9513.cpp	437	outp(addr,test);	TAm9513a::WriteData(unsigned_short)
Am9513.cpp	448	outp(addr,test); Delay(loWaitAm9513a);	TAm9513a::WriteData(unsigned_short)
Am9513.cpp	468	value = (WORD) inp(addr);	TAm9513a::ReadData()
Am9513.cpp	480	value += (WORD) ((WORD) inp(addr) << 8);	TAm9513a::ReadData()

$KOMP_1 = \text{TAm9513a},\ Anz.PGA = 8$

BraunPSD.cpp	706...	...lpfnSetPort((WORD) nBaseAddr,zeichen);...	TBraun_Psd::LoadHexFile()

$KOMP_2 = \text{TBraun\_Psd},\ Anz.PGA = 1$

Motors.cpp	2945...	lpOut1 = (LPSTR)MK_FP(FP_OFF(__C000h),0x03fc + 0x10 * (wBaseAddr;...	TC_812ISA::TC_812ISA()
Motors.cpp	2961...	lpOut1 = (LPSTR)MK_FP(FP_OFF(__D000h),0x03fc + 0x10 * (wBaseAddr...	TC_812ISA::TC_812ISA()

$KOMP_3 = \text{TC\_812ISA},\ Anz.PGA = 2$

Motors.cpp	3465	C832_Put(::baddr,::config | 0x07);status = C832_Get(::baddr+1);	TC_832::LimitWatch(unsigned_int,unsigned_int,unsigned_long,u
Motors.cpp	3500	C832_Put(wBaseAddr,cConfig | 0x07);status = C832_Get(wBaseAddr+1);	TC_832::IsLimitHit()
Motors.cpp	3522	C832_Put(wBaseAddr,cConfig | 0x07);...	TC_832::IsLimitHit()
Motors.cpp	3706	C832_Put(wBaseAddr,cConfig | (nOnBoardId << 1));...	TC_832::CheckBoardOk()
Motors.cpp	3763...	C832_Put(base,regaddr); // Ja! Fordere Auslesen des Configuration...	TC_832::Drive628(unsigned_char,unsigned_short,long)

$KOMP_4 = \text{TC\_832},\ Anz.PGA = 5$

Counters.cpp	1294	_asm { mov dx,base add dx,6...	TStoe_Psd::PsdInit()
Counters.cpp	1303	_asm {mov dx,base add dx,4...	TStoe_Psd::PsdInit()
Counters.cpp	1327..	_asm { // init PSD interface mov dx,base add dx,4...	TStoe_Psd::PsdStart()
Counters.cpp	1337..	_asm { // start PSD interface with clearing memory mov dx,base add dx,4...	TStoe_Psd::PsdStart()
Counters.cpp	1365...	_asm { // stop PSD interface mov dx,base...	TStoe_Psd::PsdStop()
Counters.cpp	1375...	_asm { // start PSD interface with clearing memory mov dx,base...	TStoe_Psd::PsdStop()
Counters.cpp	1392...	_asm { // si,di retten push di push si // initiate data transfer mov dx,0x304 // PC...	TStoe_Psd::PsdRead(int,int,int,unsigned_short_*)

$KOMP_5 = \text{TStoe\_Psd},\ Anz.PGA = 7$

C_layer.cpp	12	LPSTR pTime = (LPSTR) MK_FP(FP_OFF(__0040h),0x006c);	LibMain(HINSTANCE___*,unsigned_short,unsigned_short,char_*)
Kisl1.c	620	_asm { mov dx, rs; out dx, al;}	reset(int)
Kisl1.c	533...	/* d = inp(rd); */ _asm...	init_b(unsigned_char_*,unsigned_int,int,int)
Kmpt1.c	52	_asm {mov dx,rd; in al,dx; }	tr_message(int,int,int,unsigned_char_*,int)
Kmpt1.c	278	_asm { mov dx,rd; in al,dx;}	rc_message(int,int,int,unsigned_char_*,int,int_*)
Kmpt1.c	525...	_asm { mov dx,rd; //Portadresse des Datenregisters in dx schreiben...	out_byte(int,int,int)
Kmpt1.c	596...	_asm { sub ax,ax //Register ax loeschen...	in_byte(int,int,int_*)
Kmpt1.c	628...	_asm { mov dx,rs //Portadresse des Steuerregisters in dx schreiben...	status_rd(int)
Kmpt1.c	652...	_asm { mov dx,rs //Portadresse des Steuerregisters in dx schreiben...	status_wr(int)
Motors.cpp	3828...	C832_Put(base,regaddr + 1);temp.byte[1] = C832_Get(base + 1);...	GetWord(unsigned_short,unsigned_short)
Motors.cpp	3848...	C832_Put(base,regaddr + 1);...	GetDWord(unsigned_short,unsigned_short)
Motors.cpp	3872...	C832_Put(base,regaddr + 1);...	PutWord(int,unsigned_short,unsigned_short)
Motors.cpp	3896...	C832_Put(base,regaddr+1);...	PutWPutDWord(long,unsigned_short,unsigned_short)
Motors.cpp	3917...	// command register vom aktiven drive C832_Put(base,::config | (::drive<<1));}...	LM628Ready(unsigned_short)

$$KOMP_6 = Tx,\ Anz.PGA = 14$$

$$\mathrm{PGM}_1 = Anz.PG_1 + ... + Anz.PG_n = 37$$

$$PGM_2 = ((KOMP_1, Anz.PGA), ..., (KOMP_n, Anz.PGA)) =$$
$$((TAm9513a\ ,\ 8),\ (TBraun\_Psd\ ,\ 1),\ (TC\_812ISA\ ,\ 2),\ (TC\_832\ ,\ 5),\ (TStoe\_Psd\ ,\ 7),\ (Tx\ ,\ 14))$$

$$PGM_3 = \frac{Anzahl\ aller\ Komponenten\ mit\ PG\text{-}Zugriffen}{Anzahl\ der\ Komponenten} = \frac{6}{81} = 0,074$$

$$PG = (PGM_1, PGM_2, PGM_3) = (37,\ PGM_2,\ 0,074)$$

F 2.3 Ermittlung der Maßkomponente ’Hardware-Umgebung’

zum Maß HU , siehe Kapitel 4.2.1.2

$$PAN_{HU} = MAM_1 + PGM_1 = 563 + 37 = 600$$

$$\mathrm{KOMP}_{\mathrm{HU}} = MAM_2 + PGM_2 =$$
$$((TAdjustmentWindow\ ,\ 3),\ (TAm9513a\ ,\ 47),\ (TAreaScan\ ,\ 4),\ (TBitmapSource\ ,\ 17),\ (TBraun\_Psd\ ,\ 40),\ (TC\_812\ ,\ 20),\ (TC\_812GPIB\ ,\ 6),\ (TC\_812ISA\ ,\ 2),\ (TC\_832\ ,\ 23),\ (TChooseDeviceCmd\ ,\ 1),\ (TCounterShowParam\ ,\ 1),\ (TCounterWindow\ ,\ 1),$$

(*TDC_Drive* , 4),
(*TDevice* , 9),
(*TEditWindow* , 4),
(*TEncoder* , 1),
(*TGenericDevice* , 6),
(*TMain* , 19),
(*TMDIWindow* , 6),
(*TModalDlg* , 1),
(*TModelessDlg* , 1),
(*TMotor* , 9),
(*TMotorParam* , 4),
(*TOptimizeDC_812* , 12),
(*TPlotData* , 6),
(*TPsd* , 10),
(*TRadicon* , 10),
(*TScan* , 4),
(*TScanCmd* , 1),
(*TSetupContinuousScan* , 2),
(*TStoe_Psd* , 14),
(*Tx* 312))

$$\mathrm{VKG}_{\mathrm{HU}} = MAM_3 + PGM_3 = \frac{32}{81} = 0{,}395$$

$$HU = (PAN_{HU}, KOMP_{HU}, VKG_{HU}) = (600,\ KOMP_{HU},\ 0{,}395)$$

F 3 Maßkomponente ’System-Umgebung’

zu Maß *SYS* , siehe Kapitel 4.2.1.4

$$PAN_{SYS} = PAN_{SU} + PAN_{HU} = 356 + 600 = 956$$

$KOMP_{SYS} = KOMP_{SU} + KOMP_{HU} =$
((*TAbout* , 2),
(*TAdjustmentExecute* , 2),
(*TAdjustmentWindow* , 4),
(*TAm9513a* , 49),
(*TAngleControl* , 32),
(*TAquisition* , 2),
(*TAreaScan* , 18),
(*TAreaScanCmd* , 1),
(*TAreaScanParameters* , 1),
(*TBitmapSource* , 24),
(*TBraun_Psd* , 48),
(*TC_812* , 22),
(*TC_812GPIB* , 17),
(*TC_812ISA* , 2),
(*TC_832* , 25),
(*TCalibrate* , 5),
(*TCalibratePsd* , 8),
(*TChooseDeviceCmd* , 1),
(*TCmd* , 2),

(*TCommonDevParam* , 4),
(*TCounterShowParam* , 2),
(*TCounterWindow* , 3),
(*TCurve* , 2),
(*TDC_Drive* , 5),
(*TDevice* , 17),
(*TDList* , 1),
(*TEditWindow* , 7),
(*TEncoder* , 4),
(*TExecuteCmd* , 2),
(*TGenericDevice* , 9),
(*TMacroExecute* , 5),
(*TMain* 22),
(*TMDIWindow* 10),
(*TMList* 2),
(*TModalDlg* 4),
(*TModelessDlg* 3),
(*TMotor* 10),
(*TMotorParam* 5),
(*TOptimizeDC_812* 12),
(*TPlotData* 8),
(*TPosControl* 7),
(*TPsd* 11),
(*TPsdRemoteSync* 4),
(*TRadicon* 16),
(*TScan* 17),
(*TScanCmd* 6),
(*TScsParameters* 1),
(*TSetAdjustmentParam* 1),
(*TSetupAreaScan* 3),
(*TSetupContinuousScan* 4),
(*TSetupScanCmd* 1),
(*TSetupStepScan* 6),
(*TShowValueCmd* 3),
(*TSteering* 9),
(*TStoe_Psd* 16),
(*TTopographyExecute* 8),
(*Tx* 441))

$$VKG_{SYS} = VKG_{SU} + VKG_{HU} = \frac{57}{81} = 0,703$$

$$SYS = (PAN_{SYS}, KOMP_{SYS}, VKG_{SYS}) = (956, \quad KOMP_{SYS}, \quad 0,703)$$

F 4 Maßkomponente ’Komplexität’

	VWMC			EWMC		DIT			VK				PA				
Class Name	sum v(G)	avg v(G)	max v(G)	max ev(G)	NOC	Depth	RFC	WMC	RFC-WMC	CBO	LOCM	Parents	PubDataAcc	PubData%	DepOnChild	Kopplungsmechanismus	Kommunikationsart
TAbout	6	1,5	3	1	0	2	10	4	6	0	0	1	0	0	FALSE	8	0,6
TAdjustmentExecute	33	4,71	17	1	0	2	13	7	6	3	73	1	0	0	FALSE	9,5	1,65
TAdjustmentParameter	1	1	1	1	0	1	1	1	0	1	0	0	0	100	FALSE	0,5	0,65
TAdjustmentWindow	14	1,27	2	1	0	3	61	13	48	7	87	1	0	0	FALSE	54,5	3,35
TAm9513a	58	2,42	19	8	1	1	24	24	0	0	90	0	0	11	FALSE	12	0,3
TAngleControl	76	8,44	45	6	0	2	13	9	4	4	63	1	0	0	FALSE	8,5	2
TAquisition	11	2,2	7	7	0	2	11	5	6	0	100	1	0	0	FALSE	8,5	0,6
TAreaScan	307	8,52	45	30	0	3	72	38	34	14	90	2	0	11	FALSE	53	5,8
TAreaScanCmd	10	2	3	1	0	2	15	5	10	4	64	1	0	0	FALSE	12,5	2
TAreaScanParameters	11	5,5	10	1	1	1	2	2	0	2	51	0	0	100	FALSE	1	1
TBitmapSource	157	7,14	30	13	0	1	23	23	0	3	83	0	0	31	FALSE	11,5	1,35
TBraun_Psd	105	5,83	19	15	0	3	64	18	46	12	86	1	0	11	FALSE	55	5,1
TBurleigh	1	1	1	1	0	3	49	1	48	1	100	1	0	0	FALSE	48,5	1,25
TCalculateCmd	9	4,5	7	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TCalibrate	63	9	45	20	0	2	12	7	5	2	59	1	0	0	FALSE	8,5	1,3
TCalibratePsd	45	5	23	4	0	2	13	9	4	3	75	1	0	0	FALSE	8,5	1,65
TChooseAxisCmd	5	2,5	3	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TChooseDeviceCmd	8	4	5	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TChooseScan	14	2,33	7	1	0	2	16	6	10	4	56	1	0	0	FALSE	13	2
TCmd	26	1,73	7	1	16	1	15	15	0	1	86	0	0	67	FALSE	7,5	0,65
TCommonDevParam	28	5,6	17	1	0	2	11	5	6	2	40	1	0	0	FALSE	8,5	1,3
TControlFlankCmd	20	3,33	13	6	0	2	16	6	10	1	63	1	0	0	FALSE	13	0,95
TCounterShowParam	10	2,5	7	6	0	2	11	4	7	1	25	1	0	0	FALSE	9	0,95
TCounterWindow	36	2,4	11	1	0	2	53	15	38	2	81	1	0	25	FALSE	45,5	1,3
TCurve	149	5,73	14	12	0	1	27	27	0	0	72	0	0	9	FALSE	13,5	0,3
TCurveShowParam	27	5,4	15	1	0	2	11	5	6	3	46	1	0	0	FALSE	8,5	1,65
TC_812	87	2,81	8	6	2	3	67	33	34	15	80	1	0	100	FALSE	50,5	6,15

TC_812GPIB	20	4	11	8	0	4	63	6	57	17	50	1	0	0	FALSE	60	7,15
TC_812ISA	44	4,4	17	15	0	4	64	10	54	18	71	1	0	0	FALSE	59	7,5
TC_832	62	2,14	13	10	0	3	63	29	34	13	92	1	0	42	FALSE	48,5	5,45
TDataBase	17	1,42	3	1	0	1	12	12	0	1	73	0	0	0	FALSE	6	0,65
TDC_Drive	43	1,59	6	6	2	2	60	27	33	3	97	1	0	80	FALSE	46,5	1,65
TDevice	74	1,57	8	3	4	1	47	47	0	0	94	0	0	94	FALSE	23,5	0,3
TDList	63	3,71	18	6	0	1	17	17	0	3	78	0	0	0	FALSE	8,5	1,35
TEditWindow	28	1,47	4	3	0	2	58	19	39	3	80	1	0	100	FALSE	48,5	1,65
TEncoder	13	1,86	5	1	0	2	48	7	41	1	100	1	0	0	FALSE	44,5	0,95
TExecuteCmd	9	2,25	6	1	0	2	10	4	6	0	75	1	0	100	FALSE	8	0,6
TGenericDevice	26	2,89	12	3	0	2	70	9	61	3	0	2	0	0	FALSE	65,5	1,65
TGetData	3	1	1	1	0	2	10	3	7	0	16	1	0	0	FALSE	8,5	0,6
TGotoIntensityCmd	20	3,33	7	1	0	2	16	6	10	1	35	1	0	0	FALSE	13	0,95
TGotoPeakCmd	20	3,33	10	8	0	2	16	6	10	1	56	1	0	0	FALSE	13	0,95
TInquireCmd	3	1,5	2	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TLoadPointCmd	6	3	4	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TMacroExecute	39	7,8	27	7	0	2	17	5	12	3	70	1	0	0	FALSE	14,5	1,65
TMain	87	6,21	25	1	0	1	14	14	0	1	90	0	0	100	FALSE	7	0,65
TMDIWindow	71	1,42	9	6	3	1	51	51	0	0	95	0	0	89	FALSE	25,5	0,3
TMeasurementParam	4	1	1	1	0	2	11	4	7	0	0	1	0	0	FALSE	9	0,6
TMList	68	4,25	37	34	0	1	16	16	0	2	90	0	0	75	FALSE	8	1
TModalDlg	18	1,5	5	1	24	1	12	12	0	0	86	0	0	0	FALSE	6	0,3
TModelessDlg	21	1,17	4	1	3	1	18	18	0	0	94	0	0	17	FALSE	9	0,3
TMotor	135	2,6	32	8	2	1	54	54	0	1	95	0	0	75	FALSE	27	0,65
TMotorParam	28	5,6	12	1	0	2	11	5	6	3	75	1	0	0	FALSE	8,5	1,65
TMoveToPointCmd	9	3	6	1	0	2	15	3	12	1	0	1	0	0	FALSE	13,5	0,95
TOptimizeDC_812	8	1,6	3	1	0	2	11	5	6	4	50	1	0	0	FALSE	8,5	2
TOptimizeDC_832	8	1,6	3	1	0	2	11	5	6	4	51	1	0	0	FALSE	8,5	2
TPiezoDrive	1	1	1	1	1	2	48	1	47	1	0	1	0	0	FALSE	47,5	0,95
TPlotData	105	4,38	30	3	3	2	58	24	34	3	84	1	0	71	FALSE	46	1,65
TPosControl	30	3,75	13	4	0	2	13	8	5	2	65	1	0	0	FALSE	9	1,3
TPsd	86	2,25	12	8	2	2	58	38	20	8	94	1	0	73	FALSE	39	3,4

TPsdParameters	16	3,2	9	1	0	2	17	5	12	5	70	1	0	0	FALSE	14,5	2,35
TPsdRemoteSync	41	3,15	12	1	0	2	18	13	5	3	77	1	0	0	FALSE	11,5	1,65
TRadicon	73	6,08	14	12	0	2	51	13	38	4	69	1	0	100	FALSE	44,5	2
TSaveDataCmd	12	6	7	1	0	2	15	2	13	5	0	1	0	0	FALSE	14	2,35
TScan	104	4,51	22	22	0	3	69	27	42	11	96	2	0	0	FALSE	55,5	4,75
TScanCmd	25	5	9	4	0	2	15	5	10	4	73	1	0	0	FALSE	12,5	2
TScanParameters	6	6	6	3	1	1	1	1	0	1	0	0	0	100	FALSE	0,5	0,65
TScsParameters	14	2,8	6	3	0	2	11	5	6	2	40	1	0	0	FALSE	8,5	1,3
TSetAdjustmentParam	8	1,13	2	1	0	2	13	7	6	0	100	1	0	100	FALSE	9,5	0,6
TSetFileNameCmd	8	4	5	1	0	2	15	2	13	2	50	1	0	0	FALSE	14	1,3
TSetupAreaScan	39	7,8	23	5	0	2	11	5	6	5	50	1	0	0	FALSE	8,5	2,35
TSetupContinuousScan	22	4,4	13	1	0	2	11	5	6	2	67	1	0	0	FALSE	8,5	1,3
TSetupScanCmd	7	3,5	4	1	0	2	15	2	13	3	50	1	0	0	FALSE	14	1,65
TSetupStepScan	45	9	25	6	0	2	11	5	6	2	68	1	0	0	FALSE	8,5	
TSetWidthCmd	3	1,5	2	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TShowValueCmd	6	3	4	1	0	2	15	2	13	1	0	1	0	0	FALSE	14	0,95
TSteering	250	6,1	70	24	0	1	42	42	0	3	94	0	0	68	FALSE	21	1,35
TStoe_Psd	32	2,91	10	5	0	3	59	11	48	12	81	1	0	0	FALSE	53,5	5,1
TTopography	2	1	1	1	0	1	2	2	0	1	50	0	0	100	FALSE	1	0,65
TTopographyExecute	52	10,4	38	7	0	2	10	5	5	4	46	1	0	0	FALSE	7,5	2
TTopographySetParam	34	6,8	14	1	0	2	11	5	6	1	60	1	0	13	FALSE	8,5	0,95
----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------	----------
	V(K)			S(K)													
Class Name	sum v(G)	avg v(G)	max v(G)	max ev(G)	NOC	Depth	RFC	WMC	RFC-WMC	CBO	LOCM	Parents	PubDataAcc	PubData%	DepOnChild	KP	DK
Total:	3285	295,23	1027	398	65	177	2084	923	1161	259	4667	65	0	3662		1622,5	11,95
Average:	31,89	2,87	9,97	3,86	0,63	1,72	20,23	8,96	11,27	2,51	45,31	0,63	0	35,55			
Maximum:	307	10,4	70	34	24	4	72	54	61	18	100	2	0	100			
Minimum:	1	1	1	1	0	1	0	0	0	0	0	0	0	0			

Rows in Report: 103

Tabelle F7: Meßergebnisse des McCabe-Toolsets für die Komplexitätskomponente

F 4.1 Ermittlung Maßkomponente 'Schnittstellenkomplexität'

$$KP_1 = 0,5 \cdot RFC_1 + 0,5 \cdot VK_1 = 0,5 \cdot 10 + 0,5 \cdot 6 = 8$$

...

$$KP_{80} = 0,5 \cdot RFC_{80} + 0,5 \cdot VK_{80} = 0,5 \cdot 11 + 0,5 \cdot 6 = 8,5$$

$$KP = \sum_{i=1}^{k} KP_i = 1622,5$$

$$DK_1 = 0,35 \cdot CBO_1 + 0,35 \cdot PA_1 + 0,3 \cdot DIT_1 = 0,35 \cdot 0 + 0,35 \cdot 0 + 0,3 \cdot 2 = 0,6$$

...

$$DK_{80} = 0,35 \cdot CBO_{80} + 0,35 \cdot PA_{80} + 0,3 \cdot DIT_{80} = 0,35 \cdot 1 + 0,35 \cdot 0 + 0,3 \cdot 2 = 0,95$$

$$DK = \sum_{i=1}^{k} DK_i = 11,95$$

$$SK = (KP, DK) = (1622,5 \ , \ 11,95)$$

$$SK_{aggr} = s_1 \cdot KP, s_2 \cdot DK = 0,5 \cdot 1622,5 + 0,5 \cdot 11,95 = 817,23$$

F 4.2 Ermittlung Maßkomponente 'Interne Komponentenkomplexität'

$$VWMC_1 = \sum_{j=1}^{n} v(M_j) = 6$$

...

$$VWMC_{80} = \sum_{j=1}^{n} v(M_j) = 34$$

$$V(K) = \sum_{i=1}^{k} VWMC_i = 3285$$

$$EWMC_1 := Max_1(ev(M_{11}), ..., ev(M_{1j})) = 1$$

...

$$EWMC_{80} := Max_1(ev(M_{801}), ..., ev(M_{1j})) = 1$$

$$S(K) = \sum_{i=1}^{k} EWMC_i = 398$$

$$IKK = (V(K), S(K)) = (3285 \ , \ 398)$$

$$IKK_{aggr} = k_1 \cdot V(K) + k_2 \cdot S(K) = 0,5 \cdot 3285 + 0,5 \cdot 398 = 1841,5$$

F 4.3 Ermittlung Maßkomponente 'Komplexität'

$$KPK = \frac{KP}{Anzahl\ der\ Kopplungen\ aller\ Komponenten} = \frac{KP}{RFC + VK} = \frac{1622,5}{2048 + 1161} = 0,506$$

$$DKK = \frac{DK}{Anzahl\ aller\ Datenkopplungen} = \frac{DK}{CBO + PA + DIT} = \frac{11,95}{259 + 0 + 177} = 0,027$$

$$DLK = \overline{V(K)} = 295,23$$

Anhang G - Porting XCTL From Borland (16-Bit) to Visual C++ 6.0 (32 Bit)

The procedure for migrating XCTL project from Borland to Visual C++ consists
of the following steps:

- Create new Visual C++ projects, described in Section 1.
- Update project settings as described in Section 2.
- Make neccessary changes to source files as described in Section 3.
- Make neccessary changes to resources as described in Section 4.
- Build all executable files (exe and dlls) - Project/Rebuild all.
  If there are any errors left, these errors can guide you to make more changes
  to source files.
- Once the build is successful, copy other neccessary files to output
  directory (Debug directory under the project root for debug build, which
  is active by default), such as other dlls, *.ini, *.mak, *.hlp. *.dat).
- You should now be able to execute program. I tested it only for
  basic operations.

1. Creating Visual C++ projects

1) Create new Visual C++ Project (File/New choose Win32 Application,
   then choose empty project). Name this project XControl (it is for
   the main program).
2) Add projects for each dll in the same workspace. (Projects/Add to project/New/Project
   choose Win32 dll then choose empty project). Repeat this step for each dll.
   At any time, you can set any one of these projects as an active project
   by right clicking on its name in the workspace window. The active project
   name is shown in bold letters.
3) Define project interdependecies. In that way, VC++ knows how to build the
   whole app. Use Project/Dependencies. XControl depends on motors, counters and splib.
   Counters depends on motors and splib. Motors depend on splib. Splib is not dependent of
   anything.
4) Copy all source files (*.h, *.cpp, *.c, *rc) to the project root directory
   (for the dlls also in the same directory).
5) Add these files to corresponding projects (for *.h files this is not neccessary,
   compiler itself will place them under "external dependencies" group).

2. Updating project settings

Project Settings for splib
- General tab: set Output files to Debug (leave intermediate files in separate folder)
- C/C++ tab: add to defines: Build_Library
- C/C++ tab: make sure there is option /MD (for release) or /MDd (for debug)

```
             (eliminate /MT or /ML)

Project Settings for motors
- General tab: set Output files to Debug
- Link tab, add Debug\splib.lib to library modules
- to eliminate "unresolved external timeSetEvent" add winmm.lib to library modules
- C/C++ tab: add to defines: Build_Motors, Use_Library
- C/C++ tab: make sure there is option /MD (for release) or /MDd (for debug)
             (eliminate /MT or /ML)

Project Settings for counters
- General tab: set Output files to Debug
- Link tab, add winmm.lib, Debug\splib.lib and Debug\motors.lib to library modules
- C/C++ tab: add to defines: Build_Counters, Use_Library
- C/C++ tab: make sure there is option /MD (for release) or /MDd (for debug)
             (eliminate /MT or /ML)

Project Settings for XControl
- Link tab, add winmm.lib, Debug\splib.lib, Debug\counters.lib and Debug\motors.lib to library
  modules
- C/C++ tab: add defines Use_Counters, Use_Motors, Use_Library, Use_ImageCCD
- C/C++ tab: make sure there is option /MD (for release) or /MDd (for debug)
             (eliminate /MT or /ML)

3. Changes to source files
--------------------------

all files where applicable (use Edit/Find in files):
- predefined macro __WIN32__ should be renamed to _WIN32
- change _export to __declspec(dllexport). Be sure that this new string
  is places in front of the words WINAPI or CALLBACK.
  In few places, word _export is just deleted. That is
  mentioned elsewhere in this document
- delete all words __rtti
- rename constants MAXDIR MAXFILE MAXPATH MAXEXT MAXDRIVE to _MAX_DIR _MAX_FNAME MAX_PATH
  _MAX_EXT _MAX_DRIVE respectively
- change fnsplit to _splitpath
- change fnmerge to _makepath
- change chdir to _chdir and add #include <direct.h>
- change try { var=new..; A;} catch (xalloc){B;} to
   var = new...;
   if (var)
   { A; }
   else
   { B; }
- you can remove warning "truncation from const double to float" by adding f to the end of
```

```
  float constants e.g. 1.0f
- you can remove warning "truncation from const int to char" by adding cast to the right side,
  e.g. b[i][j] = (char)255;

m_curve.h
- change _export to __declspec(dllexport) in #define _CURVECLASS
- change _import to __declspec(dllimport) in #define _CURVECLASS

m_devcom.h and m_psd.h
- change _export to __declspec(dllexport) in #define _COUNTERCLASS
- change _import to __declspec(dllimport) in #define _COUNTERCLASS

m_dlg.cpp
- introduce variable int edgeTmp; in function TAngleControl::Dlg_OnCommand
- replace line
     SetScrollRange(BarHandle,SB_CTL,GetBarEgde(LEFT),GetBarEgde(RIGHT),TRUE);
  with
     edgeTmp = GetBarEgde(LEFT);
     SetScrollRange(BarHandle,SB_CTL,edgeTmp,GetBarEgde(RIGHT),TRUE);


comhead.h
- comment out #include <cstring.h>, <dir.h> <_defs.h>
- add #include <windows.h> in front of <windowsx.h>
- add #define M_PI        3.14159265358979323846

comclass.h
- delete all words _export

dlg_tpl.cpp
- comment out Ctl3dColorChange(); lines

kpmt1.c
- changed:
     mov dx, rd
       to
     mov dx, WORD PTR rd
  in inline asm code in several places to eliminate error "operand size conflict"
  also for      mov dx,rs      mov ax,d    and       mov i,ax
- eliminated return _AX because it is Borland specific (Visual C++ does
  not need return statement to return with result in AX according to MSDN Power2 sample).

radicon.h
- macro CALLTYPE changed to:
#define CALLTYPE __cdecl
```

```
_Cdecl changed to __cdecl

kisl1.c
- changed:
     mov dx, rd
       to
     mov dx, WORD PTR rd
and similar

l_layer.cpp
- comment out #include <bwcc.h>
- comment out HANDLE hTheModule; line
- change line
    BOOL DllEntryPoint(HANDLE hModule,DWORD dwFunction,LPVOID)
  to
    BOOL WINAPI DllMain(HINSTANCE hModule, DWORD dwFunction, LPVOID  )
- change line
    hTheModule = hModule;
  to
    hModuleInstance = hModule;
- comment out BWCCRegister line
- add ; at the end of line with SearchAgain: label
- replace timeGetTime() with clock()

m_data.cpp
- add cast (float) to MaxInt and MinInt

m_devcom.h
- change __rtti keyword to nothing

m_layer.h
- add:
    #if defined (Build_Motors)
    #define _MOTORCLASS __declspec(dllexport)
    #elif defined(Use_Motors)
    #define _MOTORCLASS __declspec(dllimport)
    #else
    #define _MOTORCLASS
    #endif
- add _MOTORCLASS before the word WINAPI in each declaration
- remove words WINAPI and CALLBACK from lines beginning with typedef

m_layer.cpp
- add _MOTORCLASS in front of WINAPI for each function that
  has _export keyword
- change line
```

```
    BOOL DllEntryPoint(HANDLE hModule,DWORD dwFunction,LPVOID)
  to
    BOOL WINAPI DllMain(HINSTANCE hModule, DWORD dwFunction, LPVOID  )
- comment out BWCCRegister line and after that line add:
    hModuleInstance = hModule;

motors.cpp
- I eliminated word _export; maybe it should be changed to __declspec(dllexport)?
- I also commented out lines with calls to inp() and outp(),
  but under Windows 95 only, _inp and _outp can be used
- I changed some casts in this manner:
     ( FARPROC )  gInitialize = GetProcAddress( hGPIBModule,"IEEE488_INITIALIZE" );
    to
     gInitialize = (TInitialize) GetProcAddress( hGPIBModule,"IEEE488_INITIALIZE" );
  Change:
      ( FARPROC )  gSend      = GetProcAddress( hGPIBModule,"IEEE488_SEND" );
  to
     gSend      = (TSend) GetProcAddress( hGPIBModule,"IEEE488_SEND" );
  etc.

c_layer.cpp
- remove #include <bwcc.h> (in all other files also)
- comment out #pragma argsused (disables unknown pragma warning) (in all other files also)
- eliminate call  BWCCRegister (hModule);
- change _export to __declspec(dllexport) . This should be placed in front
  of WINAPI or CALLBACK words (the order is important here)
- add extern "C"  to FakeDevice and pTime declarations. Add dummy initializer = "" to pTime
  declaration
- because 32bit dlls behave differently during load, you need to
  guard LibMain and WEP functions with #ifndef _WIN32. In the #else part of
  this guard, add:
    BOOL WINAPI DllMain(HINSTANCE hModule, DWORD dwFunction, LPVOID  )
    {
          switch (dwFunction)
          {
            case DLL_PROCESS_ATTACH:
              bModulLoaded = FALSE;
              hModuleInstance = hModule;
//            BWCCRegister (hModule);
              lpDList = ( LPDList ) new TDList( nMaxDeviceAllowed, hModule );
              hMotorDll = GetModuleHandle( "motors.dll" );
            break;
          };
          return TRUE;
    }
```

```
c_layer.h
- add __declspec(dllexport) in front of WINAPI and CALLBACK words

counters.cpp
- introduce local int variable _AX and insert mov _AX, ax at the end of inline asm code
  (apply to each function where _AX is returned from the function)

braunpsd.c
- change ( FARPROC ) cast to ( FARPROC& )
- change struct time to SYSTEMTIME
- change gettime to GetLocalTime
- change t1.ti_sec to (int)t1.wSecond  Do the same for t2.ti_sec

m_justag.cpp
- to eliminate warning on forcing int to bool, for example in some
  statement x = y; where x is of type bool, and y is of type int,
  you can rewrite the statement as x = ( y != 0 );

m_main.cpp
- add #include <direct.h>
- comment out BWCCRegister line
- comment out Ctl3dColorChange line
- change string Borland C++ compiler to Visual C++ compiler
- change __BORLANDC__ to _MSC_VER/2
- introduce variable char _arg[_MAX_PATH] and use GetModuleFileName() API function
   to initialize it instead of using _argv[]
- I commented out the line metioning "device32.dll" because I did not have that dll
- remove word _export from FrameWndProc declaration lines

m_arscan.cpp and m_scan.cpp
- add #include <direct.h>

m_arscan.cpp, m_scan.cpp, m_device.cpp, m_justag.h and m_justag.cpp
- remove declarations of struct date and struct time and introduce SYSTEMTIME st; instead
- change d.da_year d.da_mon d.da_day to (int)st.wYear (int)st.wMonth (int)st.wDay
  (somewhere datum is used instead of d, and zeit instead of t)
- change t.ti_hour t.ti_min ti.ti_sec to (int)st.wHour (int)st.wMinute (int)st.wSecond
- change both getdate and gettime with GetLocalTime

m_mothw.h
- remove word CALLBACK from lpfnLimitWatch declaration line

m_steerng.cpp
- remove word _export from RecallSteering declaration line
- change atof to atoi in line cmp.P1 = (TCParam)atof(p1);
```

```
4. Changes to Resources
-----------------------
main.rc

- ICON, BITMAP, CURSOR resources - delete inline definitions, (all except the first line);
  Open the .rc file in Borland IDE and use copy to clipboard of the resource, and make
   new temporary resource in VC++ with the same attributes, then use export command to save it
   to file. Delete temporary resource, and put the file name in the original
   ICON or BITMAP or CURSOR resource definition line
- to eliminate some "duplicate resource id" warnings, change 1 to -1
   in the corresponding LTEXT lines
- comment out CLASS "bordlg" lines
- change custom bwcc elements to standard controls:
  1) change word CONTROL to GROUPBOX, if there is "BorShade" string in the same line
  1b) change word CONTROL to EDITTEXT, if there is "EDIT" string in the same line
  2) change CONTROL to AUTOCHECKBOX, if there is "BorCheck" string in the same line, and there
     is no BS_3STATE in the line
  3) change CONTROL to AUTO3STATE, if there are "BorCheck" and BS_3STATE in the same line
  4) change word CONTROL to CTEXT, if there is "BorStatic" string in the same line
  5) change word CONTROL to PUSHBUTTON, if there is "BorBtn" string in the same line
  6) change word CONTROL to RADIOBUTTON, if there is "BorRadio" string in the same line
  7) change word CONTROL to PUSHBUTTON, if there is BS_USERBUTTON string in the same line
  8) in these lines, delete the part after the second comma, but do not delete the trailing
     four numbers
     e.g.  change the line
        CONTROL "Belichtungsregelung", -1, "BorShade", BSS_RGROUP | BSS_CAPTION | BSS_CENTER |
        WS_CHILD | WS_VISIBLE, 119, 93, 101, 50
     to
        GROUPBOX "Belichtungsregelung", -1, 119, 93, 101, 50
```