Entwicklung eines Testsystems zur Automatisierung der Qualitätssicherung von Software

Gregor Hartmann

Diplomarbeit

11. März 1999

Fachbereich Informatik
Universität Hamburg

Betreuer: Prof. Dr. Horst Oberquelle
Prof. Dr. Heinz Züllighoven

An dieser Stelle möchte ich mich bei Herrn Prof. Dr. Horst Oberquelle für seine Unterstützung und seine Anregungen während der Realisierung der vorliegenden Diplomarbeit bedanken. Mein Dank gilt auch Herrn Prof. Dr. Heinz Züllighoven für die Zweitbegutachtung dieser Arbeit.

Inhaltsverzeichnis

Verzeichnis der Abbildungen

Verzeichnis der Beispiele

1 Einleitung

In der vorliegenden Arbeit geht es um die Qualitätssicherung einer umfangreichen Software, die im weiteren Anwendung genannt wird. Hierfür soll ein eigenständiges Testsystem erstellt werden. Dabei soll vor allem untersucht werden, inwieweit ein Testsystem die automatische Prüfbarkeit der Benutzungsschnittstelle in Bezug auf Konsistenz und Erwartungskonformität ermöglicht.

Ziel ist das automatisierte Testen der Anwendung auf unterschiedlichen Plattformen und in verschiedenen Landessprachen. Anders als in herkömmlichen Capture/Replay Systemen (vgl. Ostrand, Anodide, Foster, Goradie 1998), bei denen das Werkzeug getrennt von der Anwendung ist, findet bei dem hier verfolgten Ansatz eine Verschmelzung der Anwendung mit dem Testsystem statt. Die Integration eines Controllers in die Anwendung ermöglicht es, die volle Kontrolle über die Anwendung auszuüben, und nicht nur über die Teile der Anwendung, die sie nach außen hin preisgibt. Durch diese Integration kann das Testsystem, unbeeinflußt von den technischen Details der graphischen Benutzungsoberfläche der jeweiligen Plattform, die Anwendung fernsteuern.

Aufgrund der Verschmelzung mit der Anwendung kann auf die Struktur der Anwendung zugegriffen werden. Funktionen zur Diagnose und Hilfsmittel zur Erstellung und Pflege der Testabläufe können somit effizient gestaltet werden. Die dabei gewonnenen Informationen werden in der Entwicklungsumgebung für die Testabläufe aufbereitet und dargestellt. Das Nachvollziehen von Fehlern in Testabläufen wird durch die Entwicklungsumgebung unterstützt, indem ein Protokoll angefertigt wird. Damit wird ein komfortables Testen ermöglicht.

2 Aufgabenstellung

2.1 Motivation

Schon oft habe ich mich über Programme geärgert, deren Bedienung so eigenartig war, daß ich ständig in falschen Menüs gelandet bin, da Tastaturkürzel verwendet wurden, die in anderen Anwendungen oder gar in der selben Anwendung an anderer Stelle anders definiert waren. Oft mußte ich auch lange nach Funktionen suchen, weil sie nicht an der vom Styleguide vorgesehenen Stelle in der Menü- und Dialogstruktur zu finden waren.

Manche Funktionen werden auf verschiedene Weisen zur Verfügung gestellt. Das Kopieren ins Clipboard beispielsweise kann auf mindestens drei verschiedene Arten veranlaßt werden. Es gibt mindestens 2 Tastaturkürzel und die Möglichkeit, über das 'Bearbeiten' Menü zu kopieren. Das Auffinden von Unstimmigkeiten bei alternativen Bedienungsmöglichkeiten in der Benutzungsoberfläche ist nahezu unmöglich, da meist nur eine der zur Verfügung stehenden alternativen Methoden benutzt und damit getestet wird.

Noch unangenehmer ist es, wenn bestimmte Funktionen nur unzulänglich funktionieren oder gar zu Programmabstürzen führen.

Die Suche nach diesen Problemen kann durch ein Testsystem wesentlich vereinfacht und beschleunigt werden, da die Tests nach ihrer einmaligen manuellen Erstellung immer wieder automatisch ablaufen können.

2.2 Generelle Idee

Eine Möglichkeit, interaktive Programme zu testen, ist es, die erstellten Testpläne von einem Benutzer ausführen zu lassen. Da diese Tests aber für jede Version der Anwendung wiederholt werden müssen, ist es weitaus schneller, die Eingaben des Benutzers durch ein Programm simulieren zu lassen. Der Testplan muß dann in geeigneter Weise aufbereitet werden, zum Beispiel in Form eines Skriptes. Das Programm kann dann die in diesem Skript festgelegten Abläufe simulieren.

Um eine exakte Reproduktion des ursprünglichen Benutzerverhaltens zu erzielen, müssen sämtliche Ereignisse wie das Bewegen der Maus, Tastatureingaben und Klicken der Maus an einer willkürlichen Position auf dem Bildschirm, sogenannte Elementarer-

eignisse, in der ursprünglichen Geschwindigkeit wiedergegeben werden. Meistens ist aber eine derart genaue Reproduktion nicht erforderlich. Es ist zum Beispiel oft irrelevant, ob ein Button durch Tastatureingabe oder durch Klicken mit der Maus ausgelöst wurde. Somit können hier mehrere dieser Elementarereignisse zusammengefaßt werden. Es ist dann nur erforderlich, ein Ereignis auf einer höheren Abstraktionsebene, nämlich das Auslösen des Buttons, zu reproduzieren.

Das zeitgenaue Reproduzieren von Elementarereignissen kann außerdem zu Problemen führen. Gibt es zum Beispiel eine Verzögerung in der Ausführung der Anwendung, so kann es passieren, daß ein Bedienelement[1] noch gar nicht am Bildschirm angezeigt wird, aber das Elementarereignis schon ausgeführt wird. Im Beispiel oben würde das bedeuten, daß der Mausklick oder die Tastatureingabe an ganz falscher Stelle ankommt und der Button, zu dem Zeitpunkt an dem er erscheint, von dem Elementarereignis, das ihn auslösen sollte, nicht ausgelöst wird, obwohl der Test schon davon ausgeht, daß er ausgelöst wurde.

Bei der Abstraktion der Ereignisse ist der Empfänger des Ereignisses bekannt, anders als bei den Elementarereignissen. Dadurch wird es möglich, zu prüfen, ob der Empfänger des Ereignisses vorhanden und bereit ist, das Ereignis entgegenzunehmen. Im obigen Beispiel würde also zuerst auf das Erscheinen des Buttons gewartet werden, bevor er ausgelöst wird.

Aber ein Benutzer kann noch mehr, als immer wieder die selben Aktionen auszuführen. Ein Benutzer kann vom Normalfall abweichende Reaktionen der Anwendung erkennen und flexibel darauf reagieren. Für ein Testsystem reicht es also nicht aus, nur die Eingaben eines Benutzers zu reproduzieren. Es muß vielmehr in der Lage sein, auf Abweichungen in der Reaktion der Anwendung zu reagieren. Dies setzt zwei Eigenschaften des Testsystems voraus. Erstens muß es möglich sein, den Zustand der Bedienelemente abzufragen. Zweitens muß der Ablauf des Tests in Abhängigkeit von diesen Zuständen variierbar sein. Beispielsweise können sich in einer Anwendung inhaltliche Teile ändern. Es kann eine neue Auswahlmöglichkeit in einer Liste hinzukommen. Will man einen Test mit allen in dieser Liste zur Verfügung stehenden Varianten durchführen, so muß das Testsystem in der Lage sein, diese Liste erst auszulesen und dann abzuarbeiten. Mit diesen Fähigkeiten des Testsystems wird es dann auch möglich, Reaktionen der Anwendung auf ihre Richtigkeit hin zu überprüfen.

1 Bedienelementen sind alle Elemente der Benutzungsoberfläche, mit denen der Benutzer interagieren kann.

Es kann jedoch auch zu Reaktionen des Programmes kommen, die beim Erstellen des Skriptes unberücksichtigt blieben. Zum Beispiel kann ein Fehlerdialog erscheinen. Ein Testsystem sollte auch in der Lage sein, solche Anomalien festzustellen und angemessen darauf zu reagieren. Ein häufig gewählter Ansatz ist es, das Testskript in einzelne voneinander unabhängige Testfälle zu unterteilen. Tritt dann eine Anomalie auf, so wird der aktuelle Testfall abgebrochen, das Programm in einen definierten Grundzustand gebracht und dann mit dem nächsten Testfall fortgefahren.

All diese Eigenschaften eines Testsystems sind natürlich nur dann sinnvoll, wenn auch ein Protokoll über den Ablauf des Tests erstellt wird, das insbesondere die Anomalien, die während eines Testlaufes auftreten, enthalten muß. Wichtig ist es hier auch, von dem Skript aus in dieses Protokoll schreiben zu können, um Fehlfunktionen, die nicht zu einem Testabbruch führen, sondern durch Abfragen des Zustands der Bedienelemente zutage treten, zu protokollieren.

2.3 Bisherige Lösung bei StarDivision

Bei der Firma Star Division wurde bisher das Testsystem 'QA Partner' der Firma Segue Software Inc. verwendet (vgl. QA Partner, 1996). Mit diesem System war es möglich, ein beliebiges Programm durch Erzeugen von Elementarereignissen zu steuern. Die Elementarereignisse konnten abstrahiert werden, um bestimmte Elemente der Benutzungsoberfläche effizienter und übersichtlicher zu bedienen. Existierten in der Anwendung Bedienelemente, die die vordefinierten Bedienelemente erweiterten, so konnten mit diesem System neue Repräsentationen der Bedienelemente definiert werden. Es konnten, ähnlich wie bei objektorientiertem Ableiten, die vordefinierten Bedienelemente um eigene Zugriffsmethoden erweitert werden. Die tatsächlichen Fähigkeiten des Systems wurden damit aber nicht erweitert.

Die Steuerung der Anwendung erfolgte durch eine Skriptsprache. Mehrere dieser Skripten konnten dann in einem Testlauf zusammengefaßt werden. Die Ergebnisse dieser Testläufe wurden protokolliert und dem Benutzer des Testsystems zur Verfügung gestellt.

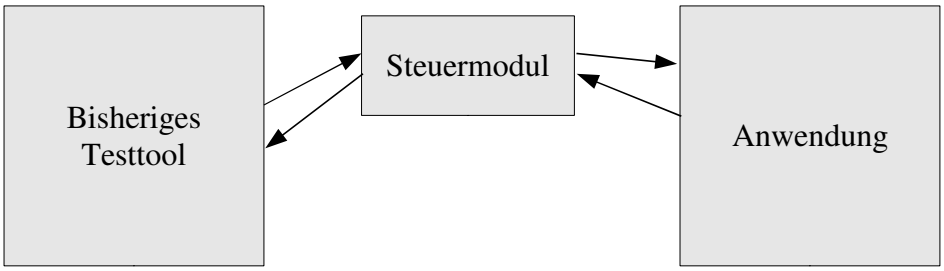

Abbildung 1: QA Partner - Übersicht

Wie Abbildung 1 zeigt, war das System in drei Module unterteilt. Das größte Modul war die Entwicklungsumgebung zum Entwerfen und Ausführen der Skripten. Zur Kommunikation mit der Anwendung wurde ein kleines Zusatzmodul verwendet. Da es bei der Steuerung der Anwendung gelegentlich zu Verklemmungen kam, mußte dann nur dieses Zusatzmodul neu gestartet werden und nicht die ganze Entwicklungsumgebung. Das dritte Modul gehört eigentlich nicht zu dem Testsystem, da es sich um die Anwendung selbst handelt.

Dem Anspruch, auf unterschiedlichen Plattformen testen zu können, wurde dieses Testsystem dadurch gerecht, daß für verschiedene Plattformen Programmversionen verfügbar waren.

2.4 Einschränkungen der bisherigen Lösung

Ein entscheidender Nachteil dieses Testsystems war es, daß vom Betriebssystem nur eine begrenzte Anzahl von Bedienelementen zur Verfügung gestellt wurde. Neue Funktionalität, die bei Weiterentwicklungen der dem Testsystem bekannten Standardbedienelemente benötigt wurde, konnte nur über Elementarereignisse angesteuert werden. Ein gravierender Nachteil davon war es, daß der Zustand der dem Testsystem unbekannten Bedienelemente nicht oder nur schwer und fehlerträchtig abgefragt werden konnte und somit zur Kontrolle von Fehlern und zum Steuern des Skriptes nicht zur Verfügung stand. Zudem stellte jede Plattform andere Bedienelemente zur Verfügung, deren Funktionsumfang nicht auf allen Plattformen identisch war, so daß die Testskripte nur bedingt plattformunabhängig waren. So konnte zum Beispiel der Inhalt einer ListBox unter Microsoft Windows 95 ausgelesen werden, nicht aber unter OS/2.

Anhand eines Beispiels sollen diese Probleme verdeutlicht werden. Bei dem hierfür verwendeten Bedienelement handelt es sich um eine *Toolbox*, auch bekannt als Symbolleiste. Dieses Bedienelement enthält mehrere Buttons, mit denen verschiedene Funk-

tionen ausgelöst werden können. Dieses Bedienelement wird von Microsoft Windows nicht zur Verfügung gestellt.

Bei einer Toolbox sind zwei Funktionen wichtig. Erstens muß ein Button ausgelöst werden können, und zweitens müssen seine Existenz und Position abgefragt werden können.

Da das Bedienelement unter Microsoft Windows unbekannt ist, hat das Testsystem die Sicht wie in Abbildung 2b auf die Toolbox. Es stehen dem Testsystem daher nur Elementarereignisse zum Steuern zur Verfügung. Das Auslösen eines Buttons wurde also durch eine Methode realisiert, der die Position des gewünschten Buttons sowie die Anzahl der Zwischenräume links davon, als Parameter übergeben wurde. Die Methode berechnete dann die Position des Buttons anhand der Konstanten d1 und d2 aus Abbildung 2a. Die Konstante d1 enthält die Breite eines Buttons, d2 enthält die Breite eines Zwischenraumes. Die Methode erzeugte dann ein Elementarereignis, das an die entsprechende Stelle auf dem Bildschirm klickte. Das Abfragen der Existenz eines Buttons erfolgte über den Vergleich einer gespeicherten Bitmap mit einem entsprechenden Ausschnitt des Bildschirms, wobei der Bildschirmausschnitt wieder mit den entsprechenden Konstanten berechnet wurde.

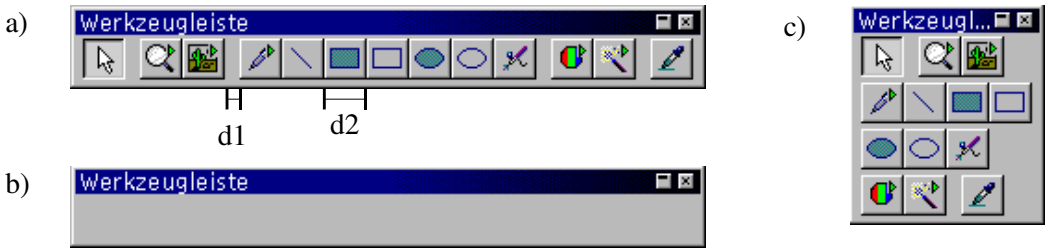

Abbildung 2: Zugriff auf eine Toolbox

a) Toolbox wie sie am Bildschirm erscheint, mit Konstanten d1 und d2
b) Toolbox wie sie dem Testsystem QAPartner erscheint
c) Andere Proportionierung der Toolbox

Probleme traten bei dieser Vorgehensweise auf, sobald sich die Proportionen, wie in Abbildung 2c gezeigt, änderten. Auch wenn zum Beispiel die Größe der Buttons oder der Zwischenräume geändert oder ein Button eingefügt wurde, so verschoben sich die weiter rechts liegenden Buttons entsprechend. Die Klicks wurden dann an der falschen Stelle erzeugt und lösten den dann dort befindlichen Button aus.

Ähnliche Probleme traten beim Überprüfen der Buttons durch Bitmapvergleiche auf.

Die selben Probleme bei der Anpassung an bestimmte GUI-Bibliotheken beschreiben auch Strauß & Hörnke (1998) bei der Arbeit mit dem Testsystem WinRunner. Sie

schließen: »Diese hier benannten Probleme zum Zusammenspiel mit GUI-Buildern verschiedener Hersteller können je nach Testwerkzeug unterschiedlich ausfallen, hier ist jedoch mehr der verwendete GUI-Builder und weniger das jeweilige Testwerkzeug ausschlaggebend. Die Anpassung des Testwerkzeuges an die verschiedenen GUI-Builder durch selbstgeschriebene (WinRunner-) Funktionen ist häufig möglich, kann aber zeitaufwendig sein!«

Neben diesen Problemen bei der Anpassung an die Bedienelemente, die von der Anwendung implementiert werden, existierten noch weitere Gründe, die gegen eine weitere Verwendung dieses Testsystems sprachen.

Durch die Notwendigkeit, für jede Plattform, auf die die Anwendung portiert wurde, auch eine zugehörige Version des Testsystems zu kaufen, entstanden hohe Kosten. Insbesondere, da für weniger verbreitete Plattformen aufgrund des geringen allgemeinen Interesses entsprechend hohe Preise zu zahlen gewesen wären.

Ein weiterer Nachteil war außerdem die Abhängigkeit von einer anderen Firma, die sich daraus ergab, daß das Testsystem natürlicherweise nicht auf jeder existierenden Plattform verfügbar war. Die Möglichkeit, die Anwendung auf einer spezifischen Plattform zu testen, war also davon abhängig, ob die Fremdfirma ein Testsystem zur Verfügung stellte. Diese Abhängigkeit galt auch für Änderungen, die am Testsystem vorgenommen werden sollten. Eine Durchführung war zusätzlich mit langen Entwicklungszyklen und hohen Kosten verbunden.

2.5 Aufgabenstellung - Forderungen an das neue Testsystem

Bei Star Division existiert eine in C++ programmierte Klassenbibliothek. Bei der Entwicklung der zu testenden Anwendung wird diese Bibliothek verwendet. Sie wurde auf mehrere Plattformen portiert und stellt alle notwendigen Funktionen des Betriebssystems zur Verfügung. Ausgehend von dieser plattformunabhängigen Klassenbibliothek soll ein Mechanismus implementiert werden, der es erlaubt, alle Bereiche der Anwendung, insbesondere alle Bedienelemente, anzusprechen und deren Funktionalität und Bedienbarkeit zu testen. Ziel ist es, verschiedene Aspekte einer Anwendung zu testen. Im einfachsten Fall soll überprüft werden, ob das Programm wenigstens soweit zu benutzen ist, daß alle Dialoge angezeigt und wieder geschlossen werden können, ohne daß es zu einem Programmabbruch kommt. Anspruchsvoller sind Aufgaben wie das Testen von gemeinsamen Basisfunktionalitäten an Dialogen, wie zum Beispiel das

Schließen und Abbrechen über Tastatur oder mit der Maus. Aber auch komplexe Probleme, wie sie beim Testen der Funktionalität der Anwendung entstehen, sollen bewältigt werden. Daraus ergeben sich vier Arten von Methoden, um den Zustand der Anwendung zu erfassen und zu kontrollieren.

1. Die Bedienelemente müssen vom Testsystem vollständig manipuliert werden können. Es müssen alle Interaktionsweisen, die auch dem Benutzer zur Verfügung stehen, ausführbar sein.

2. Um die Ergebnisse zu überprüfen, müssen Funktionen existieren, die das Abfragen des Zustandes einzelner Bedienelemente erlauben.

3. Um die Handhabung der Anwendung zu erleichtern und zu beschleunigen, werden Befehle zum direkten Aufrufen der Funktionalität der Anwendung benötigt. Es soll damit eine Alternative zum Navigieren durch Menüs und Dialoge geboten werden.

4. Zusätzliche Befehle zur Steuerung des Verhaltens des Testsystems beziehungsweise zum Aufruf von Sonderfunktionen und Diagnosefunktionen an der Anwendung werden benötigt. Wichtig sind hier Befehle zum Rücksetzen der Anwendung in einen definierten Anfangszustand und zum Aktivieren der Programmierhilfen für die Testabläufe.

Um von Instabilitäten der sich ständig in Entwicklung befindlichen Anwendung unabhängig zu sein, soll das Testsystem in zwei Teile aufgeteilt werden. Abbildung 3 zeigt die Aufteilung in zwei separate Programme. Zum einen die Anwendung, die getestet werden soll und zum anderen eine eigenständige Applikation, das Testtool.

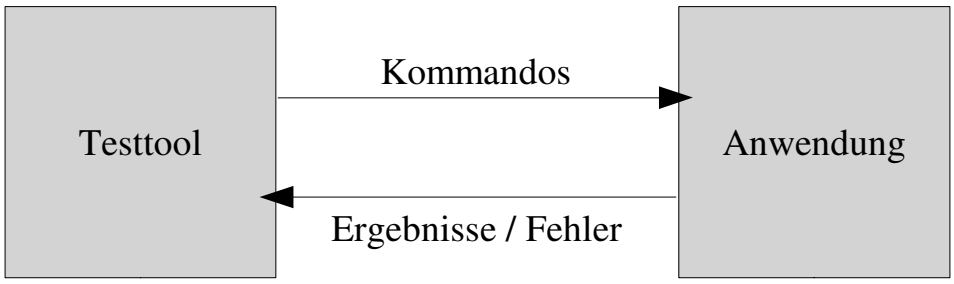

Abbildung 3: Neues Testsystem - Übersicht

Ein Absturz der Anwendung soll erkannt werden, und es sollen geeignete Maßnahmen eingeleitet werden, um den Test dennoch fortführen zu können. Hierfür muß das Testtool in der Lage sein, die Anwendung erneut zu starten.

Das Erstellen der Skripte für die Testabläufe soll in besonderem Maße unterstützt werden. Außerdem ist eine effiziente und übersichtliche Darstellung und Auswertung der gewonnenen Ergebnisse, und im besonderen der dabei aufgetretenen Fehler, in einem Journal zu ermöglichen.

Die Belastung für die Anwendung sowie die Entwicklung soll möglichst gering gehalten werden. Unter der Belastung für die Anwendung ist sowohl zusätzlicher Speicherbedarf als auch eine negative Beeinflussung des Laufzeitverhaltens zu verstehen. Die Belastung für die Entwicklung entsteht in diesem Zusammenhang aus zusätzlichen Implementierungen, die für das Testsystem an der Anwendung vorgenommen werden müssen und nicht direkt das Testsystem betreffen.

Es sollen bereits vorhandene Mechanismen genutzt werden.

3 Grundlagen des automatischen Testens

Um sicherzustellen, daß ein Programm funktioniert, gibt es verschiedene Ansätze, die in unterschiedlichen Entwicklungsstadien des Programmes angewendet werden können. Zum einen gibt es theoretische Ansätze, die versuchen, durch mathematische Beweisverfahren das korrekte Verhalten des Programmes zu beweisen. Ein Vertreter dieses Ansatzes ist Hoare (1969). Sein Verfahren beruht darauf, daß jedes Stück Quellcode garantiert, daß es, bei gültigen Eingangsdaten, auch gültige Ausgangsdaten erzeugt. Dazu werden zu dem Quellcode Vorbedingungen (engl.: preconditions) und Nachbedingungen (engl.: postconditions) derart angegeben, daß, wenn die Eingangsdaten die preconditions erfüllen, auch die Ausgangsdaten die postconditions erfüllen. Es entstehen Tripel der Form

$$\{P\}S\{Q\}$$

Wobei P und S precondition und postcondition sind und S das Stück Quellcode ist. Bei diesem Vorwärtsbeweis muß, um die Korrektheit eines Programmes zu beweisen, für jedes Stück Quellcode bewiesen werden, daß seine Precondition sich aus der Postcondition des bereits ausgeführten Codes ableiten läßt. Nach Hintereinanderausführung von Anweisungen S1, ... , Sn ist Pn durch Qn-1 gegeben. Qi enthält neben den essentiellen Bedingungen auch "Nebensächlichkeiten", so daß eine Beweisfindung erschwert wird. Daher ist das Durchführen dieser Beweise zeitaufwendig, es gibt allerdings automatische Beweissysteme, wie das auch an der Universität Hamburg verwendete LEGO von Randy Pollack(1999), die dem Entwickler einen Großteil der Arbeit abnehmen können, indem sie die meisten Beweisschritte selbst ermitteln.

Neben den theoretischen Verfahren existieren Verfahren, die auf einer informalen Vorgehensweise beruhen. Diese Verfahren sind nicht computerbasiert, sondern sie beinhalten, daß ein Team von Entwicklern den Programmcode liest und analysiert. Myers (1979) spricht auch von »human testing«. Repräsentanten sind der Walkthrough und die Codeinspektion. Beim Walkthrough wird der Code vom Entwickler schrittweise erläutert und nach bestimmten vorher festgelegten Kriterien überprüft. Diese Kriterien umfassen Konsistenz mit der Spezifikation, Programmierrichtlinien und häufige Programmierfehler. Bei der Codeinspektion werden die einzelnen Anweisungen anhand einfacher Testdaten überprüft und entdeckte Fehler soweit möglich sofort korrigiert.

Computerbasierte Testverfahren basieren darauf, daß ein Programm oder ein Teil davon ausgeführt wird. Diese Verfahren arbeiten nach folgendem Schema:

1. Auswahl von Testdaten
2. Ausführen des Tests
3. Vergleichen der Ergebnisse mit der Spezifikation

Der Umfang des getesteten Codes variiert dabei und wird nach seinem Umfang in folgende drei Kategorien unterteilt:

1. Unit-Test
2. Integrationstest
3. Systemtest

Der Begriff der Unit ist dabei nicht scharf definiert. Jorgensen (1994) führt u.a. folgende Definitionen an:

- Eine einzelne Funktion
- Eine Funktion, deren Programmcode auf eine Seite paßt
- Das kleinste eigenständig compilierbare Codesegment
- Die Menge Code, die in 4 - 40 Stunden geschrieben werden kann
- Quellcode, der einer Person zugewiesen ist
- Quellcode, den eine Person in drei Monaten entwirft, codiert und testet

Interessanterweise sei der Begriff Unit in vielen Organisationen nicht exakt definiert. Jedenfalls wird eine Unit getrennt getestet, wobei fehlende Komponenten durch sogenannte Stubs und Treiber ersetzt werden, die die Eingabe und Ausgabe emulieren. Ziel des Unit-Tests ist es, zu überprüfen, wie die Unit funktioniert, wenn sich alles andere korrekt verhält.

Der Integrationstest fügt die einzelnen Units zusammen. Hierfür können unterschiedliche Vorgehensweisen angewendet werden. Jorgensen (1994) führt folgende an:

- Schrittweises Ersetzen von Stubs und Treibern durch separat getestete Units
- Paarweise Integration aneinander angrenzender Units
- Bottom-up Integration entlang der funktionalen Abhängigkeit

- Top-down Integration entlang der funktionalen Abhängigkeit
- Urknall (big bang) Integration, wobei alle Units auf einmal zusammengefügt werden

Diese Möglichkeiten basieren auf der Annahme, daß korrekte Strukturen und Schnittstellen auch ein korrektes Verhalten der Software garantieren. Diese Annahme gilt aber nicht mehr, wenn der Entwurf fehlerhaft ist. Es ist daher notwendig, einen abschließenden Systemtest durchzuführen. Jorgensen (1994) schlägt vor, beim Systemtest ausschließlich auf die Schnittstelle zuzugreifen, die von der zu testenden Software nach außen zur Verfügung gestellt wird.

In sämtlichen Integrationsstufen kann die Fehlersuche durch in den Quellcode integrierte Prüfmechanismen unterstützt werden. Diese Prüfmechanismen können entweder während des Programmablaufes ausgeführt werden, oder nur zu besonderen Zeitpunkten verwendet werden.

Eine weitere Unterscheidungsmöglichkeit ist anhand des Kriteriums gegeben, ob man den inneren Aufbau eines Programmes kennt oder nur seine Benutzungsoberfläche zur Verfügung hat. Im ersten Falle spricht man von einem White-Box-Test, im zweiten von einem Black-Box-Test.

Der White-Box-Test läßt es zu, mit nur wenigen Testfällen aussagekräftige Tests zu erstellen, da nur besondere Grenzfälle getestet werden müssen, um jeden möglichen Programmfluß zu überprüfen. Die Bestimmung dieser Grenzfälle ist jedoch nicht immer ganz einfach.

Der Black-Box-Test dagegen muß alle möglichen Eingabekombinationen ausprobieren und das vom Programm gelieferte Ergebnis mit dem erwarteten Ergebnis vergleichen. Der Ansatz ist zwar wesentlich einfacher, aber bei komplexen Programmen nicht durchführbar. Deshalb wird man hier mehrere Testläufe durchführen und dann annehmen, daß das Programm die gewünschten Ergebnisse produziert.

Es ist bereits bei einfachen Programmen praktisch kein kompletter Test für alle Eingabedaten möglich. Man betrachte die Multiplikation von zwei ganzen Zahlen im Wertebereich von 16 Bit. Es sind also 65536 * 65536 = 4294967296 Eingabedaten zu berechnen. Mit handelsüblichen Computern kann dieses Problem noch in wenigen Sekunden gelöst werden, Im Fall von 32 Bit Zahlen würde die Berechnung allerdings schon über 10^{11} Jahre dauern. Bei interaktiven Programmen, deren Zustand von vorhergehenden Interaktionen bestimmt ist, erfordert dies nicht nur den Test aller möglichen Interaktionen, sondern auch den Test aller möglicher Abfolgen von Interaktionen. Die

Zahl der Abfolgen von Interaktionen ist jedoch so groß, daß es unmöglich ist, jeden Grenzfall auszutesten, geschweige denn, einen umfassenden Black-Box-Test durchzuführen (vgl. Myers 1979, S.8,9).

Es ist dann nur möglich, typische Abläufe zu testen, sogenannte Use-Cases. Was ein typischer Ablauf ist, muß aber zunächst ermittelt und in einem Testplan festgehalten werden. Dies geschieht zum Beispiel durch Analysieren der vorhandenen Funktionalität. Es wird dann durch Zusammenfügen von einzelnen Schritten ein möglicher Ablauf festgelegt. Eine andere Methode ist das Beobachten von Benutzern, während sie mit der Anwendung arbeiten. Im beiden Fällen müssen die gewonnenen Erkenntnisse in Form eines Skriptes niedergelegt werden.

Laut Myers (1979, S. 5) gibt es beim Testen von Software auch eine psychologische Komponente. Das Suchen von Fehlern wird als eine destruktive Tätigkeit angesehen. Dies führt zu der Sichtweise, daß die Korrektheit, und damit die Abwesenheit von Fehlern, des Programmes nachgewiesen wird. Da es in der Natur des Menschen liegt, zielorientiert zu arbeiten, kann es passieren, daß Testdaten gewählt werden, die kein fehlverhalten des Programmes produzieren. Wird jedoch der Ansatz gewählt, daß der Sinn des Testen darin liegt Fehler zu finden statt Korrektheit zu zeigen, so wird auch eine größere Anzahl von Fehlern gefunden werden.

Ein weiteres Problem entsteht, wenn ein Entwickler seinen eigenen Code testen soll. Die Destruktivität des Testens stellt laut Myers (1979, S. 12f) ein noch stärkeres psychologisches Hindernis dar, da es sich um die eigene Arbeit handelt. Ein weiteres Problem, das auftreten kann, ist, daß ein Programmierer, der die Spezifikation mißverstanden hat und daher einen Fehler gemacht hat, diesen auch nicht entdecken wird, wenn er sein Programm testet. Ähnliche Argumente gelten auch bei einer Firma, so daß auch in diesem Fall das Testen der Software von einer firmenfremden Organisation vorgenommen werden sollte.

3.1 Grundlegende Begriffe

In diesem Abschnitt werden wichtige Grundlagen für das Verständnis der Arbeit erläutert. Zum besseren Verständnis wird zunächst eine kleine Einführung in die Grundlagen der Objektorientierung gegeben. Anschließend wird auf einige wenige Konzepte der Programmierung und der Funktionsweise der Anwendung eingegangen.

3.1.1 Grundlagen des objektorientierten Ansatzes

»Objekte sind die Komponenten des Systems. Ein Objekt hat einen systemweit eindeutigen Namen und einen Zustand , der in einem privaten Speicherbereich des Objekts repräsentiert ist. Dem Objekt sind Operationen zugeordnet, die Informationen über das Objekt liefern und den Zustand des Objektes verändern können. Nur mit diesen Operationen ist es möglich, den privaten Speicherbereich eines Objektes zu lesen oder zu verändern.« (vgl. Kilberth et. al. 1994, S. 9).

»Objekte werden in Klassen beschrieben. Eine Klasse ist das "Erzeugungsmuster" für ihre Objekte, d.h. sie definiert die Eigenschaften von Objekten. Dies umfaßt die den Objekten zugeordneten Operationen und deren interne Realisierung durch Algorithmen und Datenstrukturen« (vgl. Kilberth et. al. 1994, S. 10). Die Zugriffsoperationen werden in der Objektorientierung Methoden genannt. Ein Objekt ist eine Instanz einer Klasse.

Eine Klasse A vererbt ihre Eigenschaften an eine Klasse B, wenn sich Klasse B von A ableitet. B kann dann die Funktionalität von A erweitern, indem es neue Methoden definiert. Im Fall, daß A diese Methoden ebenfalls implementiert, kann das Verhalten bei Aufruf der Methode an A unterschiedlich sein. Wird die Methode durch Voranstellen des Schlüsselwortes *virtual* virtuell definiert, so wird die in B implementierte Methode gerufen. Wird das Schlüsselwort weggelassen, so wird die Methode aus A gerufen. Dieser Mechanismus erlaubt es, Objekte, ohne ihre genaue Ausprägung zu kennen, zu verwenden.

Wird eine Instanz einer Klasse, also ein Objekt, erzeugt, so wird dessen Konstruktor gerufen. Wird das Objekt gelöscht, so wird vorher sein Destruktor gerufen.

Für eine allgemeine Einführung in die Konzepte und Terminologien der Objektorientierten Software-Entwicklung kann auch Schlüter, Behdjati, Fleischer & Bagdon (1990) herangezogen werden. Neuere Erkenntnisse über die Entwicklung von objektorientierter Software ist Züllighoven (1998) zu entnehmen. Eine ausführliche Beschreibung der Programmiersprache C++ findet sich in Stroustrup (1992).

3.1.2 Ressourcen - Die Beschreibung der Benutzungsoberfläche

Die meisten Elemente der Benutzungsoberfläche werden in einer besonderen Sprache definiert. Diese Definition enthält Informationen über Art und Anordnung der Bedienelemente, ihre Beschriftung sowie initiale Werte und Grenzwerte. Sie enthält keine Information über die weitergehende Funktionalität, die nachher in der Anwendung mit den Bedienelementen verknüpft ist. Zusammen mit den deutschsprachigen Texten

stehen hier auch sämtliche Übersetzungen. Diese Definition steht in einer separaten Datei. Um von der Anwendung eingelesen werden zu können, wird die Datei durch einen Ressourcecompiler in ein Binärformat gewandelt.

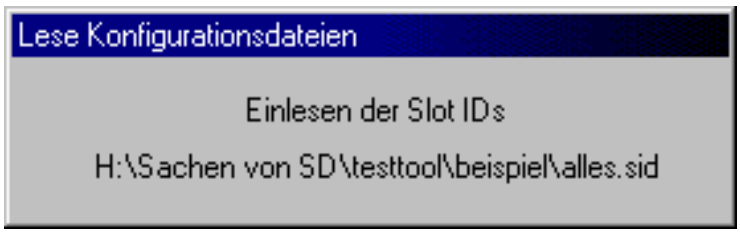

Abbildung 4: Dialog zur Anzeige der aktuell eingelesenen Datei

Die Identifikation der Bedienelemente erfolgt über Zahlen. Zur besseren Lesbarkeit können diesen, wie in Beispiel 1 dargestellt, durch in C++ Syntax formulierte Definitionen symbolische Bezeichner zugewiesen werden. Über diese Bezeichner wird dann

```
/* StarView ressource header file */
#define LOAD_CONF 256
#define WORK 1
#define FILENAME 2
```

Beispiel 1: Datei testtool.hrc

zur Laufzeit auch die Ressource geladen. Der Dialog aus Abbildung 4 beispielsweise, der während des Ladens der Konfiguration angezeigt wird, entsteht durch die in Beispiel 2 dargestellte Ressource. Der Dialog besteht aus einem Floating Window und zwei FixedTexten. Das Floating Window unterscheidet sich dadurch von einem Dialog, daß das übrige Programm bedienbar bleibt. Ein FixedText dient dazu, einen Text darzustellen. Er erlaubt keinerlei Interaktion durch den Benutzer. In der Ressourcedatei werden den einzelnen Bedienelementen Attribute zugewiesen. Im Beispiel wird mit Pos und Size die Position und Größe definiert. Das Makro MAP_APPFONT bestimmt, daß die Angaben relativ zu der Fontgröße der Anwendung definiert werden. Die Größe der Bedienelemente wird dann zur Laufzeit entsprechend angepasst. Wird das Attribut Movable auf TRUE gesetzt, so kann der Dialog auf dem Bildschirm verschoben werden. Das Attribut Center gibt an, daß der dargestellte Text innerhalb des FixedTextes zentriert dargestellt wird.

```
/* StarView ressource file */
#include "testtool.hrc"
FloatingWindow LOAD_CONF {
    Pos = MAP_APPFONT( 66, 23 );
    Size = MAP_APPFONT( 156, 51 );
    Text = "Lese Konfigurationsdateien";
    Moveable = TRUE;
    FixedText WORK {
        Pos = MAP_APPFONT( 0, 8 );
        Size = MAP_APPFONT( 155, 10 );
        Text = "Slot IDs";
        Center = TRUE;
    };
    FixedText FILENAME {
        Pos = MAP_APPFONT( 0, 21 );
        Size = MAP_APPFONT( 155, 10 );
        Text = "Datei.win";
        Center = TRUE;
    };
};
```

Beispiel 2: Datei testtool.src

Diese Trennung der Beschreibung der Benutzungsoberfläche vom Programmcode hat einige Vorteile:

Soll ein Dialog im Aussehen verändert werden, so muß nur die binäre Ressourcedatei neu erstellt werden und nicht der gesamte betroffene Code neu übersetzt werden. Ähnlich verfährt man mit den verschiedenen Übersetzungen in andere Landessprachen. Die Binärdateien enthalten jeweils nur Texte in einer Landessprache. Um die Binärdateien für die verschiedenen Sprachversionen zu erzeugen, muß also nur die neue Ressource bereitgestellt werden und nicht die gesamte Anwendung neu compiliert werden.

3.1.3 Zusammenhang von Ressourcen und Objekthierarchie zur Laufzeit

Die gesamte Benutzungsoberfläche wird in einer Anwendung durch hierarchisch gegliederte Fenster repräsentiert. Jedes dieser Fenster wird seitens des Programms durch ein Objekt der Klasse 'Window' beziehungsweise davon abgeleitete Klassen repräsen-

tiert. Die einzelnen Objekte erlauben die Navigation innerhalb der Hierarchie der Fenster, indem sie Verweise auf ihr Elternfenster und ihre Kindfenster bereitstellen. Das Hauptfenster kann als Einstieg in diese Hierarchie an der Anwendung selbst abgefragt werden. Das Testsystem benutzt diese Hierarchie, um die einzelnen Bedienelemente aufzufinden.

Ein Dialog besteht üblicherweise aus einem Fenster und mehreren Kindfenstern, die darauf plaziert sind. Die Kindfenster dienen dabei beispielsweise der Darstellung von Texten, Buttons oder Gruppierungen.

Von der Klasse 'Window' ist die Klasse 'Control' abgeleitet, die als Basis aller Bedienelemente dient und einige grundlegende Methoden implementiert.

Die Bedienelemente können bei ihrer Initialisierung die Informationen über ihr Aussehen und andere Eigenschaften aus den Ressourcen lesen. Sie bekommen hierfür im Konstruktor eine RessourceID übergeben, die neben der numerischen ID der Ressource aus den Ressourcedateien auch den Dateinamen der binären Datei enthält.

Jedes Objekt der Klasse 'Window' besitzt einen eindeutigen Ressourcetyp, der die Art des Bedienelements beschreibt und über eine Methode abgefragt werden kann. Alle Klassen, die aus den Ressourcen geladen werden können, initialisieren ihre Basisklasse 'Window', so daß ihr der entsprechende Ressourcetyp bekannt ist. Dieser Ressourcetyp dient der Erkennung des Typs des Bedienelements durch das Testsystem. Anhand dieses Typs werden die Methoden bestimmt, die an dem Objekt aufgerufen werden können.

3.1.4 Slots - Direkter Aufruf von Funktionen

Um den einzelnen Komponenten einer Anwendung das Versenden von Aufträgen an andere Komponenten zu ermöglichen, ohne deren Implementierung zu kennen, wurde an zentraler Stelle ein allgemeiner Mechanismus implementiert, dem diese Aufträge übergeben werden können. Dieser Mechanismus sorgt dann dafür, daß der Auftrag der entsprechenden Komponente zugestellt wird. Existieren mehrere Komponenten, die den Auftrag abarbeiten könnten, so wird die momentan aktive ausgewählt. Der Auftraggeber kann warten, bis der Auftrag beendet ist, oder er kann sich darauf verlassen, daß er zu einem späteren Zeitpunkt ausgeführt wird. Das Versenden von Aufträgen auf diese Art wird als *Aufruf eines Slots* bezeichnet, der Auftrag selbst als *Slot*.

3.2 Der Entwicklungsprozeß bei StarDivision

An einem so großen Projekt, wie es bei StarDivision durchgeführt wird, arbeiten viele Entwickler. Um zu überprüfen, ob die neuen und geänderten Programmteile noch reibungslos zusammenarbeiten und um den Entwicklern die Arbeit der anderen Entwickler zur Verfügung zu stellen, wird regelmäßig das gesamte Projekt compiliert. Da der Quellcode für alle Plattformen identisch ist, muß sichergestellt werden, daß die generierten Binärdateien der einzelnen Plattformen nicht vermischt werden. Dafür gibt es pro Plattform eine Verzeichnisstruktur, den Ausgabebaum, in dem die Binärdateien, zusammen mit weiteren beim Compilieren generierten Dateien, abgelegt werden.

Um zu vermeiden, daß sich jeder Entwickler jedesmal die Teilprojekte, die er bearbeitet, zusammen mit den Teilprojekten, auf denen er aufbaut, neu kopieren muß, werden immer einige komplette Sätze des Quellcodes zusammen mit den Ausgabebäumen bereitgestellt. Diese Sätze werden als Sourcestand bezeichnet.

Üblicherweise wird täglich ein neuer Sourcestand eröffnet und für etliche Plattformen compiliert. Welche Plattformen compiliert werden, hängt davon ab, was im Entwicklungsprozeß aktuell benötigt wird.

Nach einem rudimentären Funktionstest, der das Aufrufen je eines leeren Dokuments der verschiedenen Dokumenttypen sowie das Anzeigen einer Testseite umfaßt, wird dann der Stand plattformweise bereitgestellt, um von der Qualitätssicherung im einzelnen getestet zu werden.

3.3 Vorhandene Testmechanismen

Bei der Firma StarDivision existieren verschiedene Mechanismen zum Aufspüren von Fehlern in der Anwendung. Diese Mechanismen können individuell direkt in den Programmcode eingebaut werden. Es existieren zusätzlich bereits in die Anwendung integrierte Mechanismen, die an zentraler Stelle Kontroll- und Prüffunktionen übernehmen können und bei Bedarf aktiviert werden können.

Zur Diagnose von Zuständen in der Anwendung existieren Makros zur Ausgabe von Statusinformationen. Diese Makros können auch mit einer Bedingung versehen werden. Schlägt die Überprüfung der Bedingung fehl, liefert als Ergebnis also FALSE, so wird die Statusinformation ausgegeben. Dieses Konstrukt heißt Assertion (Zusicherung) und ist eine Weiterentwicklung der in Lyle & Zelkowitz (1979) besprochenen Assertions verschiedener Programmiersprachen. Im Gegensatz zu den dort vorgestellten Kon-

strukten, die bei Fehlschlagen der Bedingung das Programm abbrachen beziehungsweise eine Ausnahmebehandlung einleiteten, wird bei den hier vorgestellten Assertions lediglich eine Statusinformation erzeugt. Diese Assertions stellen einen einfachen Mechanismus dar, um Vorbedingungen, Nachbedingungen und Invarianten, wie sie von Hoare (1969) vorgeschlagen wurden, zu überprüfen.

```
DBG_ERROR( "Das Makro geht" );
DBG_ASSERT( 1 != 1 , "Eins ist und bleibt Eins" );
DBG_WARNING( "Dieser Aufruf ist ineffizient" );
DBG_TRACE( "Dieser Code wurde auch durchlaufen" );
```

Beispiel 3: Makros zur Ausgabe von Statusinformationen

Um die Wichtigkeit der Nachrichten zu verdeutlichen, gibt es drei verschiedene Ausgabekanäle für die Nachrichten. Neben einem Kanal für die Ausgabe von Fehlern, wie zum Beispiel dem Vorhandensein eines nicht initialisierten Pointers oder anderer schwerwiegender Probleme, existieren noch Kanäle für weniger wichtige Nachrichten. Es handelt sich einerseits um Warnungen und andererseits um Nachrichten, die der bloßen Protokollierung des Programmablaufes dienen. Die Syntax ist in Beispiel 3 dargestellt:

Die Ausgabe dieser Makros kann nach Belieben konfiguriert werden. Zum einen kann ein Dialog angezeigt werden, der bestätigt werden muß, bevor das Programm in seiner Ausführung fortgesetzt wird. Zum anderen kann alternativ dazu eine andere Ausgabeform gewählt werden, bis hin zum totalen Unterdrücken der Nachrichten. Als weitere Ausgabeformen können hier das Auflisten in einem separaten Fenster, das Schreiben in eine Datei, oder unter UNIX-Systemen auch die Ausgabe auf der Konsole eingestellt werden. Wird durch das Testsystem getestet, so können diese Nachrichten auch in das Journal des Testsystems aufgenommen werden. Diese Konfiguration kann für jeden der Ausgabekanäle separat vorgenommen werden.

In einer umfangreichen Anwendung gibt es viele Diagnosenachrichten. Würde man sie alle angezeigt bekommen, wäre das eine kaum überschaubare Datenflut. Insbesondere ist jeder Entwickler tendenziell nur an seinen eigenen Nachrichten interessiert. Deshalb können die Nachrichten zusätzlich gefiltert werden. Entweder können unerwünschte Nachrichten unterdrückt werden, oder es können ausschließlich die erwünschten Nachrichten selektiert werden. Die Auswahl erfolgt in beiden Fällen nach einem Teiltext, der in der Nachricht enthalten sein muß.

Auf diesen primitiven Makros wurden stärkere Mechanismen aufgebaut. Ein großes Problem bei dynamischer Speicherreservierung ist das Verwalten der Referenzen darauf. Eine Möglichkeit, dieses für Objekte zu bewerkstelligen, ist es, die Referenzen darauf zu verwalten. Das Objekt weiß dann, wieviele Referenzen noch existieren und kann sich bei Entfernen der letzten Referenz selbst löschen. Dieses Vorgehen setzt aber voraus, daß bei Erzeugung und Entfernung einer neuen Referenz das Objekt darüber benachrichtigt wird. C++ bietet als Referenz den Datentyp Pointer an. Dieser wird sowohl für Speicherreservierungen als auch zur Referenzierung von Objekten verwendet. Die einfache Zuweisung eines Pointers eignet sich nicht zur Verwaltung der Referenzen auf ein Objekt, da es keine Möglichkeit gibt, das Objekt von einer Zuweisung zu informieren. Die Lebenszeit eines Objektes wird somit nicht durch die Referenzen darauf bestimmt, sondern das Objekt wird zu einem beliebigen Zeitpunkt wieder destruiert. Es kann somit vorkommen, daß noch eine Referenz auf ein Objekt existiert, das schon wieder destruiert wurde, und die Referenz damit ungültig ist. Greift man auf dieses Objekt zu, so sind die Resultate unabsehbar. Zumindest in C++ bietet die Sprache keinerlei Unterstützung für dieses Problem (vgl. Behdjati, Bagdon, Fleischer & Schlüter 1991).

Bei Star Division existieren Makros, die bei entsprechender Instrumentierung der Klassen bei der Aufdeckung dieses Problems helfen können.

Um die Lebenszeit eines Objektes zu verfolgen, wird es bei seiner Erzeugung in eine Liste eingetragen, aus der es bei seiner Löschung wieder entfernt wird. Es muß also jeder Konstruktor der Klasse mit dem Makro zum Eintragen in die Liste und der Destruktor mit dem Makro zum Entfernen aus der Liste versehen werden. In jedem Methodenaufruf muß zu Beginn ein entsprechendes Makro eingefügt werden, das überprüft, ob das Objekt noch in der Liste enthalten ist. Ist dies nicht der Fall, so wird eine Diagnosenachricht generiert.

Da mit dieser Methode nur die Gültigkeit des Speicherbereiches des Objektes geprüft wird, nicht aber seine Konsistenz, gibt es die Möglichkeit, dem Prüfmakro noch eine Funktion mitzugeben, die dann auch die Konsistenz des Objektes überprüfen kann. Diese Funktion wird sowohl bei Betreten der Methode als auch bei ihrem Verlassen aufgerufen.

Neben dem Makro, das ein Prüfen innerhalb eines Methodenaufrufes veranlaßt, gibt es auch noch eines, mit dem beliebige Referenzen auf Objekte geprüft werden können. Voraussetzung ist auch hier, daß die entsprechende Klasse mit diesem Prüfmechanismus ausgestattet ist.

Wird vergessen, Objekte zu destruieren, so wird bei Programmende für jedes noch existierende Objekt mittels des DBG_ERROR Makros ein Fehler ausgegeben. Beispiel 4 illustriert das Instrumentieren einer Klasse mit den Makros zur Überprüfung der Lebensdauer der aus der Klasse erzeugten Objekte.

```
DBG_NAME( test );
class test
{
   test() {DBG_CTOR( test, NULL );}
   ~test() {DBG_DTOR( test, NULL );}
   void BeispielMethode()
   {
       DBG_CHKTHIS( test, NULL );
   }
}
```

Beispiel 4: Klasse mit Makros zur Überprüfung der Lebensdauer von Objekten

Will man nicht alle Klassen mit diesem Mechanismus ausstatten, so gibt es eine weitere Möglichkeit, die Reservierung und die Freigabe von Speicher zu überwachen. Hierbei werden alle entsprechenden Operationen überwacht und gesammelt. Am Ende des Programmlaufes wird dann eine Statistik ausgegeben. Es gibt zwei verschiedene Möglichkeiten, diese Ausgabe zu formatieren:

1. Zum einen können alle Speicheranforderungen mit Menge des angeforderten Speichers und Adresse im Programm, von wo aus die Anforderung getätigt wurde, aufgelistet werden.

2. Zum anderen kann eine Liste der Anforderungen und Freigaben, gruppiert nach Speichergröße, generiert werden. Die meisten Klassen haben einen charakteristischen Speicherbedarf, so daß leicht zu erkennen ist, ob alle instantiierten Objekte wieder destruiert wurden. Selbst wenn zwei Klassen den selben Speicherbedarf haben, kann diese Liste Aufschluß geben.

Es gibt aber nicht nur Unterstützungen zum Auffinden von Fehlern im Programm, sondern auch solche zur Qualitätssicherung der Benutzungsoberfläche. Es gibt einen Dialogtest, der automatisch an jedem neu geöffneten Dialog ausgeführt wird. Werden dabei Fehler festgestellt, so werden sie über eine Assertion dokumentiert. Untersucht wird im Einzelnen die Eindeutigkeit der Tastaturkürzel, mit denen Bedienelemente

direkt aktiviert werden können. Vor allem bei fremdsprachlichen Versionen kann es durch den Übersetzungsvorgang passieren, daß dasselbe Kürzel mehrfach vergeben wird. Außerdem wird versucht, die Reihenfolge zu überprüfen, in der die Bedienelemente aktiviert werden, wenn man mit der Tab-Taste von Bedienelement zu Bedienelement springt. Anschließend werden die einzelnen Bedienelemente daraufhin überprüft, ob sie sich überlappen. Beim Design eines Dialoges werden meist mehrere Bedienelemente verwendet, die einen Text anzeigen. Für die Anzeige des Textes steht allerdings nur die Fläche des Bedienelementes zur Verfügung. Ist es zu klein, so wird der Text abgeschnitten. Wird der Text verändert oder liegt er in einer anderen Landessprache vor, so kann es vorkommen, daß er länger wird. Damit dieser Text nicht abgeschnitten wird, ist es wichtig, daß das Bedienelement möglichst groß definiert wurde. Häufig ist rechts neben dem Text noch Platz bis zum Rand des Dialoges oder einem anderen Bedienelement frei. Die Bedienelemente sollten also so definiert werden, daß sie sich möglichst weit nach rechts ausdehnen, auch wenn der aktuelle Text dies nicht erfordert. Um die korrekte Dimensionierung der Bedienelemente in dem Dialog zu überprüfen, werden die Bedienelemente, die Texte darstellen, farbig hinterlegt. Ein Bedienelement, das nicht in maximaler Größe definiert ist, kann so schnell erkannt werden.

Die gesamte Konfiguration dieser Hilfswerkzeuge geschieht zur Laufzeit des Programmes über eine spezielle Tastenkombination, die einen Dialog öffnet. In diesem Dialog können sämtliche dieser Werkzeuge abgeschaltet werden, so daß nicht immer Zeit für die verschiedenen Prüfungen aufgewendet werden muß.

Da all diese Mechanismen den Code nicht unerheblich vergrößern und insbesondere im Fall der Speicher- und Objektüberwachung massive Einbußen in der Performanz mit sich bringen, werden bei StarDivision zwei Programmversionen generiert.

Die eine Programmversion wird ohne all diese Mechanismen compiliert. Durch die Definition der Mechanismen als Makros können sie einfach aus dem Quellcode ausgeblendet werden, indem die Makros einfach leer implementiert werden, was dazu führt, daß sie vom Precompiler bereits komplett entfernt werden und somit diese Programmversion in keiner Weise beeinflussen. Es ist dadurch allerdings notwendig, eine wichtige Regel zu befolgen. Da der Code in den Bedingungen der Assertions und in den Funktionen zum Testen der Konsistenz der Objekte nicht mehr ausgeführt wird, darf er keine Seiteneffekte haben. Das heißt, er darf keine Veränderung an den Daten der Objekte vornehmen. Anderenfalls würden die Ergebnisse der Programmausführung verfälscht werden.

Die zweite Programmversion, die sogenannte Debug-Version, wird mit allen Testmechanismen compiliert. Der Nachteil dieses Verfahrens ist, daß immer doppelt so viele Programmversionen komplett compiliert werden müssen. Dieser Nachteil wiegt, besonders angesichts der Notwendigkeit für mehrere Plattformen Programmversionen erstellen zu müssen, noch schwerer. Die Vorteile eines schnellen Produktes einerseits und komfortabler Testmechanismen andererseits, die dieses Vorgehen mit sich bringt, überwiegen diesen Nachteil allerdings.

3.4 Verschiedene Zielsetzungen der Tests

Es gibt mehrere Gründe, warum die Anwendung nicht jeden Tag komplett getestet wird. Da ein kompletter Test einer gesamten Anwendung mehrere Stunden dauert, aber jeden Tag eine neue Version zur Verfügung gestellt wird, wären die Ergebnisse also bereits veraltet, wenn sie zur Verfügung stünden.

Eine Alternative wäre, den Test aufzuspalten, und die einzelnen Teile auf mehreren Computern gleichzeitig durchzuführen. Für die verschiedenen Plattformen und Landesversionen wäre aber eine beträchtliche Anzahl von Computern notwendig, ganz abgesehen von dem Personal, das benötigt würde, die verschiedenen Computer zu bedienen. Dieser Kostenaufwand stünde in einem schlechten Verhältnis zu dem Nutzen. Deshalb müssen verschiedene Tests unter unterschiedlichen Aspekten gestaltet werden.

Es gibt einen schnell durchführbaren Test, der täglich ausgeführt werden kann und die einzelnen Module nur oberflächlich testet.

Darüber hinaus existieren intensivere Tests für die einzelnen Module, die bei größeren Veränderungen in den Modulen ausgeführt werden.

3.4.1 Test der Ressourcen

Da durch den Compiler nicht sichergestellt werden kann, daß alle im Programmcode angesprochenen Ressourcen auch tatsächlich in den Ressourcedateien enthalten sind, ist es wichtig, einen Test zu erstellen, der zunächst das Vorhandensein sämtlicher Ressourcen überprüft.

Dies geschieht durch Aufrufen sämtlicher Dialoge und Programmpunkte der Anwendung. Es sind allerdings nicht alle Dialoge direkt zu erreichen. Es müssen also gegebenenfalls Vorbedingungen geschaffen werden, um die Funktionalität freizulegen, in der die Dialoge enthalten sind. So muß beispielsweise erst ein Dokument geöffnet werden, bevor der Dialog zum Formatieren eines Absatzes aufgerufen werden kann.

3.4.2 Test der Funktionalität

Im Gegensatz zum Testen der Ressourcen wird beim funktionalen Testen bereits davon ausgegangen, daß die Ressourcen vorhanden sind. Sie werden jetzt auf Konsistenz und Funktionalität getestet. So wird beispielsweise überprüft, ob der einstellbare Wertebereich mit dem gewünschten übereinstimmt. Hier kann auch getestet werden, ob einzelne Bedienelemente in Abhängigkeit von anderen ausgeblendet werden. Wenn beispielsweise der Anfang eines Absatzes mit einem Großbuchstaben beginnen soll, so muß die entsprechende Option eingeschaltet sein, bevor die Höhe des Buchstabens eingestellt werden kann (siehe hierzu auch den Testfall 'FunktionalerTest' des Beispielprogramms im Anhang).

Weiterhin muß geprüft werden, ob die einzelnen Komponenten durch alle verfügbaren Methoden aufgerufen werden können. Beispielsweise kann es passieren, daß der Dialog zum Formatieren eines Absatzes durch den Aufruf eines Slots (vgl. Kap. 3.1.4) geöffnet werden kann, aber ein entsprechender Eintrag im Kontextmenü fehlt. Es kann auch vorkommen, daß der Eintrag zwar vorhanden ist, aber die falsche Funktion ausführt.

Nachdem die Benutzungsoberfläche getestet wurde, muß noch die Funktion der Anwendung getestet werden. Um einen möglichst schnellen Testablauf zu ermöglichen, werden dabei vermehrt Slots verwendet. Um beispielsweise die Importfilter für verschiedene Dateiformate zu testen, werden die Dateien direkt unter Angabe ihres Dateinamens und des gewünschten Importfilters aufgerufen, anstatt die entsprechenden Daten in den 'Datei Öffnen' Dialog einzutragen.

Ein weiteres Beispiel zum Testen der Funktionalität ist das Überprüfen der Rechenergebnisse in der Tabellenkalkulation oder im Textdokument. Leider können hier nur sehr wenige Slots verwendet werden, da die Zahlen und Formeln nur über die Benutzungsoberfläche eingetragen werden können.

3.4.3 Test der Differenz zweier Versionen

Diese Tests beziehen sich aber nur auf eine Version der Anwendung, die eine Momentaufnahme im Prozeß der Softwareentwicklung darstellt. Da in der Entwicklung aber jeden Tag eine neue Version zu testen ist, muß auch die Konsistenz von Version zu Version überprüft beziehungsweise es müssen Unterschiede aufgezeigt werden.

Unterschiede können hierbei in Form von hinzukommenden oder wegfallenden Bedienelementen sowie durch veränderte Reaktionen der Anwendung auf Eingaben

auftreten. Die Erkennung von wegfallenden Bedienelementen leistet das Testsystem quasi nebenbei. Wird der Programmteil mit den Veränderungen getestet, so treten bei dem Versuch, auf die nichtexistenten Bedienelemente zuzugreifen, Fehler auf. Auch veränderte Reaktionen der Anwendung auf Eingaben werden meistens auf dieselbe Art erkannt, da auch hier das Skript einen anderen Zustand der Anwendung erwartet und somit Fehlbedienungen durchführen wird. Wenn Bedienelemente hinzukommen, muß das Skript allerdings entsprechend erweitert werden.

Zusätzlich wird bei StarDivision für viele verschiedene Plattformen (Microsoft Windows, OS/2, MacIntosh, Linux, Solaris und andere UNIX-Derivate) parallel entwickelt. Daraus ergibt sich die Notwendigkeit, auch die Konsistenz zwischen den Plattformen zu wahren. Durch den ebenfalls plattformübergreifenden Ansatz des Testsystems sind nur wenige Randbedingungen zu berücksichtigen, um einen reibungslosen Ablauf zu gewährleisten (vgl. Kap. 4.3).

4 Das Testsystem

In diesem Kapitel sollen der Aufbau und die Funktionsweise des im Rahmen dieser Diplomarbeit entstandenen Testsystems dargestellt werden. Zunächst wird ein grober Überblick über das System gegeben. Danach wird auf die besonderen Bedürfnisse einer Skriptsprache für ein Testsystem eingegangen, gefolgt von Betrachtungen zur Plattformunabhängigkeit und zur Unterstützung landesspezifischer Versionen der Anwendung. Einer Beschreibung der Besonderheiten der Entwicklungsumgebung für die Skripten folgt eine abschließende detaillierte Beschreibung der Fernsteuerung der Anwendung.

4.1 Genereller Aufbau und Funktionsweise des Testsystems

Wie in Abbildung 5 gezeigt, wurde das Testsystem in zwei wesentliche Programmteile unterteilt. Der eine Programmteil ist eine Entwicklungsumgebung für die Testskripte. Sie dient gleichzeitig dazu, die Testskripte auszuführen und die Ergebnisse des Tests zu sammeln. Dieser Programmteil wird als unabhängige Applikation realisiert und als Testtool bezeichnet. Dieses Testtool kommuniziert mit dem zweiten Programmteil, dem Controller, innerhalb der zu testenden Anwendung. Der Controller übernimmt dann die eigentliche Bedienung der Anwendung.

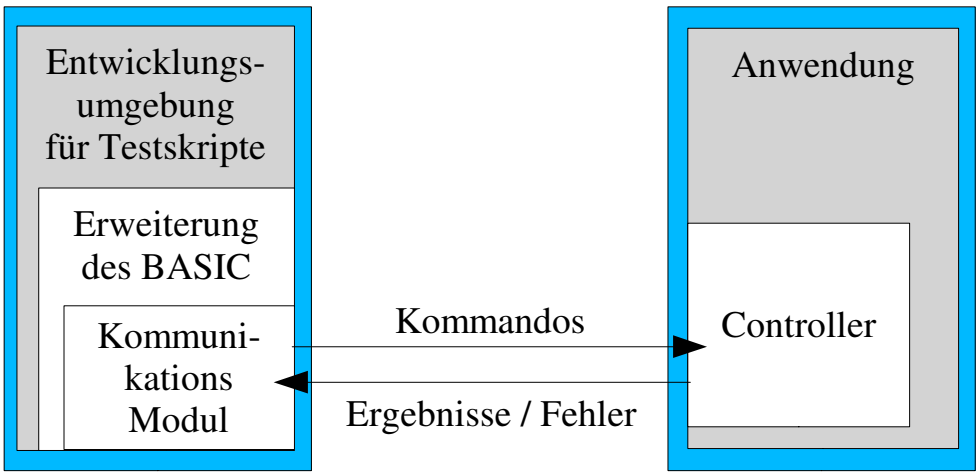

Abbildung 5: Module des neuen Testsystems

Durch diese Aufteilung ergeben sich einige Vorteile, die hier im einzelnen erläutert werden sollen.

Ein Ziel dieser Aufteilung ist es, ein möglichst stabiles Testsystem zu erhalten. Da der Kommunikationsmechanismus zwischen den Programmteilen versionsunabhängig ist, kann das Testtool in einer stabilen älteren Version verwendet werden, während die Anwendung in einer neueren, gelegentlich instabilen, Version vorliegt. Das Testtool muß lediglich bei Implementierung von neuen Fähigkeiten oder der Beseitigung von Fehlern erneuert werden.

Die Aufteilung in zwei Teile hat außerdem den Vorteil, daß im Fall eines Programmabsturzes der Anwendung das Testtool weiter läuft und den Absturz registrieren und dokumentieren kann. Im Controller ist ein Mechanismus implementiert, der einen Absturz feststellt, der direkt beim Ansteuern eines Bedienelementes auftritt. Es wird dann ein Fehler generiert, der zusammen mit anderen Ergebnissen, die sich eventuell angesammelt haben, an das Testtool zurückgeschickt wird. Anschließend wird die Anwendung direkt beendet. Häufig werden aber durch das Ansteuern von Bedienelementen weitere Aktionen zur späteren Ausführung vorgemerkt. Tritt bei der Ausführung einer solchen asynchronen Aktion ein Programmabsturz auf, so ist das Verhalten je nach Plattform unterschiedlich und kann vom Testtool nur daran festgestellt werden, daß die Kommunikation abreißt. In beiden Fällen wird die Anwendung jedoch automatisch neu gestartet.

Falls das Testtool selbst abstürzt, muß die Sicherheit des Journals gewährleistet sein. Aus diesem Grund werden die gesammelten Daten sofort in der Journaldatei gesichert und nicht nur im Speicher gehalten, bis sie explizit gespeichert werden.

Durch die Wahl von TCP/IP als Kommunikationsmechanismus ist es möglich, die Anwendung und das Testtool auf unterschiedlichen Computern zu starten. Es gibt einige Vorteile, die dadurch entstehen. Zum einen werden Wechselwirkungen zwischen Anwendung und Testtool unterbunden. Wechselwirkungen sind bezüglich des Speicherplatzbedarfs und des Zeitverhaltens zu erwarten. Zum anderen kann es zu unerwünschten Erscheinungen kommen, wenn die bisherigen Ergebnisse durch Blättern im Journal, das im Testtool angezeigt wird, kontrolliert werden. Wird die Anwendung zum Beispiel gerade über Elementarereignisse angesteuert, so könnte das Bewegen der Maus zu Fehlern im Testablauf führen, die sonst nicht aufgetreten wären.

Laufen Testtool und Anwendung auf unterschiedlichen Betriebssystemen, erspart das der Person, die die Skripten erstellt und die Testabläufe startet und überwacht, die Einarbeitung in ein anderes Betriebssystem. Darüber hinaus entfällt die Notwendigkeit, auch auf dem Ziel-Betriebssystem die Verzeichnisstruktur mit den Skripten und Konfigurationsdateien, die vom Testtool benötigt werden, verfügbar zu machen. Ein Nachteil

der Ausführung auf unterschiedlichen Computern ist allerdings, daß dann die Anwendung nicht automatisch gestartet werden kann. Dies muß dann vor dem Start des Testlaufes manuell geschehen.

Neuartig an diesem Ansatz ist die Verschmelzung der Anwendung mit dem Testsystem. Die Integration des Controllers in die Anwendung ermöglicht es, die volle Kontrolle über die Anwendung auszuüben, und nicht nur über die Teile der Anwendung, die sie nach außen hin preisgibt. Durch diese Integration entfällt die Notwendigkeit von umständlichen Anpassungen an die Anwendung, wie sie in Kapitel 2.4 beschrieben wird. Die Funktionalität kann durch direktes Aufrufen der entsprechenden Methoden an den Objekten der Anwendung bereitgestellt werden. Die verfügbare Funktionalität kann also sehr ökonomisch nach den Bedürfnissen des Testers erweitert werden.

Um einen möglichst realistischen Test zu gewährleisten, ist es erforderlich, daß der Test in dem fertigen Produkt, so wie es an den Kunden ausgeliefert werden soll, ausgeführt wird. Aus diesem Grund wird auch der Controller mit ausgeliefert. Ansonsten müßte nochmals eine letzte Version ohne den Controller erstellt werden, was eine zusätzliche Fehlerquelle wäre.

Die Belastung des Produktes durch solchen durch den Anwender nicht verwendeten Code soll möglichst gering gehalten werden. Durch die Aufspaltung des Testsystems kann dies gewährleistet werden. Die Belastung durch die Codegröße beläuft sich etwa auf 70 KByte Code, was gemessen am Gesamtvolumen je nach Plattform zwischen 0,3 % und 1 % liegt. Performanz-Einbußen sind nicht zu erwarten, da die gesamte Funktionalität nicht in den generellen Code eingebaut ist, sondern nur während des Testens verwendet wird. Lediglich bei Programmstart wird eine kleine Codesequenz durchlaufen, die bestimmt, ob das Testmodul aktiviert wird oder nicht.

Als Basis der Skriptsprache wurde das zum de facto Standard VisualBASIC kompatible hauseigene BASIC verwendet. Die Funktionalität dieses BASIC-Dialektes wurde als ausreichend angesehen, um derartige Skripten zu erstellen. Er verfügt über Prozeduren und Funktionen mit lokalen Variablen und Parameterübergabe. Da der Interpreter im Quellcode vorliegt, kann er gleichzeitig für die Bedürfnisse des Testsystems angepaßt werden.

Der Sprachumfang mußte zum einen um neue Kontrollstrukturen und Befehle erweitert werden. Zum anderen mußte ein dynamischer Mechanismus geschaffen werden, um die einzelnen Bedienelemente der Anwendung in den Namensraum der Sprache abzu-

bilden. Dies geschieht durch einfache Befehle und Objekte mit entsprechenden Methoden.

Bei Ablauf eines Skriptes generieren die dynamisch erzeugten Befehle und Methodenaufrufe eine Sequenz von Mikrobefehlen, die dann an den Controller in der Anwendung versendet wird. Nach Bearbeitung durch den Controller werden die Ergebnisse, bestehend aus einem eventuellen Fehler und einem optionalen Rückgabewert, bereitgestellt und die Skriptbearbeitung fortgesetzt. Im Fehlerfall wird dann eine Fehlerbehandlung durchgeführt.

Abbildung 6: Darstellung des Testsystems in der Titelzeile der Anwendung

C+	Es wurde eine Verbindung hergestellt	D	Es wurden Daten empfangen
C-	Die Verbindung wurde beendet	S	Es wurden Daten gesendet

Abbildung 7: Zustände der Kommunikation

Der Controller beeinflußt die Benutzungsoberfläche beim Abarbeiten der Befehle, abgesehen von den Befehlen selbst, so wenig wie möglich, um einen möglichst originalgetreuen Ablauf der Anwendung zu ermöglichen. Anderenfalls wären Wechselwirkungen, wie oben beschrieben, zu befürchten. Der Controller ist nur an einer Stelle in der Benutzungsoberfläche sichtbar. Lediglich an der Titelzeile der Anwendung ist zu sehen, ob die Anwendung durch den Controller ansteuerbar ist beziehungsweise gerade gesteuert wird. Sie wird dann, wie in Abbildung 6 zu sehen, um die Kennung 'TT' gefolgt von der Portnummer in eckigen Klammern ergänzt. Besteht eine Verbindung, so stehen hinter der Kennung noch weitere Zeichen, die den momentanen Zustand der Kommunikation darstellen. Sie werden von dem Namen des Computers ergänzt, zu dem die Verbindung besteht. Die Bedeutungen der Zeichen sind in Abbildung 7 beschrieben.

Die Aktivierung der einzelnen Befehle im Controller erfolgt über eine Funktion, die immer dann gerufen wird, wenn die Anwendung auf neue Eingaben wartet. Der Aufruf über diese Idle-Funktion soll gewährleisten, daß nicht während der Ausführung eines Befehls schon der nächste angestoßen wird.

Es existieren außerdem noch einige Sonderfunktionen, die zur Diagnose und Skriptenerstellung dienen. Diese Funktionen erfordern eine eigene Repräsentation in der Benutzungsoberfläche. Da sie aber während der Abarbeitung der Tests nicht benutzt werden, beeinträchtigen sie den Ablauf der Anwendung nicht.

4.2 Erweiterungen der Skriptsprache BASIC

Das Testen einer Anwendung stellt besondere Anforderungen an die Kontrollstrukturen der Skriptsprache. Die hier vorgestellten neuen Kontrollstrukturen lassen sich zwar auch durch die in BASIC schon vorhandenen Kontrollstrukturen realisieren, die Skripte werden dann aber unübersichtlich und damit fehlerträchtig.

Die Anwendung, die getestet werden soll, steht bei den Skripten im Vordergrund. Es ist deshalb für einen einfachen, möglichst direkten Weg zu sorgen, die Strukturen der Anwendung in die Skriptsprache abzubilden. Dies kann durch einige wenige Befehle geschehen, die dann etwa wie folgt aussehen:

```
AccessControl "ControlName", "Methode", "Parameter"
```

Derartige Skripten wären aber nur schwer lesbar, da sie voll von redundanter Information sind, die zuerst gefiltert werden muß und somit die kognitiven Fähigkeiten des Anwenders unnötig belastet.

In dem hier verfolgten Ansatz sieht der obige Befehl folgendermaßen aus:

```
ControlName.Methode "Parameter"
```

Die Information ist auf das notwendige reduziert und kann somit vom Anwender schneller aufgenommen werden.

4.2.1 Befehle zum Initialisieren des Testsystems

Bei jedem Start eines Skriptes wird zunächst das Unterprogramm mit dem Namen LoadIncludeFiles aufgerufen. In diesem Unterprogramm müssen die dynamischen Repräsentationen der Benutzungsoberfläche der Anwendung geladen werden. Eine detaillierte Erläuterung dieses Verfahrens wird weiter unten bei der Beschreibung des Befehls 'Use' gegeben.

Innerhalb dieses Unterprogrammes können weitere Module geladen werden, die beispielsweise häufig verwendete Routinen enthalten oder zur weiteren Initialisierung benötigt werden. Um die Verbindung zur Anwendung herzustellen, muß beispielsweise der Pfad zu der Programmdatei ermittelt werden, so daß die Anwendung gegebenenfalls zuerst gestartet werden kann. Der Prototyp eines Testskriptes ist in Beispiel 5 dargestellt.

```
sub main
   TesteAlles
end sub

sub TesteAlles
end sub

sub LoadIncludeFiles
rem hier wird das Skript initialisiert
   start "Dateiname des Executables", "Parameter"
end sub
```

Beispiel 5: Prototyp eines Testskriptes

4.2.2 Kontrollstrukturen

Um das Testen und Erfassen der Resultate zu unterstützen, wurde der Befehlssatz des BASIC um zwei Kontrollstrukturen erweitert. Beide Kontrollstrukturen sind sehr ähnlich. In beiden Fällen enthalten sie einen Befehlsblock, dessen Ausführung im Fehlerfall abgebrochen wird. In beiden Fällen wird dann Code zur Fehlerbehandlung ausgeführt.

Unterteilung der Testskripte in einzelne Testfälle

Um das Testen der Anwendung zu modularisieren, werden die Skripten in einzelne Abschnitte, hier Testfälle genannt, unterteilt. Die Testfälle können dann unabhängig voneinander durchlaufen werden. Um diese Unabhängigkeit zu garantieren, muß am Anfang jedes Testfalles die Anwendung in einen definierten Grundzustand versetzt werden. Normalerweise sollte das durch die Testfälle selbst erledigt werden, allerdings kann im Fehlerfall ein beliebiger undefinierter Zustand entstehen, der dann durch die Fehlerbehandlung zurückgesetzt werden muß. Vor Betreten eines Testfalles wird ein global definiertes Unterprogramm namens TestEnter aufgerufen, das diesen definierten Grundzustand herstellt. War die Anwendung nicht bereits im Grundzustand, so ist dies ein Fehler und sollte durch das Unterprogramm im Journal vermerkt werden.

Ein Testfall dokumentiert sich selbst im Journal, indem er seine Ausführung vermerkt. Weitere Ergebnisse können mit den dafür vorgesehenen Befehlen in das Journal eingetragen werden. Diese Eintragungen werden hierarchisch unter dem Testfall angeordnet.

Bei Abbruch eines Testfalles durch einen Fehler wird auch dieser zusammen mit den Unterprogrammaufrufen, die zu der Fehlerstelle führen, dem sogenannten *Call Stack*, automatisch in das Journal geschrieben. In diesem Fall wird außerdem ein weiteres, ebenfalls global definiertes Unterprogramm namens TestExit gerufen, das sicherstellen muß, daß der definierte Grundzustand, der nach jedem Testfall vorliegen soll, wiederhergestellt wird. Hierfür stellt das Testsystem einen Befehl zur Verfügung, der sämtliche geöffneten Dialoge und Dokumente schließt und entsprechende Daten bereitstellt, so daß im Journal vermerkt werden kann, welche Fenster noch geöffnet waren.

```
testcase <Bezeichner>
endcase

sub TestEnter
end sub

sub TestExit
   ResetApplication
End Sub
```

Beispiel 6: Prototyp eines Testcase

Realisiert werden die Testfälle durch ein Konstrukt, das bis auf die oben beschriebene Funktionalität einem Unterprogramm in BASIC entspricht. Die Syntax, ergänzt durch die notwendigen Unterprogramme TestEnter und TestExit, ist in Beispiel 6 angegeben.

Fehlerbehandlung für einzelne Programmabschnitte

In manchen Fällen ist es nicht notwendig oder sogar nicht wünschenswert, bei Auftreten eines Fehlers einen gesamten Testfall abzubrechen. Dies ist der Fall, wenn die Fehlerursache beispielsweise einfach zu umgehen ist, oder nur ein kleiner Teil des Testfalles durch Auftreten des Fehlers nicht getestet werden kann.

Es kann auch erwünscht sein, daß bestimmte Funktionen in bestimmten Konstellationen nicht ausgeführt werden können. In diesem Fall wäre dann das Auftreten eines Fehlers beim Testen eine korrekte Reaktion der Anwendung. Ein derart hervorgerufener Fehler soll nicht zum Abbruch des Testfalles führen und auch nicht als Fehler im Journal dokumentiert werden.

Für diese Fälle kann ein Befehlsblock *ausprobiert* werden. Im Gegensatz zu dem Testcase ist der Code zur Fehlerbehandlung hier individuell für jeden auszuprobierenden

Befehlsblock anzugeben. Es werden also zwei Befehlsblöcke verwendet, die durch die Schlüsselworte 'try', 'catch' und 'endcatch' getrennt sind. Tritt in dem Befehlsblock zwi-

```
try
   <Befehlsblock>
catch
   <Befehlsblock>
endcatch
```

Beispiel 7: Das try - catch - endcatch Konstrukt

schen 'try' und 'catch' ein Fehler auf, wird der Block sofort verlassen, und die Befehle zwischen 'catch' und 'endcatch' werden ausgeführt. Tritt kein Fehler auf, so wird hinter dem 'endcatch' fortgefahren. Die Syntax ist in Beispiel 7 dokumentiert:

4.2.3 Befehle zur Dokumentation der Ergebnisse der Testläufe

Um eine aussagekräftige Dokumentation der Testläufe zu ermöglichen, wird ein Journal erstellt. Das Testsystem sammelt bei Ausführung des Skriptes alle gewünschten Informationen. Dabei werden Eintragungen über

- Datum und Uhrzeit des Starts,
- ausgeführte Testfälle und
- Fehler, die zum Abbruch eines Testfalles geführt haben,

automatisch erzeugt.

Sollen weitere Ergebnisse protokolliert werden, stehen dem Programmierer der Skripten darüber hinaus verschiedene Möglichkeiten zur Verfügung, Informationen in das Journal aufzunehmen. Dabei kann je nach Bedeutung der Information ein anderer Befehl verwendet werden. Die Informationen werden dann im Journal je nach Bedeutung unterschiedlich dargestellt. Im einzelnen stehen folgende Befehle zur Verfügung:

```
printlog "Normaler Text"
```

Mit diesem Befehl können einfache Statusmeldungen eingefügt werden, zum Beispiel, wer den Test gestartet hat, oder auf welchen Betriebssystemen und Computern das Testtool und die Anwendung ausgeführt werden.

```
warnlog "Warnung"
```

Der Befehl fügt Warnungen in das Journal ein. Diese werden im Journal farblich hervorgehoben.

```
errorlog
```

Mit diesem Befehl kann ein Fehler, der in einem Try-Catch-Endcatch Konstrukt aufgetreten ist, in das Journal übernommen werden. Fehler werden ebenfalls farblich hervorgehoben.

```
ExceptLog
```

Der Begehl ExceptLog hat die gleiche Funktion wie der Befehl ErrorLog. Zusätzlich wird jedoch ein Call-Stack erzeugt.

4.2.4 Dynamische Repräsentation der Elemente der Anwendung

Um die vielen hundert Dialoge und Bedienelemente einer umfangreichen Anwendung ansprechen zu können, werden sie auf folgende Weise identifiziert.

Um den zusätzlichen Aufwand an Speicherbedarf und anderen Ressourcen zu minimieren, bietet sich eine numerische Repräsentation an. Mit ihr kann dann jedes einzelne Bedienelement eindeutig referenziert werden.

Hilfreich bei der Vergabe der numerischen IDs ist dabei, daß es bereits ein kontextsensitives Hilfesystem gibt, das ebenfalls an zentraler Stelle eine Identifizierung eines Dialoges oder Bedienelementes vornehmen muß. Bei StarDivision war das Problem ebenfalls durch Vergabe von numerischen IDs für jedes Hilfethema realisiert worden. Diese IDs können an jedem Fenster, also auch an jedem Bedienelement, abgefragt werden. Ist die ID gleich Null, so bedeutet das, daß keine Hilfe verfügbar ist. In den meisten Fällen trifft dies jedoch nur auf Fenster zu, die ohnehin keine Bedienelemente sind. Da das Hilfesystem an zentraler Stelle angesiedelt ist, müssen diese numerischen IDs eindeutig sein und können daher zur Identifizierung von Bedienelementen herangezogen werden. Es gibt jedoch einige Sonderfälle.

Es kommt vor, daß ein Hilfethema für mehrere Bedienelemente zutrifft. Zum Beispiel dann, wenn ähnliche Inhalte in einer tabellarischen Form dargestellt werden. Es gibt

auch Bedienelemente, die kein Hilfethema haben und damit auch keine Hilfe ID tragen dürfen. Beispielsweise trifft dies für die Dokumentfenster zu.

Um auch bei diesen Ausnahmen eine eindeutige Identifizierung zu erreichen, wurde eine zweite ID eingeführt. Sie heißt Unique ID. Wird keine Unique ID für ein Bedienelement definiert, so wird seine Hilfe ID zurückgeliefert. Das bedeutet also, daß nur in den oben genannten Spezialfällen diese Unique IDs vergeben werden müssen.

Es verbleiben aber immer noch eine Reihe von Bedienelementen, die keine ID haben. Es handelt sich hierbei um Standard-Bedienelemente wie zum Beispiel den OK-Button. Üblicherweise befindet er sich aber auf einem Dialog. Die Funktionalität des Auslösens des Buttons kann also über den Dialog angesprochen werden. Sollte ein OK-Button dennoch außerhalb eines Dialoges auftreten, so muß er eine Unique ID bekommen, um angesprochen werden zu können.

Um einen höheren Abstraktionsgrad zu erhalten und um die Ausführung von Skripten zu beschleunigen, gibt es eine zweite Möglichkeit der Interaktion zwischen Skript und Anwendung. Sie besteht darin, die Slots (vgl. Kap. 3.1.4) in der Anwendung direkt aufzurufen, anstatt sie über mehrstufige Zugriffspfade der Benutzungsoberfläche zu aktivieren. Die Verwendung von Tastenkombinationen hat eine ähnliche Wirkung, aber das Einführen einer separaten Befehlsklasse erhöht die Lesbarkeit der Skripten erheblich und bietet gleichzeitig eine sicherere Fehlererkennung. Während eine Tastenkombination nur an die Anwendung gesendet werden kann, ohne daß ihre Wirkung überprüft wird, kann bei einer neuen Befehlsklasse überprüft werden, ob der gewünschte Befehl existiert oder nicht. Außerdem wäre die Verwendung von Tastenkombinationen zum Aufruf der Slots, soweit existent, nicht sprachunabhängig. Die Slots werden ähnlich der Benutzungsoberfläche intern numerisch dargestellt. Ein Bezug zum Hilfesystem existiert jedoch nicht.

Ein Skript, das nur über Zahlen arbeitet, ist absolut unleserlich. Deshalb wird die interne, numerische Repräsentation der Benutzungsoberfläche mit sprechenden Namen versehen. Über diese Namen werden dann die Bedienelemente beziehungsweise die Slotaufrufe referenziert.

Da eine Benutzungsoberfläche eine natürliche Struktur in Form von Dialogen und Menüs aufweist, soll diese Struktur auch in dem Testsystem abgebildet werden. Die einzelnen Bedienelemente werden deshalb zu Kontexten zusammengefaßt. Die Slots weisen leider keine derartige Struktur auf, so daß hier eine flache Repräsentation gewählt werden mußte.

Versionsunabhängige Repräsentation der Bedienelemente

Die Zuordnung der numerischen IDs zu den Bedienelementen ist nicht statisch. Den einzelnen Teilprojekten werden Bereiche zugewiesen, innerhalb derer die IDs liegen dürfen. Wächst ein Teilprojekt so stark an, daß der ihm zugewiesene Bereich nicht mehr ausreicht, so wird er vergrößert, was zur Folge hat, daß sich alle nachfolgenden Bereiche verschieben. Das ist aus programmtechnischer Sicht kein ernstes Problem, da die IDs dort alle relativ definiert sind. Sie bestehen also immer aus einer Basis, nämlich dem Projektanfang und einem individuellen Offset. Wird also die Basis geändert, muß keine weitere Anpassung vorgenommen werden.

Durch diese Verschiebungen bedingt ist die interne numerische Repräsentation nur innerhalb einer Version der Anwendung stabil und kann sich zur nächsten Version ändern. Um dieses Problem zu umgehen, wurde eine zusätzliche Tabelle erstellt, die der internen numerischen Repräsentation einen eindeutigen versionsunabhängigen sogenannten *Langnamen* zuordnet.

Um auch bei den Langnamen die Eindeutigkeit sicherzustellen, werden sie aus mehreren Stufen zusammengesetzt. Zunächst wird der symbolische Name aus der Ressource mit dem des zugehörigen Dialoges verknüpft. Dieser wird dann mit dem Ressourcetyp des Bedienelementes und dem des Dialoges verknüpft. Um die Wahrscheinlichkeit einer Namensgleichheit weiter zu minimieren, wird dann noch ein Kürzel, das das Teilprojekt repräsentiert, aus dem der Dialog stammt, vorangestellt. Sollte der daraus entstandene Langname nicht eindeutig sein, so muß in den Ressourcen ein anderer Name vergeben werden. Innerhalb eines Unterprojektes sollen aber sowieso keine zwei gleichen Bezeichner auftreten. Diese Namen haben den Nachteil, üblicherweise 50 bis 60 Zeichen lang zu sein. Sie sind daher nicht für eine Anwendung durch Menschen geeignet.

Die Slots sind nicht in den Ressourcen definiert und müssen daher aus anderen Dateien gewonnen werden. Diese Dateien beschreiben die Schnittstelle der Anwendung zu externen Komponenten. Da diese Schnittstelle hier nicht relevant ist, soll auf die Beschreibung der Syntax und ihrer genauen Bedeutung verzichtet werden. Die Bezeichner der Slots sind jedoch für die gesamte Anwendung eindeutig. Sie können daher direkt verwendet werden. Auch diese Bezeichner sind allerdings zu lang, um direkt zur Programmierung der Skripten herangezogen werden zu können.

Die so gewonnenen Daten werden tabellarisch in einer Datei namens 'hid.lst' gespeichert. Das Format ist einfach. Es handelt sich um eine Textdatei, in der pro Zeile ein Langname gefolgt von der Numerischen ID steht.

Manuelle Zuordnung der eindeutigen Namen zu besser lesbaren Namen

Um den oben erwähnten Nachteil der unhandlichen Langnamen zu umgehen, werden ihnen sogenannte Kurznamen zugeordnet. Über diese Kurznamen werden die Elemente der Benutzungsoberfläche und die Slots in den Skripten angesprochen. Da es nicht möglich ist, die Kurznamen automatisch zu generieren, muß diese Zuordnung manuell vorgenommen werden.

Die Kurznamen werden nach Slots und Bedienelementen getrennt erfaßt. Dies ist notwendig, um bei der Ausführung des Skriptes zwischen Objektbezeichnern und Befehlsnamen unterscheiden zu können.

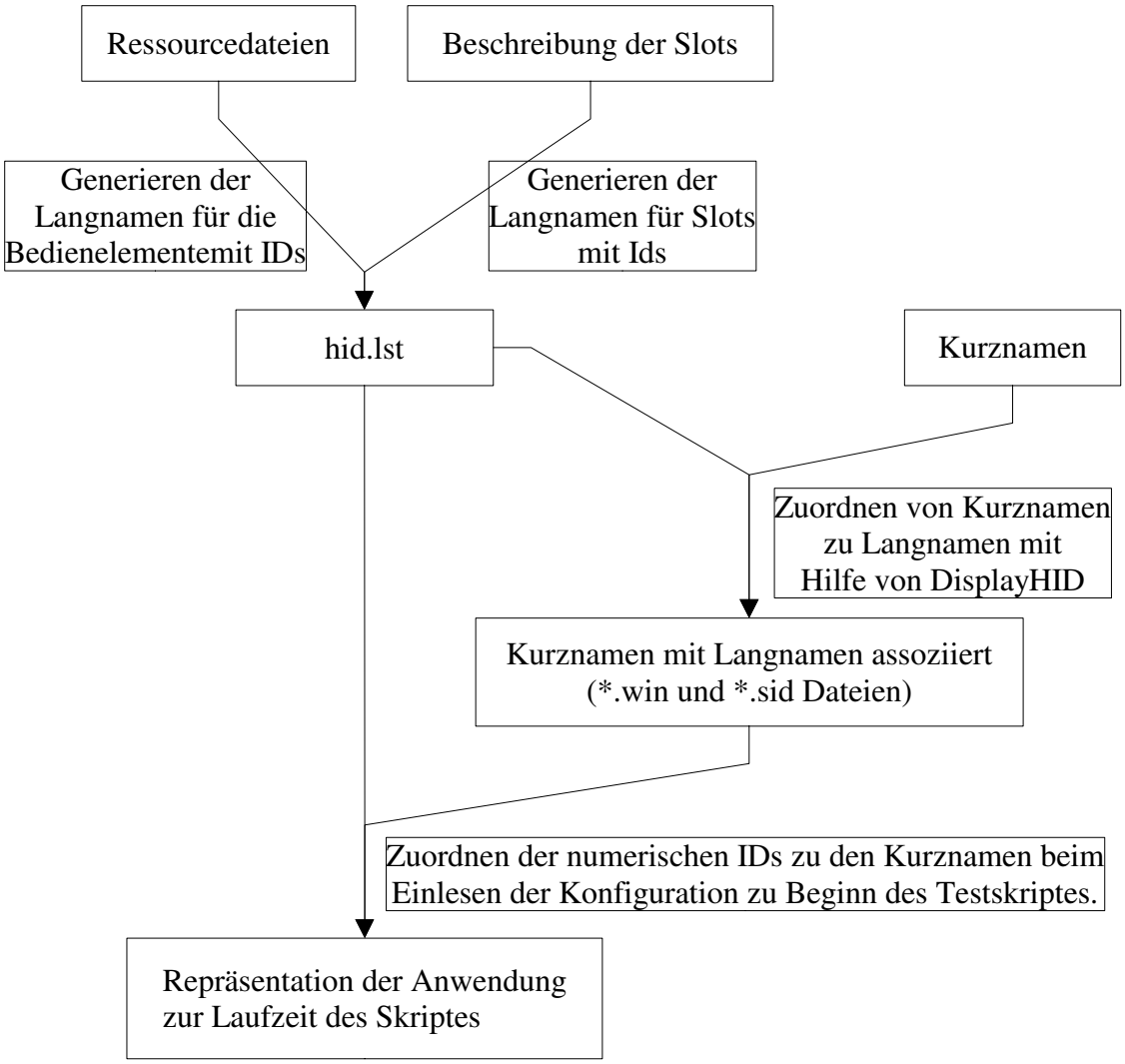

Abbildung 8: Zusammenhang von Kurzname, Langname und Numerischer ID

Die Zuordnung geschieht durch eine Tabelle, in der in jeder Zeile ein Kurzname mit einem Langnamen assoziiert wird. In einer Datei wird jeweils der *Kurzname* gefolgt von dem *Langnamen* eingetragen.

Die Bedienelemente eines Dialoges werden zu einem sogenannten Kontext zusammengefaßt. Dabei übernimmt jeweils der erste Eintrag eine Doppelrolle. Er dient einerseits dazu, auf das entsprechende Bedienelement zuzugreifen, andererseits identifiziert er auch den Kontext.

Das Unterteilen in Kontexte hat einen entscheidenden Vorteil. In zwei verschiedenen Kontexten können dieselben Kurznamen verwendet werden, was bei ähnlicher Funktionalität den Lernaufwand vermindert. Ohne Kontext müßten hier zwei ähnliche

Kurznamen für gleiche Funktionalität auf verschiedenen Dialogen vergeben werden, was häufige Fehler mit sich bringen würde, da der falsche Kurzname verwendet werden würde und das entsprechende Bedienelement dann nicht gefunden würde.

Treten innerhalb eines Kontextes zwei gleich Kurznamen auf, so wird für jeden doppelten Kurznamen eine Warnung in das Journal geschrieben. Der doppelte Eintrag wird dann ignoriert.

Objektorientierter Zugriff auf die Benutzungsoberfläche

Um die Bedienelemente von dem Skript aus ansprechen zu können, werden diese auf Objekte in BASIC abgebildet. Diese Objekte werden über die Kurznamen identifiziert. Zur Laufzeit wird dann der Typ des Bedienelementes ermittelt. Aus dem Typ ergibt sich ein spezifischer Satz von Methoden, die von dem Objekt zur Verfügung gestellt werden. Wird versucht, eine andere Methode aufzurufen, so wird ein Laufzeitfehler generiert. Darüber hinaus gibt es einen Satz von Methoden, die an jedem Bedienelement und damit an jedem Objekt definiert sind.

```
Objekt.Methode
```

Beispiel 8: Zugriff auf Methoden durch den Punkt-Operator

Die Syntax ist wie bei vielen objektorientierten Systemen durch den Punkt-Operator realisiert, wobei, wie in Beispiel 8 zu sehen ist, das Objekt mit einem Punkt mit der Methode verbunden wird.

Direkter Aufruf von Slots

Der Aufruf von Slots geschieht durch Schreiben des Kurznamens, der den Slot repräsentiert. Ein neues Dokument kann dann etwa durch Aufruf von

```
NewDirect
```

erzeugt werden.

In diesem Fall wird kein Parameter benötigt. Es wird standardmäßig ein Text-Dokument angelegt. Es kann jedoch ein optionaler Parameter mitgegeben werden, um auch die anderen Dokumenttypen zu erzeugen.

Parameter werden in diesem Zusammenhang nicht durch ihre Reihenfolge, wie bei vielen Programmiersprachen üblich, identifiziert. Vielmehr besitzen sie einen zusätzlichen Identifikationsschlüssel. Dieser Schlüssel ist wieder durch einen numerischen Wert

definiert. Ihm kann genau wie den Slots auch ein Kurzname zugewiesen werden. Oft wird der Standardparameter durch den selben numerischen Wert definiert, wie der Slot selbst, so daß es sich anbietet, die bereits erfaßten Kurznamen auch hierfür zu nutzen. Um den Aufruf eines Slots von seinem numerischen Wert syntaktisch zu unterscheiden, werden sämtliche Kurznamen der Slots zusätzlich durch Anhängen eines _ID als numerische Konstante zur Verfügung gestellt.

Aus obigem Beispiel ergibt sich dann zur Erzeugung eines Dokuments der Tabellenkalkulation der Befehl:

```
NewDirect NewDirect_ID, "scalc"
```

Ein anderes Beispiel ist das Öffnen eines Dokumentes. Durch Aufruf des Befehls

```
FileOpen
```

wird der 'Datei Öffnen' Dialog aufgerufen. Will man diesen aber nicht testen, so kann er auch umgangen werden, indem noch zusätzlich ein Dateiname mitgegeben wird. Das Dokument wird dann direkt geöffnet. Der Befehl lautet dann:

```
FileOpen Filename_ID, "MeinDokument.sdw"
```

Technische Realisierung der Zuordnung

Die Zuordnung von Kurznamen zu Langnamen wird in Textdateien gespeichert. Pro Zeile ist immer ein Kurzname mit einem Langnamen assoziiert. Dabei steht der Kurzname vorne. Trennzeichen ist eine beliebige Kombination aus Leerzeichen und Tabulatoren. Kommentare werden wie in BASIC durch ein Hochkomma oder das Schlüsselwort 'rem' eingeleitet und gelten bis Zeilenende. Es werden Dateien getrennt für die Slots und die Bedienelemente angelegt. Die Dateien für die Slots müssen auf '.sid' enden, während die Dateien für Bedienelemente auf '.win' enden müssen.

Für die Bedienelemente existieren noch einige syntaktische Erweiterungen. Eine Gruppierung zu einem Kontext wird durch Voranstellen eines Sternchens '*' eingeleitet. Alle Zeilen danach werden diesem Kontext zugefügt, bis ein weiterer Kontext durch diese Markierung begonnen wird oder das Dateiende erreicht ist.

Existieren ähnliche Dialoge, so müssen diese nicht jedesmal komplett notiert werden. Durch Voranstellen eines '+' statt eines '*' und Angabe eines Kurznamens eines bereits existierenden Kontextes wird der entsprechende Kontext kopiert und als Basis des

neuen Kontextes verwendet. Es gibt zum Beispiel mehrere Dialoge zum Öffnen einer Datei. All diese Dialoge enthalten dieselben Basiselemente zur Auswahl des Dateinamens, des Filters und anderer Einstellungen. Sie sind aber durch unterschiedliche Bedienelemente erweitert. Beispielsweise ist bei dem Dialog zum Einfügen einer Grafik noch eine Checkbox zum Aktivieren der Vorschau enthalten. Der Dialog könnte dann durch die in Beispiel 9 dargestellten Zeilen deklariert werden:

```
+EinfuegenGrafik DateiOeffnen
Vorschau SFX2:CHECKBOX:DLG_NEW_FILE:BTN_PREVIEW
```

Beispiel 9: Deklaration eines Dialoges durch Kopieren

Wie diese Dateien in das Skript integriert werden und wie die Kontexte angesprochen werden können, ist weiter unten beschrieben.

4.2.5 Befehle zur Steuerung des Testsystems

Es gibt noch einige Befehle, die diverse andere Bedeutungen haben. Eine Gruppe enthält Sonderfunktionen und Einstellungen, die zum Ausführen eines Skriptes sinnvoll oder gar notwendig sind. Eine andere Gruppe besteht aus Befehlen zur Diagnose des Testsystems und zur Unterstützung der Erstellung der Skripten.

Befehle zur Ausführung eines Skriptes

Zunächst müssen die Repräsentation der Benutzungsoberfläche und der Slots geladen werden. Dazu steht der Befehl 'use' zur Verfügung. Dieser Befehl wird jeweils mit einem Pfad zu einer Datei aufgerufen. Je nach Endung des Dateinamens wird der Inhalt der Datei unterschiedlich interpretiert. Dateien, die auf '.win' enden, enthalten Beschreibungen von Bedienelementen. Dateien mit '.sid' am Ende enthalten Definitionen von Slots.

Beim Einlesen dieser Dateien muß zunächst die Datei 'hid.lst' eingelesen werden. Sie enthält die Zuordnungen der Langnamen zu den numerischen IDs. Die Langnamen aus den Dateien zur Deklaration der Bedienelemente und Slots werden dann anhand dieser Tabelle ersetzt. Dabei können verschiedene Fehler auftreten. Einerseits kann es vorkommen, daß ein Langname in der Übersetzungstabelle nicht vorhanden ist. In diesem Fall kann es sein, daß das zugehörige Bedienelement entfallen ist. In äußerst seltenen Fällen ändern sich die Langnamen aber auch. Eine zweite Fehlerquelle besteht darin, daß Namen doppelt vergeben wurden. Dies kann sowohl in der Übersetzungstabelle als auch in den Deklarationsdateien vorkommen. Die Namen der geladenen Dateien werden

in das Journal aufgenommen. Treten Fehler beim Laden der Datei auf, so werden sie, als Warnungen klassifiziert, ebenfalls erfaßt.

Der Befehl 'Use' unterstützt noch eine dritte Dateinamenserweiterung zum Laden zusätzlicher Module. Endet eine Datei auf '.inc', so wird ein neues Modul mit Unterprogrammen geladen.

Sind die Dateinamen als relative Pfade angegeben, so wird in dem Basisverzeichnis des Testsystems danach gesucht. Dieses Verzeichnis kann über den 'Optionen Dialog' des Testtools eingestellt werden.

Beispiele zum Befehl 'Use' sind in Abbildung 10 dargestellt.

```
use "inc/tools.inc"
use "win/FileOpen.win"
use "sid/allsid.sid"
```

Beispiel 10: Der Befehl 'Use'

Nach dem Laden dieser Elemente kann die Verbindung zur Anwendung mit dem Befehl 'start' hergestellt werden. Dem Befehl werden der Pfad auf den Dateiname des Programmes und eventuelle Parameter mitgegeben.

```
Start "c:\Office50\soffice.exe", "c:\probetext.sdw"
```

Zunächst wird versucht, eine Kommunikation zu etablieren. Die Daten über gewünschten Computer und Port werden ebenfalls im 'Optionen Dialog' des Testtools eingestellt. Dabei ist es unerheblich, ob das Programm, das antwortet, dasselbe ist, das auch in dem Parameter übergeben wurde. Erst wenn dieser Versuch fehlschlägt, wird versucht, das Programm zu starten. In Abständen von zehn Sekunden wird nun erneut versucht, die Kommunikation zu etablieren. Erst wenn auch dies eine Minute lang fehlschlägt, wird ein Fehler generiert. Insbesondere kann der Befehl 'start' auch mit einem leeren String aufgerufen werden, dann wird nur eine Verbindung zu einem eventuell in Ausführung befindlichem Programm etabliert. Auch bei Ausführung auf verschiedenen Computern kann dieser Parameter leer gelassen werden, da es ohnehin nicht möglich ist, auf einem anderen Computer eine Anwendung zu starten.

Das Selektieren eines Kontextes geschieht mittels des Befehls 'Kontext'. Dazu wird der jeweilige Name des Kontextes einfach als Textparameter übergeben. Die zur Verfügung stehenden Bedienelemente sind anschließend auf diesen Kontext beschränkt. Wird

der Befehl mit einem leeren Text aufgerufen, so kann auf alle Kontextnamen zugegriffen werden. Die Syntax ist in Beispiel 11 verdeutlicht.

Für die Fehlerkorrektur existieren noch zwei Befehle, die die Anwendung wieder in

```
kontext "DateiOeffnen"
kontext ""
```

Beispiel 11: Skriptbefehl Kontext

einen definierten Zustand versetzen.

Der Befehl 'ResetApplication' bringt die Anwendung in den Grundzustand. Es werden alle Dokumente und Dialoge geschlossen. Als Ergebnis wird eine Zeichenkette geliefert, die eine Beschreibung der geschlossenen Fenster enthält. Diese Beschreibung sollte von dem Skript, zum Beispiel als Warnung, in das Journal aufgenommen werden. Können nicht alle Fenster geschlossen werden, so wird ein Fehler ausgelöst. Dies kann durch zu viele geöffnete Fenster entstehen oder durch Schleifen, die entstehen können, wenn das Schließen eines Dialoges oder Dokumentes wieder einen anderen Dialog öffnet. Wird beispielsweise ein verändertes Dokument geschlossen, so erscheint ein Dialog, der fragt, ob das Dokument gespeichert werden soll oder nicht. Da das Testsystem die Bedeutung des Textes auf dem Dialog nicht kennt, beantwortet es den Dialog mit einem der verfügbaren Buttons. Dabei geht es nach einem festen System vor. Wird durch Beantworten des Dialoges das Dokument nicht geschlossen, so wird ein erneuter Versuch unternommen, das Dokument zu schließen. Diesmal wird der dann wieder auftretende Dialog zum Speichern des Dokumentes mit einem anderen Button quittiert. Einer der Buttons führt dann zum Schließen des Dokumentes. Es kann aber dennoch vorkommen, daß sich dabei eine Schleife bildet und immer wieder dieselben Dialoge erscheinen, ohne daß das Dokument je geschlossen wird.

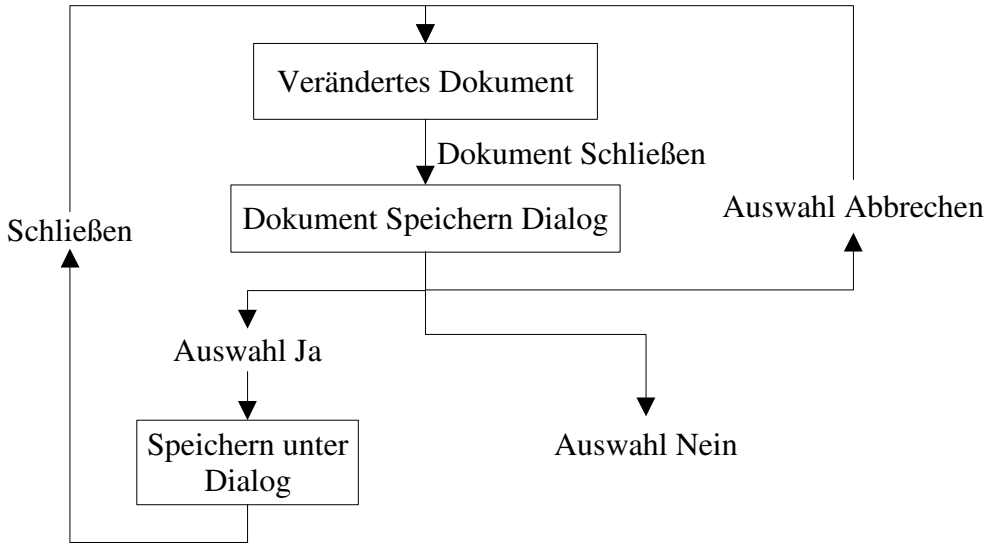

Abbildung 9: Kontrollfluß beim Schließen eines Dokumentes

Es gibt noch einige konzeptionelle Schwierigkeiten bei der Einbindung des Controllers in die Anwendung. Beide rühren daher, daß der Controller möglichst weit unten in der Klassenhierarchie angesiedelt werden sollte und möglichst wenig Code benutzen sollte. Dies ist sinnvoll, da er dadurch stabil läuft und nicht von großen Mengen potentiell instabilen Codes abhängig ist. Der Preis dafür ist, daß nicht alle Informationen, die für eine komplette Steuerung notwendig wären, zur Verfügung stehen und deshalb von dem Skript nachgeliefert werden müssen. Dies geschieht mit folgenden zwei Hilfsfunktionen.

Es können nicht alle Typen der Bedienelemente komplett erkannt werden, sondern nur die, die aus den Ressourcen (vgl. Kap. 3.1.2) geladen werden. Meistens werden diese Klassen als Basis für Weiterentwicklungen der Bedienelemente verwendet. Normalerweise kann auf die Methoden aber weiterhin zugegriffen werden, da sie an der Basisklasse virtuell definiert sind und somit die Weiterentwicklung auch an der Basisklasse aufgerufen werden kann. Es gibt jedoch Bedienelemente, die sich in ihrer Art und ihrem Funktionsumfang so stark von allen bisherigen unterscheiden, daß sie komplett neu implementiert werden. Als Basisklasse wird dann die sehr allgemeine Klasse 'Control' verwendet. Um die spezielle Funktionalität dieser Bedienelemente trotzdem zu erschließen, muß das Skript dem Controller explizit mitteilen, um welchen Typ von Bedienelement es sich handelt. Stößt der Controller dann auf ein Bedienelement vom Typ 'Control', so spricht er es gemäß der Vorgaben des Skriptes an. Zur expliziten Festlegung des Typs von Bedienelementen vom Ressourcetyp 'Control' wird der Befehl

'ControlType' verwendet. Er wird mit einer Konstante aufgerufen, die ebenfalls vom Testtool zur Verfügung gestellt wird. Es gibt nur zwei solcher Typen, so daß in Beispiel 12 alle möglichen Anwendungen illustriert sind.

```
ControlType CTBrowseBox
ControlType CTTreeListBox
```

Beispiel 12: Skriptbefehl ControlType

Bei der Ausführung von Slots werden diese gegebenenfalls an das Dokument weitergeleitet und dort ausgeführt. Teilweise wird aber ein Dokumenttyp als Teil eines anderen Dokumenttyps wiederverwendet. In diesem Fall kommt es vor, daß bestimmte Slots überlagert werden sollen. So wird beispielsweise das Textdokument auch dazu benutzt, den Text in einer Nachricht zu editieren. Die Funktion zum Speichern darf hierbei allerdings nicht von dem Textdokument selbst ausgeführt werden, da sonst nur der Text ohne die weiteren Informationen einer Nachricht gespeichert würde. Vielmehr muß der Aufruf vorher zu dem zugehörigen Nachrichtendokument umgeleitet werden.

Aufgrund interner Strukturen ist es nur möglich festzustellen, ob die Ausführung eines Slots erfolgreich war, wenn er direkt zu dem Dokument geschickt wird. Wird dagegen der Filter für die Umleitung einzelner Slots verwendet, steht diese Information nicht mehr zur Verfügung. Es kann auf Ebene des Controllers nicht festgestellt werden, ob eine Umleitung installiert ist. Somit muß die Verwendung der Umleitung ebenfalls von Hand eingestellt werden. Ein genereller Verzicht auf die Information, ob ein Slot erfolgreich ausgeführt werden konnte, scheidet hingegen aus, da die Fehlererkennung dadurch empfindlich beeinträchtigt werden würde.

Mit UseBindings kann die Umleitung beeinflußt werden. Übergibt man keinen Parameter oder TRUE, so wird sie eingeschaltet. Bei Übergabe von FALSE wieder deaktiviert. Beispiel 13 zeigt die Syntax.

```
UseBindings TRUE
UseBindings
UseBindings FALSE
```

Beispiel 13: Skriptbefehl UseBindings

Befehle zum Erstellen der Skripten und zum Debuggen des Testsystems

Der Befehl 'DisplayHID', zum Aufrufen der Programmierhilfe zum Zuordnen der hierarchischen Namen wird in Kapitel 4.7 beschrieben.

Ein weiterer Befehl erlaubt es, die gesamte Hierarchie der Fenster der Anwendung inklusive aller Bedienelemente abzufragen. Diese Funktion ist vor allem dann hilfreich, wenn es darum geht, daß an einem Bedienelement, das aus mehreren Fenstern besteht, an einem dieser Fenster eine ID vergeben wurde, an diesem Fenster aber nicht die Funktionalität des Bedienelementes zur Verfügung steht. Besonders in Verbindung mit dem Befehl 'ControlType' kommt es bei solchen Problemen regelmäßig zu Abstürzen der Anwendung.

Wird beim Übersetzen des Controllers ein C++ Define namens DEBUG gesetzt, so öffnet sich zu Beginn des Testvorgangs ein zusätzliches Fenster, in dem protokolliert wird, was im Controller gerade abläuft. Da dies ein Werkzeug zum Aufspüren von Problemen ist, ändert sich die Ausgabe gelegentlich, je nachdem, welche Informationen im Entwicklungsprozeß gerade von Bedeutung sind. Auf jeden Fall wird protokolliert, welche Befehle empfangen werden, welche Befehle ausgeführt werden, und welche Parameter übergeben wurden. Um dieses Fenster zu schließen und auch wieder zu öffnen stehen die Befehle 'nodebug' und 'debug' zur Verfügung.

Verwendet man diese Befehle, ohne den Controller mit dem C++ Define 'DEBUG' übersetzt zu haben, wird ein Fehler ausgelöst, da die Befehle dann nicht bekannt sind.

4.3 Plattformunabhängigkeit und Länderversionen

Es soll möglich sein, die Skripten auf unterschiedlichen Plattformen zu bearbeiten. Das Problem dabei ist, daß die Plattformen unterschiedliche Zeichensätze und Zeilenendekonventionen für Textdateien haben. Eine Lösung ist es, ein eigenes Dateiformat zu implementieren und je Plattform eine Routine zum Konvertieren bereitzustellen. Als Dateiformat wurde hier das DOS-Dateiformat gewählt. Der Zeichensatz wird dabei in den Standard-Zeichensatz von DOS umgewandelt und die Zeilenendekonvention von DOS verwendet. Auf diese Weise ist es auch möglich, die Dateien mit externen Editoren zu bearbeiten. Eventuell sind dabei allerdings falsche Sonderzeichen, wie zum Beispiel Umlaute, hinzunehmen. Ein Nachteil des Konvertierens der Zeichensätze ist es, daß Sonderzeichen dabei häufig verloren gehen, da sie nicht in allen verwendeten Zeichensätzen vorhanden sind, und somit nicht richtig konvertiert werden können.

Ein ähnliches Problem tritt bei der Kommunikation zwischen verschiedenen Betriebssystemen auf. Soll ein Text abgefragt oder gesetzt werden, müssen die Zeichensätze ebenfalls angepaßt werden. Es wird also auch bei der Kommunikation zwischen Testtool und Anwendung ein standardisierter Zeichensatz verwendet.

Ein weiterer Unterschied zwischen Plattformen ist der Aufbau von Zugriffspfaden zu Dateien. Unter Microsoft Windows und OS/2 werden die Verzeichnisse durch einen Backslash getrennt. Unter UNIX wird ein normaler Slash verwendet, und auf dem MacIntosh findet der Doppelpunkt Einsatz.

In den Skripten ist dies bei zwei Befehlen relevant. Bei dem Befehl 'use' muß der Pfad zu den Dateien in DOS-Konvention angegeben werden. Auf anderen Betriebssystemen wird er dann automatisch umgewandelt. Da der Pfad zu der Programmdatei der Anwendung aufgrund der unterschiedlichen Konzepte der Betriebssysteme in aller Regel stark unterschiedlich ist, muß dieser in der Konvention des Betriebssystems angegeben werden.

Die Handhabung von Groß-/Kleinschreibung bei Dateinamen unterscheidet sich auf einigen Plattformen. Microsoft Windows 95 und Unix erlauben die Vergabe von Namen mit Groß- und Kleinschreibung. Microsoft Windows 95 unterscheidet aber beim Zugriff nicht, während Unix dies sehr wohl tut. Es kann also passieren, daß eine Datei in einem Skript unter Microsoft Windows 95 erfolgreich verläuft, aber unter Unix fehlschlägt. Das Problem wird noch verschärft, da die meisten Dateien im Netzwerk gespeichert sind, die Fileserver jedoch teilweise selbständig von Groß- nach Kleinbuchstaben konvertieren. Dieses Verhalten ist unter Microsoft Windows 95 tolerierbar, führt aber unter Unix zum Fehlschlagen des Dateizugriffes. Um dieses Problem zu umgehen ist daher die Konvention entstanden, Dateinamen immer klein zu schreiben.

4.4 Entwicklungsumgebung

Das Testtool gliedert sich in die Erweiterung der Skriptsprache BASIC für das Fernsteuern der Anwendung und in die Entwicklungsumgebung für die Skripte. Die Entwicklungsumgebung basiert auf einem zu Testzwecken erstellten Programm, das zum Test des BASIC-Interpreters verwendet wurde. Es enthielt lediglich die Möglichkeit, Dateien zu laden und zu speichern sowie die Skripte auszuführen. Die Skripte konnten auch schrittweise durchgeführt werden. Dieses Programm wurde von etlichen Fehlern befreit und um einige zusätzliche Eigenschaften erweitert.

Um die Lesbarkeit der Skripte zu erhöhen, werden die einzelnen Befehle zunächst in ihre Bestandteile zerlegt. Diese Tokens werden entsprechend ihrer Art und Bedeutung unterschiedlich eingefärbt. Die Tokens entstammen jeweils einer der Kategorien Schlüsselwort, Symbol, String, Zahl, Interpunktion und Kommentar. Operatoren zählen dabei zur Interpunktion.

Die Möglichkeit des Debuggens von Skripten durch ihr schrittweises Ausführen wird durch Brechpunkte erweitert, an denen dann die Ausführung des Skriptes unterbrochen wird. Die Brechpunkte sind persistent, werden aber nur dann aktiviert, wenn die Datei, in der der Brechpunkt gesetzt ist, in einem Fenster geladen ist. Die Brechpunkte werden also nicht aktiviert, wenn ein Modul lediglich durch den Befehl 'use' geladen wurde. Die Brechpunkte werden links neben der entsprechenden Zeile des Skriptes durch einen roten Kreis angezeigt.

Um den Zustand der Variablen während der Ausführung des Skriptes verfolgen zu können, wird ihr Inhalt als Tiphilfe angezeigt, sobald der Mauscursor sich über einer Variablen befindet. Über das Kontextmenü, welches mit der rechten Maustaste aktiviert werden kann, ist es möglich, einen Dialog aufzurufen, der es erlaubt, den Variableninhalt zu verändern. Je nach Typ der Variable wird eine andere Eingabemöglichkeit verwendet, die die Eingabe von gültigen Werten erzwingt beziehungsweise unterstützt. Hat die Variable einen ganzzahligen Inhalt, so wird ein Bedienelement verwendet, das nur die Eingabe von ganzzahligen Werten im entsprechenden Wertebereich erlaubt. Im Fall eines booleschen Wertes werden nur zwei Radiobuttons angezeigt, die das Einstellen der beiden Wahrheitswerte ermöglichen. Bei Gleitkommazahlen wird bei Beenden der Eingabe überprüft, ob es sich um einen Wert im gültigen Wertebereich handelt. Ist dies nicht der Fall, so wird der Benutzer durch einen Dialog darauf aufmerksam gemacht und erhält die Möglichkeit, seine Eingabe zu korrigieren. Strings können ohne Prüfung direkt übernommen werden, da es für ihren Inhalt keine Beschränkung gibt. Eine Längenüberschreitung kann auch nicht vorkommen, da Strings in BASIC auf die Stringklasse abgebildet werden, die auch zur Eingabe bereits Verwendung findet.

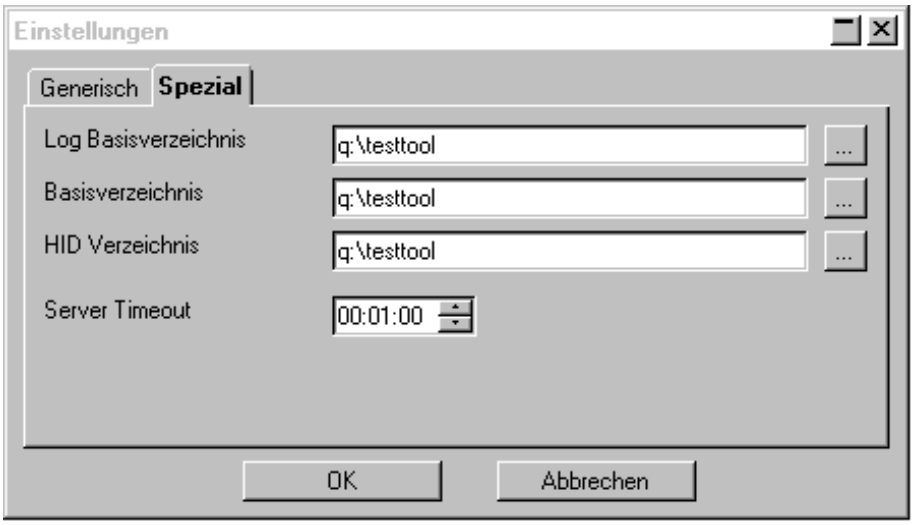

Abbildung 10: Dialog Einstellungen Tabpage 'Spezial'

Die Konfiguration des Testtools geschieht über einen 'Optionen Dialog'. Die möglichen Einstellungen sind in zwei Unterbereiche aufgeteilt. Für jeden der Bereiche wird eine separate *Tabpage* verwendet. Auf der ersten Tabpage werden die Pfade zu den Dateien, die das Testtool verwendet, eingegeben. Außerdem kann ein Timeout für die Befehle eingegeben werden. Die Eingabe der Pfade kann entweder direkt erfolgen, oder mittels eines Dialoges ausgewählt werden, der durch Klicken des Buttons rechts des Eingabefeldes aufgerufen werden kann. In Abbildung 10 sind die verschiedenen Einstellungsmöglichkeiten zu sehen. Die Einstellungen haben folgende Bedeutungen:

- Log Basisverzeichnis:
  In dem angegebenen Verzeichnis werden sämtliche Journale abgelegt.

- Basisverzeichnis:
  Das angegebene Verzeichnis wird als Basis des Zugriffspfades auf die mit dem Befehl 'use' geladenen Dateien verwendet.

- HID Verzeichnis:
  In diesem Verzeichnis wird die Datei 'hid.lst' gesucht. Sie enthält die Zuordnung von Langnamen zu den numerischen Ids.

- Server Timeout:
  Gibt die Zeit an, die das Testtool auf die Abarbeitung eines Befehls durch die Anwendung wartet. Wird die Zeit überschritten, so generiert das Testtool einen Fehler und bricht den aktuellen Befehl ab.

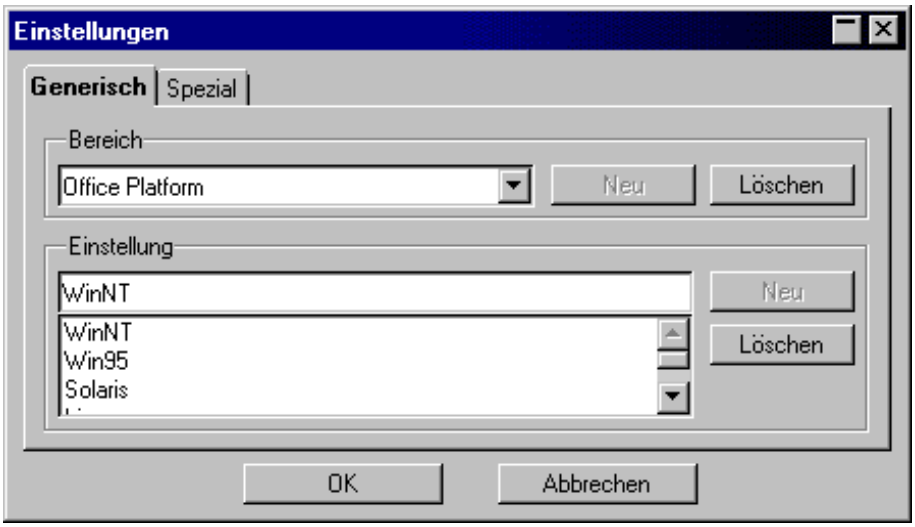

Abbildung 11: Dialog Einstellungen Tabpage 'Generisch'

Die zweite Tabpage enthält einen Mechanismus, der es erlaubt, Einträge in der Konfigurationsdatei zu machen, die später durch das Skript ausgelesen werden. In Abbildung 11 wird dem Schlüssel »Office Platform« der Wert »WinNT« zugewiesen. Der Wert kann aus den in der ListBox enthaltenen möglichen Werten ausgewählt werden. Neue Bereiche und Werte können in den jeweiligen Feldern eingetragen werden und durch Klicken des zugehörigen Neu-Buttons der jeweiligen Liste hinzugefügt werden.

4.5 Fernsteuern der Anwendung: Abarbeiten eines Skriptes

In diesem Abschnitt wird beschrieben, wie die Interaktion zwischen Testtool und Anwendung abläuft: Angefangen bei dem Mechanismus, wie der BASIC-Interpreter einen Befehl findet, über das Ausführen des Befehls in der Anwendung bis hin zur Interpretation der Ergebnisse.

Wenn der BASIC-Interpreter bei der Ausführung des Skriptes auf einen Bezeichner stößt, so ist unter Umständen nicht bekannt, was der Bezeichner repräsentiert. Der BASIC-Interpreter durchsucht zunächst seine internen Strukturen, in denen die Variablennamen und Funktionsnamen gespeichert sind. Findet er dort keine Entsprechung, so wird die Erweiterung zum Fernsteuern der Anwendung, das Testtool Objekt (TTO), befragt, ob ihm der Bezeichner bekannt ist. Das TTO enthält C++ Objekte, die die Sonderbefehle repräsentieren. Die Repräsentation der Benutzungsoberfläche und der Slots wird nicht als C++ Objekte in das TTO aufgenommen, da sie sich erstens mit jedem

Programmlauf ändern, da sie dynamisch geladen werden, und zweitens das Speichern in separaten Listen weniger Speicherplatz beansprucht. Das TTO durchsucht sich zuerst nach dem Bezeichner, wobei die Sonderbefehle und die zuletzt benutzten Slotaufrufe und Bedienelemente gefunden werden können. Ist diese Suche erfolglos, so werden die Listen, die, entsprechend den geladenen Dateien, die Repräsentation der Benutzungsoberfläche und der Slots enthalten, durchsucht. Wird ein Eintrag gefunden, so wird ein Suchergebnis erzeugt und an den BASIC-Interpreter zurückgegeben. Dieses Suchergebnis ist ein C++ Objekt, das im Falle eines Bedienelementes ein Objekt im BASIC-Interpreter darstellt und im Falle eines Slots oder eines Sonderbefehls eine Methode repräsentiert. Der BASIC-Interpreter weiß dadurch aber nur, daß dieser Name bekannt ist und welchen Typ er hat. Greift der BASIC-Interpreter auf diesen Bezeichner zu, so wird das TTO vorher darüber benachrichtigt und erhält so die Möglichkeit, den Wert des Bezeichners zu berechnen. Wird der Wert eines (BASIC-)Objektes erfragt, so wird der numerische Repräsentant des Bedienelementes zurückgegeben, der dem Objekt ja bekannt ist. Normalerweise wird allerdings an dem Objekt vom BASIC nach einer Methode gefragt und erst der Wert dieser Methode berechnet.

Das Vorgehen des TTO zur Berechnung der Ergebnisse von Methodenaufrufen an den Objekten, die die Bedienelemente repräsentieren, von Sonderfunktionsaufrufen und von Slotaufrufen ist ähnlich und wird deshalb gemeinsam beschrieben. Unterschiede werden an entsprechender Stelle aufgezeigt.

Zunächst muß dem Controller in der Anwendung mitgeteilt werden, welche Befehle er ausführen soll. Es werden dabei immer mehrere Befehle in einen Auftrag geschrieben. Es gibt vier verschiedene Befehlsklassen:

1. StatementControl:
   Dient dem Zugriff auf die Bedienelemente.

2. StatementSlot:
   Dient dem Aufruf von Slots.

3. StatementCommand:
   Dient dem Ausführen des Sonderbefehls.

4. StatementFlow:
   Enthält Befehle zum Synchronisieren des Testtools mit dem Controller.

Ein Auftrag an den Controller setzt sich immer aus folgenden drei Befehlen zusammen:

1. Um die sichere Zuordnung der Ergebnisse zu dem entsprechenden Auftrag zu gewährleisten, wird zunächst ein Befehl mit einer fortlaufenden Sequenznummer übertragen. Diese Sequenznummer ist auch im Ergebnis enthalten.

2. Dem eigentlichen Befehl, der vom Controller ausgeführt werden soll. Je nachdem, ob es sich um einen Slotaufruf, oder um einen Methodenaufruf an einem Bedienelement handelt, werden hier unterschiedlich Befehlsklassen erzeugt.

3. Abschließend steht ein Befehl, der den Controller anweist, die gesammelten Ergebnisse zurück zu senden.

Der erste und dritte Befehl sind von der Befehlsklasse StatementFlow. Der zweite Befehl kann aus einer der anderen drei Befehlsklassen stammen.

Diese Befehle werden dann zum Controller übertragen. Dieser trägt die Befehle zunächst nur in seine Befehlswarteschlange ein. Die Ausführung der Befehle erfolgt wie in Kapitel 4.1 beschrieben über den Idle-Handler.

Je nach Befehlsart unterscheiden sich die zur Ausführung notwendigen Schritte. In den folgenden Abschnitten werden die Vorgehensweisen für die einzelnen Befehlsklassen beschrieben.

- StatementControl zum Zugriff auf die Bedienelemente.
  Um auf ein Bedienelement zugreifen zu können, muß es zunächst gefunden werden. Dazu wird die Fensterstruktur, wie sie in Kapitel 3.1.3 beschrieben ist, durchsucht und an jedem Fenster überprüft, ob es die gewünschte Unique ID besitzt. Wird bei diesem Vorgang ein Bedienelement gefunden, das zwar die gewünschte ID besitzt, aber nicht bedienbar ist, sei es, weil es nicht sichtbar ist, oder sich auf einem Dialog befindet, der gerade keine Eingaben entgegennimmt, so wird dieses Bedienelement zunächst vorgemerkt. Wird während der gesamten Suche kein Bedienelement mit passender ID gefunden, das auch bedienbar ist, so wird ein Fehler generiert. Abhängig davon, ob ein Bedienelement vorgemerkt wurde oder nicht, wird in dem Fehlertext darauf hingewiesen, warum das Bedienelement nicht bedienbar ist beziehungsweise daß es gar nicht vorhanden ist.
  Konnte ein gültiges Bedienelement gefunden werden, so wird zunächst sein Ressourcetyp festgestellt und dann der entsprechende Code zur Behandlung der entsprechenden Methode an diesem Typ aufgerufen.
  In diesem Code werden zunächst eventuelle Parameter auf ihre Gültigkeit hin überprüft. Gegebenenfalls wird ein Fehler generiert und die Bearbeitung der Methode

abgebrochen. Ansonsten wird danach die entsprechende Methode an dem C++ Objekt aufgerufen und ein eventueller Ergebniswert gespeichert.

- StatementSlot
  Der Aufruf eines Slots geschieht über eine zentrale Methode an der Anwendung. Übergeben werden lediglich die numerische ID des Slots sowie eventuelle Parameter. Kann der Slot nicht ausgeführt werden, so wird ein Fehler generiert.

- StatementCommand
  Die einzelnen Befehle dieser Befehlsklassen weisen keinerlei Gemeinsamkeiten auf. Es wird einfach der entsprechende Code ausgeführt, der auch für Fehlergenerierung und Bereitstellen des Ergebnisses verantwortlich ist.

- StatementFlow
  Diese Befehlsklasse enthält lediglich zwei Befehle, auf die näher eingegangen werden soll.

  1. der Befehl zum Schreiben der Sequenznummer
     Dieser Befehl schreibt die Sequenznummer, die von dem TTO übermittelt wurde, als erstes in das Ergebnis.

  2. Der Befehl, der den Controller veranlaßt, die Ergebnisse an das TTO zurückzuschicken

Das TTO analysiert die zurückgesendeten Daten. Wird eine Sequenznummer in dem Datenstrom entdeckt, so wird sie mit der aktuellen Sequenznummer verglichen und eventuell ein Fehler generiert. Danach wird die Sequenznummer um 1 erhöht.

Werden Fehler zurückgeschickt, so werden diese ebenfalls an das BASIC weitergeleitet. Ein eventueller Ergebniswert wird in das C++ Objekt eingetragen.

Die Kontrolle wird dann an den BASIC-Interpreter zurückgegeben, der dann in seiner Ausführung des Skriptes fortfährt. Wurde ein Fehler erzeugt, so wird die entsprechende Fehlerbehandlung aufgerufen.

4.6 Präsentation der Testergebnisse im Journalfenster

Im Journal werden alle Ergebnisse aller Testläufe eines Skriptes gesammelt. Neue Testläufe werden dabei immer oben eingefügt. Das Journal wird in einer Datei mit demselben Namen wie das Skript gespeichert, nur die Namenserweiterung wird auf '.res' geändert. Aus 'test.bas' wird also 'test.res'. Die Journaldateien stehen alle in einem

gesonderten Verzeichnis, so daß es bei gleichzeitigen Testläufen desselben Skriptes nicht zu Problemen kommt. Dieses Verzeichnis kann im 'Optionen Dialog' eingestellt werden.

Der Übersichtlichkeit wegen wird das Journal in Form einer baumartigen Struktur präsentiert. Die einzelnen Zweige können dabei ausgeblendet werden, so daß nur noch ihre Wurzel sichtbar ist.

In der obersten Ebene werden die einzelnen Testläufe zusammen mit ihrem Startzeitpunkt aufgelistet. In der Ebene darunter befindet sich ein Eintrag, der Informationen zum Laden der weiteren Konfiguration innerhalb des Unterprogrammes 'LoadIncludeFiles' enthält. Es werden alle geladenen Dateien aufgelistet, ergänzt durch Fehler, die beim Zuordnen der Kurznamen zu den Langnamen auftreten. Gefolgt wird er von einem Eintrag für jeden Testfall, der ausgeführt wurde.

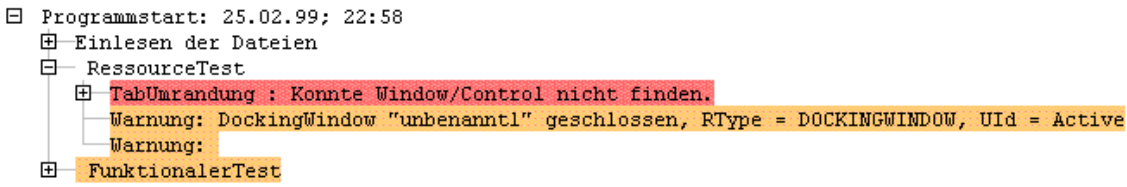

Abbildung 12: Journal nach der Durchführung eines Testlaufes

Warnungen, Kommentare und Assertions aus der Anwendung werden auf der Ebene eingefügt, auf der sich das Skript gerade befindet, also entweder als Unterpunkte der Testläufe oder auf gleicher Ebene mit ihnen.

Fehler werden immer innerhalb eines Testfalles eingefügt. Befindet sich das Skript in keinem Testfall, so wird ein Pseudotestfall mit einer entsprechenden Bezeichnung generiert. Dies kann nur dann auftreten, wenn innerhalb eines Try-Catch-Endcatch Konstruktes ein Fehler mit dem Befehl 'ErrorLog' in das Journal geschrieben wird.

Wird ein Fehler bei Abbruch eines Testlaufes generiert, so enthält er weitere Unterpunkte, die den *Call Stack* zum Zeitpunkt des Auftretens des Fehlers enthalten. Somit kann bei Fehlern, die in Unterprogrammen auftreten, erkannt werden, von wo das Unterprogramm aufgerufen wurde und damit die Fehlerursache leichter eingegrenzt werden.

Die Informationen im Journal sind außerdem farblich nach Art des Eintrages unterschieden. Fehler werden auf rotem Untergrund dargestellt. Die Assertions aus der Anwendung werden auf gelbem Untergrund mit grünen Punkten dargestellt und War-

nungen haben einen orangeroten Hintergrund. Alle anderen Eintragungen werden schwarz auf weiß dargestellt.

Um die Relevanz von Ästen auch dann zu erkennen, wenn sie ausgeblendet sind, wird die Wurzel eines zugeklappten Astes so dargestellt wie der wichtigste Unterpunkt. Höchste Relevanz haben Fehler, gefolgt von Assertions und Warnungen. Wird also ein Testfall, der Fehler enthält, zugeklappt, so wird der Eintrag statt auf weißem Hintergrund jetzt auf rotem Hintergrund dargestellt.

Um das Journal übersichtlich zu halten und nach einem Testlauf einen schnellen Überblick zu erlauben, werden automatisch alle fehlerfrei ausgeführten Testfälle kollabiert, so daß nur ihr Wurzeleintrag sichtbar ist. Ist ein Fehler aufgetreten, wird eine Ebene tiefer angezeigt, so daß die Fehlermeldung direkt eingesehen werden kann.

Abbildung 12 zeigt das Journal, wie es nach dem Testlauf des Beispielprogrammes aus dem Anhang entsteht. Zu sehen sind der Eintrag mit den Informationen zum Initialisieren des Skriptes, sowie die Testfälle »RessourceTest«, »FunktionalerTest« und »KonsistenzTest«. Die dunkel schattierten Einträge enthalten Fehler, die etwas helleren Einträge enthalten Warnungen. Der Testfall »FunktionalerTest« ist selbst keine Warnung sondern enthält lediglich Warnungen und wird deshalb auch wie eine Warnung schattiert. Die Warnungen sind nicht sichtbar, da nur Testfälle mit Fehlern automatisch eingeblendet werden.

Alle Einträge enthalten zusätzlich Informationen über ihren Herkunftsort, und zwar die Datei und Zeile im Skript beziehungsweise in den Konfigurationsdateien, die den Fehler verursacht hat. Durch doppeltes Klicken auf den entsprechenden Eintrag wird die entsprechende Datei geladen und der Cursor an die entsprechende Zeile positioniert.

Das Löschen von Einträgen im Journal ist auch möglich. Ein Ast wird dabei immer mit allen in der Hierarchie darunter angeordneten Informationen gelöscht.

4.7 Programmierhilfe zum Zuordnen der hierarchischen Namen (DisplayHID)

Ein gravierendes Problem des Testsystems ist es, daß die gesamte Benutzungsoberfläche der Anwendung zuerst deklariert werden muß. Um dies zu ermöglichen und so weit wie möglich zu vereinfachen, gibt es einen besonderen Modus des Testsystems, in dem Informationen über die Benutzungsschnittstelle angezeigt werden. Mit Hilfe dieser Informationen kann die Zuordnung der Kurznamen zu Langnamen (Vergl. Kap. 4.2.4)

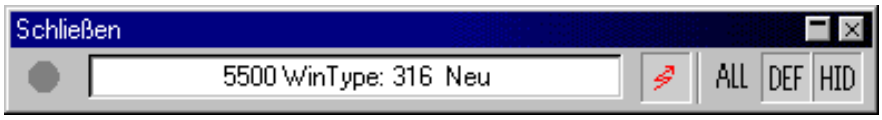

Abbildung 13: Auswahl der Bedienelemente in der Anwendung

wesentlich einfacher gestaltet werden. Gleichzeitig kann auch überprüft werden, welche Bedienelemente schon erfaßt wurden. Über den Befehl DisplayHID wird dieser Modus aktiviert. Der Name DisplayHID ist von der Funktion abgeleitet, nämlich dem Anzeigen der Hilfe IDs (HID). Durch Aktivieren des Befehls erscheint zunächst in der Anwendung das in Abbildung 13 abgebildete Fenster.

Die zur Bedienung der grundlegenden Funktionalität benötigten Bedienelemente sind der Button ganz links und das Anzeigefeld daneben. Mit dem Button kann der Modus zum Anzeigen der Informationen aktiviert werden. Es gibt dabei zwei Möglichkeiten. Durch einen normale Klick darauf wird der Modus permanent eingeschaltet, bis er durch einen erneuten Klick wieder ausgeschaltet wird. Die zweite Möglichkeit besteht darin, den Button zu ziehen, indem man den Mousebutton gedrückt hält und die Maus dann über das gewünschte Bedienelement bewegt. Wird der Mousebutton losgelassen, wird der Anzeigemodus wieder deaktiviert. Während der Modus aktiviert ist, wird der Button rot dargestellt.

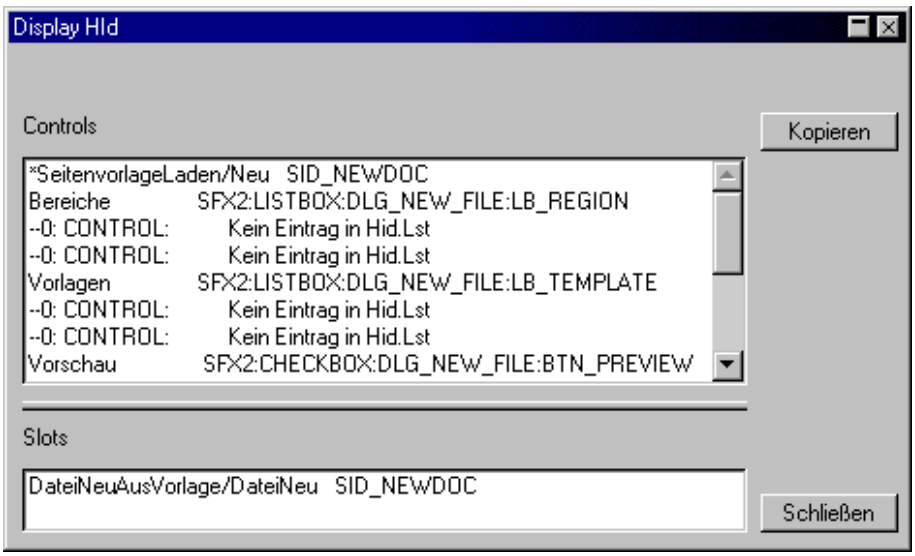

Abbildung 14: Anzeige der erweiterten Informationen

Der Anzeigemodus bewirkt, daß in dem Anzeigefeld Informationen über das Bedienelement, über dem sich der Mauszeiger gerade befindet, angezeigt werden. Es wird immer die Unique ID zusammen mit dem Inhalt oder Name des Bedienelementes sowie

seinem numerischen Typ angezeigt. Weicht die Unique ID von der Hilfe ID ab, so wird die Hilfe ID in der Titelzeile zusätzlich angezeigt.

 Der abgesetzte Button rechts neben dem Anzeigefeld aktiviert das Übermitteln der Daten zum Testtool. Dort wird dann automatisch ein Dialog geöffnet, in dem zusätzliche Informationen angezeigt werden. Im Gegensatz zu der Anzeige der Daten in der Anwendung werden hier neben dem aktuellen Bedienelement zusätzlich noch alle Bedienelemente angezeigt, die sich auf dem aktuellen Bedienelement befinden. So kann zum Beispiel ein ganzer Dialog der Anwendung auf einmal angesehen werden. Über die drei Buttons ganz rechts kann die Menge und Art der Informationen noch genauer bestimmt werden.

Wichtigstes Element des Dialoges zum Anzeigen der Informationen ist ein Bedienelement, in dem die Informationen aus der Anwendung angezeigt werden. Es ist in zwei Teile unterteilt, so daß neben den Bedienelementen auch noch eventuelle Slots angezeigt werden können. Der Sinn dieser Anzeigemöglichkeit ist nicht auf den ersten Blick zu erkennen, da nur Informationen zur Benutzungsoberfläche von der Anwendung geliefert werden. Häufig ist es aber so, daß die Dialoge dieselbe ID tragen, die auch zur Identifizierung des Slots benutzt wird, die den Dialog aufruft. Somit kann auch der Slot angezeigt werden, falls er deklariert wurde.

Das Verhältnis der beiden Teile zueinander kann durch Ziehen der Trennlinie geändert werden. Dadurch kann der zur Anzeige der Informationen zur Verfügung stehende Platz den momentanen Bedürfnissen des Benutzers des Testsystems angepaßt werden. Wird dabei ein Teil sehr klein, so wird zunächst der Beschreibungstext ausgeblendet, so daß mehr Platz für die Daten verfügbar ist. Es kann aber auch einer der Teile komplett ausgeblendet werden. Zusätzlich kann die Größe des gesamten Dialogs verändert werden. Die Bedienelemente werden dann automatisch auf die neue Größe adaptiert.

Die Informationen, die zu den einzelnen Bedienelementen angezeigt werden, hängen von dem Status der rechts angeordneten Buttons ab. Es können damit zusätzliche Informationen über die bereits vorgenommenen Deklarationen angezeigt werden. Die Daten hierfür werden aus den im aktuellen Programmlauf geladenen Definitionen entnommen. Da für den normalen Ablauf der Skripten eine Zuordnung von Namen, den Kurznamen, zu numerischen IDs erfolgt, und hier der umgekehrte Weg beschritten werden soll, müssen die Listen zunächst invertiert werden. Generell gilt dabei folgendes Vorgehen: Es wird eine Liste gebildet, die nach der numerischen ID sortiert wird. Zu jeder ID werden außerdem der Kurzname und ein eventueller Kontext gespeichert. Es kann auch vorkommen, daß dieselbe numerische ID mehrfach verwendet wird und

somit mehrere Kurznamen trägt. In diesem Fall werden alle diese Kurznamen getrennt durch einen Schrägstrich '/' aufgeführt. Dieser Zustand kann dadurch entstehen, daß identische Ressourcen an mehreren Stellen in der Anwendung benutzt werden. Im Falle des Datei Dialoges ist dies der Fall. Er wird zum Öffnen und Speichern verwendet. Außerdem werden an vielen verschiedenen Stellen in der Anwendung Dateien geöffnet, wofür ebenfalls immer derselbe Dialog, erweitert um zusätzliche Bedienelemente, benutzt wird. Ein zweiter Grund für das Auftreten mehrerer Kurznamen zu einer numerischen ID kann die doppelte Deklaration ein und desselben Dialoges sein. In diesem Fall sollte dann eine der Deklarationen entfernt werden. Eine dritte Möglichkeit besteht darin, daß der Entwickler eine ID doppelt vergeben hat oder eine, die ihm nicht zugewiesen wurde.

Zum Anzeigen der Informationen werden mehrere Listen aufbereitet. Eine Liste enthält sämtliche numerischen IDs der Bedienelemente. Für die Einträge, die einen Kontext beschreiben, wird noch eine Liste mit den Elementen des jeweiligen Kontextes verwaltet. Eine weitere Liste wird aus den Slots generiert. Um auch die Langnamen anzeigen zu können, muß die entsprechende Datei neu eingelesen werden, da die Informationen dazu nach Einlesen der Bedienelement und der Slots wieder aus dem Speicher gelöscht wurden.

Im einzelnen haben die Buttons folgende Funktion:

Normalerweise werden nur relevante Bedienelemente von der Anwendung übertragen. Relevant sind nur die Bedienelemente, die auch angesprochen werden können. Es ist aber so, daß einige Bedienelemente durch Gruppierung von mehreren anderen Bedienelementen entstehen. Diese sollen aber nicht einzeln sondern nur als Gesamtheit bedient werden. Außerdem existieren noch Elemente, die überhaupt nicht bedient werden können und normalerweise unterdrückt werden. Hierzu zählen Trenner in ToolBoxen und Texte, die einfach nur der Beschreibung dienen. Der mit 'ALL' beschriftete Button veranlaßt die Anwendung, sämtliche dieser Elemente mit zu übertragen.

Ist der Button mit der Beschriftung 'DEF' eingedrückt, so wird im Testtool versucht, den zugehörigen Kurznamen zu ermitteln. Diese Funktion kann genutzt werden, um zu prüfen, ob ein Dialog oder ein anderes Bedienelement schon in die Deklaration aufgenommen wurde. Es ist dabei möglich, durch geschickte Auswahl des zu zeigenden Bedienelementes in der Anwendung zu prüfen, ob das Bedienelement in einem bestimmten Kontext deklariert wurde, beziehungsweise, ob es überhaupt schon irgendwo deklariert wurde. Dafür wird im Fall des ersten übermittelten Eintrages

zunächst geprüft, ob es sich um die Basis eines Kontextes handelt, also üblicherweise um einen Dialog. Dazu wird im globalen Kontext gesucht, ob die ID vorhanden ist. Wird ein Eintrag gefunden, so wird der Suchkontext für die weiteren Einträge auf die Elemente dieses Kontextes eingeschränkt und dem Kurznamen für die Anzeige ein Sternchen '*' vorangestellt, um die aus den Deklarationsdateien bekannte Syntax nachzuahmen. Durch diese Einschränkung des Suchkontextes kann es passieren, daß einem Bedienelement bei Anzeige innerhalb eines Kontextes kein Kurzname zugeordnet werden kann, wenn man aber nur dieses eine Bedienelement anzeigen läßt, dieses sehr wohl einen Kurznamen zugeordnet bekommt. Das bedeutet dann, daß in einem anderen Kontext ein Bedienelement mit derselben ID existiert. Wird kein Kurzname gefunden, so wird statt dessen die numerische ID zusammen mit dem Ressourcetyp angezeigt. Zusätzlich wird ein Text angezeigt, der das Identifizieren des Bedienelementes vereinfachen soll. Bei einer CheckBox wäre das der zugehörige Text, bei einem Dialog der Inhalt der Titelzeile. Besitzt das Bedienelement keinen Text, so wird versucht, einen Hilfetext auszulesen, der dann zur Identifizierung dienen kann. Ist der Button nicht eingedrückt, so wird immer die numerische ID angezeigt

Das Eindrücken des Buttons 'HID' veranlaßt das Testtool, den zu der numerischen ID gehörenden Langnamen anzuzeigen. Kann der zugehörige Langname nicht gefunden werden, so wird der Benutzer durch einen entsprechenden Kommentar darauf hingewiesen.

Sollen die Informationen zum Erstellen der Deklarationsdateien verwendet werden, so können die gewünschten Einträge in der Liste selektiert werden und durch Betätigen des Copy Buttons in das Clipboard kopiert werden. Mit dem Close Button wird der Dialog erwartungsgemäß geschlossen.

Während der ganzen Zeit bleibt die Anwendung voll funktionsfähig und kann normal weiter benutzt werden. Der Benutzer des Testsystems muß also nicht für jeden Dialog den Testmodus neu aktivieren. Dieser Informationsmodus wird beendet, indem die Fenster geschlossen werden.

4.8 Programmierhilfe zur Erstellung der Skripte

Es soll eine interaktive Eingabemöglickeit geschaffen werden, mit der Befehle zur Bedienung von Bedienelementen in ein Skript eingefügt werden können. Die Anwendung wird dazu in einen speziellen Modus geschaltet, der es erlaubt, an jedem Bedienelement ein Kontextmenü aufzurufen, das die daran unterstützten Methoden enthält. Nach Auswahl der gewünschten Methode werden eventuell weitere Parameter in einem

Dialog erfragt. Der aktuelle Zustand des Bedienelementes dient dabei als Vorlage für die angebotenen Werte. Wird dieser Dialog bestätigt, so wird im Testtool ein entsprechender Befehl an die aktuelle Cursorposition im Skript eingefügt.

Der zur Laufzeit des Skriptes eingestellte Kontext ist zum Zeitpunkt des Aufzeichnens der Befehle nicht bekannt. Die Programmierhilfe erzeugt deshalb zu Beginn einer Sequenz immer einen Kontext-Befehl. Wird während der Aufzeichnung ein neuer Kontext betreten, oder wird der Cursor im Testtool an eine andere Position bewegt, so werden weitere Kontext-Befehle erzeugt.

5 Test des Testsystems - Ergebnisse

In diesem Kapitel soll abschließend zusammengestellt werden, inwieweit die Forderungen durch das Testsystem erfüllt werden. Ergänzend wird ein Testlauf durchgeführt, und seine Ergebnisse werden gezeigt.

5.1 Bedienbarkeit der Bedienelemente

Basis des Testsystems und wichtige Forderung war es, alle Bedienelemente in vollem Funktionsumfang ansteuern und abfragen zu können. Diese Forderung wurde größtenteils erfüllt. Es gibt jedoch einige Bedienelemente, die aufgrund verschiedener Probleme nicht oder nur eingeschränkt zu bedienen sind.

Wie in Kapitel 4.2.5 unter dem Befehl 'ControlType' beschrieben, existieren weiterentwickelte Bedienelemente, deren neue Funktionalität von der Basisklasse aus nicht zu erreichen ist. Teilweise kann dieses Problem durch explizites Setzen des Typs mittels des Befehls 'ControlType' umgangen werden. Es gibt jedoch auch Bedienelemente, die von dem Controller nicht erreicht werden können, da sie in der Projekthierarchie oberhalb des Controllers angesiedelt sind. Diese Bedienelemente können dann nur über die an jedem Bedienelement verfügbaren Methoden angesprochen werden. Es handelt sich dabei um allgemeine Methoden zum Abfragen des Zustandes und Methoden zum Generieren von Elementarereignissen. Bei diesen wenigen Bedienelementen ist also nur eine geringfügige Verbesserung gegenüber der ursprünglichen Lösung festzustellen. Ein Vorteil ist jedoch, daß die Elementarereignisse genau einem Bedienelement zugeordnet werden, und damit die Schwierigkeiten bei der Synchronisation von Testsystem und Anwendung aus Kap. 2.2 behoben werden.

Weitere Probleme treten bei der Bedienung von Menüs auf. Die Identifizierung und damit Referenzierung der einzelnen Menüpunkte ist schwierig, da sie teilweise dynamisch erzeugt werden, und somit keine statischen IDs vorhanden sind anhand derer sie eindeutig identifiziert werden können. Menüeinträge, die ein Untermenü beinhalten, können überhaupt nicht identifiziert werden, da sie keine ID haben. Eine Identifikation über den Text, der angezeigt wird, ist nicht sinnvoll, da die Skripte dadurch sprachabhängig werden. Diese Einschränkung ist aber nur insofern problematisch, daß der Inhalt der Menüs nicht überprüft werden kann. Das Auslösen eine Menüpunktes wird immer

sofort in den Aufruf eines Slots umgewandelt. Die volle Funktionalität kann also auch durch Aufrufen von Slots erschlossen werden.

5.2 Plattformunabhängigkeit

Die Plattformunabhängigkeit ist gewährt. Es ist sowohl möglich die Skripte auf jeder beliebigen Plattform zu erstellen und zu bearbeiten, als auch dort auszuführen. Die Ausführung des Testtools und der zu testenden Anwendung kann auf unterschiedlichen Computern stattfinden. Dabei können entweder beide Computer mit derselben oder einer unterschiedlichen Plattform arbeiten.

Durch die plattformübergreifende Verwendung der Testskripte ergeben sich Probleme bezüglich der unterschiedlichen Zeichensätze der einzelnen Plattformen. Diese Probleme treten an zwei Stellen zutage. Beim Laden und Speichern der Skripte muß der Zeichensatz umgewandelt werden. Wenn das Testtool und die Anwendung auf unterschiedlichen Plattformen ausgeführt werden, so muß bei Übertragung von Texten ebenfalls der unterschiedliche Zeichensatz berücksichtigt werden. Wie in Kapitel 4.3 beschrieben, werden für beide Fälle Vorkehrungen getroffen.

Es gibt auf einigen Plattformen Systemdialoge, die nicht nachgebildet werden, da sie von variablen Komponenten der jeweiligen Installation des Betriebssystemes zur Verfügung gestellt werden. Es handelt sich hierbei um den Dialog 'Eigenschaften' eines Druckers unter Microsoft Windows. Der Dialog wird von jedem Druckertreiber individuell zur Verfügung gestellt. Dieser Dialog kann von dem Testsystem nicht bedient werden, da er zusammen mit den sich darauf befindlichen Bedienelementen nicht in der Hierarchie der Fenster der Anwendung enthalten ist (vgl. Kapitel 3.1.3). Wird er durch das Testsystem geöffnet, bleibt er solange geöffnet, bis er manuell geschlossen wird. Unter den Unix-Plattformen wird kein derartiger Dialog vom Betriebssystem angeboten. Der eigene Dialog, der an seine Stelle tritt, kann dann auch von dem Testsystem wieder bedient werden.

5.3 Sprachunabhängigkeit

Die Sprachunabhängigkeit des Testsystems ist so lange erfüllt, wie nicht explizit auf Texte zugegriffen wird. Außer zu inhaltlichen Überprüfungen, wie beispielsweise dem Vergleichen des Inhalts einer Liste mit gespeicherten Werten, wird dies aber nur in folgendem Fall benötigt: Die gezielte Auswahl eines Eintrages aus einer Liste kann nur über deren Position oder über den Text selbst geschehen. Es kann jedoch vorkommen,

daß sich die Reihenfolge der Einträge in einer anderen Landessprache ändert. Beispielsweise durch Sortierung, wie es bei den Absatzvorlagen der Fall ist. Es muß also entweder der Index oder der zum Selektieren verwendete Text angepaßt werden.

Sollen Tastaturkürzel getestet werden, tritt ein ähnliches Problem auf. Da die Tastaturkürzel möglichst an die jeweilige Landessprache angelehnt sind, sind sie folglich nicht in allen Sprachversionen identisch.

Neben diesen offensichtlichen Problemen kann es aber auch vorkommen, daß sich der Funktionsumfang zwischen verschiedenen Sprachversionen unterscheidet. Auffälliges Beispiel ist, daß in einigen Sprachen des nahen und fernen Ostens die Formatierung von Texten nicht wie bei uns üblich zeilenweise von oben nach unten geschieht, sondern zusätzlich auch noch spaltenweise von rechts nach links. Zum Umschalten dieser Schreibweisen müssen speziell für diese Versionen weitere Bedienelemente vorgesehen werden. Eine sprachspezifische Anpassung der Skripte ist dann unumgänglich.

5.4 Konsistenz der Benutzungsoberfläche

Das Testsystem ermöglicht es durch geeignete Skripte, die Konsistenz der Benutzungsoberfläche zu prüfen. Als Basis hierfür kann beispielsweise der Ressourcetest dienen, da er alle Dialoge der Anwendung aufruft. Es kann dann für jeden einzelnen Dialog eine Testfunktion aufgerufen werden, die verschiedene Prüfungen vornimmt. Unterstützend kann hierbei auch die Option zum Dialogtest eingeschaltet werden, die in einer Debug-Version der Anwendung enthalten ist. Der Dialogtest überprüft die Anordnung der Bedienelemente auf dem Dialog und die korrekte Vergabe von Tastaturkürzeln (vgl. Kap. 3.3).

Der Testcase »KonsistenzTest« aus dem im Anhang angegebenen Beispielprogramm führt einen Konsistenztest anhand des Dialoges 'Navigator' durch. Es werden die unterschiedlichen Möglichkeiten untersucht, den Navigator zu öffnen und zu schließen. Volgende Möglichkeiten sind vorhanden:

- Taste F5 drücken.
- Auf der Funktionsleiste das Symbol für den Navigator klicken.
- Im Menü 'Bearbeiten' den Menüpunkt 'Navigator' auslösen.

Zusätzlich kann der Navigator durch die Taste ESCAPE geschlossen werden. Zur Verdeutlichung des Prinzips ist es ausreichend, daß der Testcase lediglich die verschiedene Kombinationen, die mit der Tastatur ausgelöst werden können, untersucht.

5.5 Probelauf des Beispieles aus dem Anhang

Bei der Durchführung des Testskriptes aus dem Anhang entsteht das in Abbildung 15 gezeigte Journal. Die Unterpunkte sind so dargestellt wie sie durch das Testsystem während des Testlaufs geöffnet wurden. Es wurden nur die Testfälle geöffnet, in denen Fehler aufgetreten sind. Die Regeln für die Farbgebung und das Anzeigen von Unterpunkten sind in Kapitel 4.6 gegeben.

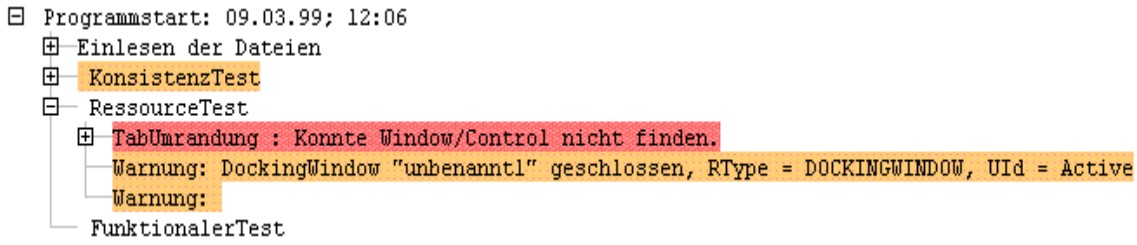

Abbildung 15: Journal nach Testlauf des Beispieles

Um das Gesamte Ergebnis betrachten zu können, sind in Abbildung 16 säntliche Zweige der Darstellung aufgeklappt.

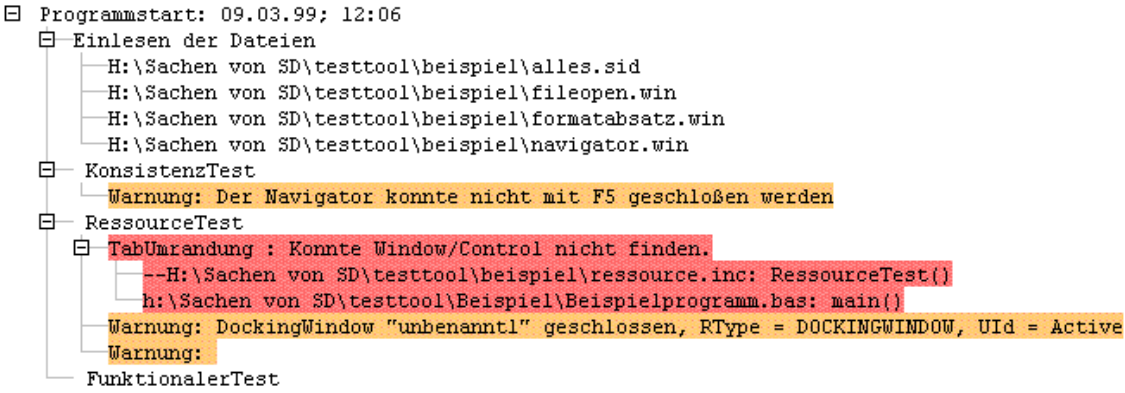

Abbildung 16: Journal nach Testlauf des Beispieles mit allen Unterpunkten sichtbar

6 Ausblick

6.1 Lernmodus, zum automatischen Generieren eines Skriptes

Um die Erstellung der Skripten zu vereinfachen wäre es sinnvoll, die Abläufe durch eine Art Lernmodus erfassen zu können. In diesem Lernmodus würden die Eingaben eines Benutzers registriert und als fertiges Skript aufgezeichnet. Die so entstandenen Skripte können entweder direkt verwendet werden, oder als Basis für komplexere Skripte dienen, die beispielsweise durch Ergänzung mit zusätzlichen Abfragen und Schleifen entstehen.

Beim Aufzeichnen der Skripte ergäbe sich ein Problem daraus, daß die Ereignisse, die vom Benutzer ausgelöst werden, jeweils auf unterschiedlicher Abstraktionsebene aufgezeichnet werden könnten. Das System müßte also entweder intelligent sein und jeweils eine passende Interpretation zur Verfügung stellen, oder der Benutzer müßte vor Beginn der Aufzeichnung einstellen, welcher Abstraktionsgrad gewünscht ist. Noch komplexer wird es, wenn auch die Aufrufe von Slots generiert werden sollen. Es müßte dann beispielsweise die gesamte Bearbeitung eines Dialogs 'Datei Öffnen' in einem einzigen Slotaufruf zusammengefaßt werden. Es könnte aber auch passieren, daß ein Dialog 'Datei Öffnen' aufgerufen wird und innerhalb dieses Dialoges beispielsweise ein neues Verzeichnis angelegt wird und er dann abgebrochen wird, was dann nicht zum Aufzeichnen des entsprechenden Slotaufrufes führen darf. In diesem Fall müßten die gesamten Ereignisse zwischengespeichert werden, um dann doch einzeln aufgezeichnet zu werden.

Eine Erweiterung in diese Richtung würde, besonders in Hinblick auf die Generierung von Ereignisse auf einer hohen Abstraktionsebene, erfordern, daß jedes Bedienelement befähigt wird, diese Ereignisse durch einen einheitlichen Mechanismus zur Verfügung zu stellen. Da nur das Bedienelement selbst 'weiß', welches Ereignis höherer Abstraktion durch die Elementarereignisse ausgelöst wird, kann dies nicht an zentraler Stelle geschehen.

Da das Instrumentieren sämtlicher Bedienelemente mit dieser Fähigkeit einigen Entwicklungsaufwand bedeuten würde und außerdem zu einer nicht unerheblichen Menge Codes führen würde, der beim Anwender nie verwendet würde, wurde dieser Lernmodus bisher nicht realisiert.

7 Literaturverzeichnis

Behdjati, A., S. Bagdon, P. Fleischer, P. Schlüter (1991). Objektorientierte Sprachen im Vergleich. Softwaretechnik-Trends, Bd.11 Heft 4, S. 11 - 31

Hoare, C. A. R. (1969). An Axiomatic Basis For Computer Programming. CACM, Vol. 12, Heft 10, S. 576 - 583

Jorgensen, Paul C., Carl Erickson (1994). Object-Oriented Integration Testing. Communications of the ACM, Vol. 37 (1994), 9, S. 30 - 38

Kilberth, Gryczan, Züllighoven, Bäumer, Budde, Hasbron-Blume, Sylla, Weimer (1994). Objektorientierte Anwendungsentwicklung. Konzepte, Strategien, Erfahrungen. 2. Auflage, Braunschweig/Wiesbaden: Vieweg.

Lyle, James R., Marvin V. Zelkowitz (1979). Assertion Mechanisms in Programming Languages. Computer Science Technical Reports Series; University of Maryland, Maryland 1979

Myers, Glenford J. (1979). The Art Of Software Testing. New York, Chichester, Brisbane, Toronto: John Wiley & Sons

Ostrand, Thomas, Aaron Anodide, Herbert Foster, Tarak Goradie (1998). A Visual Test Development Environment for GUI Systems. Software Engineering Notes, Vol. 23 Nr. 2, Pp. 82-92. ACM SigSoft

Pollack, Randy(1999). The LEGO Proof Assistant. Edinburgh / Internet: http://www.dcs.ed.ac.uk/home/lego/ am 7.3.1999

QA Partner User's Guide, Segue Software, Inc., Newton Centre, MA 1994

Schlüter, P., A. Behdjati, P. Fleischer, S. Bagdon (1990). Objektorientierte Software-Entwicklung: Konzepte und Terminologie. Softwaretechnik-Trends, Bd.10, Heft 2, S. 22 - 44

Strauß, Friedrich, Thiemo Hörnke (1998). Testfallentwicklung und -automatisierung in großen objektorientierten Projekten - Ein Erfahrungsbericht. Softwaretechnik-Trends. Bd. 18 Heft 3, S. 81 - 89

Stroustrup, B. (1992). Die C++ Programmiersprache. 2. Auflage, Bonn: Addison-Wesley

Züllighoven, Heinz (1998). Das objektorientierte Konstruktionshandbuch. Heidelberg: dpunkt-Verlag

8 Anhang

8.1 Beispiel eines Testskriptes

Es werden sämtliche für den Ablauf des Skriptes benötigten Dateien angegeben. Die Hauptfunktionen des Skriptes ist in der Datei »beispielprogramm.bas« enthalten. Weitere Unterprogramme und die Testfälle finden sich in den Dateien »ressource.inc«, »funktion.inc« und »tools.inc«. Die Beschreibung der Bedienelemente und der Slots sind in den Dateien »fileopen.win«, »formatabsatz.win«, sowie »alles.sid« enthalten. Aus der Datei »hid.lst« wird nur der für dieses Skript relevante Teil gezeigt. Die Selektion findet statt, da der gesamte Umfang der Datei etwa 6800 Langnamen umfasst, was etwa 130 Seiten Text entspricht.

8.1.1 Datei »beispielprogramm.bas«

```
Rem Beispielprogramm
sub main

    ' Aufrufen der Testcases
    KonsistenzTest
    RessourceTest
    FunktionalerTest

end sub

sub LoadIncludeFiles

    ' Laden der allgemeinen Hilfsfunktionen
    use "beispiel\tools.inc"

    ' Laden der Beschreibung der Benutzungsoberfläche
    ' und der Slots
    use "beispiel\alles.sid"
    use "beispiel\formatabsatz.win"
    use "beispiel\navigator.win"

    ' Laden der Testcases
    use "beispiel\funktion.inc"
    use "beispiel\ressource.inc"
    use "beispiel\konsistenz.inc"

    ' Herstellen der Verbindung zur Anwendung.
    ' Da ein leerer Parameter mitgegeben wird,
    ' muß die Anwendung bereits gestartet sein.
    start ""

    ' Rücksetzen der Anwendung in den
    ' definierten Grundzustand.
    dim Memo
```

```
    Memo = resetapplication
    if Memo > "" then
        warnlog Memo
    endif
end sub
```

Datei »beispielprogramm.bas« enthält das Hauptprogramm und das Unterprogramm zum Initialisieren des Skriptes

8.1.2 Datei »ressource.inc«

```
testcase RessourceTest
' In diesem Testcase werden sämtliche Tabpages des
' Dialoges Format/Absatz aufgerufen

   NeuesDokument
   FormatAbsatz        ' Dialog aufrufen

   ' Die einzelnen Tabpages aufrufen
   Active.SetPage TabAusrichtungAbsatz
   Active.SetPage TabInitialen
   Active.SetPage TabEinzuegeUndAbstaende
   Active.SetPage TabTabulator
   Active.SetPage TabNumerierungAbsatz
   Active.SetPage TabTextfluss
   Active.SetPage TabHintergrund
   Active.SetPage TabUmrandung

   ' Dialog wieder schließen
   TabUmrandung.Close

   'Dieser Befehl löst einen Fehler aus
   TabUmrandung.Close

   ' Dokument schließen
   DokumentSchliessen

endcase
```

Datei »ressource.inc« enthält den Testcase zum Testen der Ressourcen

8.1.3 Datei »funktion.inc«

```
testcase FunktionalerTest

    ' Verwendete Variable deklarieren
    dim Memo as String
    NeuesDokument
    FormatAbsatz
    Active.SetPage TabEinzuegeUndAbstaende
    Active.SetPage TabInitialen

    ' Kontext der Tabpage einstellen, so daß die darauf
    ' befindlichen Bedienelemente verfügbar werden.
    Kontext "TabInitialen"

    ' CheckBox "Anzeigen" auswählen
    Anzeigen.Check

    ' In der Schleife wird der Wert des
    ' Bedienelementes "Anzahl" solange erhöht,
    ' bis er sich nichtmehr verändert.
    Memo = 0
    while Anzahl.GetText <> Memo
        Memo = Anzahl.GetText
        Anzahl.More
    wend

    ' Der letzte Wert wird mit dem gewünschten Makimalwert
    ' verglichen. Unterscheidet er sich, so wird eine
    ' Warnung in das Journal ausgegeben.
    if Memo <> "10" then
        warnlog "Anzahl Zeichen geht nicht bis 10, sondern
            bis ", Memo
    endif

    TabInitialen.Close
```

```
    DokumentSchliessen

endcase
```

Datei »funktion.inc« enthält den Testcase zum funktionalen Testen

8.1.4 Datei »konsistenz.inc«

```
testcastestcase KonsistenzTest
' In diesem Testcase werden die verschiedenen
' Aufrufmethoden des Explorers untersucht

    dim Prolog
    Prolog = "Der Navigator konnte nicht mit "

    NeuesDokument

    ' Öffnen mit F5
    DokumentWriter.TypeKeys "<f5>"
    if not NavigatorWriter.exists then
        warnlog prolog, "F5 geöffnet werden"
        BearbeitenNavigator  'Navigator über Slot aufrufen
    endif

    'Schließen mit F5
    DokumentWriter.TypeKeys "<f5>"
    if not NavigatorWriter.notexists then
        warnlog prolog, "F5 geschloßen werden"
    else
        BearbeitenNavigator     'Navigator wieder öffnen
    endif

    'Schließen im Navigator mit ESCAPE
    NavigatorWriter.TypeKeys "<escape>"
    if not NavigatorWriter.notexists then
        warnlog prolog, "F5 geschloßen werden"
```

```
    else
        BearbeitenNavigator     'Navigator wieder öffnen
    endif

    'Schließen im Navigator mit F5
    NavigatorWriter.TypeKeys "<f5>"
    if not NavigatorWriter.notexists then
        warnlog prolog, "F5 geschloßen werden"
    else
        BearbeitenNavigator     'Navigator wieder öffnen
    endif

    ' Navigator schließen
    if NavigatorWriter.exists then
        BearbeitenNavigator
    endif
    ' Dokument schließen
    DokumentSchliessen

endcase
```

Datei »konsistenz.inc« enthält den Testcase zum Test des Navigators

8.1.5 Datei »tools.inc«

```
' Hilfsroutinen

sub NeuesDokument

   NeuDirekt NeuDirekt_ID, "swriter"

end sub

sub DokumentSchliessen

   try
      DateiSchliessen
   catch
      Exceptlog
      Exit sub
   endcatch

   Kontext
   ' Überprüfen, ob eventuell eine Dialogbox zum Spei-
chern
   ' einer geänderten Datei erscheint
   if Active.Exists(2) then
      Active.No
   end if
end sub

sub testenter

   kontext

end sub

sub testexit
```

```
    dim Memo
    Memo = resetapplication
    if Memo > "" then
        warnlog Memo
    endif

end sub
```

Datei »tools.inc« enthält die Unterprogramme TestEnter und testExit sowie allgemeine Hilfsfunktionen

8.1.6 Datei »formatabsatz.win«

```
*TabAusrichtungAbsatz HID_FORMAT_PARAGRAPH_ALIGN
Links SVX:RADIOBUTTON:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_LEFTALIGN
Rechts SVX:RADIOBUTTON:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_RIGHTALIGN
Zentriert SVX:RADIOBUTTON:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_CENTERALIGN
Blocksatz SVX:RADIOBUTTON:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_JUSTIFYA-
  LIGN
LetzteZeile SVX:LISTBOX:RID_SVXPAGE_ALIGN_PARAGRAPH:LB_LASTLINE
EinzelnesWortAustreiben
  SVX:CHECKBOX:RID_SVXPAGE_ALIGN_PARAGRAPH:CB_EXPAND

*TabInitialen HID_DROPCAPS
Anzeigen sw:CheckBox:TP_DROPCAPS:CB_SWITCH
GanzesWort sw:CheckBox:TP_DROPCAPS:CB_WORD
Anzahl sw:NumericField:TP_DROPCAPS:FLD_DROPCAPS
AbstandZumText sw:MetricField:TP_DROPCAPS:FLD_DISTANCE
Zeilen sw:NumericField:TP_DROPCAPS:FLD_LINES
Initialentext sw:Edit:TP_DROPCAPS:EDT_TEXT
Zeichenvorlage sw:ListBox:TP_DROPCAPS:BOX_TEMPLATE

*TabEinzuegeUndAbstaende HID_FORMAT_PARAGRAPH_STD
Automatisch svx:CheckBox:RID_SVXPAGE_STD_PARAGRAPH:CB_AUTO
Vonlinks svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_LEFTINDENT
ErsteZeile svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_FLINEINDENT
VonRechts svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_RIGHTINDENT
Oben svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_TOPDIST
Unten svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_BOTTOMDIST
Zeilenabstand svx:ListBox:RID_SVXPAGE_STD_PARAGRAPH:LB_LINEDIST
Um svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_LINEDISTPERCENT
Von SVX:METRICFIELD:RID_SVXPAGE_STD_PARAGRAPH:ED_LINEDISTMETRIC
Registerhaltigkeit svx:CheckBox:RID_SVXPAGE_STD_PARAGRAPH:CB_REGISTER
```

```
*TabTabulator HID_TABULATOR
AlleLoeschen svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_DELALL
Fuellzeichen1 svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_NO
Fuellzeichen2
  svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_POINTS
Fuellzeichen3
  svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_DASHLINE
Fuellzeichen4
  svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_UNDERSCORE
Fuellzeichen5
  svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_OTHER
FuellzeichenZeichen svx:Edit:RID_SVXPAGE_TABULATOR:ED_FILLCHAR_OTHER
Loeschen svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_DEL
Neu svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_NEW
Position svx:MetricBox:RID_SVXPAGE_TABULATOR:ED_TABPOS
TypLinks svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_LEFT
TypRechts svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_RIGHT
TypZentriert svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_CENTER
TypDezimal svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_DECIMAL
TypZeichen svx:Edit:RID_SVXPAGE_TABULATOR:ED_TABTYPE_DECCHAR

*TabNumerierungAbsatz Hid_NUMPARA
Vorlage SW:LISTBOX:TP_NUMPARA:LB_NUMBER_STYLE
NumerierungNeuBeginnen SW:TRISTATEBOX:TP_NUMPARA:CB_NEW_START
  SW:CHECKBOX:TP_NUMPARA:CB_NEW_START
NumerierungBeginnenBei SW:NUMERICFIELD:TP_NUMPARA:NF_NEW_START
ZeilenumerierungZeilenMitzaehlen
  SW:TRISTATEBOX:TP_NUMPARA:CB_COUNT_PARA
ZeilenumerierungNeuBeginnen
  SW:TRISTATEBOX:TP_NUMPARA:CB_RESTART_PARACOUNT
ZeilenumerierungBeginnenBei
  SW:NUMERICFIELD:TP_NUMPARA:NF_RESTART_PARA

*TabTextfluss HID_FORMAT_PARAGRAPH_EXT
Automatisch svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_HYPHEN
AbZeichenende svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_HYPHEN-
  BEFORE
AbZeichenAnfang
  svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_HYPHENAFTER
Trennstellen svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_MAXHYPH
Umbruch svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAK
Seite svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_BREAKPAGE
Spalte svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_BREAKCOLUMN
```

```
Davor svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAKBEFORE
Danach svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAKAFTER
MitSeitenvorlage svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGE-
  COLL
Vorlage svx:ListBox:RID_SVXPAGE_EXT_PARAGRAPH:LB_PAGECOLL
AbsatzNichtTrennen
  svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_KEEPTOGETHER
AbsaetzeZusammenhalten
  svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:CB_KEEPTOGETHER
Schusterjungenregelung
  svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_WIDOWS
SchusterZeilen svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_WIDOWS
Hurenkinderregelung
  svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_ORPHANS
HurenkinderZeilen svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_ORP-
  HANS
Seitenzahl SVX:NUMERICFIELD:RID_SVXPAGE_EXT_PARAGRAPH:ED_PAGENUM

*TabHintergrund HID_BACKGROUND
Als svx:ListBox:RID_SVXPAGE_BACKGROUND:LB_SELECTOR
Fuer svx:ListBox:RID_SVXPAGE_BACKGROUND:LB_TBL_BOX
FuerAbsatz SVX:LISTBOX:RID_SVXPAGE_BACKGROUND:LB_PARA_BOX
Hintergrundfarbe HID_BACKGROUND_CTL_BGDCOLORSET
Verknuepfen svx:CheckBox:RID_SVXPAGE_BACKGROUND:BTN_LINK
Vorschau svx:CheckBox:RID_SVXPAGE_BACKGROUND:BTN_PREVIEW
Durchsuchen svx:PushButton:RID_SVXPAGE_BACKGROUND:BTN_BROWSE
Position svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_POSITION
Flaeche svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_AREA
Kachel svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_TILE
ArtPosition HID_BACKGROUND_CTL_POSITION

*TabUmrandung HID_BORDER
Vorgaben HID_BORDER_CTL_PRESETS
'Rahmen
AbstandZumInhalt svx:MetricField:RID_SVXPAGE_BORDER:ED_XSPACE
Stil svx:ListBox:RID_SVXPAGE_BORDER:LB_LINESTYLE
StilFarbe svx:ListBox:RID_SVXPAGE_BORDER:LB_LINECOLOR
Position HID_BORDER_CTL_SHADOWS
Groesse svx:MetricField:RID_SVXPAGE_BORDER:ED_SHADOWSIZE
SchattenFarbe svx:ListBox:RID_SVXPAGE_BORDER:LB_SHADOWCOLOR
Zurueck HID_TABDLG_RESET_BTN
Standard HID_TABDLG_STANDARD_BTN
```

```
*Active active
```

Datei »formatabsatz.win« enthält die Beschreibung des Dialoges zur Formatierung eines Absatzes. Die eingerückten Zeilen mußten umgebrochen werden und sind als eine Zeile zu verstehen.

8.1.7 Datei »navigator.win«

```
*NavigatorWriter HID_NAVIGATION_PI
Auswahlliste HID_NAVIGATOR_TREELIST
Dokumentliste HID_NAVIGATOR_LISTBOX
Toolbox HID_NAVIGATOR_TOOLBOX
Umschalten HID_NAVI_TBX17
Navigation HID_NAVI_TBX24
Seitennummer HID_NAVI_TBX16
VorherigeSeite HID_NAVI_TBX2
NaechsteSeite HID_NAVI_TBX3
Dragmodus HID_NAVI_TBX4
KapitelHoch HID_NAVI_TBX5
KapitelRunter HID_NAVI_TBX6
Auswahlbox HID_NAVI_TBX7
Inhaltsansicht HID_NAVI_TBX8
Merker HID_NAVI_TBX9
Kopfzeile HID_NAVI_TBX10
Fusszeile HID_NAVI_TBX11
AnkerText HID_NAVI_TBX12
Ebenen HID_NAVI_TBX13
Hoch HID_NAVI_TBX14
Senken HID_NAVI_TBX15

*DokumentWriter HID_EDIT_WIN
```

Datei »navigator.win« enthält die Beschreibung des Dialoges 'navigator'.

8.1.8 Datei »alles.sid«

```
DateiOeffnen SID_OPENDOC
FormatAbsatz SID_PARA_DLG
NeuDirekt SID_NEWDOCDIRECT
DateiSchliessen SID_CLOSEDOC
BearbeitenNavigator SID_NAVIGATOR
```

Datei »alles.sid« enthält die Deklaration der Slots

8.1.9 Ausschnitt der Datei »hid.lst«

```
HID_FORMAT_PARAGRAPH_ALIGN 33795
svx:RadioButton:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_LEFTALIGN 703775262
svx:RadioButton:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_RIGHTALIGN 703775263
svx:RadioButton:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_CENTERALIGN 703775264
svx:RadioButton:RID_SVXPAGE_ALIGN_PARAGRAPH:BTN_JUSTIFYALIGN
  703775265
svx:ListBox:RID_SVXPAGE_ALIGN_PARAGRAPH:LB_LASTLINE 703778354
svx:CheckBox:RID_SVXPAGE_ALIGN_PARAGRAPH:CB_EXPAND 703775795
HID_DROPCAPS 53168
sw:CheckBox:TP_DROPCAPS:CB_SWITCH 878150669
sw:CheckBox:TP_DROPCAPS:CB_WORD 878150672
sw:NumericField:TP_DROPCAPS:FLD_DROPCAPS 878155778
sw:MetricField:TP_DROPCAPS:FLD_DISTANCE 878156294
sw:NumericField:TP_DROPCAPS:FLD_LINES 878155780
sw:Edit:TP_DROPCAPS:EDT_TEXT 878151689
sw:ListBox:TP_DROPCAPS:BOX_TEMPLATE 878153227
HID_FORMAT_PARAGRAPH_STD 33793
svx:CheckBox:RID_SVXPAGE_STD_PARAGRAPH:CB_AUTO 700810288
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_LEFTINDENT 700815883
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_FLINEINDENT 700815885
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_RIGHTINDENT 700815887
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_TOPDIST 700815893
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_BOTTOMDIST 700815895
svx:ListBox:RID_SVXPAGE_STD_PARAGRAPH:LB_LINEDIST 700812840
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_LINEDISTPERCENT
  700815914
svx:MetricField:RID_SVXPAGE_STD_PARAGRAPH:ED_LINEDISTMETRIC 700815915
svx:CheckBox:RID_SVXPAGE_STD_PARAGRAPH:CB_REGISTER 700810324
HID_TABULATOR 33782
svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_DELALL 700748340
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_NO 700744232
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_POINTS 700744233
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_DASHLINE 700744234
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_UNDERSCORE
  700744235
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_FILLCHAR_OTHER 700744236
svx:Edit:RID_SVXPAGE_TABULATOR:ED_FILLCHAR_OTHER 700745773
svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_DEL 700748339
svx:PushButton:RID_SVXPAGE_TABULATOR:BTN_NEW 700748338
svx:MetricBox:RID_SVXPAGE_TABULATOR:ED_TABPOS 700753418
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_LEFT 700744212
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_RIGHT 700744214
```

```
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_CENTER 700744216
svx:RadioButton:RID_SVXPAGE_TABULATOR:BTN_TABTYPE_DECIMAL 700744218
svx:Edit:RID_SVXPAGE_TABULATOR:ED_TABTYPE_DECCHAR 700745756
HID_NUMPARA 53214
sw:ListBox:TP_NUMPARA:LB_NUMBER_STYLE 879185409
sw:TriStateBox:TP_NUMPARA:CB_NEW_START 879183363
sw:NumericField:TP_NUMPARA:NF_NEW_START 879187974
sw:TriStateBox:TP_NUMPARA:CB_COUNT_PARA 879183368
sw:TriStateBox:TP_NUMPARA:CB_RESTART_PARACOUNT 879183369
sw:NumericField:TP_NUMPARA:NF_RESTART_PARA 879187979
HID_FORMAT_PARAGRAPH_EXT 33794
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_HYPHEN 700827186
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_HYPHENBEFORE 700831796
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_HYPHENAFTER 700831799
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_MAXHYPH 700831827
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAK 700827196
svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_BREAKPAGE 700826173
svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_BREAKCOLUMN 700826174
svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAKBEFORE
  700826175
svx:RadioButton:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGEBREAKAFTER
  700826176
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_PAGECOLL 700827201
svx:ListBox:RID_SVXPAGE_EXT_PARAGRAPH:LB_PAGECOLL 700829250
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_KEEPTOGETHER 700827203
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:CB_KEEPTOGETHER 700827216
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_WIDOWS 700827204
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_WIDOWS 700831813
svx:TriStateBox:RID_SVXPAGE_EXT_PARAGRAPH:BTN_ORPHANS 700827207
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_ORPHANS 700831816
svx:NumericField:RID_SVXPAGE_EXT_PARAGRAPH:ED_PAGENUM 700831820
HID_BACKGROUND 33784
svx:ListBox:RID_SVXPAGE_BACKGROUND:LB_SELECTOR 700730881
svx:ListBox:RID_SVXPAGE_BACKGROUND:LB_TBL_BOX 700730912
svx:ListBox:RID_SVXPAGE_BACKGROUND:LB_PARA_BOX 700730914
HID_BACKGROUND_CTL_BGDCOLORSET 33868
svx:CheckBox:RID_SVXPAGE_BACKGROUND:BTN_LINK 700728342
svx:CheckBox:RID_SVXPAGE_BACKGROUND:BTN_PREVIEW 700728343
svx:PushButton:RID_SVXPAGE_BACKGROUND:BTN_BROWSE 700731924
svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_POSITION 700727835
svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_AREA 700727833
svx:RadioButton:RID_SVXPAGE_BACKGROUND:BTN_TILE 700727834
HID_BACKGROUND_CTL_POSITION 33871
```

```
HID_BORDER 33783
HID_BORDER_CTL_PRESETS 33874
svx:MetricField:RID_SVXPAGE_BORDER:ED_XSPACE 700799520
svx:ListBox:RID_SVXPAGE_BORDER:LB_LINESTYLE 700796457
svx:ListBox:RID_SVXPAGE_BORDER:LB_LINECOLOR 700796459
HID_BORDER_CTL_SHADOWS 33875
svx:MetricField:RID_SVXPAGE_BORDER:ED_SHADOWSIZE 700799546
svx:ListBox:RID_SVXPAGE_BORDER:LB_SHADOWCOLOR 700796476
HID_TABDLG_RESET_BTN 33157
HID_TABDLG_STANDARD_BTN 33158
HID_NAVIGATION_PI 54424
HID_NAVIGATOR_TREELIST 52974
HID_NAVIGATOR_LISTBOX 52978
HID_NAVIGATOR_TOOLBOX 52977
HID_NAVI_TBX17 53011
HID_NAVI_TBX24 53018
HID_NAVI_TBX16 53005
HID_NAVI_TBX2 52988
HID_NAVI_TBX3 52989
HID_NAVI_TBX4 52990
HID_NAVI_TBX5 52991
HID_NAVI_TBX6 52992
HID_NAVI_TBX7 52993
HID_NAVI_TBX8 52994
HID_NAVI_TBX9 52995
HID_NAVI_TBX10 52996
HID_NAVI_TBX11 52997
HID_NAVI_TBX12 52998
HID_NAVI_TBX13 52999
HID_NAVI_TBX14 53000
HID_NAVI_TBX15 53001
HID_EDIT_WIN 52821
SID_OPENDOC 5501
SID_PARA_DLG 10297
SID_NEWDOCDIRECT 5537
SID_CLOSEDOC 5503
SID_NAVIGATOR 10366
```

Ausschnitt der Datei »hid.lst« enthält die Zuordnung von Langnamen zu den numerischen IDs

9 Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, daß ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

Hamburg den 11. März 1999

Gregor Hartmann