HUMBOLDT-UNIVERSITÄT ZU BERLIN

ATOSj: Ein Werkzeug für den automatisierten Regressionstest oberflächenbasierter Java-Programme

Diplomarbeit

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

Eingereicht von:

Volker Janetschek
180086

und

Nicos Tegos
176934

Betreuer: Prof. Dr. Klaus Bothe

Berlin, den 9. Januar 2007

Inhaltsverzeichnis

1 Einführung

Der Entwicklungs- und Wartungsprozess von Software ist durch häufige Änderungen gekennzeichnet. Bestehende Funktionen müssen an geänderte Anforderung angepasst und um neue ergänzt werden. Im Verlauf dieses Prozesses können Fehler entstehen, die z.B. durch Seiteneffekte bei der Integration neuer Komponenten verursacht werden. Fehler können sowohl in bestehenden als auch in neuen Programmteilen auftreten, deshalb ist nach jeder Änderung die erneute Verifikation der Korrektheit des Gesamtsystems notwendig. Dieser Vorgang wird als Regressionstest bezeichnet. Mit fortschreitender Entwicklungsdauer steigen Umfang und Komplexität des Systems und damit auch Umfang und Komplexität des Regressionstests. Zusätzlich macht die häufige Wiederholung den Regressionstest sehr zeitaufwändig. Um den Aufwand zu reduzieren, empfiehlt sich deshalb eine automatisierte Teststrategie. Im Vergleich zum manuellen Test spart ein automatisierter Test nicht nur Zeit, sondern sichert gleichzeitig auch die genaue Reproduzierbarkeit der Testfälle. Grundsätzlich lassen sich zwei Strategien für den automatisierten Test unterscheiden. Die erste Möglichkeit ist die Integration eines Testrahmens in den Quelltext des Testobjekts. Dieses Verfahren setzt die Kenntnis der internen Struktur des Testobjekts voraus und erschwert damit den Einsatz eines unabhängigen Testers. Außerdem ist ein Testrahmen Testobjekt-spezifisch und kann nicht auf andere Programme angewendet werden. Als zweite Möglichkeit bietet sich die Verwendung eines autarken Testtreibers an. Dieser ermöglicht die Steuerung des Testobjektes von außen und ist auf Grund seiner Unabhängigkeit vom Testobjekt weitaus flexibler als ein Testrahmen [15]. Für die Steuerung des Testobjekts wird eine Schnittstelle benötigt, die die Eingabe von Testdaten und das Auslesen der Ergebnisse der Datenverarbeitung ermöglicht. Grafische Oberflächen sind zu diesem Zweck besonders geeignet, denn sie werden in vielen Programmen eingesetzt und bestehen aus einzelnen formalisierten Bausteinen, deren Funktionsweise unabhängig vom Testobjekt ist.

Der oberflächenbasierte Regressionstest ist ein ganzheitlicher Systemtest aus Anwendersicht. Die Eingabe der Testdaten erfolgt durch die Simulation von Nutzeraktionen. Sie sorgt für den Übergang des Testobjekts in einen neuen Zustand. Dieser wird anschließend durch die Prüfung einzelner Oberflächenelemente validiert. Der Test irrelevanter Systemzustände ist ausgeschlossen, da nur Testfälle beschrieben werden können, die eine Entsprechung in einer realen Anwendungssituation haben. Ziel dieser Diplomarbeit war es, ein autarkes Testsystem für den oberflächenbasierten Regressionstest von Java-Programmen zu entwickeln. Ausgangspunkt war das System ATOS, das bereits den Test Windows-basierter Programme realisiert. Es wurde im Rahmen der Diplomarbeit von Jens Hanisch und Johann Letzel entwickelt und in einer weiteren Diplomarbeit durch Andreas Hirth erweitert [7][8].

2 Motivation für eine Portierung nach Java

Die Bedeutung von Java hat bei der Entwicklung von Software-Systemen in den letzten Jahren immer mehr zugenommen. Das wird in einer Pressemitteilung von Sun Microsystems aus dem Jahre 2001 deutlich [23]. Sun erklärt darin Java zur beliebtesten Programmiersprache weltweit und belegt dies mit eindrucksvollen Zahlen. Laut Sun arbeiten rund 54% aller Softwareentwickler mit Java. Damit ist Java sogar erfolgreicher als die Programmiersprachen C und C++, welche nur von 50% der Softwareentwickler genutzt werden. Die Aussagen von Sun werden untermauert durch Daten des Internetportals GULP, welches sich der Vermittlung von externen Mitarbeitern an IT-Projekte widmet. Der Anteil an Java-Projekten aus der Gesamtheit der eingetragenen Projekte hat im Jahr 2006, nach einem Einbruch in den vorangegangen Jahren, fast wieder den Höchststand von 20% vom Anfang des Jahres 2001 erreicht, wie Abbildung 1 zeigt [24]. Java ist aber nicht nur in der Wirtschaft sondern auch im universitären Bereich erfolgreich. Sun erklärt, dass in rund 80% aller Universitäten weltweit die Programmiersprache Java gelehrt wird.

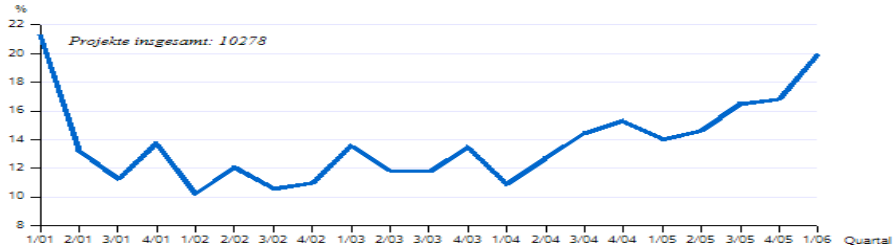


Abbildung 1: Anteil von Java-Projekten

Grund für den Erfolg von Java in den letzten Jahren ist nicht zuletzt die Entwicklung besserer und schnellerer Hardware, die den hohen Anforderungen von Java an die Prozessorleistung gerecht wird. Java ist eine interpretierte Sprache, d.h. ein Java-Programm liegt in einer Zwischensprache vor und muss zur Laufzeit in Instruktionen des realen Prozessors umgewandelt werden. Diese Umwandlung kostet Zeit und machte Java in früheren Zeiten langsam. Vor allem bei der Entwicklung grafischer Oberflächen (GUI) machten sich die Geschwindigkeitseinbußen bemerkbar und wirkten sich auf die Responsivität der GUI Komponente aus. Für die Implementation grafischer Anwendungen stehen dem Java-Programmierer zwei verschiedene Standardbibliotheken zur Verfügung, zum einen das AWT (Abstract Window Toolkit) und zum anderen Swing. Beide verfolgen unterschiedliche Ansätze. Das AWT nutzt direkt die Components des Betriebssystems und definiert unabhängige Schnittstellen, die deren Manipulation in einer Java-Umgebung erlauben. Swing dagegen emuliert die verschiedenen Steuerelemente, d.h. es bildet ihre Funktionsweise nach. Die Components in Swing haben keine native Entsprechung sondern werden nur mit Hilfe grafischer Basisroutinen gezeichnet und haben deshalb ein plattformunabhängiges Erscheinungsbild [1]. Diese Extreme in beiden Philosophien

führen zu Problemen. Die Betriebssystemabhängigkeit des AWT begrenzt die Zahl der unterstützten Componenttypen. Componenttypen, die nicht in allen Betriebssystemen gleichermaßen verfügbar sind, fehlen in der Bibliothek. Dazu gehören beispielsweise Tabellen, Bäume und Karteikarten. Auf Grund der mangelnden Verfügbarkeit der genannten Components wird das AWT nur noch selten für den Entwurf grafischer Oberflächen verwendet. Swing begegnet diesem Problem durch sein betriebssystemunabhängiges Design, welches die Darstellung beliebiger Components, so auch die von Tabellen, Bäumen und Karteikarten, ermöglicht. Diese Stärke ist gleichzeitig auch die größte Schwäche von Swing, denn die Emulation der Components kostet viel Zeit und macht Swing-basierte Anwendungen teilweise spürbar träge. Das plattformunabhängige Erscheinungsbild der Swing-Components birgt ebenfalls Usability-Probleme, da sich der Nutzer beim Wechsel von nativen zu Swing-basierten Anwendungen auf das veränderte Aussehen der Oberfläche einstellen muss.

Die dritte bedeutende grafische Bibliothek für Java ist das SWT (Standard Widget Toolkit). Es gehört nicht zum Java Standard sondern ist eine Eigenentwicklung von IBM im Rahmen des Open-Source-Projektes Eclipse. Das Konzept des SWT vereinigt die beiden gegensätzlichen Philosophien, die dem Design der Bibliotheken AWT und Swing zugrunde liegen. SWT dient als Mittel zur Entwicklung portabler grafischer Anwendungen und bedient sich dabei, wie auch das AWT, nativer Components. Diese Strategie verleiht den SWT-Components ein Aussehen entsprechend der Plattformspezifika und erhöht die Responsivität der GUI, so dass für den Nutzer kein Unterschied zu einer nativen Anwendung merkbar ist. Jedoch wird, im Gegensatz zum AWT, auf eine Strategie des kleinsten gemeinsamen Nenners verzichtet. Wichtige Standard-Components, die nicht vom Betriebssystem unterstützt werden, können, wie in Swing, durch die Bibliothek emuliert werden.

Resümierend kann gesagt werden, dass mit dem SWT ein grosser Schritt in Richtung Nutzerakzeptanz von Java-basierten Programmen mit grafischer Oberfläche gelungen ist. SWT überzeugt durch seine hohe Geschwindigkeit und ein dem Nutzer vertrautes Aussehen der GUI-Elemente. Aber auch Swing wird konkurrenzfähiger, da der Geschwindigkeitsnachteil durch ständig verbesserte Hardware kompensiert wird. Die gesteigerte Wettbewerbsfähigkeit der beiden Bibliotheken führt zu einer zunehmenden Verbreitung GUI-basierter Java-Programme, deren Entwicklung durch ein geeignetes Testwerkzeug unterstützt werden sollte. Für Windows-Programme existieren schon einige Systeme, die den oberflächengestützten Regressionstest automatisieren. Dazu gehören z. B. die kommerziellen Programmme WinRunner und GUI-do. Im Bereich Java, gibt es dagegen nur wenige Alternativen, die sich zumeist ausschließlich auf die Swing-Architektur konzentrieren, wie z.B. das Werkzeug Marathon.
Auch am Lehrstuhl für Softwaretechnik wurde bereits ein System zum Oberflächentest entwickelt, welches den Namen ATOS trägt. Es ist jedoch nur auf den Test von Windows-Programmen beschränkt. Ein entsprechendes Werkzeug für Java-Programme fehlt am Lehrstuhl, weshalb das in jüngster Zeit entstandene Programm Seminarorganisation bisher nicht getestet werden konnte. ATOS hat sich im Rahmen des XCTL-Projektes be-

währt und sollte deshalb in ähnlicher Weise auch für das Projekt Seminarorganisation eingesetzt werden. Es stellte sich uns die Frage nach einer Erweiterung von ATOS für Java. Folgende Aspekte sprachen jedoch gegen eine direkte Änderung des ursprünglichen Programmes:

1. Die beschränkte Einsatzmöglichkeit von ATOS

2. Die tiefe Verzahnung mit Windows-spezifischen Funktionen

zu 1.) Das Testsystem ATOS ist ein Windows-Programmm und damit nur auf diesem Betriebsystem ausführbar. Eine Erweiterung von ATOS für den Test von Java-Programmen führt zu einem Ungleichgewicht zwischen Testsystem und Testobjekten, da Java-Programme ihrer Natur nach plattformunabhängig sind. Dennoch würde ein Test unter Windows für die Verifikation eines Java-Programmes im Allgemeinen ausreichen, weil Java nach der Maxime entwickelt wurde „test once, run everywhere“. Es sprechen jedoch praktische Erwägungen gegen ein solches Konzept. Häufig wird bei der Entwicklung und auch beim Test ein anderes Betriebssystem als Windows verwendet. In diesem Fall ist für die Nutzung von ATOS ein Wechsel des Betriebssystems erforderlich, der aber in Bezug auf Lizenskosten und Zeitaufwand nicht zu rechtfertigen ist. Eine mögliche Lösung für dieses Problem wäre die Portierung von ATOS auf alle gängigen Betriebssysteme. Der Portierungsaufwand kann durch den Anteil von plattformabhängigem Code an der ATOS-Implementation abgeschätzt werden. Dieser beträgt mindestens 36%, was dem Anteil der Windows-spezifischen GUI-Komponente am Gesamtumfang entspricht, siehe Tabelle 1.

zu 2.) Das Pflichtenheft grenzt den Anwendungsbereich von ATOS auf den Regressionstest oberflächenbasierter Windows-Anwendungen ein. Dieses Kriterium spiegelt sich demzufolge auch in der Architektur des Testsystems und insbesondere bei der Verwendung der Skriptsprache HTS wider. HTS selbst ist zwar plattformunabhängig, wird aber intern auf die Windows-spezifische Sprache ATS abgebildet, deren Kommandos dann vom ATOS-Interpreter ausgeführt werden. Somit ist auch die Interpreterkomponente bei einer Erweiterung für Java nicht wiederverwendbar und müsste komplett neu entwickelt werden, ebenso wie eine Java-spezifische Skriptsprache, die ATS ersetzt.

Der Kern der Implementierung von ATOS gliedert sich in 5 Hauptkomponenten, wie Tabelle 1 zeigt. Wie in Punkt 1 erwähnt, müsste die GUI-Komponente durch eine unabhängige grafische Bibliothek ersetzt werden. Des Weiteren kann die Klasse `ATSParser` auf Grund ihrer engen Verzahnung mit der Windows API, nicht für die Steuerung und Abfrage von Java-Components genutzt werden. Auch die Klasse `ATOS` kann nicht ohne Änderungen übernommen werden. Sie ist für die Verwaltung eines ATOS Projektes zuständig und unterstützt bisher keine Java-spezifischen Projektparameter, die zusätzliche Informationen wie z.B. die Main-Klasse und den Klassenpfad des Testobjektes enthalten müssten. Lediglich die Klassen `HTSParser` und `CTEParser` könnten änderungsfrei genutzt werden. In Summe müssen also circa 81% der Implementierung von ATOS für

die Java-Erweiterung entweder vollständig neu implementiert oder teilweise abgeändert werden. Dieses Missverhältnis zwischen Kosten und Nutzen hat uns zu der Entscheidung bewogen, ATOS vollständig in Java zu reimplementieren. Der Name der Testsuite in der Java-Version lautet demzufolge auch ATOSj (Automatisierter Test oberflächenbasierter Systeme in Java).

Komponente	**LOC**	**Anteil in %**
GUI-Komponenten	11000	36
Klasse ATOS	7500	25
Klasse ATSParser	6000	20
Klasse HTSParser	3900	13
Klasse CTEParser	1700	6
Summe	30100	100

Tabelle 1: ATOS: Umfang der Implementierung [7, S. 191]

3 Aufgabenteilung

Die Entwicklung des Testsystems ATOSj war der Kern dieser Diplomarbeit. Das Testsystem gliedert sich in viele unterschiedliche Komponenten. Es verwendet die Skriptsprache HTS, die überarbeitet und erweitert wurde. Für die Integration der Skriptsprache wurde ein Parser sowie ein Interpreter entwickelt. Eine graphische Oberfläche ermöglicht die Verwaltung von Projekten und das Erstellen und Ausführen von Testsequenzen. Im Kern der Implementation wurden Wrapperklassen zu den verschiedenen Components der Bibliotheken Swing und SWT erstellt. Diese realisieren die Simulation und die Aufzeichnung von Nutzeraktionen. Zusätzlich wurde ein Modul zur dynamischen Methodenüberdeckungsanalyse in das Testsystem eingefügt. Für die Entwicklung der vorgenannten Komponenten sowie für die Dokumentation des Quelltextes und das Erstellen eines Handbuchs war Nicos Tegos verantwortlich.

Volker Janetschek war bei der Entwicklung von ATOSj für die Interprozesskommunikation zwischen Testsystem und Testobjekt verantwortlich. Außerdem erstellte er ein Modul, das den Zugriff von ATOSj auf SWT-Components ermöglicht. Zusätzlich entwickelte er das Programm Seminarorganisation an Hand einer Spezifikation von Helmut Balzert. Die graphische Oberfläche des entwickelten Programms basiert auf der Java-Bibliothek SWT. Unter Verwendung des Testsystems ATOSj entwickelte Volker Janetschek Testfälle für den automatisierten Regressionstest der Seminarorganisation. Dabei unterzog er ATOSj einem intensiven Nutzertest.

Bei der gemeinsamen Entwicklung von ATOSj war Zusammenarbeit von äußerster Wichtigkeit. Deshalb wurden regelmäßige Treffen durchgeführt und auch telefonisch Rücksprache gehalten. Für die Teamarbeit wurde das Versionssystem CVS genutzt, welches den Austausch und das Editieren des gemeinsamen Quelltextes ermöglichte.

4 Wahl der Skriptsprache

Testfälle werden in ATOSj mit Hilfe eines Testskripts beschrieben. Das Testskript enthält Kommandos, die zur Manipulation und zur Überprüfung des Testobjektes dienen. Die Definition der Kommandos kann nicht willkürlich erfolgen sondern muss einer definierten Struktur und Semantik unterliegen. Eine Skriptsprache soll Struktur und Semantik der Kommandos beschreiben. Sie ist der zentrale Kern des Systems und muss für den Tester verständlich, einfach anwendbar und der Aufgabe angemessen sein. Aus diesem Grund muss die Entscheidung, welche Skriptsprache verwendet werden soll, wohl überlegt sein. Es gibt bereits eine Menge vorhandener Sprachen, welche potentiell als Skriptsprachen für ATOSj in Frage kommen. Aus dieser Menge haben wir XML in die engere Wahl genommen, da es für seine universelle Einsetzbarkeit bekannt ist. Zusätzlich haben wir Python und HTS eingehender untersucht, weil sie bereits in den Systemen Marathon und ATOS zum selben Zweck genutzt werden. Diese drei Sprachen sollen kurz mit ihren Vor- und Nachteilen beschrieben werden.

4.1 XML

Die Sprache XML ist ein universelles Mittel zur strukturierten Speicherung von Daten. Sie erlaubt eine freie Semantikdefinition und eignet sich deshalb auch als Skriptsprache für ATOSj. Für die vollständige Integration einer XML-basierten Skriptsprache in das Testsystem sind zwei Komponenten nötig. Die erste Komponente ist ein Parser, der Skripte einliest und auf Wohlgeformtheit sowie Gültigkeit prüft. Die zweite Komponente ist ein Interpreter, der die Ausführung der eingelesenen Kommandos übernimmt. Die Eigenimplementation der ersten Komponente ist nicht nötig, da bereits ein entsprechender Parser in der Java API existiert. Aufwand entsteht lediglich bei der Entwicklung der Interpreterkomponente.
XML ist gleichzeitig menschen- sowie maschinenlesbar, da es ein textbasiertes Format ist. Ein XML-Dokument wird durch so genannte Markups strukturiert. Diese kennzeichnen bestimmte Textabschnitte und ordnen ihnen eine Bedeutung zu. Die Verwendung von Markups schränkt jedoch die Lesbarkeit eines XML-Dokuments für den Menschen sehr ein, da sich Strukturinformationen mit den eigentlichen Daten vermischen. Ein weiterer Nachteil von XML ist die typische Schachtelung der Elemente, die zu mehrzeiligen Kommandos führt, wie Listing 1 zeigt. Im Vergleich dazu bietet ein einzeiliges Format, wie HTS, einen schnelleren Überblick und damit eine bessere Lesbarkeit. Nicht nur die Lesbarkeit wird bei der Verwendung von XML eingeschränkt auch das Bearbeiten von Kommandos wird erschwert, da sich der Schreibaufwand auf Grund der zusätzlichen Strukturinformationen stark erhöht. Diese Schwächen haben auch schon die Entwickler des Projektes Marathon erkannt. Sie setzten in den ersten Programmversionen XML als Skriptsprache ein und hatten praktische Erfahrungen mit den genannten Problemen. Diese waren so akut, dass man sich entschloß die Skriptsprache zu wechseln [17].

Nicht zuletzt der Erfahrungsbericht der Marathon-Entwickler hat uns dazu bewogen eine Entscheidung gegen XML als Skriptsprache zu treffen. XML-Skripte müssen in ein nutzerfreundlicheres Format transformiert werden, um eine effiziente Arbeit zu ermöglichen. Dadurch entsteht ein zusätzlicher Aufwand, der durch die Vorteile, wie das automatische Parsing, nicht aufgewogen wird.

```
<action windowTitle="Fenster" componentType="TABLE"
  componentName="Mitarbeiter">
  <edit>Mustermann</edit>
  <subitems>
    <item>1</item>
    <item>1</item>
  </subitems>
</action>
```

Listing 1: Entwurf eines Kommandos in XML

Das gleiche Kommando in HTS:
`ACTION, ‘‘Fenster‘‘, TABLE, ‘‘Mitarbeiter‘‘, EDIT, ‘‘Mustermann‘‘, SUBITEM, 1, 1`

4.2 Python

Python ist eine beliebte Skriptsprache, die z.B. beim Rapid Prototyping oder bei der Testautomatisierung zum Einsatz kommt. Sie unterstützt sowohl das objektorientierte als auch das funktionale Programmierparadigma. Die Integration häufig verwendeter Datentypen wie Listen oder Maps in den Sprachkern oder die Verwendung typungebunder Variablen sind zusätzliche Konzepte der Sprache, die einen kurzen Entwicklungszyklus ermöglichen. Die Überprüfung der Typkompatibilität von Anweisungen, wie auch die Überprüfung der syntaktischen Korrektheit, erfolgt zur Laufzeit. Das spart langwierige Compilezeiten und erlaubt die sofortige Ausführung eines Skriptes. Skripte sind plattformunabhängig, da Python eine interpretierte Sprache ist. Ein weiterer Vorteil von Python ist die gute Lesbarkeit, die z.B. durch die obligatorische Einrückung von Blöcken gefördert wird.

Die geschilderten Eigenschaften bewegten die Entwickler des Projektes Marathon, Python als Skriptsprache zu wählen. Für die Integration von Python in das Testsystem wurde der Interpreter Jython genutzt. Dieser ist vollständig in Java implementiert und bildet eine Brücke zwischen beiden Sprachen. Jython ermöglicht die Umwandlung von Java- in Python-Typen und umgekehrt, d.h. es können in Python Java-Objekte instanziiert und manipuliert werden und Python-Klassen von Java-Klassen abgeleitet werden [19]. Auf diese Weise haben die Marathon Entwickler eine Möglichkeit geschaffen, die Components eines Java-Testobjekts aus einem Python-Skript heraus zu steuern. Dem Tester ist mit Python ein komplexes Werkzeug gegeben, mit dem er seine Testfälle ohne

jegliche Einschränkung spezifizieren kann. Ein Rahmen, der das Gebiet des Oberflächentests abgrenzt und eine einheitliche Nomenklatur und Semantik von Kommandos für diesen Bereich bereitstellt, existiert in Python nicht. Die uneingeschränkte Komplexität der Python-Skripte macht diese selbst zur Fehlerquelle und schränkt die allgemeine Verständlichkeit ein. Die effiziente Erstellung von Python-Skripten setzt eine umfangreiche Sprachkenntnis voraus, deren Erwerb mit einem nicht unerheblichen Aufwand verbunden ist, insbesondere bei Personen, die keine andere Programmiersprache beherrschen. Die Komplexität der Sprache und die daraus resultierenden Probleme haben uns, trotz aller genannten Vorteile, dazu veranlaßt, Python nicht als Skriptsprache für ATOSj zu nutzen.

4.3 HTS

Die Sprache HTS ist aus dem Vorgänger ATOS bekannt. Sie wird zur Definition von Testsequenzen genutzt. Eine Testsequenz beschreibt einen Testfall und besteht aus einer Folge von Testschritten. Ein Testschritt ist weiter unterteilt in eine Sequenz von Aktionsschritten, auf die wiederum eine Sequenz von Auswertungsschritten folgt. Aktionsschritte manipulieren das Testobjekt und überführen es in einen Zustand, der in den folgenden Auswertungsschritten auf seine Gültigkeit geprüft wird [7]. Für die konkrete Definition von Aktions- und Auswertungsschritten stehen verschiedenen Kommandos der Sprache HTS zur Verfügung. Jedes Kommando ist einzeilig und damit optisch genau von den anderen Kommandos abgegrenzt. Es wird durch eine kommaseparierte Liste von Parametern beschrieben und enthält nur wesentliche Informationen, die nicht mit Strukturinformationen vermengt werden. Die einfache Gestaltung der Kommandos macht HTS-Skripte besser les- und wartbar als vergleichbare XML-Skripte. HTS ist speziell für den Einsatz im Gebiet Oberflächentest konzipiert und schafft mit seinen spezifischen Kommandos eine Basis, die die allgemeine Verständlichkeit fördert. Der überschaubare Satz von Kommandos erleichtert das Erlernen der Sprache und steckt einen Rahmen für den Tester, der kaum Abweichungen vom Testziel erlaubt. Die begrenzte Komplexität macht HTS-Skripte im Allgemeinen besser verständlich und weniger fehleranfällig als Python-Skripte. Nachteil des starren HTS-Gerüsts ist die schlechte Anpassbarkeit an neue Anforderungen. Diese können nicht, wie in Python, mit den Mitteln der Sprache umgesetzt werden sondern erfordern eine Erweiterung der Sprachdefinition. Änderungen der Sprachdefinition bedingen Eingriffe in die Module des Testsystems, wie z.B. des HTS-Interpreters, und können nicht direkt durch den Tester vorgenommen werden. Dennoch haben wir uns für die Nutzung von HTS entschieden. Neben guter Lesbarkeit und Einfachheit überzeugte uns der erfolgreiche Einsatz von HTS in der Praxis. Es bewährte sich beim Regressionstest des Programmes XCTL, im Rahmen der Diplomarbeit von Herrn Hanisch und Herrn Letzel. Trotz der Komplexität des Testobjektes, welches mit 70000 Quelltextzeilen nicht mehr als trivial einzustufen ist, konnte mit den einfachen Mitteln der Sprache HTS eine fast vollstänge Funktionsprüfung durchgeführt werden, wie die Überdeckungsanalyse der von Herrn Hanisch erstellten Testskripte zeigt [7, S. 217ff.].

4.3.1 Änderungen zur ursprünglichen Version von HTS

Die Skriptsprache HTS wurde überarbeitet und erweitert um neuen Anforderungen gerecht zu werden und Inkonsistenzen der alten Version zu beheben. Herr Hirth nannte in seiner Diplomarbeit die Unterstützung weiterer Components als einen der wichtigsten Arbeitspunkte [8]. Auch wir haben das als ähnlich wichtig erachtet, denn die ursprüngliche HTS-Version war in diesem Punkt sehr eingeschränkt. So war es zum Beispiel nicht möglich Tabellen, Bäume und Karteikarten zu manipulieren, obwohl sie zum Standard gehören und in vielen Programmen zum Einsatz kommen, wie Tabelle 2 zeigt. Für die spezifische Aufgabe der Entwicklung von Regressionstests für das Programm XCTL waren die Möglichkeiten von HTS ausreichend, da es keines der genannten Elemente verwendet. Will man das Testwerkzeug jedoch universeller gestalten, muss das Spektrum der unterstützten Components zumindest um die wichtigsten Standard-Components erweitert werden. Deshalb haben wir den Baum, die Tabelle und die Karteikarte in die HTS-Syntax aufgenommen. Das erforderte die Definition neuer grafischer Objekttypen für HTS und eine entsprechende Erweiterung der Kommandos zur Manipulation, zur Abfrage und zur Verifikation der Zustände von Components. Für jeden neuen Typ wurde eine neues Schlüsselwort definiert, wie Tabelle 3 zeigt. Es wurden aber nicht nur bestehende Kommandos überarbeitet sondern HTS wurde auch um ein neues Kommando zur Ausführung bedingter Schleifen ergänzt. Die entwickelten Neuerungen werden nun für jedes Kommando im Einzelnen beschrieben.

Programm	**Baum**	**Tabelle**	**Karteikarte**
Explorer (explorer.exe)	X	X	
Systeminformationen (msinfo32.exe)	X	X	
Task-Manager (taskmgr.exe)		X	X
Systemkonfiguration (msconfig.exe)	X	X	X

Tabelle 2: Beispiele zur Verwendung von Standard-Components

ACTION
Das Kommando `ACTION` dient der Simulation von Nutzeraktionen und wurde für die Unterstützung der neuen Components ergänzt. Zu den neuen Möglichkeiten zählen z.B. die Auswahl von Einträgen in einem Baum oder das Editieren einer Tabellenzelle. Der vollständige Umfang aller neuen Aktionen kann der HTS Spezifikation im Anhang A.2 entnommen werden.
Eine weitere wichtige Änderung betrifft die Reihenfolge der Kommandoparameter. Der erste Parameter enthält den Titel des Fensters in dem sich das Component befindet, der zweite Parameter den Typ des Components und der dritte dessen Namen. Diese Parameter spielen in ATOSj eine besondere Rolle, denn sie bilden einen eindeutigen Identifikator, der zur Referenzierung des Components zur Laufzeit dient. Auf Grund ihrer inhaltlichen Zusammengehörigkeit werden sie auch syntaktisch zusammengehörig verwendet. Der Fenstertitel als unspezifischstes Element bildet den Anfang, so dass beim Betrachten der Kommandos eine grobe Orientierung möglich ist. In der ursprünglichen

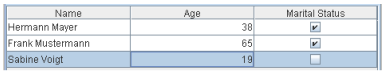
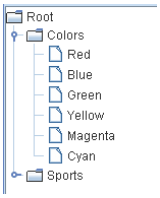
Bezeichnung	Schlüsselwort	Funktion	Swing Beispiel
Tabelle	`TABLE`	Matrixartige Darstellung von Daten mit der Möglichkeit zur Auswahl von Elementen	
Baum	`TREE`	Hierarchische Darstellung von Daten mit der Möglichkeit zur Auswahl von Elementen	
Karteikarte	`TABFOLDER`	Bietet die Möglichkeit der Auswahl einzelner Karteireiter	

Tabelle 3: Neue HTS Schlüsselwörter

HTS Version ist die Struktur etwas anders. Der Name des Components steht hier erst nach der Aktion, was die Lesbarkeit behindert. Deutlich wird dieser Nachteil in folgendem Beispiel:

`ACTION, “Fenster“, EDITBOX, EDIT, “Dr.“, “Titel“`
Dieses Kommando nach alter Syntax soll in die Editbox mit dem Namen „Titel“ die Zeichenkette „Dr.“ eintragen. Beide Parameter stehen direkt nacheinander und können somit leicht verwechselt werden.

`ACTION, “Fenster“, EDITBOX, “Titel“, EDIT, “Dr.“`
Dieses Kommando in neuer Syntax ist semantisch gleich zum vorgenannten Kommando. Eine Verwechslung des Namens und des Eingabetextes ist ausgeschlossen, wenn die ersten drei Parameter als zusammengehörige Einheit betrachtet werden.

Es gibt viele Arten von Components, die aus mehreren Unterelementen zusammengesetzt sind. Dazu gehören Bäume, Tabellen, Menüs, Listen und Karteikarten. Unterelemente einer Tabelle sind beispielsweise deren Zeilen oder auch einzelne Zellen. Bei der Präsentation von Daten in einer Tabelle entspricht eine Zeile häufig einem Datenobjekt in seiner Gesamtheit, wohingegen eine Zelle meist nur ein Attribut dieses Objekts darstellt. Zusammengesetzte Components bilden also eine konzeptuelle Einheit, die sich auch in der Syntax des Kommandos widerspiegeln sollte. Aktionen auf den Unterelemente, wie zum Beispiel das Editieren einer Tabellenzelle, sollten gut unterscheidbar sein von Aktionen auf dem zusammengesetzten Component selbst, wie z.B. ein Rechtsklick zum Öffnen eines Kontextmenüs. Zu diesem Zweck wurde das Schlüsselwort `SUBITEM` eingeführt. Es

gibt an, dass die Aktion auf einem Unterelement ausgeführt werden soll. Auf das Schlüsselwort folgt eine Liste von Parametern, die als Subelementpfad bezeichnet wird. Der Subelementpfad spezifiziert, welches der Unterelemente manipuliert werden soll. Die einzelnen Parameter des Pfades variieren je nach Typ des zusammengesetzen Components und nach der auszuführenden Aktion. Der Pfad kann für einige Aktionen beliebig lang sein und steht aus Gründen der Übersichtlichkeit immer am Ende des Kommandos. Die folgenden Beispiele zeigen kurz die Manipulation eines Unterelementes und die entsprechende Angabe des Subelementpfades.

`ACTION, ‘‘Fenster‘‘, TABLE, ‘‘kunden‘‘, SELECT, SUBITEM, 0`
Wählt die Zeile mit dem Index 0 aus der Tabelle mit dem Namen „kunden“ im Fenster „Fenster“ aus. Der Subelementpfad besteht nur aus dem Zeilenindex.

`ACTION, ‘‘Fenster‘‘, TREE, ‘‘geräte‘‘, EDIT, ‘‘Vorwerk‘‘, SUBITEM, ‘‘Geräte‘‘, ‘‘Staubsauger‘‘, ‘‘Vowerk‘‘`
Setzt den Text des Knotens mit der Beschriftung „Vowerk“ im Baum mit dem Namen „geräte“ auf „Vorwerk“. Der Subelementpfad besteht aus allen Beschriftungen der Elternknoten, beginnend beim Wurzelknoten, und dem zu editierenden Knoten selbst.

Die Unterstützung horizontaler und vertikaler Scrollbalken entfällt in der neuen HTS-Version, da deren direkte Verwendung nur für sehr spezielle Anwendungen nötig ist. Aus diesem Grund entfallen auch die Schlüsselwörter für die Typen `HSCROLLBAR` und `VSCROLLBAR`. Bei Zugriffen auf Unterelemente von zusammengesetzten Components werden die betroffenen Unterelemente automatisch in den sichtbaren Bereich gerückt, ohne die explizite Angabe von Kommandos zur Manipulation der Scrollbalken. Das verringert den Umfang der Testsequenzen und verbessert die Les- und Wartbarkeit.

Das Kommando `ACTION` wurde um zwei allgemeine Aktionsarten erweitert, die auf alle Typen anwendbar sind. Die Aktionsart `PRESSKEY` simuliert den einfachen Tastendruck eines Zeichens, optional kombiniert mit einer Sequenz von Steuertasten. Gültige Steuertasten sind Shift, Alt und Steuerung. Das wichtigste Anwendungsgebiet dieser Aktionsart ist die Simulation der Eingabe von Tastaturkürzeln. Tastenkürzel werden in vielen GUI-Programmen verwendet, um häufig durchgeführte Aktionen zu beschleunigen. Die Aktionsart `PRESSKEY` ersetzt das gleichnamige Kommando aus der Vorgängerversion. Die Spezifikation des Kommandos `PRESSKEY` erforderte die Angabe einer Zeitdauer für den Tastendruck. Eine solche Verwendung erscheint fragwürdig, denn sie ist semantisch nicht eindeutig. Der Tester kann aus dem Kommando nicht die Reaktion des Testobjektes ableiten, da nicht definiert ist wieviele Elementarereignisse (siehe Kaptitel 5.5) in der angegebenen Zeitspanne generiert werden sollen. Des Weiteren macht die Verwendung von Threads in ATOSj die exakte Einhaltung der Zeitdauer unmöglich. Ein Thread kann in seiner Ausführung unterbrochen werden, um einem konkurrierenden Thread die Abarbeitung zu ermöglichen. Auf diese Weise wird die quasi-parallele Abarbeitung von verschiedenen Aufgaben desselben Programmes realisiert. Durch die mögliche Unterbrechung ist die Abarbeitungszeit für einen Thread nicht determiniert. Deshalb wurde die

Semantik für die neue Aktionsart `PRESSKEY` umdefiniert. Sie ist eindeutig und bewirkt, dass die Eingabe der angegebenen Tastenkombination genau einmal simuliert wird.

Die zweite allgemeine Aktionsart, `CLICK`, simuliert einen Mausklick auf den Mittelpunkt eines Components. Es muss die Maustaste, wahlweise links oder rechts, und die Anzahl der auszuführenden Klicks angegeben werden. Optional besteht die Möglichkeit Steuertasten anzugeben, die während des Mausklicks aktiv sein sollen. Gültige Steuertasten sind Shift, Alt und Steuerung. Ein Kommando mit vergleichbarer Funktionalität fehlte bisher in HTS. Diese Einschränkung macht die Simulation von Doppel- oder Rechtsklicks in ATOS unmöglich und somit viele Programme auch untestbar. Die Aktion `CLICK` behebt dieses Problem und ermöglicht eine direkte Manipulation von Components für den Fall, dass die vordefinierten, semantischen Aktionen, wie z.B. das Auswählen einer Tabellenzeile, nicht genügen.

READ
Das Kommando `READ` liest Zustandswerte eines Components aus und speichert sie zur späteren Verwendung in einer Variable. Die Reihenfolge der Parameter des Kommandos wurde geändert, so dass auch hier die ersten drei Parameter den eindeutigen Identifikator des Components bilden. Der Umgang mit Unterelementen zusammengesetzter Components erfolgt ebenfalls konsistent zum Kommando `ACTION`.

Die wichtigste Neuerung für `READ` ist die Einführung zusätzlicher Zustandsabfragen für die neu unterstützten Components, siehe Tabelle 3. Dazu gehört beispielsweise das Auslesen des Textes von Tabellenzellen oder des Titels von Karteireitern. Es wurden aber auch Zustandsabfragen für die bisherigen Typen erweitert. So wurde das Konzept des Knopfes vereinheitlicht und es ist nun auch möglich die Beschriftung für die Typen `RADIOBUTTON` und `CHECKBOX` auszulesen und ggf. als Zahl zu interpretieren. Das Auslesen der Beschriftung war in der vorhergehenden Version von HTS lediglich für den Typ `BUTTON` möglich. Eine weitere Neuerung ist die Möglichkeit, die Anzahl der Elemente bzw. Zeilen in einer Tabelle oder Liste auszulesen. Die Elementanzahl ist ein wichtiges Merkmal, welches Rückschlüsse über die Korrektheit des Testobjektes zuläßt. Tabellen und Listen werden häufig genutzt, um eine Sammlung von Datenobjekten darzustellen. Meist können dieser Sammlung Objekte hinzugefügt oder daraus entfernt werden, diese Veränderung muss sich dann in der Elementanzahl der Tabelle oder Liste widerspiegeln. Die letzte wichtige Erweiterung ermöglicht das Auslesen des Anwahlstatus (Häkchen/ kein Häkchen) einer Menüoption. Diese Möglichkeit fehlt in der Vorgängerversion, weil sie technisch nicht umsetzbar war [7].

TEST
Die Änderungen bezüglich der Angabe des eindeutigen Identifikators und der Verwendung von Unterelementen gelten auch für das Kommando `TEST`. Das Kommando `TEST` ermittelt in einem Schritt den aktuellen Zustandswert eines Components und vergleicht ihn mit einem Sollwert. Über das Kommando `READ` werden ebenfalls Zustandswerte ermittelt, jedoch für einen späteren Vergleich in einer Variablen gespeichert. Die Verwandschaft

der beiden Kommandos legt nahe, dass für das Ermitteln der Zustandswerte dieselben Schlüsselwörter gebraucht werden.

Neue Syntax:

```
READ, MAIN, EDITBOX, ‘‘Name‘‘, ENABLESTATE, ‘‘var‘‘
TEST, MAIN, EDITBOX, ‘‘Name‘‘, ENABLESTATE, TRUE
READ, MAIN, EDITBOX, ‘‘Name‘‘, TEXT, ‘‘var‘‘
TEST, MAIN, EDITBOX, ‘‘Name‘‘, TEXT, ‘‘Mustermann‘‘
```

Alte Syntax:

```
READ, MAIN, EDITBOX, ENABLESTATE, var, ‘‘Name‘‘
TEST, MAIN, EDITBOX, DISABLED, ‘‘Name‘‘
READ, MAIN, EDITBOX, TEXT, var, ‘‘Name‘‘
TEST, MAIN, EDITBOX, TEXT, ‘‘Mustermann‘‘, ‘‘Name‘‘
```

Wie man im Beispiel leicht sieht, ist das Kommando `TEST` nach der alten Syntax weder intern noch extern konsistent. Die interne Konsistenz fehlt, denn beim Abprüfen einiger Zustände muss der Sollwert explizit angegeben werden, wie z.B. beim textuellen Sollwertvergleich, wohingegen beim Abprüfen anderer Zustände der Sollwert implizit in der Anweisung enthalten ist, wie z.B. beim Überprüfen des Aktivierungsstatus eines Components. Auf Grund der impliziten Verwendung von Sollwerten wird auch die externe Konsistenz zum Kommando `READ` verletzt. Nicht alle Bezeichnungen für einen mit `READ` ausgelesenen Zustand, beispielsweise `ENABLESTATE`, finden sich im Kommando `TEST` wieder. Stattdessen werden teilweise Schlüsselwörter verwendet, die gleichzeitig Zustand und Sollwert beinhalten, wie z.B. `DISABLED`. Ursache für diese Inkonsistenz ist die fehlende Explizierung eines booleschen Vergleichs. Um diese Inkonsistenz zu beseitigen, ermöglicht die neue HTS Version auch eine Zustandsprüfung mit booleschen Sollwerten. Dazu wurden die Schlüsselwörter `TRUE` und `FALSE` für die Werte „wahr“ und „falsch“ eingeführt. Die Struktur des booleschen Vergleichs stimmt mit der Struktur aller anderen Zustandsprüfungen überein und erfordert zuerst die Angabe des abzuprüfenden Zustandes, gefolgt vom Sollwert. Damit sind sowohl interne als auch externe Konsistenz hergestellt. Ein Beispiel für einen booleschen Zustand ist der Aktivierungsstatus, denn er besitzt zwei duale Ausprägungen, aktiviert und deaktiviert, wobei aktiviert dem Wert „wahr“ und deaktiviert dem Wert „falsch“ entspricht. Außerdem gehören die Sichtbarkeit (`VISIBLESTATE`) und der Fokusstatus (`FOCUSSTATE`) zu den booleschen Zuständen eines Components.

COMPARE

Das Kommando `COMPARE` arbeitet eng mit dem Kommando `READ` zusammen und ermöglicht den Vergleich von Variablen mit anderen Variablen oder festen Werten. Es wurde, ebenso wie das Kommando `TEST`, um die Möglichkeit eines booleschen Vergleichs erweitert. Diese Erweiterung war zwingend notwendig, da bisher ein numerischer Vergleich den Vergleich mit booleschen Werten ersetzte. Das entspricht nicht der Intuition und erfordert die Kenntnis der internen Repräsentation einer Variablen. Boolesche Zustände, die mit Hilfe von `READ` ausgelesen wurden, müssen nun auch mit booleschen Werten ver-

glichen werden. Die nachfolgenden Beispiele sollen den Unterschied zwischen alter und neuer Syntax illustrieren.

Alte Syntax:

```
READ, MAIN, BUTTON, ENABLESTATE, var, ‘‘OK‘‘
COMPARE, var, NUM, VAL, 0
```

Im ersten Schritt wird der Aktivierungsstatus des Knopfes mit dem Namen „OK“ ausgelesen und in der Variable „var“ gespeichert. Im zweiten Schritt wird dann die eigentlich boolesche Variable mit dem numerischen Wert 0 verglichen. Die 0 steht für den Wert „falsch“, d.h. es wird erwartet, dass der Knopf nicht aktiviert ist. Soll ein Vergleich mit dem Wert „wahr“ durchgeführt werden, muss eine 1 angegeben werden. Die Eingabe ist jedoch nicht auf diese beiden Werte beschränkt sondern offen für alle reellen Zahlen und die Angabe von Vergleichsoperatoren wie „größer als“ oder „kleiner als“. Für die Durchführung eines booleschen Vergleichs ist die alte Syntax unangemessen.

Neue Syntax:

```
READ, MAIN, BUTTON, ‘‘OK‘‘, ENABLESTATE, ‘‘var‘‘
COMPARE, ‘‘var‘‘, BOOL, VAL, FALSE
```

Dieses Beispiel ist gleichbedeutend mit dem zuvor gezeigten. Die neue Version erfordert keine Transformation von booleschen in numerische Werte und vereinfacht damit die Eingabe und Interpretation des Kommandos.

WHILE

Das Kommando `WHILE` wurde neu eingeführt. Es stellt eine bedingte Schleife dar, wie sie aus vielen anderen Programmiersprachen bekannt ist. Alle Kommandos im Schleifenkörper werden so lange ausgeführt, bis die Bedingung im Schleifenkopf zu „falsch“ evaluiert. Syntaktisch stimmen die Parameter der Kommandos `WHILE` und `COMPARE` überein, jedoch führt ein Vergleich, der das Ergebnis „falsch“ liefert nicht zu einem Fehler sondern lediglich zum Abbruch der Schleife. Dieses Kommando wurde entwickelt, um dem Tester eine zusätzliche Möglichkeit zur Vermeidung von Synchronisationproblemen zu geben. Synchronisationsprobleme treten immer dann auf, wenn mit der Ausführung einer Testsequenz erst fortgefahren werden kann, nachdem das Testobjekt einen bestimmten Zustand erreicht hat.

Zur Synchronisation mit dem Testobjekt stand bislang nur das Kommando `WAIT` zur Verfügung [8]. Es ermöglicht die Unterbrechung einer Testsequenz für eine festgelegte Zeitdauer. Dies ist jedoch ineffizient und fehleranfällig, denn Zustandswechsel treten häufig nach einer nicht deterministischen Zeitspanne ein. Ineffizient deshalb, weil immer die komplette Zeit abgewartet werden muss, auch wenn der erwartete Zustand schon früher eingetreten ist. Insbesondere für einen automatisierten Test ist es aber von höchster Priorität, einen Testlauf in möglichst kurzer Zeit durchzuführen. Die Anwendung von `WAIT` ist fehleranfällig, weil die Zeit, die für einen Zustandswechsel benötigt wird, leicht unterschätzt werden kann. Eine Unterschätzung bewirkt das Fortfahren der Testsequenz, bevor der erwartete Zustand erreicht wurde. Dann sind Annahmen, die für den weiteren

Testablauf getroffen wurden, ungültig und können zu Fehlern führen.

Das neue Kommando `WHILE` löst die beschriebenen Probleme. Es ist effizienter als `WAIT`, da es nur so lange als nötig auf einen Zustandswechsel wartet. Es ist weniger fehleranfällig, da eine Unterschätzung der Wartedauer unmöglich ist. Nachteilig dagegen ist die Gefahr einer Endlosschleife. Sie kann zwei Ursachen haben. Zum einen einen Fehler im Testskript, der auf die fehlende Aktualisierung der abzutestenden Zustandsvariable zurückzuführen ist. Zum anderen einen Fehler im Testobjekt auf Grund dessen der erwartete Zustand nie erreicht wird.

Zur Veranschaulichung der Synchronisationsproblematik soll ein einfaches Beispiel genügen. Ein fiktives Testobjekt soll, laut Pflichtenheft, Dateitransfers zwischen zwei Instanzen des Programmes via TCP/IP ermöglichen. Diese Anforderung sei über einen einfachen Dialog realisiert, der die Datei und den Empfänger ermittelt. Sind beide Angaben korrekt eingetragen, kann die Datei durch Drücken des Buttons „Senden“ abgeschickt werden. Weiterhin enthält der Dialog die Knöpfe „Abbrechen“ - zum Abbrechen des Sendevorgangs und den Knopf „Beenden“ - zum Schließen des Dialogs. Der Knopf „Beenden“ soll nur aktiv sein, wenn die Datei vollständig gesendet oder der Vorgang abgebrochen wurde. Nach dem Abschicken einer Datei zeigt der Aktivierungsstatus des Knopfes „Beenden“ an, ob der Sendevorgang beendet ist oder nicht. Das Senden der Datei stellt den Tester vor ein Synchronisationsproblem, denn es muss eine unbestimmte Zeitspanne gewartet werden bevor der „Beenden“ Knopf gedrückt werden kann. Die genaue Sendezeit hängt von nicht kalkulierbaren Faktoren, wie der Netzlast oder der Geschwindigkeit der Netzwerkverbindung, ab. Die Lösung dieses Problems sähe unter Anwendung des neuen Kommandos `WHILE` wie folgt aus:

```
READ, ‘‘Datei verschicken‘‘, BUTTON, ‘‘Beenden‘‘, ENABLESTATE, ‘‘senden beendet‘‘
WHILE, ‘‘senden beendet‘‘, BOOL, VAL, FALSE
READ, ‘‘Datei verschicken‘‘, BUTTON, ‘‘Beenden‘‘, ENABLESTATE, ‘‘senden beendet‘‘
ENDWHILE
```

5 Architektur des Testsystems

5.1 Grobstruktur mittels UML-Diagrammen

Beim Entstehen dieser Diplomarbeit ging es zunächst darum zu überprüfen, ob ein automatisiertes Testen, ähnlich dem wie es in ATOS geschieht, für Java-Programme überhaupt möglich ist. Sowohl durch theoretische Überlegungen, als auch durch bereits existierende Projekte erschien dies sehr wahrscheinlich. Für den praktisch Nachweis entstand zunächst eine rudimentäre Implemtentation der die Machbarkeit bestätigte. Die eigentliche Implemtation setzte später auf dieser Studie auf. Das gewählte Entwicklungsmodell entspricht also durchaus dem Grundansatz des Prototypings. Durch dieses Vorgehen ist das entstandene Programm eigentlich als einheitliches System anzusehen. Dennoch lassen sich insgesamt 14 verschiedene Hauptkomponenten identifizieren.

Da ATOSj für seine Funktion als Java-Testumgebung ein sehr breites Funktionsspektrum abdecken muss, werden einige Funktionen durch eingebundene Bibliotheken realisiert. Insgesamt benutzt ATOSj drei Fremdbibliotheken:

IText: Aufgabe dieses Moduls ist die Generierung von PDF-Dateien. In ATOSj werden die generierten Reporte der Testsequenzen mittels IText in das PDF-Format konvertiert. Für Java gibt es diverse Bibliotheken, die sich dieser Aufgabenstellung annehmen.(http://java-source.net/open-source/pdf-libraries) Die Wahl auf IText fiel Aufgrund der höheren Verbreitung im Verhältnis zu sämtlichen Konkurrenzprodukten.

Javassist: Diese Bibliothek ermöglicht das Editieren der SWT-Klassen, für die spätere Überwachung während des Capturens bzw. die Interaktion während des Abspielens von Testsequenzen. Ursprünglich entwickelt unter der Leitung von Shigeru Chiba am Tokyo Institute of Technology, bildet diese Bibliothek mittlerweile einen Bestandteil des JBoss Projektes. In ATOSj bildet sie des Weiteren die Grundlage für die Überdeckungsanalyse mittels ACover.

JAccess: Die Java Accessibility Utilities wurden von Sun ursprünglich als optionale Erweiterung der Java API konzipiert. In ATOSj sind Sie Hilfsmittel für die Interaktion mit der GUI von Swing-Testobjekten. Obwohl JAccess-Klassen Oberflächenvents noch vor den eigentlichen Swing- und AWT-Klassen erhalten sollen, müssen Sie dem Compiler nicht gesondert bekannt gemacht werden. Dies ist dadurch begründet, dass die Java Virtual Machine seit Version 1.3 automatisch nach den Jaccess Klassen sucht, und falls sie diese findet implizit eine geänderte AWT-Eventqueue, die für das Verteilen der Events zuständig ist, lädt.

Alle im folgenden aufgeführten weiteren Komponenten sind vollständig selbst implementiert.

SWTEventMonitor:

Diese Komponente stellt das Pendant der Java Accessibility Utilities für die SWT-Klassen dar. Da es keine vorgefertigte Funktionalität zur Überwachung von SWT-Fenstern gibt, musste dieser Teil selbständig implementiert werden. Eine detaillierte Erläuterung der Funktionsweise ist in Kaptitel 5.10) zu finden.

- Implementation in Paket atosj.util.swteventmonitor

RMIConnectionManager:

ATOSj führt Testobjekte stets in einem anderen Prozess aus. Damit das Testsystem mit diesen zu testenden Oberflächen kommunizieren kann, muss es Nachrichten an die zugehörigen Prozesse verschicken. In der Java-API existiert bereits eine RMI genannte Komponente, die eine solche Interprozesskommunikation ermöglicht. Auf dieser basierend verwaltet der RMIConnectionManager sämtliche existierende RMI-Verbindungen für ATOSj. Des Weiteren ist er für das Erstellen der Prozesse der Testobjekte zuständig. Die exakten Vorgänge werden in Kapitel 5.8 näher beschrieben.

- Implementation in Paket atosj.rmi

ACover:

Als unterstützendes Instrument zur Untersuchung der Testobjekte bietet ATOSj die Möglichkeit, parallel zur Ausführungen von Skripten im Testobjekt eine Überdeckungsanalyse vorzunehmen. Sämtliche Klassen, die mit dieser Funktion in Berührung stehen, sind in der Komponente ACover zusammengefasst.

- Implementation in Paket acover

Testsequenzen (Skripte):

Bevor ATOSj mit der Ausführung von Testsequenzen im Testobjekt beginnen kann, werden die referenzierten Skripte und Pakete in eine interne Darstellung gebracht. Die Komponente der Testsequenzen enthält diese Darstellung und die für die Erstellung, Verwaltung und das anschließende Zurückschreiben in die HTS-Datei zuständigen Klassen.

- Implementation in Paket atosj.model.script

Pakete:

Analog zu den Testsequenzen sind in diesem Paket alle Klassen enthalten, die für die Verwaltung der Paketdateien (.pak) verantwortlich sind.

- Implementation in Paket atosj.model.pack

HTS-Befehle:

Auch HTS-Befehle werden vor Ihrer Ausführung zunächst in eine Java-Klassendarstellung gebracht. Für jeden HTS-Befehl gibt es eine eigene Implementationsklasse. Diese Klassen sind hierarchisch angeordnet und vererben sich Ihre Fähigkeiten. So ist Beispielsweise die Klasse HTSLoop, die eine Schleife in einem HTS-Script ermöglicht, von der Klasse HTSCollectionCommand abgeleitet, die für die Verwaltung der umschlossenen Befehle zuständig ist. Diese Klasse ist wiederum Instanz der allgemeinsten Klasse HTSCommand, die die Basisfunktionalitäten, die für alle HTS-Befehle nötig sind, zur Verfügung stellt.

- Implementation in Paket atosj.model.hts.command

Komponentenklassen (Swing & SWT):

In dieser Komponente sind sämtliche Funktionen, die für die Interaktion mit den Oberflächenelementen nötig sind, zusammengefasst. Zu jedem GUI-Standardelement existiert eine Wrapperklasse, die die jeweiligen Aktionen ermöglicht. Eventuelle Wrapperklassen, die für eigene CustomComponents geschrieben werden, sind eine Erweiterung dieses ATOSj-Teils. Auch hier wurde so weitestgehend versucht Funktionalitäten in Basisklassen zu verlagern, weshalb die Klassen eine tiefe Hierarchie aufweisen.

Um eine saubere Architektur zu gewährleisten, sind beide Implementation streng voneinander getrennt. Für beide Oberflächentypen existiert ein eigenes Paket in der die Implementation enthalten ist.

- atosj.component.swing

- atosj.component.swt

Die folgenden drei Komponenten sind sich durchaus ähnlich. Allen ist gemein, dass Sie jeweils die Umgebung für ein Testobjekt darstellen, das vom ATOSj-Hauptprogramm in einem anderen Prozess gestartet wurde. Alle drei übernehmen dabei jeweils die Kommunikation mit diesem, und führen ggf. spezielle Aktionen im Testobjekt aus, wenn Sie von ATOSj dazu aufgefordert werden. Die Kommunikation mit dem Hauptprogramm erfolgt dabei jeweils mittels RMI.

HTSPlayer:

Der HTSPlayer ist das Verwaltungsobjekt für ein Testobjekt das zur Ausführung eines Skriptes oder Paketes erstellt wurde (Replay). Aufgabe des HTSPlayers ist es, dem Hauptprogramm ein Interface bereitzustellen, auf dem ihm dieses den Auftrag zu Ausführung eines neuen HTS-Befehls übergeben kann. Anschließend ist er für dessen Abarbeitung, die Ermittlung des Ergebnisses dieser Aktion und die anschließenden Statusmeldung an ATOSj verantwortlich.

- Implementation in Paket atosj.rmi

HTSRecorder:

Im Gegensatz zum HTSPlayer ist der HTSRecorder nicht für das Abspielen, sondern für die Aufnahme von Skripten im Testobjekt zuständig (Capture). Er überwacht das ihm zugewiesene Testobjekt auf Nutzeraktionen, erzeugt bei passenden Events die entsprechenden HTSBefehle und übermittelt diese anschließend dem ATOSj-Hauptprogramm.

- Implementation in Paket atosj.model.recorder

URFGenerator:

Auch der URFGenerator ist für die Überwachung eines Testobjekts in einem separaten Prozess zuständig. Seine spezielle Aufgabe ist die Überwachung sämtlicher zur Ausführungszeit existierender Fenster. Wird im Testobjekt durch eine Nutzeraktion ein neues Fenster erstellt, so genierert er zu jedem in diesem Fenster vorhandenen GUI-Element eine entsprechende ID. Falls diese zuvor noch unbekannt war, so übermittelt der URF-Generator diese an die ATOSj-HauptGUI. Auf diese Weise erfährt ATOSj nach und nach von allen exsitierenden GUI-Elementen, und kann somit die zu diesem Testobjekt zugehörige .urf-Datei vervollständigen. Die .urf-Datei wird später genutzt, um dem Anwender beispielsweise Vorschläge für die manuelle Generierung von HTS-Befehlen zu unterbreiten.

- Implementation in Paket atosj.model.urf

GUI: Der vom Quellcodeumfang deutlich größte Abschnitt von ATOSj sind die Klassen für die Nutzeroberfläche. Die beträchtliche Größe von vielen Tausend LOC liegt zum einen im etwas umständlichen Entwicklungsprinzip von Java-GUIs begründet. So muss jedes verwendete Objekt vor seiner Verwendung zunächst detailliert konfiguriert werden. Bei aufwändigeren Layouts kann dies durchaus zu durchschnittlich 10 Zeilen Code pro Element führen. Zum anderen wurde bei der Entwicklung der ATOSj-GUI verstärkt auf Anwenderfreundlichkeit geachtet. So sind etwa für viele Aktionen Tastenkombinationen definiert, und es wurde ein recht aufwändiger Copy&Paste-Algorithmus für sämtliche Objekte implementiert, um ein schnelles Bearbeiten der Skripte zu ermöglichen.

Aufgrund Ihres Umfangs ist diese Komponente in sich erneut gegliedert. Schnittstellen zur GUI finden sich in allen Komponenten von ATOSj. Die folgenden Pakete beinhalten den größten Teil der GUI-Implementation:

- atosj.app
- atosj.dialog
- atosj.editors
- atosj.gui

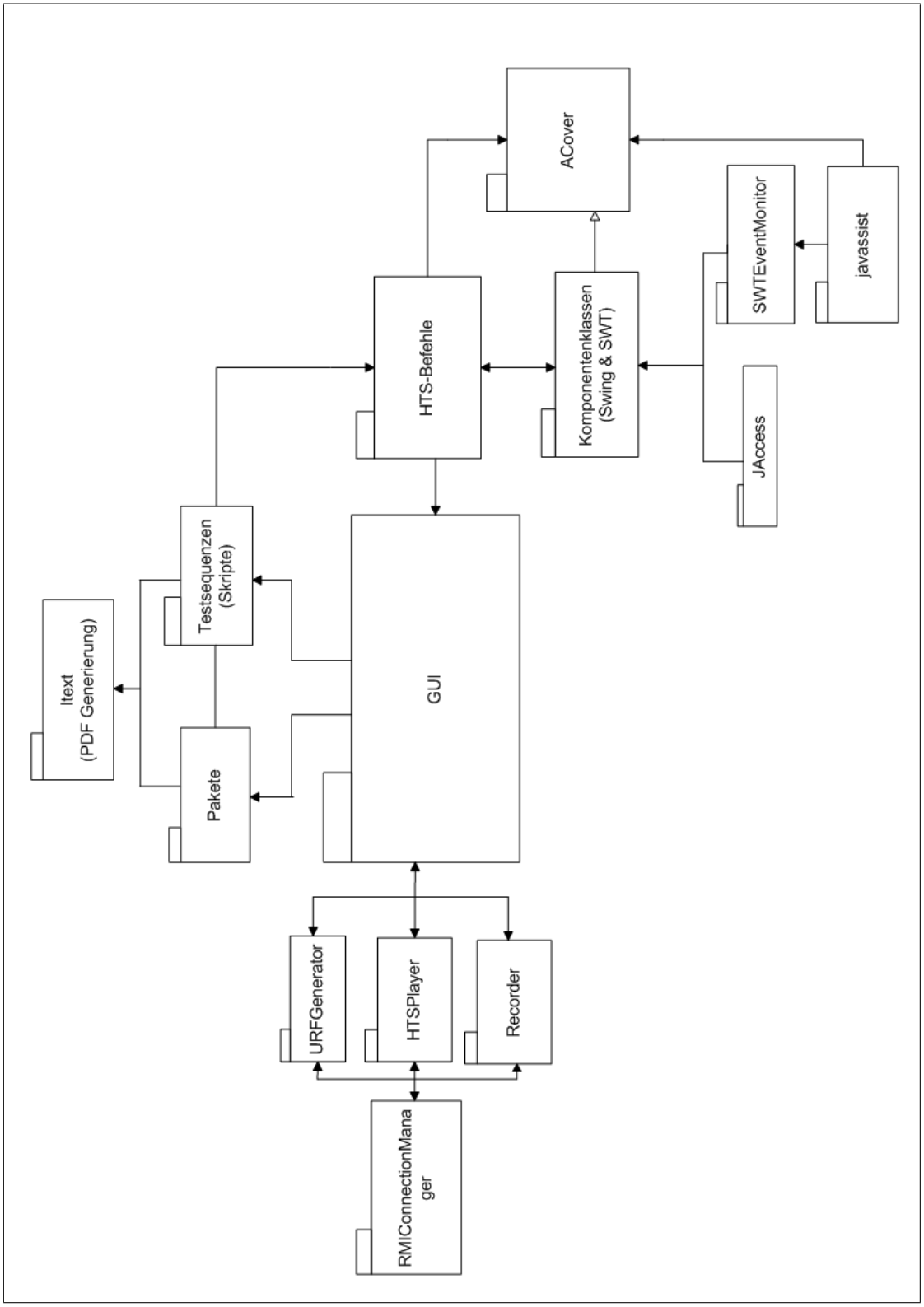


Abbildung 2: Gesamtübersicht Pakete

5.2 Kapselung GUI-spezifischer Funktionalität

Eine der wichtigsten Aufgaben der Programmbibliothek ATOSj ist es, den Zugriff auf die Components des Testobjekts zu ermöglichen. Zu diesem Zweck wurde eine 2-Schichten-Architektur entwickelt, die den spezifischen Code für die Unterstützung der unterschiedlichen GUI-Bibliotheken kapselt und einen einheitlichen Zugriff auf alle Components ermöglicht.
Die erste Schicht bilden plattformunabhängige Interfaces, die vergleichbar mit den Peer-Interfaces des AWT sind, siehe Kapitel 5.5. Grundlage für diese Schicht waren die Typen der Skriptsprache HTS. Ein HTS-Typ ist die Abstraktion eines grafischen Elements auf Basis seiner funktionalen Eigenschaften. Diese Eigenschaften lassen sich allgemein in zwei Gruppen unterteilen. Auf der einen Seite stehen die Manipulationsmöglichkeiten, die das Component dem Nutzer bietet, auf der anderen Seite stehen die verschiedenen Zustände, die das Component annehmen kann. Beide Gruppen sind kausal voneinander abhängig, denn Aktionen führen zu Zustandsänderungen. Diese Abhängigkeit begründet das Prinzip des Oberflächentests und damit auch der Skriptsprache HTS. Aktionen werden in HTS durch das Kommando `ACTION` ausgeführt, das abbhängig vom Componenttyp verschiedene Aktionsarten zur Verfügung stellt. Dual dazu definieren die Kommandos `READ` bzw. `TEST` für jeden HTS-Typ eine Menge von Zuständen, die zur Verifikation der Wirkung von Aktionen abgefragt werden können. Zu jedem HTS-Typ wurde ein Java-Interface erstellt, dessen Methoden den Aktionsarten und Zustandsabfragen des HTS-Typs entlehnt wurden. Da die Schnittmenge der Eigenschaften einiger HTS-Typen nicht leer ist, wurde zur Vermeidung von Redundanzen eine Vererbungshierarchie zwischen den Interfaces festgelegt. So erbt z.B. das Interface für den Typ `BUTTON` vom Interface für den Typ `LABEL`. Trotzdem ist jedem HTS-Typ immer genau ein Interface zugeordnet, welches im Weiteren als Typ-Interface bezeichnet werden soll. Die Assoziation zwischen einem HTS-Typ und seinem Typ-Interface wird in den Tabellen 4 und 5 gezeigt.

HTS-Typ	**Interface**	**Aktionsart**	**Methode**
EDITBOX	IEditbox	EDIT	setText(String)
CHECKBOX	ICheckbox	CHECK	setChecked(boolean)
TABFOLDER	ITabFolder	SELECT	select(int)

Tabelle 4: Simulation von Nutzeraktionen

HTS-Typ	**Interface**	**Zustand**	**Methode**
EDITBOX	IEditbox	TEXT	getText()
CHECKBOX	ICheckbox	CHECKSTATE	isChecked()
TABFOLDER	ITabFolder	ITEMCOUNT	getItemCount()

Tabelle 5: Auslesen von Zuständen

Die zweite Schicht unserer Architektur bilden die plattformabhängigen Implementationen

der Typ-Interfaces. Sie werden in zwei Schritten erstellt. Zuerst wird zu jedem HTS-Typ eine Klasse identifiziert, die seinen funktionalen Eigenschaften entspricht. Dieser Schritt wird getrennt für jede der unterstützten GUI-Bibliotheken ausgeführt. Dem HTS-Typ `TABFOLDER` beispielsweise entsprechen die Klassen `JTabbedPane` aus der Bibliothek Swing und `Tabfolder` aus der Bibliothek SWT. Im nächsten Schritt wird für jede identifizierte Klasse eine Wrapperklasse entwickelt, die ein Typ-Interface implementiert. So ummantelt z.B. der `SWTTabFolder` die Klasse `Tabfolder` und implementiert das Interface `ITabFolder`.

5.3 Integration von HTS in ATOS und ATOSj

Sowohl ATOS als auch ATOSj verwenden die Skriptsprache HTS. Dennoch unterscheidet sich die Arbeitsweise des HTS-Interpreters bzw. -Parsers in beiden Systemen grundlegend. Im Gegensatz zu ATOSj verwendet ATOS intern eine zweite Windows-spezifische Skriptsprache mit dem Namen ATS (Atomic Testscript). Kennzeichen für diese Lowlevel-Sprache ist die Steuerung und Abfrage von Components durch einige elementare Kommandos, die in ihrer Kombination komplexe Aktionen ermöglichen. In ATOS werden HTS-Skripte nicht direkt geparset sondern zuvor in ATS-Skripte umgewandelt. Dabei wird jedes HTS-Kommando auf eine Menge von ATS-Kommandos abgebildet. Die Abbildungsvorschriften müssen in einer speziellen Regeldatei definiert werden. Jede einzelne Regel besteht aus zwei Teilen. Im ersten Teil wird ein Muster (PATTERN) definiert, das ein einzelnes HTS-Kommando beschreibt. Im zweiten Teil steht die Ausgabe der Abbildung (OUTPUT). Während des Parsevorgangs wird jede Zeile des HTS-Skripts geprüft, ob sie auf das Muster einer Regel passt. Wurde eine passende Regel gefunden, werden alle ATS-Kommandos generiert, die in ihrem Ausgabeteil definiert wurden. Einzelne Parameter des HTS-Kommandos können in Variablen gespeichert werden, die dann bei der Erzeugung der Ausgabe verwendet werden können.
Das Beispiel in Listing 2 zeigt die Umwandlung des HTS-Kommandos zur Simulation eines Knopfdrucks. Für die Abbildung muss die semantische Aktion in eine Folge von elementaren Aktionen umgewandelt werden. Zu jeder elementaren Aktion wird ein entsprechendes ATS-Kommando generiert. Im Beispiel gehört dazu das Setzen des Eingabefokus (Zeile 8) und der eigentliche Mausklick (Zeile 10). Vor der Simulation der Aktionen muss noch geprüft werden, ob der Knopf aktiviert ist (Zeile 4). Tritt bei der Ausführung eines Kommandos ein Fehler auf, sorgen die `ERROR` Direktiven für eine klare Fehlerbeschreibung.

```
1 RULE
2     STATE     "NORMAL"
3     PATTERN   "ACTION",<SAVE:Window>,"BUTTON","CLICK",<SAVE:Target>
4     OUTPUT    "ISENABLED","CONTROL",<LOAD:Window>,<LOAD:Target>
5     OUTPUT    "ERROR","14","Das Fenster konnte nicht gefunden werden!"
6     OUTPUT    "ERROR","15","Der Button konnte nicht gefunden werden!"
7     OUTPUT    "ERROR","18","Der Button ist nicht aktiv!"
8     OUTPUT    "SETFOCUS",<LOAD:Window>,<LOAD:Target>
```

```
9     OUTPUT    "ERROR","20","Eingabefokus konnte nicht gesetzt werden!"
10    OUTPUT    "POST",<LOAD:Window>,<LOAD:Target>,"245","0","0"
11    OUTPUT    "WAIT","500"
12    NEWSTATE  "NORMAL"
13 ENDRULE
```

Listing 2: Umwandlung eines HTS-Kommandos

Nach der Umwandlung in ein ATS-Skript übernimmt die Klasse `ATSParser` die syntaktische Prüfung und auch die Ausführung der ATS-Kommandos. Die wichtigsten Kommandos sind `SEND` bzw. `POST`. Sie nutzen die Funktionen `SendMessage` und `PostMessage` aus der Windows-API und ermöglichen das Senden von Nachrichten an die Components einer Anwendung. Wie in Kapitel 5.5 beschrieben, verschickt auch das Betriebssystem Nachrichten, um Components über das Auftreten von Nutzeraktionen zu informieren. Mit den genannten Kommandos ist also die Simulation von Nutzeraktionen möglich. Außerdem können sie zur Statusabfrage genutzt werden, denn das Senden bestimmter Nachrichten liefert einen Rückgabewert mit Informationen über den Empfänger.

In ATOSj wurde auf die Lowlevel-Sprache ATS verzichtet. Stattdessen werden die HTS-Kommandos direkt geparset und interpretiert. Da ein HTS-Kommando eine abgeschlossene Einheit bildet, bot es sich an für jedes Kommando eine eigene Klasse zu definieren. Die Kommando-Klasse übernimmt die syntaktische Prüfung und die Ausführung eines HTS-Kommandos. Die vorgestellte 2-Schichten-Architektur ermöglicht den transparenten Zugriff auf Components innerhalb der Kommando-Klassen. Es werden nie direkt die Schnittstellen der GUI-Bibliothek des Testobjekts genutzt sondern nur die von ATOSj bereitgestellten Typ-Interfaces. Damit wird die Unabhängigkeit des HTS-Parsers von den unterstützten Bibliotheken gewährleistet.

Die externe Defininition von Regeln macht den Umgang mit HTS unter ATOS für den Nutzer transparent und auch flexibel. Mit Kenntnis der Windows-API und der Skriptsprache ATS kann genau bestimmt werden welche Nachrichten ATOS zur Simulation von Nutzeraktionen an die Components des Testobjekts sendet. Von Vorteil ist auch die freie Konfiguration von Fehlernachrichten, die beispielsweise für die englische ATOS-Version in der Regeldatei leicht abgeändert werden können. Durch die Definition neuer Regeln kann der Nutzer den Sprachumfang von HTS selbständig erweitern. Dennoch sind die Erweiterungsmöglichkeiten sehr begrenzt. Insbesondere die Unterstützung komplexer Components ist mit der bisherigen ATS-Version nicht möglich, denn dazu müssen Nachrichten mit komplexen Parametern versendet und ausgewertet werden. Die Kommandos `SEND` und `POST` können jedoch nur Nachrichten mit einfachen Zeichenketten oder numerischen Parametern behandeln. So ist es z.B. nicht möglich, den Text einer Tabellenzelle auszulesen, denn die entsrechende Nachricht `LVM_GETITEMTEXT` beinhaltet einen komplexen Parameter:

```
wParam = (WPARAM) (int) iItem;
lParam = (LPARAM) (LPLVITEM) pitem;
```

Der Parameter `wParam` enthält den Zeilenindex der Tabellenzelle. Der zweite Nachrichtenparameter (`lParam`) enthält einen Zeiger auf eine komplexe Struktur, die die Tabellenzelle näher beschreibt. Sie enthält unter anderem den Spaltenindex und einen Puffer, der den ermittelten Text aufnehmen soll. Für Nachrichten dieser Art müssten neue ATS-Kommandos auf einem höheren Abstraktionsniveau eingeführt werden. Auch die Einführung des Kommandos `ISENABLED` in ATS hat schon gezeigt, das grundlegende Funktionalität nicht allein mit den Kommandos `SEND` und `POST` gewährleistet werden kann und abstraktere Kommandos zwingend erforderlich sind. Das weicht die Trennung zwischen ATS und HTS auf und stellt das gesamte Konzept der Zweiteilung in frage. Auch in Hinblick auf die Erweiterung für die Unterstützung zusätzlicher GUIs scheint die Trennung zwischen ATS und HTS problematisch. Theoretisch ließe sich ATS durch andere Lowlevel-Sprachen austauschen, praktisch ist eine Umsetzung eher schwierig. In Java beispielsweise können GUI-Elemente nicht wie in Windows durch einen einheitlichen Nachrichtenmechanismus angesprochen werden. Vielmehr bedarf es des Aufrufes unterschiedlicher Methoden zur Zustandsabfrage und der Generierung von Elementarereignissen zur Steuerung der Components. Diese Heterogenität macht die Behandlung von Java-Components durch wenige elementare Kommandos, wie für eine ATS-ähnliche Sprache gefordert, unmöglich.

Die Abbildung einer Hochsprache auf eine elementare Sprache ist ein häufig verwendetes Konzept, dass aber in ATOS nicht zufriedenstellend umgesetzt werden konnte. ATS blockiert die Erweiterung von HTS, da es nicht mächtig genug ist, auch komplexe Components anzusteuern. Das Plugin-Konzept von ATOSj dagegen sichert die externe Erweiterbarkeit und gibt dem Nutzer die Chance, zusätzliche Components ohne Einschränkung einer Lowlevel-Sprache zu unterstützen.
Die Trennung zwischen Highlevel- und Lowlevel-Sprache steigert außerdem den Aufwand bei der Unterstützung zusätzlicher GUI-Bibliotheken, denn dazu ist die Definition einer neuen Lowlevel-Sprache nötig, die an die Spezifika der neuen GUI-Bibliothek angepasst werden muss. Zwar bleibt der HTS-Parser dabei unangetastet, jedoch müssen für die neue Sprache zusätzlich Parser und Interpreter implementiert werden. Die aufwändige und fehleranfällige Entwicklung dieser Komponenten entfällt in ATOSj, da HTS direkt geparset und interpretiert wird. Um die Erweiterung in ATOS zu komplettieren, ist das Erstellen einer Regeldatei notwendig. Diese kann sehr umfangreich sein, wie die Regeldatei für die Abbildung von HTS auf ATS, welche 2700 Zeilen umfaßt. Das Gros der Regeln beschreibt `READ`, `TEST` und `ACTION` Kommandos für jeden einzelnen HTS-Typ. Die Funktion dieser Regeln wird in ATOSj durch Wrapperklassen ersetzt, die auf den vollen Umfang von Java zurückgreifen können und somit auch die Erstellung komplexer Parameter ermöglichen, wie sie beispielsweise bei der Abfrage von Tabellenzellen nötig sind. Wrapperklassen ermöglichen außerdem die Vererbung von Funktionalität, die bei der Definition von Regeln nicht möglich ist. So muss z.B. die Regel zum Auslesen des Aktivierungszustandes (`ENABLESTATE`) für jeden Componenttyp neu definiert werden.
Das Konzept in ATOS ist historisch gewachsen. Vor der Einführung von HTS wurde ausschließlich ATS verwendet. HTS wurde entwickelt, um ATS durch eine verständlichere Sprache zu ersetzen [7]. Dabei sparte das Abbildungsverfahren Entwicklungsaufwand,

da es die Übernahme der ATS-Komponenten ermöglichte. Fragen wie eine spätere Erweiterbarkeit standen dabei nicht im Vordergrund. Für ATOSj dagegen war im Vorhinein klar, dass mindestens zwei unterschiedliche Bibliotheken unterstützt werden sollen. Unter dieser Prämisse schien die Übernahme des Konzepts aus ATOS wegen der genannten Schwächen nicht sinnvoll. Die direkte Interpretation von HTS eignet sich besser für die Ansteuerung komplexer Componenents und vereinfacht die Unterstützung weiterer GUI-Bibliotheken.

5.4 Der Plug-In Mechanismus

Um das System ATOSj beliebig skalierbar zu machen, haben wir ein Konzept entworfen, welches es ermöglicht selbstdefinierte Components mit ATOSj anzusteuern und zu testen. Dieser Mechanismus versetzt den Tester in die Lage ATOSj beliebig, entprechend der Spezifika seines Projektes, zu erweitern. Dabei ist nur die Kenntnis einiger weniger Klassen aus der ATOSj-Bibliothek nötig. Eine derartige Funktionalität erschien uns besonders wichtig, da in vielen GUI-Systemen nicht nur Standard-Components sondern auch selbstentwickelte Components für spezielle Anwendungsbereiche verwendet werden. Die selbstdefinierten Components sollen, der kürzeren Schreibweise wegen und in Ahnlehnung an die englische Literatur, im Weiteren als Custom-Components bezeichnet werden. Die Verwendung von Custom-Components kann die unterschiedlichsten Gründe haben:

- Das Programm soll sich von anderen abheben, z.B. ein bestimmtes Corporate Identity widerspiegeln.

- Die Bedienbarkeit soll verbessert werden. Das kann z.B. durch die Nutzung von Metaphern, die der Realwelt entlehnt sind, erreicht werden. Metaphern erleichtern die Übertragung der Erfahrungen und des Wissens eines Nutzers aus der Realität auf die Funktionsweise des Computersystems. Dadurch wird der Lernaufwand erheblich reduziert und dem Nutzer werden Berührungsängste mit dem unbekannten System genommen. Zu einer der bekanntesten Metaphern gehört wohl die Desktop-Metapher, die bei vielen grafischen Betriebssystemen ihre Anwendung findet. Der Desktop ist in seiner Funktionalität und teilweise auch im Aussehen einem realen Schreibtisch nachempfunden. Auf dem Desktop kann der Nutzer, wie auch auf einem realen Schreibtisch, seine Dokumente ablegen, verschieben und bearbeiten. In vielen Fällen führt eine natürliche Systemstruktur zu einer geringeren Fehlerrate, wie eine Studie von Barnard und Hammond aus dem Jahre 1983 zeigt. Ihre Untersuchung beschäftigt sich mit dem Zusammenhang zwischen der Grammatik von textuellen Kommandos und der Fehlerrate bei deren Anwendung. Das Ergebnis der Untersuchung zeigt, dass natürlichsprachliche Grammatiken einprägsamer als artifizielle Grammatiken sind und damit zu weniger Fehlern führen [16].

- Die Lesbarkeit soll verbessert werden. Statt rein textueller Ausgaben, werden häufig

grafische Darstellungen genutzt, welche besser geeignet sind optische Detektions-, Diskriminations-, Identifikations- und Zuordnungsprozesse zu untertützen. Als Beispiel kann hier die Darstellung einer Füllstandsanzeige genannt werden, wie sie häufig bei der Prozessregulation z.B. in Chemiebetrieben vorkommt. Statt einer einfachen Zahl (z.B. 63%) wird der Füllstand als abstrakter Balken dargestellt, der bis zu einer gewissen Höhe eingefärbt ist. Die Abbildung 3 zeigt diese Art von Balkendiagramm, wie sie in der Medizintechnik auf dem Monitor eines Respirators zu sehen ist. Ein Balken dient zur Anzeige des Sauerstoffgehaltes im Gasgemisch, dass für die Beatmung des Patienten produziert wird.

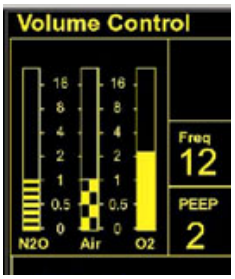


Abbildung 3: Anzeige des Respirators „Fabius“

Diese wenigen Beispiele lassen schon erahnen, dass die Anwendungsgebiete und damit auch die Arten von Custom-Components mannigfaltig sind. Sollen all diese Components durch ATOSj gesteuert werden können, muss es dafür einen universellen Mechanismus geben. Aus diesem Grund wurde die Skriptsprache HTS für die Kommandos `ACTION`, `READ` und `TEST` um das Konzept der Eigenschaften erweitert. Eine Eigenschaft entspricht einem Zustand des Custom-Components und besitzt einen entsprechenden Zustandswert. Jede Eigenschaft ist durch einen eindeutigen Namen gekennzeichnet. Der Zustandswert der Eigenschaft kann mit Hilfe der genannten HTS Kommandos gesetzt oder ermittelt und überprüft werden. Das Setzen eines Zustandswertes ist dabei definiert als die Simulation einer semantisch äquivalenten Nutzeraktion, die den aktuellen Zustand des Custom-Components in den neuen Zustand überführt. Für die Festlegung der inhaltlichen Bedeutung und des Namens einer Eigenschaft ist allein der Entwickler des Custom-Components verantwortlich. Die Benamung der Eigenschaften unterliegt dabei keinerlei Restriktionen, jedoch sollte sie möglichst plakativ sein, um das Erstellen und die Lesbarkeit der Kommandos zu vereinfachen.

Als Beispiel eines Custom-Components soll im Weiteren ein grafischer Kalender dienen, wie er auch im Programm Seminarorganisation verwendet wird. Für das Kalender-Component wurde eine Eigenschaft mit dem Namen „date“ definiert. Diese Eigenschaft steht für das vollständige Datum, das im Kalender-Component eingestellt wird. Das Setzen der Eigenschaft „date“ simuliert eine Datumseingabe des Nutzers auf dem Kalender-Component. Es können nach Bedarf beliebig viele weitere Eigenschaften definiert werden. So könnte beispielsweise die Eigenschaft „date“ weiter in Tag, Monat und Jahr unterteilt werden, um das Auslesen und Setzen der einzelnen Datumskomponenten zu ermögli-

chen. Das Einbinden von Custom-Components in ATOSj wird durch einen Tester oder Entwickler des Testobjekts vorgenommen. Dabei wird in 3 Schritten vorgegangen.

1. Erstellen einer Wrapperklasse für das Custom-Component

2. Bekanntgabe des Custom-Components an ATOSj

3. Verwendung des Custom-Components in Kommandos

Im ersten Schritt muss eine Wrapperklasse für die Klasse des Custom-Components implementiert werden. Die Wrapperklasse ist das Bindeglied zwischen ATOSj und dem Custom-Component. Sie umschließt das Custom-Component und realisiert das Setzen und Auslesen der definierten Eigenschaften. Dazu müssen die Methoden `setProperty` und `getProperty` aus dem Interface `IComponent` der Bibliothek ATOSj implementiert werden.

Die Methode `setProperty` hat die Aufgabe, Eigenschaftswerte zu setzen, d.h. eine entsprechende Nutzeraktion zu simulieren. Zu diesem Zweck werden zwei Parameter des Typs `String` an die Methode übergeben. Der erste Parameter muss den Namen der Eigenschaft enthalten, der zweite Parameter trägt den neuen Eigenschaftswert, der als Zeichenkette codiert ist. Der Programmierer hat dafür zu Sorge zu tragen, dass undefinierte Eigenschaften, ebenso wie Eigenschaftswerte, die nicht im korrekten Format vorliegen, zurückgewiesen werden. Die Definition des Formats für Eigenschaftswerte liegt in den Händen des Programmierers. Beispielsweise könnte das Kalender-Component für die Eigenschaft „date“ nur Werte akzeptieren, die in der Form dd.mm.jjjj vorliegen. Nach diesem Muster wäre die Zeichenkette „24.06.1980“ ein legitimer Datumswert, wohingegen die Zeichenkette „24.6.1980“ inkorrekt wäre.

Die Methode `getProperty` hat die Aufgabe den aktuellen Wert einer Eigenschaft auszulesen. Sie besitzt einen Parameter, welcher den Namen der auszulesenden Eigenschaft beinhaltet. Wie auch bei der Methode `setProperty` sind undefinierte Eigenschaftsnamen durch das Werfen einer Ausnahme zurückzuweisen. Die Methode `getProperty` liefert den ausgelesenen Eigenschaftswert, als Zeichenkette codiert, zurück. Die Formate der Eigenschaftswerte sollten für beide Methoden übereinstimmen. Für die Kodierung der Eigenschaftswerte wurde das Format der Zeichenkette gewählt, da dieses universell und gut lesbar ist. Es werden in HTS keine weiteren Datenformate für Eigenschaftswerte angeboten, da deren Zahl, genauso wie die Zahl der Eigenschaften, theoretisch unbegrenzt ist. Die Interpretation und etwaige Umwandlung der Zeichenketten in andere Datenformate, muss durch den Programmierer der Wrapperklasse vorgenommen werden.

Der Aufwand für die Eigenimplementation einer Wrapperklasse kann gesenkt werden, in dem von vordefinierten Klassen der Bibliothek ATOSj abgeleitet wird. Zu den beiden Bibliotheken Swing und SWT gibt es in ATOSj je zwei vordefinierte Wrapperklassen. Diese beiden Wrapperklassen umschließen die Basisklassen der Components für die je-

weilige Bibliothek. Die Wrapperklasse `SWTComponent` umschließt die Basisklasse des SWT - `Widget` und die Wrapperklasse `SwingComponent` umschließt die Basisklasse des Swing - `Component`. Alle in ATOSj implementierten Wrapperklassen für Standard-Components erben von diesen beiden Klassen. Sie ermöglichen die Simulation atomarer Nutzeraktionen durch Generierung von Elementarereignissen, deren Bedeutung im Kapitel 5.5 beschrieben wird. Dazu definieren sie die Methoden `pressKey` und `click`, welche für die Simulation eines Tastendrucks und eines Mausklicks verantwortlich sind. Jede ihrer Ableitungen kann auf diese grundlegende Funktionalität zurückgreifen.

Im zweiten Schritt müssen das Custom-Component und die dazugehörige Wrapperklasse bei ATOSj registiert werden. Dazu muss die Wrapperklasse einer Bibliotheksdatei mit dem Namen „ext.jar“ hinzugefügt werden, die dann im Verzeichnis EXT des ATOSj Projektes abgelegt wird. Diese Bibliotheksdatei wird automatisch beim Start des Testobjektes in den Klassenpfad aufgenommen. Somit wird garantiert, dass ATOSj während eines Testlaufs Zugriff auf die neu erstellten Wrapperklassen hat. Danach muss im Verzeichnis EXT eine Datei mit dem Namen „custom.lst“ angelegt werden, die Informationen zur Beziehung zwischen den Wrapperklassen und den Klassen der Custom-Components enthält, siehe Abschnitt 5.4.1.

Im dritten und letzten Schritt des Plug-In Vorganges kann das Custom-Component in den Kommandos einer Testsequenz verwendet werden. Die folgenden Kommandos demonstrieren den Umgang mit Custom-Components bei der Verwendung der Kommandos `READ`, `TEST` und `ACTION`.

Soll z.B. das Geburtsjahr bei der Ersterfassung eines Kunden gesetzt werden, muss in der Testsequenz ein Kommando ähnlich dem Folgenden definiert werden:
`ACTION, ‘‘Neu - Kunde*‘‘, SWTCALENDAR, ‘‘Geburtsdatum‘‘, PROPERTY, ‘‘year‘‘, ‘‘1980‘‘`

Soll z.B. der aktuelle Monatswert in einer Variable gespeichert werden, so könnte das Kommando dafür wie folgt aussehen:
`READ, ‘‘Neu - Kunde*‘‘, SWTCALENDAR, ‘‘Geburtsdatum‘‘, PROPERTY, ‘‘month‘‘, ‘‘varjahr‘‘`

Soll das aktuelle Datum des Kalender-Components mit einem Sollwert verglichen werden, muss ein Kommando ähnlich dem Folgenden definiert werden:
`TEST, ‘‘Neu - Kunde*‘‘, SWTCALENDAR, ‘‘Geburtsdatum‘‘, PROPERTY, ‘‘date‘‘, ‘‘24.06.1980‘‘`
Auf Grund der Beschränkung auf Zeichenketten, können Eigenschafts- und Sollwert nur auf Gleichheit geprüft werden.

5.4.1 Syntax der Datei custom.lst

Für das Einbinden von Custom-Components wird die Datei custom.lst benötigt. Der Aufbau der Datei folgt einer definierten Syntax, welche nachfolgend in EBNF vorge-

stellt werden soll. Jede Definition einer Erweiterung wird mit einem Zeilenumbruch (\n) abgeschlossen. Folgende Morpheme werden vereinbart und besonders gekennzeichnet:

`<string>` string ist eine Zeichenkette ohne Leer- und Anführungszeichen.

lst = {definition \n}
definition = `COMPONENT, <WRAPPERKLASSE>, <CUSTOMKLASSE>, <TYPENAME>`

`<WRAPPERKLASSE>:` Ist der vollständig qualifizierte Name der Wrapperklasse für die Klasse des Custom-Components.

`<CUSTOMKLASSE>:` Ist der vollständig qualifizierte Name der Klasse des Custom-Components, diese muss von der Basisklasse der GUI-Bibliothek abgeleitet sein.

`<TYPENAME>:` Gibt den Typnamen für das Custom-Component an, dieser kann in HTS Kommandos verwendet werden. Der Typname ist eine Abstraktion, welche innerhalb der verwendeten GUI-Bibliothek eindeutig sein muss.

Beispiel:
`COMPONENT, CalendarControlWrapper, semorg.gui.util.CalendarControl, SWTCALENDAR`

Zu der von `org.eclipse.swt.Composite` abgeleiteten Klasse `semorg.gui.util.CalendarControl` gibt es eine Wrapperklasse namens `CalendarControlWrapper`. Der in HTS verwendete Name für diesen Typ von Component soll `SWTCALENDAR` sein.

5.5 Simulation von Nutzeraktionen

Die Kommunikation zwischen dem Betriebssystem und einer grafischen Anwendung erfolgt durch Nachrichten. Das Betriebssystem verschickt Nachrichten an eine Anwendung, um diese über bestimmte Ereignisse zu informieren. Diese Ereignisse werden durch bestimmte Nutzeraktionen ausgelöst. Zu den auslösenden Nutzeraktionen gehören beispielsweise Mausklicks oder Tastatureingaben. Diese Aktionen lösen, abhängig vom bedienten Component, unterschiedliche Ereignisse aus. Zum Beispiel führt das Klicken mit der linken Maustaste auf eine Zeile einer Tabelle zur Markierung dieser Zeile und das Programm erhält darüber eine entsprechende Nachricht. Diese Nachricht enthält ereignisspezifische Parameter, wie z. B. die Indizes der markierten Zeilen. Dem Programmierer der Anwendung fällt die Aufgabe zu, solche Ereignisse auszuwerten.

Bei der Programmierung von Java-Anwendungen wird nicht mehr von Nachrichten sondern nur noch von Ereignissen (Events) gesprochen. Nachrichten sind betriebssystemspezifisch wohingegen Events eine semantische Abstraktion bilden, welche plattformunabhängig ist. Unterschiedliche Arten von Ereignissen werden durch unterschiedliche

Java-Klassen dargestellt. Diese Klassen wiederum unterscheiden sich je nach verwendeter GUI-Bibliothek. Zu jeder Art von Ereignis gibt es einen entsprechenden Ereignisempfängertyp, repräsentiert durch ein Java-Interface. Der Ereignisempfänger (Eventlistener) hat die Aufgabe auf das Auftreten eines Ereignisses zu reagieren und muss sich zu diesem Zweck zuvor bei der Ereignisquelle registrieren. Tritt ein Ereignis auf, so werden die passenden Ereignisempfänger durch die Ereignisquelle darüber benachrichtigt. Allgemein sind Ereignisquellen alle Objekte, die andere Objekte über Zustandsänderungen informieren. Typische Ereignisquellen bei grafischen Anwendungen sind deren Components, wie z.B. ein Textfeld, das über die Eingabe von Text informiert. Die Tabellen 6 und 7 zeigen auszugsweise für die Bibliotheken Swing und SWT je einen Ereignisquellentyp, einige der möglichen Ereignistypen sowie die dazugehörigen Methoden der Ereignisempfängerschnittstellen und deren Bedeutung.

Ereignisquelle	**Ereignisklasse**	**Ereignismethode**	**Bedeutung**
`JTextField`	`MouseEvent`	`mouseClicked`	Eine Maustaste wurde gedrückt und wieder losgelassen.
		`mouseEntered`	Der Mauszeiger betritt das Component.
		`mouseExited`	Der Mauszeiger verläßt das Component.
		`mousePressed`	Eine Maustaste wurde gedrückt.
		`mouseReleased`	Eine Maustaste wurde losgelassen.
	`KeyEvent`	`keyPressed`	Eine Taste wurde gedrückt.
		`keyReleased`	Eine Taste wurde losgelassen.
		`keyTyped`	Eine Taste wurde gedrückt und wieder losgelassen.
	`ActionEvent`	`actionPerformed`	Die Eingabe wurde mit Enter bestätigt.

Tabelle 6: Beispiel für Ereignisse in Swing

Ereignisquelle	**Ereignisklasse**	**Ereignismethode**	**Bedeutung**
`Text`	`MouseEvent`	`mouseDown`	Eine Maustaste wurde gedrückt.
		`mouseUp`	Eine Maustaste wurde losgelassen.
		`mouseDoubleClick`	Eine Maustaste wurde zwei Mal gedrückt.
	`KeyEvent`	`keyPressed`	Eine Taste wurde gedrückt.
		`keyReleased`	Eine Taste wurde losgelassen.
	`ModifyEvent`	`modifyText`	Der Text wurde geändert.

Tabelle 7: Beispiel für Ereignisse in SWT

Ereignisse können grundsätzlich in zwei Kategorien unterteilt werden. Die erste Kategorie sei Elementarereignis genannt. Elementarereignisse bilden eine direkte Entsprechung

zu Nutzeraktionen, ohne weitere Interpretation. Dazu gehören alle Arten von Maus- und Tastatureingaben. Elementarereignisse lösen Ereignisse der zweiten Kategorie aus, welche semantische Ereignisse genannt werden sollen. Elementarereignisse und semantische Ereignisse stehen in einer 1 zu N Beziehung zueinander. Ein semantisches Ereignis ordnet einem Elementareignis, abhängig von der Ausprägung und dem Zustand der Ereignisquelle, eine Bedeutung zu. Das im ersten Absatz erwähnte Beispiel der Selektion einer Tabellenzeile verdeutlicht die Beziehung zwischen Elementarereignis und semantischem Ereignis. Der Mausklick des Nutzers ist das Elementarereignis. Es löst in seiner Folge ein semantisches Ereignis aus, das Informationen über die ausgewählten Tabellenzeilen beinhaltet. Die in den Tabellen 6 und 7 aufgeführten Klassen `MouseEvent` und `KeyEvent` stehen für Elementarereignisse, die Klassen `ActionEvent` und `ModifyEvent` beschreiben semantische Ereignisse.

In der Implementation der Bibliothek AWT werden alle Ereignisse in einer Ereigniskette (Eventqueue) gesammelt. Tritt ein neues Ereignis auf, wird es am Ende der Ereigniskette angefügt. Alle Ereignisse werden dann in Einfügereihenfolge abgearbeitet. Während der Abarbeitung eines Ereignisses können weitere Ereignisse, beispielsweise durch die Auslösung von Hardware-Interrupts, auftreten, welche dann ebenfalls an die Ereigniskette angehangen werden.

Die Verwaltung von GUI Ereignissen in Java muss zwei getrennte Welten miteinander verbinden. Auf der einen Seite die Welt des Betriebssystem und auf der anderen Seite die Java-Welt. Zu diesem Zweck wurde bei der Entwicklung der Bibliothek AWT eine 3-Schichten-Architektur entworfen, siehe Abbildung 4. Die oberste Schicht bilden die Component-Klassen der Bibliothek, welche die Klasse `java.awt.Component` als gemeinsame Basisklasse haben. Auf Objekte dieser Klassen hat der Java-Programmierer Zugriff und kann sie erzeugen und manipulieren. Die zweite Schicht bilden die so genannten Peers. Sie dienen als Mittler zwischen den Java-Components und den nativen Components, welche die unterste Schicht der Architektur bilden. Zu jeder Component-Klasse existiert ein korrespondierendes Peer-Interface. Die konkrete Implementation eines Peers ist betriebssystemspezifisch und realisiert den Zugriff auf ein bestimmtes natives Component. Für die Implementation eines Peers wird das Java Native Interface genutzt. Jedes Java-Component besitzt einen Peer an den es Funktionsaufrufe delegiert. So existiert für die Klasse `Button` das entsprechende Peer-Interface `ButtonPeer`, dessen konkrete Implementation für Windows in der Klasse `WButtonPeer` realisiert ist. Diese strikte Trennung kapselt betriebssystemspezifische Funktionalität in den Peers und macht damit die Component-Klassen völlig plattformunabhängig. Ein Peer erlaubt nicht nur den Zugriff auf ein natives Component sondern überwacht auch die Nachrichten des Betriebssystems, wandelt diese in Java-Events um und fügt sie der Java-Eventqueue hinzu. Die Abarbeitung eines Events wird von dem Java-Component durchgeführt, dessen Peer das Event in die Eventqueue eingetragen hat. Im Verlauf der Abarbeitung benachrichtigt das Java-Component seine Eventlistener.

Die von ATOSj unterstützte Bibliothek Swing erweitert das AWT um so genannte leicht-

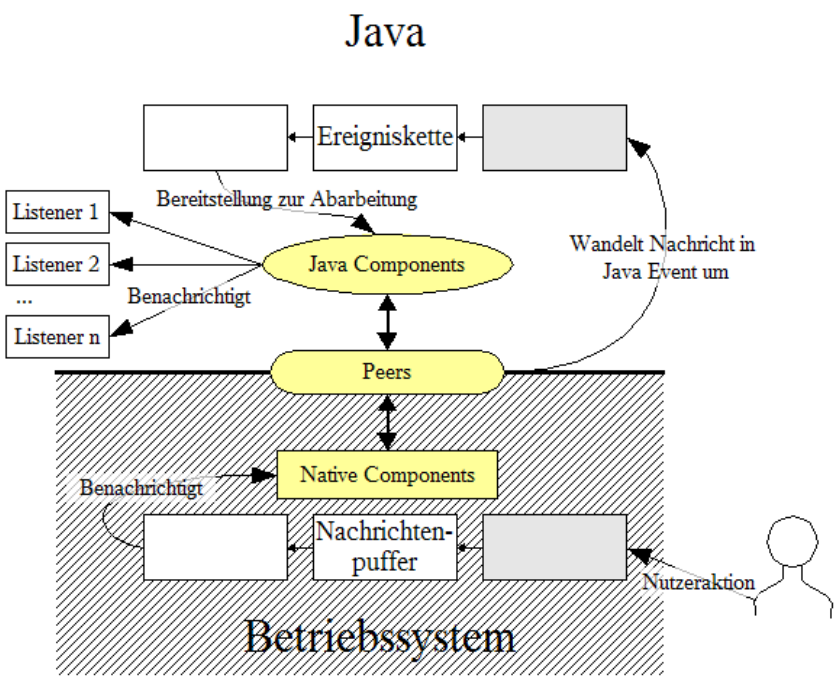


Abbildung 4: AWT 3-Schichten-Architektur

gewichtige Components. Diese haben, im Gegensatz zu den AWT Components, keine direkte native Entsprechung. Sie besitzen dennoch alle einen Peer auf ein natives Component, da alle Component-Klassen des Swing von `java.awt.Component` abgeleitet sind. Das zugeordnete native Component dient jedoch nur als Leinwand, um darauf das tatsächliche Erscheinungsbild eines Swing-Components zu zeichnen. Hierzu werden lediglich grafische Primitiven des Betriebssystems genutzt. Dem Betriebssystem ist die eigentliche Funktion des Swing Components verborgen. Es hat keine Information darüber, ob es sich um eine Tabelle, Liste oder ein anderes grafisches Element handelt. Aus diesem Grund werden durch das native Component respektive durch seinen Peer auch nur Elementarereignisse generiert. Das Swing-Component wird zu einer zusätzlichen Ereignisquelle und erzeugt zu den Elementarereignissen semantische Ereignisse entsprechend seiner Funktion.

Die zweite von ATOSj unterstützte Bibliothek SWT verzichtet auf die strikte Trennung zum Betriebssystem, wie sie beim Design des AWT postuliert wird. In der Architektur des SWT entfällt die Mittlerschicht zwischen Java- und nativen Components. Der Zugriff auf die zugrunde liegenden nativen Components erfolgt ohne den Umweg über einen Peer durch das SWT-Component selbst. Diese Designentscheidung hat zur Folge, dass die Component-Klassen des SWT, abhängig vom Betriebssystem, völlig unterschiedlich implementiert sind. Dennoch definieren je zwei Klassen desselben Components für unterschiedliche Betriebssysteme dieselben Methoden mit derselben funktionalen Spezifikation. Damit ist die Portabilität eines SWT-Programmes, trotz unterschiedlicher Im-

plementation der SWT-Components, für alle unterstützten Betriebssysteme garantiert. Des Weiteren gibt es zur Verwaltung von Ereignissen in SWT keine eigene Ereigniskette, wie im AWT. Stattdessen werden Nachrichten direkt aus dem Puffer des Betriebssystems gelesen, in Java-Events umgewandelt und an die Ereignisempfänger weitergeleitet.

Ein Aktionsschritt in ATOSj, repräsentiert durch das Kommando `ACTION`, soll möglichst genau eine komplexe Nutzeraktion simulieren. Eine Nutzeraktion wird als vollständig simuliert betrachtet, wenn durch die Simulation die gleichen Ereignisse in der gleichen Reihenfolge generiert werden wie bei der realen Aktion und wenn das Component nach der Simulation denselben Zustand einnimmt wie nach der realen Aktion. Es gibt zwei verschiedene Ansätze zur Simulation von Nutzeraktionen, welche auf die Erfüllung der vorgenannten Voraussetzungen überprüft werden sollen:

1. Der programmatische Ansatz

2. Der ereignisbasierte Ansatz

5.5.1 Der programmatische Ansatz

Der programmatische Ansatz basiert auf der Manipulation der Java Components über den direkten Aufruf von Methoden der jeweiligen Klasse. Als Beispiel soll das folgende `ACTION`-Kommando dienen, welches die Eingabe von Text in ein Textfeld simulieren soll. Das Textfeld sei für die Eingabe von Stückzahlen konzipiert. Ein Ereignisempfänger überwacht alle Tastatureingaben und läßt nur Ziffern zu. Die korrekte Funktionsweise des Textfeldes soll durch das anschließende `TEST`-Kommando verifiziert werden.

```
ACTION, ‘‘Bestellung‘‘, EDITBOX, ‘‘stueck‘‘, EDIT, ‘‘1000a‘‘
TEST, ‘‘Bestellung‘‘, EDITBOX, ‘‘stueck‘‘, TEXT, ‘‘1000‘‘
```

Die Klassen für Textfelder in Swing und in SWT deklarieren eine Methode `setText(String)`, mit deren Hilfe die Zeichenkette „1000a“ in das Textfeld eingetragen werden kann. Die Verwendung der Methode `setText` macht ein schwerwiegendes Problem des programmatischen Ansatzes deutlich. Der Aufruf der Methode setzt zwar den Text des Textfeldes, generiert jedoch keine Ereignisse. Damit bleibt auch die Reaktion der Ereignisempfänger aus und der Programmablauf entspricht nicht mehr der Intention des Testers. Das `TEST`-Kommando schlägt fehl, unabhängig von der Korrektheit der Implementation des Testobjekts. Der programmatische Ansatz ist für den allgemeinen Fall nicht praktikabel.

5.5.2 Der ereignisbasierte Ansatz

Der ereignisbasierte Ansatz spaltet eine komplexe Nutzeraktion in eine Folge atomarer Aktionen auf. Zu jeder atomaren Aktion wird ein Elementarereignis, entsprechend der zugrunde liegenden GUI Bibliothek, gebildet. Die generierten Elementarereignisse werden dann zur Verarbeitung an die Java-Components weitergeleitet. Als Beispiel für die Dekomposition eines Aktionsschrittes in atomare Nutzeraktionen soll das `ACTION` Kommando aus dem vorigen Abschnitt dienen, das Ergebnis sieht wie folgt aus:

1. Falls das Textfeld nicht den Eingabefokus hat, Anklicken des Textfeldes, damit es den Fokus erhält.

2. Vollständige Auswahl des bereits enthaltenen Textes, um diesen zu überschreiben. Die Auswahl kann beispielsweise durch Drücken der Tastenkombination Strg + A erreicht werden.

3. Eingabe der einzelnen Zeichen des Textes „1001a“.

Bei der Verwendung von Swing werden Elementarereignisse vom Testsystem selbst generiert und der Java-Eventqueue hinzugefügt. ATOSj übernimmt dabei die Rolle des Peers, welcher Nachrichten des Betriebssystems über reale Nutzeraktionen in Java Ereignisse umwandelt, siehe Abbildung 4. Statt Nachrichten des Betriebssystems werden Anweisungen eines `ACTION`-Kommandos in Elementarereignisse umgewandelt, die in Folge wie reale Ereignisse verarbeitet werden und somit eine perfekte Simulation liefern.

Elementarereignissen müssen im Rahmen des SWT anders an die Components weitergeleitet werden, da keine zusätzliche Ereigniskette auf der Java-Seite existiert. Die generierten Java-Ereignisse werden, über eine Schnittstelle der Bibliothek, in Nachrichten umgewandelt und an das Betriebssystem weitergeleitet. Es verteilt dann die Nachrichten an die nativen Components, die wiederum von den SWT Components überwacht werden. Die SWT Components übernehmen die Verarbeitung der Nachrichten und deren Umwandlung in Ereignisse. Folglich scheint auch hier eine perfekte Simulation gelungen, was aber ein Trugschluss ist. Der ereignisbasierte Ansatz ist für das SWT nicht ausreichend. Grund hierfür ist die Betriebssystemabhängigkeit der SWT Components, die zu einer betriebssystemabhängigen Dekomposition der Aktionsschritte führt. Das wird bei der Betrachtung von Schritt 2 des Beispiels deutlich. Hier soll eine atomare Aktion die vollständige Selektion des Textes im Eingabefeld bewirken. In vielen Betriebssystemen ist dafür die Tastenkombination Strg + A vorgesehen, diese gilt jedoch nicht zwingend für die Menge aller Betriebssysteme. Eine vom Betriebssystem abhängige Dekomposition von Aktionsschritten kann jedoch nicht Ziel der Implementation von ATOSj sein, da dies zu einem großen Mehraufwand führen würde. Deshalb wurde für SWT ein hybrider Ansatz verwendet, welcher den programmatischen und den ereignisbasierten Ansatz vereint. Eine einfache Regel gibt vor, wann welches Verfahren zum Einsatz kommt. Ist eine

atomare Aktion betriebssystemabhängig, wird der programmatische Ansatz genutzt, in allen anderen Situationen wird nach dem ereignisbasierten Ansatz verfahren. Für den praktischen Einsatz wird das programmatische Vorgehen leicht erweitert, d.h. fehlende semantische Ereignisse werden ergänzt und mit Hilfe der Methode `notifyListeners` an die Ereignisempfänger weitergeleitet.

5.6 Javassist

Die Bibliothek Javassist ist ein wichtiger Bestandteil der Implementation von ATOSj. Sie wird sowohl für das Modul ACover, siehe Kapitel 5.11, als auch für die Anbindung von ATOSj an SWT genutzt. Die interessanten Eigenschaften und die Funktionalität von Javassist, sowie dessen Philosophie im Rahmen der aspektorientierten Programmierung, sollen in diesem Abschnitt beleuchtet werden.

Javassist erweitert die Möglichkeiten der Reflexion (engl. reflection) aus der Java API. Reflexion in Java beinhaltet die Introspektion von Klassen- oder Interfacetypen, das heißt es können zur Laufzeit Informationen über die interne Struktur eines Typs gewonnen werden. Durch Reflexion kann beispielsweise eine Liste der definierten Methoden einer Klasse mit ihren Eigenschaften, wie dem Namen der Methode oder deren Rückgabetyp, ermittelt werden. Die Reflexion ermöglicht jedoch nicht nur das Sammeln von Informationen über Klassen sondern auch die direkte Manipulation von Klasseninstanzen. Die folgenden manipulativen Mittel stehen dem Nutzer des Java Reflection API zur Verfügung:

- Erzeugen einer Klasseninstanz, auch wenn der Klassenname erst zur Laufzeit bekannt ist.
- Ermitteln und Setzen des Wertes eines Datenfeldes in einem Objekt, auch wenn der Name des Feldes erst zur Laufzeit bekannt ist.
- Methodenaufruf an einem Objekt, auch wenn die Methode erst zur Laufzeit bekannt ist.
- Erzeugen eines neuen Feldes (engl. array), auch wenn dessen Größe und enthaltener Typ erst zur Laufzeit bekannt sind.
- Ändern der Elemente eines Feldes.

Diese Art von Laufzeitmanipulation, findet ihre Anwendung häufig in Netzwerkprogrammen, die Klassen, deren Namen zur Compilezeit noch nicht bekannt sind, dynamisch über das Netzwerk laden. Um diese Klassen auch verwenden zu können, muss auf die Mittel der Reflexion zurückgegriffen werden.

In einigen Fällen reichen diese Mittel jedoch nicht aus. An dieser Stelle setzt Javassist an und bereichert die Reflexion um Möglichkeiten der strukturellen Änderung von Klassendefinitionen. Die Liste der bereitgestellten Möglichkeiten zur Strukturänderung ist lang, deshalb sollen einige wenige Beispiele genügen, um die Mächtigkeit von Javassist zu demonstrieren:

- Hinzufügen neuer Methoden zu einer Klasse.
- Änderung der Zugriffsrechte einer Methode.
- Hinzufügen von Anweisungen in eine Methodendefinition.

Javassist realisiert diese Funktionen durch die Änderung der Klassendefinition auf Binärebene. Eine Klassendefinition im Binärformat ist in einer plattformunabhängigen Zwischensprache verfaßt. Sie wird vom Java Compiler bei der Übersetzung des Quelltextes erstellt. Diese Zwischensprache wird Bytecode genannt. Der Bytecode ist der Befehlssatz der virtuellen Java-Maschine, welcher aus aus 1-Byte langen Befehlen besteht. Die virtuelle Java-Maschine ist die Abstraktion eines realen Prozessors. Ihr obliegt es den Bytecode eines Java-Programmes zu interpretieren, d.h. in Befehle eines realen Prozessors umzuwandeln und diese auszuführen. Aufgrund dieses Konzeptes ist ein Java-Programm ohne Änderung auf allen Rechnern lauffähig, für die eine virtuelle Java-Maschine getreu der Spezifikation implementiert werden kann [6]. Eine Klassendefinition muss dem class-Dateiformat genügen. Dieses Format legt die Struktur der Binärdarstellung einer Klasse fest, wie z.B. die Reihenfolge der Bytes in Multibyte-Daten (big-endian - höherwertige Bytes zuerst) oder die Anordnung und Codierung von Informationen über enthaltene Felder, Methoden usw.

Das Verständnis der Funktionsweise von Javassist setzt einige Kenntnisse über den Lebenszyklus einer Klasse bei der Ausführung eines Java-Programmes voraus, welche im Folgenden kurz dargelegt werden. Soll eine Klasse verwendet werden, muss sie zuvor geladen werden. Laden wird das Auffinden der Binärdarstellung einer Klasse zur Laufzeit genannt. Im Normalfall bedeutet das die Suche nach einer class-Datei im Dateisystem oder im Netzwerk. Für das Laden ist der so genannte Klassenlader (engl. class loader) verantwortlich, welcher abgesehen vom Bootstrap Klassenlader selbst eine Java-Klasse darstellt. Es kann gleichzeitig mehrere verschiedene Klassenlader geben, welche hierarchisch angeordnet sind. Das Laden einer Klasse wird zunächst rekursiv an den übergeordneten Klassenlader delegiert, kann dieser die Klassedefinition nicht finden, wird diese Aufgabe direkt vom aktuellen Klassenlader übernommen. Hat der Klassenlader die Klassendefinition gefunden, so leitet er diese als einen Strom von Bytes an die virtuelle Java-Maschine weiter. Der Klassenlader darf zuvor Änderungen an der Klassendefinition vornehmen. Ein Beispiel für eine solche Änderung wäre das Entschlüsseln einer verschlüsselten Klassendefinition. Mit der Übergabe der Klassendefinition an die virtuelle Maschine ist der Ladevorgang beendet.

Nach dem Laden folgt das Binden. Beim Binden wird die geladene Klasse zuerst auf strukurelle Integrität überprüft, d.h. ob sie der Spezifikation des class-Dateiformats entspricht. Dieser Vorgang wird Verifikation genannt. Schlägt die Verifikation fehl, führt dies zum Abbruch des Programmes. Auf die Verifikation folgen die Vorbereitung und die Auflösung und beschließen somit das Binden. Auf diese beiden Vorgänge soll nicht näher eingegangen werden, da sie für das Verständnis von Javassist nicht von Bedeutung sind.

Javassist ermöglicht die Änderung einer Klassendefinition zur Lauf- oder Compilezeit. Der Quelltext muss dazu nicht vorliegen. Über die von Javassist bereitgestellte Schnittstelle kann der Programmierer die Definition einer Klasse ändern. Dazu ist keine Kenntnis des class-Dateiformats oder des Java-Bytecodes erforderlich, denn Javassist bietet die Möglichkeit der Manipulation auf Quelltextebene. Dieses Vorgehen soll kurz mit dem nachfolgenden Beispiel verdeutlicht werden:

```
CtClass clazz = ClassPool.getDefault().get("java.util.Vector");
CtMethod.make(
        "public String getStr(int index)" +
        "{" +
                "return (String) get(index);" +
        "}",
        clazz);
```

Das Beispiel zeigt das Hinzufügen der Methode `getStr(int)` zur Klasse `java.util.Vector`. Hierzu ist lediglich die Definition der Methode in korrekter Java-Syntax nötig. Ein speziell für die Bibliothek Javassist entwickelter Compiler, überprüft die Gültigkeit des angegebenen Quelltextes und transformiert diesen in Java-Bytecode, welcher dann der Klassendefinition hinzugefügt wird. Ist der angegebene Quelltext korrekt, garantiert Javassist den Erfolg der Verifikation der geänderten Klasse durch die virtuelle Java-Maschine. Die Änderung einer Klassendefinition zur Laufzeit, erfolgt in Javassist über einen speziell definierten Klassenlader L (siehe Abbildung 5). L übernimmt direkt das Laden einer Klasse X und delegiert diesen Vorgang nicht, wie sonst üblich, an den übergeordneten Klassenlader. Somit kann L die Definition von X, vor der Bekanntgabe an die virtuellen Java-Maschine, ändern. Nach der Änderung entsteht eine virtuelle Klasse C, welche nur zur Laufzeit existiert.

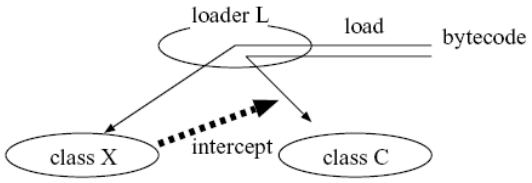


Abbildung 5: Laden einer Klasse in Javassist [13]

Mit der Bibliothek Javassist wurde die Basis für die Implementation neuer Technologien geschaffen. Insbesondere sei in diesem Rahmen die aspektorientierte Programmierung genannt. Die aspektorientierte Programmierung hat zum Ziel bestimmte Anforderungen, die an ein System gestellt werden, besser zu modellieren als es die bestehenden Programmierparadigmen vermögen. Zu solchen Anforderungen gehören die so genannten *cross-cutting concerns*. Sie sind dadurch gekennzeichnet, dass sie sich nicht von den anderen Anforderungen separieren lassen und somit in vielen Komponenten zu finden sind, aber nicht zu deren eigentlicher Funktionalität zählen. Diese Art von Anforderungen zerstört die Ziele der Modularisierung - Überschaubarkeit, Verständlichkeit und Wartbarkeit. Ein cross-cutting concern schneidet bildlich durch andere Anforderungen. Die Protokollierung ist ein häufig genanntes Beispiel für einen cross-cutting concern. Hierbei wird die Ausführung bestimmter Funktionen im Programmverlauf aufgezeichnet, wie z.B. das Anmelden und Abmelden eines Nutzers. Dazu ist es notwendig in jeder Funktion des Programms, die protokolliert werden soll, einen zusätzlichen Aufruf einzufügen. Das Protokollieren ist also verstreut über viele Komponenten, gehört aber nicht zu deren Funktionalität.

Um eine Unterstützung für diese Art von Problemen zu liefern, wurde das Konzept der aspektorientierten Programmierung (AOP) entwickelt, welches erstmals 1997 von Gregor Kiczales auf der ECOOP (European Conference on Object-Oriented Programming) vorgestellt wurde [12]. „Die aspektorientierte Programmierung löst diese Probleme in dem sie einen Mechanismus bereitstellt, der zusätzliches Verhalten von außerhalb einer Klasse in die Klasse selbst einfügt“ und somit die Modularisierung wieder herstellt [14, S. 17 Übersetzung durch den Autor]. Durch eine spezielle Beschreibungssprache soll die Implementation von Aspekten von der Implementation der Basiskomponenten getrennt werden. Dies setzt nicht zwingend eine neue Programmiersprache voraus sondern erfordert lediglich die Erweiterung einer bestehenden Sprache. Die bestehende Sprache wird genutzt, um die Basiskomponenten (component language) zu definieren und die Erweiterung ist für die Beschreibung der Aspekte zuständig (aspect language). Basiskomponenten und Aspekte werden dann zur Compile- oder zur Laufzeit miteinander „verwoben“ [12], d.h. der Code zur Implementation der Aspekte wird an wohldefinierten Punkten in den Code der Basiskomponenten eingefügt. Die Punkte, welche dafür potentiell in Frage kommen, werden Erweiterungspunkte oder in der AOP-Sprache *join points* genannt. Sie markieren wichtige Stellen im Programmfluss, wie den Aufruf einer Methode oder das Werfen einer Ausnahme. Werden join points tatsächlich verwendet, so wird das als *point cut* bezeichnet, d.h. es wird aus der Menge aller join points eine Teilmenge zur Manipulation ausgewählt.

Javassist bietet die Möglichkeit, direkt Code an Erweiterungspunkten einzufügen. Dazu werden semantische Methoden angeboten, wie z.B. die Methode `insertAfter` aus der Klasse `javassist.CtMethod`, welche neuen Code am Austrittspunkt einer Methode einfügt. Diese Funktionalität haben wir uns bei der Realisierung der Anbindung von ATOSj an SWT zu Nutze gemacht. Die Aufgabe bestand darin, selbstdefinierte Listener an alle Instanzen der Components des Testobjekts anzuhängen. Eine zusätzliche Bedingung war, dass diese Listener vorderhand über auftretende Ereignisse informiert werden, sie-

he Kapitel 5.10. Diese Anforderung bedingt das Einfügen zusätzlicher Funktionalität in bestehende Basiskomponenten, den Klassen der SWT-Bibliothek, und kann somit als ein cross-cutting concern im Sinne des AOP identifiziert werden. Der Austrittspunkt des Konstruktors der jeweiligen SWT-Klasse stellt den join point für die neue Anforderung dar. Diese Vorgehensweise zeigt starke Parallelen zu den Konzepten des AOP, ist aber dennoch keine vollständige Umsetzung des Paradigmas. Insbesondere fehlt die Verwendung einer aspektorientierten Sprache, denn das Einfügen der neuen Anforderung erfolgt manuell über die Schnittstellen von Javassist. Dieses Verfahren liegt in der Notwendigkeit begründet, Klassen von Drittanbietern zu manipulieren, deren Quelltext nicht vorliegt. Sind jedoch die Quellen verfügbar, kann direkt eine aspektorientierte Sprache verwendet werden. An dieser Stelle sei die Java-Syntaxerweiterung JBossAOP genannt. Mit dieser Erweiterung können point cuts durch Anmerkungen (engl. annotations) im Quelltext definiert und dazu entsprechende Aspekte implementiert werden [22]. Im Hintergrund wird zur technischen Verwirklichung von JBossAOP die Bibliothek Javassist verwendet.

5.7 Ausführung des Testobjekts

Bei der Entwicklung eines Programms für den automatisierten Regressionstest stellt die Art und Weise der Interaktion mit dem Testobjekt eine wichtiges Merkmal des Systementwurfs dar.

Das Vorgängersystem ATOS nutzte die wohl einzige Möglichkeit, die für rein nativ programmierte Oberflächen unter Windows existiert: Die Kommunikation über Windows-Nachrichten. Bei dieser Kommunikationsvariante wird der Nachrichtenpuffer eines jeden zum Testobjekt dazugehörigen Fensters überwacht. Soll eine Aktion an einem Element erfolgen, so wird in den Nachrichtenpuffer des betroffenen Fensters eine entsprechende Nachricht abgelegt.

Für ATOSj ist dieser Ansatz aufgrund der Multiplattformfähigkeit der genutzten Architektur aus zwei Gründen weniger geeignet. Zum einen wäre er mit einem unnötig hohen Aufwand verbunden, da für jede von Java unterstützte Plattform eine eigene Implementation erfolgen müsste. Würde diese Implemtentation dann nur für ein Betriebssystem erfolgen, so wäre jeder Tester dazu gezwungen, auf die von ATOSj unterstützte Plattform zu wechseln, was für ein Java-System höchst unangebracht wäre. Zum Zweiten wäre aber selbst in diesem Fall noch immer nicht sichergestellt, dass dieser Implementationsansatz überhaupt zum Ziel führen würde. So nutzt beispielsweise die Oberflächenbibliothek Swing nur teilweise die vom Betriebssystem zur Verfügung gestellten GUI-Funktionalitäten. Daher ist auch fraglich, inwieweit die zu überwachenden Kommunikationspuffer überhaupt genutzt werden, und ob somit eine Manipulation wie bei einer nativen Implementation überhaupt möglich wäre.

Diese beiden Problemfelder verdeutlichen also, dass dieser Ansatz für ein Java-Testsystem

nicht geeignet ist. Allerdings muss dies noch lange nicht bedeuten, dass eine solche Funktionalität für Java nicht zu erreichen ist. Dies liegt darin begründet, dass Java-Programme nicht als Maschinencode, sondern in einer Zwischensprache, dem Java-Bytecode, vorliegen. Bestandteil von Java ist u.a. eine API zur Nutzung von Funktionalitäten an Klassen, von denen lediglich der Bytecode vorhanden ist. Mittels diesem „Reflections“ genannten Abschnitt der Java-API ist es möglich, public Methoden und Member einer beliebigen Klasse allein über Ihren Namen zu referenzieren, aufzurufen und zu manipulieren.

Mit diesem Mittel ist es möglich, Überwachungsfunktionalitäten in demselben Prozess, in dem auch das Testobjekt läuft, zu platzieren. Statt des eigentlichen Aufrufs der main-Methode des Testobjekts wird dazu ein ATOSj Überwachungsprozess gestartet, welcher dann mittels Reflection die eigentliche main-Methode startet. Dieser Aufruf erfolgt in ATOSj in der inneren Klasse MainRunner in atosj.util.GUIRunner:

```
try {
    private Method main;

    Class mainClass = getClass().getClassLoader().loadClass(mainClassName);
    main = mainClass.getMethod("main", new Class[]{String[].class});

    main.invoke( null, new Object[] {args} );
} catch(Exception e) {...}
```

Listing 3: Aufruf der main-Funktion des Testobjekts

Zunächst wird die vom Nutzer angegebene main-Klasse vom ClassLoader geladen. Danach wird die Methode „main“, die als Parameter ein String[] erfordert, referenziert. Sind beide Operationen erfolgreich, so erfolgt schließlich der Aufruf mittels invoke(). Der erste Parameter von invoke() ist null, da die ausgewählte Methode static sein soll, der zweite Parameter enthält die vom Nutzer übergebenen Programmargumente.

Dank dieses Mechanismus ist es somit möglich das Testobjekt in einem beliebigen Kontext auszuführen. Via RMI können nun zwischen ATOSj und dem Testobjekt Nachrichten ausgetauscht werden.

5.8 Interprozesskommunikation mit dem Testobjekt mittels RMI

5.8.1 über RMI

Eine wichtige Eigenschaft von ATOSj ist die Methodik der Kommunikation mit seinen Testobjekten. In diesem Abschnitt wird die finale Lösung für dieses Problem erläutert. Außerdem wird beschrieben, warum dieser Systementwurf gewählt wurde.

Damit ein Testsystem wie ATOSj mit einem Testobjekt interagieren kann, muss eine Möglichkeit gefunden werden, die Überwachungs- und Manipulationsfunktionen der Testumgebung mit dem Prozess des zu testenden Oberflächenprogramms in Verbindung zu bringen. Die in ATOSj umgesetzte Lösung besteht darin, das Testobjekt mit den Manipulationsfunktionen zu „ummanteln“, und es anschließend in einem separaten Prozess zu starten. Wie dies genau geschieht, ist in Kapitel 5.7 detailliert beschrieben.

Durch diese Architektur bedingt ergibt sich eine Problemstellung, wie sie im Vorgänger ATOS bereits ähnlich vorhanden war. Es handelt sich hierbei um die Fragestellung, wie zwei unabhängige Prozesse miteinander in Kontakt treten können. Dieser Vorgang ist allgemein unter dem Namen Interprozesskommunikation (IPC) bekannt.

In ATOS ist das Problem mittels Nachrichtenpuffern, die jedem Windows-Fenster automatisch vom Betriebssystem bereitgestellt werden, gelöst. Auf diese Puffer kann von allen Prozessen sowohl lesend als auch schreibend zugegriffen werden.

Da allerdings jedes Betriebssystem sein eigenes System der Nachrichtenpuffer hat, ist dies für ATOSj aufgrund der projektierten Multiplattformfähgkeit keine Alternative. Ebenfalls wären diese nicht ohne eigene Java-Native Erweiterungen erreichbar. Es musste daher eine Möglichkeit gefunden werden, eigene IPC-Interfaces zu erzeugen.

Im engeren Sinne bezeichnet der Begriff Interprozesskommunikation zwar lediglich die Kommunikation zweier Prozesse auf demselben Computer, allerdings wird dieser Begriff des Öfteren auch allgemeiner gebraucht. Häufig spricht man auch von IPC beim Datenaustausch in Verteilten Systemen, sowohl bei Threads eines Laufzeitsystems, als auch bei Programmen, die auf unterschiedlichen Rechnern laufen.

Als Programmiersprache, die häufig in verteilten Web-Applikationen verwendet wird, muss auch Java die Möglichkeit bieten, eine solche Kommunikation herzustellen. Prinzipiell gibt es in Java zwei Ansätze für eine praktische Umsetzung. Zum einen hat man als Entwickler unter Java stets die Möglichkeit, auf einer niedrigen Abstraktionsebene eine IPC via Betriebssystemsockets zu entwickeln. Als Socket wird ein Endpunkt einer Kommunikationsverbindung zwischen zwei Teilnehmern bezeichnet. Über einen solchen Socket können lediglich einfache Nachrichtenstreams versendet werden. Die Serialisierung, also die sequenzielle Abbildung von Objekten auf eine flache und einfache Darstellungsform, muss daher zuvor manuell erfolgen. Nach dem Versand eines solchen serialisierten Datenstreams müsste dieser anschließend beim Empfänger wieder in seine Ursprungsform rückgewandelt werden.

Der Aufwand, eine Socketkommunikation zu verwalten, ist daher sehr umfangreich. Daher gibt es als zweite Möglichkeit der Interprozesskommunikation die Java-Erweiterung RMI. Die Abkürzung RMI steht für Remote Method Invocation, also die virtuelle Möglichkeit des Aufrufs einer Methode durch ein beliebiges, beispielsweise entferntes Objekt.

Unter einem entfernten Methodenaufruf (RPC, Remote Procedure Call) versteht man folgenden Vorgang: Ein Client-Objekt sendet zunächst eine Nachricht an das entfernte Server-Objekt. Inhalt der Nachricht sind die aufzurufende Methode sowie die dafür benötigten Parameter. Daraufhin führt das Server-Objekt die entsprechende Methode aus und schickt das Resultat wieder an den Client zurück. Dabei werden sämtliche Probleme, die eine solche Kommunikation mit sich bringt, wie Objektfindung, Serialisierung der zu übertragenden Objekte, und die Synchronisation in der Kommunikation, von RMI übernommen und gelöst.

Damit dies möglich ist, werden die über das Netzwerk zu übertragenden Daten zunächst von speziellen Objekten, genannt *Stubs*, auf dem Client-Rechner gekapselt. Dabei werden die zu übermittelnden Objekte in eine Darstellung gebracht, die von verschiedenen Betriebssystemen einheitlich interpretiert wird. Bei reinen Integer-Zahlen entspricht dies wirklich lediglich dem Wechsel der Darstellungsform. Bei komplizierter strukturierten Objekten ist dieser Mechanismus jedoch deutlich aufwändiger. So sind komplizierter aufgebaute Java-Objekte zunächst lediglich über Pointer verbundene Speicherstellen, die nicht notwendigerweise hintereinander liegen müssen. Das Client-Stub muss diese einzelnen Speicherstellen also abwandern und in ein zusammenhängendes Objekt umwandeln. Dieser Vorgang wird Objektserialisierung genannt. Damit ein Stub diesen Vorgang durchführen kann, müssen die Objekte bereits bei Erstellung mittels des Interface Serializable darauf vorbereitet werden. Sämtliche Klassen, von denen später einmal Objekte als Funktionsparameter von RMI-Aufrufen auftreten, müssen dieses Interface daher implementieren.

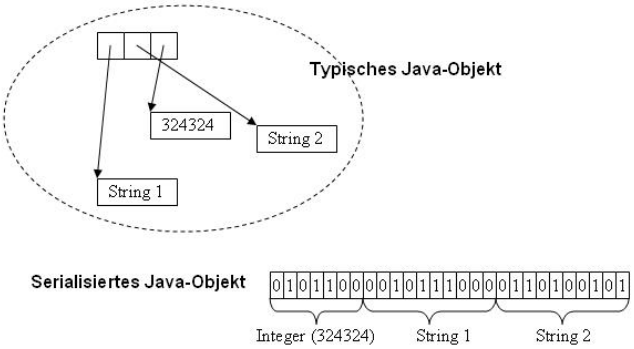


Abbildung 6: Serialisierung von Java-Objekten

Wird dies befolgt, so erledigt Java das Kapseln der Daten in Stubs und die eigentliche Datenübermittlung selbstständig. Das anschließende Arbeiten mit den verteilten Objekten ist daher kaum von dem mit lokalen Objekten zu unterscheiden. Auf der Serverseite, also der Seite, die den Aufruf empfängt, existieren so genannte *Skeletons*, die als Gegenstück zu den *Stubs* fungieren, und die empfangenen Daten wieder deserialsieren und in gewöhnliche Java-Objekte zurückwandeln, und anschließend die gewünschte Methode mit

diesen Objekten aufrufen. Auch das Zurückversenden des Rückgabewertes einschließlich der Serialisierung übernimmt das *Skeleton*.

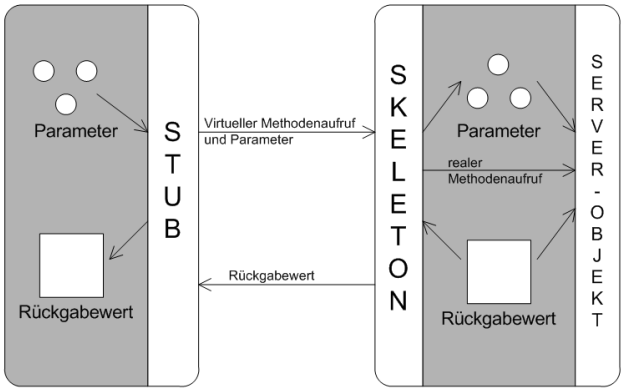


Abbildung 7: Funktionsprinzip von RMI

Neben dem Vorgang der Kommunikation stellt in der Praxis auch das Finden des gesuchten Kommunikationspartners ein Problem dar. Auch diese Verwaltung der Kommunikationsendpunkte wird von Java unterstützt. Zu diesem Zweck beinhaltet die Java-API die so genannte RMI-Registry. Einmal auf einem bestimmten Port initialisiert können in ihr beliebige Serverobjekte mittels Naming.bind() abgelegt werden. Danach können diese Serverobjekte aus einem beliebigen Prozess mittels Naming.lookup() referenziert und anschließend von einem Client aufgerufen werden. In ATOSj wird der Port, auf dem die Registry gestartet wurde, dem Prozess des Testobjekts mittels Kommandozeilenparamter mitgeteilt.

5.8.2 RMI in ATOSj

Wie bereits durch Marathon anschaulich demonstriert, wäre eine weitere Vereinfachung der Kommunikation mittels RMI durch die Bibliothek RMI-Lite möglich. Diese Bibliothek ist eine schmale Zwischenschicht, die es ermöglicht, beliebige Funktionsrufe an entfernten Objekten vorzunehmen, scheinbar ohne Vorhandensein von Skeletons und Stubs. Möglich wird dies, indem sämtlich RPC's durch ein „Universalpaar“ von Stub & Skeleton getunnelt werden. Die Einschränkung, dass dies lediglich auf einem festen Rechner funktioniert, war aufgrund der lokalen Beschränkung von ATOSj nebensächlich. Zur Verringerung des Arbeitsaufwandes basierte daher sämtliche Kommunikation in der frühen Entwicklungsphase von ATOSj auf dieser Hilfsbibliothek.

Mit zunehmenden Komplexitäts- und Qualitätsanforderungen zeigten sich jedoch recht bald Unzulänglichkeiten von RMI-Lite bei der Nutzung für ATOSj. Beispielsweise waren bei Fehlern in der Kommunikation die zugehörigen Ausschriften nicht auslesbar. Als

Hauptmanko aber stellte sich bald die Tunnelung durch stets dasselbe Objekt heraus. Durch diese Architektur bedingt, war es nicht möglich, zum selben Zeitpunkt unterschiedliche Kommunikationsinterfaces auf demselben Port arbeiten zu lassen. Da ATOSj pro Testobjekt bis zu drei Interfaces gleichzeitig nutzt, und außerdem die Möglichkeit bietet, gleichzeitig mehrere Testobjekte zu verwalten, hätte dies eine Ausweitung der ATOSj-Kommunikation auf theoretisch beliebig viele und realistisch bis zu 10 Ports zur Folge gehabt. Dies stellte eine nicht zu akzeptierende Unsauberkeit im Kommunikationsentwurf dar, weshalb die Kommunikation zu einem Zeitpunkt wieder auf eine eigene RMI-Implementation umgestellt wurde.

In ATOSj kann man drei Anwendungsfälle für RMI ausmachen: Zum Einen jeweils beim Abspielen und Aufzeichnen der Testsequenzen, und zum Anderen bei der Verwendung des URFGenerators für die automatische Generierung der Component-IDs zur eindeutigen Identifikation der grafischen Elemente.

Im Wesentlichen werden in ATOSj für jeden dieser drei Anwendungsfälle dieselben Vorbereitungen getroffen. Zunächst wird der Kommunikationspunkt, über den anschließend die Kommunikation erfolgen soll, vereinbart. Anschließend wird der entfernte Prozess gestartet, bevor er sich dann auf dem vereinbarten Kommunikationspunkt meldet. Danach wird das invertierte Kommunikationsinterface initialisiert, womit das Gerüst zur Interaktion von Testobjekt und Hauptprogramm vollständig ist.

Verantwortlich für das Erstellen des Kommunikationspunktes und die anschließende Prozesskreierung zeichnet sich der so genannte RMIConnectionManager. Seine allererste Aufgabe nach Programmstart ist das Erstellen einer neuen RMIRegistry. Standardmäßig wird versucht, diese auf dem von ATOSj reservierten Port 28675 zu initialisieren. (Die Wahl fiel auf gerade diesen Port, da die Eingabe des Wort „ATOSj“ auf einer Standard-Telefontastatur dem aufeinander folgenden Druck auf die Tasten 2, 8, 6, 7 und 5 entspricht.) Danach wird der RMIConnectionManager bei jedem benötigten Erstellen eines Testobjekts gerufen. Dieser generiert daraufhin einen eindeutigen String zur Referenzierung eines Kommunikationspunktes in der erstellten RMIRegistry. Anschließend ist der RMIConnectionManager für die Erstellung des neuen Testobjektprozesses zuständig. Via Kommandozeilenparameter wird diesem dabei sowohl der Port der zu nutzenden RMIRegistry, als auch die Bezeichnung des Kommunikationspunktes, auf dem er sein Kommunikationsinterface hinterlegen soll, mitgeteilt. Des Weiteren ist der RMIConnectionManager für Beendigung sämtlicher Testobjekte, die im Moment des Schließens des ATOSj-Hauptfensters noch aktiv sind, und die Leerung der RMIRegistry zum selben Zeitpunkt, zuständig.

Der RMIConnectionManager ist somit für die Initialisierung der Kommunikation mit den verschiedenen Testobjekten zuständig. Nach seiner Nutzung kann sich das ATOSj-Hauptprogramm zu dem Testobjekt verbinden und durch die im jeweiligen Kommunikationsinterface definierten Funktionen das Testobjekt steuern. Da es teilweise allerdings auch nötig ist, dass das Testobjekt direkt auf ATOSj einwirkt, wird ihm durch einen

Funktionsaufruf ein weiteres Interface mitgeteilt, mit dessen Hilfe es ihm beispielsweise mitteilen kann, dass das Testobjekt geschlossen wurde.

Zur Illustration dieses komplexen und elementar wichtigen Teils der Diplomarbeit, hier am Beispiel die von ATOSj vollzogenen Schritte zur Ausführung eines einzelnen HTS-START-Kommandos im HTSPlayer:

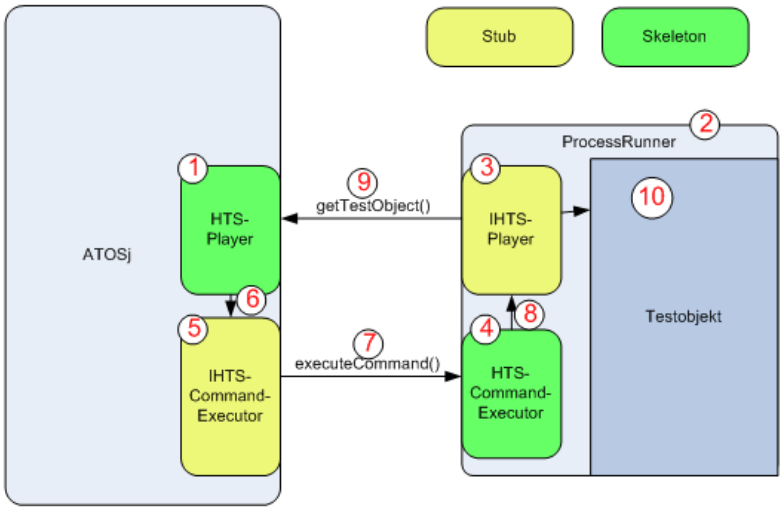


Abbildung 8: Abspielen eines HTS-Kommandos

Ausgehend von der Situation, dass ATOSj via Nutzerinteraktion der Befehl zum Ausführen dieses nur aus einem START-Befehl bestehenden Skripts gegeben wurde, wird intern zunächst ein neuer HTSPlayer aktiviert (1). Dieser fungiert gleichzeitig als Stub und registriert sich mittels RMIConnectionManager zugleich in der RMIRegistry. Wie bereits vorangehend beschrieben, startet der RMIConnectionManager anschließend den neuen Prozess, in dem später das Testobjekt ausgeführt werden soll (2).

Direkt nach seiner Aktivierung erstellt dieser Prozess eine Verbindung zu dem ihm bereitgestellten Interface und eröffnet somit ein neues Stub, über das später Funktionsaufrufe am Hauptprogramm ausgeführt werden können (3). Damit auch das ATOSj-Hauptprogramm aktiv auf den entfernten Prozess Einfluss nehmen kann, registriert dieser nun im nächsten Schritt ein eigenes Skeleton in der RMIRegistry (4).

Parallel dazu versucht sich der HTSPlayer im ATOSj-Hauptprogramm in kurzen Abständen zu dem vorher vereinbarten Komunikationspunkt zu verbinden. Dies schlägt solange fehl, bis der Prozess des Testclients seine Vorbereitungen abgeschlossen hat und bei (4) angelangt ist. Nun kann das Hauptprogramm sich zu diesem Client-Interface verbinden (5). Durch das zeitweilige Warten des Hauptprogramms erfolgt somit eine automatische

Synchronisierung.

Da zu diesem Zeitpunkt alle Kommunikationwege aufgebaut sind, kann der HTSPlayer anschließend mit dem abspielen der HTS-Kommandos beginnen. Der HTSPlayer übermittelt jedes Kommando durch einen Ruf der Funktion executeCommand() im zuvor erstellten Stub (6). In diesem Beispiel erfolgt also ein entfernter Methodenaufruf executeCommand(HTStart) (7).

Der gerufene entfernte HTSCommandExecuter sorgt direkt nach Empfang des HTS-Kommandos für dessen Ausführung. Da das HTSStart-Kommando das Testobjekt starten soll, muss es zunächst sämtliche für den Start benötigten Eigenschaften vom ATOSj-Hauptprogramm erfragen. Dies erfolgt durch Aufruf der Funktion getTestObject() auf dem Stub, dass mit dem vom Hauptprogramm registrierten HTSPlayer-Objekt verbunden ist (8 & 9). Der Rückgabewert dieser Funktion enthält alle für den Start des Testobjekts benötigten Daten, was somit direkt im Anschluss geschieht (10).

Erst nachdem dies abgeschlossen ist, wird der Funktionsaufruf von executeCommand beendet. Es ist also keine weitere Synchronisation nötig. Falls weitere HTS-Befehle existierten, könnte der HTSPlayer diese nun zur Ausführung bringen.

5.9 Identifikation grafischer Elemente

Damit ein Werkzeug für das automatische Testen eines Oberflächenprogramms funktionieren kann, bedarf es streng genommen keinerlei Kenntnisse über die zu testende GUI. Ein Programm, das Nutzereingaben unabhängig von den auf dem Bildschirm dargestellten Fenstern simuliert, ist beispielsweise das Programm GUI-do (http://www.gui-do.luebit.de/). Mit ihm ist es möglich, Maus- und Tastaturaktionen aufzuzeichnen und anschließend wieder abzuspielen. Die Nutzbarkeit eines auf diese Weise aufgezeichneten Skripts ist jedoch stark eingeschränkt, da allein eine veränderte Position eines Fensters die Wirkung sämtlicher Mausaktionen verändern könnte. Eine mögliche Lösung dieses Problems wäre der Ansatz, die Aktionen relativ zur Position des angesprochenen Fensters zu speichern. Da aber viele Fensterlayouts inzwischen dynamisch und von der Größe des Fenster abhängig sind, wäre auch dieser Ansatz in der Praxis wohl nicht nutzbar. Ebenfalls erscheint das sinnvolle nachträgliche Editieren der aufgezeichneten Skripte für die Testfallerstellung höchst kompliziert, da sich kein klares Muster in dieser Art Skript finden lässt.

Sowohl für eine stabilere Funktionsweise, als auch aus Gründen der besseren Nutzbarkeit ist es somit nötig, Aktionen an Oberflächenobjekten an Ihre Adressaten zu binden. Zumindest sollten die genutzten Skripte diese Form haben. Bei der tatsächlichen Manipulation der Objekte zur Laufzeit erscheint Retransferierung in einfache Tastatur- und Mauskommandos durchaus sinnvoll. Genaueres dazu findet sich im dazugehörigen Kapitel

5.5.

Damit diese Objekte referenziert werden können, muss ihnen ein Bezeichner zugeordnet werden. Dabei muss es bei jedem Start des Testobjekts möglich sein, das grafische Element allein mittels dieses Bezeichners eindeutig zu wieder zu finden.

Im Vorgängermodell ATOS wurden grafische Basiselemente weder bei ihrer Instanziierung noch durch deren Referenzierung im Quelltext ermittelt. Vielmehr nutzen die beiden Entwickler die bei Windows-Programmen obligatorischen Resource-Dateien, in denen die Strukutur der GUI eindeutig festgelegt ist. Darin ist jedes grafische Basiselement mit einer dazugehörigen eindeutigen ID gelistet. Somit wurde sichergestellt, dass alle Oberflächenelemente identifiziert und überwacht werden können.

In Java existieren derartige Resource-Dateien jedoch nicht. Man muss die grafischen Elemente daher zur Laufzeit identifizieren und einer generierten ID zuordnen. Für diese Identifikation müssten theoretisch die vom Betriebssystem an das Programm verschickten Nachrichten überwacht, und anschließend daraus die Adressaten ermittelt werden. Dies ist zum einen höchst unpraktikabel, und zum anderen auch nicht nötig, da wie in Kapitel 5.5 beschrieben, Java diese Nachrichten automatisch in interne Events umwandelt. Es ist also nur nötig, diese Events zu überwachen. Aus ihnen lassen sich alle für das Capture & Replay nötigen Informationen extrahieren.

Ein weiteres Problem stellt die in Kapitel 2 beschriebene Eigenschaft von Java dar, dass mit Swing und SWT zwei verschiedene GUI-Typen existieren. Ziel von ATOSj war die Unterstützung beider Oberflächen.

Für ATOSj war daher eine Architektur zu entwerfen, die es ermöglicht, beide Oberflächen zu überwachen und zu manipulieren. Dennoch sollte das System ein einheitliches Objekt darstellen. Es musste also von vornherein unter dem zusätzlichen Aspekt der Erweiterbarkeit entworfen werden. Dazu wurde zunächst eine allgemeine Strategie für die Arbeitsweise von Recorder und Player festgelegt:

5.9.1 Namensgenerierung beim Capture im Recorder

Erfolgt ein durch eine Nutzeraktion ausgelöstes Java-Event im Testobjekt, das die Generierung eines HTS-Kommandos erforderlich macht, so muss zunächst ein eindeutiger Bezeichner für das auslösende Element gefunden werden. In ATOSj bestehen diese Bezeichner aus drei Komponenten:

- Fenstertitel: Titel des Fensters in dem sich das Element befindet bzw von sich selbst, falls das Element ein Fenster ist.

- Elementname: Der durch eine Benamungsstrategie generierte Name des Elements
- Elementtyp: Typ des Objektes, z.B. BUTTON, WINDOW oder EDITBOX

Das für die Generierung des Fenstertitels benötigte umgebende Fenster lässt sich recht einfach durch rekursives ermitteln des Elternelements berechnen.

Die restlichen Elemente des HTS-Kommandos lassen sich aus dem Typ des auslösenden Java-Events herleiten. Liegt beispielsweise ein SelectionEvent an einer Checkbox vor, so muss lediglich ermittelt werden, ob ein CHECK oder ein UNCHECK Kommando zu erzeugen ist. Dies kann durch einfaches ermitteln des Auswahlzustandes des auslösenden Objektes nach dem Event bestimmt werden.

5.9.2 Elementfindung beim Replay

Beim Ausführen eines HTS-Kommandos während des Replays muss zunächst anhand der gegebenen ID das passende Element ermittelt werden. Dazu wird im ersten Schritt ein Fenster mit dem in der ID angegebenen Titel gesucht. Wurde dies gefunden, so wird dieses Fenster rekursiv absteigend dahingehend untersucht, ob es ein Element des angegebenen Basistyps enthält. Wird eine solches Element gefunden, so wird diesem Element mittels der in ATOSj implementierten Benamungsstrategie ein Name zugewiesen. Entspricht dieser Name dem gesuchten Elementbezeichner, so wurde das geforderte Element ermittelt.

Anschließend kann am Element die gewünschte Aktion ausgeführt werden. Dies geschieht in Abhängigkeit vom Typ der durchzuführenden Aktion entweder direkt am Element (programmatischer Ansatz) oder durch simulierte Nutzereingaben (ereignisbasierter Ansatz). Direkt etwa beim Auslesen des Inhalts eines Textfeldes, ereignisgesteuert beispielsweise beim Klicken auf einen Button.

5.9.3 Strategie für die Benamung grafischer Elemente

Sämtliche in der HTS-Syntax existierenden ACTION-Befehle sind an einen Empfänger gebunden. Dies ist immer ein grafisches Element. Optional ist mittels des Schlüsselworts SUBITEM auch noch eine detaillierte Adressierung möglich. Der Empfänger wird dabei stets durch die drei Bestandteile der Component-ID angegeben. Wichtigster Bestandteil dieser ist der Name, der dem Element zugeordnet wurde. Er erfüllt zwei Hauptzwecke: Zum einen sorgt er für die Unterscheidung von grafischen Elementen, damit immer eindeutig festgelegt ist, welches Element aktuell referenziert wird. Es ist daher wichtig, dass sämtlichen Elementen gleichen Typs in einem Fenster stets ein unterschiedlicher Name

zugeordnet wird. Zum anderen ist es mittels des Namens möglich, ein Skript zu charakterisieren. Werden die Elemente durch Entwickler bereits durch aussagekräftige Namen charakterisiert, so lässt sich später beim Lesen eines Skriptes viel leichter auf dessen Semantik schliessen.

ATOSj versucht den Namen eines Elements in zwei Stufen zu bilden. Zuerst wird überprüft, ob der Entwickler einen Namen für die spätere Verwendung in ATOSj vergeben hat. Falls kein solcher vorgegebener Name gefunden wird, so wird versucht, in Abhängigkeit von der Ausprägung des Elements einen Namen zu generieren.

Ein vordefinierter Name muss dem Element vom Entwickler als Eigenschaft mitgegeben werden, ohne dabei einen Einfluss auf die Funktionalität zu haben. Da Swing und SWT dem Anwender unterschiedliche Möglichkeiten bieten, werden diese Werte in Abhängigkeit von der genutzen Oberfläche auf unterschiedliche Weisen gesetzt.

Unter Swing existiert für jedes grafische Element die Möglichkeit, direkt einen Namen zu verteilen. Diese nicht funktionale Eigenschaft kann mittels der Funktionen getName() und setName() gesetzt und wieder ausgelesen werden. Wird also von ATOSj beim Rufen der getName()-Funktion ein Wert gefunden, so wird dieser anschließend für das Element verwendet.

Unter SWT besteht diese Möglichkeit nicht. SWT erlaubt es aber jedem grafischen Basiselement beliebige Datenpaare zuzuordnen. Ein solches Datenpaar besteht aus einem String, der als Schlüssel für ein anschließendes Ändern oder Auslesen fungiert, und einem beliebigen Java-Objekt, dass den dazugehörigen Wert beinhaltet. Diese Möglichkeit wird in ATOSj verwendet, um dem Objekt einen vordefinierten Namen zu geben. Dazu muss dem Objekt ein Datenpaar mit dem Wert „ATOSJ_COMPONENT_NAME_KEY“ als Schlüssel zugeordnet werden. ATOSj versucht in diesem Fall, das zugeordnete Wert-Objekt in einen String zu casten. Gelingt dies, so wird der gefundene String als Name für das Element verwendet. Die manuelle Vergabe eines Namens ist unter SWT also mittels `setData(‘‘ATOSJ_COMPONENT_NAME_KEY‘‘, ‘‘Elementname‘‘)` möglich.

Schlägt das Auffinden eines vom Entwickler vergebenen Namens fehl, so versucht ATOSj anhand des Typs des zu benamenden Elements und dessen speziellen Eigenschaften einen Namen zu generieren. Typische Eigenschaften sind beispielsweise die Referenz auf ein Label oder der dargestellte Text auf einem Button oder einem Menü.

Sollte auch dies fehlschlagen, so versucht ATOSj in der dritten Stufe die Position des Elements im aktuellen Fenster zu bestimmen, und ge neriert daraus eine Zeichenkette. Ein Beispiel für einen auf diese Art und Weise entstandenen Namen ist etwa „Pos [x=4, y=22, b=100, h=38]“.

Da auch dies in einigen Sonderfällen nicht immer einen Wert liefert, wird in der vierten und letzten Stufe an dem betreffenden Element die an einem Java-Objekt immer exis-

tierende Funktion `toString()` gerufen, und der Wert dieser Funktion als Name für das Element verwendet.

5.9.4 Benötigte Java-Erweiterungen

Für die Umsetzung der gewählten Strategie sind zwei unter Java nicht vorhandene Zusatzfunktionen nötig. Für die Funktion des Recorders ist es nötig, globale Listener zu erstellen. Dies ermöglicht es, alle Events, die die Generierung eines HTS-Kommandos erforderlich machen könnten, zu überwachen. Die Alternative, an sämtlichen in der GUI existierenden Elementen separat einen Listener zu registrieren, führt nicht zum Ziel, da die sich potentiell verändernde Struktur der GUI ständig auf neue Elemente überprüft werden müsste. Bei umfangreichen grafischen Oberflächen würde dies zu einer nicht zu akzeptierenden Prozessorauslastung führen.

Die Umsetzung der Playerstrategie erfordert zusätzlich die Möglichkeit, über sämtliche sich öffnenden und schließenden Fenster informiert zu werden, da ATOSj das Testobjekt lediglich startet, jedoch keine Informationen über die aufgebaute GUI hat. Da jedoch zum Auffinden eines referenzierten Elements in der GUI rekursiv alle aktuell existierenden Fenster nach diesem Element durchsucht werden müssen, muss ATOSj darüber informiert werden, wann sich Fenster öffnen und schließen.

Für Swing sind beide benötigten Features im Rahmen der Java Accessibility Utilities verfügbar. Sie bieten die Möglichkeiten zur Überwachung von sämtlichen Java-AWT-Events und den aktuell existierenden Fenstern.

Für die Arbeit mit SWT stehen derartige Hilfsmittel derzeit noch nicht zur Verfügung. Um ein Capture & Replay aber dennoch zu ermöglichen, mussten die benötigten Funktionalitäten daher selbst implementiert werden. Es stellte sich recht schnell heraus, dass für eine eigene Lösung eine Veränderung der SWT-Klassen zu unseren Zwecken unumgänglich wäre. Dies würde aber erfordern, dass in jeder ATOSj-Distribution eine eigene, angepasste Version der SWT-Klassen integriert sein müsste. Daher müsste für jede SWT-Version eine eigene ATOSj-Bibliothek mitgeliefert werden, andernfalls würde dem Anwender stets eine spezielle - möglicherweise die aktuelle - SWT-Version vorgegeben sein. Des Weiteren würde dies eine für jedes Betriebssystem verschiedene ATOSj-Distribution erforderlich machen. Zusätzlich hätte bei jedem Update der SWT-Klassen seitens des Herstellers IBM eine aktuelle Version von ATOSj mit der aktualisierten Version der angepassten Klassen geliefert werden müssen.

Aufgrund dieser vielseitigen Einschränkungen wurde die Möglichkeit einer derartigen Implementation verworfen. Als zweiten Ansatz einer möglichen eigenen Implementation der benötigten GUI-Überwachungsfunktionen wurde das Verändern des Java-Classloaders in Betracht gezogen. Angedacht war, den Klassenlader derart zu verändern, dass er beim La-

den einer normalen GUI-Klasse nicht die Originalklasse, sondern eine abgeleitete Klasse lädt, die über unsere zusätzlich benötigten Eigenschaften verfügt. Auch dieser Ansatz war jedoch nicht umsetzbar, da der entstehende Namenskonflikt zwischen Originalklassenname und dem Namen der abgeleiteten Überwachungsklasse nicht gelöst werden konnte.

Da sich der Ansatz des Ladens von manipulierten abgeleiteten Klassen als äußerst kompliziert erwies, wurden verschiedene Hilfsmittel getestet. Eine der getesteten Bibliotheken war Javassist. Die vielfältigen Möglichkeiten dieser Klassensammlung ermöglichten die Umsetzung eines dritten, vorher noch nicht angedachten Ansatzes. Javassist ermöglicht das Verändern der zu ladenden Klasse zur Laufzeit. Die SWT-Oberflächenklassen konnten daher während des regulären Ladens um die benötigten Zusatzfunktionen erweitert werden.

Mittels der in Javassist enthalten Funktionalitäten konnte unter SWT ein Pendant zu den unter Swing existierenden Java Accessibility Utilities geschaffen werden. Die Funktionsweise dieses SWTEventMonitor genannten Pakets wird im folgenden Kapitel erläutert. Mit dessen Hilfe konnte sowohl das Capture als auch das Replay nach demselben Schema wie schon unter Swing erfolgt, implementiert werden. Lediglich der Start des Testobjekts musste um eine weitere Schicht erweitert werden, die sicherstellt, dass nicht die „normalen“, sondern die von Javassist manipulierten Klassen vom Java Classloader geladen werden.

5.10 SWTEventMonitor

Mit den Java Accessiblitiy Utilities existiert für die Arbeit unter Swing ein Hilfsmittel, dass sowohl einen Capture- als auch einen Replayvorgang wie er für ein Automatisierungstool wie ATOSj benötigt wird, ermöglicht. Die API-Erweiterung liefert benötigte Informationen über sämtliche in der GUI erstellten Fenster. Auch bietet sie die Möglichkeit, Java-Events global zu empfangen, d.h. ohne zuvor Kenntnis von deren Auslöser zu haben. In Kombination mit der Java-API-Funktion `java.awt.EventQueue.postEvent()` zur Simulation von Mausklicks und Tastendrücken stehen unter Swing sowohl für Capure als auch Replay alle benötigten Funktionalititäten zur Verfügung.

Für die Simulation dieser Nutzereingaben unter SWT existiert in der SWT-API die Funktion `org.eclipse.swt.widgets.Display.post()`. Sie bildet das benötigte Pendant zur unter Swing existierenden `postEvent()`-Funktion. Um genügend Informationen über die zu manipulierende GUI zu erhalten, beispielsweise um diese Events zu adressieren, wird jedoch auch zu den Java Accessiblitiy Utilities eine Entsprechung benötigt. Allerdings ist von seiten des SWT-Entwicklers, der Eclipse Foundation, eine solche Erweiterung weder existent noch zukünftig angedacht.

Wie bereits im vorangehenden Kapitel beschrieben, zeigte sich während des Entwick-

lungsprozesses, dass die Bibliothek Javassist für die Implementation dieser benötigten Zusatzfunktionen ein nützliches Hilfsmitel sein könnte. Mittels Javassist kann der Binärcode von beliebigen zu ladenden Klassen während des Ladevorgangs geändert werden.

Ziel dieser Änderung muss ein Mechanismus sein, der ATOSj darüber informiert, welche Elemente in der GUI existieren. Genauer gesagt müssten - analog zur Implementationsstrategie unter Swing - Events an sämtlichen grafischen Elementen gecaptured werden können. Außerdem wird ein Mechanismus benötigt, der ATOSj über alle neu erstellten Fenster informiert. Sämtliche enthaltenen Objekte können dann durch ihre Vater-Kind-Beziehung zum diesem Fenster identifiziert werden.

Damit die Änderungen an den grafischen Klassen vorgenommen werden können, muss zunächst das Laden der veränderten Klassen durch Javassist vorbereitet werden.

```
public void onLoad(ClassPool pool, String classname)
...
    Loader loader = new Loader();
    Tranlator trans = new SWTElementTranslator();
    loader.addTranslator( ClassPool.getDefault(), trans );
    loader.run(mainClassName, realArgs);
...
```

Listing 4: Vorbereiten der Manipulation der SWT-Originalquellen

Dazu wird zunächst ein so genannter Loader erstellt, der später für das Laden verantwortlich sein wird. Damit der erstellte Loader die Klassen verändert lädt, können ihm Modifikatoren hinzugefügt werden, die das veränderte Verhalten der Klassen festlegen. Mittels dieser in Javassist Translator genannten Modfikatoren kann spezifiziert werden welche Klassen auf welche Art und Weise verändert werden sollen. Anschließend wird der Loader mit dem Laden sämtlicher weiteren Klassen mittels loader.run() beauftragt.

Die in ATOSj für die Veränderungen der SWT-Klassen zuständige Implementation eines Translators nennt sich SWTElementTranslator. Nach seiner Registrierung werden sämtliche Klassen vor Ihrem Laden durch die Funktion onLoad() des Modifikators „geschleust“. In dieser Funktion werden nun in Abhängigkeit vom Namen der zu ladenden Klasse zuvor Veränderungen an dessen Klassenkörper vorgenommen.

Bevor auf das genaue Verhalten des SWTElementTranslators eingegangen werden kann, muss zuvor noch die Zielstrategie des die veränderten Klassen später benutzenden SWTEventMonitors erläutert werden. Der SWTEventMonitor ist für die interne Verwaltung aller erstellten Listener und Fenster verantwortlich. Er stellt damit das Gegenstück zur Hauptklasse SwingEventMonitor der Java Accessibility Utilities dar. Analog zu dessen Verhalten soll es später möglich sein, beispielsweisse mittels SWTEventMonitor.addModifyListener sich über sämtliche Modifikationsevents in der zu überwachenden GUI informieren zu lassen. Analog dazu erfolgt die Überwachung aller Fenster und aller

weiteren Eventtypen.

Damit der SWTEventMonitor dies leisten kann, muss er selbst von sämtlichen Events erfahren. Strategie ist es daher, die SWT-GUI-Klassen derart bei Ihrem Laden zu modifizieren, dass sämtliche ihrer Instanzen dem SWTEventMonitor von den an ihnen erfolgten Events mitteilen. Dieser leitet diese anschließend an die wiederum bei ihm registrierten Listener weiter.

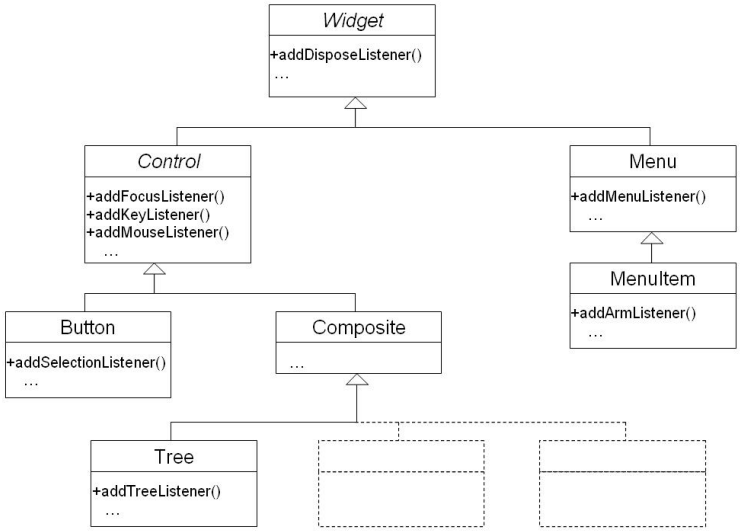


Abbildung 9: Ausschnitt aus der Hierarchie der SWT-Typen und einiger ihrer abfragbaren Eventtypen

Die SWT-Klassen werden dazu durch den SWTElementTranslator derart verändert, dass jede Instanz nach dem Beenden des regulären Konstruktors für jeden an ihr erlaubten Eventtyp einen Listener anhängt. Seine einzige Aufgabe ist die Weiterleitung des Events. Diese CallbackListener genannten Empfänger sind Teil des SWTEventMonitors und haben Kenntnis von den für diesen Eventtyp im SWTEventMonitor registrierten Listener. Tritt tatsächlich ein Event dieses Typs auf, so wird der Listener vor allen an diesem grafischen Element regulär registrierten Empfängern aktiviert. Im SWTEventMonitor registrierte Listener erfahren daher von der vorgenommenen Aktion direkt nachdem sie passierte, also noch bevor die regulär im Testobjekt registrierten Listener darauf reagieren können.

Ausschlaggebendes Implementationsdetail ist daher die Veränderung der SWT-Oberflächenklassen durch den SWTElementTranslator. Ziel ist die Veränderung der Klassen derart, dass jede neu erstellte Instanz dieser Klassen für sämtliche erlaubten Eventtypen automatisch einen Listener einfügt, der die Weiterleitung an den SWTEventMonitor übernimmt. Der

SWTEventMonitor übermittelt dann das Event an die in Ihm registrierten Listener.

Um dies zu erreichen, wurden zunächst die Quelltexte der SWT-Klassen studiert. Teilweise existieren mehrere öffentliche Konstruktoren. Im Allgemeinen rufen diese jedoch alle einen nicht öffentlichen „Hauptkonstruktor“, der das Objekt tatsächlich erstellt. Die Veränderungen wurden daher an das Ende dieses Konstruktors eingefügt. Aufgrund dieses Vorgehens könnten größere Veränderungen der SWT-Quellen in zukünftigen Versionen eine Anpassung des SWTEventTranslators erforderlich machen.

```
public void onLoad(ClassPool pool, String classname)
...

  if (classname.equals("org.eclipse.swt.widgets.Button"))    {

    CtClass cc = pool.get(classname);

        CtConstructor constructor = cc.getConstructor(
          "(Lorg/eclipse/swt/widgets/Composite;I)V");

        constructor.insertAfter("addSelectionListener(" +
          "SWTEventMonitor.Internal.callBackSelectionListener);");
  }

...
```

Listing 5: Veränderung der SWT-Quellen

Das voranstehende Beispiel zeigt die Veränderung der SWT-Klasse Button mittels Javassist. Zunächst wird der Original-Bytecode der Klasse eingelesen. Anschließend wird ein spezieller Konstruktor ausgewählt und an sein Ende eine Quelltextzeile eingefügt. Der in Javassist integrierte Kompiler wandelt diese Zeile automatisch in Java-Bytecode um und fügt sie anschließend in das Ende des Konstruktors ein. Die Leistungsfähigkeit der Bibliothek Javassist ist bemerkenswert. So ermöglicht sie in diesem Falle den Zugriff auf ein Objekt der Klasse SWTEventMonitor.Internal, einer Klasse, von der die zu verändernde Klasse Button zu ihrem ursprünglichen Kompilierungszeitpunkt noch nicht einmal wusste, dass sie überhaupt existiert.

Zum Abschluss dieses Kapitels noch eine Anmerkung zum Thema Java Accessibility Utilities: Während des ausführlichen Tests von ATOSj stellten sich einige Unzulänglichkeiten dieser Bibliothek heraus. Dies liegt wohl darin begründet, dass dieses Hilfsmittel 1997 entstand und seit 1999 nicht mehr aktualisiert wurde. Die im SWTEventMonitor genutzten Ideen könnte ebenso auch für AWT/Swing umgestetzt werden, was in ATOSj zu Verbesserungen bezüglich Programmumfang, Stabilität und Einheitlichkeit führen würde. Bei einer weiteren Pflege und Erweiterung des Programms ist eine solche Anpassung daher anzuraten.

5.11 Überdeckungsanalyse

„Ziel der Entwicklung von Tests ist der Nachweis der Erfüllung der Anforderungen durch das implementierte Programmstück...“ [3, S. 96]. Dieser Nachweis kann nur erbracht werden, wenn alle Tests fehlerfrei verlaufen sind und wenn bekannt ist, ob auch tatsächlich alle Anforderungen durch die erstellten Testfälle abgedeckt wurden. Zu diesem Zweck entwickelten Hanisch und Letzel die Idee, eine Überdeckungsanalyse der ausgeführten Funktionen des Testobjektes durchzuführen. Die Idee der Überdeckungsanalyse stammt ursprünglich aus dem Bereich des Whitebox-Tests. Ein Whitebox-Test setzt die Kenntnis der inneren Struktur des Programmes, das heißt des Quelltextes, voraus und wird deshalb auch Strukturtest genannt. Basierend auf dieser Kenntnis werden die Testfälle erstellt. Zur Überprüfung der Vollständigkeit der Testfälle werden verschiedene Verfahren zur Überdeckungsanalyse genutzt. Zu diesen Verfahren gehören die Anweisungsüberdeckung, die Zweigüberdeckung und verschiedene Verfahren zur Analyse der Bedingungsüberdeckung. Am Ende einer solchen Analyse steht eine Maßzahl. Die Maßzahl C_0 für die Anweisungsüberdeckung, beispielsweise, gibt an welcher Prozentsatz von Anweisungen während des Tests tatsächlich zur Ausführung gelangt ist. Sie berechnet sich nach folgender Formel:

$$C_0 = \frac{Anzahl\ durchlaufene\ Anweisungen}{Anzahl\ Anweisungen\ insgesamt} \cdot 100\%$$

Je größer die Maßzahl desto größer ist die Güte der definierten Testfälle. Eine Überdeckung von 100 Prozent wäre ideal und bedeutet, dass das Testobjekt im Rahmen des gewählten Testverfahrens lückenlos getestet wurde. Ein lückenloser Test ist aber nicht immer mit vertretbarem Aufwand zu erreichen, denn ein hoher Überdeckungsgrad kann häufig nur durch eine hohe Anzahl an Testfällen realisiert werden. Deshalb wird das Testendkriterium in der Praxis häufig entschärft und ein geringerer Überdeckungsgrad als Testziel gewählt. Dieser richtet sich nach der Priorität der zu testenden Anforderung. Eine Priorisierung der Anforderungen wird häufig anhand ihrer Kritikalität vorgenommen. Zu den besonders kritischen Anforderungen gehören besonders häufig genutzte oder auch sicherheitsrelevante Funktionen [3].

ATOSj basiert auf dem Prinzip des Blackbox-Tests. Der Programmcode ist dem Tester unbekannt und das Testergebnis kann nur aus den Reaktionen des Testobjektes abgeleitet werden. Das Testobjekt bildet eine Blackbox (schwarzer Kasten), welche die Transformationen zwischen Ein- und Ausgabe verbirgt. Ein Test mit ATOSj soll das Testobjekt gegen seine Spezifikation testen. Dem Test liegen die funktionalen Anforderungen zu Grunde, weshalb man auch von einem Funktionstest spricht. Ziel ist es, die Funktionalität des Testobjektes möglichst vollständig zu testen, deshalb kann man in Analogie zu den strukturellen Überdeckungsmaßen beim Funktionstest von Funktionsüberdeckung sprechen [4].

Beim Testdesign für das Programm Seminarorganisation haben wir unsere Testfälle aus den im Pflichtenheft beschriebenen Anwendungsfällen und den damit verbundenen Produktfunktionen abgeleitet. Hiernach stellte sich für uns die Frage nach der Vollständigkeit, der von uns definierten Testfälle, also der von uns erreichten Funktionsüberdeckung. Die genaue Messung der Funktionsüberdeckung stellt sich als schwierig dar. Sie könnte beispielsweise im Rahmen eines statischen Reviews, welches die erstellten Testfälle mit den Produktfunktionen vergleicht, ermittelt werden. Dieses Verfahren ist aber insbesondere bei einer großen Menge an Testfällen fehleranfällig. Deshalb entschlossen wir uns, dem Ansatz von Hanisch und Letzel zu folgen und eine Teilautomatisierung der Funktionsüberdeckungsanalyse durchzuführen. Beim Oberflächentest wird das Testobjekt über seine GUI-Schnittstelle manipuliert. Diese Art der Manipulation ist in ihrer Granularität sehr grob, denn sie führt zu einer ganzen Reihe von Funktionsaufrufen innerhalb des Testobjektes und macht die Anwendung der Überdeckungsmaße aus dem Whitebox-Test wenig sinnvoll. Es wird ein Maß benötigt, welches noch gröber als die Anweisungsüberdeckung ist. Deshalb soll zur Approximation der Funktionsüberdeckung die Anzahl der ausgeführten Methoden der Klassen des Testobjektes dienen. Diese Art der Überdeckungsanalyse soll im Weiteren, in Abgrenzung zur Funktionsüberdeckungsanalyse, Methodenüberdeckungsanalyse genannt werde. Die Methodenüberdeckungsanalyse liefert das Maß C_{-1}, welches von uns wie folgt definiert wird:

$$C_{-1} = \frac{Anzahl\ durchlaufene\ Methoden\ pro\ Klasse}{Anzahl\ Methoden\ pro\ Klasse\ insgesamt} \cdot 100\%$$

Hierarchisch darauf aufbauend folgen die Maße C_{-2} als Ergebnis der Paketüberdeckung und C_{-3} als Ergebnis der Systemüberdeckung.

$$C_{-2} = \frac{Anzahl\ überdeckte\ Klassen}{Anzahl\ Klassen\ insgesamt} \cdot 100\%$$
$$C_{-3} = \frac{Anzahl\ überdeckte\ Pakete}{Anzahl\ Pakete\ insgesamt} \cdot 100\%$$

Diese drei Maße bieten eine gute Unterstützung zur Überprüfung der Vollständigkeit der Testfälle. Sollte ein Paket oder eine Klasse nur eine geringe oder gar keine Überdeckung aufweisen, kann dies ein Indiz dafür sein, dass der Test wesentlicher Funktionalität vergessen wurde. Der fehlende Aufruf einer Methode kann aber auch andere Ursachen haben, wie von Hanisch und Letzel beschrieben [7, S. 201]. Die Ursachen einer mangelhaften Methodenüberdeckung zu erkennen und daraus die richtigen Schlussfolgerungen in Bezug auf die Funktionsüberdeckung zu treffen, liegt allein beim Tester.

Die Frage nach der Vollständigkeit der Testfälle stellt sich eigentlich bei jedem Testprojekt. Darum haben wir uns entschieden die Methodenüberdeckungsanalyse direkt in ATOSj zu integrieren. Damit entfällt der Verwaltungsaufwand, der durch die Nutzung

eines externen Programmes, wie des Werkzeugs CTC++, welches von Hanisch und Letzel für die Analyse verwendet wurde, entsteht. Für die Unterstützung der Methodenüberdeckungsanalyse suchten wir ein Werkzeug, welches folgenden von uns aufgestellten Kriterien genügen sollte.

- Es soll keine Instrumentierung des Quellcodes vorgenommen werden. Bei der Instrumentierung werden in den Quelltext Anweisungen an allen relevanten Stellen eingefügt. Sie bewirken bei ihrer Ausführung das Inkrementieren eines Zählers. Dieses Verfahren ist gängige Praxis bei vielen Werkzeugen zur Überdeckungsanalyse. Es birgt jedoch gravierende Nachteile in sich. Zum einen ist die Verfügbarkeit des Quelltextes zwingend notwendig, zum anderen wird der Quelltext direkt manipuliert. Eine derartige Manipulation verringert oder verhindert gar die Lesbarkeit der Quellen. Um dieses Problem zu umgehen, wird häufig ein weiteres Projekt zur Verwaltung der instrumentierten Quellen angelegt, was wiederum den Aufwand erhöht.

- Es soll ein separat ausführbares Programm sein. Häufig sind Werkzeuge zur Überdeckungsanalyse nur für bestimmte Entwicklungsumgebungen verfügbar und somit nicht zur Integration in ATOSj geeignet.

- Es soll unabhängig vom Testframework sein. Viele Werkzeuge im Bereich Überdeckungsanalyse sind untrennbar mit dem Framework JUnit zum Modultest von Java-Programmen verbunden. ATOSj hingegen arbeitet auf Systemtestebene und schließt somit die Nutzung von JUnit aus.

- Es soll kostenlos zur Verfügung stehen.

- Es soll die Methodenüberdeckung messen. Die Ebenenen unter der Methodenüberdeckung sind für unsere Zwecke zu genau.

In der nachfolgenden Tabelle haben wir die von uns gefundenen, kostenfreien Programme aufgelistet.

Programm	**Ausschlußkriterium**
Quilt	Quilt bietet nur die Analyse der Anweisungsüberdeckung und ist für die Arbeit mit JUnit entwickelt.
NoUnit	NoUnit arbeitet auf der Ebene der Methodenüberdeckung, ist aber nur zur Ermittlung der Überdeckung durch JUnit-Testfälle geeignet.
InsECTj	InsECTj ist lediglich ein Framework und bedingt die Implementation eigener Klassen. Es fehlt insbesondere ein vordefinierter Generator für Reports.
Hansel	Liefert nur eine Überdeckungsanalyse für JUnit-Tests.

Jester	Liefert nur eine Überdeckungsanalyse für JUnit-Tests.
jcoverage	jcoverage ist kein eigenständiges Programm sondern nur als Erweiterung für die Entwicklungsumgebung Eclipse verfügbar.
ACover	Liefert nur eine Überdeckungsanalyse für JUnit-Tests.

Tabelle 8: Programme zur Überdeckungsanalyse

Da keines der gefundenen Programme vollständig unseren Wünschen entsprach, haben wir uns für die Eigenentwicklung eines Werkzeuges entschieden, welches genau auf unsere Anforderungen zugeschnitten ist. In diesem Rahmen ist das Modul „ACover“ entstanden. Seine Funktionalität wird in ATOSj genutzt, um die Methodenüberdeckung für alle durchgeführten Testläufe zu ermitteln. Die genauen Anforderungen für das Modul „ACover“ sind dem Pflichtenheft im Anhang B zu entnehmen. Das Modul besteht aus drei Kernstücken:

1. Generator für eine XML-Datenbasis der Klassen und Methoden des Testobjektes.

2. Komponente zur Laufzeitüberwachung von Funktionsaufrufen.

3. Generator für eine in HTML formatierte Ausgabe der Ergebnisse.

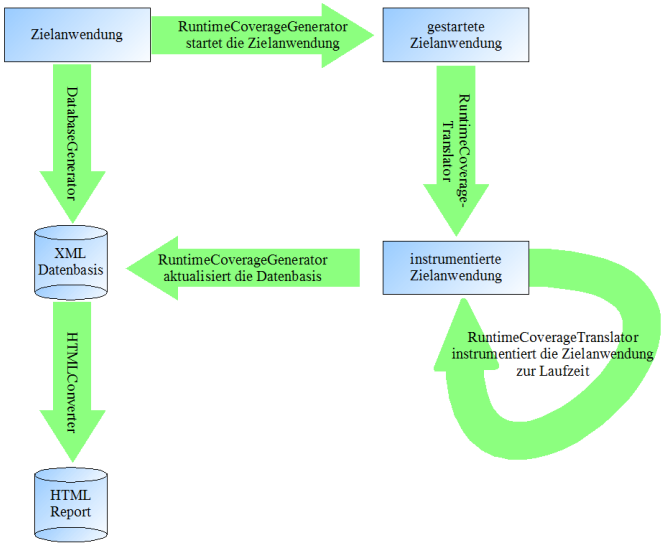


Abbildung 10: Arbeitsweise des Moduls ACover

Die Arbeit mit dem Modul erfolgt in drei Schritten. Im ersten Schritt wird mit der Generatorkomponente, implementiert in der Klasse `acover.DatabaseGenerator`, eine XML-Datenbasis mit Informationen über die Zielanwendung angelegt. Dazu muss der

Nutzer alle Pakete der Zielanwedung angeben. Die XML-Datei enthält Informationen zu den Klassen und Methoden, die in den Paketen enthalten sind. Für jede Methode existiert ein Zähler ihrer Aufrufe. Das Anlegen der Datenbasis stellt sicher, dass nur Metriken, die die Zielanwendung betreffen erstellt werden. Dies ist sinnvoll, da viele Anwendungen Standardbibliotheken oder Bibliotheken von Drittanbietern verwenden, deren Ausführung nicht dokumentiert werden soll, da sie keine Relevanz für einen Test der Zielanwendung haben. Zur Validierung der XML-Datei wird das in Listing C.2 dargestellte XML-Schema benutzt. Ein XML-Schema spezifiziert alle gültigen Elemente und Attribute einer XML Instanz. Weiterhin legt es die Hierarchie für geschachtelte Elemente fest und ermöglicht die Restriktion von Attributwerten sowie viele weitere Restriktionen bezüglich der XML Instanz [5, S. 5]. Der von uns verwendete Parser aus der Java API (JAXP - Java API for XML Processing) führt beim Einlesen automatisch eine Validierung der XML Datenbasis gegen das erstellte Schema durch und meldet einen Fehler, falls das Dokument nicht der Schemadefinition entspricht.

Auf das Erstellen der Datenbasis erfolgt dann im zweiten Schritt die Ausführung der Zielanwendung. Die Ausführung wird durch die Klasse `acover.RuntimeCoverageTranslator` überwacht. Sie wird jeweils vor dem Laden einer Klasse benachrichtigt. Anhand des Namens der zu ladenden Klasse wird dann festgestellt, ob diese in der Datenbasis enthalten ist. Wenn ja, wird in alle ausführbaren Funktionen dieser Klasse, mit Hilfe der Bibliothek Javassist (siehe Kapitel 5.6), eine neue Zeile an den Anfang der Funktion geschrieben. Diese Zeile beinhaltet eine Anweisung, die den Aufrufzähler für diese Funktion um eins erhöht. Nach dem Beenden der Zielanwendung wird die Anzahl der Funktionsaufrufe in der Datenbasis aktualisiert. Die Datenbasis ist persistent und kann somit Daten über beliebig viele Ausführungen der Zielanwendung sammeln.

Im dritten und letzten Schritt werden dann durch die Klasse `acover.HTMLConverter` auf Basis der Daten aus der XML-Datei die Maßzahlen berechnet und übersichtlich formatiert in eine HTML-Datei ausgegeben. Die generierte HTML-Datei ist hierarchisch aufgebaut und beinhaltet zuerst die Maßzahlen für die Paketüberdeckung. Zu jeder Klasse eines Paketes wird die Methodenüberdeckung berechnet und für jede Methode einer Klasse wird die Anzahl ihrer Aufrufe angegeben.

6 Systementwicklung als Kombination von Reverse Engineering und Prototyping

Mit zunehmender Größe eines Softwareprojektes nimmt ein stukturiertes, einem Plan unterliegendes Vorgehen immer größere Bedeutung ein. Um den Entwicklungsprozess in dieser Hinsicht zu unterstützen, wurden daher verschiedene Modelle entwickelt. Am weitesten verbreitet ist das klassische Wasserfallmodell. Es unterteilt die Entwicklung in die Hauptteile Anforderungsanalyse, Spezifikation, Systementwurf, Implementation, Testen und Wartung. Allerdings sind Situationen vorstellbar, in denen die Verwendung eines alternativen Entwicklungsmodells deutlich effizienter sein kann. Ein typischer Bereich für das Auftreten dieser Fälle sind Situationen, bei denen das zu entwerfende Softwaresystem keine Neuentwicklung darstellt.

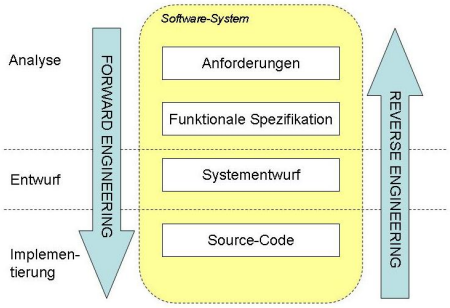


Abbildung 11: Funktionsprinzip des Reverse Engineering

So sind vielerorts Softwaresysteme im Einsatz, die regelmäßiger Anpassung und Wartung unterliegen. Durch diese iterative Adaption bedingt, findet in vielen Fällen eine kostenintensive Weiterentwicklung statt, die die ursprünglich bestehende Softwarearchitektur verschlechtert bzw. verkompliziert. So bildete sich in den 1980er Jahren der Begriff der „Krise in der Software-Wartung“. In Folge dessen wurden Modelle gesucht, die eine Lösung dieses Problems aufzeigten. In dieser Zeit entstand der Begriff des Reverse Engineerings.

Damit ein System nicht vollständig neu entwickelt werden muss, wird beim Reverse Engineering versucht, ein bestehendes System zu analysieren. Ziele sind sowohl die Identifikation von Systemkomponenten und deren Beziehungen untereinander, als auch die anschließende Darstellung des untersuchten Systems auf einem anderen, höheren Abstraktionsniveau. Ausgehend von den in dieser Phase gewonnenen Spezifikationen kann die Software anschließend mittels Methoden des klassischen Forward Engineerings reimplementiert werden.

Für ATOSj erschien dieser Ansatz sehr geeignet, da es mit Marathon und ATOS gleich zwei verschiedene Vorlagen gab. Aus beiden ließen sich höchstwahrscheinlich Ideen und

Ansätze wieder verwenden, aufgrund ihrer Differenzen jedoch immer nur in Teilen.

Noch vor Beginn der eigentlichen Arbeiten an ATOSj zeigte sich jedoch ein weiteres Problem. Der gewählte Themenkomplex der „Entwicklung eines Tools für den automatisierten Regressionstest unter Java“ war sehr weit gefasst. Da wir noch keinerlei Erfahrungen auf diesem Gebiet hatten, war es zunächst unklar, wie die gestellten Probleme zu lösen seien. Daher war eine Abschätzung des jeweils einzuplanenden Zeitaufwandes nicht bzw. nur eingeschränkt möglich. Aufgrund dieser Ungewissheit war es uns vor Beginn der Entwicklungsarbeit nicht möglich abzuschätzen, welchen Funktionsumfang die anzustrebende Endversion haben sollte.

Der unbekannte Weg der technischen Umsetzung und des damit verbundenen Zeitaufwandes machten somit eine Entwicklung nötig, bei der der bisherige Entwicklungsstand in regelmäßigen Abständen kontrolliert und bewertet werden kann. In der Softwareentwicklung ist diese Vorgehensweise allgemein als Prototyping bekannt. Beim Prototyping wird die Entwicklung eines Softwaremodells inhaltlich in einzelne Schritte geteilt. Dabei werden die einzelnen Teile jeweils mittels klassischen Methoden des Software-Engineerings entwickelt. Die Lösung eines dieser Teilabschnitte wird Prototyp genannt. Die inhaltliche Trennung kann dabei vertikal oder horizontal erfolgen. Beim vertikalen Prototyping werden spezielle Teile des Systems ausgewählt und zunächst vollständig implementiert. In ATOSj könnten diese Teile beispielsweise der Recorder und der Player sein. Dem gegenüber steht das horizontale Prototyping, bei dem das Gesamtsystem evolutionär weiterentwickelt wird, also jeweils nur bis zu einer gewissen Detailebene. Da das System in den weiteren Schritten erweitert wird, ist es dabei dringend notwendig, dass diese Ebene möglichst vollständig und so weit wie möglich vorausschauend entwickelt wird. Des Weiteren lässt sich das Prototyping dahingehend unterteilen, welchem Zweck die einzelnen Prototypen dienen. Für ATOSj spielte dabei das evolutionäre Protoyping eine Rolle. Dabei wird der Prototyp in jeder Phase einen Schritt weiterentwickelt und dabei lauffähig gehalten. Nach der Bewertung der erreichten Ziele und der aufgewendeten Zeit werden anschließend die Ziele für den nächsten Prototyp definiert. Zwei weitere Varianten, die bei der Entwicklung von ATOSj jedoch keine Rolle spielten, sind das experimentelle und das explorative Prototyping. Beim experimentellen Prototyping werden zu Forschungs- und Testzwecken Wegwerf-Prototypen erstellt, um die gemachten Erfahrungen anschließend ins eigentliche System einzubringen. Das explorative Prototyping überschneidet sich teilweise mit dem evolutionären Prototyping. Es dient vornehmlich zur Klärung von Machbarkeit und Nützlichkeit von bestimmten Systemideen. Die erstellten Prototypen können somit weiterverwendet werden, müssen es jedoch nicht.

Bei der Entwicklung von ATOSj musste zu Beginn entschieden werden, welche Eigenschaften in der späteren Eigenentwicklung aus den bekannten mit ATOSj verwandten Projekten übernommen werden sollten. Ursprüngliche Aufgabenstellung war die Erweiterung des lehrstuhleigenen Testprogramms ATOS auf Java. Daher stand bereits im Vorfeld der Entwicklung fest, dass als Skriptsprache HTS genutzt werden wird. In ATOS ist es möglich mehrere HTS-Befehle zunächst in einem Skript zusammenzufassen. An-

schließend ist die Kombination der Skripte zu so genannten Paketen möglich. Intention dieser Struktur ist, dass HTS-Befehle, die einen logisch zusammenhängenden Aktionsschritt wie etwa das Eintragen von Werten in eine Maske bilden, zunächst als solcher markiert und modular verfügbar gemacht werden. In einem Paket können diese Skripte danach wiederverwendet und zu komplexen Programmabläufen kombiniert werden. Da uns diese Strukturierung als sehr sinnvoll und praktisch erschien, entschlossen wir uns, sie logisch zu übernehmen und in ATOSj nachzubilden. Daher bilden auch in ATOSj die Objekte Befehl, Skript und Paket den Kern des zu verwaltenden Datenmodells.

Da uns sowohl für das Open-Source-Projekt Marathon als auch für ATOS der Quellcode zur Verfügung stand, konnten wir ebenfalls beide Architekturen bewerten. Die Analyse zeigte eine deutlich größere Strukturiertheit in der Architektur Marathons. In Marathon war beispielsweise jeder Befehlstyp durch eine eigene Ableitung einer allgemeinen Befehlsklasse realisiert. In ATOS werden sämtliche Befehle in einer Klasse implementiert. Dies hat sowohl einen sehr großen Umfang als auch eine hohe zyklomatische Komplexität der Klasse zur Folge. Das Design von Marathon wirkt gegenüber dem von ATOS hier deutlich durchdachter.

Auch die Funktionen des Players und des Recorders sind in Marathon jeweils gut strukturiert in einem separaten Paket implementiert. In ATOS sind diese nicht als eigenständige Komponenten identifizierbar, sondern Teil von umfangreichen, allgemeiner gehaltenen Klassen. Da jedoch das von ATOS umgesetzte Funktionsprinzip wie in Kapitel 5.7 besprochen, in ATOSj nicht umgesetzt werden kann, wäre eine mögliche Abwandlung von in Marathon umgesetzten Ideen ohnehin deutlich vorteilhafter. Ein Studium der Quellen ergab, dass die Marathon-Implementation ebenfalls die von uns bereits vorab ausgwählten Java Accessibility Utilities verwendete.

6.1 1. Prototyp

Basierend auf diesen gesammelten Informationen entstand zu diesem frühen Zeitpunkt ein erster Protyp, der allerdings kaum über das praktische Nachweisen der bereits in der Theorie festgestellten Machbarkeit hinausging. So war es uns mittels dieser Studie möglich, den Wert eines Textfeldes in einem festgelegten Testobjekt zu verändern. Das verwendete Testobjekt war das „Simple Widgets“ genannte Demonstrationsobjekt der Marathonentwickler. Es besteht aus einer so einfach wie möglich gehaltenen Oberfläche, die fast alle Swing-Standardelemente enthält.

Dieser erste rudimentäre Prototyp verfügte über keinerlei eigene GUI, und war daher nicht interaktiv. Er war jedoch sehr nützlich, da er, obwohl er sehr wenig Funktionalität in einer scheinbar unnötig komplizierten Architektur lieferte, eine hervorragende Basis für eine Erweiterung bzw die eigentliche Implementation lieferte. Zusammen mit dieser Version wurde auch die Entwicklungsinfrastruktur aufgebaut. Wir legten uns auf Eclip-

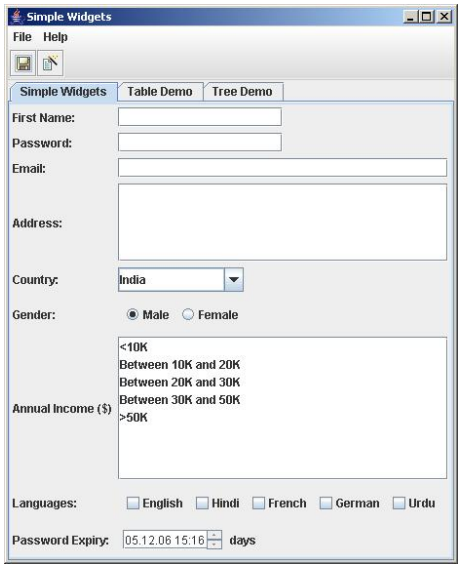


Abbildung 12: Testobjekt Simple Widgets

se als gemeinsame Entwicklungsumgebung fest, und zur Ermöglichung einer parallelen Entwicklung im Team wurde das ATOSj-Projekt in das bereits existierende lehrstuhleigene CVS aufgenommen. In diesem CVS waren bereits alle Projekte enthalten, die in Zusammenhang mit der Arbeit am XCTL-Projekt entstanden sind, unter anderem auch ATOS.

6.2 2. Prototyp

Da durch den ersten Prototyp eine mögliche Interaktion nachgewiesen wurde, konnte mit der eigentlichen Arbeit an ATOSj begonnen werden. Im ersten Schritt wurde dazu eine einfache GUI entwickelt, die von der Fensteraufteilung der von ATOS entsprach. Zur Vorbereitung des Ausführens von Skripten entstand zudem eine erste Version des HTSParsers. Da die einzelnen HTS-Befehle jedoch noch nicht implementiert waren, übernahm der Parser in dieser Version auch die Konstruktion der einzelnen HTS-Befehle. Eine erste Implemtentation der HTS-Befehle selber entstand auch zu diesem Zeitpunkt, wenn auch noch nicht mit allen Funktionalitäten. So erfolgte beispielsweise an der bedeutendsten HTS-Befehlsklasse HTSAction ein fortwährendes Anpassen und Erweitern. Die Version dieses Prototyps war gekennzeichnet von direkten Manipulationen an den jeweiligen Oberflächenobjekten. Die Wrapperklassen für die GUI-Elemente waren ebenfalls noch nicht vollständig, da wir zunächst unseren Schwerpunkt auf eine vollständige Implementation der HTSBefhle legten. Die Interaktion war zunächst weiterhin nur auf die Elemente Textfield und Button beschränkt. Der 2. Prototyp war durch das parallel ablaufenden Reverse Engineering stark von ATOS und Marathon geprägt. So war auch die erste Version

der HTSPlayers von Marathon geprägt. So liefen ATOSj und das Testobjekt bereits in separaten Prozessen, und die Kommunikation erfolgte in nur leicht angepasster Variante wie in Marathon mittels der Hilfsbibliothek RMI-Lite.

6.3 3. Prototyp

Mit dem dritten Prototyp wurde das Reverse Engineering von ATOS und Marathon beendet. Letzter Einfluss von ATOS war die Aufnahme der Testpakete in ATOSj. Durch die unserer Meinung nach unpassende Struktur von ATOS wurde allerdings nur die grundsätzliche Idee übernommen, jedoch keinerlei Ansätze der Umsetzung. Von Marathon wurde der Ansatz der Java Accessibility Utilities für die Erstversion des Recorders übernommen, der in diesem Prototyp zum ersten Mal enthalten war. Neu in dieser Version hinzugekommen waren auch Wrapperklassen für sämtliche in Swing existierenden Basiselemente. Dadurch ließen sich in dieser Version erstmals umfangreichere, sinnhaftere Skripte ausführen. Mit den in dieser Version hinzugefügten Funktionen waren sämtliche sinnvoll erscheinenden Prinzipien der beiden Vorlagen eingearbeitet. Dies und die Tatsache, dass wir durch die bis dahin gesammelten Erfahrungen die technischen Möglichkeiten besser einschätzen konnten, führten dazu, dass die weitere Entwicklung sich auf eigene Verbesserungen und Erweiterungen von ATOSj konzentrierten. So wurde bereits in dieser Version die Kommunikation mit dem Testobjekt mittels RMI-Lite durch eine eigene unidirektionale RMI-Lösung ersetzt, u.a. um eine spätere Erweiterung des Kommunikationsinterfaces zu ermöglichen. Weitere wichtige Änderung dieser Version war die Vereinfachung des HTSParsers. So wird die Erstellung der eigentlichen HTS-Befehle nun von dem zur Befehlsklasse zugehörigen Konstruktor übernommen. Dies vereinfacht den HTSParser deutlich und macht eine leichtere Anpassung in der Konstruktion der Befehle möglich, falls die HTS-Syntax geändert wird. Der HTSParser übernimmt von nun an lediglich die Funktion eines Scanners zuzüglich der Identifikation des Befehlstyps.

6.4 4. Prototyp

Wie bereits angedeutet erfolgte in dieser Version eine umfassende Änderung der bisher von ATOS und Marathon übernommenen Ideen. Die wichtigste konzeptionelle Anpassung wurde an der Syntax der Skriptsprache HTS vorgenommen. Die Syntax der Befehle wurde vereinheitlicht und der Sprachumfang erweitert. Außerdem kann der Nutzer nun eigene Aktionstypen an so genannten Custom-Components definieren. Dazu muss er lediglich eine eigene Wrapperklasse zur Interaktion mit der neuen Komponentenart definieren. Neben der Anpassung des HTSParsers war somit auch die Integration der Schnittstellen für die Wrapperklassen eine bedeutende Änderung des 4. Prototyps gegenüber seinem Vorgänger. Um eine höhere Stabilität der Verbindung mit dem Testobjekt zu gewährleisten, wurden die zugehörigen Interfaces auf bidirektionale Kommuniktion umgestellt.

Diese neue Version hat fast nichts mit der Marathonimplementation gemein, obwohl diese zunächst der Ausgangspunkt war. Mit den nun gegebenen erweiterten Möglichkeiten wurde auch der Recorder angepasst und vervollständigt. Durch die Änderung der Syntax von HTS wurden auch Befehle für Tabellen und Bäume möglich. Die Wrapperklassen für diese beiden zugehörigen Swing-Components konnten somit auch implementiert und hinzugefügt werden. Weitere äußerlich sofort erkennbare Veränderung war die Erweiterung der ATOSj-GUI für eine erhöhte Benutzerfreundlichkeit. So wurden Tastenkombinationen für gängige Aktionen definiert und ein umfangreicher Copy & Paste Mechanismus hinzugefügt.

Mit dieser Version war somit eine erste unseren Ansprüchen genügende Version erstellt, die wir Professor Bothe und seinem Lehrstuhl vorstellten. Nach Auswertung der erreichten Ziele stellte sich die Frage, ob eine Erweiterung auf SWT-Oberflächen möglich wäre. Nach einiger Recherche stießen wir auf die Möglichkeit einer Implementation unter Zuhilfenahme der Bibliothek Javassist. Daher fiel die Entscheidung, eine Erweiterung für SWT zu implementieren.

6.5 5. Prototyp

Die Bibliothek Javassist ist lediglich eine allgemeine Hilfsbibliothek. Sie kann für vielfältige Zwecke dienen, auch für eine Implementation der von uns benötigten Zwischenschicht. Daher wurde Javassist zunächst als Hilfsmittel verwendet, diesen Layer zu entwickeln. In Anlehnung an den aus den Java Accessibility Utilities bekannten und genutzten SwingEventMonitor wurde diese Zwischenschicht SWTEventMonitor genannt. Nachdem diese fertig gestellt war, wurde zunächst erneut eine rudimentäre Testversion erstellt, um zu prüfen, ob sich die erdachte Variante auch umsetzen ließ. Der Nachweis dessen mittels der als Studie zu interpretierenden Rumpferweiterung gelang.

6.6 6. Prototyp, Betaversion

In einem letzten großen Entwicklungsschritt erfolgte anschließend die Erweiterung der bereits für Swing fertig gestellten Funktionalität auf das gesamte SWT-Paket. Die Erweiterung ließ sich nahtlos in das existierende Gerüst einfügen, da die vorhandene Architektur bereits für eine solche Erweiterung ausgelegt war. Einmal auf Javassist aufmerksam geworden, stellten wir die Mächtigkeit dieses Tools fest. Javassist lässt sich für diverse Zwecke nutzen. So entwickelten wir als letzte ATOSj-Erweiterung das Modul ACover, das eine Überdeckungsanalyse realisiert. Dank ACover lassen sich also durch den Test am ausgeführten Programm Rückschlüsse auf die Qualität der Implementation ziehen.

Nach Fertigstellung dieser Version folgte eine umfassende Phase des Testens. Da wir

zum Zeitpunkt der Swing-Implementation noch deutlich weniger Erfahrungen mit dem Arbeitsgebiet hatten, wurde bereits während der Entwicklung sehr viel getestet. Als Testobjekt diente uns das bereits erwähnte „Simple Widgets“ Programm, das Marathon beiliegt. Für das Testen von SWT existierte eine derartige Vorlage nicht. Zeitgleich mit der Testphase entstand jedoch im Zusammenhang mit der Tätigkeit von Volker Janetschek am Lehrstuhl von Professor Bothe ein Programm zur Seminarorganisation. Dieses wurde in SWT implementiert und bildete somit das ideale Testobjekt. Der kombinierte Einsatz beider Programme zeigte uns sowohl Fehler in der Seminarorganisation als auch Fehler in ATOSj auf.

Während der Arbeit am schriftlichen Teil kommentierte Nicos Tegos die Quelltexte zusätzlich und komplettierte das bereits begonnene Javadoc. Dadurch und durch die hervorragende Struktur der Klassen denken wir, dass der Quellcode trotz seines erheblichen Umfangs auch für Neueinsteiger beherrschbar ist.

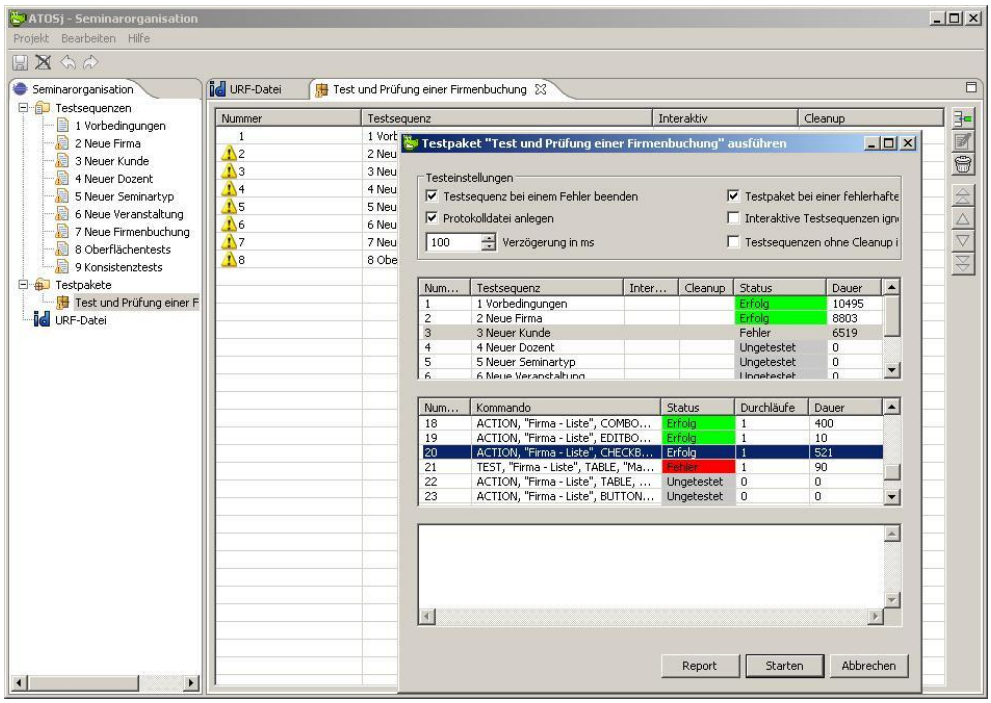


Abbildung 13: Programmansicht ATOSj

6.7 Bewertung

Rückblickend lässt sich sagen, dass die Kombination von Reverse Engineering und Prototyping sehr gut funktioniert hat. Die Stärken beider Vorlagen waren sehr verschieden verteilt. Da ATOS der Ursprung für die Idee von ATOSj war, haben wir versucht, einige

Eigenschaften zu übernehmen. Der interne Aufbau von ATOS war allerdings häufig nicht strukturiert genug. Es existieren zu wenig Klassen, mit zu großem Umfang. Die Lesbarkeit und Verständlichkeit der ATOS-Architektur leidet durch derartige Unausgereiftheiten an mehreren Stellen. Marathon hingegen zeichnete sich durchgängig durch eine große Durchdachtheit in der Architektur aus. Der Grund hierfür ist wohl in der Open-Source-Idee zu suchen, bei der sich viele Entwickler schnell in die Architektur hineindenken müssen, was einen sauber geplanten Entwurf bedingt. Wenn man weiß, dass Marathon Einfluss auf ATOSj hatte, lassen sich durchaus Ähnlichkeiten in der Klassenhierarchie erkennen. Dies liegt in der frühen Entwicklungsphase begründet. Bedingt durch die in dieser Zeit gemachten Erfahrungen, haben wir allerdings den Großteil der von Marathon genutzten Ideen im Laufe der Entwicklung durch eigene Ansätze und Verbesserungen ersetzt. Die Funktionsweisen sind somit stark unterschiedlich. Der Aufbau weist jedoch einige Ähnlichkeiten auf.

Das Prototyping war durchaus geeignet, um den Einstieg in das Thema zu finden. Es war vorteilhaft, bereits in frühen Phasen der Entwicklung eine lauffähige Version zu haben. So ließen sich Grundideen bereits früh auf ihre Funktionsweise testen. Auch unterstützte es die Arbeit im Team, da leichter war, neue Meilensteine zu setzen und sich aufeinander abzustimmen. Als Nachteil des Prototypings wird häufig die verminderte Arbeitsgeschwindigkeit angesehen. Die Arbeit an ATOSj dauerte etwa ein Jahr. Dies mag zunächst als sehr viel erscheinen, betrachtet man jedoch den Umfang der Arbeit, ist dies durchaus angemessen.

7 Testobjekt Seminarorganisation

7.1 Begründung der Wahl des Haupttestobjekts

Der Test eines Programms, dessen Funktion darin besteht, andere Oberflächenprogramme zu testen, kann logischerweise nicht ohne ein solches Testobjekt erfolgen. Da diese Diplomarbeit unter der Führung des Lehrstuhls für Softwaretechnik entstand, wurde als primäres Testobjekt ein Programm gesucht, das dem Lehrstuhl weitere Nutzungsmöglichkeiten bieten würde.

Wichtigste Veranstaltung des Lehrstuhls im Hauptstudium ist der Halbkurs Software-Engineering. In dieser Veranstaltung werden den Studenten die Methoden, die bei der Entwicklung von größeren Softwareprojekten angewendet werden, näher gebracht. Die Vorlesung handelt den gesamten Entwicklungsprozess von Planung über Organisation und Entwicklung hin zur Qualitätssicherung ab. Dabei orientiert sich die Vorlesung an dem Buch „Lehrbuch der Software-Technik“ von Helmut Balzert [12]. Dieses Buch bietet den Studenten eine zusätzliche Hilfe bei der Nacharbeit der Vorlesung. Zum besseren Verständnis stützt sich der Autor dabei regelmäßig in seinen Ausführungen auf die so genannte „Fallstudie Seminarorganisation“, einem Programm zur Kunden- und Seminarverwaltung einer fiktiven Firma. Dem Buch beigelegt ist eine CD-Rom, die bereits eine komplette Implementation einer möglichen Lösung für das Problem der Seminarorganisation als ausführbares Windows-Programm beinhaltet.

Da des Öfteren auch in der Vorlesung direkt auf dieses Programm Bezug genommen wird, wünschte sich der Lehrstuhl eine eigene Implementierung. In den Jahren 2004/2005 entstand an der Universität von Novi Sad (Serbien) im Rahmen einer europäischen Zusammenarbeit eine Java-AWT Lösung der Seminarorganisation. Da diese Variante allerdings fehlerbehaftet war, und eine nach Ansicht des Lehrstuhls ungeeignete Menüführung aufwies, entschloss man sich, eine komplett eigene Version zu entwickeln. Diese Version sollte dem Look-and-Feel der Balzert-Version nachempfunden werden. Als Oberflächenbibliothek wurde SWT gewählt.

Die Entwicklung dieser Version fiel zeitlich mit der Implementation von ATOSj zusammen, so dass beide Programme beim Testen voneinander profitieren konnten. Zum einen konnte ATOSj im Sinne seiner eigentlichen Programmierung genutzt werden, um das Testen der Seminarorganisation weitestgehend zu automatisieren. Zum anderen war das Erstellen, Editieren und Ausführen der Skripte auch für ATOSj ein nahezu vollständiger Funktionstest, was zu einer Fehlerbereinigung darin führte.

7.2 Beschreibung der Seminarorganisation

Die Seminarorganisation in Ihrer derzeitigen Form benötigt zwingend eine installierte MySQL Datenbank. Ohne diese bricht bereits der Start des Programms ab, da es keine sinnvolle Nutzung für das Programm gibt, wenn keine Daten verwaltet werden können. Wichtigster Bestandteil des Programms sind Fenster, die jeweils der Aufnahme bzw Editierung eines Datenbankobjektes dienen. Objekte dieser Art sind beispielsweise Kunden und Dozenten, aber auch Firmen, Veranstaltungen und Buchungen für diese.

Die Hauptaufgaben des Programms sind die korrekte Darstellung der Datenbankobjekte und die Übergabe der gemachten Änderungen an diese. Die Struktur der Implemtentation weist eine strikte Trennung von Oberflächenklassen und Klassen zur Anbindung der Datenbank auf. Dabei haben die Klassen für die Datenbank lediglich die Aufgabe der Übergabe der Daten. Die Wahrung der Konsistenz der Daten übernimmt die Datenbank.

7.3 Einschränkungen bezüglich ATOSj bei der Zusammenarbeit mit der Seminarorganisation

Bestandteil des Seminarorganisationsprogramms sind diverse Fenster. Diese Fenster können klassifiziert werden in modale und nicht-modale Fenster. Der Unterschied in diesen beiden Klassen liegt darin begründet, dass bei nicht-modalen Dialogen jedes Fenster zu einem beliebigen Zeitpunkt durch einen einfachen Mausklick aktiviert werden kann. In einer modalen Hierarchie muss immer erst das aktuelle Fenster geschlossen werden, bevor auf das darunter liegende wieder zugegriffen werden kann.

Zu Problemen bei der Zusammenarbeit mit ATOSj kommt es bei einer nicht modalen Fensterhierarchie, bei gleichzeitiger Gleichbenamung dieser Fenster. ATOSj kann dann nicht mehr nachvollziehen, welches das momentan aktivierte Fenster ist, und es kann zu Inkonsistenzen bzgl Capture & Replay kommen. Als Beispiel sei hier nur der Fall genannt, dass das Editierfenster für ein beliebiges Objekt doppelt geöffnet ist. Die Fenster sind in diesem Falle äußerlich nicht mehr voneinander unterscheidbar.

Der Fehler sollte hier jedoch nicht bei ATOSj, sondern im Testobjekt gesucht werden. Denn nicht nur für ATOSj sind die Fenster nicht mehr unterscheidbar. Auch der Nutzer kann die beiden Fenster lediglich an Ihrer Positionierung auf dem Bildschirm unterscheiden. Desweiteren scheint der zusätzliche Nutzen einer solchen Möglichkeit für den Nutzer fraglich. Zu guter letzt sind die mit dieser Hierarchie verbundenen möglichen Wechselwirkungen auch aus programmiertechnischer Sicht problematisch anzusehen. Daher sollte also eher das Fensterkonzept der Seminarorganisation überdacht werden, als einen weiteren Mechanismus für solche Fälle in ATOSj zu implementieren.

7.4 CalendarControl als Custom-Component

Wie bereits erwähnt unterstützt ATOSj die Aufnahme eigener, so genannter Custom-Components (siehe Abschnitt 5.4) für diverse Anwendungsfälle. Zur Demonstration dieses Mechanismus' wurde hier die von org.eclipse.swt.widgets.Composite abgeleitete Klasse CalendarControl gewählt. Diese Klasse stellt eine Funktionalität zur Auswahl eines Datums dar. Die Wahl des Datums ist optional. Ist kein Datum gewählt, so ist die Checkbox des Composites deaktiviert. Zur Auswahl wird der Button für den Datumsdialog an der rechten Seite des Controls gewählt. Im Anschluss daran erscheint ein kleines Popup, das einen SWTCalendar enthält.

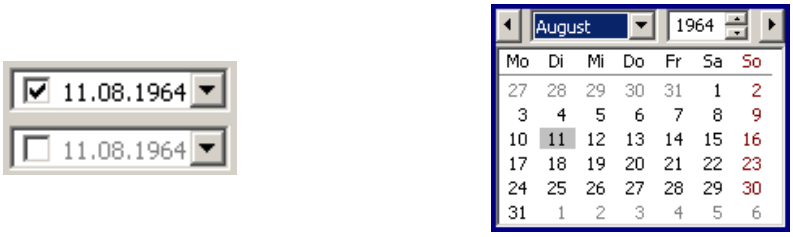


Abbildung 14: aktiviertes bzw deaktiviertes CalendarControl und dazugehöriges Popup

Da sämtliche Bestandteile des Controls als auch des dazugehörigen Popups lediglich Standardelemente sind, wäre es möglich, sämtliche Aktionen auf diesem Control auch ohne die Einrichtung eines Custom-Components vorzunehmen. Allerdings würde das Einstellen eines Datums bis zu 5 Standard- HTS-Befehle benötigen. (i) Checkbox anwählen ii) Popup öffnen, iii) Jahr auswählen, iv) Monat auswählen, v) Tag auswählen). Um dies übersichtlicher zu gestalten, wurde ein Custom-Component eingerichtet, das eine deutlich semantikbezogenere Syntax aufweist, als es die 5 Befehle tun, die ohne das Custom-Component entstehen würden. Der Implementation des CalendarControlWrappers (siehe Anhang C.3) ist zu entnehmen, dass drei Properties für das CalendarPopup definiert wurden. Zum einen kann mittels des Properties „open“ das Popup zum Vorschein (für true) gebracht, oder geschlossen werden (für false). Mittels des zweiten definierten Properties „date“ kann der ausgewählte Wert direkt gesetzt werden. Erlaubte Werte sind hier sämtliche Datumsangaben, die sich an das Format „dd.MM.yyyy“ halten. Das letzte Property „activ“ dient der Manipulation der Checkbox des CalendarControls. Wie bereits beim Property „open“ sind lediglich die Werte „true“ und „false“ zulässig.

7.5 Testfallspezifikation anhand von Use-Cases

Die Aufgaben der Seminarverwaltung sind im dazugehörigen Pflichtenheft [10] als Use-Cases spezifiziert. Die einzelnen Use-Cases überschneiden sich hierbei jedoch häufig. Zur Illustration wurde daher ein Testpaket gewählt, das die Aufgabe hat, eine Firmenbuchung zu erstellen. Dieser Anwendungsfall eignet sich daher besonders, weil fast alle Fenster der

GUI daran beteiligt sind. Sämtliche beteiligten Objekte sollen neu erstellt werden. Hier die Übersicht der dazu nötigen Schritte:

- neue Firma anlegen
- neuen Kunden anlegen und als Mitarbeiter der Firma registrieren
- neuen Dozenten anlegen
- neuen Seminartyp anlegen
- neue firmeninterne Veranstaltung anlegen und den Dozent als Seminarleiter und Referenten eintragen
- neue Firmenbuchung zwischen der Veranstaltung und der Firma erstellen

Anschließend wird überprüft, ob die getätigten Änderungen korrekt übernommen wurden, ob die Zustände der grafischen Elemente den Erwartungen entsprechen, und ob referentielle Integritäten im Programm beachtet werden.

Vorab noch eine Anmerkung zur Eingabereihenfolge in den Dialogen: Da in der Theorie exponentiell viele unterschiedliche Abfolgen des Editierens vorkommen, und diese unmöglich alle getestet werden können, wurden sämtliche der nun folgenden Testskripte so entworfen, dass zwar Eingaben in fast allen Feldern erfolgen, diese jedoch in einer einfachen, kanonischen Reihenfolge vorgenommen werden. Der Grund hierfür liegt darin, dass somit der Dialog sowohl ausführlich getestet wird, als auch, dass damit der wohl häufigste Anwendungsfall abgedeckt wird. Des Weiteren erscheint die Eingabereihenfolge nicht derart wichtig, weil das Testobjekt dahingehend entwickelt wurde, dass sowohl das Prüfen von Konsistenzbedingungen als auch die Manipulation der Datenbank erst dann erfolgt, wenn der jeweilige Dialog zur Übernahme der getätigten Daten aufgefordert wird.

7.5.1 Anlegen der neuen Objekte

Aufgabe

Diese Testskripte öffnen die jeweiligen Dialoge zur Aufnahme des in der Datenbank neu anzulegenden Objektes, tätigen die Eingaben in den Dialogfeldern und sorgen durch das Schließen des Dialogs dafür, dass das Testobjekt die Daten in die Datenbank übernehmen soll.

Bemerkungen

Bestandteil dieser Scripte sind u.a. Befehle zur Steuerung des vorhin bereits erwähnten Custom-Controls CalendarControl. Dementsprechend setzt beispielsweise im Script zur Aufnahme einer neuen Firma der Befehl

```
ACTION, ‘‘Neu - Firma*‘‘, CALENDAR, ‘‘CpBirthDayCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘26. 03. 1955‘‘
```

das Datum des CalendarControls, das für die Einstellung des Geburtsdatums der Kontaktperson verantwortlich ist, auf den 6. April 1971.

7.5.2 Test der zuvor getätigten Eingaben

Aufgabe

Diese Skripte sind dafür verantwortlich, das Programmverhalten bei bereits existierenden Daten in der Datenbank zu testen. Zustände von Oberflächenobjekten werden abgefragt, und mit einem Erwartungswert verglichen. Danach werden bewusst Objekte gelöscht, die von anderen referenziert werden. Anschließend wird das Konsistenzverhalten des Programms getestet. Auch hier geht es nicht um einen vollständigen Test, sondern es wird versucht, typische Anwendungsfälle zu simulieren. Teilweise sind solche Tests auch schon im Eingabeabschnitt des Skripts erfolgt. Beispielsweise bei der Auswahl der Firma im Kontext der Arbeitgeberbeziehung des Kunden.

In diesem Teil des Testpaketes soll es einmal nicht nur um den Test des Testobjektes, sondern auch um die Demonstration und den Test der Möglichkeiten von ATOSj gehen. Bestandteil dieses Scriptes sind daher Testbefehle, die möglichst unterschiedliche Eigenschaften an möglichst verschiedenen Objekten testen.

Bemerkungen

Da das Testobjekt bereits vor dem Verfassen der Skripte einigen manuellen Testläufen unterzogen wurde, sind bei den reinen Funktionstests keine Fehler mehr gefunden worden. Jedoch wurde beim Test des Konsistenzverhaltens ein Widerspruch zum erwarteten Programmablauf festgestellt. Im letzten Skript wird mittels der Befehle

```
ACTION, ‘‘Seminartyp - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Seminartyp - Stammliste‘‘, BUTTON, ‘‘DeleteItem‘‘, SELECT
```

der zuvor eingegebene Seminartyp wieder gelöscht. Ergebnis dieser Aktion sollte sein, dass sowohl alle Veranstaltungen dieses Seminars, als auch alle Buchungen die zu diesen Veranstaltungen zugeordnet sind, gelöscht bzw. nicht mehr angezeigt werden.

Dieses Verhalten wurde getestet, und sowohl die Veranstaltung als auch die Buchung zu

dieser werden weiterhin angezeigt. Allerdings bewirkt eine einfache Aktualisierung der Liste, beispielsweise mittels

```
ACTION, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, BUTTON, ‘‘UpdateItem‘‘, SELECT
```

dass der anschließende Test von

```
TEST, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 0, EQ
```

wieder das korrekte Ergebnis liefert. Offensichtlich werden die Daten aus der Datenbank entfernt, allerdings ohne ein Update der zugehörigen Fenster. Hier schließt sich nun der Kreis des fortwährenden Testens und Weiterentwickelns. Der durch die Testumgebung gefundene Fehler wird nun im darauf folgenden Zyklus behoben, um anschließend erneut sämtliche Testskripte auf Ihren korrekten Ablauf zu testen. Das Problem der fehlenden Aktualisierung einiger Fenster könnte beispielsweise durch einen Listener-Mechanismus ausgeräumt werden: Für jede Datenbanktabelle wird ein Verwaltungsobjekt geschaffen, bei dem sich sämtliche offenen Fenster als Listener registrieren. Erfolgt nun eine Operation auf dieser Tabelle in der Datenbank, so benachrichtigt das entsprechende Verwaltungsobjekt sämtliche registrierte Fenster, welche sich daraufhin aktualisieren.

Diese Anpassung kann durchaus tiefere Strukturanpassungen erforderlich machen. Daher ist es stets notwendig, auch jene Skripte, die in der Vorgängerversion bereits problemlos liefen, erneut zu testen. Durch die ständigen Wiederholungen kann der Aufwand daher schnell ansteigen. Der Nutzen der Automation dieser Abläufe wird in solchen Fällen besonders offensichtlich.

8 Fazit und Ausblick

8.1 Vergleich mit Marathon

Die Entwicklung von ATOSj orientierte sich an den bereits bestehenden Systemen ATOS und Marathon. Beide Programme sind für den Einsatz auf dem Gebiet des Oberflächentests entwickelt worden, unterscheiden sich jedoch konzeptuell in einigen Punkten voneinander. ATOSj nutzt die Stärken beider Programme, um sie in einem System zusammenzuführen. Aus ATOS wurden im Wesentlichen funktionale Anforderungen wie die Struktur eines Projektes oder die Definition von Testfällen mittels HTS übernommen. Aus Marathon wurde ein Teil der internen Struktur übernommen. Sowohl Marathon als auch ATOSj widmen sich dem Test von Java-Programmen und ähneln sich deshalb stark in ihren technischen Anforderungen. Aus diesem Grund konnte das Klassendesign von Marathon teilweise für ATOSj adaptiert werden. Trotz der ähnlichen internen Struktur beider Systeme gibt es jedoch einige Unterschiede in ihrer äußeren Funktionsweise. Diese Unterschiede sollen im Folgenden miteinander verglichen und bewertet werden.

8.1.1 Projektorganisation

Marathon: Ein Marathon-Projekt gliedert sich in 3 Hauptteile. Den ersten Teil bilden die Testfälle (Testcases). Sie enthalten Kommandos zur Manipulation und Zustandsprüfung für den Test einer Funktion des Testobjektes. Ein Testfall wird in Python abgefasst. Zum zweiten Teil gehören Sammlungen aufgezeichneter Aktionen (Capturescripts). Jedes Capturescript definiert eine eigene Python-Funktion, die in allen Testfällen eines Projekts genutzt werden kann. Den letzten Teil eines Marathon-Projektes bilden die Fixtures. Jedem Testfall muss ein Fixture zugeordnet werden. Es sorgt für den Start des Testobjektes und hat die Aufgabe, eine definierte Ausgangs- und Endsituation für die Ausführung eines Testfalles zu schaffen. Die Struktur eines Projektes spiegelt sich auch im Dateisystem wider, mit eigenen Ordnern für alle 3 Teile.

ATOSj: Ein ATOSj-Projekt besteht ebenfalls aus 3 Hauptteilen. Den ersten Teil bilden die Testsequenzen. Sie haben die gleiche Aufgabe wie die Testfälle in Marathon, werden jedoch in HTS spezifiziert. Zum zweiten Teil gehören die Testpakete. Ein Testpaket soll alle, unter einem bestimmten Aspekt zusammengehörigen, Testfälle zusammenfassen. Es kann beispielsweise alle Testsequenzen zu einem Anwendungsfall enthalten. Den dritten Teil bildet die Datenbasis über die GUI des Testobjektes. Alle Projektteile werden in getrennten Ordnern gespeichert. Außerdem gibt es noch weitere Ordner, die für spezielle Aufgaben genutzt werden können. Im Ordner „env“ beispielsweise, kann der Tester alle Dateien ablegen, die zur Konfiguration des Testobjekts benötigt werden.

Die Projektorganisation gestaltet sich in beiden Programmen in etwa gleich. Testfälle

werden in ATOSj und Marathon in speziellen Skripten beschrieben und gespeichert. In ATOSj sichern Testpakete die Wiederverwendbarkeit von Code und in Marathon bieten die Capturescripts ebenfalls die Möglichkeit zur Modularisierung. Sowohl in ATOSj als auch in Marathon werden projektrelevante Daten im Dateisystem abgelegt. Diverse Ordner sorgen für die Strukturierung der Daten. Ein ATOSj-Projekt definiert einige zusätzliche Ordner für spezielle Aufgaben. Diese sind dem Testsystem bekannt und können mit Hilfe spezieller Schlüsselwörter in Dateioperationen verwendet werden. Das erhöht die Portabilität, da auf absolute Pfadangaben verzichtet werden kann. In Marathon fehlt diese erweiterte Ordnerstruktur.

Benamung und Semantik einiger Projektteile in Marathon sind dem Testframework JUnit entlehnt. Es unterstützt und formalisiert den automatisierten Modultest (unit test) bzw. Komponententest in Java. JUnit arbeitet auf Quelltextebene und erfordert die Definition einer eigenen Testklasse für jede Komponente. Als Komponente gilt bei objektorientierten Programmiersprachen meist deren natürliche Abstraktionseinheit - die Klasse. Die Testklasse muss von der Klasse `junit.framework.TestCase` abgeleitet sein. Alle öffentlichen, nicht statischen und parameterlosen Methoden dieser Klasse, deren Namen mit „test“ beginnen, beschreiben einen Testfall. Weitere wichtige Bausteine einer Testklasse in JUnit sind die Methoden `setUp()` und `tearDown()`. Die Methode `setUp()` wird vor jeder Ausführung eines Testfalles gerufen und versetzt das Testobjekt in einen definierten Ausgangszustand. Als Pendant dazu existiert die Methode `tearDown()`, welche nach der Ausführung eines jeden Testfalles gerufen wird und die Aufgabe hat, alle verwendeten Ressourcen, wie z.B. erstellte Dateien, wieder freizugeben. Einen weiteren essentiellen Teil des JUnit-Testkonzepts bilden die so genannten assert-Methoden. Sie sind auch Teil einer Testklasse und vergleichen Soll- und Istwerte miteinander. Stimmen diese nicht überein, gilt der Testfall als fehlgeschlagen und die assert-Methode wirft eine Ausnahme vom Typ `junit.framework.AssertionFailedError`. JUnit bezeichnet diese Art von Fehler als *Failure*. Ein *Failure* ist eine Diskrepanz zwischen der funktionalen Spezifikation und dem tatsächlichen Verhalten eines Moduls bei der Ausführung. Zur Laufzeit eines Testfalls können aber auch Ausnahmen auftreten, die auf Programmierfehler in der Testmethode selbst oder im getesteten Modul zurückzuführen sind. Diese Art von Fehler wird in Abgrenzung zum *Failure* als *Error* bezeichnet. Testfälle können in einer Testsuite miteinander kombiniert und zur Ausführung gebracht werden. Dazu muss die Funktion `suite()` in der Testklasse implementiert werden [2]. Zur Verdeutlichung des Modultest mit JUnit, ist ein Beispiel für den Test der Klasse `java.util.Vector` im Anhang unter Listing C.1 zu finden.

Die Verwendung des spezifischen Vokabulars aus dem JUnit-Test scheint in Marathon nicht passend. Zum Einen erschließt sich einem Nutzer, der nicht mit JUnit vertraut ist, die Bedeutung von Begriffen wie Failure, Error oder Fixture nicht. Zum Anderen werden durch die Einführung von JUnit in den Oberflächentest zwei völlig unterschiedliche Teststrategien miteinander vermischt. Die Vorgehensweise bei der Erstellung von Testfällen in JUnit ist eine andere als bei einem Capture and Replay Werkzeug. JUnit propagiert den Test-First-Ansatz. Der Begriff Test-First entstammt dem Vorgehensmodell des leichtge-

wichtigen Entwicklungsprozesses XP (Extreme Programming) und wurde durch dessen Begründer Kent Beck geprägt. XP verzichtet auf umfangreiche Dokumentation und setzt dagegen mehr auf selbsterklärenden Code und auf den dokumentarischen Charakter des Testrahmens, der im Test-First-Verfahren entwickelt werden muss. Test-First bedeutet, dass zuerst ein Testfall, d.h. eine test-Methode einer Testklasse, geschrieben wird und erst danach die Implementation der eigentlichen Funktion erfolgt. Test und Implementation werden dabei Schrittweise um neue Funktionen erweitert und für bestehende verfeinert. Mit der Implementation wird erst fortgefahren, wenn alle Tests des Moduls erfolgreich verlaufenen sind. Beim oberflächenbasierten Regressionstest wird jedoch genau entgegengesetzt vorgegangen. Testfälle werden erst generiert nachdem eine Funktion des Testobjekts vollständig fertiggestellt und verifiziert wurde. Erst dann können die Methoden des Capture und des Replay angewendet werden. Auf Grund der Gegensetzlichkeit beider Strategien sollte Marathon auf einen direkten Bezug zu JUnit verzichten.

8.1.2 Testvor- und Testnachbereitung

Marathon: Die Idee der Vor- und Nachbereitung eines Testfalls wurde aus dem Modultest mit JUnit auf den oberflächenbasierten Test mit Marathon übertragen. Sie wird in den Fixture-Skripten aufgegriffen, die die Funktionen `setup()` und `teardown()` enthalten. Beide Funktionen haben die in JUnit definierte Bedeutung.

ATOSj: In ATOSj sind Vor- und Nachbereitung keine separaten Projektbestandteile sondern Teil einer Testsequenz. Zur Vorbereitung dienen alle Kommandos, die vor dem HTS-Kommando `START` stehen. An dieser Stelle können zum Beispiel Dateioperationen, wie das Kopieren von Konfigurationsdateien, ausgeführt werden. Für die Nachbereitung steht das Kommando `CLEANUP` zur Verfügung. Alle darauf folgenden Kommandos dienen dazu, einen definierten Endzustand zu schaffen, wie z.B. durch das Löschen von Konfigurationsdateien, die in der Vorbereitungsphase bereitgestellt wurden.

Die Fixture-Skripte in Marathon erlauben die Verwendung des Codes zur Vor- und Nachbereitung in unterschiedlichen Testfällen. In ATOSj ist das nicht möglich, da Vor- und Nachbereitung statisch in eine Testsequenz eingebunden sind. Die Trennung von Fixture und Testfall gestaltet Testfälle in Marathon außerdem übersichtlicher als in ATOSj.

8.1.3 Unterstützte Oberfläche

Marathon: Es werden fast alle Standard-Components der Bibliothek Swing für oberflächenbasierte Java-Programme unterstützt.

ATOSj: Es wird nicht nur die Bibliothek Swing sondern auch die relativ junge Bibliothek

SWT unterstützt.

Beide Programme realisieren eine umfangreiche Unterstützung der grafischen Elemente in den angebotenen Bibliotheken, denn jedes nicht unterstützte Component senkt den Nutzen des Testwerkzeuges. Verwendet ein Testobjekt ein nicht unterstütztes Component, ist das manuelle Eingreifen des Testers in den Testablauf erforderlich, da das Component direkt manipuliert werden muss. Eine solche Vorgehensweise widerspricht dem Prinzip des automatisierten Tests, weil sie zeitaufwändig ist und die genaue Reproduzierbarkeit des Testablaufs behindert. Die Zahl der unterstützen Elemente für die Bibliothek Swing ist bei Marathon und ATOSj in etwa gleich. Im Gegensatz zu ATOSj unterstützt Marathon zusätzlich das Auslesen und Manipulieren von Tabellenköpfen, die in vielen Programmen zur Sortierung der Tabelleninhalte genutzt werden können. Eindeutiger Vorteil von ATOSj gegenüber Marathon ist die Unterstützung der Bibliothek SWT. Sie macht ATOSj zukunftssicher, denn SWT gewinnt bei der Entwicklung grafischer Oberflächen einen immer größeren Marktanteil.

8.1.4 Aufbau einer Datenbasis über die GUI des Testobjekts

Marathon: Marathon enthält keine Datenbasis über die Components des Testobjektes.

ATOSj: Es wird eine Datenbasis angelegt, die eindeutige Bezeichner zu den Components des Testobjekts beinhaltet. Sie kann automatisch generiert und manuell bearbeitet werden. Ein Abgleich mit der Datenbasis verhindert die Verwendung ungültiger Components in den Testsequenzen. Des Weiteren dient die Datenbasis zur Unterstützung des Nutzers bei der manuellen Erstellung von Testsequenzen.

Die Verwendung der Datenbasis in ATOSj ermöglicht die Überprüfung der referentiellen Integrität von Kommandos, d.h. wenn in einem Kommando ein Bezeichner verwendet wird, der nicht in der Datenbasis enthalten ist, gilt es als fehlerhaft und darf nicht ausgeführt werden. Fehlt eine solche Prüfung, wie in Marathon, werden falsch referenzierte Components erst zur Laufzeit erkannt. Das verlangsamt die Testentwicklung, da eine wiederholte Ausführung des betroffenen Testskriptes nötig ist, um den Fehler zu diagnostizieren und dann zu beheben. Wird der Fehler schon vor der Ausführung gemeldet, kann er viel schneller beseitigt werden. Um der Fehleingabe von Bezeichnern vorzubeugen, wird dem Nutzer bei der Erstellung eines Kommandos in ATOSj eine Liste aller gültigen Bezeichner vorgeschlagen, aus der einer ausgewählt werden muss.
Während der Entwicklungsphase ist das Testobjekt häufigen Veränderungen unterworfen, dazu gehört unter anderem auch die Änderung von Fenstertiteln oder Beschriftungen. Damit der Tester schnell darauf reagieren kann, ermöglicht ATOSj die Änderung von Bezeichnern in der Datenbank. Diese wird automatisch für alle Kommandos eines Projektes übernommen. In Marathon fehlt eine solche Möglichkeit, deshalb müssen betroffene

Kommandos vom Tester per Hand gesucht und geändert werden. Im Gegensatz zum Automatismus in ATOSj ist das manuelle Vorgehen in Marathon fehleranfällig und langsam. Die Pflege der Datenbasis erhöht zwar den Verwaltungsaufwand für ein ATOSj-Projekt, macht das aber durch die genannten Vorteile wieder wett.

8.1.5 Anpassbarkeit

Marathon: Die Verwendung der Skriptsprache Python macht Marathon in höchstem Maße an die Anforderungen des Nutzers anpassbar. Der Tester kann das System durch die Definition eigener Funktionen beliebig erweitern. Beispielsweise gibt es in Marathon keine vordefinierte Funktion für den numerischen Sollwertvergleich, wie in ATOSj. Wird eine solche Funktion aber dennoch benötigt, so kann sie der Tester selbst implementieren wie das folgende Beispiel demonstrieren soll. Die neue Funktion `assertNum` basiert auf der Funktion `getText`, welche die aktuelle Beschriftung eines Components ermittelt. Der Text wird in einen ganzahligen Wert umgewandelt und mit dem Sollwert verglichen. Ist die Konvertierung des Textes in eine ganze Zahl nicht möglich, wird eine Exception geworfen. Stimmen der Sollwert und der ermittelte Wert nicht überein, wird ebenfalls eine Exception geworfen.

```
def assertNum(componentName, mustValue):
try:
    valStr = getText(componentName)
    num = int(valStr)
    if num != mustValue:
        raise Exception("Der Istwert: " + str(num)
        + " entspricht nicht dem Sollwert: " + str(mustValue))
except ValueError:
    raise Exception("Der Wert: " + valStr
    + " ist keine natürliche Zahl")
```

Listing 6: Numerischer Sollwertvergleich in Python

Marathon kann nicht nur um neue Funktionen sondern auch zur Unterstützung von Custom-Components erweitert werden. Zu diesem Zweck sind zwei Komponenten nötig. Zum einen eine Wrapperklasse für das Custom-Component und zum anderen ein Component-Resolver. Beide müssen vom Tester implementiert werden. Die Wrapperklasse bildet die Schnittstelle zwischen Marathon und dem Custom-Component. Sie muss alle gewünschten Nutzeraktionen auf diesem Component simulieren können, z.B. einen Mausklick. Alle Wrapperklassen müssen von einer Basisklasse abgeleitet sein, die durch Marathon bereitgestellt wird. Der Component-Resolver hat die Aufgabe zur Laufzeit den Components des Testobjektes die entsprechenden Wrapper zuzuordnen. Dazu ruft Marathon alle bekannten Component-Resolver nacheinander auf, bis ein Resolver gefunden wird, der einen passenden Wrapper liefert. Zuletzt wird der Standard Component-Resolver aufgerufen, der immer einen passenden Wrapper findet [18].

ATOSj: Die Anpassung von ATOSj an neue Anforderungen, wie die Verwendung von Custom-Components in einem Testobjekt, ist durch den Plug-In Mechanismus gewährleistet, welcher ausführlich im Kapitel 5.4 beschrieben wird.

Beide Programme bieten dem Nutzer die Freiheit, Grundfunktionalität zur Unterstützung von Custom-Components zu erweiteren. Das jeweilige Vorgehen ist dabei ähnlich und in seiner Effektivität vergleichbar. Marathon ist durch die Fähigkeiten von Python noch flexibler und legt mehr Verantwortung in die Hände des Testentwicklers. Der Kerngedanke beim Einsatz von HTS - Einfachheit - soll Fehler vermeiden, schränkt aber auch die Funktionalität auf die vordefinierten Kommandos ein.

8.1.6 Aufzeichnung von Testskripten

Marathon: Es wird die Möglichkeit geboten, zwei unterschiedliche Kategorien von Kommandos automatisch zu generieren. Zur ersten Kategorie zählen alle Kommandos, die die Simulation von Nutzeraktionen übernehmen. Zur zweiten Kategorie zählen alle Kommandos zur Zustandsprüfung von Components. Die Aufzeichnung von Nutzeraktionen erfolgt kontextabhängig, d.h. es werden Kommandos generiert, die spezifisch für das Component sind auf dem die Nutzeraktion ausgeführt wurde. Die Aufzeichnung einer Zustandsprüfung ist ebenfalls kontextsensitiv. Zu jedem Component des Testobjekts stellt Marathon eine Liste aller prüfbaren Zustände zur Verfügung. Der Tester kann aus dieser Liste ein Element wählen, um ein entsprechendes assert-Kommando in das Testskript zu übernehmen. Der Sollwert entspricht dabei dem aktuellen Zustandswert des Components.

ATOSj: Es können sowohl `ACTION`- als auch `TEST`-Kommandos aufgezeichnet werden.

Die Möglichkeiten zur Aufzeichung von Testskripten in ATOSj und Marathon unterscheiden sich nur marginal. In beiden Systemen können sowohl Kommandos zur Simulation von Nutzeraktionen als auch Kommandos für die Zustandsprüfung aufgezeichnet werden. Keines der beiden Testsysteme unterstützt einen analogen Modus, der Maus- und Tastatureingaben ohne Berücksichtigung des Kontextes aufzeichnet. Grafische Anwendungen, die hauptsächlich durch langanhaltende Mausbewegungen gesteuert werden, wie z.B. Zeichenprogramme, können deshalb weder mit Marathon noch mit ATOSj getestet werden.

8.1.7 Kombinierte Testdurchläufe

Marathon: In Marathon besteht die Möglichkeit, eine Auswahl von Testskripten zu einer Testsuite zusammenzufassen. Jedes Testskript enthält eine „test“-Funktion und wird auf

einen JUnit-Testfall abgebildet. Die Testfälle einer Testsuite werden dann nacheinander ausgeführt. Es ist nicht möglich einen Testfall mehrfach in eine Testsuite aufzunehmen.

ATOSj: Mehrere Testsequenzen können durch die Verwendung von Testpaketen in beliebiger Reihenfolge kombiniert werden. Ein Testpaket ist eine Sammlung semantisch zusammengehöriger Testsequenzen. Ein typisches Testpaket beinhaltet alle Testsequenzen zu einem bestimmten Use-Case aus den Anforderungsdokumenten des Testobjekts.

Die Metaphern Testpaket und Testsuite sind einander ähnlich. Aber Testpakete können persistent gespeichert werden. Testsuiten dagegen werden dynamisch durch den Tester erzeugt, d.h. für einen wiederholten Test muss dieselbe Konfiguration wieder von Hand eingerichtet werden. Dieser Automatisierungsmangel macht die Verwendung von Testsuiten fehleranfällig, da die Reihenfolge der Testskripte vertauscht oder Testskripte vergessen werden können. Testsuiten schränken außerdem die Kombination von Testfällen ein, da jeder Testfall nur einmal enthalten sein darf.

8.1.8 Sollwertvergleich der Eigenschaften von Components

Marathon: Die Einhaltung von Sollwerten wird in Marathon, in Anlehnung an JUnit, über assert-Methoden sichergestellt. Sie prüfen, ob ein Zustand eines Components dem vorgegebenen Sollwert entspricht, ist dies nicht der Fall meldet das Testsystem einen Fehler. Für viele Eigenschaften der Standard-Components gibt es vordefinierte assert-Methoden. Dem Nutzer steht es frei, eigene assert-Methoden zu definieren, um nicht unterstützte Eigenschaften abzutesten.

ATOSj: ATOSj stellt eine Reihe vordefinierter Zustandsprüfungen zur Verfügung. Hierfür wird das HTS-Kommando `TEST` verwendet.

Die Menge der vordefinierten Zustandsprüfungen ist in Marathon und ATOSj in etwa gleich, es gibt nur geringe Unterschiede. So ist es in Marathon z.B. möglich, die Hintergrundfarbe eines Components mit einem RGBA-Farbwert zu vergleichen, nicht so in ATOSj. Dafür bietet ATOSj die Möglichkeit für numerische Sollwertvergleiche, die in Marathon nicht verfügbar ist. Assert-Methoden und `TEST`-Kommandos unterscheiden sich weniger in ihrer Funktionalität, als vielmehr in ihrer Eindeutigkeit und Lesbarkeit. In einem `TEST`-Kommando muss, neben dem Fenstertitel und dem Namen, auch der Typ des Components angegeben werden, wohingegen eine assert-Methode nur den Namen zur Referenzierung benötigt. Das hat zur Folge, dass für eine assert-Methode in Marathon nicht immer eindeutig erkennbar ist auf welche Art von Component sie sich bezieht. Außerdem muss vorher über die Funktion `window` global der Titel des Fensters angegeben werden, dessen Inhalte im Folgenden geprüft werden sollen.

Wie die beiden Beispiele aus Tabelle 13 zeigen, kann unterschiedliche Semantik des Soll-

Semantik	**Marathon**	**ATOSj**
Es soll geprüft werden, ob der Radiobutton mit dem Namen „Male“ im Fenster „Simple Widgets“ gesetzt ist.	`window('Simple Widgets')` `assertText('Male', 'true')`	TEST, ‘‘Simple Widgets‘‘, RADIOBUTTON, ‘‘Male‘‘, CHECKSTATE, TRUE
Es soll geprüft werden, ob die Checkbox mit dem Namen „Male“ im Fenster „Simple Widgets“ gesetzt ist.	`window('Simple Widgets')` `assertText('Male', 'true')`	TEST, ‘‘Simple Widgets‘‘, CHECKBOX, ‘‘Male‘‘, CHECKSTATE, TRUE

Tabelle 13: Beispiel für einen Sollwertvergleich

wertabgleichs in Marathon zu derselben Folge von Kommandos führen. Obwohl zwei unterschiedliche Componenttypen abgetestet werden, ist das für den Tester nicht ersichtlich. Dadurch wird eine Interpretation der Kommandos erschwert und es besteht Verwechslungsgefahr. Das Kommando `TEST` dagegen, liefert bei unterschiedlicher Semantik auch immer eine unterschiedliche Parameterfolge. Außerdem bietet es einen schnelleren Überblick, da der Fenstertitel direkt im Kommando enthalten ist. In einem Marathon-Skript leidet der Überblick teilweise, da die Angabe des Fenstertitels durch die Funktion `window` viele Kommandos vorher stehen kann.

8.1.9 Testauswertung

Marathon: Die Ergebnisse eines Testdurchlaufs werden in einem Dialog präsentiert. Der Dialog zeigt eventuell aufgetretene Fehler, deren Beschreibung und die entsprechende Zeile im Skript. Die Fehler werden entsprechend der Nomenklatur in JUnit entweder als Error oder Failure gekennzeichnet. Das Ergebnis kann zusätzlich noch in einem übersichtlichen HTML-Report gespeichert werden.

ATOSj: Die Ergebnisse eines Testdurchlaufs werden nach der Ausführung in einem Dialog präsentiert. Im Fehlerfall wird die Nummer des Kommandos angegeben und eine Fehlerbeschreibung. Die Ergebnisse können dann in einem PDF-Report gespeichert werden. Zusätzlich werden die Ausgaben des Testobjektes in den Standardausgabepuffer (stdout) und in den Standardfehlerpuffer (stderr) angezeigt.

Beide Werkzeuge sind in ihrer Funktionalität im Bereich Testauswertung fast gleichwertig und unterscheiden sich lediglich in der Art der Fehlerausschriften und Präsentation der Ergebnisse. Die Visualisierung der Ausgaben des Testobjektes war in Marathon ebenfalls angedacht, wie der Benutzerleitfaden konstatiert, ist aber nicht funktionstüchtig. Diese Funktion ist aber bei der Suche nach den Ursachen eines Laufzeitfehlers besonders hilfreich, da in Java alle nicht abgefangenen Ausnahmen standardmäßig nach stderr

geschrieben werden. Die Fehlerausgabe ist sehr aufschlussreich, denn sie beinhaltet eine kurze Fehlerbeschreibung sowie den kompletten Call-Stack, inklusive der Zeilennummern der gerufenen Funktionen im Quelltext. Mit dieser präzisen Angabe kann ein Fehler meist sehr schnell eingegrenzt werden.

8.2 Erreichte Ziele

Ursprünglich sollte mit dieser Diplomarbeit das Ziel verfolgt werden, das bereits existierende System ATOS dahingehend zu erweitern, dass es in Zukunft sowohl mit C++, als auch mit Java genutzt werden können sollte. Bei genauerer Betrachtung stellte sich jedoch die Frage nach dem Nutzen einer solchen Erweiterung. Zum einen ist fraglich, ob die Kombination beider Programmiersprachen einen zusätzlichen Nutzen darstellt. Im Regelfall bestehen Projekte aus nur einer der beiden Programmiersprachen, was zu der Situation führt, dass dem Nutzer ledglich die Wahl bleibt, welchen Teil von ATOS er nutzen möchte: den Java- ODER den C++- Teil. Ein gleichzeitiges Nutzen scheint eher unrealistisch.

Dennoch sind solche Projekte dank Java-Native durchaus vorstellbar. Allerdings würde die grafische Benutzeroberfläche bei einer derartigen Kombination wohl dennoch immer entweder eine Java- ODER eine Native GUI sein. Ein Zusammenspiel beider ist nahezu unmöglich, da keine direkte Kommunikation zwischen beiden möglich ist. Beide Schnittstellen sind voneinander völlig unabhängig. Die Kommunikation zwischen beiden Teilen wäre also nur mittels selbst implementierter Klassen, die sozusagen als Verbindungsglied zwischen beiden Teilen fungieren müssten, möglich.

Da also eine Kombination ohnehin fast auszuschließen war, wurde der Ansatz der ATOS-Erweiterung bezüglich Java verworfen. Das danach entstandene System ATOSj genügt aber dennoch sämtliche Anforderungen, die auch ATOS bereits erfüllt, und geht noch deutlich darüber hinaus.

Die Erweiterungen in HTS ermöglichen nun auch die Manipulation von absolut grundlegenden GUI-Elementen, wie Table und Tree, welche in ATOS noch fehlten. Des Weiteren wurden einige allgemeine Unzulänglichkeiten an HTS, die wohl aus dem Kontext des XCTL-Programms heraus entstanden sind, beseitigt.

Völlig neue Möglichkeiten werden dem Nutzer durch die Verwendung der neuen Custom-Components gegeben. Für sie können beliebige Properties definiert werden, die an dem jeweiligen Objekt gesetzt und ausgelesen werden können. Hierbei hat der Anwender absolut freien Handlunsgsspielraum, da er seine eigenen Wrapperklassen definieren, und somit ATOSj um seinen eigenen Quelltext erweitern kann. Die Möglichkeit der Bündelung kleiner Einzelschritte zu einem gesamtheiltichen Schritt an einem Custom-Component, erhöht Übersichtlichkeit und Semantik der erstellten Skripte.

Hervorzuheben ist des Weiteren die Tatsache, dass ATOSj sowohl mit den Java Standardoberflächen AWT und Swing, als auch mit dem heutzutage ebenso gebräuchlichen SWT zusammenarbeitet. Insbesondere die Zusammenarbeit mit SWT - und hier ganz speziell der Capture-Teil - war äußerst kompliziert zu realisieren. Unterstrichen wird dies durch die Tatsache, dass uns derzeit kein anderes Programm außer ATOSj bekannt ist, das diese Möglichkeit bietet.

Durch die Architektur von Java bedingt, kann als weiteres erreichtes Ziel die Multiplattformfähigkeit von ATOSj genannt werden. Eine Möglichkeit, die höchstwahrscheinlich entfallen wäre, wenn ATOSj - wie ursprünglich geplant - als Erweiterung von ATOS entwickelt worden wäre, da ATOS lediglich unter Windows lauffähig ist.

8.3 Erweiterungsmöglichkeiten

Betrachtet man die Implementation von ATOS, so fällt schnell auf, dass die Quellen nur sehr schwach strukturiert sind. Dies erschwert eine potentielle Weiterentwicklung. Aufgrund des gewählten Entwicklungsweges hat ATOSj hier deutliche Vorteile gegenüber seinem Vorgänger. Die Kombination der gut strukturierten Quellen mit dem ebenfalls vorhandenen Javadoc sollten es Einsteigern ermöglichen mit akzeptablem Aufwand in die Thematik einzusteigen und sich in ATOSj einzuarbeiten. Aufgrund des erheblichen Umfangs der Quellen kann dies jedoch nur im Rahmen einer Studien- oder Diplomarbeit geschehen. Für derartige Arbeiten sind mehrere Themen vorstellbar.

Die bisher getätigten Veränderungen an der HTS-Syntax stellen bereits eine deutliche Verbesserung der Nutzbarkeit dar. Allerdings sind viele Aktionen bisher sehr statisch. Die Aufnahme und der Zugriff auf Variablen in den Kommandos ist bisher lediglich bei READ, COMPARE und WHILE möglich. Dies ist in vielen Fällen nicht ausreichend. Insbesondere für den Fall, dass Zustände von GUI-Elementen abgefragt werden, wäre die Benutzbarkeit von Variablen wünschenswert. Vorstellbares Anwendungsszenario wäre beispielsweise der Test, ob zwei Tabellen wirklich die laut Programmlogik erforderlich gleiche Anzahl von Zeilen aufweisen. Aber nicht nur beim Test, sondern auch im Falle der Veränderung von Oberflächenelementen wären solche dynamischen Variablen wünschenswert. Würde HTS dahingehend erweitert, wäre es beispielsweise möglich einen in einem Element ausgelesenen Wert in ein anderes zu übertragen. Nur so wird eine echte Reaktion auf durch das Programm produzierte und nicht vorhersehbare Werte erst wirklich möglich.

ATOSj unterstützt in der aktuellen Version bereits vollständig das SWT. Eine Erweiterung des SWT ist das swt.custom-Package. In Ihm sind grafische Elemente enthalten, die sich äußerlich von den Standard-Components leicht unterscheiden. Starke Verwendung finden diese sie in der Entwicklungsumgebung Eclipse. Die gehäufte Verwendung dieser geschwungenen Elemente in Eclipse geben dem Progamm sein typisches Erschei-

nungsbild. Entwickler können diese Components ebenfalls für eigene Projekte nutzen. Das gesamte swt.custom-Package wird derzeit noch nicht von ATOSj unterstützt. Für eine Implementation müsste lediglich der SWTEventmonitor erweitert werden. Eine weitere Anpassung wäre nicht nötig, da dieselben Events wie unter dem Standard-SWT auftreten.

Ebenso wäre eine Erweiterung des SWTEventMonitors auf Swing und AWT denkbar. Die Entwicklung der bisher genutzten Java Accessibility Utilities wurde offensichtlich bereits vor mehreren Jahren eingestellt. Stellenweise ist die aktuellste Version 1.3 auch fehlerbehaftet. Eine Lösung mittels Javassist und dem SWTEventMonitor wäre problemlos möglich und würde die Architektur von ATOSj weiter vereinheitlichen. Eine solche Änderung würd ebenfalls eine Anpassung der Swing-Wrapperklassen bedingen.

Damit ATOSj nicht nur im deutschsprachigen Raum verwendet werden kann, wäre eine Internationalisierung von ATOSj wünschenswert. Diese nicht allzu aufwendige Erweiterung könnte mittels Sprachdateien realisiert werden, die alle benötigen Dialogtexte enthalten. Diese Lösung würde es Angehörigen von weniger verbreiteten Sprachen zudem ermöglichen eigene lokalisierte Versionen zu erstellen, indem sie lediglich die Sprachdateien übersetzen müssten.

Dass ATOSj durchaus Potential zur Weiterentwicklung besitzt, zeigte sich in den letzten Tagen vor Abschluss dieser Arbeit. So ist derzeit eine Arbeit in Planung, deren Ziel es ist, die für eine feste Testsuite und ein beliebiges Testobjekt automatisch Testskripte zu erzeugen. Nach aktuellem Stand wird ATOSj die Grundlage für diese Diplomarbeit bilden.

8.3.1 Erweiterung für Windows

Mit der Entwicklung von ATOSj ist es gelungen, den Test oberflächenbasierter Java-Programme zu automatisieren. Die starke Abhängigkeit des Vorgängers ATOS vom Betriebssystem Windows hat leider eine Integration der Java-Unterstützung in das bestehende System verhindert. Im Resultat stehen sich nun zwei unabhängige Systeme gegenüber, die sich zwar in ihrer Funktionalität sehr ähneln aber in der technischen Realisierung völlig unabhängig voneinander sind. Obwohl beide einen gemeinsamen Ursprung haben, driften sie schon jetzt auseinander. So arbeitet ATOSj mit einer leicht abgewandelten HTS-Version, die beispielsweise zusätzliche Standard-Components unterstützt. Dagegen bietet ATOS z.B. die Möglichkeit der automatischen Testsequenzgenerierung mit Hilfe eines CTE-Baums, die in ATOSj fehlt. Sollen beide Systeme einen einheitlichen Funktionsumfang aufweisen, müssen Änderungen an der einen Version immer auf die jeweils andere Version übertragen werden. Das bedeutet doppelten Wartungsaufwand und ist auf Grund der unterschiedlichen Implementationssprachen nicht trivial. Um den Wartungsaufwand zu reduzieren und Einheitlichkeit zu garantieren, müssen beide Systeme

zusammengeführt werden. ATOSj ist wegen seiner Betriebssystemunabhängigkeit besser als Plattform für ein gemeinsames System geeignet als ATOS. Außerdem legt es mit seiner 2-Schichten-Architektur schon die Grundlage für die unkomplizierte Erweiterung der GUI-Unterstützung, siehe Kapitel 5.2. Für eine Windows-Erweiterung sind nur einige wenige Schritte nötig, die analog zur Implementation der bisher unterstützten Bibliotheken ausgeführt werden. Zuerst muss zu jedem HTS-Typ ein funktionsgleiches Windows-Steuerelement identifiziert werden. Danach wird für jedes identifizierte Steuerelement ein Wrapper erstellt, der das Interface des korrespondierenden HTS-Typs implementiert. Der Zugriff auf Windows-interne Funktionen, die zur Manipulation des Steuerelements benötigt werden, wird durch das Java Native Interface ermöglicht. Hierzu muss eine Laufzeitbibliothek (Dynamic Link Library) erstellt werden, die alle als „native“ deklarierten Methoden der Windows-Wrapperklassen enthält. Wird als Implementationssprache für die Laufzeitbibliothek C/C++ verwendet, können bestimmte Komponenten, wie z.B. die Funktion zum Auslesen des Aktivierungsstatus eines Components, direkt aus ATOS übernommen werden. Zum Schluss müssen die Parameter eines ATOSj-Projektes für die Ausführung von Windows-Testobjekten angepasst werden. Auch wenn der Weg zur Unterstützung von Windows-Testobjekten bekannt ist, wurde er von uns nicht realisiert, da das den Rahmen der Diplomarbeit gesprengt hätte.

A Die HTS-Spezifikation

A.1 Die Syntax

Hier wird die Grammatik der geänderten Scriptsprache in EBNF (erweiterte Backus-Naur-Form) vorgestellt. Jede Anweisung wird mit einem Zeilenumbruch (\n) abgeschlossen. Folgende Morpheme werden vereinbart und besonders gekennzeichnet:

`<zahl>`	ist eine beliebige nichtnegative natürliche Zahl
`<+zahl+>`	ist eine beliebige positive natürliche Zahl
`<.float.>`	ist eine beliebige Fließkommazahl, als Dezimalzeichen wird ein Punkt verwendet
`$string$`	ist eine beliebige Zeichenkette in Anführungszeichen, als Maskierungszeichen wird das Backslash verwendet
`$+string+$`	ist eine Zeichenkette ohne Leer- und Anführungszeichen

```
skript              =  {anweisung}
                       CLEANUP
                       {anweisung}
anweisung           =  aktionsschritt | auswertungsschritt |
                       interaktion | dateioperation | loopstruktur |
                       whilestruktur | kommentar | deakt_kommando \n
aktionsschritt      =  warten | aktion
auswertungsschritt  =  lesen | vergleich | testen | fenstersichtbarkeit
interaktion         =  frage | nachricht
dateioperation      =  kopieren | löschen | exitenz | starten
```

Unbedingte Schleife

```
loopstruktur  =  LOOP, <+Wiederholungsanzahl+>
                 (anweisung)
                 ENDLOOP
```

Bedingte Schleife *neu*

```
whilestruktur  =  WHILE, vergleichskörper
                  (anweisung)
                  ENDWHILE
```

Nicht ausführbare Anweisungen

```
kommentar       =  COMMENT, $text$
deakt_kommando  =  DISABLE, anweisung
```

Warten für eine Zeitspanne

```
warten  =  WAIT, <+Millisekunden+>
```

Abfragen der Components *geändert*

```
lesen          =  READ, $Fenstertitel$ | MAIN, typ_lesen, $Variablenname$
typ_lesen      =  component_lesen | custom_lesen | editbox_lesen | label_lesen |
                  radiobutton_lesen | checkbox_lesen | button_lesen | combobox_-
                  lesen | liste_lesen | tabelle_lesen | baum_lesen | menü_lesen |
                  kartei_lesen | fenster_lesen

component_-    =  COMPONENT, $Name$, ENABLESTATE |
lesen             COMPONENT, $Name$, FOCUSSTATE |
                  COMPONENT, $Name$, VISIBLESTATE
custom_lesen   =  $+Custom+$, $Name$, PROPERTY, $Eigenschaft$
editbox_-      =  EDITBOX, $Name$, TEXT |
lesen             EDITBOX, $Name$, NUM |
                  EDITBOX, $Name$, ENABLESTATE |
                  EDITBOX, $Name$, FOCUSSTATE |
                  EDITBOX, $Name$, VISIBLESTATE
label_lesen    =  LABEL, $Name$, TEXT |
                  LABEL, $Name$, NUM |
                  LABEL, $Name$, ENABLESTATE |
                  LABEL, $Name$, FOCUSSTATE |
                  LABEL, $Name$, VISIBLESTATE
radiobutton_-  =  RADIOBUTTON, $Name$, CHECKSTATE |
lesen             RADIOBUTTON, $Name$, TEXT |
                  RADIOBUTTON, $Name$, NUM |
                  RADIOBUTTON, $Name$, ENABLESTATE |
                  RADIOBUTTON, $Name$, FOCUSSTATE |
                  RADIOBUTTON, $Name$, VISIBLESTATE
checkbox_-     =  CHECKBOX, $Name$, CHECKSTATE |
lesen             CHECKBOX, $Name$, TEXT |
                  CHECKBOX, $Name$, NUM |
                  CHECKBOX, $Name$, ENABLESTATE |
                  CHECKBOX, $Name$, FOCUSSTATE |
                  CHECKBOX, $Name$, VISIBLESTATE
button_lesen   =  BUTTON, $Name$, TEXT |
                  BUTTON, $Name$, NUM |
                  BUTTON, $Name$, ENABLESTATE |
                  BUTTON, $Name$, FOCUSSTATE |
                  BUTTON, $Name$, VISIBLESTATE
combobox_-     =  COMBOBOX, $Name$, ITEMCOUNT |
lesen             COMBOBOX, $Name$, TEXT |
                  COMBOBOX, $Name$, NUM |
                  COMBOBOX, $Name$, ENABLESTATE |
                  COMBOBOX, $Name$, FOCUSSTATE |
                  COMBOBOX, $Name$, VISIBLESTATE
```

```
liste_lesen    =  LIST, $Name$, ITEMCOUNT |
                  LIST, $Name$, TEXT, SUBITEM, <+Listenindex+> |
                  LIST, $Name$, ENABLESTATE |
                  LIST, $Name$, FOCUSSTATE |
                  LIST, $Name$, VISIBLESTATE
tabelle_-      =  TABLE, $Name$, TEXT, SUBITEM, <+Zeilenindex+>, <+Spaltenindex> |
lesen             TABLE, $Name$, NUM, SUBITEM, <+Zeilenindex+>, <+Spaltenindex> |
                  TABLE, $Name$, CHECKSTATE, SUBITEM, <+Zeilenindex+>,
                  <+Spaltenindex> |
                  TABLE, $Name$, ITEMCOUNT |
                  TABLE, $Name$, ENABLESTATE |
                  TABLE, $Name$, FOCUSSTATE |
                  TABLE, $Name$, VISIBLESTATE
baum_lesen     =  TREE, $Name$, ENABLESTATE |
                  TREE, $Name$, FOCUSSTATE |
                  TREE, $Name$, VISIBLESTATE
menü_lesen     =  MENU, $Name$, ENABLESTATE, SUBITEM {,$Menüpunkt$} |
                  MENU, $Name$, CHECKSTATE, SUBITEM {,$Menüpunkt$} |
                  MENU, $Name$, ENABLESTATE |
                  MENU, $Name$, FOCUSSTATE |
                  MENU, $Name$, VISIBLESTATE
kartei_lesen   =  TABFOLDER, $Name$, TEXT, SUBITEM, <+Tabindex+> |
                  TABFOLDER, $Name$, ITEMCOUNT |
                  TABFOLDER, $Name$, ENABLESTATE |
                  TABFOLDER, $Name$, FOCUSSTATE |
                  TABFOLDER, $Name$, VISIBLESTATE
fenster_-      =  WINDOW, $Name$, ENABLESTATE |
lesen             WINDOW, $Name$, FOCUSSTATE |
                  WINDOW, $Name$, VISIBLESTATE
```

Abfragen und Sollwertvergleich der Components *geändert*

```
testen         =  TEST, $Fenstertitel$ | MAIN, typ_testen
typ_testen     =  component_testen | custom_testen | editbox_testen | label_testen |
                  radiobutton_testen | checkbox_testen | button_testen | combobox_-
                  testen | liste_testen | tabelle_testen | baum_testen | menü_testen
                  | kartei_testen | fenster_testen

component_-    =  COMPONENT, $Name$, ENABLESTATE, bool_wert |
testen            COMPONENT, $Name$, FOCUSSTATE, bool_wert |
                  COMPONENT, $Name$, VISIBLESTATE, bool_wert
custom_-       =  $+Custom+$, $Name$, PROPERTY, $Eigenschaft$, $Sollwert$
testen
editbox_-      =  EDITBOX, $Name$, TEXT, $Sollwert$ |
testen            EDITBOX, $Name$, NUM <.Sollwert.> [,vergleichsmodus] |
                  EDITBOX, $Name$, ENABLESTATE, bool_wert |
                  EDITBOX, $Name$, FOCUSSTATE, bool_wert |
                  EDITBOX, $Name$, VISIBLESTATE, bool_wert
label_testen   =  LABEL, $Name$, TEXT, $Sollwert$ |
                  LABEL, $Name$, NUM, <.Sollwert.> [,vergleichsmodus] |
                  LABEL, $Name$, ENABLESTATE, bool_wert |
                  LABEL, $Name$, FOCUSSTATE, bool_wert |
                  LABEL, $Name$, VISIBLESTATE, bool_wert
```

```
radiobutton_-  =  RADIOBUTTON, $Name$, CHECKSTATE, bool_wert |
testen            RADIOBUTTON, $Name$, TEXT, $Sollwert$ |
                  RADIOBUTTON, $Name$, NUM, <.Sollwert.> [,vergleichsmodus] |
                  RADIOBUTTON, $Name$, ENABLESTATE, bool_wert |
                  RADIOBUTTON, $Name$, FOCUSSTATE, bool_wert |
                  RADIOBUTTON, $Name$, VISIBLESTATE, bool_wert
checkbox_-     =  CHECKBOX, $Name$, CHECKSTATE, bool_wert |
testen            CHECKBOX, $Name$, TEXT, $Sollwert$ |
                  CHECKBOX, $Name$, NUM, <.Sollwert.> [,vergleichsmodus] |
                  CHECKBOX, $Name$, ENABLESTATE, bool_wert |
                  CHECKBOX, $Name$, FOCUSSTATE, bool_wert |
                  CHECKBOX, $Name$, VISIBLESTATE, bool_wert
button_-       =  BUTTON, $Name$, TEXT, $Sollwert$ |
testen            BUTTON, $Name$, NUM, <.Sollwert.> [,vergleichsmodus] |
                  BUTTON, $Name$, ENABLESTATE, bool_wert |
                  BUTTON, $Name$, FOCUSSTATE, bool_wert |
                  BUTTON, $Name$, VISIBLESTATE, bool_wert
combobox_-     =  COMBOBOX, $Name$, ITEMCOUNT, <+Sollwert+> [,vergleichsmodus2] |
testen            COMBOBOX, $Name$, TEXT, $Sollwert$ |
                  COMBOBOX, $Name$, NUM, <.Sollwert.> [,vergleichsmodus] |
                  COMBOBOX, $Name$, ENABLESTATE, bool_wert |
                  COMBOBOX, $Name$, FOCUSSTATE, bool_wert |
                  COMBOBOX, $Name$, VISIBLESTATE, bool_wert
liste_testen   =  LIST, $Name$, ITEMCOUNT, <+Sollwert+> [,vergleichsmodus2] |
                  LIST, $Name$, TEXT, $Sollwert$, SUBITEM, <+Listenindex+> |
                  LIST, $Name$, NUM, <.Sollwert.> [,vergleichsmodus], SUBITEM,
                  <+Listenindex+> |
                  LIST, $Name$, ENABLESTATE, bool_wert |
                  LIST, $Name$, FOCUSSTATE, bool_wert
tabelle_-      =  TABLE, $Name$, ITEMCOUNT, <+Sollwert+> [,vergleichsmodus2] |
testen            TABLE, $Name$, TEXT, $Sollwert$, SUBITEM, <+Zeilenindex+>,
                  <+Spaltenindex+> |
                  TABLE, $Name$, NUM, <.Sollwert.> [,vergleichsmodus], SUBITEM,
                  <+Zeilenindex+>, <+Spaltenindex+> |
                  TABLE, $Name$, CHECKSTATE, bool_wert, SUBITEM, <+Zeilenindex+>,
                  <+Spaltenindex> |
                  TABLE, $Name$, ENABLESTATE, bool_wert |
                  TABLE, $Name$, FOCUSSTATE, bool_wert |
                  TABLE, $Name$, VISIBLESTATE, bool_wert
baum_testen    =  TREE, $Name$, ENABLESTATE, bool_wert |
                  TREE, $Name$, FOCUSSTATE, bool_wert |
                  TREE, $Name$, VISIBLESTATE, bool_wert
menü_testen    =  MENU, $Name$, CHECKSTATE, bool_wert, SUBITEM {,$Menüpunkt$}] |
                  MENU, $Name$, ENABLESTATE, bool_wert [, SUBITEM {,$Menüpunkt$}] |
                  MENU, $Name$, FOCUSSTATE, bool_wert |
                  MENU, $Name$, VISIBLESTATE, bool_wert
kartei_-       =  TABFOLDER, $Name$, ITEMCOUNT, <+Sollwert+> [,vergleichsmodus2] |
testen            TABFOLDER, $Name$, TEXT, $Sollwert$, SUBITEM, <+Tabindex+> |
                  TABFOLDER, $Name$, ENABLESTATE, bool_wert |
                  TABFOLDER, $Name$, FOCUSSTATE, bool_wert
fenster_-      =  WINDOW, $Name$, ENABLESTATE, bool_wert |
testen            WINDOW, $Name$, FOCUSSTATE, bool_wert |
                  WINDOW, $Name$, VISIBLESTATE, bool_wert
```

Vergleich von Ist- und Sollwerten *geändert*

```
vergleich                =  COMPARE, $Variable1$, vergleichskörper
vergleichskörper         =  vergleich_mit_variable | vergleich_mit_wert
vergleich_mit_variable   =  vergleich_var_textuell | vergleich_var_numerisch |
                            vergleich_var_bool
vergleich_mit_wert       =  vergleich_wert_textuell | vergleich_wert_numerisch |
                            vergleich_wert_bool

vergleich_var_textuell   =  STR, VAR, $Variable2$
vergleich_var_numerisch  =  NUM, VAR, $Variable2$ [,vergleichsmodus]
vergleich_var_bool       =  BOOL, VAR, $Variable2$

vergleich_wert_textuell  =  STR, VAL, $Wert$
vergleich_wert_numerisch =  NUM, VAL, <.Wert.> [,vergleichsmodus]
vergleich_wert_bool      =  BOOL, VAL, bool_wert
bool_wert                =  TRUE | FALSE

vergleichsmodus          =  EQ | GRT | GEQ | LEQ | LSS | NEQ | toleranz
toleranz                 =  TOL, <.Toleranzwert.>
vergleichsmodus2         =  EQ | GRT | GEQ | LEQ | LSS | NEQ
```

Aktionen auf Components *geändert*

```
aktion         =  ACTION, $Fenstertitel$ | MAIN, aktions_art
aktions_art    =  component_aktion | custom_aktion | editbox_aktion | label_aktion |
                  radiobutton_aktion | checkbox_aktion | button_aktion | combobox_-
                  aktion | listen_aktion | tabellen_aktion | baum_aktion | menü_-
                  aktion | kartei_aktion | fenster_aktion

component_-    =  COMPONENT, $Name$, tasten_druck |
aktion            COMPONENT, $Name$, maus_klick
custom_-       =  $+Custom+$, $Name$, PROPERTY, $Eigenschaft$, $Wert$
aktion
editbox_-      =  EDITBOX, $Name$, tasten_druck |
aktion            EDITBOX, $Name$, maus_klick |
                  EDITBOX, $Name$, EDIT, $Text$
label_aktion   =  LABEL, $Name$, tasten_druck |
                  LABEL, $Name$, maus_klick
radiobutton_-  =  RADIOBUTTON, $Name$, tasten_druck |
aktion            RADIOBUTTON, $Name$, maus_klick |
                  RADIOBUTTON, $Name$, CHECK |
                  RADIOBUTTON, $Name$, SELECT
checkbox_-     =  CHECKBOX, $Name$, tasten_druck |
aktion            CHECKBOX, $Name$, maus_klick |
                  CHECKBOX, $Name$, CHECK
                  CHECKBOX, $Name$, UNCHECK
                  CHECKBOX, $Name$, SELECT
```

```
button_-       =  BUTTON, $Name$, tasten_druck |
aktion            BUTTON, $Name$, maus_klick
                  BUTTON, $Name$, SELECT
combobox_-     =  COMBOBOX, $Name$, tasten_druck |
aktion            COMBOBOX, $Name$, maus_klick |
                  COMBOBOX, $Name$, SELECT, SUBITEM, <+Listenindex+> |
                  COMBOBOX, $Name$, EDIT, $Text$
listen_-       =  LIST, $Name$, tasten_druck |
aktion            LIST, $Name$, maus_klick |
                  LIST, $Name$, SELECT, SUBITEM, <+Listenindex+> |
                  LIST, $Name$, ADDSELECT, SUBITEM, <+Listenindex+> |
                  LIST, $Name$, DESELECT, SUBITEM, <+Listenindex+>
tabellen_-     =  TABLE, $Name$, tasten_druck |
aktion            TABLE, $Name$, maus_klick |
                  TABLE, $Name$, SELECT, SUBITEM, <+Zeilenindex+> |
                  TABLE, $Name$, ADDSELECT, SUBITEM, <+Zeilenindex+> |
                  TABLE, $Name$, DESELECT, SUBITEM, <+Zeilenindex+> |
                  TABLE, $Name$, EDIT, $Text$, SUBITEM, <+Zeilenindex+>,
                  <+Spaltenindex+> |
                  TABLE, $Name$, CHECK, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+> |
                  TABLE, $Name$, UNCHECK, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>
baum_aktion    =  TREE, $Name$, tasten_druck |
                  TREE, $Name$, maus_klick |
                  TREE, $Name$, SELECT, SUBITEM {,$Knoten$} |
                  TREE, $Name$, ADDSELECT, SUBITEM, {$Knoten$} |
                  TREE, $Name$, DESELECT, SUBITEM {,$Knoten$} |
                  TREE, $Name$, EDIT, $Text$, SUBITEM, {$Knoten$}
menü_aktion    =  MENU, $Name$, tasten_druck |
                  MENU, $Name$, maus_klick |
                  MENU, $Name$, SELECT, SUBITEM {,$Menüpunkt$} |
                  MENU, $Name$, CHECK, SUBITEM {,$Menüpunkt$} |
                  MENU, $Name$, UNCHECK, SUBITEM {,$Menüpunkt$}
kartei_akion   =  TABFOLDER, $Name$, tasten_druck |
                  TABFOLDER, $Name$, maus_klick |
                  TABFOLDER, $Name$, SELECT, <+Tabindex+>
fenster_-      =  WINDOW, $Name$, tasten_druck |
aktion            WINDOW, $Name$, maus_klick |
                  WINDOW, $Name$, CLOSE |
                  WINDOW, $Name$, MAXIMIZE |
                  WINDOW, $Name$, MINIMIZE |
                  WINDOW, $Name$, NORMALIZE

tasten_druck   =  PRESSKEY, $Zeichen$ [,ALT] [,CTRL] [,SHIFT]
maus_klick     =  CLICK, LEFT | RIGHT, <+Klickanzahl+> [,ALT] [,CTRL] [,SHIFT]
```

Fenster vorhanden/nicht vorhanden

```
fenstersichtbarkeit = WINDOWEXISTS, $Fenstertitel$, YES|NO
```

Nachrichten/Fragen an Nutzer

```
nachricht  =  MESSAGE, $Text$
frage      =  QUESTION, $Text$, YES|NO
```

Datei kopieren

```
kopieren  =  COPY, dir, $Quelldatei$, dir, $Zieldatei$ [,FORCE] [,IFSRCEXISTS]
dir       =  ABS | TGT | REF | BIN | ENV | LOG
```

Existenz Datei/Verzeichnis

```
existenz              =  existenz_datei | existenz_verzeichnis
existenz_datei        =  EXISTS, dir, $Dateiname$
existenz_verzeichnis  =  EXISTS, dir, $Verzeichnisname$, DIRECTORY
```

Datei loeschen

```
loeschen = DELETE, dir, $Dateiname$ [,FORCE] [,IFEXISTS]
```

Starten einer Anwendung

```
starten           =  testobjekt_start | extern_start
testobjekt_start  =  START, $Parameter$
```

Starten eines externen Programmes

```
extern_start           =  start_warten_auf_ende | start_warten_auf_zeit |
                          start_ohne_warten
start_warten_auf_ende  =  LAUNCH, dir, $Programmname$, $Parameter$, FOREVER,
                          <Returncode>, [, $Logfile$]
start_warten_auf_zeit  =  LAUNCH, dir, $Programmname$, $Parameter$,
                          TIME, <+Terminierungsdauer+>, <Returncode>, [,
                          $Logfile$]
start_ohne_warten      =  LAUNCH, dir, $Programmname$, $Parameter$, NOWAIT
```

A.2 Die Semantik der HTS-Kommandos

A.2.1 ACTION

Aufruf:
ACTION, <FENSTER>, <COMPONENTTYP>, <COMPONENTNAME>, <AKTION>

Parameter:

<FENSTER>: Der initiale Titel des Fensters in dem sich das Component befindet, welches manipuliert werden soll. Für das Hauptfenster dient das Schlüsselwort MAIN als Platzhalter für den tatsächlichen Titel.

<COMPONENTTYP>: Ist der Typ eines Components. Dieser ist eine unabhängige Abstraktion basierend auf der Funktionalität des Components. Folgende Typen sind möglich: COMPONENT, EDITBOX, LABEL, RADIOBUTTON, CHECKBOX, BUTTON, COMBOBOX, LIST, TABLE, TREE, MENU, TABFOLDER und WINDOW. Zu jedem HTS Typ existiert ein entsprechender Java-Typ aus der jeweiligen GUI-Bibliothek. Die folgende Tabelle zeigt die Zuordnung von HTS Typen zu den Klassen der graphischen Objekte, getrennt nach den, von ATOSj unterstützten Bibliotheken SWT und Swing.

Componenttyp	**SWT**	**Swing**
COMPONENT	Widget	Component
EDITBOX	Text	JTextComponent
COMBOBOX	Combo	JComboBox
LIST	List	JList
TABLE	Table	JTable
TREE	Tree	JTree
TABFOLDER	TabFolder	JTabbedPane
LABEL	Label	JLabel
RADIOBUTTON	Button Stil: SWT.RADIO ToolItem Stil: SWT.RADIO	JRadioButton
CHECKBOX	Button Stil: SWT.CHECK oder SWT.TOGGLE ToolItem Stil: SWT.CHECK	JToggleButton
BUTTON	Button Stil: SWT.PUSH oder SWT.ARROW	AbstractButton
MENU	Menu	JMenu
WINDOW	Shell	java.awt.Frame java.awt.Dialog

<COMPONENTNAME>: Der Name des Components. Dieser bildet zusammen mit <FENSTER> und <COMPONENTTYP> einen eindeutigen Identifikator. Um bei der Testausführung ein Component anhand seiner ID zu finden, muss ATOSj die Namen der Compenents des Testobjektes ermitteln. Dies geschieht zur Laufzeit des Testobjektes nach einer genau definierten Strategie. Diese soll möglichst eindeutige und plakative Namen für ein Component generieren. Des Weiteren soll sie auch die Arbeit mit Testobjekten ermöglichen, deren Quelltext nicht zur Verfügung steht. Das verwendete Vorgehensmuster ist in der nachfolgenden Tabelle beschrieben. Die Möglichkeiten der Benamung werden der Reihe nach durchgegangen bis ein nicht leerer Name gefunden wird.

Componenttyp	**SWT**	**Swing**

`COMPONENT`
`EDITBOX`
`COMBOBOX`
`LIST`
`TABLE`
`TREE`
`TABFOLDER`
	1. Der Name des Components. Festgelegt durch den Programmierer und abrufbar über die Funktion `getData(String key)` aus der Klasse `Widget`, wobei `key` den Wert "ATOSJ_COMPONENT_NAME_KEY" erhält.
	2. Die Grenzen des Components relativ zu seinem Elterncomponent. Die Grenzen werden definiert durch die linke obere Ecke des Components sowie dessen Höhe und Breite in Pixeln.
	3. Der Rückgabewert der Funktion `toString()` des Components.
		1. Der Name des Components. Festgelegt durch den Programmierer und abrufbar über die Funktion `getName()` aus der Klasse `java.awt.Component`.
		2. Der Text eines, diesem Component zugewiesenen, Labels, abrufbar über die Funktion `getClientProperty(Object property)` aus der Klasse `javax.swing.JComponent`, wobei `property` den Wert "labeledBy" erhält.
		3. Die Grenzen des Components relativ zu seinem Fenster. Die Grenzen werden definiert durch die linke obere Ecke des Components sowie dessen Höhe und Breite in Pixeln.

`LABEL`
`RADIOBUTTON`
`CHECKBOX`
`BUTTON`
	1. Der Name des Components. Festgelegt durch den Programmierer und abrufbar über die Funktion `getData(String key)` aus der Klasse `Widget`, wobei `key` den Wert "ATOSJ_COMPONENT_NAME_KEY" erhält.
	2. Der Text des Components, wie er zur Darstellung gesetzt wurde.
	3. Die Grenzen des Components relativ zu seinem Elterncomponent. Die Grenzen werden definiert durch die linke obere Ecke des Components sowie dessen Höhe und Breite in Pixeln.
	4. Der Rückgabewert der Funktion `toString()` des Components.
		1. Der Name des Components. Festgelegt durch den Programmierer und abrufbar über die Funktion `getName()` aus der Klasse `java.awt.Component`.
		2. Der Text des Components, wie er zur Darstellung gesetzt wurde.
		3. Der Text eines, diesem Component zugewiesenen, Labels, abrufbar über die Funktion `getClientProperty(Object property)` aus der Klasse `javax.swing.JComponent`, wobei `property` den Wert "labeledBy" erhält.
		4. Die Grenzen des Components relativ zu seinem Fenster. Die Grenzen werden definiert durch die linke obere Ecke des Components sowie dessen Höhe und Breite in Pixeln.

`MENU`	1. Der Name des Menüs. Festgelegt durch den Programmierer und abrufbar über die Funktion `getData(String key)` aus der Klasse `Widget`, wobei key den Wert "ATOSJ_COMPONENT_NAME_KEY" erhält. 2. Der Rückgabewert der Funktion `toString()` des Menüs.	1. Der Name des Menüs. Festgelegt durch den Programmierer und abrufbar über die Funktion `getName()` aus der Klasse `java.awt.Component`. 2. Der Text des Menüs, wie er zur Darstellung gesetzt wurde.
`WINDOW`	1. Der Titel des Fensters.	1. Der Titel des Fensters.

`<AKTION>`: Folgende Aktionen können auf einem Component, in Abhängigkeit von `<COMPONENTTYP>`, durchgeführt werden. Es ist zu beachten, dass alle Aktionsarten für den Typ `COMPONENT` auch für alle anderen Typen gültig sind.

COMPONENT	PRESSKEY, $Zeichen$ [,ALT] [,CTRL] [,SHIFT]	Setzt den Eingabefokus auf das Component und simuliert einen Tastendruck mit dem angebenen Zeichen und optional mit den Tasten Alt (ALT), Steuerung (CTRL) und Umschalt (SHIFT).
COMPONENT	CLICK, LEFT | RIGHT, <+Klickanzahl+> [,ALT] [,CTRL] [,SHIFT]	Simuliert einen Mausklick entweder mit der linken Taste (LEFT) oder mit der rechten Taste (RIGHT). Es werden <+Klickanzahl+> Klicks ausgeführt. Optional kann gleichzeitig das Drücken der Steuertasten Alt (ALT), Steuerung (CTRL) und Umschalt (SHIFT) simuliert werden.
$+Custom+$	PROPERTY, $Eigenschaft$, $Wert$	Setzt die Eigenschaft eines Custom-Components auf den angegegebenen Wert. Der Typname $+Custom+$ wird in der Datei custom.lst festgelegt.
EDITBOX	EDIT, $Text$	Setzt den Eingabefokus auf das Textfeld und simuliert die Eingabe von $Text$
CHECKBOX	CHECK	Wählt die Checkbox an, sofern sie vorher nicht schon angewählt war.
CHECKBOX	UNCHECK	Wählt die Checkbox ab, sofern sie nicht schon vorher abgewählt war.
CHECKBOX	SELECT	Simuliert einen einfachen Mausklick mit der linken Taste, ohne Rücksicht auf den vorhergehenden Status. Dies führt zu einem Umschalten auf den jeweils komplementären Anwahlzustand.
RADIOBUTTON	CHECK	Wählt den Radiobutton an, sofern er vorher nicht schon angewählt war.
RADIOBUTTON	SELECT	Simuliert einen einfachen Mausklick mit der linken Taste, ohne Rücksicht auf den vorhergehenden Status.
BUTTON	SELECT	Simuliert einen einfachen Mausklick mit der linken Taste.
COMBOBOX	SELECT, SUBITEM, <+Listenindex+>	Wählt den Eintrag an Position <+Listenindex+> aus der Liste der Combobox aus.
COMBOBOX	EDIT, $Text$	Setzt den Eingabefokus auf die Combobox und simuliert die Eingabe von $Text$.
LIST	SELECT, SUBITEM, <+Listenindex+>	Setzt die Auswahl der Liste auf die Zeile an Position <+Listenindex+>.
LIST	ADDSELECT, SUBITEM, <+Listenindex+>	Fügt der Auswahl der Liste, die Zeile an Position <+Listenindex+> hinzu, sofern noch nicht in der bisherigen Auswahl enthalten.

LIST	DESELECT, SUBITEM, <+Listenindex+>	Entfernt die Zeile an Position <+Listenindex+> aus der aktuellen Auswahl Liste, sofern enthalten.
TABLE	SELECT, SUBITEM, <+Zeilenindex+>	Setzt die Auswahl der Tabelle auf das Element an Position <+Zeilenindex+>.
TABLE	ADDSELECT, SUBITEM, <+Zeilenindex+>	Fügt der Auswahl der Tabelle das Element an Position <+Zeilenindex+> hinzu, sofern noch nicht in der bisherigen Auswahl enthalten.
TABLE	DESELECT, SUBITEM, <+Zeilenindex+>	Entfernt das Element an Position <+Zeilenindex+> aus der aktuellen Auswahl Tabelle, sofern enthalten.
TABLE	EDIT, $Text$, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Aktiviert den Zelleneditor für die Zelle an Position <+Zeilenindex+> und <+Spaltenindex+> und simuliert die Eingabe von $Text$.
TABLE	CHECK, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Setzt den Status eines Zelleneditors vom Typ CHECKBOX an Position <+Zeilenindex+> und <+Spaltenindex+> auf angewählt, sofern sie nicht schon vorher angewählt war.
TABLE	UNCHECK, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Setzt den Status eines Zelleneditors vom Typ CHECKBOX an Position <+Zeilenindex+> und <+Spaltenindex+> auf abgewählt, sofern sie nicht schon vorher abgewählt war.
TREE	SELECT, SUBITEM {,$Knoten$}	Setzt die Auswahl des Baumes auf den angegebenen Knoten. Der Pfad beinhaltet die Beschriftung aller Elternknoten und des auszuwählenden Knotens, beginnend beim Wurzelknoten.
TREE	ADDSELECT, SUBITEM {,$Knoten$}	Fügt den angegebenen Knoten der Auswahl des Baumes hinzu, sofern noch nicht enthalten. Der Pfad beinhaltet die Beschriftung aller Elternknoten und des hinzuzufügenden Knotens, beginnend beim Wurzelknoten.
TREE	DESELECT, SUBITEM {,$Knoten$}	Entfernt den angegebenen Knoten aus der Auswahl des Baumes, sofern enthalten. Der Pfad beinhaltet die Beschriftung aller Elternknoten und des abzuwählenden Knotens, beginnend beim Wurzelknoten.
TREE	EDIT, $Text$, SUBITEM {,$Knoten$}	Aktiviert den Zelleneditor für den Knoten mit dem angegebenen Pfad und simuliert die Eingabe von $Text$. Der Pfad beinhaltet die Beschriftung aller Elternknoten und des hinzuzufügenden Knotens, beginnend beim Wurzelknoten.

MENU	SELECT, SUBITEM {,$Menüpunkt$}	Klickt den Menüpunkt mit dem angegeben Pfad an. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des zu klickenden Menüpunktes selbst, beginnend beim Wurzelmenü.
MENU	CHECK, SUBITEM {,$Menüpunkt$}	Aktiviert den Menüpunkt mit dem angegeben Pfad. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des zu aktivierenden Menüpunktes selbst, beginnend beim Wurzelmenü.
MENU	UNCHECK, SUBITEM {,$Menüpunkt$}	Deaktiviert den Menüpunkt mit dem angegeben Pfad. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des zu deaktivierenden Menüpunktes selbst, beginnend beim Wurzelmenü.
TABFOLDER	SELECT, <+Tabindex+>	Wählt den Karteireiter an Position <+Tabindex+> aus.
WINDOW	CLOSE	Schließt das Fenster.
WINDOW	MAXIMIZE	Maximiert die Größe des Fensters.
WINDOW	MINIMIZE	Minimiert die Größe des Fensters.
WINDOW	NORMALIZE	Setzt die Größe des Fensters auf die ursprüngliche Größe.

Beschreibung:
Simuliert Nutzeraktionen auf den angegebenen Components.

Beispiel:
`ACTION, ‘‘Dozent‘‘, TABFOLDER, ‘‘Tab‘‘, SELECT, SUBITEM, ‘‘Seminare‘‘`
Im Fenster mit dem Titel „Dozent“ wird aus der Kartei mit dem Namen „Tab“ der Karteireiter mit der Beschriftung „Seminare“ ausgewählt.

A.2.2 CLEANUP

Aufruf: `CLEANUP`

Parameter: keine

Beschreibung:
Falls während des Testvorgangs ein Fehler auftritt, wird direkt zu diesem Kommando gesprungen, sofern es vorhanden ist und der Ausführungsmodus das Anhalten bei einem Fehler fordert. Die mit `START` aufgerufene Anwendung wird ggf. geschlossen. Alle folgenden Kommandos bis zum Ende der Datei dienen der Wiederherstellung des Zustandes vor Beginn der Ausführung des aktuellen Testskriptes. Temporär angelegte Dateien sollten

hier gelöscht und originale Konfigurationsdateien wieder hergestellt werden, um das Testobjekt in seinen Ausgangszustand zu versetzen. Fehler beim Ausführen von Kommandos nach dem `CLEANUP`-Kommando werden ignoriert.

A.2.3 COMMENT

Aufruf: `COMMENT, <Kommentar>`

Parameter:

`<Kommentar>`: Der verbale Kommentar.

Beschreibung:
Realisiert einen verbalen Kommentar in einer Testsequenz. Dieses Kommando wird bei der Testausführung ignoriert.

Beispiel: `COMMENT, ‘‘Das ist ein Kommentar‘‘`

A.2.4 COMPARE

Aufruf: `COMPARE, <ISTWERT>, <TYP>, <VAR/VAL>, <SOLLWERT> [,<MODUS>]`

Parameter:

`<ISTWERT>`: Ist ein, in „-Zeichen eingeschlossener, Bezeichner für eine Variable, die den zu überprüfenden Istwert (Zahl, Zeichenkette oder boolescher Wert) enthält.

`<TYP>`: Gibt den Datentyp für den Vergleich an. Bei einem numerischen Vergleich von Soll- und Istwert steht an dieser Stelle `NUM`. Für einen Vergleich von Zeichenketten wird stattdessen `STR` verwendet. Das Schlüsselwort `BOOL` steht für einen booleschen Vergleich.

`<VAR/VAL>`: Gibt die Art des Sollwertes an. `VAR` steht für eine Variable und `VAL` für einen festen Wert.

`<SOLLWERT>`: Der Sollwert ist in Abhängigkeit vom Parameter `VAR/VAL` entweder der Bezeichner für Variable oder ein Wert entsprechend des Vergleichstyps, der im Parameter `TYP` angegeben wurde. Der Bezeichner für eine Variable muss eine, in Anführungszeichen eingeschlossene, Zeichenkette sein. Dabei wird das Backslash (\) als Maskierungszeichen verwendet. Ein numerischer Wert wird als reelle Zahl

angegeben. Als Dezimalzeichen dient der Punkt. Eine Zeichenkette wird in Anführungzeichen eingeschlossen. Dabei wird das Backslash (\) als Maskierungszeichen verwendet. Ein boolescher Wert wird entweder mit dem Schlüsselwort `TRUE` für wahr oder `FALSE` für falsch angegeben.

`<MODUS>`: Dieser optionale Parameter bestimmt bei einem numerischen Vergleich den Vergleichsoperator. Folgende Möglichkeiten werden unterstützt:

`EQ`	Istwert und Sollwert müssen gleich sein
`GRT`	Istwert muss größer als der Sollwert sein
`GEQ`	Istwert muss größer-gleich dem Sollwert sein
`LSS`	Istwert muss kleiner als der Sollwert sein
`LEQ`	Istwert muss kleiner-gleich dem Sollwert sein
`NEQ`	Istwert und Sollwert müssen unterschiedlich sein
`TOL, <Tol.Wert>`	Istwert darf maximal um ± `<Tol.Wert>` vom Sollwert abweichen

Beschreibung: Vergleicht Istwerte aus Variablen entweder direkt mit Sollwerten oder mit Werten aus anderen Variablen. Der Vergleich kann zeichenweise, numerisch oder boolesch sein. Dieses Kommando arbeitet eng mit `READ` zusammen, welches das Abspeichern von Werten in Variablen realisiert. Diese können dann zu einem späteren Zeitpunkt unter Verwendung von `COMPARE` mit Sollwerten verglichen werden.

Beispiel:
`COMPARE, ‘‘weiblich‘‘, BOOL, VAL, TRUE`
Es wird geprüft, ob die Variable mit dem Namen „weiblich“ mit dem booleschen Wert „wahr“ belegt ist.
`COMPARE, ‘‘var1‘‘, BOOL, VAR, ‘‘var2‘‘`
Es wird geprüft, ob die booleschen Variablen „var1“ und „var2“ den gleichen Wert besitzen.

A.2.5 COPY

Aufruf: `COPY, <QUELLVERZEICHNIS>, <QUELLDATEI>, <ZIELVERZEICHNIS>, <ZIELDATEI> [,FORCE] [,IFSRCEXISTS]`

Parameter:

`<QUELLVERZEICHNIS>, <ZIELVERZEICHNIS>`: Einige Pfade zu Verzeichnissen sind ATOSj zur Ausführungszeit bekannt und können mit diesen Parametern angesprochen werden. Das ermöglicht die Portabilität von ATOSj-Projekten (bzw. Testskripten), da

sie weitestgehend unabhängig von absoluten Pfadangaben sind. Mögliche Werte sind:

ABS	in <QUELLDATEI> bzw. <ZIELDATEI> steht ein absoluter Pfad
TGT	Ausführungsverzeichnis des Testobjekts
REF	Verzeichnis mit Solldateien (\REF)
BIN	Binary-Verzeichnis des Projektes (\BIN)
ENV	Verzeichnis der Umgebungsdateien des Projektes (\ENV)
LOG	Verzeichnis zur Aufnahme der Logdateien (\LOG)

<QUELLDATEI>, <ZIELDATEI>: Ist ein in "-Zeichen eingeschlossener Dateiname.

FORCE: Diese Direktive ist optional und notwendig, wenn die Zieldatei bereits existiert und versteckt oder schreibgeschützt ist. Ohne FORCE würde das Kopieren in diesem Falle scheitern und einen Fehler verursachen.

IFSRCEXISTS: Dieser optionale Parameter bewirkt, dass nur dann eine Kopieraktion ausgeführt wird, wenn die Quelldatei vorhanden ist. Dadurch werden Fehler unterbunden, wenn die Quelldatei nicht existiert.

Beschreibung: Kopieren von Dateien aus Verzeichnissen eines ATOSj-Projektes oder aus Verzeichnissen mit absoluten Pfaden.

Beispiel:
COPY, REF, ''t1kunden.dat'', TGT, ''kunden.dat'', FORCE
Kopiert die Datei „t1kunden.dat" aus dem Unterverzeichnis „REF" des Projektverzeichnisses in das Ausführungverzeichnis des Testobjekts. Die Kopie erhält den Namen „kunden.dat". Existiert die Zieldatei bereits, wird sie überschrieben.

A.2.6 DELETE

Aufruf: DELETE, <VERZEICHNIS>, <DATEI> [,FORCE] [,IFEXISTS]

Parameter:

<VERZEICHNIS>: Mögliche Werte sind: ABS, TGT, REF, BIN, ENV und LOG (siehe COPY)

<DATEI>: Ist ein in "-Zeichen eingeschlossener Dateiname.

FORCE: Diese Direktive ist optional und notwendig, wenn die Datei versteckt oder schreibgeschützt ist. Ohne FORCE würde das Löschen in diesem Falle scheitern und

einen Fehler verursachen.

`IFEXISTS`: Diese Direktive ist optional und unterbindet einen Fehler, wenn die zu löschende Datei nicht existiert.

Beschreibung: Löschen einer Datei.

Beispiel:
`DELET, TGT, ‘‘kunden.dat‘‘, IFEXISTS`
Löscht die Datei „kunden.dat“ aus dem Ausführungsverzeichnis des Testobjektes. Existiert die Datei nicht, so wird dies ignoriert.

A.2.7 DISABLE

Aufruf: `DISABLE, <HTS-Kommando>`

Parameter:

`<HTS-Kommando>`: Ein beliebiges anderes HTS-Kommando, das beim Testvorgang nicht ausgeführt werden soll.

Beschreibung: Dient zur Deaktivierung eines HTS-Kommandos. Wird in ATOSj benötigt, um Kommandos zu Testzwecken zu deaktivieren. Deaktivierte Kommandos sollten nicht teil einer vollendeten Testsequenz sein. Das deaktivierte Kommando wird beim Testvorgang nicht ausgeführt.

Beispiel:
`DISABLE, WAIT, 500`
Das `WAIT` Kommando wird nicht ausgeführt.

A.2.8 ENDLOOP

Aufruf: `ENDLOOP`

Parameter: keine

Beschreibung: Bildet das Ende eines `LOOP`-Schleifenblocks.

Beispiel: siehe Kommando `LOOP`

A.2.9 ENDWHILE

Aufruf: `ENDWHILE`

Parameter: keine

Beschreibung: Bildet das Ende eines `WHILE`-Schleifenblocks.

Beispiel: siehe Kommando `WHILE`

A.2.10 EXISTS

Aufruf: `EXISTS, <VERZEICHNIS>, <ZIEL> [,DIRECTORY]`

Parameter:

`<Verzeichnis>`: MöglicheWerte sind: `ABS, TGT, REF, BIN, ENV` und `LOG` (siehe `COPY`)

`<ZIEL>`: Ist ein in "-Zeichen eingeschlossener Datei- oder Verzeichnisname.

`DIRECTORY`: Dieser optionale Parameter zeigt an, dass es sich bei `<ZIEL>` um ein Verzeichnis und nicht um eine Datei handelt.

Beschreibung: Überprüft die Existens einer Datei oder eines Verzeichnisses.

Beispiel:
`EXISTS, ABS, ''C:\\DATA'', ''kunden.dat''`
Überprüft ob im angegebenen Verzeichnis die Datei „kunden.dat" existiert.

A.2.11 LAUNCH

1. Warten auf Terminierung

Aufruf: `LAUNCH, <VERZEICHNIS>, <PROGRAMMNAME>, <PARAMETER>, FOREVER, <RETURNCODE>, [,<LOGFILE>]`

Parameter:

`<Verzeichnis>`: MöglicheWerte sind: `ABS, TGT, REF, BIN, ENV` und `LOG` (siehe `COPY`)

`<PROGRAMMNAME>`: Ist ein in "-Zeichen eingeschlossener Dateiname.

`PARAMETER`: Sind in "-Zeichen eingeschlossene Parameter zur Übergabe an das zu startende Programm.

`<RETURNCODE>`: Der Rückkehrcode des aufgerufenen Programmes wird mit `<RETURNCODE>` verglichen. Bei Nichtübereinstimmung wird wird ein Fehler gemeldet.

`<LOGFILE>`: Ist ein in "-Zeichen eingeschlossener Dateiname. Die Ausgabedatei (Protokolldatei o.ä.) des aufgerufenen Programmes wird durch Angabe ihres Namens in `<LOGFILE>` aus dem Ausführungsverzeichnis des Programms in das `LOG`-Verzeichnis des ATOSj-Projekts verschoben.

Beschreibung: Startet ein externes Programm für testbegleitende Aufgaben (Dateivergleicher, Grafikviewer etc.), wartet auf dessen Terminierung und schreibt gegebenenfalls ein Logfile in das `LOG`-Verzeichnis des ATOSj-Projektes.

Beispiel:
`LAUNCH, ABS, ‘‘mysql‘‘, ‘‘-e\‘‘Drop database husemorg;\‘‘‘‘, FOREVER, 0`
Startet das Programm „mysql“ mit den angegebenen Parametern und wartet unbegrenzte Zeit auf dessen Beendigung. Überprüft ob der Rückgabewert des Programmes dem Wert 0 entspricht.

2. Warten auf Zeit

Aufruf: `LAUNCH, <VERZEICHNIS>, <PROGRAMMNAME>, <PARAMETER>, TIME, <TERMINIERUNGSDAUER>, <RETURNCODE>, [,<LOGFILE>]`

Parameter:

`<Terminierungsdauer>`: Dauer in Millisekunden, nach der das aufgerufene Programm beendet sein muss. Ist das Programm nach Ablauf dieser Dauer nicht fertig, wird ein Fehler gemeldet.

Alle anderen Parameter wie bei 1.

Beschreibung: Startet ein externes Programm für testbegleitende Aufgaben (Dateivergleicher, Grafikviewer etc.), wartet auf dessen Terminierung und schreibt gegebenenfalls ein Logfile in das `LOG`-Verzeichnis des ATOSj-Projektes.

Beispiel:
`LAUNCH, ABS, ‘‘mysql‘‘, ‘‘-e\‘‘Drop database husemorg;\‘‘‘‘, TIME, 2000, 1`
Startet das Programm „mysql“ mit den angegebenen Parametern und wartet 2 Sekunden

auf dessen Beendigung. Überprüft ob der Rückgabewert des Programmes dem Wert 1 entspricht.

3. Nicht auf Terminierung warten

Aufruf: `LAUNCH, <VERZEICHNIS>, <PROGRAMMNAME>, <PARAMETER>, NOWAIT`

Parameter: Parameter wie bei 1.

Beschreibung: Startet ein externes Programm für testbegleitende Aufgaben (Dateivergleicher, Grafikviewer etc.), und wartet nicht auf dessen Terminierung.

Beispiel:
```
LAUNCH, ABS, ‘‘mysql‘‘, ‘‘-e\‘‘Drop database husemorg;\‘‘‘‘, NOWAIT
```
Startet das Programm „mysql“ mit den angegebenen Parametern. Es wird sofort mit der Ausführung der Testsequenz fortgefahren.

A.2.12 LOOP

Aufruf: `LOOP, <ZYKLEN>`

Parameter:

`<ZYKLEN>`: Gibt die Anzahl der Schleifendurchläufe an.

Beschreibung: Bildet den Anfang eines Schleifenblocks. Alle Kommandos zwischen `LOOP` und `ENDLOOP` werden bei jedem Zyklus wiederholt. Beliebig tiefe Verschachtelungen der Schleifenblöcke sind zulässig.

Beispiel:
```
LOOP, 10
ACTION, ‘‘Mitglieder‘‘, BUTTON, ‘‘add‘‘, SELECT
ENDLOOP
```
Führt 10 Mal hintereinander das `ACTION` Kommando aus.

A.2.13 MESSAGE

Aufruf: `MESSAGE, <NACHRICHT>`

Parameter:

`<NACHRICHT>`: Eine in „-Zeichen eingeschlossene Nachricht, die in einem Dialog angezeigt wird.

Beschreibung: Zeigt einen Dialog mit einer frei definierbaren Nachricht an den Tester. Das Testobjekt kann während der Existenz dieses Dialogs manipuliert werden. Nach dem Drücken des Buttons “OK“ wird mit der Testausführung fortgefahren.

Beispiel:
`MESSAGE, ‘‘Zeichnen Sie einen Kreis!‘‘`
Öffnet ein Fenster in ATOSj mit der angegebenen Aufforderung an den Tester.

A.2.14 QUESTION

Aufruf: `QUESTION, <FRAGE>, <ANTWORT>`

Parameter:

`<FRAGE>`: Eine in „-Zeichen eingeschlossene Frage, die in einem Dialog angezeigt wird.

`<ANTWORT>`: Die erwartete Antwort auf die gestellte Frage. Mögliche Werte sind `YES` für eine Antwort mit „Ja“ und `NO` für eine Antwort mit „Nein“.

Beschreibung: Zeigt einen Dialog mit einer frei definierbaren Ja/Nein-Frage. Erst nach Beantwortung der Frage, durch Anklicken des „Ja“- oder „Nein“-Buttons, wird mit der Testdurchführung fortgefahren. Stimmt die Antwort des Testers nicht mit dem in `<ANTWORT>` definierten Wert überein, wird ein Fehler gemeldet.

Beispiel:
`QUESTION, ‘‘Sehen Sie in der Statusanzeige einen grünen Punkt?‘‘, YES`
Öffnet einen Ja/Nein-Dialog in ATOSj mit der angegebenen Frage. Die erwartete Antwort ist „Ja“.

A.2.15 READ

Aufruf: `READ, <FENSTER>, <COMPONENTTYP>, <COMPONENTNAME>, <ZUSTAND>, <VARIABLE>`

Parameter:

`<FENSTER>`: Der initiale Titel des Fensters in dem sich das Component befindet, dessen Zustand ermittelt werden soll. Für das Hauptfenster dient das Schlüsselwort MAIN

als Platzhalter für den tatsächlichen Titel.

<COMPONENTTYP>: Ist der Typ eines Components. Dieser ist eine unabhängige Abstraktion basierend auf der Funktionalität des Components. Folgende Typen sind möglich: COMPONENT, EDITBOX, LABEL, RADIOBUTTON, CHECKBOX, BUTTON, COMBOBOX, LIST, TABLE, TREE, MENU, TABFOLDER und WINDOW.

<COMPONENTNAME>: Der Name des Components. Dieser bildet zusammen mit <FENSTER> und <COMPONENTTYP> einen eindeutigen Identifikator, über den das Component zur Laufzeit ermittelt wird. Der Name wird nach einer Strategie ermittelt, die möglichst gute Klarnamen für das Component liefert, siehe ACTION.

<VARIABLE>: Der Bezeichner für die Variable, die den ausgelesenen Zustandswert speichern soll.

<ZUSTAND>: Folgenden Zustände können, in Abhängigkeit von <COMPONENTTYP>, ausgelesen werden. Es ist zu beachten, dass alle Zustände des Typs COMPONENT auch für alle anderen Typen gültig sind.

COMPONENT	ENABLESTATE	Ermittelt, ob das Component aktiviert ist, d.h. Eingaben verarbeitet. Das Ergebnis ist ein boolescher Wert, mit TRUE für aktiviert und FALSE für deaktiviert.
COMPONENT	FOCUSSTATE	Ermittelt, ob das Component den Eingabefokus besitzt. Das Ergebnis ist ein boolescher Wert, mit TRUE für fokussiert und FALSE für nicht fokussiert.
COMPONENT	VISIBLESTATE	Ermittelt, ob das Component sichtbar ist. Das Ergebnis ist ein boolescher Wert, mit TRUE für sichtbar und FALSE für unsichtbar.
$+Custom+$	PROPERTY, $Eigenschaft$	Liest eine Eigenschaft eines Custom-Components aus. Ergebnis ist eine Zeichenkette. Der Typname $+Custom+$ wird in der Datei custom.lst festgelegt.
EDITBOX	TEXT	Liest den Text eines Textfeldes. Das Ergebnis ist eine Zeichenkette.
EDITBOX	NUM	Liest den Text eines Textfeldes und interpretiert ihn als reelle Zahl. Für die Dezimalstelle wird ein Punkt erwartet.
LABEL	TEXT	analog zu EDITBOX
LABEL	NUM	analog zu EDITBOX
RADIOBUTTON	CHECKSTATE	Ermittelt, ob der Radiobutton angewählt ist. Das Ergebnis ist ein boolescher Wert, mit TRUE für angewählt und FALSE für nicht angewählt.
RADIOBUTTON	TEXT	analog zu EDITBOX
RADIOBUTTON	NUM	analog zu EDITBOX

CHECKBOX	CHECKSTATE	Ermittelt, ob die Checkbox angewählt ist. Das Ergebnis ist ein boolescher Wert, mit TRUE für angewählt und FALSE für nicht angewählt.
CHECKBOX	TEXT	analog zu EDITBOX
CHECKBOX	NUM	analog zu EDITBOX
BUTTON	TEXT	analog zu EDITBOX
BUTTON	NUM	analog zu EDITBOX
COMBOBOX	ITEMCOUNT	Ermittelt die Anzahl der Einträge in der Liste der Combobox. Ergebnis ist ein ganzzahliger Wert.
COMBOBOX	TEXT	Liest den Text aus dem Textfeld der Combobox. Ergebnis ist eine Zeichenkette.
COMBOBOX	NUM	Liest den Text aus dem Textfeld der Combobox und interpretiert ihn als reelle Zahl. Für die Dezimalstelle wird ein Punkt erwartet.
LIST	ITEMCOUNT	Ermittelt die Anzahl der Einträge in der Liste. Ergebnis ist ein ganzzahliger Wert.
LIST	TEXT, SUBITEM, <+Listenindex+>	Liest den Text des Eintrages in der Zeile <+Listenindex+>.
TABLE	ITEMCOUNT	Ermittelt die Anzahl der Zeilen in der Tabelle. Ergebnis ist ein ganzzahliger Wert.
TABLE	TEXT, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Liest den Text der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+>.
TABLE	NUM, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Liest den Text der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+> und interpretiert ihn als reelle Zahl. Für die Dezimalstelle wird ein Punkt erwartet.
TABLE	CHECKSTATE, SUBITEM, <+Zeilenindex+>, <+Spaltenindex+>	Ermittelt, ob die Checkbox in der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+> angewählt ist. Das Ergebnis ist ein boolescher Wert, mit TRUE für angewählt und FALSE für nicht angewählt.
MENU	ENABLESTATE, SUBITEM {,$Menüpunkt$}	Ermittelt, ob der Menüpunkt mit dem angegebenen Pfad aktiviert ist. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.
MENU	CHECKSTATE, SUBITEM {,$Menüpunkt$}	Ermittelt, ob der Menüpunkt mit dem angegebenen Pfad angewählt ist. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.
TABFOLDER	ITEMCOUNT	Ermittelt die Anzahl an Karteireitern.

TABFOLDER	TEXT, SUBITEM, <+Tabindex+>	Liest den Text des Karteireiters an Position <+Tabindex+>.
WINDOW	siehe COMPONENT	

Beschreibung: Abfragen und Zwischenspeichern von Zuständen und Werten eines Components in einer Variable. Der Wert der Variable kann zu einem späteren Zeitpunkt, unter Verwendung des Kommandos COMPARE, mit einer anderen Variable oder einem festen Wert verglichen werden.

Beispiel:
READ, ‘‘Dozentliste‘‘, TABLE, ‘‘Dozenten‘‘, TEXT, ‘‘name‘‘, SUBITEM, 0, 0
Liest den Text der Tabellenzelle in Zeile 0, Spalte 0 der Tabelle mit dem Namen „Dozenten“ im Fenster „Dozentenliste“ aus und speichert den ermittelten Wert in der Variable „name“.

A.2.16 START

Aufruf: START, <PARAMETER>

Parameter:

<PARAMETER>: Eine Zeichenkette mit Parametern, die dem Testobjekt beim Start übergeben werden.

Beschreibung: Startet das Testobjekt und wartet auf das Erscheinen des Hauptfensters. Der Titel des Hauptfensters, die Main-Klasse und der Klassenpfad für das Testobjekt werden in den Projektparametern von ATOSj definiert. Hat sich das Hauptfenster nach Ablauf einer festgelegten Wartezeit nicht geöffnet oder kann das Programm nicht gestartet werden, wird ein Fehler gemeldet.

Beispiel:
START, ‘‘-exclusive kunden.dat‘‘
Startet das Testobjekt und übergibt die angegebenen Programmparameter.

A.2.17 TEST

Aufruf: TEST, <FENSTER>, <COMPONENTTYP>, <COMPONENTNAME>, <SOLLWERTVERGLEICH>

Parameter:

<FENSTER>: Der initiale Titel des Fensters in dem sich das Component befindet, dessen Zustand ermittelt werden soll. Für das Hauptfenster dient das Schlüsselwort MAIN als Platzhalter für den tatsächlichen Titel.

<COMPONENTTYP>: Ist der Typ eines Components. Dieser ist eine unabhängige Abstraktion basierend auf der Funktionalität des Components. Folgende Typen sind möglich: COMPONENT, EDITBOX, LABEL, RADIOBUTTON, CHECKBOX, BUTTON, COMBOBOX, LIST, TABLE, TREE, MENU, TABFOLDER und WINDOW.

<COMPONENTNAME>: Der Name des Components. Dieser bildet zusammen mit <FENSTER> und <COMPONENTTYP> einen eindeutigen Identifikator, über den das Component zur Laufzeit ermittelt wird. Der Name wird nach einer Strategie ermittelt, die möglichst gute Klarnamen für das Component liefert, siehe ACTION.

<SOLLWERTVERGLEICH>: Zu jedem Typ gibt es eine Menge von Zuständen, die mit Sollwerten verglichen werden können. Folgende Zustandsprüfungen können, in Abhängigkeit von <COMPONENTTYP>, vorgenommen werden. Es ist zu beachten, dass alle Zustandsprüfungen für den Typ COMPONENT auch für alle anderen Typen gelten.

COMPONENT	ENABLESTATE, TRUE	Überprüft, ob das Component aktiviert ist.
COMPONENT	ENABLESTATE, FALSE	Überprüft, ob das Component deaktiviert (ausgegraut) ist.
COMPONENT	FOCUSSTATE, TRUE	Überprüft, ob das Component den Eingabefokus hat.
COMPONENT	FOCUSSTATE, FALSE	Überprüft, ob das Component nicht den Eingabefokus hat.
COMPONENT	VISIBLESTATE, TRUE	Überprüft, ob das Component sichtbar ist.
COMPONENT	VISIBLESTATE, FALSE	Überprüft, ob das Component unsichtbar ist.
$+Custom+$	PROPERTY, $Eigenschaft$, $Sollwert$	Überprüft, ob der Eigenschaftswert des Custom-Components dem Sollwert entspricht. Es wird nur auf Gleichheit geprüft. Der Typname $+Custom+$ wird in der Datei custom.lst festgelegt.
EDITBOX	TEXT, $Text$	Überprüft, ob der Text in der Editbox mit $Text$ übereinstimmt.
EDITBOX	NUM, <.Wert.> [,<MODUS>]	Liest den Text einer Editbox interpretiert ihn als reelle Zahl und vergleicht ihn mit <.Wert.> entsprechend des angegeben <MODUS>.
LABEL	TEXT, $Text$	analog zu EDITBOX
LABEL	NUM, <.Wert.> [,<MODUS>]	analog zu EDITBOX
RADIOBUTTON	CHECKSTATE, TRUE	Überprüft, ob der Radiobutton angewählt ist.
RADIOBUTTON	CHECKSTATE, FALSE	Überprüft, ob der Radiobutton abgewählt ist.
RADIOBUTTON	TEXT, $Text$	analog zu EDITBOX

RADIOBUTTON	NUM, <.Wert.> [,<MODUS>]	analog zu EDITBOX
CHECKBOX	CHECKSTATE, TRUE	Überprüft, ob die Checkbox angewählt ist.
CHECKBOX	CHECKSTATE, FALSE	Überprüft, ob die Checkbox abgewählt ist.
CHECKBOX	TEXT, $Text$	analog zu EDITBOX
CHECKBOX	NUM, <.Wert.> [,<MODUS>]	analog zu EDITBOX
BUTTON	TEXT, $Text$	analog zu EDITBOX
BUTTON	NUM, <.Wert.> [,<MODUS>]	analog zu EDITBOX
COMBOBOX	ITEMCOUNT, <+Wert+>, [,<MODUS>]	Vergleicht die Anzahl der Einträge in der Liste der Combobox mit <+Wert+> entsprechend des angegebenen <MODUS>.
COMBOBOX	TEXT, $Text$	analog zu EDITBOX
COMBOBOX	NUM, <.Wert.> [,<MODUS>]	analog zu EDITBOX
LIST	ITEMCOUNT, <+Wert+> [,<MODUS>]	Vergleicht die Anzahl der Einträge in der Liste mit <+Wert+> entsprechend des angegebenen <MODUS>.
LIST	TEXT, $Text$, SUBITEM, <+Listenindex+>	Überprüft, ob der Text des Eintrages in Zeile <+Listenindex+> mit $Text$ übereinstimmt.
TABLE	ITEMCOUNT, <+Wert+> [,<MODUS>]	Vergleicht die Anzahl der Zeilen in der Tabelle mit <+Wert+> entsprechend des angegebenen <MODUS>.
TABLE	TEXT, $Text$, SUBITEM, <+Zeilenindex+>, <+Spaltenindex>	Überprüft, ob der Text der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+> mit $Text$ übereinstimmt.
TABLE	NUM, <.Wert.> [,<MODUS>], SUBITEM, <+Zeilenindex+>, <+Spaltenindex>	Liest den Text der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+>, interpretiert ihn als reelle Zahl und vergleicht ihn mit <.Wert.> entsprechend des angegeben <MODUS>.
TABLE	CHECKSTATE, TRUE, SUBITEM, <+Zeilenindex+>, <+Spaltenindex>	Überprüft, ob die Checkbox in der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+> angewählt ist.
TABLE	CHECKSTATE, FALSE, SUBITEM, <+Zeilenindex+>, <+Spaltenindex>	Überprüft, ob die Checkbox in der Tabellenzelle an Position <+Zeilenindex+>, <+Spaltenindex+> abgewählt ist.
MENU	ENABLESTATE, TRUE, SUBITEM {,$Menüpunkt$}	Überprüft, ob der Menüpunkt mit dem angegebenen Pfad aktiviert ist. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.
MENU	ENABLESTATE, FALSE, SUBITEM {,$Menüpunkt$}	Überprüft, ob der Menüpunkt mit dem angegebenen Pfad deaktiviert ist. Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.

MENU	CHECKSTATE, TRUE, SUBITEM {,$Menüpunkt$}	Überprüft, ob der Menüpunkt mit dem angegebenen Pfad angewählt ist (mit Häkchen). Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.
MENU	CHECKSTATE, FALSE, SUBITEM {,$Menüpunkt$}	Überprüft, ob der Menüpunkt mit dem angegebenen Pfad abgewählt ist (ohne Häkchen). Der Pfad beinhaltet die Beschriftung aller übergeordneten Menüs und des gesuchten Menüpunktes selbst, beginnend beim Wurzelmenü.
TABFOLDER	ITEMCOUNT, <+Wert+> [,<MODUS>]	Vergleicht die Anzahl der Karteireiter mit <+Wert+> entsprechend des angegebenen <MODUS>.
TABFOLDER	TEXT, $Text$, SUBITEM, <+Tabindex+>	Überprüft, ob der Text des Karteireiters an Position <+Tabindex+> mit $Text$ übereinstimmt.
WINDOW	siehe COMPONENT	

Beschreibung: Fragt Zustände von Components ab und vergleicht sie mit den angegebenen Sollwerten. Stimmen der ausgelesene Wert eines Zustands und der Sollwert nicht überein, wird ein Fehler gemeldet.

Beispiel:
`TEST, ‘‘Mitglieder‘‘, BUTTON, ‘‘add‘‘, ENABLESTATE, TRUE`
Überprüft, ob der Knopf mit dem Namen „add“ im Fenster „Mitglieder“ aktiviert ist.

A.2.18 WAIT

Aufruf: `WAIT, <TIMEOUT>`

Parameter:

`<TIMEOUT>`: Gibt die Dauer in Millisekunden an, die gewartet werden soll.

Beschreibung: Wartet die in `<TIMEOUT>` angegebene Zeit bevor mit der Ausführung des Testskriptes fortgefahren wird.

Beispiel:
`WAIT, 1500`
Hält die Ausführung der Testsequenz wird für 1,5 Sekunden an.

A.2.19 WINDOWEXISTS

Aufruf: `WINDOWEXISTS, <FENSTER>, <ABFRAGE>`

Parameter:

`<FENSTER>`: Der initiale Titel des Fensters, dessen Existenz übepüft werden soll. Für das Hauptfenster dient das Schlüsselwort MAIN als Platzhalter für den tatsächlichen Titel.

`<ABFRAGE>`: Soll überprüft werden, ob ein Fenster momentan existiert bzw. sichtbar ist, muss YES verwendet werden, andernfalls NO.

Beschreibung: Abfrage der Existenz bzw. Sichtbarkeit eines Fensters. Entspricht der ermittelte Wert nicht der `<ABFRAGE>` dann wird ein Fehler gemeldet.

Beispiel:
`WINDOWEXISTS, ‘‘Datensatz speichern‘‘, NO`
Überprüft, ob das Fenster mit dem Namen „Datensatz speichern“ nicht existiert.

B ACover Pflichtenheft

Version	Autor	Datum	Status	Kommentar
1.1	Nicos Tegos	21.09.06	Ersterfassung ergänzt	

B.1 Zielbestimmung

Das Produkt soll eine Methodenüberdeckungsanalyse für eine Zielanwendung durchführen, die Ergebnisse speichern und visualisieren.

B.1.1 Mußkriterien

- Ermitteln der Methoden-, Paket- und Systemüberdeckung mit ihren jeweiligen Maßzahlen.
- Für die Instrumentierung des Testobjektes wird kein Quelltext benötigt.
- Die Instrumentierung des Testobjektes soll zur Laufzeit erfolgen.
- Die ermittelten Daten sollen in einer portablen XML-Datei gespeichert werden.
- Die gespeicherten Daten sollen für eine Visualisierung der Ergebnisse aufbereitet werden.

B.1.2 Abgrenzungskriterien

Die Ermittlung anderer Maßzahlen, wie C_0, C_1 und C_2, ist nicht nötig. Die Überdeckungsmaße werden ausschließlich für Java-Programme ermittelt.

B.2 Produkteinsatz

Das Produkt dient zur Analyse der Methodenüberdeckung.

B.2.1 Anwendungsbereich

Die Methodenüberdeckung wird im Rahmen des Softwaretest ermittelt. Sie gibt an welche Methoden der Klassen eines Testobjektes während der Ausführung von Testfällen tatsächlich aufgerufen wurden. Sie liefert somit ein Maß für die Vollständigkeit der Testfälle und dient als Approximation der Funktionsüberdeckung. Da die Methodenüberdeckung nur sehr grobgranular ist, eignet sie sich besonders im Rahmen des Oberflächentests mit einem „Capture and Replay"-Werkzeug.

B.2.2 Zielgruppen

Tester der Zielanwendung.

B.3 Produktumgebung

B.3.1 Hardware

Das Produkt läuft auf einem Arbeitsplatzrechner.

B.3.2 Software

Lauzeitumgebung: Java ab Version 1.5.2
Betriebssystem: beliebig
Webbrowser zum Betrachten des generierten HTML-Reports.

B.3.3 Produktschnittstellen

Die ermittelten Daten zur Durchführung der Überdeckungsanalyse werden in einer XML-Datei abgelegt. Diese kann von anderen Programmen zur Ermittlung der Maßzahlen und zur Visualisierung genutzt werden.

B.4 Produktfunktionen

/F10/ Erstellen einer Datenbasis über alle Klassen und Methoden nutzerspezifizierter Pakete der Zielanwendung.

/F20/ Lauzeitinstrumentierung der Zielanwendung zur Ermittlung aller Aufrufe, der in der Datenbasis enthaltenen Funktionen.

/F30/ Aktualisieren der Datenbasis nach jedem Durchlauf der Zielanwendung.

/F40/ Berechnung der Maßzahlen C_{-1}, C_{-2} und C_{-3} anhand, der in der Datenbasis gesammelten Informationen. Festschreiben der Ergebnisse in einer HTML-Datei.

B.5 Produktdaten

/D10/ XML-Datenbasis, beinhaltet alle Klassen der Zielanwendung mit allen enthaltenen und aufrufbaren Methoden. Zu jeder Klasse ist der vollqualifizierte Java-Name zu speichern. Zu jeder Methode einer Klasse sind deren Modifier (glossar), der vollqualifizerte Name der Klasse des Rückgabewertes, die vollqualifizierten Klassennamen der Parameter und die vollqualifizierten Klassennamen der möglichen Ausnahmen. Zu jeder Methode ist insbesondere die Anzahl ihrer bisherigen Aufrufe zu speichern.

C Beispiele

C.1 Modultest in JUnit

Das folgende Beispiel soll die Arbeit mit dem Testframework JUnit verdeutlichen. Es zeigt die Erstellung eines Testfalles für die Klasse `Vector` unter Verwendung von Klassen und Methoden des Testframeworks.

```
import java.util.Vector;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import junit.swingui.TestRunner;

public class VectorTest extends TestCase
{
    private Vector v;
    public VectorTest(String name) {
        super(name);
    }

    protected void setUp() throws Exception {
        v = new Vector();
    }

    protected void tearDown() throws Exception {
        v.clear();
    }

    public void testAdd() {
        v.add("Hallo");
        assertFalse( "Der Vector darf nicht leer sein.", v.isEmpty() );
    }

    public void testContains() {
        v.add("Hallo");
        assertTrue( v.contains("Hallo") );
    }

    public static Test suite() {
        TestSuite vTest = new TestSuite("Vector-Test");
        vTest.addTest( new VectorTest("testAdd") );
        vTest.addTest( new VectorTest("testContains") );

        return vTest;
    }
```

```
    public static void main(String[] args) {
        TestRunner.run(VectorTest.class);
    }
}
```

C.2 XML-Schema der ACover-Datenbasis

Das folgende XML-Schema zeigt die Struktur einer XML-Datenbasis, die durch das Modul ACover für die Methodenüberdeckungsanalyse eines Java-Programms erstellt wurde.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:annotation>
  <xsd:documentation xml:lang="DE">
    Schema für die Datenbasis der
    Überdeckungsanalyse mit ACover
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="coverage">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="class" minOccurs="0" maxOccurs="unbounded"
       type="classType"/>
    </xsd:sequence>
    <xsd:attribute name="jar" type="xsd:string" use="required" />
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="classType">
  <xsd:sequence>
    <xsd:element name="method" minOccurs="0" maxOccurs="unbounded"
    type="methodType" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required" />
</xsd:complexType>

<xsd:complexType name="methodType">
  <xsd:all>

  <xsd:element name="modifiers">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="modifier" minOccurs="0"
         maxOccurs="unbounded">
          <xsd:complexType>
```

```
            <xsd:attribute name="type" type="modifierType"
            use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="returnType">
    <xsd:complexType>
      <xsd:attribute name="type" type="xsd:string" use="required" />
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="name">
    <xsd:complexType>
      <xsd:attribute name="value" type="xsd:string" use="required" />
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="parameters">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="param" minOccurs="0"
        maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="type" type="xsd:string"
            use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="exceptions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="ex" minOccurs="0" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:attribute name="type" type="xsd:string"
            use="required" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  </xsd:all>
```

```
  <xsd:attribute name="runs" type="runsType" use="required" />
</xsd:complexType>

<xsd:simpleType name="modifierType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="final" />
    <xsd:enumeration value="private" />
    <xsd:enumeration value="protected" />
    <xsd:enumeration value="public" />
    <xsd:enumeration value="static" />
    <xsd:enumeration value="synchronized" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="runsType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="0" />
  </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

C.3 Custom Control des CalendarControls

```
package semorg.atosj;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Vector;

import semorg.gui.util.CalendarControl;
import semorg.gui.util.CalendarControl.SWTCalendarEvent;
import semorg.gui.util.CalendarControl.SWTCalendarListener;
import semorg.gui.util.CalendarControl.SWTCalendarPopupListener;
import atosj.component.ComponentId;
import atosj.component.ICommandCreator;
import atosj.component.swt.SWTButton;
import atosj.component.swt.SWTControl;
import atosj.component.swt.SWTThreadSafeAction;
import atosj.exception.ComponentException;
import atosj.model.hts.parser.HTSSyntax;
import atosj.model.recorder.IHTSCommandReceiver;

public class CalendarControlWrapper extends SWTControl   {

   private static SimpleDateFormat formatter =
```

```
        new SimpleDateFormat("dd. MM. yyyy");

    public CalendarControlWrapper(CalendarControl component,
        ComponentId id) {
       super(component, id);
    }

    private CalendarControl getCalendarControl() {
       return (CalendarControl)getComponent();
    }

    // Simulation von Nutzeraktionen
    public void setProperty(String property, String value)
        throws ComponentException {

       //Setzen des Hakens im Auswahlfeld
       if( property.equals("activ") )   {
          if (getCalendarControl().isPopupOpen())
             throw new ComponentException("Das angegebene
                CalenderPopup ist bereits geöffnet, daher kann
                das Property \"activ\" nicht gesetzt werden.");

          final boolean boolVal;
          if (value.equalsIgnoreCase("true")) {
             boolVal = true;
          }
          else if (value.equalsIgnoreCase("false")) {
             boolVal = false;
          }
          else
             throw new ComponentException("Der angegebene Wert muss
                true oder false sein.");

          new SWTThreadSafeAction(){
             protected Object performAction() throws Throwable {
                getCalendarControl().setActivated(boolVal);
                return null;
             }}.go();
       }

       //Oeffnen bzw Schliessen des Popups des Kalenders
       else if( property.equals("open") )  {

          if (value.equalsIgnoreCase("true")) {
             if (getCalendarControl().isPopupOpen())
                throw new ComponentException("Das angegebene
                   CalenderPopup ist bereits geöffnet.");
             else  {
                SWTButton button = new SWTButton(
```

```
                getCalendarControl().getOpenPopupButton(),
                new ComponentId("openButton",
                    HTSSyntax.TYPE_BUTTON,
                    getComponentId().getWindowName()));

            button.select();
         }
      }
      else if (value.equalsIgnoreCase("false")) {
         if (!getCalendarControl().isPopupOpen())
            throw new ComponentException("Das angegebene
               CalenderPopup ist nicht geöffnet.");
         else  {

            new SWTThreadSafeAction(){
               protected Object performAction() throws Throwable {
                  getCalendarControl().closePopup();
                  return null;
               }}.go();
         }

      }
      else
         throw new ComponentException("Der angegebene Wert muss
            true oder false sein.");
   }

   //Einstellen eines neuen Datums
   else if( property.equals("date") )
   {
      try
      {
         final Date date = formatter.parse(value);
         new SWTThreadSafeAction(){

            protected Object performAction() throws Throwable {
               getCalendarControl().setDate(date);
               return null;
            }}.go();

      } catch (ParseException e) {
         throw new ComponentException("Das angegebene Datum muss
            das Format \"dd. MM. yyyy\" haben.");        }
   }
   else
      super.setProperty(property, value);
}

//Auslesen eines Controlzustands
```

```
public String getProperty(String property)
      throws ComponentException {

   //Auslesen ob das Popup momentan geoeffnet ist
   if( property.equals("open") )
      return getCalendarControl().isPopupOpen()? "true" : "false";

   //Auslesen des aktuell eingestellten Datums
   else if( property.equals("date") )
   {
      return formatter.format(getCalendarControl().getDate());
   }
   else
      return super.getProperty(property);
}

//Capturen von Nutzeraktionen
protected class CalendarControlCommandCreator
   extends ControlCommandCreator
   implements SWTCalendarListener,
   SWTCalendarPopupListener
{
   public CalendarControlCommandCreator(
         CalendarControlWrapper comp,
         IHTSCommandReceiver receiver) {
      super(comp, receiver);
      comp.getCalendarControl().addSWTCalendarlistener(this);
      comp.getCalendarControl().addSWTCalendarPopuplistener(this);
   }

   //Festlegen der durchführbaren Testkommandos
   public Vector getTestCommands() {
      Vector commands = super.getTestCommands();

      try {
         Vector values = new Vector(2);
         String val =
         ((CalendarControlWrapper)getMComponent())
                  .getProperty("date");
         values.add("\"date\"");
         values.add( '"' + val + '"' );
         val = ((CalendarControlWrapper)getMComponent())
                  .getProperty("open");
         values.add("\"open\"");
         values.add( '"' + val + '"' );
         val = ((CalendarControlWrapper)getMComponent())
                  .getProperty("activ");
         values.add("\"activ\"");
         values.add( '"' + val + '"' );
```

```
            commands.add(
               createTestCommand(HTSSyntax.READ_PROPERTY,
                     values, null) );

         } catch(ComponentException e){}

         return commands;
      }

      //Erstellen des HTS-Kommandos bei
      //Veraenderung des eingestellten Datums
      public void dateChanged(SWTCalendarEvent event) {
         Vector vec = new Vector();

         Date date = getCalendarControl().getDate();
         if (date!=null)   {
            String valueString = formatter.format(date);
            vec.add("\"date\"");
            vec.add('"'+valueString+'"');
         }
         else  {
            vec.add("\"activ\"");
            vec.add("\"false\"");
         }

         publishActionCommand(HTSSyntax.ACTION_PROPERTY, vec, null);
   }

      //Erstellen des HTS-Kommandos beim
      //Oeffnen oder Schliessen des Popups
      public void popupOpened(SWTCalendarEvent event) {
         Vector vec = new Vector();
         vec.add("\"open\"");
         vec.add("\"true\"");
         publishActionCommand(HTSSyntax.ACTION_PROPERTY, vec, null);
      }

      public void popupClosed(SWTCalendarEvent event) {
         Vector vec = new Vector();
         vec.add("\"open\"");
         vec.add("\"false\"");
         publishActionCommand(HTSSyntax.ACTION_PROPERTY, vec, null);
      }
   }

   public ICommandCreator initCommandCreator(
         IHTSCommandReceiver receiver)
   {
      return new CalendarControlCommandCreator(this, receiver);
```

```
    }

}
```

D HTS-Script zur Seminarorganisation

D.1 Vorbedingungungen.hts

```
START
```

D.2 Neue Firma.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Firma‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘ShortNameText‘‘, EDIT, ‘‘GRS‘‘
ACTION, ‘‘Neu - Firma*‘‘, COMBOBOX, ‘‘SalutationCombo‘‘, EDIT, ‘‘Firma‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘FirstNameText‘‘, EDIT, ‘‘Gebäudereinigung‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘NameText‘‘, EDIT, ‘‘Schmidt‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘StreetText‘‘, EDIT, ‘‘Am Industriepark 1‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘ZipCodeText‘‘, EDIT, ‘‘13484‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CityText‘‘, EDIT, ‘‘Berlin‘‘
ACTION, ‘‘Neu - Firma*‘‘, COMBOBOX, ‘‘CountryCombo‘‘, EDIT, ‘‘Deutschland‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘PhoneText‘‘, EDIT, ‘‘+49 30 394839 0‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘FaxText‘‘, EDIT, ‘‘+49 30 394839 1‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘EmailText‘‘, EDIT, ‘‘info@gr-schimdt.de‘‘
ACTION, ‘‘Neu - Firma*‘‘, COMBOBOX, ‘‘CpSalutationCombo‘‘, EDIT, ‘‘Herr‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpFirstNameText‘‘, EDIT, ‘‘Wolfgang‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpNameText‘‘, EDIT, ‘‘Schmidt‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpPhoneText‘‘, EDIT, ‘‘+49 30 394839 5‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpMobileText‘‘, EDIT, ‘‘+49 170 9083243‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpEmailText‘‘, EDIT, ‘‘wolfgang.schmidt@gr-schmidt.de‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpDepartmentText‘‘, EDIT, ‘‘Geschäftsführung‘‘
ACTION, ‘‘Neu - Firma*‘‘, CALENDAR, ‘‘CpBirthDayCalendar‘‘, PROPERTY, ‘‘open‘‘, ‘‘true‘‘
ACTION, ‘‘Neu - Firma*‘‘, CALENDAR, ‘‘CpBirthDayCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘26. 03. 1955‘‘
ACTION, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘CpTaskText‘‘, EDIT, ‘‘Geschäftsführer‘‘
ACTION, ‘‘Neu - Firma*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
READ, ‘‘Neu - Firma*‘‘, EDITBOX, ‘‘NumberText‘‘, NUM, ‘‘CompanyId‘‘
ACTION, ‘‘Neu - Firma*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.3 Neuer Kunde.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Kunde‘‘
ACTION, ‘‘Neu - Kunde*‘‘, COMBOBOX, ‘‘SalutationCombo‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Neu - Kunde*‘‘, COMBOBOX, ‘‘TitleCombo‘‘, EDIT, ‘‘Dr.‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘FirstNameText‘‘, EDIT, ‘‘Thilo‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘NameText‘‘, EDIT, ‘‘Mahr‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘StreetText‘‘, EDIT, ‘‘Meisenstr. 107‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘ZipCodeText‘‘, EDIT, ‘‘43993‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘CityText‘‘, EDIT, ‘‘Himmelpfort‘‘
ACTION, ‘‘Neu - Kunde*‘‘, COMBOBOX, ‘‘CountryCombo‘‘, EDIT, ‘‘Deutschland‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘PhoneText‘‘, EDIT, ‘‘+49 3433 93402‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘MobileText‘‘, EDIT, ‘‘+49 172 8043248‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘EmailText‘‘, EDIT, ‘‘thilo_mahr@web.com‘‘
ACTION, ‘‘Neu - Kunde*‘‘, CALENDAR, ‘‘BirthDayCalendar‘‘, PROPERTY, ‘‘open‘‘, ‘‘true‘‘
ACTION, ‘‘Neu - Kunde*‘‘, CALENDAR, ‘‘BirthDayCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘11. 08. 1964‘‘
ACTION, ‘‘Neu - Kunde*‘‘, CALENDAR, ‘‘FirstContactCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘16. 08. 2006‘‘
ACTION, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘TaskText‘‘, EDIT, ‘‘Bohnerpersonal‘‘
ACTION, ‘‘Neu - Kunde*‘‘, BUTTON, ‘‘EmployerSelector_opener‘‘, SELECT
```

```
ACTION, ‘‘Firma - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Firma - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘GRS‘‘
ACTION, ‘‘Firma - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
ACTION, ‘‘Firma - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Firma - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Kunde*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
READ, ‘‘Neu - Kunde*‘‘, EDITBOX, ‘‘NumberText‘‘, NUM, ‘‘ClientId‘‘
ACTION, ‘‘Neu - Kunde*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.4 Neuer Dozent.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Dozent‘‘
ACTION, ‘‘Neu - Dozent*‘‘, COMBOBOX, ‘‘SalutationCombo‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Neu - Dozent*‘‘, COMBOBOX, ‘‘TitleCombo‘‘, EDIT, ‘‘Dr.‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘FirstNameText‘‘, EDIT, ‘‘Frank‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘NameText‘‘, EDIT, ‘‘Specht‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘StreetText‘‘, EDIT, ‘‘Schulallee 32 I‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘ZipCodeText‘‘, EDIT, ‘‘13058‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘CityText‘‘, EDIT, ‘‘Berlin‘‘
ACTION, ‘‘Neu - Dozent*‘‘, COMBOBOX, ‘‘CountryCombo‘‘, EDIT, ‘‘Deutschland‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘MobileText‘‘, EDIT, ‘‘0163 3240803‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘EmailText‘‘, EDIT, ‘‘specht@lehrerweb.org‘‘
ACTION, ‘‘Neu - Dozent*‘‘, CALENDAR, ‘‘BirthDayCalendar‘‘, PROPERTY, ‘‘open‘‘, ‘‘true‘‘
ACTION, ‘‘Neu - Dozent*‘‘, CALENDAR, ‘‘BirthDayCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘06. 04. 1971‘‘
ACTION, ‘‘Neu - Dozent*‘‘, CALENDAR, ‘‘FirstContactCalendar‘‘, PROPERTY, ‘‘open‘‘, ‘‘true‘‘
ACTION, ‘‘Neu - Dozent*‘‘, CALENDAR, ‘‘FirstContactCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘01. 08. 2006‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘BioText‘‘, EDIT, ‘‘Spielte u.a. bereits in einer TV-Serie mit.‘‘
ACTION, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘HourlyFeeText‘‘, EDIT, ‘‘20.00‘‘
ACTION, ‘‘Neu - Dozent*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
READ, ‘‘Neu - Dozent*‘‘, EDITBOX, ‘‘NumberText‘‘, NUM, ‘‘LecturerId‘‘
ACTION, ‘‘Neu - Dozent*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.5 Neuer Seminartyp.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Seminartyp‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘ShortTitleText‘‘, EDIT, ‘‘Bohnern‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘TitleText‘‘, EDIT, ‘‘Parkett Bohnern - Richtig gemacht!‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘ObjectiveText‘‘, EDIT, ‘‘Sie wollen Ihren Mitarbeitern...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘MethodologyText‘‘, EDIT, ‘‘Wir stellen jeden ...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘TopicText‘‘, EDIT, ‘‘1. Einführung\n2.-10. Erklärung...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘RoutineText‘‘, EDIT, ‘‘Gearbeitet wird nach dem Motto...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘DurationText‘‘, EDIT, ‘‘90‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘DocumentsText‘‘, EDIT, ‘‘Es werden zu jeder Veranstaltung...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘AudienceText‘‘, EDIT, ‘‘Reingungspersonal‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘RequirementsText‘‘, EDIT, ‘‘Putzabschluss in ...‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘ChargeText‘‘, EDIT, ‘‘350‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘MaxEntrantsText‘‘, EDIT, ‘‘30‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘MinEntrantsText‘‘, EDIT, ‘‘10‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
READ, ‘‘Neu - Seminartyp*‘‘, EDITBOX, ‘‘NumberText‘‘, NUM, ‘‘SemTypeId‘‘
ACTION, ‘‘Neu - Seminartyp*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.6 Neue Veranstaltung.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Firmeninterne Veranstaltung‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘DurationText‘‘, EDIT, ‘‘90‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CALENDAR, ‘‘BeginningDayCalendar‘‘, PROPERTY, ‘‘open‘‘,
‘‘true‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CALENDAR, ‘‘BeginningDayCalendar‘‘, PROPERTY, ‘‘date‘‘,
‘‘01. 01. 2007‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CALENDAR, ‘‘EndingDayCalendar‘‘, PROPERTY, ‘‘open‘‘,
‘‘true‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CALENDAR, ‘‘EndingDayCalendar‘‘, PROPERTY, ‘‘date‘‘,
‘‘30. 06. 2007‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CHECKBOX, ‘‘StartingTimeSpinner_checker‘‘, CHECK
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, SPINNER, ‘‘StartingTimeSpinner_hours‘‘, EDIT, ‘‘13‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CHECKBOX, ‘‘EndingTimeSpinner_checker‘‘, CHECK
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, SPINNER, ‘‘EndingTimeSpinner_hours‘‘, EDIT, ‘‘15‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, CHECKBOX, ‘‘FirstStartingTimeSpinner_checker‘‘, CHECK
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, SPINNER, ‘‘FirstStartingTimeSpinner_hours‘‘, EDIT,
‘‘11‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘LocationText‘‘, EDIT, ‘‘Stadthalle‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘StreetText‘‘, EDIT, ‘‘Dorfstr 34‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘ZipCodeText‘‘, EDIT, ‘‘16565‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘CityText‘‘, EDIT, ‘‘Königs Wusterhausen‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, COMBOBOX, ‘‘CountryCombo‘‘, EDIT, ‘‘Deutschland‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘SemTypeSelector_opener‘‘, SELECT
ACTION, ‘‘Seminartyp - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Seminartyp - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Bohnern‘‘
ACTION, ‘‘Seminartyp - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Seminartyp - Liste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1
ACTION, ‘‘Seminartyp - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Seminartyp - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘PackagePriceText‘‘, EDIT, ‘‘5000‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘SupervisorControl_ChooseItem‘‘, SELECT
ACTION, ‘‘Dozent - Liste‘‘, BUTTON, ‘‘ListFilter_Extender1‘‘, SELECT
ACTION, ‘‘Dozent - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 3
ACTION, ‘‘Dozent - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Frank‘‘
ACTION, ‘‘Dozent - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo1‘‘, SELECT, SUBITEM, 4
ACTION, ‘‘Dozent - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText1‘‘, EDIT, ‘‘Specht‘‘
ACTION, ‘‘Dozent - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Dozent - Liste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1
ACTION, ‘‘Dozent - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Dozent - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, TABFOLDER, ‘‘Tabs‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘InstructorControl_ChooseItem‘‘, SELECT
ACTION, ‘‘Dozent - Liste‘‘, BUTTON, ‘‘ListFilter_Extender1‘‘, SELECT
ACTION, ‘‘Dozent - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 3
ACTION, ‘‘Dozent - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Frank‘‘
ACTION, ‘‘Dozent - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo1‘‘, SELECT, SUBITEM, 4
ACTION, ‘‘Dozent - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText1‘‘, EDIT, ‘‘Specht‘‘
ACTION, ‘‘Dozent - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Dozent - Liste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1
ACTION, ‘‘Dozent - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Dozent - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘&Übernehmen‘‘, SELECT
READ, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, EDITBOX, ‘‘NumberText‘‘, NUM, ‘‘PresentationId‘‘
ACTION, ‘‘Neu - Firmeninterne Veranstaltung*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.7 Neue Firmenbuchung.hts

```
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Ersterfassung‘‘, ‘‘Firmenbuchung‘‘
```

```
ACTION, ‘‘Neu - Firmenbuchung*‘‘, CALENDAR, ‘‘EnrolledCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘21. 08. 2006‘‘
ACTION, ‘‘Neu - Firmenbuchung*‘‘, CALENDAR, ‘‘BilledCalendar‘‘, PROPERTY, ‘‘date‘‘, ‘‘21. 08. 2006‘‘
ACTION, ‘‘Neu - Firmenbuchung*‘‘, BUTTON, ‘‘CompanySelector_opener‘‘, SELECT
ACTION, ‘‘Firma - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Firma - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘GRS‘‘
ACTION, ‘‘Firma - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
ACTION, ‘‘Firma - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Firma - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Firmenbuchung*‘‘, BUTTON, ‘‘PresentationSelector_opener‘‘, SELECT
ACTION, ‘‘Firmeninterne Veranstaltung - Liste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT,
SUBITEM, 1
ACTION, ‘‘Firmeninterne Veranstaltung - Liste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Bohnern‘‘
ACTION, ‘‘Firmeninterne Veranstaltung - Liste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
ACTION, ‘‘Firmeninterne Veranstaltung - Liste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Firmeninterne Veranstaltung - Liste‘‘, BUTTON, ‘‘Auswählen‘‘, SELECT
ACTION, ‘‘Neu - Firmenbuchung*‘‘, BUTTON, ‘‘&OK‘‘, SELECT
```

D.8 Oberflächentests.hts

```
COMMENT, ‘‘Test ob Buttons ausgegraut, andere States abtesten‘‘
COMMENT, ‘‘Firma löschen und Firmenbuchuungsfehler notieren‘‘
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Stammdatenlisten‘‘, ‘‘Firma‘‘
COMMENT, ‘‘Filter in Liste setzen und auswählen‘‘
ACTION, ‘‘Firma - Stammliste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Firma - Stammliste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘GRS‘‘
TEST, ‘‘Firma - Stammliste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECKSTATE, FALSE
ACTION, ‘‘Firma - Stammliste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Firma - Stammliste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECKSTATE, TRUE
TEST, ‘‘Firma - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1, EQ
TEST, ‘‘Firma - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, TEXT, ‘‘GRS‘‘, SUBITEM, 0, 1
TEST, ‘‘Firma - Stammliste‘‘, BUTTON, ‘‘EditItem‘‘, ENABLESTATE, FALSE
ACTION, ‘‘Firma - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
TEST, ‘‘Firma - Stammliste‘‘, BUTTON, ‘‘EditItem‘‘, ENABLESTATE, TRUE
ACTION, ‘‘Firma - Stammliste‘‘, BUTTON, ‘‘EditItem‘‘, SELECT
TEST, ‘‘CompanyWindow‘‘, EDITBOX, ‘‘NumberText‘‘, VISIBLESTATE, TRUE
TEST, ‘‘CompanyWindow‘‘, EDITBOX, ‘‘ShortNameText‘‘, TEXT, ‘‘GRS‘‘
TEST, ‘‘CompanyWindow‘‘, CALENDAR, ‘‘CustomerSinceCalendar‘‘, PROPERTY, ‘‘CHECKSTATE‘‘, ‘‘FALSE‘‘
ACTION, ‘‘CompanyWindow‘‘, WINDOW, ‘‘CompanyWindow‘‘, CLOSE
ACTION, ‘‘Firma - Stammliste‘‘, WINDOW, ‘‘Firma - Stammliste‘‘, CLOSE
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Stammdatenlisten‘‘, ‘‘Kunde‘‘
ACTION, ‘‘Kunde - Stammliste‘‘, BUTTON, ‘‘ListFilter_Extender1‘‘, SELECT
ACTION, ‘‘Kunde - Stammliste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 3
ACTION, ‘‘Kunde - Stammliste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Thilo‘‘
ACTION, ‘‘Kunde - Stammliste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo1‘‘, SELECT, SUBITEM, 4
ACTION, ‘‘Kunde - Stammliste‘‘, EDITBOX, ‘‘ListFilter_FilterText1‘‘, EDIT, ‘‘Mahr‘‘
ACTION, ‘‘Kunde - Stammliste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Kunde - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1, EQ
ACTION, ‘‘Kunde - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Kunde - Stammliste‘‘, BUTTON, ‘‘EditItem‘‘, SELECT
TEST, ‘‘ClientWindow‘‘, BUTTON, ‘‘SubstituteSelector_disconnector‘‘, ENABLESTATE, FALSE
TEST, ‘‘ClientWindow‘‘, CALENDAR, ‘‘BirthDayCalendar‘‘, PROPERTY, ‘‘DATE‘‘, ‘‘11. 08. 1964‘‘
ACTION, ‘‘ClientWindow‘‘, WINDOW, ‘‘ClientWindow‘‘, CLOSE
ACTION, ‘‘Kunde - Stammliste‘‘, WINDOW, ‘‘Kunde - Stammliste‘‘, CLOSE
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Stammdatenlisten‘‘, ‘‘Seminartyp‘‘
ACTION, ‘‘Seminartyp - Stammliste‘‘, COMBOBOX, ‘‘ListFilter_ColumnCombo‘‘, SELECT, SUBITEM, 1
ACTION, ‘‘Seminartyp - Stammliste‘‘, EDITBOX, ‘‘ListFilter_FilterText‘‘, EDIT, ‘‘Bohnern‘‘
ACTION, ‘‘Seminartyp - Stammliste‘‘, CHECKBOX, ‘‘FilterItem‘‘, CHECK
TEST, ‘‘Seminartyp - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 1, EQ
ACTION, ‘‘Seminartyp - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, SELECT, SUBITEM, 0
ACTION, ‘‘Seminartyp - Stammliste‘‘, BUTTON, ‘‘EditItem‘‘, SELECT
```

```
TEST, ‘‘SeminarTypeWindow‘‘, EDITBOX, ‘‘MaxEntrantsText‘‘, NUM, 30, EQ
TEST, ‘‘SeminarTypeWindow‘‘, EDITBOX, ‘‘ChargeText‘‘, NUM, 350.5, LSS
TEST, ‘‘SeminarTypeWindow‘‘, EDITBOX, ‘‘ChargeText‘‘, NUM, 349.01, GEQ
ACTION, ‘‘SeminarTypeWindow‘‘, WINDOW, ‘‘SeminarTypeWindow‘‘, CLOSE
```

D.9 Konsistenztests und Bereinigung.hts

```
TEST, ‘‘Seminartyp - Stammliste‘‘, BUTTON, ‘‘DeleteItem‘‘, ENABLESTATE, TRUE
ACTION, ‘‘Seminartyp - Stammliste‘‘, BUTTON, ‘‘DeleteItem‘‘, SELECT
TEST, ‘‘Seminartyp - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 0, EQ
ACTION, ‘‘Seminartyp - Stammliste‘‘, WINDOW, ‘‘Seminartyp - Stammliste‘‘, CLOSE
ACTION, MAIN, MENU, ‘‘MainMenu‘‘, SELECT, SUBITEM, ‘‘&Stammdatenlisten‘‘, ‘‘Firmeninterne
Veranstaltung‘‘
TEST, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 0, EQ
ACTION, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, BUTTON, ‘‘UpdateItem‘‘, SELECT
TEST, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, TABLE, ‘‘MainTable‘‘, ITEMCOUNT, 0, EQ
ACTION, ‘‘Firmeninterne Veranstaltung - Stammliste‘‘, WINDOW, ‘‘Firmeninterne Veranstaltung -
Stammliste‘‘, CLOSE
ACTION, MAIN, WINDOW, MAIN, CLOSE
CLEANUP
COMMENT, ‘‘Leeren der Datenbank mittels Drop Database möglich, da Tabellen neu erstellt werden‘‘
LAUNCH, ABS, ‘‘C:\\Programme\\MySQL\\MySQL Server 5.0\\bin\\mysql‘‘, ‘‘-usemorg -psemorg -e\‘‘drop
database husemorg;\‘‘‘, FOREVER, 0
```

E Einrichten der ATOSj-Entwicklungsumgebung

Bis zum Zeitpunkt des Einreichens dieser Diplomarbeit wurde ATOSj ausschließlich mittels der Java-Entwicklungsumgebung Eclipse SDK entwickelt. Diese Anleitung beschreibt schrittweise das nötige Vorgehen um eine Projekt unter Eclipse einzurichten. Mittels dieser Hilfestellung sollte es dem Leser möglich sein die Quellen aus dem CVS des Lehrstuhls zu laden und diese anschließend zu kompilieren. Des weiteren wird beschrieben, wie das Programm anschließend bequem aus Eclipse heraus gestartet werden kann. Diese Anleitung wurde für Eclipse 3.2 geschrieben. In neueren Eclipse-Versionen können Anpassungen des Vorgehens nötig sein.

Anleitung:

- laden Sie sich das Eclipse SDK von www.eclipse.org/downloads/ herunter
- Extrahieren sie den Inhalt des geladenen Zip-Archivs in ein neues Verzeichnis, beispielseweise C:\Programme\Eclipse
- Starten Sie die Datei eclipse.exe und erstellen Sie eine neue Eclipse-Arbeitsumgebung (workspace) in einem Verzeichnis Ihrer Wahl.
- Starten Sie Eclipse in der neuen Arbeitsumgebung
- Standardmäßig verwendet Eclipse eine Java Runtime Environment (Java JRE) als Laufzeitumgebung. Für die Kompilierung der ATOSj-Quellen benötigen Sie jedoch das erweiterte Java Development Kit (Java JDK). Falls Sie dies noch nicht installiert haben, laden Sie es sich bitte von der Seite java.sun.com herunter und installieren Sie es zuvor (kann einen Neustart erfordern). Zum Wechseln der Laufzeitumgebung wählen Sie bitte Window->Preferences->Java->Installed JREs aus dem Eclipse Hauptmenü aus. Nun fügen Sie mittels „Add“ das Java JDK aus dem Pfad in das Sie es zuvor installiert haben aus, und wählen es als Standardlaufzeitumgebung aus. Bitte starten sie nun Eclipse neu.
- Die ATOSj Quellen liegen auf dem lehrstuhleigenen CVS. Insbesondere lagern dort nicht nur die Quellen, sondern auch die Einstellungen für das ATOSj-Eclipse-Projekt. Um diese aus dem CVS zu laden wählen Sie File->New-Project->CVS->Projects from CVS aus dem Eclipse Hauptmenü. Erstellen Sie dazu im folgenden Dialog folgendes neues Repository:
  Host: ein beliebiger Server des Instituts für Informatik.
  Beispielsweise amsel.informatik.hu-berlin.de
  Repository path: /vol/baal-vol3/projekt98/quellen
  Als Nutzer und Passwort nutzen Sie wie gewohnt Ihren Unix-Login des Instituts.
  Connection type: extssh

Anmerkung: Das nun folgende Auschecken des CVS ist nur nach vorheriger Freischaltung Ihres Accounts für das CVS möglich. Für eine Freischaltung wenden Sie sich bitte an Herrn Sacklowski vom Lehrstuhl für Softwaretechnik (sacklows@informatik.hu-berlin.de)

- Checken Sie nun das Modul „AtosJ“ aus dem CVS aus.

- Wurde Ihr Eclipse mit einem korrekten JDK gestartetet, so erfolgt die Kompilierung zu einer vollständigen Version nun automatisch. Dies geschieht in drei Schritten:

  1. Kompilierung aller .java Quelltexte in .class Java-Bytecode-Dateiem

  2. Erstellung aller benötigten RMI-Stubs für die Kommunikation von ATOSj mit seinen Testobjekten. Die Stubs werden automatisch mit dem Programm rmic kompiliert. Dieses Programm ist nicht in der einfachen Java JRE enthalten. Hierin liegt der Ursache für den nötigen Wechsel der Laufzeitumgebung auf das vollständige Java JDK.

  3. Abschliessend werden automatisch alle erstellen Kompilate in die Datei atosj.jar gepackt. ATOSj kann nicht mit Quellen in einer Verzeichnisstruktur gestartet werden. Nur so können die einzelnen Bestandteile der Klassenpfads von ATOSj vollständig analysiert werden. Dies ist für die spätere Kommunikation mit den Testobjekten zwingend nötig. Wird dies nicht eingehalten, so wird die spätere Kommunikation mit den Testobjekten fehlschlagen!

- Für den Start von ATOSj wählen Sie nun Run->Run->Java Application->New aus dem Eclipse Hauptmenü. Wählen Sie das ATOSj-Projekt und als Main-Klasse atosj.app.AtosJApp aus. Unter Classpath ENTFERNEN SIE NUN DAS ATOSj-PROJEKT VON DEN USER-ENTRIES. Stattdessen fügen Sie mittels „Add jars“ sämtliche Jars des Projekts hinzu. Inklusive der Datei atosj.jar sind dies 8 Einträge.

- Geben Sie der neuen Lauch-Konfiguration nun noch einen passenden Namen und wählen Sie „Run“. ATOSj wird nun starten.

F Glossar

ATOS Vorgängersystem für den oberflächenbasierten Regressionstest von Windows-Programmen

API Engl. Application Program Interface - Sammlung aller Funkionen eines Dienstes oder einer Programmiersprache

Component Ein Element der grafischen Oberfläche, wie z.B. ein Knopf. Die Bezeichnung wurde der kürzeren Schreibweise wegen, in Anlehnung an die Klasse `java.awt.Component`, gewählt.

Custom-Component Ein grafisches Element, das nicht zum Standard gehört.

CVS Engl. Concurrent Version System - System zur Versionsverwaltung für die Arbeit im Team

Exception Engl. Ausnahme. Ein Fehler der den Programmablauf unterbricht und den gesamten Call-Stack durchläuft, bis zu einer Stelle, an der er behandelt wird (try-catch-Block in Java).

GUI Engl. Graphical User Interface - Die grafische Oberfläche eines Programms, die zur Nutzereingabe dient.

Interrupt Ein Ereignis, dass die Arbeit des Prozessors unterbricht und zur Ausführung einer speziellen Routine führt. Ein solches Ereignis ist z.B. ein Tastendruck.

Marathon Open-Source-System für den Test Swing-basierter Java-Programme

Steuertaste Taste, die allein gedrückt kein Zeichen erzeugt und zumeist nur in Kombination mit anderen Tasten verwendet wird.

Testfall Vorgehen für den Test einer Programmfunktion

Testobjekt Ein Programm, das einem Test unterzogen werden soll.

UML Grafische Sprache zur Modellierung von Software-Systemen

Use-Case Grundlegende Teilfunktion eines Softwaresystems mit Wert für den Nutzer

Wrapperklasse Eine Klasse, die ein Objekt einer anderen Klasse beinhaltet und dessen Funktionalität über eine abgewandelte Schnittstelle teilweise oder auch vollständig zur Verfügung stellt.

XCTL X-Control (X von X-ray) - Bezeichnung für das Steuerprogramm zur Halbleiter-Strukturanalyse des Lehrstuhls für Physik an der HU Berlin.

Literatur

[1] Krüger, G.; Otto, A.: *Handbuch der Java-Programmierung*,
4. Auflage, Addison-Wesley, 2006
`http://www.javabuch.de`

[2] Link, J.: *Unit Tests mit Java*,
1. Auflage, dpunkt.verlag, 2002

[3] Spillner, A.; Linz, T.: *Basiswissen Softwaretest*,
2. Auflage, dpunkt.verlag, 2004

[4] Balzert, H.: *Systemisches Testen mit Tensor*,
1. Auflage, BI-Wissenschaftsverlag, 1993

[5] Binstock, C. et al.: *The XML Schema Complete Reference*,
1. Auflage, Addison-Wesley, 2003

[6] Lindholm, T.; Yellin, F.: *Die Spezifikation der virtuellen Maschine*,
1. Auflage, Addison-Wesley, 1997

[7] Hanisch, J.; Letzel, J.: *Automatisierung von Regressionstests eines Programms zur Halbleiter-Strukturanalyse*,
HU Berlin - Institut für Informatik, 2002

[8] Hirth, A.: *Automatische Generierung von Testskriptkommandos durch Capturing von Nutzereingaben in GUI-Programmen*,
HU Berlin - Institut für Informatik, 2005

[9] Balzert, H.: *Lehrbuch der Software-Technik*,
Spektrum Verlag, 1998

[10] Balzert, H.: *Pflichtenheft des Programms Seminarorganisation Version 3.0*, 2000

[11] Klösch, R.; Gall, H: *Objektorientiertes Reverse Engineering*,
Springer-Verlag, 1995

[12] Kiczales, G. et al.: *Aspect-Oriented Programming* in: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*,
Springer-Verlag, 1997

[13] Chiba, S.: *Load-time Structural Reflection in Java* in: *ECOOP 2000 - Object-*

Oriented Programming,
Springer-Verlag, 2000, S. 313-336

[14] Jacobson, I.; Ng, P.-W.: *Aspect-Oriented Software Development with Use Cases*, 1. Auflage, Addison-Wesley, 2005

[15] Bothe, K.: *Folien zur Vorlesung „Software-Engineering“*, 2003

[16] Wandke, H.: *Folien zur Vorlesung „Einführung in die Software-Ergonomie“*, 2006

[17] Lowell, C.; Stell-Smith, J.: *Successful Automation of GUI Driven Acceptance Testing*, 2004,
`http://marathonman.sourceforge.net`

[18] *Marathon User Guide*, 2004,
`http://marathonman.sourceforge.net`

[19] *Jython Homepage*, 2006,
`http://www.jython.org/Project/index.html`,

[20] *Java Accessibility Utilities Homepage*, 2006,
`http://java.sun.com/products/jfc/accessibility/`

[21] *Javassist Homepage*, 2006
`http://www.jboss.org/products/javassist`

[22] *JBossAOP - Aspect-Oriented Annotations*, 2006,
`http://labs.jboss.com/portal/jbossaop/docs/1.5.0.GA/docs/aspect-framework/userguide/en/html/annotations.html`

[23] *Java ist die populärste Programmiersprache der Welt*, 2001,
`http://de.sun.com/company/press-releases/2001/PM01_140.html`

[24] *GULP Trend Analyser*, 2006
`http://www.gulp.de/kb/tools/trend.htm`

Selbständigkeitserklärung
Wir erklären hiermit, dass wir die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt haben.

Berlin, den 9. Januar 2007 Volker Janetschek Nicos Tegos

Einverständniserklärung
Wir erklären hiermit unser Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 9. Januar 2007 Volker Janetschek Nicos Tegos