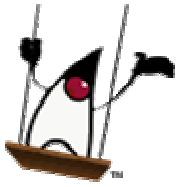
Studienarbeit

Testtools für Java/Swing



-Benutzungsoberflächen

Florian Krämer
Matrikelnummer: 724603
florian.kraemer@stud.fht-esslingen.de

Betreuerin
Prof. Dipl.-Inform. Astrid Beck

Fachhochschule Esslingen – Hochschule für Technik
Fachbereich Informationstechnik
Studiengang Softwaretechnik und Medieninformatik
Februar 2005

Inhaltsverzeichnis

1 Aufgabenbeschreibung

Titel Testtools für Java/Swing-Benutzungsoberflächen

Aufgabenstellung

In dieser Arbeit wird der Stand der Technik zum Thema Testtools für Java/Swing-Benutzungsoberflächen untersucht.

In einem ersten Teil sind die wichtigsten Begriffe zu erklären (z.B. Java/Swing, Test Tools, GUI Test) und es ist in das Thema einzuführen.

Der Hauptteil stellt die Marktuntersuchung dar, eine Zusammenstellung der momentan verfügbaren SW-Werkzeuge mit beschreibenden Attributen. Anhand dieser Untersuchung sollen zwei relevante Tools ausgewählt und anhand einer Testapplikation überprüft oder verglichen werden. Die Ergebnisse sind zu dokumentieren und zu kommentieren.

Im letzten Teil sollen Empfehlungen für die Anwendung in der Lehre gegeben werden. Wünschenswert sind außerdem Hinweise für die Durchführung automatisierter Usability-Tests.

2 Einführung

Softwaresysteme sind ein großer Bestandteil in vielen Bereichen unseres täglichen Lebens und aus unserem Alltag nicht mehr wegzudenken. Sie werden aufgrund größerer Anforderungen zunehmend größer und komplexer. Die Ziele des Herstellers bei der Entwicklung eines Systems sind die schnelle Erstellung, die Kostenreduktion sowie die Qualitätssteigerung. Zur Kostenreduzierung und Beschleunigung der Softwareentwicklung werden vorwiegend Softwaremodule wiederverwendet. Somit wird bereits bewährte Software nicht mehr neu programmiert und kann mit dieser Baukastenmethode leichter aktualisiert werden. Die Produktentwicklung entsteht seit der verstärkten Globalisierung nicht mehr am Reißbrett in einem Hinterhof, sondern zum Teil weltweit in verschiedenen Zeitzonen unter völlig verschiedenen Voraussetzungen. Dafür sprechen einerseits wirtschaftliche Interessen, andererseits möchte der Hersteller direkten Kontakt zu seinen Kunden haben. Eine direkte Folge des daraus resultierenden Anwachsens von Entwicklungsproblemen sind Fehler in diesen Softwaresystemen. Software werden in relativ unkritischen Anwendungen im privaten bis hin zu lebenskritischen Steuerungssystemen eingesetzt. Entsprechend unterschiedlich können die Folgen von Softwarefehler ausfallen. Hierzu einige Beispiele:

Fehlstart Ariane 5

„Es hätte eine beeindruckende Antwort auf die Herausforderung durch neue Konkurrenten auf dem Markt für Satellitenstarts werden sollen. Der Jungfernflug der Ariane 5 im Juni 1996 endete in einem spektakulären Feuerwerk, als die Rakete aufgrund eines Softwarefehlers unkontrollierbar Richtung Erde trudelte und nur kurz nach dem Start mitsamt ihrer Nutzlast - den vier europäischen Cluster-Forschungssatelliten - gesprengt werden musste. Die Sprengung verursachte einen wirtschaftlichen Schaden von ca. 800 Millionen Euro.
Es stellte sich heraus, dass die in Teilen von der Ariane 4 übernommene Software nicht den nötigen Anforderungen entsprach. Die Flugbahn der Rakete wird durch das „Inertial Reference System (SRI)“ gemessen. Ein Teilsystem von „Inertial Reference System (SRI)“ , dass die Flugbahn bei der Ariane-4 gemessen hat, rechnete nach dem Start weiter, obwohl seine Ergebnisse in Ariane-5 nicht mehr benötigt wurden. Andere Flugbahndaten von Ariane-5 (als bei Ariane-4) erzeugten Überlauf bei Real-Integer-Konvertierung und verursachten Fehlfunktion des SRI-Systems. Dadurch wurden wichtige Flugdaten durch ein Testmuster überschrieben. Das SRI-System und sein Backup schalteten sich aufgrund des Fehlers ab.“ [Stein02]

Unglückliche Geldvermehrung

„Kunden der Dresdner Bank, die in den vergangenen Tagen eine Online-Überweisung durchgeführt haben, sollten den Betrag noch einmal kontrollieren. Das Web-Interface wies im Betragsfeld des Überweisungsformulars einen Fehler auf: Das Komma funktionierte nicht mehr als Trenner und Übergang zu den Centbeträgen. Das heißt, die Software ignorierte die Komma-Eingabe, sodass aus einem Betrag 234,50 ohne eigenes Zutun 23450,00 wurde. Wer dann bei der Bestätigung nicht genau den Betrag überprüfte, wird nun möglicherweise Mühe haben, das zu viel überwiesene Geld wieder zurückzuholen. Seit heute ist der Fehler offenbar behoben.“[ct04]

Eine Qualitätssicherung bei der Softwareentwicklung ist zur Vermeidung solcher folgenschweren Fehler unstrittig eine wertsteigernde Eigenschaft. Es werden zur Zeit neue Strategien entwickelt und umgesetzt. Das Testen hat sich dabei zur Verbesserung der Qualitätssicherung als verlässliche Vorgehensweise bewährt. Im gewünschten Normalfall wird die entwickelte Software getestet und aufgetretene Fehler ausgebessert. Die Auslieferung des Produkts sollte erst bei hundertprozentiger Zuverlässigkeit gewährleistet sein. Denn viele Anwender kennen den Unmut, wenn ihr neugekauftes Programm nicht richtig funktioniert. Andererseits sind die Entwickler genauso bei der Qualitätssicherung unter Druck. Bei größeren und komplexen Projekten können Testverfahren eine Menge an Ressourcen verschlingen. Wird irgendwann der Aufwand größer als die Implementierung der Software, lohnt es sich spätestens dann die Tests zu automatisieren und somit die stupide manuelle Arbeit einzusparen. Die Testautomatisierung kann ein Ausweg sein, ist aber wiederum von der Einarbeitungszeit und den weiter entstehenden Kosten abhängig. In den letzten Jahren konnte durch die Einführung von Unittests jeder Entwickler die grundlegenden Funktionen überprüfen und Tests automatisch ausführen. Weitere Eigenschaften sind die bessere Protokollierbarkeit und die einfachere Kontrolle über die Tests.

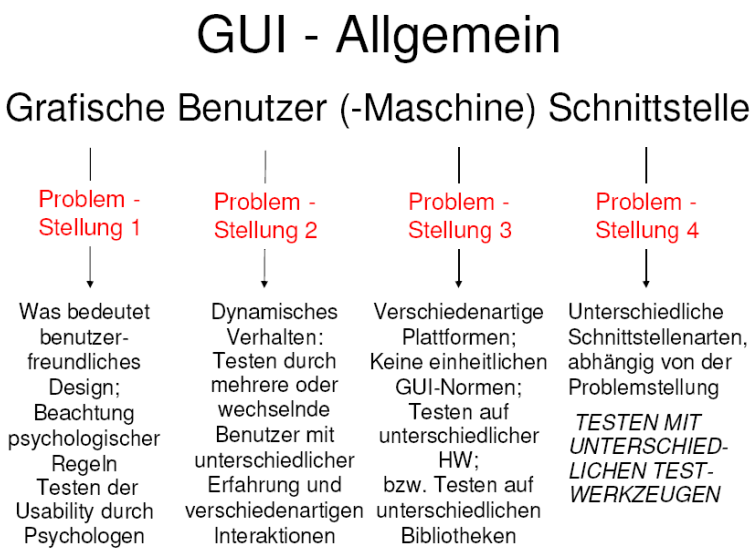


Abbildung 1- Problemstellungen von GUI-Schnittstellen

Die Anforderungen an grafische Oberflächen werden in der Zukunft immer größer. Der Benutzer verlangt seit der Einführung der Apple Fenstertechnik nach grafischen Oberflächen und Multimedia. Sie sind schon ein großer Bestandteil von fast allen Software-Produkten. Darüber hinaus muss die Software Spaß machen, gebrauchstauglich und lernfördernd sein. Dabei werden immer höhere Ansprüche an die Implementierung gestellt. Der Erfolg eines Produkts hängt dabei sehr von der Qualität der GUIs ab. Bei der Überprüfung von Software haben inzwischen GUI-Tests eine größere Gewichtung und eine grundlegende Bedeutung bekommen. Die Tests werden seit geraumer Zeit aus der Sicht des Anwenders durchgeführt. Der Entwickler bekommt dadurch einen besseren Zugang zum Gesamtsystem und kann die einzelnen Funktionalitäten umfassender testen. Es reicht nicht mehr aus ausgefeilte Algorithmen mit Unittests zu überprüfen, wenn der Anwender über eine komplexe grafische Oberfläche auf diese zugreift. Ein Beispiel wäre ein einfaches Datenbank-Interface, das hauptsächlich nur aus einer

Oberfläche besteht und ohne spezielle Algorithmen. Es gibt Szenarien, die nur im Dialog zu überprüfen sind. Das oben erwähnte Beispiel mit der „Komma-Eingabe“ zeigt, das GUI-Tests bei der Qualitätssicherung einen wichtigen Bestanteil sind. Denn die Banken und Firmen möchten ihren Kunden weiterhin einen besseren Service mit entsprechenden Sicherheitsaspekt anbieten können. Weitere Fehlerquellen wären zum Beispiel :

- Soll- Zustand unstimmig mit Ist-Zustand bei Aussehen und Verhalten des Frameworks (z.B. fehlende Dialoge)
- Funktionsaufrufe und / oder Rückgabewerte falsch
- Fehlende Fenster-Aktualisierung nach Datenmodifikation
- Keine Pixelgenaue Darstellung (Problemstellung bei SW-Entwicklung in der Medizin)

Die Studienarbeit wird sich nur auf Testtools für Java Swing Anwendungen mit grafischen Benutzeroberflächen konzentrieren. Bevor jedoch auf eine Auswahl von Testtools für Java/Swing-Benutzeroberflächen selbst eingegangen wird, führt das Kapitel 2 in die grundlegenden Begriffe über das Testen von Software ein. Es umreißt die Begriffe der Softwarequalität, geht insbesondere auf das Testen bzw. auf die aktuellen Ansätze der Testautomatisierung durch Testwerkzeuge ein und befasst sich mit der GUI-Programmiersprache Java Swing. Kapitel 3 greift die Thematik der GUI-Testtools auf. Welchen Stellenwert haben diese Tools in der Industrie und in welchem Rahmen können sie die Arbeit des Entwicklers und Testers erleichtern. Insbesondere werden dann die einzelnen Schritte des Testablaufs und die verwendete Technik näher erläutert. In Kapitel 4 werden kommerzielle und frei verfügbare Testwerkzeuge für GUI-Anwendungen vorgestellt. Im folgenden Kapitel 5 werden zwei evaluierte Testtools mit ihren beschriebenen Attributen miteinander verglichen. Die Wahl dieser Tools begründet sich auf deren Verfügbarkeit und Benutzung in der Industrie und an den Hochschulen. Die Ergebnisse der Untersuchung der ausgewählten Tools wird an einer Beispielanwendung dokumentiert. Anhand einer kleinen Testapplikation wurden die Tools aufgrund ihrer Bedienung und Testauswertung überprüft und verglichen. Im letzten Kapitel wird eine Empfehlung aus dem resultierenden Ergebnis für die Anwendung der Testtools in Studienprojekten gegeben und auf die Durchführung von Usability-Test eingegangen.

Über das Testen von Software wurden schon zahlreiche Bücher und Berichte veröffentlicht. Deswegen werden im Folgenden nur Punkte angesprochen, die für das Verständnis dieser Arbeit notwendig sind.

Wie ist Qualität definiert?

> *„Software-Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Software-Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung gegebener Erfordernisse bezieht" [ISO9126].*

Das bedeutet, das die Qualität eines Produktes somit die Summe aller seiner Eigenschaften bezogen auf die gestellten Anforderungen ist. Die Eigenschaften für die Qualität von Software wurde schon im Jahre 1977 im Buch „*Factors in software quality*" von J. McCall, P.K. Richards und G. Walters beschrieben und ist heute in der ISO-Norm für Software-Engineering wieder zu finden. Die Norm beschreibt ein Qualitätsmodell, das die Softwareeigenschaften in sechs Kategorien einordnet [ISO9126]:

- **Funktionalität** ist die Übereinstimmung der Software mit der Spezifikation. Sie ist eine der wichtigsten Eigenschaften von Software, denn sie bestimmt, inwiefern die Software ihren eigentlichen Zweck erfüllt.
- **Zuverlässigkeit** ist ein Maß, das sich aus der fehlerfreien Funktion einer Software über einen Zeitraum ergibt. Diese Eigenschaft beinhaltet deshalb auch die Fehlertoleranz sowie die Fähigkeit einer Software aufgetretene Fehler zu behandeln und weiter zu funktionieren.
- **Benutzerfreundlichkeit** beinhaltet Unterkategorien wie die Erlernbarkeit, Verständlichkeit und Benutzererwartungskonformität. Sie ist ein Maßstab für den Aufwand, den ein Benutzer der Software für das Verstehen und die Verwendung der Software aufbringen muss.
- **Effizienz** misst die Leistung der Software durch deren Zeitbedarf bei der Verarbeitung oder deren Ressourcenverbrauch.
- **Wartbarkeit** ist ein Maß, das durch den Aufwand bestimmt ist, der betrieben werden muss, um die Software zu verbessern, Fehler aufzufinden und zu korrigieren. Diese Kategorie umfasst Unterpunkte wie die Testbarkeit einer Software, deren Analysierbarkeit und deren Veränderbarkeit.
- **Portierbarkeit** ist die Fähigkeit einer Software, in unterschiedlichsten Anwendungsumgebungen zu funktionieren.

Alle diese Kriterien sind durch subjektive Einflüsse teilweise nicht messbar. Zum Beispiel kann die Benutzerfreundlichkeit von verschiedenen Nutzergruppen unterschiedlich beurteilt werden. Zumindest gibt es dadurch die Möglichkeit diese Eigenschaften durch Test eingeschränkt zu bestimmen.

Die Motivation und das Ziel jeder industriellen Software-Entwicklung muss es sein, ein möglichst fehlerfreies Software-Produkt zu erstellen und dadurch die Qualität zu verbessern. Es gibt verschiedene Auffassungen von Qualität. Jeder Ansatz spiegelt sich in verschiedenen Sichten auf das Produkt wider [Muen98]:

- der transzendente Ansatz,
- der produktbezogene Ansatz (Entwicklung),
- der benutzerbezogene Ansatz (Marketing/Vertrieb),
- der prozessbezogene Ansatz (Fertigung),
- der Kosten/Nutzen-bezogene Ansatz (Finanzen).

Die Ziele der Softwareentwicklung sind hauptsächlich durch wirtschaftliche Interessen definiert. Diese sind die schnelle Erstellung neuer Produkte, die Kostenreduktion bei der Entwicklung sowie die Qualitätssteigerung der Produkte. Die Beschleunigung der Softwareentwicklung wird durch die Wiederverwendung von Software erreicht. Durch diese immer größere Steigerung der Wiederverwendung von Funktionen, Klassen, Komponenten und Architektur wird die Softwareentwicklung vereinfacht, aber auch fehleranfällig. Dem Ansehen eines Unternehmens wird durch eine fehlerhafte Softwareauslieferung schnell geschadet und kann zu einem mehr oder weniger großen wirtschaftlichen Schaden führen. Zum Beispiel verursachte der Fehlstart der Ariane 5 Gesamtkosten des Projekts (1987 bis 1998) von über 6,7 Milliarden Euro. Die größten Anteile der Verluste waren die vier verlorenen Satelliten (450 Mio. Euro), der Verzug im Entwicklungsprogramm (>500 Mio. Euro) und die zwei zusätzlichen Erprobungsstarts. Aus diesem Beispiel können wir für Software Engineering unter anderem daraus lernen, dass

- bestehende Software nicht unbesehen für eine neue Aufgabe wieder verwendet werden darf. Vorher muss geprüft werden, ob ihre Fähigkeiten den Anforderungen der neuen Aufgabe entsprechen (Spezifikation).
- die Fähigkeiten einer Software sowie alle Annahmen, die sie über ihre Umgebung macht, dokumentiert sein müssen. Andernfalls ist die Prüfung auf Wiederverwendbarkeit extrem aufwendig (Dokumentation).
- wenn zwei Software-Komponenten miteinander kooperieren, so eindeutige Zusammenarbeitsregeln definiert, dokumentiert und eingehalten werden müssen. Wer liefert wem was unter welchen Bedingungen (Design by Contract).
- jede potentielle Fehlersituation in einer Software entweder behandelt werden muss oder die Gründe für die Nichtbehandlung so dokumentiert werden müssen, dass die Gültigkeit der dabei getroffenen Annahmen überprüfbar ist (Fehlerbehandlung).
- jedes Programm – neben einem sorgfältigen Test – durch kompetente Fachleute inspiziert werden muss, weil insbesondere die Erfüllbarkeit und Adäquatheit von Annahmen und Ergebnissen häufig nicht testbar ist (Review).
- Software, die nicht benötigt wird, auch nicht benutzt werden sollte (Effektivität).
- bei der Prüfung von Software, die aus mehreren Komponenten besteht, es nicht genügt, jede Komponente nur isoliert für sich zu prüfen (Test).

Abbildung 2 – „Fehlerkritikalität“ Comic aus DIE ZEIT

Qualität ist unstrittig eine wertsteigernde Eigenschaft einer Software. Diese kann jedoch nur durch Tests, die diese bestätigen sichergestellt werden. Umfangreiche Systemtests unter möglichst realistischen Bedingungen sind notwendig. Die Übersicht bei großen und komplexen Systemen ohne Kommentare können bei hunderttausend Zeilen Programmcode mehrere zehntausend Fehler enthalten [McC93].
Das Finden solcher Fehler durch Festlegen von Tests und Testkriterien und die Ermittlung der Fehlerquellen, um diese zu terminieren, sind Aufgaben des Qualitätsmanagements. Somit kann die Gesamtqualität eines Produktes gesteigert werden. In der Softwareindustrie beschäftigt sich das Qualitätsmanagement nicht nur mit der Kontrolle auf Fehler am Endprodukt, die dann beseitigt werden müssen, sondern auch mit der Untersuchung der Ursache solcher Fehler. Auf diese Weise können Strategien erarbeitet werden, diese in Zukunft zu vermeiden bzw. das Risiko ihres Auftretens zu verringern.

Um eine Bewertung der inneren Qualität und damit der Entwicklung eines Softwaresystems während seiner Evolution vornehmen zu können, werden ausgehend von den Ergebnissen der Anforderungsanalyse und allgemein den Fallstudien geeignete Qualitätsmodelle mit geeigneten Operationalisierungen von Qualität entwickelt. Die theoretische Grundlage bilden Qualitätsmodelle wie Factor-Criteria-Metrics (FCM), in denen Qualität durch Qualitätsmerkmale, ausgedrückt in messbaren Qualitätskriterien, beschrieben wird. Zur Messung der Qualitätskriterien und damit der inneren Qualität werden passende Softwaremetriken identifiziert und weiterentwickelt. Des Weiteren werden Vorgehensmodelle angewandt: Modelle, deren Anwendung zu einem entwicklungs- oder unternehmensspezifischem Qualitätsmodell führt - beispielsweise Goal-Question-Metric [QBench].

Qualität durch Testen

> *„**Testen** ist ein Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.“ [Myers79]*

Das Testen einer Software ist neben Analyse, Entwurf, Implementierung, sowie Einsatz und Wartung, teil jeder Vorgehensweise bei der Entwicklung von Softwaresystemen und wichtiger Bestandteil der Qualitätssicherung. Laut Myers werden 50% der Gesamtzeit und der Gesamtkosten von Softwareprojekten nur für das Testen aufgewendet. Um die Qualität eines Produktes zu sichern, muss die Implementierung geprüft werden. Entsprechen die Eigenschaften

ihrer Spezifikationen? Für den Vergleich zwischen der Spezifikation und den wirklichen Softwareeigenschaften muss die Software ausgeführt werden. Testen ist nichts anderes als das Ausführen eines Programms bzw. einzelner Programmteile, unter Verwendung verschiedener Kombinationen von Zuständen und Eingaben, wobei erwartete und reale Ausgabe verglichen werden. Die Aufgabe des Testens ist es die Fehler in einem Programm zu entdecken und nicht die Fehlerfreiheit eines Programms zu bestätigen.

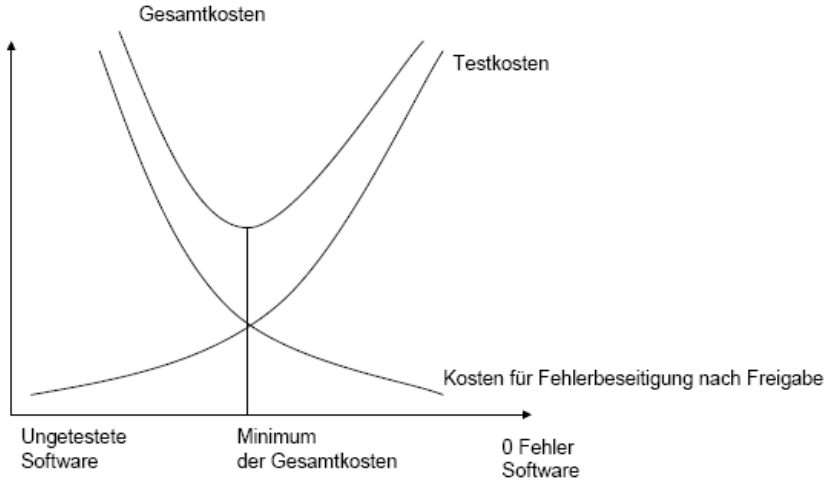


Abbildung 3 - Wirtschaftlichkeit des Testens

Wurde ein Programm sorgfältig getestet und sind alle gefundenen Fehler korrigiert, so steigt zumindest die Wahrscheinlichkeit, dass das Programm sich auch in den nicht getesteten Fällen wunschgemäß verhält. Die Korrektheit eines Programms kann aber durch Testen außer in trivialen Fällen nicht vollständig bewiesen werden. Dafür müssten alle Kombinationen aller möglichen Werte der Eingabedaten getestet werden. Selbst wenn alle Programmanweisungen ausgeführt würden, wäre das noch keine Garantie für fehlerfreie Software. Alle Programmanweisungen in allen möglichen Programmzuständen auszuprobieren ist praktisch nicht möglich, besonders bei interaktiven Programmen. Ein Programm kann auch nach einer bestimmter Anzahl von erfolgreichen Testläufen weitere Fehler enthalten, denn es kann immer noch weitere Konstellationen des Programms mit auftretenden Fehlern geben. Denn abhängig vom Programmzustand kann eine Anweisung unterschiedliche Ergebnisse liefern. So ist z.B. die Anweisung x = 1/a nur dann erfolgreich, wenn a ungleich Null ist.

> *„program testing can be a very effective way to show the presence of bugs,*
> *but it is hopelessly inadequate for showing their absence." [Dij72]*

Mit anderen Worten kann Testen nicht formal die Korrektheit von Software beweisen, noch die Evaluation oder die konstruktive Kritik bei der Qualitätssicherung ersetzen.

Den wichtigsten Faktor für den Erfolg eines Projektes ist der Mensch. Nach einem fehlerhaften Testdurchlauf erweist sich schon die Fehlersuche als psychologisches Problem. Welcher Programmierer sucht schon mit Begeisterung und unter Zeitdruck nach seinen eigenen Fehlern. Es wird auch als unbeliebte und oft störende Tätigkeit empfunden. Die Akzeptanz der Betroffenen muss bei qualitäts- und produktivitätssteigernden Maßnahmen gegeben sein.

Ansonsten kommt es im besten Fall zur Ignoranz der Maßnahme und im schlimmsten Fall zur Demotivation und Unzufriedenheit.

Das neuere Vorgehensverfahren „Test-Driven-Developement“ beim Extreme Programming ist nur eine bedingte Abhilfe zur schnellen Fehlersuche. Die Programmierer müssen die Testfälle auf jeden Fall festlegen, bevor sie zu implementieren beginnen. Tests sammeln sich in einer Ablage und werden automatisiert abgearbeitet. Jedes Mal, wenn ein Programmierer eine neue Funktion in das System einbaut, ist er verpflichtet, alle bisher vorhandenen Tests wieder ablaufen zu lassen, um die Funktionstüchtigkeit des Systems zu bewahren. Er muss explizit zeigen, das seine Modifikation keinen Schaden angerichtet hat. Aber auch konventionelle Softwareentwicklungen müssen weiterhin gründlich geprüft werden, bevor die Kunden das Produkt erhalten. [ct01]

Die Qualität von Software kann aus zwei Sichten betrachtet werden, der des Entwicklers und der des Nutzers. Der produktorientierte Ansatz des Testens, der einem Funktionstest gleichkommt, stellt die Qualität der Software aus der Sicht des Nutzers sicher. In der Software wäre dies ein Test nach den Anwendungsfällen der Software, der auch als Abnahmetest betrachtet werden kann. Der prozessorientierte Ansatz zeichnet sich dagegen durch den Entwicklungsprozess begleitende Kontrollen aus. Die Qualität wird vom Entwickler während seiner Arbeit durch fortlaufende Tests sichergestellt. Dabei testet er die internen Abläufe der Software, deren Korrektheit auch die Fehlerfreiheit der Funktionen der Software bestätigen [Ginz00]. Qualität kann nicht im nachhinein in ein Produkt hineingeprüft werden. Daher ist es unerlässlich, die Erzeugung von Qualität durch konstruktive Testmaßnahmen zu fördern.

Für eine stabile und qualitativ hochwertige Software gelten auch heute noch die Regeln, die das amerikanische Magazin Byte bereits 1995 formulierte [LE05]:

- Kämpfe für ein stabiles Design.
- Teile die Aufgaben sauber auf.
- Vermeide Abkürzungen.
- Baue reichlich Absicherungen ein.
- Nutze die Werkzeuge vernünftig.
- Verlasse Dich auf weniger Entwickler.
- Kämpfe stetig gegen die Zunahme von Funktionen.
- Nutze formale Methoden, wo es Sinn macht.
- Starte mit dem Testen, wenn Du die erste Zeile Code schreibst.

Testaktivitäten

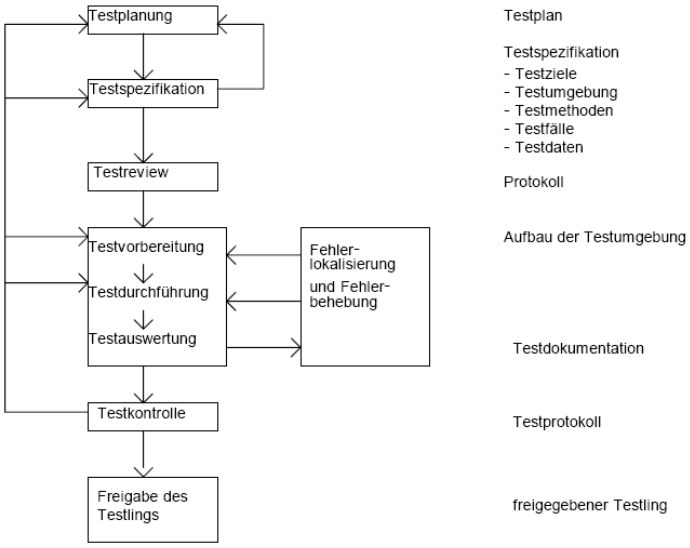


Abbildung 4 - Testaktivitäten

Vor dem Beginn jedes Tests sollte eine sattelfeste *Testplanung* stehen. Bei der Automatisierung von Tests ist dies besonders wichtig, da ansonsten durch den – im Vergleich zum manuellen Testen - stark erhöhten Testdurchsatz im "Sumpf" der Testergebnisse versinken können. Konkret bedeutet dies, dass bereits zu Beginn der Testaktivitäten der Grundstein für eine erfolgreiche und effiziente Auswertung der Testergebnisse gelegt werden muss. Es macht keinen Sinn, Zeit durch die Automatisierung von Tests einzusparen, um diese dann bei der Testauswertung wieder zu verlieren. Aus diesem Grunde ist die gekonnte Verknüpfung von Anforderungsmanagement mit der Automatisierung von Test extrem sinnvoll und erfolgsversprechend. Ein grober Testplan ist ein organisatorischer Rahmen, der den zeitlichen Ablauf beispielsweise auf der Basis der Meilensteine eines Projektes und die administrativen Verfahren für die Test festlegt. Für die Verfeinerung des Testplans ist die *Testspezifikation* notwendig. Sie hält die Testmodelle, Teststrategie, Testmethodik, Testfälle und Testdaten schriftlich fest. In der Testspezifikation wird auch die Reihenfolge der Implementierung von Modulen und Komponenten und der notwendigen Tests festegelegt. Entsprechend werden die einzusetzenden Testwerkzeuge definiert bzw. festgehalten, wann sie beschafft oder erstellt sein müssen. Der Aufwand wird durch die rechtzeitige Planung des Testablaufs und der Testumgebung reduziert. Beim *Testreview* wird der Testplan und die Testspezifikationen überprüft. In der *Testvorbereitung* wird die erforderliche Umgebung für den Test bereitgestellt. Die Testfälle sollen bis auf die Eingabedaten in maschinenlesbarer Form an der Mensch-Maschine-Schnittstelle vorliegen. Bei der *Testdurchführung* werden die Testfälle ausgeführt. Beim Ablauf ist es ratsam, das sie mit möglichst wenigen Bedienereingriffen maschinell durchgeführt werden sollte. Die Testergebnisse werden in der *Testauswertung* gegen die erwarteten Solldaten geprüft. Dabei stellt sich dann die Frage, ob die bearbeiteten Testfälle ausreichen. Zum Schluss dient die Testkontrolle der Überwachung und Steuerung des gesamten Testprozesses auf der Basis der Testplanung.

Testmodelle und Teststrategie

Der Softwareentwickler steht bei einer unendlichen Anzahl der Eingaben kombiniert mit den Zuständen vor einem gewaltigen Problem. Welche Eingaben soll er denn Testen? Zielloses herumexperimentieren führt zu keinem Erfolg und das Vertrauen des Entwicklers schwindet schnell. Es muss dort getestet werden, wo Fehler am wahrscheinlichsten auftreten.
Aufgrund der großen Anzahl der möglichen Tests für eine Software, muss es auf ein Testmodell

beschränkt werden. Ein Testmodell ist eine prüfbare Repräsentation der Zusammenhänge von Teilen des Systems, dass sich wiederum an einem Fehlermodell orientiert. Das Fehlermodell erkennt Teile oder Beziehungen eines Systems in denen mit großer Wahrscheinlichkeit Fehler gefunden werden können. Bei den Teststrategien werden verschiedene Stufen des Testens während den entwicklungsbegeleitenden Testphasen unterschieden. Zunächst werden hierbei die Methoden und Funktionen der Software getestet, anschließend die Komponenten eines Systems, gefolgt dann von deren Interaktionen. Zu guter Letzt folgt ein kompletter Systemtest. Der Hintergrund dieser Vorgehensweise ist das Finden von Fehlern in den frühen Phasen und in weniger unübersichtlichen Softwareentwürfen, in denen das Auffinden der Fehler verhältnismäßig einfach ist. Für die einzelnen Softwaretests gibt es eine klassische Definition der Klassifikation. Als Unterscheidungsmerkmal wird die Größe und Zusammengehörigkeit der jeweiligen Komponenten gewählt [SW02]:

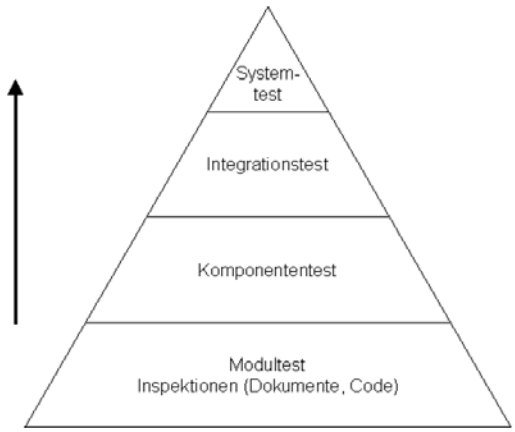


Abb. 5 - Testphasen bei der Softwareentwicklung

- Der Modultest, auch Unittest genannt, dient dem Test einzelner Klassen und deren Methoden. Es wird deren Korrektheit getestet, das heißt, ob sie spezifikationskonform funktionieren. Für diese Art des Testens bietet sich ein Whitebox-Test an, um die Funktionsweise der Methoden zu verifizieren. Weitere Kontrollmethoden, die nicht zum Testen im eigentlichen Sinne gezählt werden, wie das Review, sind ebenfalls für diese Testphase geeignet. (Verifikation der Module)

- Eine Komponente ist eine Klasse oder eine Gruppe von Klassen, die interagieren und eine gemeinsame Aufgabe erfüllen. Bei einem Komponententest wird eine Komponente als Einheit getestet. Häufig kommt zu diesem Zweck ein Blackbox-Test zum Einsatz. (Verifikation der Komponente)

- Der Integrationstest dient der Kontrolle des Zusammenspiels von Komponenten. Es wird also die Kollaboration der Klassen bzw. Komponenten untersucht. Hierbei werden Testfälle gewählt, an denen das Verhalten der Testlinge untereinander beobachtet werden kann. Der Integrationstest ist ein wichtiger Bestandteil von Tests komplexer Anwendung. Kombinierte einzeln bereits getestete Teile können somit als Einheit betrachtet werden. (Validation bzw. Verifikation zwischen den Komponenten)

- Der Systemtest ist eine Prüfung der Software als vollständige Einheit. Das Programm ist vollständig und es werden üblicherweise alle Anwendungsfälle als Blackbox-Tests gegen die Softwarespezifikation durchgeführt. Dies ist auch häufig der Abnahmetest des Systems.(Verifikation bzw. Validation des Systems)

Verifikationstests dienen zum Überprüfen der Funktionalität eines Softwaresystems bezüglich der Spezifikation im Pflichtenheft. Verifikationstests werden sowohl auf einzelne Klassen oder

Module eines Systems als auch auf die Integration dieser Komponenten durchgeführt. Validierungstests dienen zur Überprüfung der Funktionalität des Gesamtsystems auf Brauchbarkeit. Diese Tests werden deshalb oft auch als Systemtests oder Gesamttests bezeichnet. Bei Validierungstests werden wiederum Subsystemtests und Integritätstests unterschieden. Subsystemtests beziehen sich auf einen Teilbereich des Gesamtsystems, der größtenteils unabhängig von den anderen Bereichen benutzt wird. Integritätstests beschäftigen sich mit dem Testen des aus den Subsystemen bestehenden Gesamtsystems.

Für den Tester bedeutet eine gewisse Reihenfolge ein geringerer Arbeitsaufwand, leichteres Fehler finden und lässt leicht Rückschlüsse auf die Fehlersuche zu. Die Strategien sind für die Kombination der Komponenten in dieser Testphase [Ri97]:

- Bottom-Up-Strategie: Die Komponenten werden in der Reihenfolge ihrer Abhängigkeiten untereinander von unten nach oben integriert. Der Vorteil für den Tester bei dieser Vorgehensweise ist, dass er nur Testtreiber schreiben muss.
- Top-Down-Strategie: Analog zur Bottom-Up-Strategie werden die Komponenten von oben nach unten integriert. Der Tester muss lediglich Stubs schreiben, welche die Aufrufe der Komponenten abfangen und benötigte Testdaten zurückgeben.
- Big-Bang-Strategie: Die Big-Bang-Strategie erfolgt durch das Testen aller Komponenten auf einmal als monolithische Einheit. Der Tester muss keinerlei Treiber oder Stubs schreiben, jedoch sind Fehler innerhalb der komplettierten Anwendung schwieriger zu finden.

Testmethoden

In den einzelnen Testphasen finden unterschiedliche Testmethoden Verwendung. Die Auswahl der Methoden ist von den Zielen der jeweiligen Testphase und dem Testling abhängig.

Es gibt zwei unterschiedliche Vorgänge bei Tests. Beim destruktiven Testen werden durch die Testfallauswahl die Problembereiche eines Programms überprüft mit dem erklärtem Ziel, möglichst viele Fehler zu entdecken. Alternativ wird beim demonstrativen Testen Testfälle ausgewählt, die den korrekten Ablauf des Programms zeigen sollen [Ri97]. Bei den Test unterscheidet man Testmethoden, die sich nach der Art der Testdatengenerierung und der Testfälle unterscheiden. Allgemein sind dies spezifikationsorientierte und implementierungsorientierte Tests [Ri97].

Spezifikationsorientierte Tests werden gewöhnlich als Blackbox-Tests bezeichnet. Bei dem Blackbox-Testing wird nur anhand der Spezifikation und der Schnittstelle getestet, unabhängig davon, wie die Implementierung aussieht. Sie betrachten eine zu testende Software als Einheit, die mit Eingaben versorgt wird und deren Ausgaben kontrolliert werden. Der interne Ablauf innerhalb der Software wird dabei nicht beachtet.

Implementierungsorientierte Tests werden hingegen als Whitebox-Tests bezeichnet. Bei Whitebox-Tests wird vorausgesetzt, dass die Implementierung des zu prüfenden Objektes

bekannt ist. Sie basiert also auf der Analyse des Sourcecodes. Dabei wird versucht mögliche Schwachstellen zu erkennen und mit Testfällen abzudecken.

Testmetriken

Um die Qualität der Software und die Wirksamkeit der durchgeführten Tests zu bewerten, werden Metriken angewandt. Sie versuchen über verschiedene Abdeckungsgrade eine Antwort auf die Frage zu finden, ob ein Softwaresystem hinreichend getestet wird. Eine Teststrategie legt fest, welche Testfälle zu erstellen sind. Allgemein besagt der Abdeckungsgrad welcher Prozentsatz der zu erstellenden Testfälle geschrieben wird. Die zu verwendeten Testmetriken hängen von den vorliegenden Informationen und dem Ziel der Beurteilung ab. Zum Beispiel kann die Testdauer als Grundlage zur Erfassung des Aufwands und der Wirtschaftlichkeit des Tests benutzt werden. Dieses Kriterium ist für eine Aussage über die Qualität ungeeignet. Zur Beurteilung der Qualität von Tests werden Kriterien herangezogen, die die Anzahl der Tests, Überdeckung der Eingabe und des Quelltextes als Grundlage haben. Das Ziel jeder dieser Kriterien ist die möglichst vollständige Abdeckung aller aufgestellten Testfälle. Eine Ausführliche Beschreibung der Abdeckungsgrade sei auf [Bin99] und für Überdeckungskriterien und -maße auf [Ze02] verwiesen. Für die Unterstützung des Softwaretesters zu Entwicklung, Durchführung und Kontrolle der Überdeckung des Tests, ist aufgrund der hohen Anzahl der Testfälle eine Unterstützung durch ein Testwerkzeug wertvoll.

Der Testaufwand für Softwareprodukte wird durch die steigende Größe und Komplexität natürlich noch größer. Trotzdem soll sich die gesamte Entwicklungsdauer nicht erhöhen, sondern sogar noch verringern. Deswegen werden in vielen Bereichen eine Automatisierung der Testfällen angestrebt. Die Erwartungen durch den Einsatz von Testwerkzeugen sind bei der Zeitersparnis und der Personalentlastung groß. Dabei muss aber beachtet werden, dass der Einsatz nicht immer gerechtfertig ist.

Testautomatisierung in der Softwareentwicklung

Das die Qualität eines Produktes durch Softwaretests sichergestellt wird, hat sich inzwischen schon herumgesprochen. Genauso ist bekannt, dass die Arbeit wegen der aufzunehmenden Mühe und Zeit sehr unbeliebt ist. Und dieser Aufwand ist abhängig vom Produktumfang. Es bleibt aber auch weiterhin unerreichbar durch ein Testwerkzeug per Knopfdruck die Korrektheit eines Programms zu überprüfen. Das Ergebnis der Testfälle ist weiterhin vom Tester abhängig, der die Tests manuell erstellt und ausführt. Die Arbeit des Testers kann mit Hilfe der Testautomatisierung nur erleichtert werden.

> *„Ein Testtool ist ein automatisiertes Hilfsmittel, das bei einer oder mehreren Testaktivitäten, beispielsweise Planung und Verwaltung, Spezifikation, Aufbau von Ausgangsdateien, Testdurchführung und Beurteilung, Unterstützung leistet.“ [Pol00]*

Der Nachdruck liegt hierbei auf der Unterstützung. Ein Werkzeug ist nur dann ein Hilfsmittel, wenn sein Einsatz zu höherer Produktivität und Effizienz führt. Die für manuelle Tests wichtige

Forderung nach einer möglichst minimalen Testfallmenge lässt sich durch die Testautomatisierung abschwächen. Dies begründet sich dadurch, dass

a) Maschinenzeit deutlich kostengünstiger ist als menschliche Arbeitszeit und

b) Tests ohne erforderliches menschliches Zutun deutlich schneller ablaufen können, als manuelle oder teilmanuelle Tests.

So können mittels automatisch generierter Testsuiten effektiv Kosten eingespart werden. Die Testautomatisierung hat sich besonders bei wiederholenden Aufgaben bewährt, vor allem bei der Durchführung und der anschließenden Auswertung. Dies ist voraussichtlich auch der Grund für die derzeitige Konzentration von Testtools, die auf diese Aufgaben spezialisiert sind. Die Testfallerstellung und Testdatengenerierung wird anderseits von sehr wenigen Tools unterstützt, zudem sind sie sehr zeitaufwendig und besonders abzuarbeiten . Der Grund hierfür ist in der Art der Tests zu suchen. Die Testwerkzeuge zur Testdurchführung sind meist auf Blackbox-Tests spezialisiert, für deren Testdesign das Tool auf eine automatisch verwertbare Spezifikation zurückgreifen müsste. Eine solche Voraussetzung ist aber in den seltensten Fällen gegeben, so dass die Testfallerstellung bei automatisierten Blackbox-Tests Aufgabe des Testers bleibt.
Durch die Testautomatisierung lässt sich darüber hinaus sicherstellen, dass die Tests regelmäßig, vollständig, konsequent und regelmäßig durchgeführt werden. Fehler können somit früh erkannt und behoben werden. Wobei man nicht vergessen darf, dass immer eine hochqualifizierte menschliche Betreuung für die Automatisierung, Auswertung, Pflege und Wartung benötigt werden. Bei ändernden Anforderungen wird die Testfallerstellung zudem sehr zeitaufwendig. Die geforderten Testkriterien zu erfassen und jeden Testfall korrekt auszuarbeiten, muss besonders sorgfältig ausgearbeitet werden. Der Aufwand wird in der Anfangsphase bei Personal zunächst ansteigen. Die Kosten ergeben sich durch die Integration der Testautomation in das Projekt und in die Einarbeitung des Personals, die für die Erstellung der Testfälle aufgrund der hohen Qualitätsmaßstäbe hochqualifiziert sein müssen. Die Wartung der Testfälle darf dabei nicht vergessen. Durch die Automatisierung und deren Auswertung wird verstärkt die menschliche Betreuung benötigt. Der Aufwand der Automatisierung kann anfangs um die 125% bis 150% höher sein als dessen manuelle Durchführung [Spg00]. Die möglicherweise ineffizienter und damit größer werdenden Testsuiten erzeugen anfangs zusätzlich noch höhere Maschinenkosten. Es zeigt sich aber, dass Testautomatisierung einerseits durch die mehrfache Ausführungen der Tests mit vielen der heutigen Ansätze eine Investition in die Zukunft darstellt und die Kosten durch die spätere Reduzierung des Personalaufwands und über den Lebenszyklus der Software sich amortisieren.

Cem Kaner beschreibt zwei Ansätze der Automatisierung von Systemtests, die sich in Projekten als erfolgsversprechend herausgestellt haben [Cem97]:

- data-driven test design
- framework-based test design

Beim „data-driven test design“ werden Tabellen erstellt, die in jeder Zeile eine Kombination von Benutzereingaben und daraus resultierenden, zu erwartenden Ergebnissen zusammenfassen. Jede einzelne Zeile repräsentiert einen Testfall. Um die Tests automatisch ablaufen lassen zu können, benötigt man ein Testtreiber-Programm, das die erstellten Tabellen einlesen und interpretieren kann. Der Testtreiber steuert die zu testende Anwendung entsprechend der Eintragungen in den Tabellen. Der Vorteil dieses Ansatzes besteht darin, dass die Tabellen leicht zu erstellen sind und somit mit geringem Zeitaufwand viele Tests entstehen können. Außerdem sind zum Erstellen der Tabellen keine Programmierkenntnisse erforderlich, jedoch muss im Gegenzug ein relativ aufwendiger Testtreiber programmiert werden.

Beim „framework-based test design“ wird, z.B. mit Hilfe eines GUI Capture/Playback Testtools, für jede Benutzeraktion, die man an der Anwendung tätigen kann, eine Funktion programmiert, die genau diese Aktion durchführt. Um die entwickelten Funktionen anwenden zu können, wird ein Grundzustand der Anwendung definiert. Die Anwendung wird nach dem Start zunächst in diesen Grundzustand versetzt. Das Aufrufen einer Funktion löst nun die ihr zugeordnete Aktion aus, z.B. indem das Testtool die nötigen GUI-Interaktionen an der Anwendung durchführt, und lässt die Anwendung in den Grundzustand zurückkehren. Damit können alle Funktionen in beliebiger Reihenfolge angewendet werden. Alle Testfälle greifen nun auf diese Funktionen zurück, um die vom Testfall vorgegebenen Aktionen in gewünschter Reihenfolge auszulösen. Bei diesem Ansatz besteht der Vorteil darin, dass wenn sich das Auslösen einer Aktion in der Anwendung verändert, lediglich die zugehörige Funktion angepasst werden muss, nicht jedoch die darauf basierenden Tests.

Durch die Umstellung auf automatische Tests darf nicht davon ausgegangen werden, dass zwangsläufig in kürzester Zeit mehr Fehler entdeckt werden. Die Testfälle werden weiterhin noch manuell von Softwaretestern erstellt und ausgeführt. Die Testfälle der Testautomatisierung sind nur so gut und erfolgreich, wie von den Testern entwickelt. Die Automatisierung unterstützt hauptsächlich die Qualitätssicherung. Die Testautomatisierung wird trotz Einschränkungen weiterhin eine bedeutende Rolle in der Softwareentwicklung spielen. Die Entwicklungszeiten werden immer kürzer, so dass wiederverwendbare Software und effiziente Tests unabdingbar sind. Weiterhin lässt sich durch logische Analysen der zu testenden Software auch eine effizientere Pfadabdeckung als durch manuelle Testfallerstellung erreichen, so dass auch – bei gleich bleibender Anzahl der Testfälle – die Anzahl der gefundenen Fehler steigt. Dies wiederum sorgt zu einer höhere Qualität der Software.

Das Quality Assurance Institute (http://www.qaiusa.com) führte einen Versuch durch, bei dem derselbe Test sowohl manuell, als auch automatisiert ausgeführt wurde. Der Test bestand aus 1750 Testfällen und 700 Fehlern. Dabei war eine Rückgang des Aufwandes durch Automatisierung um 75 % zu erreichen [DRP00]. Man sieht also, dass sich das automatisierte Testen bei ausreichender Größe des Tests durchaus bezahlt macht.

Bei der Auswahl von Testwerkzeugen muss man mit Bedacht auswählen, besonders wenn Testwerkzeuge in Betracht gezogen werden, die einen erheblichen finanziellen Beschaffungsaufwand bedeuten oder der effiziente Einsatz dieser Werkzeuge mit großem

Mehraufwand verbunden ist. Der Prozess der Auswahl selbst ist eine sehr zeitaufwendige Arbeit. Kriterien für die Werkzeugauswahl müssen sorgfältig auf die Testanforderungen abgestimmt werden. Jedes Werkzeug muss gegen diese Kriterien getestet werden. Abbildung 6 zeigt einen aus vier Schritten bestehenden Prozess für die erfolgreiche Auswahl von Testwerkzeugen [Post92]:

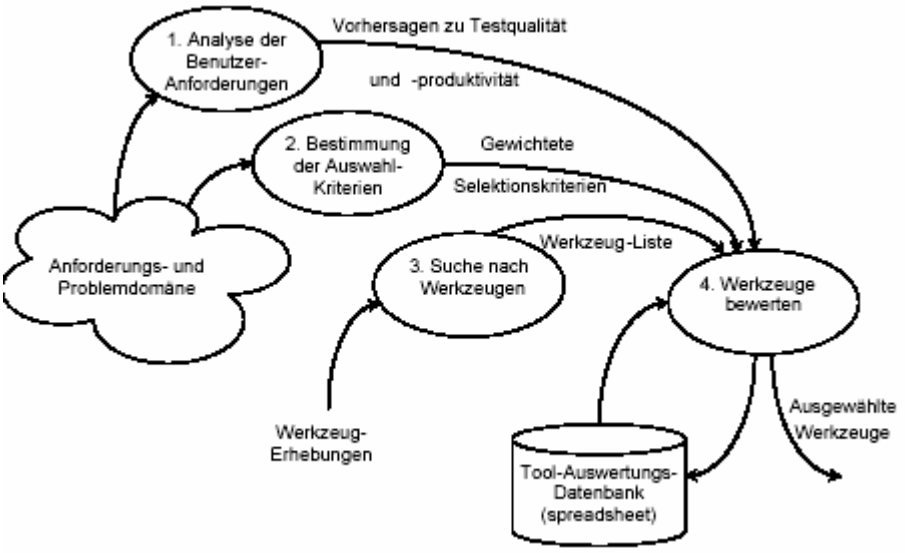


Abbildung 6 - Prozess zur Auswahl von Testwerkzeugen nach Poston

1. Der Auswertende muss die Anforderungen der Benutzer identifizieren und quantifizieren. Das bedeutet, dass er die Art des benötigten Werkzeugs ermitteln muss und dass er Daten zu den Kosten und zur Produktivität schätzen muss, z.B. die Testkosten pro Phase oder die gewünschte Testüberdeckung.

2. Als nächstes ermittelt der Auswertende Kriterien zur Werkzeugauswahl, z.B. Preis oder Anforderungen an die Funktionalität, und gewichtet sie. Keine zwei Kriterien dürfen dasselbe Gewicht bekommen.

3. Der Auswertende sucht nach vorhandenen Werkzeugen mittels Erhebungen. Nur Werkzeuge, welche die gewünschte Plattform und Programmiersprache unterstützen werden berücksichtigt.

4. Als letztes vergleicht der Auswertende die vom Verkäufer gelieferten oder anderweitig ermittelten Informationen über das Werkzeug mit den gewichteten Auswahlkriterien. Er bewertet die Werkzeuge je nach Grad ihrer Erfüllung der Kriterien, und trägt die Daten in eine Datenbank ein. Er kann auch Daten aus früheren Bewertungen verwenden. Das Ergebnis der Auswertung ist eine Empfehlung von einem oder mehreren Werkzeugen für den gegebenen Test.

GUI-Programmierung mit Java Swing

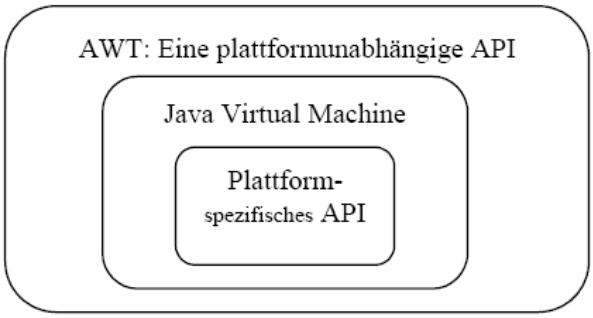


Abbildung 7 - plattformunabhängiges AWT

In seiner ersten Version enthielt Java schon eine sehr einfache Grafikbibliothek, das Abstract Window Toolkit (AWT). Das eigentliche Ziel der AWT-Entwicklung war es, ein vielseitiges, aber einfach zu bedienendes System für die Gestaltung grafischer Oberflächen mit verschiedenen grafischen Primitiven wie Buttons, Menüs etc. zur Verfügung zu stellen, die auf unterschiedlichen Plattformen lauffähig sind. Die Kompromisslösung nach einer Entwicklungszeit von nur sechs Wochen hieß AWT-Peers. Die Entwicklung von AWT war im wahrsten Sinne ein „Schnellschuss“ und nur der kleinste gemeinsame Nenner für GUIs. Es bietet neben den grafischen Primitivoperationen zum Zeichnen von einfachen geometrischen Objekten (Linien, Kreisen, Rechtecken) Fülloperationen und Methoden zur Textausgabe einen Mechanismus zur ereignisbasierten Ablaufsteuerung an, der es erlaubt auf externe Ereignisse wie Mauseingaben zu reagieren. Ferner bietet AWT die weitgehend bekannten GUI-Grundelemente wie Fenster, Dialogboxen, Menüs, ... an. Durch die Peer-Klassen wird die Oberfläche schnell und der Speicherverbrauch hält sich durch die wenigen erforderlichen Zeilen Programmcode im Rahmen. Das Event-Modell war aber unzugänglich und die Auswahl an Oberflächenelementen war sehr gering. Ausgerechnet bei der Portabilität, schließlich die Stärke von Java, bereitet das Toolkit vielen Entwicklern erhebliches Kopfzerbrechen [Stein97].

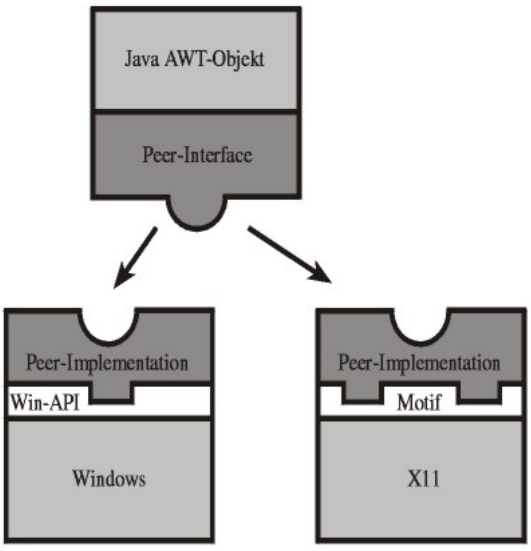


Abb. 8 - Schichtenmodell der AWT-Implementierung

Die schwergewichtigen AWT-Komponenten (engl. *heavyweight components*) benutzen so genannte Peer-Klassen, die als 'native' Funktionen auf der jeweiligen Betriebssystemplattform implementiert sind. Mit Hilfe der Peer-Methoden kann die Darstellung der AWT-Objekte am Bildschirm geändert werden. Die in Java geschriebenen Peers schaffen eine einheitliche Schnittstelle, die von der zugrunde liegende Benutzungsoberfläche unabhängig ist. Wenn nun eine weitere grafische Oberfläche unterstützt werden soll, ist es erforderlich, die Peer-Schnittstellen aller AWT-Komponenten unter Verwendung der betreffenden Plattform-API zu implementieren. Erst wenn alle Peers auf der betreffenden Plattform umgesetzt sind, können Java-Anwendungen, die von AWT-Objekten Gebrauch machen, auf der neu unterstützten Oberfläche laufen. In Abbildung 8 ist das Drei-Schichten-Modell der AWT-Implementierung am Beispiel von Windows und X11 erläutert. Für beide Oberflächen sind die Peer-Schnittstellen implementiert, so dass ein AWT-Objekt auf beiden Plattformen dargestellt

werden kann [Java02]. Die Peer-Objekte stellen die Brücken zwischen der Java API und dem darunter liegenden Betriebssystems dar. Eine AWT-Schaltfläche kennt somit auf der Betriebssystemseite einen Partner, der die Visualisierung vornimmt. Ein einheitliches plattformübergreifendes „Look and Feel“ war aber nicht realisierbar, weil alle Fenster- und Dialogelemente von dem darunter liegenden Betriebssystem zur Verfügung gestellt wurden. Durch die Abhängigkeit der betriebsspezifischen Komponenten gab es bei der Portabilität enorme Probleme. Für die Realisierung aufwendiger grafischer Benutzungsoberflächen musste teilweise viel Aufwand betrieben werden. Aufgrund der unterschiedlichen Komponenten der verschiedenen Betriebssysteme verwendete AWT nur eine Schnittmenge der gängigen Komponenten.

»The AWT was something we put together in six weeks to run on as many platforms as we could, and its goal was really just to work. So we came out with this very simple, lowest-common-denominator thing that actually worked quite well. But we knew at the time we were doing it that it was really limited. After that was out, we started doing the Swing thing, and that involved working with Netscape and IBM and folks from all over the place.« [Gos98]

Swing – „write once run multiple“

Abb. 9 - "Duke" Swing-Logo

Für die JDK-Entwickler war es ein erklärtes Ziel die Nachteile von AWT zu beseitigen. Die Vorstellung der neuen Java-Version 1.2 mit der Entwicklung der Java Foundation Classes (JFC) von Sun, Netscape und IBM erregte somit auf der JavaOne in San Francisco großes Aufsehen. Die JFC bestehen aus fünf Pakete Swing, AWT, Accessibility, Java 2D und Drag and Drop. Swing ist ein komplett neuer Satz von Java-Oberflächenelementen. Durch diese Erweiterung wurde das Abstract Window Toolkit (AWT) vollständig überarbeitet und somit eine echte Alternative, wobei Swing auf AWT basiert. Swing verwendet von AWT das überarbeitete Event-Handling und den Layout-Manager, arbeitet aber nur noch auf der Ebene der Toplevel-Windows in sehr eingeschränkter Weise mit Peers. Durch die komplette Implementierung der Swing-Komponenten in Java ist eine bessere Kompatibilität zwischen den Programmen über unterschiedlichen Plattformen hinweg gewährleistet und erweitert die Auswahl an visuellen Komponenten gegenüber AWT, wie beispielsweise einige mächtige Primitive zur Darstellung von Bäumen. Alle GUI-Elemente, abgesehen von Top-Level-Fenstern, Dialogen und grafischen Primitivoperationen, werden von Swing selbst erzeugt und gezeichnet. Unter Windows wird ein Swing-Button nicht mehr von einem Windows-UI-Manager dargestellt. Auf allen Betriebssystemen findet der Anwender dasselbe Aussehen und diese Bedienung vor. Ein weiteres Merkmal von Swing ist die konzeptionelle Trennung von Funktionalität und Aussehen einer Komponente und der Austausch von Daten zwischen den GUI-Elementen per Drag&Drop.

Komponenten

Mit der Einführung der *Lightweight Components*, den Java Beans, im JDK1.1 wurde Swing technisch umgesetzt. Durch die angebotenen Swing-Komponenten erfolgt die Realisierung der grafischen Primitivoperationen nicht mehr durch die Operationen des zugrunde liegenden GUI-Systems. Die gesamte Swing-API benötigt keine nativen Methoden und stellt die gesamte Verwaltung und Verarbeitung der Primitiven zur Verfügung. Die Abhängigkeit eines Partners auf der Betriebssystemseite ist somit unterbunden. Swing ist also plattformunabhängig. Durch diesen Ansatz setzt Swing sein eigenes Look-and-Feel um.

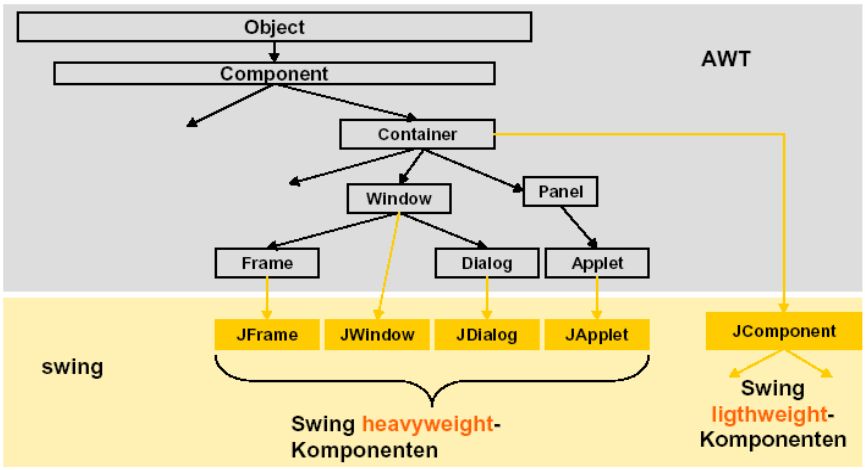


Abbildung 10 - Swing-Komponenten

Die Swing-Klassenbibliothek baut auf AWT auf. Es besteht aus den AWT übernommen schwergewichtigen Komponenten (heavyweight) und den neuen leichtgewichtigen Swing-Komponenten (lightweight). Übernommen wurden die Containerklassen Frame, Window, Dialog und Applet und wurden durch Ableitungen erweitert (siehe Abbildung). Die abgeleiteten Klassen stellen Bildschirmfenster dar. Diese schwergewichtigen Komponenten werden auch als Top-Level-Container bezeichnet. Das Betriebssystem stellt diese Komponenten zur Verfügung und werden von den Swing-Komponenten als Zugang zu der vom Betriebssystem verwalteten Oberfläche verwendet. Die Klasse Component bildet die Basisklasse der Objekte, die als grafische AWT-Komponenten auf den Schirm kommen. Die leichtgewichtigen Swing-Komponenten in der Klasse JComponent sind nicht direkt von Component abgeleitet, sondern erst von Container. Das hat die Konsequenz, dass jedes JComponent automatisch auch ein Container ist. JComponent beinhaltet einen großen Satz von zusätzlichen neuen Komponenten beginnend bei sehr einfachen, beispielsweise Labels, bis zu sehr komplexen Komponenten, wie zum Beispiel Tabellen und Trees. Alle Elemente von Swing sind Bestandteil des Pakets *javax.swing*. Um eine Namensverwirrung zu vermeiden, wurde an die entsprechenden neuen Klassen ein „J“ vorangestellt.

Zwischen den AWT- und Swing-Hauptfenstern besteht ein bedeutender Unterschied in ihrer Komponentenstruktur und den sich daraus ergebenden Unterschieden in der Bedienung. Während jede Komponente eines AWT-Fensters in ein eigenes „undurchsichtiges“ Fenster gelegt wird, besitzt ein Swing-Hauptfenster eine einzige Hauptkomponente (Heavyweight-Behälter; Frame, Window,...), die alle anderen Komponenten aufnimmt und einen durchsichtigen Hintergrund besitzen kann. Das Mischen beider Komponenten ist nicht zu

empfehlen, es kann zu Überlappungen kommen. Außerdem bietet Swing neben einer vollständigen Auswahl von grafischen Elementen eine Verbesserung der Architektur an.

MVC-Architektur

Bei der Entwicklung von Swing gehört die Verbesserung der Architektur des Gesamtsystem zu einer weiteren „Errungenschaft". Swing basiert auf die Model-View-Controller-Architektur(MVC). Es ist ein bekanntes Entwurfmuster aus SmallTalk und ein bewährter Lösungsansatz. Bei diesem Konzept wird der gesamte Code nicht in eine einzelne Klasse gepackt. In der klassischen Architektur besteht jede Komponente aus den drei unterschiedlichen Bestandteile.

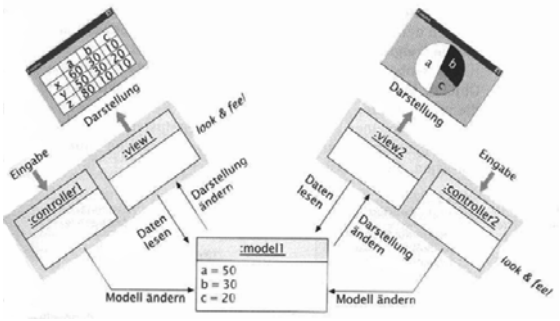


Abb.11 - Eingabe u. Anzeige von Daten in 2 Dialogen

Model

Das Model verwaltet die Daten und Eigenschaften der Komponenten und speichert seinen Zustand. Es enthält die gesamte Verarbeitungslogik (Semantik) der Komponente. Bei einem Schieberegler könnte das der aktuelle Wert, die Minimal- und die Maximal-Position sein. Es bietet zusätzlich dem Controller und der View weitere Methoden, seine Informationen abzufragen oder zu verändern. Somit können verschiedene Views auf ein und demselben Model Zugriff haben.

Controller

Der Controller ist das Verbindungsglied zwischen Model und View. Es empfängt die Benutzer-Aktionen, wertet sie aus und leitet die erforderlichen Maßnahmen zur Änderungen an das Model bzw. an die View weiter. In unserem Beispiel das Verschieben des Reglers.

View

Der View ist verantwortlich für die visuelle Darstellung der Komponente. Es benutzt die Model-Methoden, um die für ihn relevanten Daten abzufragen und nach der Vorgabe des Designers darzustellen.

Damit die Views und Controller bei Datenänderungen informiert werden, lassen sie sich beim Model registrieren. Durch einen Nachrichtenmechanismus bekommen sie die Änderungen mitgeteilt. Nach Erhalt einer Änderungsnachricht werden die Daten beim Model abgefragt und die Darstellung auf den aktuellen Stand gebracht. Diese Vorgehensweise entspricht dem Observer-Pattern. Der Controller lässt sich nur beim Model registrieren, wenn das Verhalten des Controllers vom Zustand des Models abhängt. Zum Beispiel bei Überschreitung der maximalen Anzahl der zu speichernden Werte oder wenn eine Änderung des Models einen Menüeintrag de- oder aktiviert.

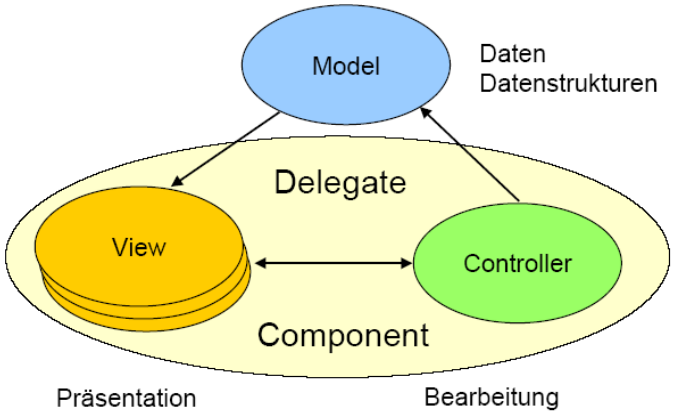


Abbildung 12 - Zusammenlegen von Controller und View

Durch die Trennung der einzelnen Teile ist es möglich das Aussehen einer Komponente zu verändern ohne ihr Verhalten zu modifizieren. Die drei Teilsysteme kommunizieren nur über Protokolle. Bei Einhaltung des Protokolls ist es möglich ein Teilsystems gegen ein anderes auszutauschen. Für neue Benutzer-Ereignisse muss zum Beispiel nur der Controller-Teil geändert werden. Es ergibt sich aus der strikten Trennung von View und Controller auch einige Probleme. Die Kommunikation zwischen den beiden Teilsystemen kann sehr schnell komplex und unübersehbar werden. Deshalb wurde die klassische Architektur modifiziert und Swing verwendet das MVC-Konzept in vereinfachter Form. Der View- und Controller-Teil wurde zu einem Teil „Delegate“ zusammengefasst, weil das Erscheinungsbild und die spezifische Bedieneigenschaft fast immer zusammengehören.

Frei wählbares Look-and-Feel

Die Entwickler gaben dem Swing mit dem „Pluggable Look-and-Feel“ eine herausragende Eigenschaft mit. Applikationen mit einem Windows-Look sehen jetzt unabhängig von der Plattform auf dem Mac- oder Unix-Rechner gleich aus. Dieses Feature ermöglicht einem Anwender auch zur Laufzeit nach seinem Geschmack zwischen vier vordefinierten Look-and-Feels (L&F) zu wählen. Zur Auswahl stehen Metal (java-eigenes L&F), Motif (Unix Oberfläche), Mac (Apple Macintosh) und Windows (MS Windows). Eine Applikation muss bei Änderung des Look-and-Feel nicht mehr herunterfahren, sondern kann ohne großen zusätzlichen Aufwand praktisch über einen speziellen Menüpunkt mit einem einzigen Methodenaufruf von einem UI-Manager vollständig und einfach ersetzt werden. Die Klasse UIManager steuert das Aussehen und kann ermitteln, welche zur Verfügung stehen und aktuell verwendet werden.

Abbildung 13 - Verschiedene Look-and-Feels

Swing enthält für jedes Look-and-Feel einen Satz von Strategieobjekten (WidgetUI, ListBoxUI, ButtonUI,...), die dafür zuständig sind, den jeweiligen Oberflächenelementen ein entsprechendes Aussehen zu verleihen. Für jedes GUI-Element, das ein solches L&F erhalten soll, muss ein entsprechendes WidgetUI-Objekt vorhanden sein. Damit nicht immer ein L&F sofort vollständig implementiert werden muss, kann auf entsprechende Defaults zurückgegriffen werden[Stein97]. Ist der Anwender mit keinem der Auswahlmöglichkeit zufrieden, ist es möglich eigene Look-and-Feels zu entwickeln. Diese Arbeit ist schwierig und sehr anspruchsvoll und sollte lieber von Experten übernommen werden. Für Erweiterung und Modifizierung steht für jede Komponente ein Default Look-and-Feel zur Verfügung oder es lässt sich explizit überschreiben. Swing Komponenten gestatten ein benutzerdefiniertes Look-and-Feel ohne von einem konkreten „windowing system“ abhängig zu sein.

Die Verwendung von Swing ist neben den vielen geschilderten Eigenschaften nicht frei von „Nebenwirkungen“ [Krueg03]:

- Swing-Anwendungen sind ressourcenhungrig. Das alle Komponenten selbst gezeichnet werden müssen, erfordert viel CPU-Leistung und eine Menge Hauptspeicher. Die Attribute "speicherfressend" und "langsam", die Java-Programmen oft pauschal zugeschrieben werden, resultieren zu einem nicht unerheblichen Teil aus der Verwendung der Swing-Bibliotheken.

- Zweitens Reifegrad und die Entwicklungsstabilität. Es stecken noch eine Menge kleinerer und größerer Fehler in den Swing-Bibliotheken. Eines der hässlichsten Probleme im JDK 1.2 war die Eigenschaft von Swing-Fensterelementen, belegte Ressourcen nicht vollständig freizugeben und so bei wiederholtem Öffnen und Schließen nach und nach den gesamten Hauptspeicher aufzubrauchen. Seit der Version 1.3 scheint dieses Problem nicht mehr aufzutreten.

- Für Applet-Programmierer ist vor allem unschön, dass es nach wie vor keine Browser mit eingebauter Swing-Unterstützung gibt. Sie müssen daher entweder das Java-Plugin verwenden (und dem Anwender damit einen zusätzlichen Download- und Installationsschritt zumuten) oder sind nach wie vor auf die Verwendung des AWT beschränkt

Mit den neuen JDK-Versionen gehören diese Probleme der Vergangenheit an. Die steigenden Rechnerleistung kompensieren die Hardwareanforderungen. Mit dem JDK 1.3 hat SUN zudem die Startzeit der Anwendungen verkürzt, den Speicherhunger der Swing-Komponenten eingedämmt und ihre Performance weiter verbessert, und schließlich viele der früheren Bugs behoben. Sun behält sich aber das Recht vor, wie bei allen anderen Bestandteilen des javax-Paketes auch, Schnittstellen und Funktionalität ohne Ankündigung zu verändern oder aus dem Angebot herauszunehmen. Sun zeigt bisher auch weiterhin keine großen Ambitionen, das AWT weiter zu entwickeln, obwohl es bei Geräten mit wenig Speicher noch lange nicht abgeschrieben ist. IBM hat reagiert darauf und entwickelte eine Alternative, das SWT (Standard Widget Toolkit). Es erfreut sich mittlerweile größerer Beliebtheit und ist auch Basis der Entwicklungsumgebung Eclipse.

3 GUI-Testtools

Ohne Unterstützung von Tools müssen die aufwendigen GUI-Tests von einer Person mühsam manuell durchgeführt werden. Die vorgegebene Aktionen werden anhand eines Testplans ausgeführt und die Reaktion der Anwendung visuell überprüft. Diese aufwendige Prozedur kann mit Hilfe eines GUI-Testtools automatisiert werden. Bei grafischen Benutzungsoberflächen gibt es eine beinahe unendliche Vielfalt von Befehlskombinationen und Reihenfolgen. Die Erwartungen an die Testautomatisierung von Seiten einiger Manager sind sehr hoch und teilweise nicht umsetzbar. Sie erhoffen sich eine schnelle Kostenreduktion, Qualitätsverbesserung und Verringerung des Personal- und Zeitaufwands. Es ist eine Fehlannahme, das durch effektivere Tests bei der Automatisierung sich die Kosten sofort reduzieren. Eine Erstellung allgemeiner Tests ist sogar kostenintensiver. Bei Testfällen auf GUI-Ebene rechnet man bei der Erstellung mit einem 3 bis 10-facheren Aufwand. Sind spezielle GUI-Elemente enthalten, so kann dieser Faktor auf das 30-fache steigen. Die Testautomatisierung ist bei heutigen Testmethoden abhängig von der Flexibilität, Wartbarkeit und der Lebensdauer des zu testenden Produkts. In der Anfangsphase entstehen zunächst deutlich höhere Kosten. Der Aufwand lohnt sich erst in der Weiterentwicklung des Produkts durch die Wiederverwendbarkeit und die effektivere Durchführung der bereits vorhandenen Testfälle [Cem97].

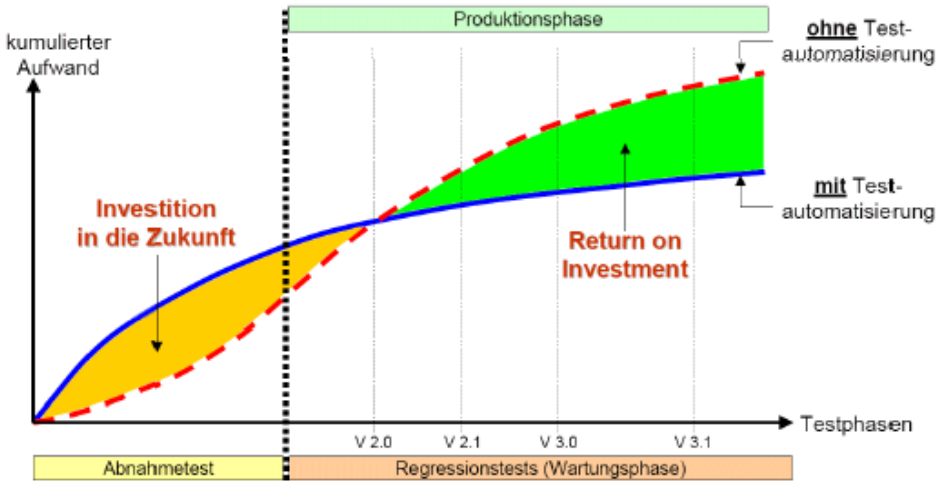


Abbildung 14 - Return of Investment

Stand der Praxis

Das Fraunhofer Institut *IESE* hat 2004 eine Studie über den „Stand der Praxis von Software-Tests und deren Automatisierung“ herausgebracht [IESE04]. Die Studie basiert auf einer Unternehmens-Umfrage und auf gleichzeitig durchgeführten Interviews mit einigen ausgewählten Unternehmen, welche als typische Vertreter kleiner und mittelständiger Unternehmen angesehen werden können.

Von insgesamt 123 befragten Unternehmen führt eine sehr hohe Anzahl Softwaretest durch. Von diesen Unternehmen praktizieren fast zwei Drittel nach eigener Einschätzung systematische Testaktivitäten und weniger als 1% unterstützen keine Testdurchführung. Gründe für nicht

vorhandene Testplanung wurden Zeitmangel und erhebliche Zweifel am Sinn dieser Tätigkeiten genannt. Personalmangel und der damit verbundene Zeitaufwand zählen zu den Hauptgründen für einen Verzicht auf systematische Tests. Wenig überraschend ist weiterhin die Begründung für die Einführung von systematischen Verfahrensweisen bei Testaktivitäten. Diese begründet sich überwiegend durch das Wachstum eines Unternehmens. Auffallend waren die Ergebnisse über den Überblick der verwendeten Testtechniken in den Unternehmen.

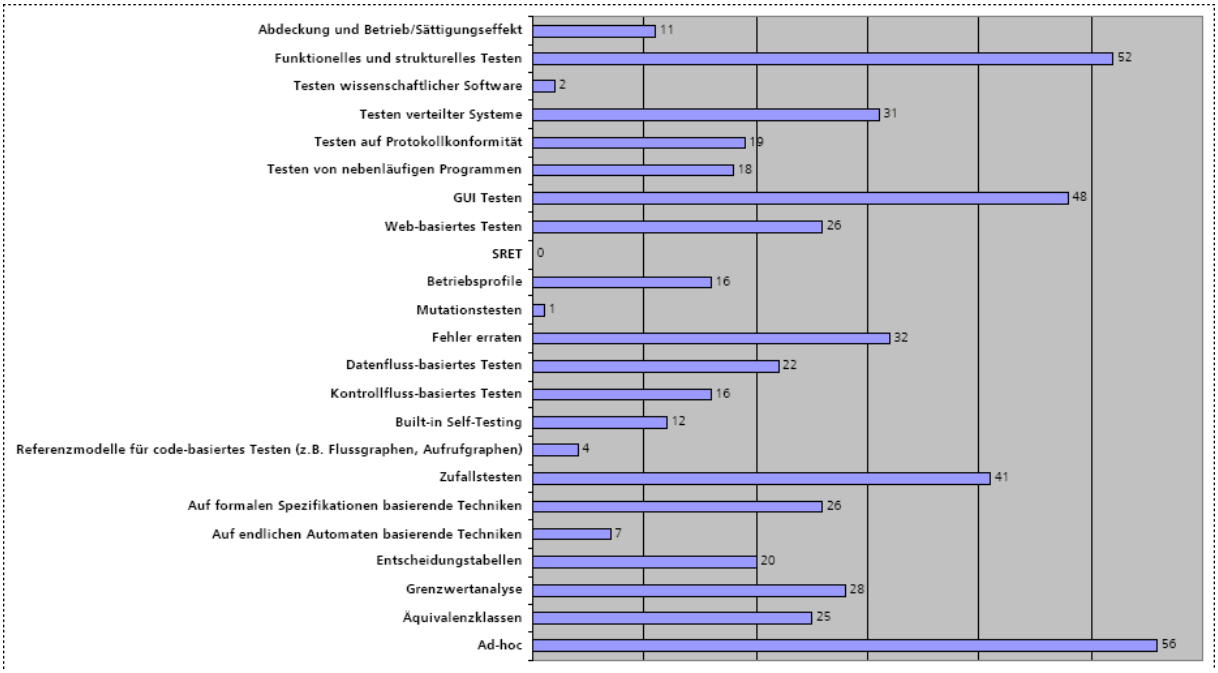


Abbildung 15 - Überblick der verwendeten Testtechniken

Von 94 Unternehmen gehört GUI-Testen zu der drittwichtigsten Testtechnik mit einem geringem Abstand hinter Ad-hoc und funktionales strukturelles Testen. Weiterhin sind Zufallstesten und das Erraten von Fehlern eine häufig eingesetzte „Technik". Inzwischen haben GUI-Tests bei der Überprüfung von Software eine größere Gewichtung und eine grundlegende Bedeutung bekommen. Der Entwickler bekommt aus der Sicht des Anwenders einen besseren Zugang zum Gesamtsystem und kann die einzelnen Funktionalitäten umfassender testen.

Die Verteilung zwischen kommerziellen Werkzeugen und Eigenentwicklungen ist in allen Bereichen recht ähnlich. Etwas mehr als die Hälfte der eingesetzten Werkzeuge sind Eigenentwicklungen. Dies lässt auf ein *Portfolio* verfügbarerer kommerzieller Werkzeuge schließen, die nicht ausreichend für die Einsatzgebiete anpassbar sind. Alternativ besteht auch die Möglichkeit, dass bei kleinen und mittelständischen Unternehmen aufgrund von Ressourcen-Knappheit solche Werkzeuge nicht angeschafft werden. In den Interviews wurde vorwiegend die mangelnde Verfügbarkeit geeigneter Tools als Grund für den Einsatz selbst entwickelter Tools genannt. Weiterhin wurden mangelnde Anpassbarkeit kommerzieller Werkzeuge als Gründe für den Einsatz von Eigenentwicklungen benannt. Kommerzielle Werkzeuge, welche in den jeweiligen Tests von den befragten Unternehmen verwendet werden, sind im Folgenden exemplarisch aufgelistet:

- **Unittests:** CAST, Clover, Compuware Tools (TestPartner), Mercury WinRunner, Purify, QAC, Rational Entwicklertools, SOTOGRAPH, SQS-TEST(/Professional), Testcomplete 3, utplsql/unit, xUnit family (jUnit, CxxTest, jfcUnit, sUnit, httpUnit, dUnit, WebUnit) und Debugmöglichkeiten von Compilern
- **Integrationstests:** Araxis, Cactus, CAN-Tools (CANanalyzer, CANoe), Compuware Tools (TestPartner), Mercury Tools (Test Director, WinRunner), Polyspace, Rational Tools (TestManager, RationalRobot), Segue-Tools (Silk, SilkPerformer), SQS-TEST (/Professional), Testcomplete 3, xUnit family (jUnit)
- **Systemtests:** C&R-Tools, CAN-Tools (CANanalyzer, CANoe), CATT, Compuware Tools, ControlDesk, DE-Tester, dSpace, LabVIEW, Memory-Leakage-Tools, Mercury Tools (Test Director, LoadRunner, WinRunner), Rational Tools (Testmanager, RationalRobot), Segue-Tools (Silk, SilkTest, SilkPerformer), SQS-TEST (/Professional), sSpace, TestPartner, TestStand, xUnit Family (httpUnit, WebUnit), Lasttests-Tools
- **Abnahme-/Akzeptanztests:** CAN-Tools (CANanalyzer, CANoe), CATT, Compuware Tools, DriveImage, LabVIEW, Mercury Tools (Test Director, WinRunner), Rational Tools (Testmanager, RationalRobot), Segue-Tools (Silk-Performer, SilkTest), SQS-TEST (/Professional), TestStand, XDE-Tester

Die Aufwandsverteilung in der Softwareentwicklung liegt bei 79 Unternehmen – im Durchschnitt – für die Bereiche „Erstellung von Anforderungen, „Architektur, Spezifikation“, „Entwicklung von Code“ und „Qualitätssicherung“ bei einem Verhältnis von etwa ¼ zu ½ zu ¼. Bei den hier angegebenen Werten handelt es sich um Mittelwerte. Die Aufwandverteilung weicht stark von den identifizierten Werten ab. Der niedrigste Werte für Erstellung und Dokumentation von Anforderungen, Architekturen, Spezifikation lag bei 3% und der niedrigste Wert für Entwicklung von Code bei 10%. Ebenso war der niedrigste Wert für Erstellung und Durchführung von Tests mit 10% beziffert. Spitzenwerte wurden für Erstellung und Dokumentation von Anforderungen z.B. mit 60% erreicht und mit 85% für Entwicklung von Code. Dabei ist zu beachten, dass sich diese Werte nicht zu 100% aufsummieren, da sie aus der Betrachtung verschiedener Unternehmen hervorgehen.

Hinsichtlich „Software Product Qualitiy“ und „Qualitiy in use“ stellt mit 54 Nennungen bei 70 der befragten Unternehmen „Functionality (Funktionalität der Anwendung)“ den wichtigsten Qualitätstreiber. Die Eigenschaften „Usability“, „Reliability“ und „Interoperability“ stehen mit 20 und mehr Nennungen in zweiter Linie. In den Interviews wurden als Schwierigkeiten, die im Hinblick auf die den angegebenen Qualitätstreiber auftreten, z.B. bei Funktionalität stark divergierende Kundenanforderungen angegeben. Dabei ist das Feedback zu Produkten, die an einem breiten Markt positioniert sind, schlechter als das Feedback zu Produkten, die für einen kleinen Kreis von Kunden entwickelt wurden.

Auf die Frage im welchen Verhältnis die Testaufwände zum Gesamtaufwand des Projektes sind, schätzten 132 Unternehmen die Testaufwände im Kontext eines Projektes auf durchschnittlich ca. 20 % ein. Die Werte liegen insgesamt zwischen 2% und 45% und lassen durch die Standard-

Abweichung von ca. 12% eine Einschätzung im Bereich von einem Drittel bis einem Fünftel des Gesamtprojektaufwandes für das Testen der Software zu.

Die Frage nach der betrieblichen Organisation der im Hause stattfindenden Softwaretests wurde von 105 Unternehmen beantwortet. Dabei konnten jedoch ca. 30% der Unternehmen keine klare Zuordnung zu einer der möglichen Antworten vornehmen. Unter den gegebenen Antworten fällt auf, dass ein hohes Maß an Übereinstimmung bei der Organisation von System- und Abnahme/ Akzeptanz-Tests besteht. In diesen Fällen dominiert die Durchführung von Tests in losen Gruppen und in eigenen Testabteilungen das Testen durch Einzelpersonen. Wesentliche Unterschiede zu den drei bisher angesprochenen Testarten zeigen sich im Bereich der Unittests: Hier dominieren die Einzelpersonen-Tests mit ca. 2/3 gegenüber den losen Gruppen und organisierten Abteilungen, die das verbleibende 1/3 ausmachen.

Das größte der 64 beantwortenden Unternehmen beschäftigt ca. 150.000 Mitarbeiter, wohingegen das kleinste eine „Ein-Mann-Firma“ darstellt. Durchschnittlich werden ca. 4.800 Mitarbeiter beschäftigt. Dieser sehr hohe Wert wird allerdings durch die recht hohe Standard-Abweichung, welche durch ‚extreme Ausreißer’ verursacht wird, nicht gestützt, sondern lässt eine deutlich geringere Mitarbeiterzahl der meisten Unternehmen erkennen, wie der bei 96 Mitarbeitern liegende Median (mittlerer Wert der Stichprobe) belegt. Bei der Anzahl der Mitarbeiter in der Entwicklung und Wartung von Software ist deutlich eine geringere Zahl der Beschäftigten zu erwarten, als mit dem Durchschnittswert angegeben, da auch hier wieder eine sehr hohe Standardabweichung auf wenige extreme Ausreißer im maximalen Bereich der Mitarbeiterzahlen hindeutet. Der Median der Stichprobe liegt bei 25 Mitarbeitern im Bereich Software Entwicklung und stellt einen plausiblen Wert im Verhältnis zum Median von 96 Gesamtbeschäftigten dar.

Unter den 70 Unternehmen, die Auskunft über Investitionen zur Erreichung des aktuellen Grades an Testautomatisierung erteilten, lag die Mehrzahl der investierten Mittel für Testautomatisierung im Bereich unter 50.000 EUR sowohl für Hard- und Software als auch für Investitionen in Personal.

Für die weitere Verbesserung im Bereich Testen/Testautomatisierung wurden von den 70 antwortenden Unternehmen folgende Erfolgs- und Widerstandsfaktoren genannt:

Erfolgsfaktoren

- Mitarbeiter selbst (durch Motivation und Leistungsbereitschaft)
- Günstiges, motiviertes Personal (z.B. Diplomand mit eigenem Interesse am Ergebnis)
- Positive Erfahrungen
- Erarbeitung von Know-how
- Schrittweise Automatisierung mit Offenlegung der Zeitersparnis

Widerstandsfaktoren

- Zeitmangel (z.B. durch Projektdruck/Tagesgeschäft, Einsatzgarantie Kundensysteme, kurze Realisierungszeiträume)
- Geldmangel für Testen (Testen wird oft nicht honoriert, teure Testsoftware, hoher Personalaufwand, Rechtfertigungsprobleme Kosten/Nutzen)
- Aufwand Einführung/Anpassung/Wartung (z.B. Modularisierung von Testfällen)
- Zusatzarbeit (Dokumentation, ...)

Anhand der weiteren Ergebnisse aus der Umfrage stellte das Fraunhofer Institut *IESE* fest, dass ein großes Interesse an den Themen Testen und Testautomatisierung besteht. Erste Ansätze, dieses Thema – und damit die Qualität der eigenen Software – besser in den Griff zu bekommen, werden bereits in vielen Firmen geleistet. Dieser ‚Stand der Praxis' zeigt letztlich, dass im Bereich von Testen und Testautomatisierung ein großes Potential zur Verbesserung der Software, der Methoden und zur Kosteneinsparung für die Unternehmen sichtbar ist und dass dieses erst bei wenigen Unternehmen durchgängig etabliert ist. Ein Großteil der Möglichkeiten, welche derzeit existieren, wird noch nicht ausgeschöpft und bietet daher große Chancen, ökonomischere und stabilere Softwareprodukte herzustellen. Um dies zu erreichen, müssen jedoch – wie die Umfrage zeigte – in vielen Unternehmen verschiedene Widerstandfaktoren überwunden und Voraussetzung noch geschaffen werden. Der komplette Bericht ist im Internet zu finden [IESE04].

Automatisches funktionales Testen

Die rasche, produktive Einsetzbarkeit des Produkts ist in erster Linie von der Erlernbarkeit abhängig, aber auch von der Akzeptanz bei Test-Experten und Einsteigern. Eine ausführliche Dokumentation der verwendeten Skriptsprache trägt maßgeblich zum Erfolg bei. Tutorials demonstrieren den praxisnahen Einsatz der Software und helfen beim flotteren Einstieg.

Zur Testerstellung dienen meist Aufzeichnungs- und Wiedergabefunktionen. GUI-Testtools arbeiten nach dem Capture-and-Replay-Prinzip. Es teilt sich in drei Schritte: Aufnahme, Programmierung und Abspielen. Zuerst zeichnen sie Benutzereingaben auf und generieren daraus den Testcode. Im Allgemeinen arbeiten diese Programme objektorientiert, das heißt, sie zeichnen den Klick auf einen Button und nicht die entsprechenden Bildschirmkoordinaten auf. Im zweiten Schritt lässt sich dieser wieder abspielen und damit das zu testenden Programm virtuell bedienen. Anders ausgedrückt, bilden aufgezeichnete Nutzeraktionen die definierte Testeingabe. Das zu testenden Programm wandelt die Eingabe in Fensteraktionen und in der Regel in gespeicherte Daten um, zum Beispiel Datenbankeinträge oder Dateien.

Bei dieser Vorgehensweise handelt es sich um reines funktionales Testen: Das Programm wird in seiner Gesamtheit, in seiner Interaktion mit all seinen Komponenten getestet – im Gegensatz zu *Unittests*, die in der Regel kleinere Einheiten wie Klassen oder Module kontrollieren. Für das Ergebnis des Tests ist nur entscheidend, ob die Software ihre aufgezeichnete Funktion erfüllt.

Einer der Vorteile dieser Methode liegt darin, dass relativ wenige Nutzeraktionen große Teile des Programms schnell prüfen. Allerdings funktioniert das nur bei sorgfältiger Planung. Eine gründliche Analyse hilft dabei, die benötigte Qualität zu gewährleisten und frühzeitig den notwendigen Testaufwand zu minimieren.

Besonders leicht lassen sich Tests durch schlichtes Ausführen des zu untersuchenden Programms erzeugen. Als Ergebnis dieses Aufzeichnens erzeugen moderne Tools ein objektorientiertes Script, das die relevanten Informationen wie Fensteraktionen und Events abbildet. Im Script kapseln Objekte und deren Methode Fensternachrichten, Tastatur- und Mausereignisse. Zur Repräsentation der einzelnen Tests dienen Funktionen, die Objektzugriffe gruppieren. Einzelne von ihnen lassen sich zusammenfassen und können so komplexe Testsuites bilden.

Auf Fehler während des Tests muss das Werkzeug reagieren können. Die meisten Produkte erlauben es, einen Basiszustand der Anwendung anzugeben, zu dem sie bei einem Fehler zurückkehren, der die Weiterarbeit der Applikation verhindert. Mit eingefügten Checkpoints überprüft das Testtool, ob zuvor definierte Ergebnisse erreicht werden oder ob Fehler auftreten. Somit können beispielsweise layoutspezifische Eigenschaften (Position, Größe, Farbe) im direkten Zusammenhang mit den funktionalen Eigenschaften (Inhalt einer Message-Box, etc.) der Applikation überprüft werden. Nach dem Testablauf werden Reports erstellt, die alle Ergebnisse auflisten

Die Verständlichkeit des Codes kann durch ein sogenanntes Name-Mapping deutlich verbessert werden und erlaubt es dem Tester, leicht wartbare Skripte zu erstellen. Dies wird durch die Festlegung eines logischen Bezeichners für die oft unhandlichen, aus Laufzeitinformationen gewonnenen Namen grafischer Elemente umgesetzt.

Neben der Aufzeichnung mit dem Recorder lassen sich Scripts im Editor erstellen und nachbearbeiten. So kann man sie sowohl um neue Nutzeraktionen als auch um das zusätzliche aufrufen von Prüfmethoden erweitern. Zu den Bearbeitungsmöglichkeiten dieser Scripts gehört es, sie beliebig umstrukturieren zu können. Funktionen lassen sich aufteilen oder zusammenfassen. Durch geschickte Nutzung von Funktionsparametern kann man aus dem Mitgeschnittenen wiederverwendbare Blöcke extrahieren. Aus Gründen der Wartbarkeit stellt die Nachbearbeitung nicht nur eine Option dar, sondern ist unbedingt notwendig.

Liegen Scripts vor, kann das automatische Testen beginnen. Durch das wiederholte Abspielen der zuvor aufgezeichneten Skripte sind die Testszenarien zu jeder Zeit reproduzierbar. Dazu wandelt das Tool wieder in Betriebssysteminformationen, die es an die Benutzungsoberfläche des Probanden schickt. Der Mauszeiger bewegt sich wie von Geisteshand über die Oberfläche, Fenster werden geöffnet oder geschlossen und Textfelder werden mit Zeichen gefüllt. Sollte sich mal das Layout geändert haben, so können die GUI-Elemente aufgrund der Objektorientierung bei Testwiederholung dennoch wieder erkannt werden.

Funktionserweiterung durch Bildvergleich

Zusätzlich zu diesen Standardprüfungen kann man die Scripts etwa durch benutzerdefinierte Funktionen um eigene Testbedingungen erweitern. Erwähnenswert ist weiterhin die Möglichkeit, während des Testens Bildschirmbereiche, zum Beispiel ein Fenster oder ein Control, mit aufgezeichneten Versionen zu vergleichen. Bei der Testerstellung speichert man Teile der Anwendung als Bild und verwendet sie während der Kontrolle als Referenz. Das Ergebnis dieses Vergleichs erscheint als Differenzbild, in dem Abweichungen schnell zu finden sind. Einige Produkte stellen OCR-Funktionen zur Verfügung, um Texte aus den Bildern zu extrahieren. Dieser Bildvergleich wirkt auf den ersten Blick ein wenig plump und ist wegen äußerer Faktoren wie der aktuellen Farbtiefe abhängig vom jeweiligen Testrechner. Trotzdem kann er in einigen Fällen eine sinnvolle Ergänzung zur Programmkontrolle im Blackbox-Test sein.

Eigentlich könnte die Welt jetzt in Ordnung sein: die Anwendung ist programmiert, ein Testtool angeschafft, die Tests erstellt und man kann testen lassen – immer und immer wieder. Aber leider steckt der Teufel wie so oft im Detail. Wer Testtools der beschriebenen Art einsetzen möchte, sollte die damit einhergehenden Schwierigkeiten bei seiner Entscheidung berücksichtigen.

Testen ist kein Kinderspiel

Sieht das Erstellen der Tests anfänglich noch wie ein einfaches Aufnehmen aus, entpuppt es sich bald als komplexe Programmieraufgabe. Dies liegt insbesondere an dem in der Regel notwendigen Einbau weiterer Prüfbedingungen und der erforderlichen Umstrukturierung der Testscripts zur Verbesserung der Wartbarkeit, weil in den meisten Fällen die aufgezeichneten Scripts bloß lineare Sequenzen von Aktionen und Ergebnisprüfungen sind. Besonders bei größeren Projekten steht und fällt mit Letzterer der erfolgreiche Einsatz der Technik. Zudem können Eigenheiten der Anwendung Anpassung erfordern. Gerade bei komplexen GUIs mit verschachtelten Eingabe-Controls beziehungsweise benutzerdefinierten Komponenten kann es vorkommen, dass der beim Aufzeichnen generierte Code nicht die gewünschten Benutzereingaben erzeugt. In solch einem Fall steigt der Aufwand zusätzlich durch die Fehlersuche im Testscript.

Nicht zu unterschätzen ist der Aufwand, den die Wartung der Tests während des Lebenszyklus eines Programms erfordert. Einfach ist die Erweiterung der Programmoberfläche etwa durch neue Eingabemasken – dafür sind nur neue Tests zu erzeugen. Ändert sich jedoch das GUI, müssen schnell große Teile des Testcodes neu erstellt werden. Aufgrund dieser Anpassungen sollten man die Testprojekte in der Versionsverwaltung einbinden.

4 GUI-Testwerkzeuge im Vergleich

In diesem Kapitel wird ein Überblick über einige Testwerkzeuge geliefert, die größtenteils in der Industrie, an den Universitäten und Fachhochschulen eingesetzt werden. Auf der Suche nach geeigneten Testtools für Java Swing Anwendungen wird man in Büchern über Softwaretests und im Internet schnell fündig. Für verschiedene Testverfahren und Technologien gibt es eine sehr große Auswahl. Neben den großen bekannten Werkzeugen, wie den IBM Rational Robot, den Mercury WinRunner oder Compuware Silktest, die auf das Capture-and-Replay Verfahren basieren, gibt es natürlich eine große Anzahl von kleinen freien Werkzeugen. Sie sind in der Regel nicht so umfangreich, bieten meist keine Integration in die weiteren Entwicklungswerkzeuge und decken nur reine Funktionstests ab. Dafür stehen sie frei zur Verfügung.

Robot / Functional Tester von IBM Rational

Vor dem Kauf von IBM unterstützte Rational Software Corporation Unternehmen darin, Software-Entwicklungsprojekte plan- und vorhersehbar zu machen, Projektzeiträume zu verkürzen und dabei gleichzeitig die Qualität der erstellten Lösung zu sichern. Durch die Kombination von Software Engineering Best Practices, marktführenden Werkzeugen und umfassenden Dienstleistungen decken Rationals integrierte Lösungen den gesamten Software-Entwicklungszyklus ab. Diese offenen Plattform-Lösungen werden von Hunderten von Partnerfirmen ergänzt, die mehr als 500 komplementäre Produkte und Dienstleistungen liefern. 96 Unternehmen der Fortune 100 nutzen Rational-Lösungen zur schnelleren Entwicklung von hervorragender Software. IBM bietet jetzt in ihrer Rational-System-Testing-Produktfamilie zwei Testtools für automatische Funktionstest von grafischen Benutzeroberflächen an. IBM Rational Robot ist eines der bekanntesten und marktführenden Produkten in der Industrie und wurde im September 2002 von Yphise als bestes Automatisierungswerkzeug für Funktions- und Regressions-Tests ausgezeichnet. Functional Tester for Java und Web ist der Nachfolger des IBM XDE Tester. Beide Produkte haben einen weitgehend gleichen Funktionsumfang, weisen aber auch wesentliche Unterschiede auf.

Robot

IBM Rational Robot gibt Testern die Möglichkeit, die Funktions- und Regressionstests für .NET-, Java-, Web- und andere GUI-basierte Anwendungen zu automatisieren. Zusätzlich zur Integration in andere Werkzeuge der Rational Suite (z.B. TestManager zur Verwaltung von Testfällen) unterstützt Rational Robot auch die Einbindung des Testwerkzeugs in verschiedenste Entwicklungsumgebungen wie Microsoft Visual Studio, Oracle Developer, Delphi und verschiedene Java-Entwicklungsumgebungen. Robot kann sowohl für Tests auf Windows- als auch auf Unix- Plattformen verwendet werden.

Robot bietet ähnliche Unterstützung zur Aufzeichnung von Testfällen wie vergleichsweise WinRunner. Anders als bei WinRunner muss hier der Quellcode der zu testenden Applikation

verändert werden, um Ergebnisse aus dem System zu bekommen. Die Erstellung eines Tests geschieht entweder durch klassisches Capture-and-Replay-Verfahren oder durch das Testfactory-Tool. Bei Capture-and-Replay werden die aufgezeichneten Testskripts in einer BASIC-ähnlichen Sprache, die SQABASIC genannt wird, erstellt. Erstellt man hingegen einen Test mit dem Testfactory-Tools, analysiert das Werkzeug das GUI der zu testenden Applikation. Basierend auf diesen Daten erzeugt es anschließend automatisch ein Testskript.

Rational Robot unterstützt ebenfalls zahlreiche Programmiersprachen. Besonders hervorzuheben sind die Unterstützung von HTML, XML, Oracle Forms und verschiedene UI-Technologien für alle möglichen Einsatzzwecke, angefangen von Java und webbasierter Funktionalität bis hin zu allen VS.NET-Steuerelementen, einschließlich VB.NET, J#, C# und Managed C++. Somit bietet Rational Robot anscheinend nach als einziges Produkt auf dem Markt umfassende Unterstützung zum automatisierten Testen von Microsoft .Net-Software.

Tests von Web-Applikationen können auf verschiedenen Web- Browsern mit denselben Test-Skripts ausgeführt werden. Zusätzlich bietet Robot umfangreiche Debug-Funktionen zum Beheben von Fehlern in Test-Skripts.

Rational Robot stellt Anwendungsbeispiele für allgemeine Objekte wie Menüs, Listen und Bitmaps sowie spezialisierte Anwendungsbeispiele für spezifische Objekte der Entwicklungsumgebung zur Verfügung.

Robot erleichtert die Umsetzung von manuellen Tests in automatische Tests. Regressionstests mit IBM Rational Robot stellen einen geeigneten frühen Schritt in der Automatisierung dar, weil das Tool laut einiger Anwender leicht zu handhaben ist und Tester bei der Untersuchung von Automatisierungsprozessen während des Ablaufs unterstützt.

Im Vergleich zu den anderen Produkten fällt vor allem die Integration in die Rational-Suite und damit die Ausrichtung auf einen gesamtheitlichen Software-Entwicklungsprozess positiv auf. Rational bietet allerdings kein Name-Mapping an, worunter die Lesbarkeit der Skripte leidet.

Unterstützende Umgebung

Operating Systems: Windows 98, Windows NT, Windows 2000 und Windows XP
Browser: Netscape Navigator, Internet Explorer
Supports multiple languages: Java, HTML and DHTML, Visual Basic, Visual C++, ActiveX, XML
Supports multiple IDEs: Microsoft VisualStudio .NET, Oracle Developer/2000, Delphi, PeopleSoft, PowerBuilder

Quellen: http://www-306.ibm.com/software/info/ecatalog/de_DE/products/K108274U58759I63.html?
http://www-306.ibm.com/software/awdtools/tester/robot/ (Stand: 12/2004)

Functional Tester for Java and Web

Der Functional Tester for Java and Web ist ein Eclipse-basiertes Tool für die Automation funktionaler Tests von Java- und Webanwendungen. Es wird die Bedienung der Anwendungsoberfläche aufgezeichnet, wobei Verifikationspunkte auf Ausgabefelder und

Oberflächeninhalte erstellt werden können. Die Aufzeichnung wird anschließend in ein "Testscript" bestehend aus Java-Klassen umgewandelt.

Das System stellt ein hochentwickeltes Funktional- und Regressions-Testtool für Tester und GUI-Entwickler dar. Es unterstützt mit hoher Kontrollfunktionalität das Testen von Java, Microsoft Visual Studio .NET wie auch Web-basierter Anwendungen. Die Testanwendung unterstützt Test-Neulinge mit Test-Automationen wie "Data-driven Testing". Ein Highlight ist sicherlich die ScriptAssure-Technologie, welche widerstandsfähiges, wiederverwendbares Scripting in Java-basierten Testscripts ermöglicht. Dieses Novum dürfte bei häufigen Code-Änderungen von Entwicklern und Testern sehr geschätzt werden.

Wichtig bei der Entwicklung der aktuellen Testsoftware-Generation war die Anforderung mit der gleichen, leistungsstarken und professionellen Eclipse-Entwicklungsumgebung zu arbeiten, welche auch von den Entwicklungstools der WebSphere Studio-Familie schon bekannt sind. Ferner gibt es nun ein Versions-Management, welches die parallele Entwicklung von Testscripts unterstützt. Dies berücksichtigt auch die Integration von Rational ClearCase LT, das Versions-Management-Tool von Rational.

Der Rational Functional Tester überprüft mit J2EE, J2SE, HTML, DHTML, XML, JavaScript erstellte Anwendungen und Java Applets – und schließt insbesondere eine Unterstützung für die SWT-Java-Bibliothek der Eclipse-Shell mit ein. Es wertet dynamisch generierte Daten ohne manuellen Eingriff aus und minimiert so die Scriptverwaltung. Interessant ist auch die Verwaltung aller Testobjekte in einem zentralen Repository. Änderungen an Objektmerkmalen werden so auf alle entsprechenden Testscripts übertragen.

Das Testsystem unterstützt die Betriebssysteme MS Windows 2000, Linux, MS Windows NT und MS Windows XP. Ferner werden die IBM-Plattformen der 3270 (zSeries) and 5250 (iSeries) von der Testsoftware berücksichtigt.

Quellen: http://www-306.ibm.com/software/info/ecatalog/de_DE/products/C108274I57640Y75.html
http://www-306.ibm.com/software/awdtools/tester/functional/index.html (Stand: 12/2004)

Rational Software hat eine neue, automatisierte Technologie für das Testen von Java-Anwendungen entwickelt. Mit Rationals neuer Adaptive-Test-Playback-Technologie passen sich Tests automatisch an Änderungen und Revisionen von Anwendungen an. So erweitert die neue, funktionale Regressionstest-Technologie Rationals umfassende Softwareentwicklungs-umgebungen für Projektteams, die java- und webbasierte Business-Applikationen erstellen. Die neue Adaptive-Test-Playback-Technologie von Rational bietet die erste selbstkorrigierende Objekterkennung, welche die Anzahl der erforderlichen Anpassungen von bestehenden Tests während des gesamten Lebenszyklus einer Anwendung deutlich reduziert. Testeinrichtungen profitieren von verkürzten Testzeiten und typischerweise automatisierenden Test-Tools: So lassen sich Test-Scripts fortlaufend mit den Objekten abgleichen, die in der neuen Version einer Software verändert wurden. Darüber hinaus identifiziert Rationals Adaptive-Test-Playback-Technologie selbsttätig Zielobjekte in java- und webbasierten Anwendungen - automatisch und ohne eine zeitaufwändige neue Zuordnung der Objekte. Damit steigt die Zuverlässigkeit der

Tests. Mit den Java-Lösungen von Rational können Entwickler Runtime-Fehler, Speicherlecks und Performance-Abfälle erkennen und beseitigen. Rational's Entwicklungs-Tools decken den gesamten Software-Entwicklungszyklus ab vom Requirement Management, Software Configuration Management und Test-Management bis zur Steuerung des Entwicklungsprozesses, womit die Kommunikation in den Projekt-Teams erleichtert wird. Vielfältige Trainings und Services sowie eine webbasierte Java-Entwicklungs-Community ergänzen die umfassenden Softwareentwicklungslösungen von Rational.

Für einen raschen Start sowie zur Gewährleistung einer optimalen Produktivität mit IBM Rational-Produkten bietet IBM verschiedene Serviceoptionen, die u. a. technische Unterstützung, professionelle Services sowie von Ausbildern geleitetes und webbasiertes Training anbieten.

WinRunner™ / Quicktest Professional™ von Mercury

Zu Mercurys Functional Testing gehören die eigenständigen Funktionstestprodukte WinRunner™ und Quickstep Professional™. Es ist nach Unternehmensangaben „die umfassendste Lösung für Funktionstest und Regressionstestautomatisierung der Branche mit Unterstützung für nahezu alle Softwareapplikationen und Umgebungen".

WinRunner ™

Anders als manch andere Produkte besteht es nur aus einer Applikation für die gesamte Testerstellung und –durchführung und kann auch in das Testmanagement-Werkzeug TestDirector von Mercury integriert werden. WinRunner kann auf verschiedenen Windows-Versionen eingesetzt werden. Zusätzlich zum Prüfen von Programmen unter Windows bringt WinRunner erweiterte Unterstützung für die Kontrolle von Web-, Visual Basic- und Powerbuilder-Applikationen mit.

Die Erstellung eines Testscripts beginnt bei diesem Produkt mit dem – weitgehend automatischen – Anlegen einer sogenannten GUI-Map, die für die Umsetzung von Laufzeitkennung auf logische Bezeichner zuständig ist. Sofern die Entwickler des Testobjektes den internen Bezeichner eines GUI-Elements von einer Version zur nächsten ändern, führt das beim nächsten Ablauf der Testautomatisierung zu einem Automation Error. Bei Winrunner werden die Bezeichner, welche in der Testautomatisierung verwendet werden, von den Bezeichnern, welche der Entwickler verwendet, getrennt und dieses Mapping an einer zentralen Stelle - der sogenannten „GUI-Map" - verwaltet. Als Ergebnis ist die Testautomatisierung sehr schnell und leicht an geänderte Bezeichner von GUI-Elementen anzupassen. Danach kann man damit beginnen, Testscripts aufzuzeichnen und anschließend im Editor zu bearbeiten. Darüber hinaus können während der Aufzeichnung manuell Änderungen im Testscript vorgenommen werden. Neben der Aufzeichnung von Eingaben können Kontrollpunkte gesetzt werden, bei welcher definierte Ergebnisse mit den tatsächlichen Ergebnissen verglichen werden. Diese Kontrollpunkte müssen jedoch nicht in den Quellcode eingefügt werden, sondern werden von WinRunner verwaltet. Es gibt verschiedene Typen von Checkpoints: Text, GUI, Bitmap, URL Links

und Datenbank. Mit einem Bitmap-Checkpoint kann verifiziert werden, dass ein Bild (z.B. das Firmenlogo) an einer bestimmten Stelle erscheint. Bei fehlgeschlagenen Kontrollpunkten werden Screenshots für die weitere Auswertung automatisch aufgenommen. Die Speicherung der Testfälle erfolgt in einem proprietären Format, das zur Erstellung von Testfällen entwickelt wurde und TSL (Test Scripting Language) genannt wird. Neben dem Erstellen und Abspielen von Tests kann WinRunner Datenbank-Werte verifizieren, um zu überprüfen, dass eine Datentransaktion tatsächlich stattgefunden hat. Nach dem Ausführen der Tests werden die Ergebnisse übersichtlich dargestellt und farblich unterschiedlich gekennzeichnet.

Um Tests mit unterschiedlichen Eingabedaten auszuführen, bietet WinRunner einen DataDriver Wizard. Damit kann ein aufgezeichneter Prozess in einen datengesteuerten Test umgewandelt werden. Die dazu benötigten Daten können entweder direkt in ein Spreadsheet eingegeben oder von einer externen Applikation wie z.B. einer Datenbank eingelesen werden. Auch das Einfügen von Funktionen in das Testscript findet über einen Wizard statt. Aus verschiedenen Gruppen von Funktionen kann die Geeignete ausgewählt werden. Mit einem weiteren Wizard können Schnittstellen zu unbekannten, non-standard Objekten erstellt werden. Durch eine weitere Benutzerschnittstelle kann die Ausführung von Tests gesteuert werden. Die zu testende Applikation wird durch Winrunner automatisch gesteuert.

Zusätzlich zu Tests auf den bereits beschriebenen Programmiersprachen unterstützt WinRunner Delphi, HTML, XML, Oracle Forms und weitere 30 Umgebungen. Test-Scripts von Web-Applikationen können auf verschiedenen Web-Browsern ausgeführt werden. Das heißt, die Testfälle müssen nicht für jeden Browser neu aufgezeichnet werden. WinRunner bietet Funktionen zum automatischen Neustart der zu testenden Software beim Auftreten eines Fehlers oder eines Absturzes. Mercury bietet auch ein Werkzeug für Belastungstests an, in das WinRunner integriert werden kann. Mit der neu hinzugefügten Funktion „E-Mail Benachrichtigung“ kann bei einem fehlgeschlagenen Test sogar der Tester unterwegs per E-Mail an seinen Posteingang, Handy oder PDA benachrichtet werden.

„Gegenüber anderen Produkten fallen bei diesem Tool vor allem das ausgereifte Name-Mapping sowie die einfache übersichtliche Skriptsprache TSL auf.“ (ix 10/2003)

Unterstützende Umgebung

Operating Systems: Windows NT, Windows 95, Windows 98, Windows 2000 und Windows XP
Web Environments: Netscape Navigator, Internet Explorer, AOL, DHTML, HTML, XML, JavaScript
Client-Server: Windows applications (Win32), Visual Basic, Java, ActiveX
Enterprise Applications: Oracle, PeopleSoft, Siebel, Baan
Programming Environments: PowerBuilder, Oracle Developer, Delphi, Centura, VisualAge Smalltalk, Forte, Janus Grids
Terminal Emulators : 3270, 5250, VT100
Server Technologies : Oracle, Microsoft, IBM, ODBC

Quelle: http://www.mercury.com/de/products/quality-center/functional-testing/winrunner/
(Stand: 12/2004)

Quicktest Professional™

Mit Quickstep Professional bietet Mercury eine Alternative zur kompletten Testsuite Mercury Functional Testing und unterstützt gegenüber WinRunner andere Umgebungen. Die Einarbeitung in Mercury QuickTest Professional wird unerfahrenen Testern mit Hilfe des Computer Based Training (CBT) erleichtert und innerhalb weniger Minuten können eigene Testskripts durch Capture and Replay erstellt werden. Durch einen einfachen Knopfdruck wird ein typischer Geschäftprozess für eine Applikation aufgezeichnet. Jeder aufgezeichnete Schritt des Geschäftsprozesses wird automatisch mit einem Satz und Screenshot in englischer Sprache dokumentiert. In der Stichwortansicht können Benutzer Testschritte einfach ändern, entfernen oder neu anordnen.

In QuickTest Professional wird die tastaturgesteuerte Testtechnologie eingeführt, bei der keine Konfigurationen erforderlich sind. Dadurch wird eine schnelle Testentwicklung, einfachere Testwartung und leistungsstärkere Datensteuerungsfähigkeit ermöglicht. QuickTest Professional führt automatisch Prüfpunkte ein, die die Eigenschaften und Funktionen von Applikationen prüfen, beispielsweise zur Validierung der Ausgabe oder zur Prüfung der Linkgültigkeit. Für jeden Schritt in der Stichwortansicht werden auf einem entsprechenden ActiveScreen Einzelheiten zu der getesteten Applikation angezeigt. Es können auch für alle Objekte mehrere Arten von Prüfpunkten hinzugefügt werden, um zu prüfen, ob sich die Komponenten erwartungsgemäß verhalten. Dazu klickt man einfach auf das Objekt auf dem ActiveScreen. Anschließend können Testdaten in die Datentabelle – eine integrierte Tabellenkalkulation mit der vollen Funktionalität von Excel – einfügt werden. Dort hat der Tester die Möglichkeit, Datensätze zu bearbeiten und mehrere Testiterationen zu erstellen, ohne jegliche Programmierung. So werden mehrere Testfälle abgedeckt. Daten können über die Tastatur eingegeben oder aus Datenbanken, Tabellenkalkulationen oder Textdateien importiert werden. Fortgeschrittene Tester können ihre Testkripte in der Expertenansicht anzeigen und bearbeiten. Dort ist das zu Grunde liegende branchenführende VBScript zu sehen, das QuickTest Professional automatisch generiert. Alle in der Expertenansicht vorgenommenen Änderungen werden automatisch mit der Stichwortansicht synchronisiert.

Nachdem ein Tester ein Skript ausgeführt hat, zeigt ein TestFusion Bericht alle Aspekte des Tests an: eine Ergebnisübersicht, eine erweiterbare Strukturansicht des Testskripts mit genauen Angaben zu Applikationsfehlern, verwendeten Testdaten, Screenshots der Applikation für jeden Schritt, auf denen Abweichungen markiert sind, sowie genauen Erläuterungen aller Prüfpunkterfolge und Fehler. Durch die Kombination von TestFusion Berichten mit Mercury TestDirector können alle Mitarbeiter der Qualitätssicherung und Entwicklung Berichte gemeinsam nutzen. Dank der Technologie für die automatische Dokumentierung werden Testdokumentation und Testentwicklung in einem Schritt ausgeführt.

QuickTest Professional vereinfacht zudem den Aktualisierungsprozess. Wenn sich eine getestete Applikation ändert, z. B. bei der Umbenennung einer Anmeldungsschaltfläche, kann eine Aktualisierung am Shared Object Repository vorgenommen werden. Diese Aktualisierung wird dann auf alle Skripte übertragen, die mit diesem Objekt verbunden sind. Testskripte lassen sich

in Mercury TestDirector veröffentlichen, so dass andere Mitglieder des Qualitätssicherungsteams Ihre Testskripte erneut verwenden können. Auf diese Weise werden keine Aufgaben doppelt ausgeführt.

Unterstützende Umgebung

Operating Systems: Windows NT, Windows 95, Windows 98, Windows 2000 und Windows XP
Web Environments: Netscape Navigator, Internet Explorer, AOL, ActiveX, DHTML, HTML, XML
Client-Server: Windows applications (Win32), Visual Basic, Java, ActiveX
Terminal Emulators : 3270, 5250, VT100
Enterprise Applications: Oracle, PeopleSoft, Siebel, mySAP, .Net Add-in
Server Technologies : Oracle, Microsoft, IBM, ODBC

Quelle: http://www.mercury.com/de/products/quality-center/functional-testing/quicktest-professional/
(Stand: 12/2004)

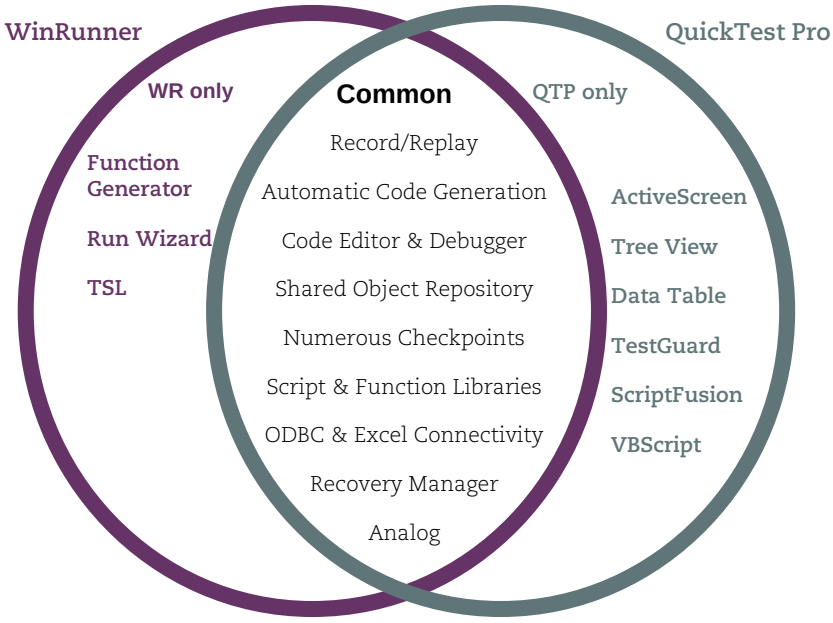


Abbildung 16 - WinRunner vs QuickTest Pro

Werden die Funktionen beider Produkte benötigt, bietet das Unternehmen mit Mercury Functional Testing Produktsuite die Komplettlösung. Es können „gemischte Skripte“ erstellt werden und jedes Produkt kann die Skripte des Anderen aufrufen. Die Ergebnisse werden nach dem Test in einer gemeinsamen Berichtsoberfläche gesammelt. Beide Produkte lassen sich vollständig in andere Mercury Testlösungen integrieren, z. B. LoadRunner für Lasttests und TestDirector für ein globales Testmanagement. Für X-Window basierte Systeme bietet Mercury das Produkt XRunner mit weitestgehend ähnlichen Features an.

SilkTest von Seague

SilkTest ist ein klassisches Funktions-, Regressions- und GUI-Test-Tool mit einer benutzerfreundlichen Umgebung. Seague bietet mit diesem Produkt eine Palette von Features an, die es dem Benutzer ermöglichen bei der Softwaretestautomatisierungsumsetzung in kürzester Zeit produktive Ergebnisse zu erzielen und zu unterstützen. Diese Features enthalten Ablaufelemente für die Testentwicklung und Anpassung, die Testplanung und Management, direkten Datenbankzugriff und Validation, die flexible und robuste 4Test® Scripting-Sprache, ein eingebautes Wiederherstellungssystem für die unbeaufsichtigten Tests und die Fähigkeit über mehrere Plattformen, Browser und Technologien mit einem Satz von Scripts zu testen. Zusammen mit WinRunner und JUnit gehört es zu den meist genutzten Testausführungstools für Tesmanagement-Programmen in der Industrie.

SilkTest ermöglicht das Erstellen automatischer Testabläufe sowohl für Programme mit grafischen Benutzerschnittstellen als auch für Aufrufe auf der Kommandozeile. Das Testen von Datenbank-Schnittstellen und API-Aufrufen ist ebenfalls möglich. Ähnlich wie das Konkurrenzprodukt von Rational-IBM besteht es aus mehreren Applikationen, die zum Testen von Windows-, Web- oder Java-Applikationen dienen: Editor, Controller und Agent. SilkTest ermöglich das Testen einer Applikation auf unterschiedlichen Plattformen, wobei die erstellten Testpläne unverändert plattformübergreifend genutzt werden können. Die Tests können über Szenarios auf unterschiedliche Maschinen mit unterschiedlichen Betriebssystemen verteilt werden.

Auch hier beginnt die Erstellung eines Testskripts mit dem Aufzeichnen. Als Sprache kann wahlweise die objektorientierte Skriptsprache genutzt oder direkt in C programmiert werden, wobei der volle Sprachumfang zur Verfügung steht. Während der Testaufnahme entsteht eine GUI-Abstraktions-Schicht ähnlich der GUI-Map des WinRunner, anhand derer sich Änderungen im GUI-Design zentral nachvollziehen lassen, da sie sich nur an einer Stelle auswirken. Das Recovery-System ermöglicht eine definierte Behandlung von unerwarteten Fehlern oder Programmabstürzen während des Testlaufes. Die zu testende Applikation kann nach einem solchen Fehler in einen vorher definierten Zustand gebracht oder neu gestartet werden. Die mit SilkTest erstellten Testpläne können mit gängigen Versionsverwaltungsprogrammen verwaltet werden. Die zentrale Verwaltung der GUI-Objekte durch SilkTest gewährleistet eine maximale Wartbarkeit der Skripte.

Pluspunkte von SilkTest sind laut eines Vergleichs mit GUI-Testwerkzeugen in der Ausgabe 10/2003 des ix-Magazins, das gut durchdachte Name-Mapping sowie die objektbasierte Skriptsprache, wodurch seine Skripte zu den am einfachsten nachvollziehbaren und wartbaren im Test wurden. Im Vergleich waren unter anderem die Produkte Teststudio (IBM), WinRunner (Mercury) und Testcomplete (Automated QaCorp.)

Unterstützende Umgebung

Operating Systems: Windows NT, Windows 95, Windows 98, Windows 2000, Windows XP und Windows Server 2003
Web Environments: Netscape Navigator, Internet Explorer, AOL, DHTML (JavaScript and CSS) , HTML, XML, Microsoft HTC/HTA

.NET Standalone WinForms and WebForms, Visual Basic 6/ActiveX, JDK 1.1x - 1.4.2_01, Web browser control, PowerBuilder 6.5, 7.0, 8.0 & 9.0, Win32, MFC, 3270/5250 Applications via BlueExpress, Unicode, Siebel 6, 7 & 7.5, PeopleSoft 8, Citrix Transactions

Quelle: http://www.segue.com/products/functional-regressional-testing/silktest.asp (Stand: 12/2004)

QARun von Compuware

Der Softwarehersteller Compuware bietet mit seinem Produkt QARun eine weitere alternative Lösung im Bereich GUI-Testen und ist Teil der QACenter-Produktreihe. QACenter ist eine Sammlung von Werkzeugen zum Testen von Software und zur Test-Automatisierung. Ein besonderer Schwerpunkt liegt dabei auf der Bewertung von Anwendungen in unterschiedlichen Umgebungen sowie auf der Identifizierung und Behebung von Problemen. QARun kann, wie zum Beispiel auch WinRunner, Tests auf allen Windows-Plattformen durchführen.

Die Aufzeichnung von Testfällen erfolgt wie auch schon bei anderen Werkzeugen, durch Aufzeichnung von Testfällen mittels Recorder. Bei QARun muss der Quellcode der zu testenden Applikation erweitert werden, um Rückmeldungen des Systems zu erhalten. QARun verwendet, wie auch die anderen Produkte, ein eigenes Format zur Definition von Testfällen und unterstützt diverse Programmiersprachen. Zusätzlich zu den Standardsprachen werden HTML und Oracle Applikationen unterstützt. Außerdem ist es möglich, Tests von SAP R/3 und Siebel-Systemen durchzuführen.

Compuware bietet seit kurzer Zeit mit Testpartner einen Nachfolger bzw. ein umfangreicheres Produkt für die QACenter-Familie an. TestPartner erweitert die Funktionalitäten von QARun für Anwendungstests: es bietet mehr Flexibilität, Produktivität und darüber hinaus nicht-proprietäres Skripting durch die weitverbreitete Sprache Microsoft VBA (Visual Basic for Applications). Der Nachfolger baut weitestgehend auf denselben Konzepte und Features des bis dato für Compuware erfolgreichen QARun:

- Capture & Replay
- Database Repository
- Object/Image Mapping
- Kontrollen
- Events
- Logs
- Identifizierung
- Alias Mapping
- Support for multiple environments
- Asset Versioning
- Modular Scripting (Funktionen)

Mit diesem Lösungsangebot stellt Compuware nach eigenen Angaben gegenüber QARun den Entwicklungsteams eine leistungsstärkere und flexiblere Testautomatisierungstechnolgie zur Verfügung. Die visuelle aufzeichnende Technologie und das automatische Scripting beschleunigen das funktionale Testen von komplexen Microsoft, Java und webbasierten Anwendungen. Anfänger können durch die schnelle Einführung und Hilfe eigene Tests einfacher erzeugen und ausführen.

Weitere relevante Informationen für GUI-Tests waren über die Produktbeschreibung von Compuware und weiteren Quellen nicht umfassend beschrieben. Auf mehrere Anfragen beim Support blieb bis jetzt eine Rückantwort aus. Ein weiteres Manko für die Anschaffung von QARun oder anderen Produkten von Compuware ist, dass für eine sorgfältige Evaluierung der Produkte keine kostenfreie Testlizenz erhältlich ist.

Quelle: http://www.compuware.com/products/qacenter/default.htm (Stand: 12/2004)

qftestJUI von QFS

Die Quality First Software GmbH (QFS) widmet sich hauptsächlich dem Thema Softwarequalität und bietet mit qftestJUI ein innovatives Produkt zur Erstellung, Ausführung und Verwaltung von automatischen Tests für Java Swing Anwendungen mit grafischer Benutzungsoberfläche. Durch die vollständige Implementierung in Java legt das Produkt den Grundstein für plattformübergreifende Tests. Die Wiederverwendbarkeit der Tests streicht der Hersteller als wichtiges Features seines Produktes heraus und qftestJUI kommt mit größeren Veränderungen im Programm zurecht. Mit qftest ist es möglich, beliebig viele Programme gleichzeitig zu starten und damit komplexe Systeme als Ganzes zu testen. Es läuft unter Windows und Unix mit JDKs von Sun, IBM und Blackdown von 1.1 bis 1.5. Die Arbeit mit qftestJUI ist intuitiv und Tester sollten damit bereits nach wenigen Stunden erste lauffähige Tests erstellt haben. Die grundlegenden Funktionen sind direkt über die grafische Oberfläche verfügbar und ermöglichen es einfache Tests zu erstellen. Die Aufnahme eines manuellen Test lassen sich mit einem Mausklick starten. Tastatur- und Mausaktionen sowie die Reaktionen der Anwendung werden aufgezeichnet, welche dann mit den Elementen der GUI und den zugehörigen Daten automatisch in eine Baumstruktur integriert werden. Mit der hierarchischen Struktur soll unter anderem den schnellen Zugriff auf die aufgezeichneten Informationen gewährleisten und eine gute Übersicht über die Daten und deren Zusammenhänge geben.

Die direkte Aufnahme von Testsequenzen ist nur ein kleiner Teil der Automatisierung von GUI Tests mit qftest. Mit modernen Kontrollstrukturen und Steuerungsmöglichkeiten bietet es weitere Funktionalitäten:

- Variablen, Packages, Prozeduren, Parameter
- Bedingungen
- Fehler-/Ausnahmebehandlung
- Starten von beliebigen Programmen
- Schleifen

Die Testsuiten und Protokolle werden als XML Dateien abgespeichert, was sich bereits zu einem weitverbreitenden Industriestandard entwickelt hat. Zum Suchen und Ersetzen und bei der Überprüfung von Daten, die an der Oberfläche angezeigt werden, können reguläre Ausdrücke verwendet werden. qftestJUI bindet hierzu das gnu.regexp-Paket ein, welches eine Java-Implementierung der klassischen Bibliothek für reguläre Ausrücke auf der Basis endlicher Automaten ist. Das Paket enthält das Java-Archiv und den kompletten Quellcode, da es unter der GNU Library General Public License steht. Da das gesamte Java API zur Verfügung steht, sind die Möglichkeiten grenzenlos. QftestJUI bietet noch für den fortgeschrittenen Anwender, der noch mehr Kontrolle über seine Anwendung benötigt, mit der Skriptsprache Jython Zugang zu einem weiteren Standardpaket. Jython ist eine 100% pure Java-Implementierung der Programmiersprache Python. Jython ermöglicht die Ausführung von Python-Programmen auf jeder Java Plattform. Damit hat der Tester Zugriff auch auf die internen Informationen seiner Anwendung, zu denen kein direkter Zugang über das GUI besteht.

Häufige Ablaufmuster eines Tests lassen sich in Prozeduren auslagern, die von beliebigen Stellen aus aufgerufen werden können. Diese können über Parameter gesteuert und somit vielseitig wiederverwendet werden. Ein analoger Mechanismus erlaubt es, vielfältige Informationen in statischen oder dynamischen Variablen abzulegen und später darauf zuzugreifen, z.B. um den Inhalt einer Tabellenzelle auszulesen und später mit dem Inhalt eines Textfeldes zu vergleichen. Die Münchner Firma hat sich in der Qualitätssicherungsbranche einen guten Namen gemacht und präsentierte sich erstmalig als Aussteller bei der CeBIT2004.

JCFUnit (JUnit)

Die JUnit-Erweiterung JFCUnit wurde initial von Matt Caswell und Greg Houston ins Leben gerufen und das Open-Source-Projekt zur Zeit maßgeblich von Vija R. Aravamudhan weiter entwickelt. Mit dessen Hilfe können mit der Programmiersprache Java Komponententests geschrieben und automatisch ausgeführt werden. Somit können direkt mit der selben Entwicklungsumgebung wie sie für die Anwendungsentwicklung benutzt wird, Tests geschrieben werden. Das in Java geschriebene Testframework JUnit ist ein Unittest-Tool, dient also dem Testen von isolierten Programmeinheiten wie einzelne Methoden, Klassen oder Modulen. Es setzt an den öffentlichen Schnittstellen dieser Einheiten an. Trotzdem ist es ein White-Box-Testverfahren, weil es die Möglichkeit und auch die Bereitschaft voraussetzt, bestehenden Code zu ändern, insbesondere um Schnittstellen zu ändern bzw. diese überhaupt erst einzuführen um eine bestimmte Testbarkeit zu erreichen. Mit JUnit werden wiederholbare Testfälle, also Testdaten und das erwartete Ergebnis, in derselben Programmiersprache erstellt wie der Testling selbst. Diverse gesammelte Tests können als Suiten zusammengestellt werden.

JFCUnit ist eine Bibliothek zum Durchführen JUnit-kombatibler Tests auf Swing basierender Komponenten. Sie ist unter der GNU Lesser General Public License (LGPL) über die Projekt-Homepage [jfc04] frei verfügbar. Sie bietet im wesentlichen Unterstützung beim automatisierten Testen auf Swing basierender Komponenten durch weitere Methoden zum

- Aufspüren von java.awt.Window-Instanzen (z.B. Frames und Dialoge), die vom zu testenden Code geöffnet wurden.
- Lokalisieren von Swing-Komponenten im Komponentenbaum eines Fensters anhand des Typs, des Namens oder beliebiger anderer Eigenschaften.
- Abschicken gezielter Events im AWT Event Handling Thread, z.B. das Klicken auf einen Button oder die Auswahl eines JTree-Teilbaumes.
- Thread-sichere Testen und Abfragen von Komponenten.

Seit der Version 2.0 stellt JFCUnit XML-Recording und -Playback zur Verfügung und ermöglicht es dem Benutzern schnelle und automatische Testscripte zu erzeugen und editieren. Die XML APIs sind OpenSource und erlauben es dem Entwickler dort eigene XML Tag Handlers zu definieren.

JFCUnit hakt bei der Simulation von Events in die Event Queue von Swing oder in die des Betriebssystems ein. So können Komponenten in Swing-Oberflächen lokalisiert und explizit für diese Events ausgelöst werden. Ein Testprogramm simuliert damit die Bedienung durch den Benutzer nach und prüft erwartete Reaktionen. Gleichzeitig können auch Events aufgenommen und gespeichert werden. Dieses Feature erleichtert das Debugging.

Auf der Ebene automatisierter Klassen- oder Komponententests hat sich im Java-Umfeld das Test-Framework „Junit“ mit seinen Erweiterungen als Standard etabliert und wird zunehmend integraler Bestandteil gängiger Entwicklungsumgebungen, wie z.B. „Eclipse“.

Quelle: http://sourceforge.net/projects/jfcunit/
http://jfcunit.sourceforge.net/ (Stand: 12/2004)

Jemmy von NetBeans

Jemmy ist eine Java-Bibliothek, mit der sämtliche Benutzeraktionen wie etwa Knopfdrücke, Texteingabe oder Baumexpandieren gespeichert und wiederholt werden können. Dadurch wird der regelmäßige, entwicklungsbegleitende Test auch über Benutzerschnittstellen ermöglicht. Es kann sowohl getrennt als auch zusammen mit dem NetBeans IDE verwendet werden. Jemmy kann auch für automatische Demos von Java GUI-Anwendungen verwendet werden. Somit kann einfach gezeigt werden, wie eine Anwendung arbeitet. JemmyTest ist ein in Java geschriebenes Programm, das die API von Jemmy verwendet, um GUI-Anwendungen zu testen. Jemmy ist ein NetBeans unabhängiges Modul.

Quelle: http://jemmy.net-beans.org (Stand: 12/2004)

Abbot

Abbot Framework ist eine Java-Bibliothek für GUI-Unittests und GUI-Funktionstests. Es stellt Methoden zur Erzeugung von User-Level-Aktionen (ähnlich wie *java.awt.robot*) und zur Überprüfung des Zustandes von GUI-Komponenten zur Verfügung. Zwei Möglichkeiten stehen zum Erstellen eines Tests zur Verfügung. Entweder wird der Test innerhalb eines JUnit-Testfalls direkt im Code eingebunden (Test-First-Entwicklung) oder über das Erstellen von Skripten mittels des Skripteditors.

Quelle: http://abbot.sourceforge.net/ (Stand: 12/2004)

Welches ist das beste Produkt?

Der anfängliche Aufwand Testwerkzeuge einzuführen ist auf Grund des großen Funktionsumfangs der Produkte relativ hoch. Die Einarbeitungszeit variiert stark vom Umfang der beiliegenden Dokumentation der einzelnen Produkte. Generell sollte man vor dem Erwerb eines kommerziellen Testwerkzeugs mit einer Evalierungsversion die grundsätzliche Eignung des Produktes für den gewünschten Einsatz ausprobieren. Da die Software-Entwicklung unterschiedliche Anforderungen stellt, ist es sehr schwierig, objektiv ein „bestes“ Produkt zu bestimmen. Eine weiterer Aspekt bei der Auswahl des geeigneten Testtool ist die Kostenrechnung. Ein kommerzielles GUI-Testtool kostet leicht einen vierstelligen Euro-Betrag pro Lizenz. Bei Änderungen sind mehr oder weniger aufwändige Anpassungen an den Testscripten erforderlich, auch das verursacht Kosten. Dann sollte man noch berücksichtigen, dass nicht alle Testfälle automatisiert werden können. Man muss nach wie vor ca. 30% - 50% des Tests manuell durchführen. Es ist also alles andere als einfach, automatisches Testen einzuführen und dennoch Kosten zu sparen. Einige neu erschienene Handbücher für Testautomation erläutern einige Beispiele für diese Kostenrechnungen. Für kleinere GUI-Anwendungen oder Projekte sind die kostengünstigen und einfach zu Hand habende OpenSource GUI-Testtools für Java-Oberflächen geeignet.

Die Übersicht zeigt, dass einige der beschriebenen Werkzeuge großteils dieselben oder ähnliche Funktionen bieten. Leider gibt es keine Vereinheitlichung der Testscripts und auch keine Konvertierungsmöglichkeiten, was einen Austausch von Testfällen oder einen Umstieg auf ein anderes Werkzeug aufwendig macht. Die Unterschiede zwischen den Produkten sind lediglich in Details zu finden, die jedoch hier nicht alle angesprochen werden können. Eine Entscheidung für ein Testwerkzeug kann demnach nur bei Betrachtung dieser Details im Hinblick auf eine spezielle Anwendung getroffen werden. Auch eine mögliche Integration in bereits vorhandene Software kann dabei eine ausschlaggebende Rolle spielen. Für detaillierte Informationen zu den einzelnen Produkten sei auf die Homepage des jeweiligen Herstellers verwiesen. Im Literaturverzeichnis sind einige Links aufgelistet, die eine weitere Auswahl von Testwerkzeugen anbieten.

5 Evaluation der Testwerkzeuge

Bei der Auswahl von Testwerkzeugen muss man mit Bedacht auswählen. Welches Werkzeug erfüllt die vorgegeben Kriterien der Testanforderungen. Der Prozess der Auswahl selbst ist eine sehr zeitaufwendige Arbeit. In diesem Kapitel werden anhand einer kleiner GUI-Anwendung die Unterschiede zweier Testwerkzeuge in der Bedienung und Einarbeitung herausgefunden und erläutert. Bei der Auswahl wurde als Kriterium die Verbreitung und Verfügbarkeit betrachtet. Aufgrund meiner fehlenden Erfahrung mit Testwerkzeugen, fiel die Wahl durch Empfehlungen und Erfahrungen Dritter für die Evaluation auf JFCUnit und qftestJUI von der Firma QFS.

Im Java-Umfeld hat sich das Test-Framework und OpenSource-Produkt JUnit auf der Ebene automatisierter Unittests mit seinen Erweiterungen als Standard etabliert. Die JUnit-Erweiterung JFCUnit sind das automatisierte Testen von Swing-Komponenten möglich und wird mit JUnit zunehmend integraler Bestandteil gängiger Entwicklungsumgebungen. QFS hat sich schwerpunktmäßig auf die Qualitätssicherung für die Java Plattform spezialisiert. Mit qftestJUI haben sie ein Framework zur Testautomatisierung entwickelt, das zu den Vertretern der Capture/Replay-Testtools gehört. Innerhalb von zwei Jahren seit der Markteinführung von qftestJUI hat QFS einen großen Kundenstamm auf der ganzen Welt für sich gewinnen können und war 2004 auf der CeBIT in Hannover vertreten.

JFCUnit steht zum Download frei zur Verfügung. QFS bietet seinen Kunden eine kostenlose Evaluationsversion zum freien Download an. Die bereitgestellte Version ist für den Zeitraum von vier Wochen mit uneingeschränkter Funktionalität der kommerziellen Version ausführbar.

Für eine annähernde Beurteilung und Auswertung wurden einige Testfälle an einer einfachen Java Swing Beispielanwendung angewendet.

Ein Java Swing Beispiel

Der in Abbildung 17 dargestellte Anmelde-Dialog dient zur Demonstration automatisierter Tests beider Testtools. Über die beiden Textfelder wird der Benutzername und das Passwort eingegeben. Per Maus-Klick am Anmelden-Button wird der Anmeldevorgang überprüft. Bei erfolgreicher Anmeldung erscheint ein Nachricht-Dialog „Anmeldung erfolgreich“ oder bei einer falschen Eingabe die Fehlermeldung „Passwort ist falsch. Versuchen Sie es noch einmal“. Mit dem Abbrechen-Button kann der Anmeldedialog beendet werden. Der Anwendungsabbruch wird mit einem Bestätigungs-Dialog “Programm wirklich beenden?“ bestätigt.

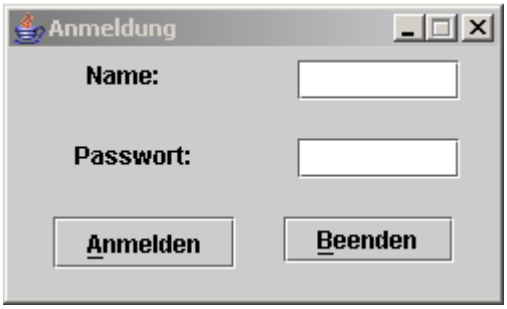


Abbildung 17 - Anmelde-Dialog

JFCUnit

Die JUnit-Erweiterung JFCUnit ist, wie wir schon aus dem vorherigen Kapitel erfahren haben, ein OpenSource-Produkt und in Java geschriebenes Framework zum Test von Swing-Komponenten. Somit können direkt mit derselben Entwicklungsumgebung wie sie für die Anwendungsentwicklung benutzt wird, Tests geschrieben werden. Im Grunde bleibt die Struktur und Ausführung der Testfälle von JUnit erhalten, nur das durch JFCUnit weitere Methoden zur Verfügung gestellt werden. Diese zusätzlichen Methoden und Erweiterungen unterstützen folgende Aufgaben:

- Aufspüren von java.awt.Window-Instanzen (z.B. Frames und Dialoge), die vom zu testenden Code geöffnet wurden
- Lokalisieren von Swing-Komponenten im Komponentenbaum eines Fensters anhand des Typs, des Namens oder beliebiger anderer Eigenschaften
- Abschicken gezielter Events im AWT Event Handling Thread, z.B. das Klicken auf einen Button oder die Auswahl eines JTree-Teilbaumes
- Thread-Sicherheit
- Vereinfachte Testfallbeschreibung mit XML

JFCunit steht in der Version 2.08 unter der Lesser General License (LGPL) auf der sourgeforce Projekt-Homepage[jfc] zum Download zur Verfügung und jeder kann sich an der Weiterentwicklung beteiligen. JFCUnit wird als ZIP Archiv mitsamt Sourcecode und einer kurzen, einführenden Dokumentation ausgeliefert. Für den einwandfreien Testablauf kann JFCUnit nur in Verbindung mit JUnit und Jakarta Regexp installiert werden. Die Zip-Files werden in einem entsprechenden Verzeichnis entpackt und der CLASSPATH für die jar-Files gesetzt. Eclipse- und JBuilder-Anwender brauchen nur ihr entsprechendes Plugin herunterladen und in ihrem HOME-Verzeichnis entpacken[inst]. Über die JUnit-Oberfläche, den Testrunner, können die Testsuites gesteuert werden. Der Testrunner ermöglicht den Aufbau der Testsuiten zu betrachten, sie zu starten, abzubrechen und die Ergebnisse zu betrachten.

Vorbereitung des Test

Testklasse erstellen

Die Erstellung einer Testsuite unterscheidet sich kaum vom Programmieren einer gewöhnlichen Suite. JFCUnit verwendet die Klassen *JFCTestCase*, welche von der Klasse *junit.framework.TestCase* abgeleitet ist, die Klasse *Testsuite* von Junit und einer unterstützenden Klasse *JFCHelper*. Die Klasse JFCTestCase bietet den Rahmen für Testfälle und enthält die interne Thread-Steuerung zu einem Testfall. Die Klasse JFCHelper bietet den Zugriff auf die Swing-Komponenten.

Jeder Testfall wird durch eine eigene Klasse abgebildet. Im ersten Schritt muss zunächst die Testklasse angelegt werden, die von junit.extensions.jfcunit.JFCTestCase ableiten. Somit hat man den Zugriff auf alle Hilfsklassen, deren Methoden zum Finden der Komponenten und Auslösen

der Events. JFCUnit sorgt intern für das Wechseln zwischen Test-Thread und AWT-Event-Thread.

```
import javax.swing.*;
import junit.extensions.jfcunit.*;
import junit.extensions.jfcunit.finder.*;

public class LoginScreenTest extends JFCTestCase
{
    private LoginScreen loginScreen = null;
    public LoginScreenTest(String name)
    {
        super(name);
    }
...
```

In der *setUp()*-Methode wird ein *TestHelper* instanziiert und mit der *setHelper*-Methode zugewiesen. Der *TestHelper* simuliert die Benutzeraktionen und es stehen ihm die Klassen *JFCTestHelper* und *RobotTestHelper* (Package junit.extension.jfcunit) zur Verfügung. Der *JFCTestHelper* erzeugt AWT-Events, die in der AWT-Event–Queue eingestellt werden. Dagegen benutzt *RobotTestHelper* das awt.Robot-API und stellt die Benutzeraktionen auf dem Betriebssystem nach. Obwohl er langsamer ist als der *JFCTestHelper*, deckt er bei den Tests die Schnittstelle zwischen Betriebssystem und Anwendung ab. Das zu testende Fenster kann in der *setUp*-Methode erzeugt werden.

```
// wird vor jeder Testmethode ausgeführt
protected void setUp( ) throws Exception
    {
    super.setUp( );
    // setzen des JFCTestHelper
    setHelper( new JFCTestHelper());
       // AnwenderDialog
    loginScreen = new LoginScreen();
    loginScreen.setVisible(true);
}
```

In der tearDown()-Methode entfernt nach einem Test alle übrig gebliebenen Fenster und Dialoge vom Desktop, um somit einen konsistenten Zustand für die anderen nachfolgenden Tests zu gewährleisten.

.

```
//wird nach jeder Testmethode ausgeführt
protected void tearDown( ) throws Exception
    {
    loginScreen = null;
    TestHelper.cleanUp(this);
    super.tearDown( );
    }
```

Testfälle erstellen

Jeder Test wird in dem Testfall durch eine eigene Methode geschrieben, die anhand ihrer Signatur erkannt wird. Methoden, die mit dem Präfix test beginnen, den Rückgabewert *void* und eine leere Parameterliste haben, werden als Tests ausgeführt.

```
//...Testmethode...
public void testUserAndPasswordEmpty()
{
...
}
```

Ein wichtiger Bestandteil von JFCUnit sind die sogenannten „Finder“, deren Aufgabe ist, einzelne GUI-Komponenten innerhalb eines Containers aufzufinden. Die Komponenten können anhand ihres Namens oder ihres Typs identifiziert werden. Stellen wir uns zum Beispiel ein Fenster vor, das mehrere verschiedene Komponenten einschließlich eines Buttons mit dem Text, "OK", enthält. Die erste Möglichkeit wäre einen Code zu schreiben, der alle Komponenten des Window Containers durchläuft bis der passenden Buttons gefunden ist. Alternativ könnte dieses Vorgehen die *findButton()*-Methode des JFCTestHelpers übernehmen. Zum Auffinden weiterer Komponenten steht eine große Anzahl von weiteren Methoden zur Verfügung. Beispielsweise sucht der CompontFinder die Komponente anhand ihres Typs und der NamedComponentFinder benutzt den Namen zur Suche.

Im ersten Schritt werden die GUI-Komponenten, die für den Test benötigt werden, mithilfe des Named-ComponenentFinder ermittelt. Der Finder sucht nach einer Komponente des Typs JComponent mit dem Namen *Name_jTextField* und *Passwort_PasswordField*. Mit der find()-Methode wird die erste Komponente mit dem Index 0, auf die diese Suchkriterien zutreffen, gesucht. Die assert()-Methode überprüft, ob das Textfeld gefunden wurde und nicht ungültig ist.

```
// Komponenten auffinden
NamedComponentFinder name_finder = new NamedComponentFinder (JComponent.class,
"Name_jTextField");
JTextField jtField = (JTextField)name_finder.find(0);
assertNotNull("Could not find  Name_jTextField", jtField);

name_finder = new NamedComponentFinder (JComponent.class,
"Passwort_jPasswordField");
JTextField jpField = (JPasswordField)name_finder.find(0);
assertNotNull("Could not find  Passwort_jPasswordField", jpField );
…
```

Der „Finder“ ist ein erweiterbares Interface, dass es dem Entwickler ermöglicht die Suchkriterien für jeden Komponententyp selbst zu definieren damit die Objekte gefunden werden. Verschiedene allgemeine Finder-Ausführungen werden zur Verfügung gestellt.

```
public class AbstractButtonFinder
public class AbstractWindowFinder
public class FrameFinder
public class DialogFinder
```

```
public class JLabelFinder
public class JmenuItemFinder
…
```

JUnit stellt für das Testen eine Reihe von Zusicherungsmethoden zur Verfügung. Die Assert-Methoden dienen jeweils dazu, eine bestimmte Aussage zuzusichern. *assertEquals* etwa überprüft auf Gleichheit und *assertTrue* testet den Wahrheitswert eines boolschen Ausdrucks. Die Methode *fail* dagegen dient dazu, einen Test fehlschlagen zu lassen. Dies wird beispielsweise benötigt, wenn getestet werden soll, ob eine bestimmte Exception geworfen wird. Alle Zusicherungsmethoden besitzen einen optionalen ersten Parameter *message* vom Typ *String*, der dazu dient aussagekräftige Fehlermeldungen anzugeben, falls der Test an dieser Stelle fehlschlagen sollte.

`assertTrue( booleancondition)`	ist eine Bedingung wahr
`assertEquals(Object expected, Object actual)`	sind zwei Objekte gleich
`assertFalse(booleanconditon)`	ist eine Bedingung falsch
`assertNull(Object object)`	ist eine Objektreferenz null
`assertNotNull(Object object)`	ist eine Objektreferenz nicht null
`assertSame(Object expected, Object actual)`	zwei Objekte gleiche Referenzen
`assertNotSame(Object expected, Object actual)`	zwei Objekte nicht gleiche Referenzen
`fail()`	Testfall einfach fehlschlägt.

Die Überprüfung des Anfangszustandes der Maske kann durch die assert()-Methoden erfolgen. Das Name- und Textfeld soll editierbar und leer sein.

```
//Anfangszustand prüfen
//Textfeld editierbar?
assertTrue( „Username field is not editable”, userNameField.isEditable() == true);

//Textfelder leer?
assertEquals( "Username field is not empty", " ", userNameField.getText( ) );
assertTrue( "Password field is not empty",passwordField.getText( ).length() == 0 );
```

Im nächsten Testschritt werden Benutzername und Passwort in die entsprechenden Textfelder über den JFCTestHelper eingegeben. Der Anmelde-Button sollte *enable* sein und ausgelöst werden. Sobald die Komponenten alle gefunden und überprüft wurden, wird ein Maus-Klick mit der *enterClickAndLeave()*-Methode simuliert. Das zu erwartende Ergebnis ist die Anzeige eines Nachricht-Dialogs. Mit dem *DialogFinder* wird überprüft wie viele Dialoge sichtbar sind und ob der erwartete Nachrichten-Dialog angezeigt wird. Zu guter letzt wird die *dispose()*-Metode aufgerufen, damit im Falle eines unterbrochenen Test oder geöffneter Dialoge, alle Fenster und Dialoge geschlossen werden.

```
// Daten eingeben
getHelper().sendString( new StringEventData (this, passwordField ,  "startme") );
//Button enable?
assertTrue("ExitButton is not enable", exitButton.isEnabled() == true);
...
```

```
//Nachricht Dialog finden  "Anmeldung erfolgreich
getHelper().enterClickAndLeave( new MouseEventData( this, enterButton ) );
DialogFinder dFinder = new DialogFinder("");
//2 Sekunden warten
dFinder.setWait(2);
List showingDialogs = dFinder.findAll( );
assertEquals( "Number of dialogs showing is wrong", 1, showingDialogs.size( ) );
     dialog = ( JDialog )showingDialogs.get( 0 );
assertEquals( "Wrong dialog showing up", "Nachricht", dialog.getTitle( ) );
//Dialog(e)& Fenster schließen
TestHelper.disposeWindow( dialog, this );
```

Um Mausaktionen ausführen zu können, stellt die JFCTestHelper Klasse mehrere MouseEventData-Ausführungen zur Verfügung. Diese enthält Methoden, die unterschiedlich parametrisiert werden können und somit viele verschiedene Aktionen ermöglichen. Beim Aufruf der Methode *enterClickAndLeave()* wird entsprechend der gewünschten Aktion ein parametrisierter MouseEventData mitgegeben. In diesem Fall ist dies ein einfacher Klick auf den *enterButton* (2. Parameter). Als erster Parameter wird immer die aktuelle Testklasse mit übergeben

```
public void enterClickAndLeave(AbstractMouseEventData evtData)

public void enterDragAndLeave(AbstractMouseEventData srcEvtData,
                              AbstractMouseEventData dstEvtData,
                              int incr)
```

Weitere MouseEventData Ausführungen werden für viele verschiedene Swing-Klassen zur Verfügung gestellt.

```
MouseEventData                  für Generic Component
JComboBoxMouseEventData         für JComboBox
JListMouseEventData             für JList
JTabbedPaneMouseEventData       für JTabbedPane
JTableHeaderMouseEventData      für JTableHeader
JTableMouseEventData            für JTable
JTextComponentMouseEventData    für JtextComponent
```

Standardmäßig ist die Click- und Drop-Positionen im Zentrum des Komponenten-Rechtecks durch das MouseEventData vorgegeben. Jedoch können diese Werte durch Positionskonstanten (z.B. NORTH, EAST, CENTER...), mit kundenspezifischen Werten (x, y) ausführlich in die Komponente oder durch proportionale Prozent-Werte (xpercent, ypercent) angepasst werden. JFCTestHelper stellt zur Unterstützung von Drag & Drop die *enterDragAndLeave()*-Methode zur Verfügung und erleichtert unter anderem Tastatur-Events. Die Methode *sendString()* und *sendKeyAction ()* ermöglichen das Senden von Tasteneingaben in die Event Queue für bestimmte Komponente. Weitere hilfreiche Helper-Methoden und deren Beschreibung sind in der API-Dokumentation auf der JFCUnit Homepage zu finden.

Test durchführen

Zur Ausführung und Übersetzung der Testfälle ist die Erweiterung des Klassenpfades um Junit3.8 und JFCUnit unter Java 2 notwendig. Das Ausführen geschieht über folgenden Aufruf:

```
java junit.textui.Testrunner jm.jfcunit.LoginScreenTest
```

Benutzer von Eclipse müssen nur den Run-Button drücken. Der Testrunner vom Junit compiliert und startet den Test. Junit stellt bei der Ausführung eine konsolenbasierte oder eine grafische Ausgabe zur Verfügung. Die grafische Variante hat sich dabei mehr durchgesetzt. Der grüne Balken stellt den Fortschritt beim erfolgreichen Testen dar. Wenn er sich rot färbt, ist mindestens ein Test fehlgeschlagen. Der Test wird sofort unterbrochen, wenn eine Bedingung nicht erfüllt ist. In dem unteren Feld „Failure Trace“ wird angezeigt in welchem Test und in welcher Codezeile der Fehler aufgetreten ist. Eine Besonderheit dabei ist, dass man dem Test zuschauen kann. Das Anmelde-Fenster wird geöffnet und die Aktionen werden darauf sichtbar ausgeführt. Man kann sogar selbst in dem Fenster arbeiten, was allerdings nicht empfehlenswert ist, da es den Testablauf stören kann.

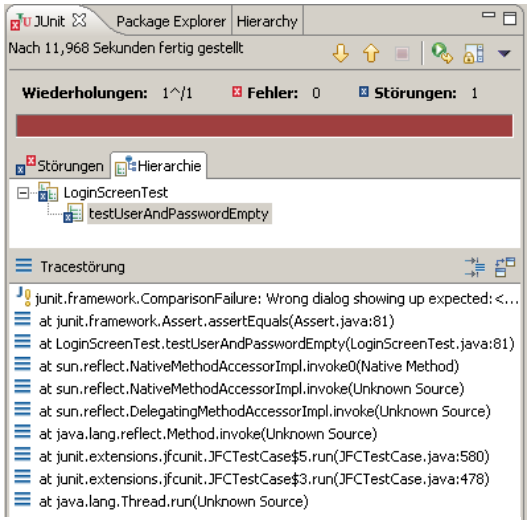


Abbildung 18 - Test nicht erfolgreich

JFCUnit mit XML

In der beschriebenen Vorgehensweise könnte man seine kompletten Testfälle ausprogrammieren. Bei umfangreichen und komplexen Tests kann dies aber schnell unübersichtlich und fehleranfällig werden, was unbedingt vermieden werden sollte. Dieses weitere Feature erlaubt es dem Entwickler XML-Code zu schreiben, der den Hauptteil der Testsuite enthält. Das meiste vom Testgenerationsprozess kann über die Aufnahme zum XML automatisiert werden. Auf diese Vorgehensweise wird später noch eingegangen. Mit diesem Feature ist es aber nicht die Absicht der Entwickler die Java Kodierung komplett zu ersetzen. Es soll eher die Definition und Manipulation von Testdatensätzen unterstützen und vereinfachen. Das aufwändige Programmieren jedes kleinsten Schrittes soll vermieden werden.

XML-Datei erstellen

Für jedes vorkommende Tag (<*suite*>, <*test*>, <*find*>,...) gibt es einen sogenannten TagHandler. Wenn beim Auswerten der Datei ein Tag gelesen wird, so wird der zuständige TagHanlder erzeugt und übernimmt die Bearbeitung des Tags. Der ClickTagHandler ist beispielsweise für das Simulieren von Mausklicks zuständig und der FindTagHandler für das Auffinden von Komponenten. Abhängig von der Konfiguration der Atribute in dem XML-Element können verschiede Finder-Klassen genutzt werden.

Das untere Beispiel zeigt den Einsatz der Klassen, die bereits im vorigen beschriebenen Testfall verwendet wurden. Die Referenz auf die gefundene Komponente wird einer Variablen *id = kompontenname* zugewiesen. Später kann dann diese Referenz mit *refid = komponentname* verwendet werden.

```
<suite name= "XMLTestSuite">
   <test name="LoginScreen" robot="true">

   <!--Komponenten finden-->
      <find finder="NamedComponentFinder" id="userNameField"
           class="java.swing.JTextField" index="0" name="LoginNameTextField"/>
      <find finder="NamedComponentFinder"id="exitButton"
           class="java.swing.JButton" index="0"name="ExitButton"/>
      …
   <!--Prüfungen-->
      <assertenable refid="exitButton" message=" ExitButton is not enable "
           enable="true"/>
      …
   <!--Eingaben-->
      <key refid="userNameField" string="admin"/>

    </test>
</suite>
```

Zum Bedienen der verschiedenen GUI-Komponenten gibt es eine Vielzahl von speziellen TagHandlern, die die Beschreibung des Testfälle vereinfachen. Es gibt weitere Tags zum Prüfen von erwarteten Werten. Daneben existieren Tags, die es ermöglichen, die Durchführung von einzelnen Testschritten von Bedingungen abhängig zu machen oder aber Schleifen bei der Durchführung einzubauen. Damit können Tabellen und Listen zeilenweise ausgelesen werden. Eine besondere große Rolle spielen das *<suite>* und das *<test>*-Tag. Sie veranlassen den dynamischen Aufbau der TestSuite mit den TestCases. Hier eine kleine Auflistung von gängigen Tags und ihren TagHandlers.

Mithilfe des *<procedure>*-Tags können Testabläufe definiert werden und mehrfach verwendet werden. Wenn ein bestimmter Ablauf wiederholt gebraucht wird, wie z.B. das Ausfüllen der Felder zum Anlegen eines Datensatzes, kann er in einer *<procedure>* gekapselt werden. An den Stellen, an den der Ablauf verwendet werden soll, kann die *<procedure>* dann mit *call* aufgerufen werden. Es besteht auch die Möglichkeit, Parameter zu übergeben, die dann in der *<procedure>* verwendet werden können.

Mit dem *<taghanderls>*-Tag können neue Tags registriert oder bereits vorhandene überlagert werden. Der Name des Tags und die Tag-Handler-Klasse werden angegeben und mit *add* wird das entsprechende Tag registriert. Wenn das Tag bereits vorhanden ist, wird es schließlich überlagert. Mit *remove* kann ein Tag dann wieder entfernt werden..

```
<suite name= "XMLTestSuite">
   <test name="LoginScreen" robot="true">
```

```
    <!--Registrieren eines neuen TagHanlder-->
       < taghandlers action="add" tagname="mytag"
            classname="myproject.taghandler.MyHandler"/>

       find finder="NamedComponentFinder" id="userNameField"
            class="java.swing.JTextField" index="0" name="LoginNameTextField"/>
       <find finder="NamedComponentFinder"id="exitButton"
            class="java.swing.JButton" index="0"name="ExitButton"/>
       …
    <!--Definition der Procedure-->
       <procedure name="insert">
       <find finder="NamedComponentFinder" id="userNameField"
            class="java.swing.JTextField" index="0" name="LoginNameTextField"/>
       <key refid="userNameField"string="${name}"/>
       </procedure>

    <test name="LoginScreen" robot="true">

    <!--Verwendung der Procedure-->
       <procedure call="insert" nachname="Mustermann"/>

    <!--Verwendung des neuen Tags-->
       <mytag name="test"/>

     </test>
  </suite>
```

Voraussetzung für Java Codierung

Um eine XML-Datei ausführen zu können, muss lediglich die Java-Klasse *XMLTestSuite* anstelle der *TestSuite* verwendet werden. Dies ist die einzige Java-Klasse, die zum Ausführen der Testfälle geschrieben werden muss. Sie kennt die XML-Datei und veranlasst die Auswertung der Tags.

```
       public class TestXMLLoginScreen extends XMLTestSuite
       {

           public static final String DOCUMENT_FACTORY =
                "javax.xml.parsers.DocumentBuilderFactory";

           public TestXMLLoginScreen() throws Exception
        {
               super("testLoginScreen.xml",
                     XMLUtil.readFileFromClassContext(
                        TestXMLLoginScreen.class, "testLoginScreen.xml"));
               SwingSet.main(new String[] {});
                }

           public static Test suite() throws Exception
        {
```

```
            return new TestXMLLoginScreen();
             }

        public static void main(String[] args) throws Exception
    {
            if (System.getProperty(DOCUMENT_FACTORY) == null) {
                System.setProperty(DOCUMENT_FACTORY,
                "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
                }
            TestRunner.run((Test) TestXMLLoginScreen.suite());
             }
    }
```

Aufnahme mit JFCUnit

Seit der JFCUnit Version 2.00 können jetzt auch Benutzeraktionen zur Testfallerstellung aufzeichnet werden. JFCUnit stellt dafür die JFCEventManager API zur Verfügung um bei der Ausprogrammierung der Testfälle zu helfen. Dazu müssen lediglich die XML-Tag <*record*> und <*save*> in die XML-Datei eingefügt werden.

Wenn beim Testdurchlauf das <*record*>-Tag erreicht wird, zeichnet JFCUnit alle Aktionen auf der Oberfläche der zu testenden Anwendung auf. Die Aufzeichnung der Aktionen wird mit CTRL+D beendet. Die neuen Tags werden vor dem <*record*>-Tag in die Datei hinzugefügt. Auf diese Wiese können bestehende Dateien an beliebiger Stellen erweitert werden. Anschließend sorgt das <*save*>-Tag dafür, dass die neuen Aktionen die Datei unter einem anderen Namen gespeichert werden wird. Für die Aufzeichnung von Benutzeraktionen kann folgendes Template verwendet werden.

```
<suite name="Suite">
    <test name="Login" robot="true">
        <record/>
    </test>
    <test>
        <save file="reclogin.xml"/>
    </test>
</suite>
```

Damit die aufgenommen Aktionen für einen Testfall verwendet werden können, müssen die erstellten Tags für die noch nicht vorhandene Prüfung überarbeitet werden. An einige Stellen sollte der generierte Ablauf optimiert werden. Einige Texteingaben könnten beispielsweise zusammengefasst oder Fehleingaben/Fehlklicks entfernt werden.

Weitere nützliche Informationen über das Aufnehmen mit JFCUnit und class extensions des TagHandler können in der API-Dokumentation auf der JFCUnit Homepage gefunden werden.

Erfahrung mit JFCUnit

JFCUnit erweitert JUnit, mit der Möglichkeit Swing-Funktionalitäten im Ganzen, die auf Nutzinteraktionen beruhen, zu testen. Bei üblichen JUnit-Tests macht die Oberfläche einer Anwendung einen erheblicher Teil des Codes aus, der nicht erfasst wird. Der Vorteil liegt vor allem in der Implementierung in Java. JFCunit kann somit die Swing-Komponenten besser handhaben als andere nicht in Java geschriebene Testtools. Die Tests sind sehr dicht am tatsächlichen Geschehen einer grafischen Oberfläche und das Verhalten von mehreren offenen Fenstern und Dialogen kann getestet werden.

JUnit ist vor allem als Testumgebung von Extreme Programming bekannt geworden. In dieser Entwicklungsmethode wird der Testcode geschrieben bevor überhaupt der zu testende Code dazu geschrieben wird. Der Test wird also gleichzeitig mit der Software entwickelt. Der Programmierer muss sich also zuerst überlegen was seine Klasse tun soll. Mit den Tests legt er den Rahmen dazu fest. Wird dann die eigentliche Software dazu geschrieben, hat man eine klare Kontrolle darüber, ob die Bedingungen die man vorher in der Form eines Tests festgelegt hat, erfüllt sind oder nicht. Schlägt der Test fehl, sind entweder die Erwartungen an die Klasse falsch oder die Umsetzung weist Fehler auf. Durch die gleichzeitige Entwicklung von Tests und Software wird sichergestellt, dass jeder Teil des Programms durch Tests geschützt ist. Ändert man die Software zu einem späteren Zeitpunkt und die Tests scheitern, so weiss man, dass die Annahmen, die man früher gemacht hat, nicht mehr zutreffen und ein Fehler sehr wahrscheinlich ist. Regressionsprobleme werden so schnell erkannt.

Wird JFCUnit in der XP-Weise genutzt, so geht das Schreiben von Tests in die normale Arbeitsweise über. Mit der Übung verringert sich auch der Aufwand für die Erstellung der Testfälle. Außerdem erhält der Programmierer direktes Feedback, ob die Funktionalität der Swing-Oberfläche korrekt sind und hat die exakte Kontrolle bei korrekt implementierten Tests. Dies kann die Effektivität der normalen Programmiertätigkeit steigern.

Mit JFCUnit lassen sich einfache funktionale Tests abbilden. Beim funktionalen Test (Blackbox-Test) geht man ausschließlich anhand der Funktionsbeschreibung des Prüflings vor. Im Entwicklungszyklus grafischer Anwenderschnittstellen sind funktionale Tests essentiell für das GUI-Testen. Im Vordergrund steht nicht mehr die Anwenderfreundlichkeit, die sollte im Prototyping verifiziert werden, sondern vielmehr, ob alle erwartete Funktionalitäten der Anwendung erfüllt werden[Glat04]. In Bezug auf funktionale Tests bietet JFCUnit in der jetzigen Form keinen Ersatz zu kommerziellen Umgebungen, aber das kann auch kein Ziel von Unit-Tests sein. Trotz des zur Zeit geringen Funktionsumfanges kann JFCUnit teilweise mit geringen Erweiterungen zum Test von kleinen Komponenten sinnvoll eingesetzt werden. Die Implementierung der Testfälle müssen größtenteils selbst programmiert werden und sind aufwendig. Der komplette Quellcode der Anwendung sollte beim Erstellen der Testfälle dem Tester bekannt sein. Komplexe Anwendungen werden schnell unübersichtlich und Testdurchläufe dauern sehr lang. Tests sollten daher modular aufgebaut sein, um die Wiederverwendbarkeit zu gewährleisten.

JFCUnit bietet neben der Unterstützung der einfachen GUI-Komponenten auch Lösungen für komplexe Komponenten wie Bäume und Tabellen. Mit der Möglichkeit eigene Ergänzungen hinzuzufügen, kann die Behandlung der Standardkomponenten auf die eigenen Bedürfnisse zugeschnitten werden. Auf diese Wiese können auch projektspezifische Komponenten gut in den automatischen Test aufgenommen werden. Im Vergleich zu den anderen Test-Frameworks bietet JFCunit keine komfortable Oberfläche zum Erstellen der Tests. Der JUnit-Testrunner unterstützt nur das Starten des Tests und das Anzeigen der Ergebnisse. Das Erstellen der Testfälle wird durch die XML-Erweiterung mit der Record-Funktion und die Wiederverwendung von Abläufen deutlich vereinfacht.

Um den Umgang mit JFCunit zu erlernen, benötigt man eine gewisse Einarbeitungszeit. JUnit-Erfahrene haben dabei einen entscheidenden Vorteil, dass es sich dabei um eine JUnit-Erweiterung handelt. Dadurch ähneln die GUI-Tests in diesem Falle stark den gewohnten Unit-Tests. Die übersichtliche Struktur und die gute Bezeichnung der Klassen und Methoden erweist sich JFCUnit bei der Benutzung mit einigem Ausprobieren als mehr oder weniger intuitiv. Die Dokumentation, Beispiel, die JFCUnit-Sourcen selbst und das Help-Forum stehen dem Anwender zur Hilfe zum effektiven Einsatz von JFCunit. Im Gegensatz zur Dokumentationsvielfalt von JUnit hat JFCunit noch Nachholbedarf. In nächsten Zeit werden aber auch Bücher erscheinen, die den Umgang mit JFCUnit genauer erläutern.

qftestJUI

Mit qftestJUI bietet Quality First Software ein innovatives Produkt zur Testautomatisierung für Java/Swing Anwendungen mit grafischer Benutzeroberfläche. qftestJUI beinhaltet eine reiche Palette an Funktionen für hohe Produktivität im Testprozess.

- Plattformübergreifende Tests
- Unterstützt Bäume und Tabellen
- Tests tolerant gegenüber Änderungen
- Intuitive Bedienoberfläche
- Übersichtliche Strukturierung
- komplexer Testfälle
- Integrierter Debugger
- Direkter Zugriff auf Java API
- Industriestandards XML, Python

Die Installation von qftestJUI für Windows kann entweder direkt mit dem ausführbaren *exe-File* erfolgen, oder mit dem Zip-Archiv. Die Installation mittels *exe-File* folgt den unter Windows üblichen Konventionen. Um qftestJUI aus dem Archiv zu installieren, wird das Archiv zunächst an einem geeigneten Ort ausgepackt, z.B. C:\Programme\qfs. Zum Start benötigt qftestJUI eine Lizenzdatei, die von Quality First Software GmbH vergeben werden. Ohne Lizenzdatei können keine Testsuites gespeichert und geladen werden, die mit qftestJUI ausgeliefert werden. Die vier Wochen gültige Lizenz wird über ein Formular angefordert und nach Erhalt in das

Wurzelverzeichnis von qftestJUI abgelegt. Danach steht dem Benutzer eine vollständige kommerzielle Version zur Verfügung.

Aufbau von qftestJUI

Der linke Bereich des Hauptfensters enthält eine Baumstruktur und ist zentrales Element qftestJUIs grafischer Oberfläche. Dieser Baum, der die Testsuite repräsentiert, wieder spiegelt die hierarchische Struktur der GUI-Anwendung und vereinigt die Kontrollstrukturen und die Daten zum Automatisieren von GUI Test.

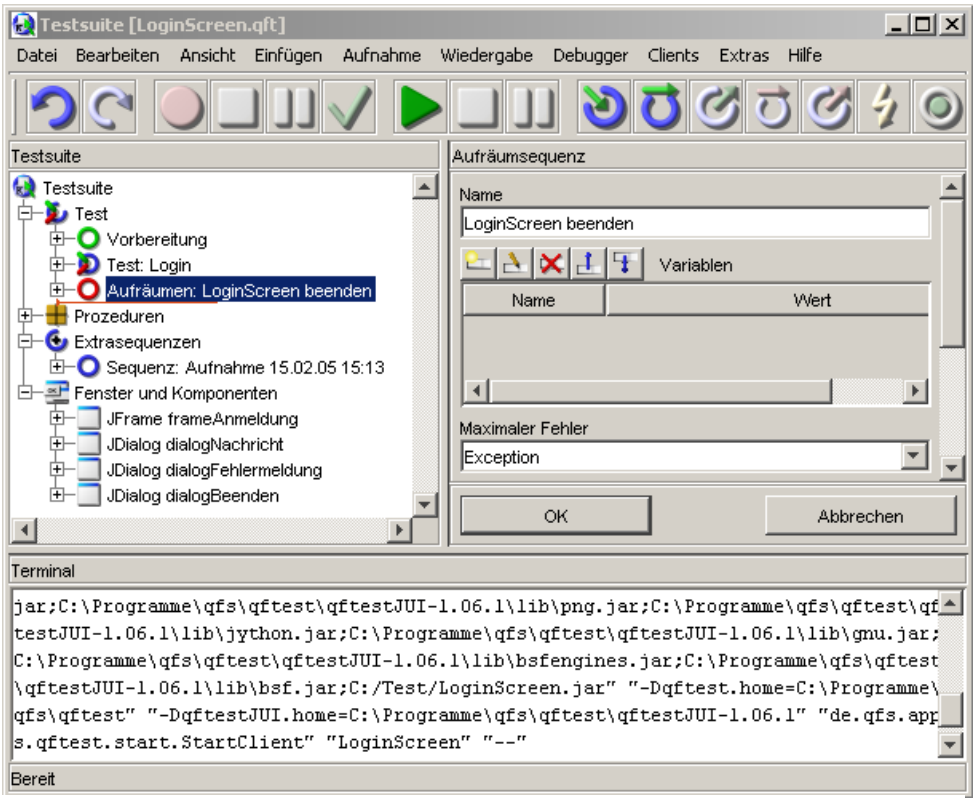


Abbildung 19- Hauptfenster einer Testsuite

Die Grundstruktur einer Testsuite ist fest definiert. Der Wurzel-Testknoten enthält eine beliebige Anzahl von *Tests* und Sequenzen. Sequenzen geben einer Testsuite die Struktur und bestimmen den Ablauf eines Tests. In ihrer Grundform sind sie Container, die ihre Childknoten einen nach dem anderen ausführen. Sie können dabei Werte für Variablen festlegen, die während der Ausführung der Children Gültigkeit besitzen. Ein Test ist eine besondere Sequenz, die es erlaubt, vor und nach der Ausführung jedes ihrer Childknoten spezielle Maßnahmen zu treffen, um einen gesicherten Ausgangszustand herzustellen. Im „Prozeduren"-Knoten können wieder verwertbare Sequenzen in Prozeduren organisiert werden, welche mittels eines Prozeduraufrufs von einer beliebigen anderen Stelle aufgerufen werden können. Der „Extrasequenzen"-Knoten ist so eine Art Spielwiese zum Ausprobieren und Zusammenstellen von Tests. In dieser „Zwischenablage" können beliebig viele Knoten unbehindert von den normalen Einschränkungen abgelegt werden. Im „Fenster und Komponenten"-Knoten sind alle aufgenommenen Fenster und Komponenten des testenden Programms (System Under Test, SUT) mit ihren Eigenschaften enthalten.

Eine Testsuite kann aus mehr als 40 verschiedenen Arten von Knoten bestehen. Jeder Knoten hat bestimmte Eigenschaften, darunter seine Attribute, die *qftestJUI* in der Detailansicht im rechten Bereich des Hauptfensters darstellt und direkt in den Feldern editiert werden können. Zum Editieren einer Testsuite gibt es sowohl Funktionen zur Bearbeitung der Knoten als Ganzes, wie Ausschneiden/Kopieren/Einfügen, als auch für deren Attribute. Im unteren Fensterbereich befindet sich das Terminal, welches die Standardausgaben des zu testenden Clients protokolliert. Die grundlegenden Funktionen zum schnellen Erstellen von einfachen Test sind direkt über die grafische Oberfläche verfügbar. Sie ermöglicht den schnellen Zugriff auf jedes Detail der aufgezeichneten Information und gibt eine gute Übersicht über die Daten und deren Zusammenhänge. Durch die Testsuite navigiert man sich mithilfe des Baumes und wählt einzelne Knoten aus, für die dann jeweils die Details im rechten Fensterbereich eingeblendet werden.

Aufnahme und Wiedergabe

Vorbereitungen für Testsuite

Für die Organisation von den Sequenzen der Testsuite gibt es vier verschiedene Knoten. In jedem Knoten werden die Variablen gebunden und die Childknoten einer nach dem anderen ausgeführt. Anschließend werden die Variablen der Sequenz wieder gelöscht.

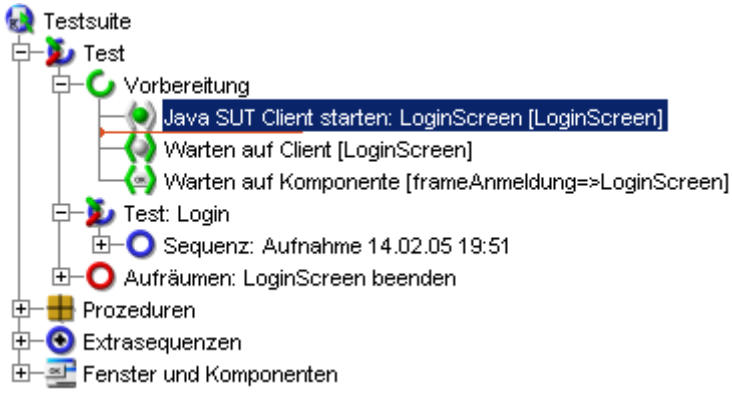


Abbildung 20 - Testsuite

Um die unterschiedlichen Bedürfnisse der Anwender gerecht zu werden, stellt qftestJUI vier verschiedene Möglichkeiten eine Anwendung aus qftestJUI heraus zu starten zur Verfügung. Für die Testanwendung wird in diesem Fall ein „Java SUT Client starten“ Knoten verwendet und mittels `java -classpath <Pfad> <Startklasse>` gestartet. Anschließend wartet Knoten „Warten auf Client“ bis qftestJUI sich erfolgreich mit der neuen Virtuellen Maschine (JVM), in der das SUT läuft, verbunden hat. Nach diesem Vorgang kann es noch eine Weile dauern, bis das erste Fenster auf dem Bildschirm erscheint. Der Knoten ”Warten auf Komponente” wartet auf das Erscheinen einer bestimmten Komponente des SUT. Damit kann davon ausgegangen werden, dass das SUT erfolgreich hoch gelaufen und bereit für Tests ist.

Die aufgenommenen Sequenzen werden im „Test“ Knoten organisiert. Ein Test ist auch eine besondere Sequenz, die es erlaubt, vor und nach der Ausführung jedes ihrer Childknoten spezielle Maßnahmen zu treffen, um einen gesicherten Ausgangszustand herzustellen.

Zum Schluss wird eine ”Aufräumen” Sequenz erzeugt um die Anwendung zu beenden. Sie besteht aus zwei Schritten: Der Knoten „Programm Beenden“ beendet den von qftestJUI gestarteten Prozess und Knoten „Warten auf Programmende“ stellt sicher, dass er auch wirklich terminiert wurde. Die Aufräumsequenz wird, wie auch bei der Vorbereitungssequenz, nach jeder Sequenz bzw. jedem Test innerhalb des umschließenden Tests ausgeführt. Sie dient dazu, die Applikation nach dem Ausführen eines Tests in einen spezifischen Zustand zu versetzen, welcher die Basis für den darauf folgenden Test bildet. Die Aufräumsequenz ist vergleichbar mit ”tear-down” Methoden in Unit-Tests.

Aufnahme von Testsequenzen

Bevor eine Sequenz von Events im SUT aufgenommen werden kann, muss zunächst das *SUT* aus qftestJUI heraus gestartet worden sein und die Verbindung zum *SUT* muss bestehen. Dazu muss ”Java SUT Client starten” Knoten markiert und der „Wiedergabe“ Knopf gedrückt werden. Beim Start des SUT aus qftestJUI heraus, werden Java Anwendungen in separaten (JVM) ausgeführt. Dabei wird zunächst spezieller Code ausgeführt, der eine RMI Verbindung zu qftestJUI herstellt und eine eigene EventQueue installiert, bevor die Kontrolle an das SUT weitergegeben wird. Die EventQueue wird benötigt, um die GUI Komponenten des SUT zu verfolgen und um Events aufzuzeichnen.

Nachdem die Anwendung läuft, wird der Aufnahmeknopf rot dargestellt und aktiviert. Die Aufnahme der Sequenz wird einfach „mit einem Mausklick“ auf den Aufnahmeknopf gestartet. Jetzt befindet man sich im Aufnahmemodus und kann einige Kommandos an der SUT ausführen. Die Aufnahme wird dann mit dem Startknopf beendet. Die aufgezeichneten Events werden in der Testsuite an gewünschter Stelle bzw. als neue Sequenz unterhalb des „Extrasequenz“-Knotens eingefügt. Alle Komponenten, auf die sich die aufgezeichneten Events beziehen, werden automatisch unterhalb des „Fenster und Komponenten“-Knotens in die Testsuite integriert, falls sie noch nicht vorhanden sind.

Abbildung 21 – erstellte Sequenz

In diesem Sequenzbeispiel ist zu erkennen, dass die Sequenz mit einem Mausklick im Login-Textfeld eingeleitet wird. MPRC steht hierbei für Move-Pressed-Released-Clicked, was eine zusammengefasste Version der einzelnen Mauseevents darstellt. Die Koordinaten der Klicks werden relativ zum Nullpunkt der Komponenten „frameAnmeldung.text“ angegeben. Durch den „Texteingabe“-Knoten wird der Text „mustermann“ in das Textfeld geschrieben. Mit dem Tabulator-Event im „PTR“-Knoten wird dann in das Passwort-Textfeld gewechselt. Nachdem das Passwort eingegeben und der Anmelde-Button gedrückt wird, bewirkt der „Warten auf

Komponente“ Knoten, dass auf das Erscheinen eines Fehlermeldungs-Dialogs gewartet wird. Zum Schluss wird der OK-Button gedrückt.

Abspielen von Tests

Den oder die entsprechende Knoten müssen für das Abspielen einer oder mehrerer Test markiert und mit dem „Wiedergabe“ Knopf gestartet werden. Anschließend lässt sich der Ablauf in der Baumansicht verfolgen Bei einer Ausführung werden die Tests der obersten Ebene der Reihe nach ausgeführt. qftestJUI hebt Knoten, die gerade ausgeführt werden, durch einen kleinen Pfeil hervor. Außerdem werden Meldungen über den Fortschritt des Tests in der Statuszeile und die Ausgaben des SUT Clients im Terminal ausgegeben. Nach Abschluss der Startsequenz erscheint die Testanwendung am Bildschirm. Sie ist vollständig unter Kontrolle und den wachsamen Augen von qftestJUI. Die Events der aufgenommene Sequenz wird, wie „von Geisterhand“, ausgeführt. Nach dem Ende des Tests wird das Ergebnis in der Statuszeile angezeigt. Im Idealfall steht dort ”Wiedergabe beendet - keine Fehler”. Andernfalls wird die Zahl der Warnungen, Fehler und Exceptions angezeigt. Im Fehlerfall wird zusätzlich ein entsprechender Meldungsdialog geöffnet.

Aufnehmen von Checks

Auch wenn die Aufzeichnung und Abspielen von Sequenzen unterhaltsam sein kann, geht es doch eigentlich darum herauszufinden, ob sich das *SUT* dabei auch korrekt verhält. Diese Aufgabe übernehmen Checks. Checks vergleichen die Anzeige oder den Status von Komponenten im *SUT* mit vorgegebenen Werten. Der „Check Text“ Knoten zum Beispiel, liest den Text aus einer Komponente aus und vergleicht ihn mit einem vorgegebenen Wert. Stimmen diese nicht überein, wird ein Fehler signalisiert. Weitere Checks:

Check Elemente	Vergleicht eine Liste von vorgegebenen Texten mit der Anzeige einer Komponente oder eines Unterelements.
Checks selektierbare Elemente	Vergleicht zusätzlich zu einer Liste von vorgegebenen Texten auch die Selektion der Unterelemente einer Komponente.
Check enabled Status	Überprüft den enabled Status einer Komponente. Der Check funktioniert mit allen Komponenten, nicht aber mit Unterelementen, mit Ausnahme der Reiter einer JtabbedPane.
Check editable Status	Überprüft den editable Status einer Komponente.
Check selected Status	Überprüft den selected Status einer Komponente.
Check Abbild	Vergleicht ein Abbild einer Komponente mit dem aktuellen Zustand. Der Check funktioniert mit allen Komponenten und mit Unterelementen.
Check Geometrie	Überprüft Position und Größe einer Komponente. Der Check funktioniert mit allen Komponenten, nicht aber mit Unterelementen.

Am einfachsten werden Checks erstellt, indem man sie aufnimmt. Die Aufnahme funktioniert nur, wenn der SUT Client im Aufnahmemodus läuft. Über den Check Knopf wird das SUT in den Checkmodus geschaltet. In diesem Modus werden Events nicht weiter aufgezeichnet. Stattdessen wird die Komponente unter dem Mauszeiger hervorgehoben, was die aktuelle Auswahl signalisiert. Der Check wird durch Anklicken der Komponente aufgezeichnet und der Anwender erhält ein Menü mit allen für diese Komponente zur Verfügung stehenden Arten von Checks. Nach der Auswahl wird der aktuellen Wert der Komponente als Maßgabe verwendet. Welche Komponente in Frage kommen, hängt von der Jeweiligen ab. So stellen manche Komponenten keinen Text dar, so dass z.B. ein Check Text Knoten für einen Scrollbar keinen Sinn ergibt. Um zurück in den Aufnahmemodus zu gelangen, wird der Check Button gedrückt.

Checks und Events lassen sich sehr gut mischen. Durch diese Konstellation kann die Logik der Anwendung getestet werden und nicht nur die zugrunde liegende Swing Implementierung. Damit das ständige Umschalten zwischen dem *SUT* und qftestJUI nicht mühsam wird, kann mittels einer Tastenkombination direkt im *SUT* zwischen Aufnahmemodus und Checkmodus hin- und herschalten.

Abbildung 22 – Check-Erweiterung

Die Sequenz „falsches Passwort“ wird mit der Überprüfung der beiden Textfelder und dem Anmelde-Button in der Testanwendung ergänzt. Die ersten beiden „Check“ Knoten überprüfen, ob die beiden Textfelder leer sind. Nach der Texteingabe kontrolliert der dritte „Check“ Knoten den enable Status des Anmelde-Buttons. Bei einem fehlerhaften Ablauf einer Testsequenz erscheint ein Dialog mit der Meldung „Fehler bei der Wiedergabe...“.

Protokoll

Nachdem ein Testlauf beendet ist, erscheint in der Statuszeile des Hauptfensters von *qftestJUI* eine Meldung mit dem Ergebnis. Im Idealfall lautet diese ”Keine Fehler”. Sind Probleme aufgetreten, wird die Zahl der Warnungen, Fehler und Exceptions angezeigt und gegebenenfalls zusätzlich ein Dialogfenster geöffnet. Dann ist es an der Zeit herauszufinden, was schief gelaufen ist. Mithilfe des Protokolls können diese aufgetretenen Probleme genauer veranschaulicht werden.

Das Protokollfenster ähnelt im Aufbau dem einer Testsuite. Der Baum auf der linken Seite repräsentiert nun aber die zeitliche Darstellung des Testlaufs. Die Zeitachse verläuft von oben

nach unten. Analog zur Testsuite befindet sich auf der rechten Seite eine Detailansicht des jeweils ausgewählten Protokollknotens im Baum.

Jeder Knoten eines Protokolls hat einen von vier Fehlerzuständen: *Normal*, *Warnung*, *Fehler* oder *Exception*. Dieser Zustand wird durch einen Rahmen um das Icon des Knotens dargestellt, dessen Farbe Orange für *Warnung*, rot für *Fehler* oder fett rot für *Exception* ist.

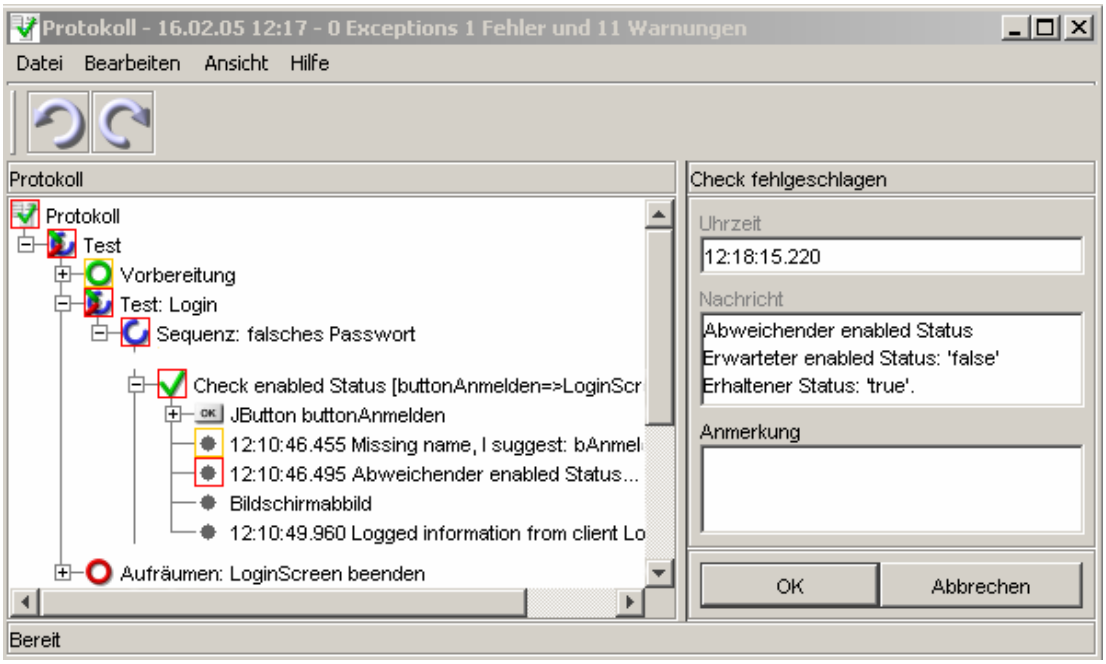


Abbildung 23 - Protokoll

Was sofort auffällt ist der rote Rahmen um den Wurzelknoten des Protokollbaumes. Er verrät uns, dass sich ein Fehler in einem der Kindknoten verbirgt. Die Detailansicht auf der rechten Seite zeigt die Abweichung zwischen erwartetem und gefundenem Text.

Die grundlegenden Bearbeitungsmöglichkeiten im Protokoll sind analog zur Testsuite, mit dem Unterschied, dass die Attribute der Knoten, die aus der Testsuite übernommen wurden, nicht geändert und dass keine Knoten entfernt oder eingefügt werden können. Knoten können aber mit einer Bemerkung versehen werden, z.B. um den Grund für einen Fehler zu dokumentieren.

Reportgenerierung

Im Qualitätssicherungsprozess ist es wichtig Testergebnisse zu dokumentieren und auch zu archivieren. Das Protokoll ist zwar ein hervorragendes Werkzeug zum Aufspüren von Fehlerursachen nach einem Testlauf, als Zusammenfassung bzw. Report jedoch ungeeignet. Andererseits enthält ein Protokoll natürlich alle relevanten Informationen über einen Testlauf, so dass sich das Erstellen eines Reports durch das Zusammenfassen und Formatieren dieser Informationen bewerkstelligen lässt. qftestJUI bietet die Möglichkeit aus Protokollen Testreports zu generieren. *qftestJUI* bietet zwei Arten von Reports - einen einfachen HTML Report und einen XML Report. Die XML Form kann der Anwender als Grundlage verwenden und sie mit Hilfe selbst geschriebener XSLT Stylesheets in einen beliebigen eigenen Report transformieren. Die

Ausgabeform lässt sich durch das entsprechende Auswahlfeld bestimmen. Der Auswahldialog enthält noch weitere Felder um Einfluss auf Inhalt und Aussehen des Reports zu nehmen.

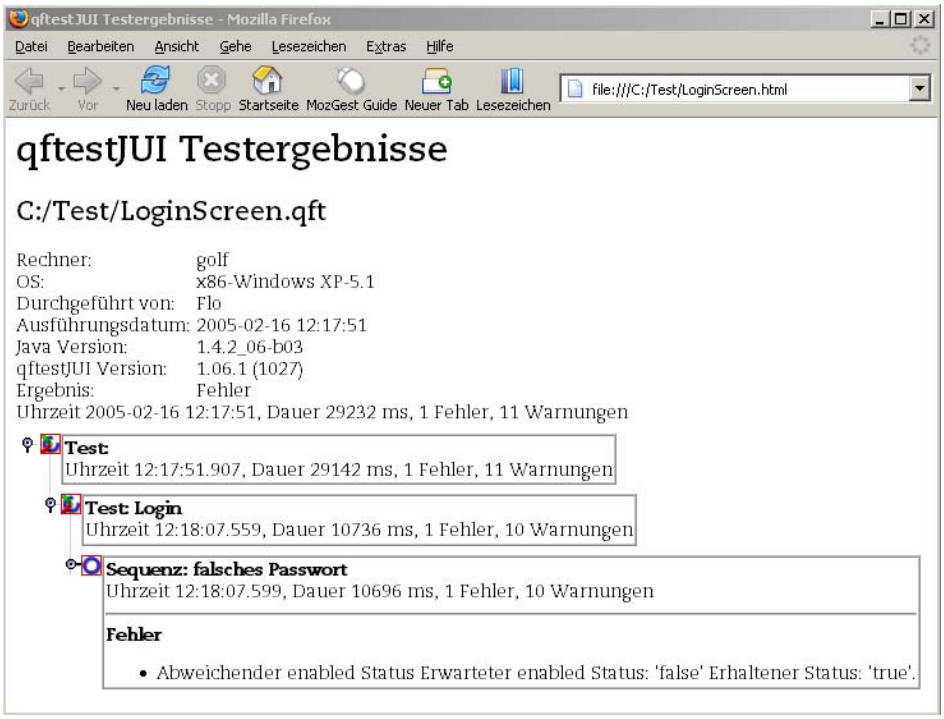


Abbildung 24 – Testergenisse im HTML-Report

Der Report beginnt mit einer Referenz auf die zugehörige Protokolldatei. Es schließt sich ein Teil mit allgemeinen Systeminformationen an. Dann folgen das Testresultat, die Testlaufzeit sowie die Anzahl an aufgetretenen Exceptions, Fehlern und Warnungen. Die sich anschließende Baumdarstellung zeigt relevante Informationen zum Testablauf und Details der Knoten. Hier sind vor allem die Knoten von Interesse, die Fehler enthalten. Auch die genaue Fehlerbeschreibung ist dokumentiert.

Weitere Eigenschaften von qftestJUI

Der Debugger

Wie bei jeder komplexen Entwicklung wird es ab einem gewissen Punkt nötig sein, Probleme in einer Testsuite zu debuggen, die nicht mehr einfach durch Analysieren der Elemente und der Struktur einer Testsuite zu lösen sind. Zu diesem Zweck verfügt qftestJUI über einen intuitiven Debugger.

Skripting

Es ist einer der großen Vorteile von qftestJUI, dass komplexe Tests erstellt werden können, ohne eine einzige Zeile Code zu schreiben. Allerdings gibt es Dinge, die sich mit einem GUI alleine nicht bewerkstelligen lassen. Für ein Programm, das Daten in eine Datenbank schreibt, könnte es z.B. sinnvoll sein, zu überprüfen, ob die Daten korrekt geschrieben wurden. Oder man könnte Testdaten aus einer Datenbank oder einer Datei lesen und mit diesen einen Test ausführen. All das und mehr wird mit Hilfe der mächtigen Skriptsprache Jython ermöglicht. Jython ist eine Implementierung der Programmiersprache Python in Java, die nahtlos in qftestJUI integriert ist.

Beim Skripting ist die Herangehensweise von qftestJUI genau umgekehrt zu der anderer GUI Testprogramme. Anstatt den gesamten Test durch ein Skript zu steuern, bettet qftestJUI kleine Skripte in die Testsuite ein.

Prozeduren

Bei automatischen GUI Tests müssen immer wieder ähnliche Abläufe mit kleinen Variationen ausgeführt werden. Um die Komplexität einer Testsuite zu reduzieren und besser auf Änderungen reagieren zu können, ist es sinnvoll, wiederverwendbare Teile in Prozeduren auszulagern. Eine Prozedur kann von jedem anderen Punkt der Testsuite und sogar aus anderen Testsuiten heraus aufgerufen werden. Dabei können Parameter in Form von Variablen Definitionen übergeben werden.

Variablen

Variablen sind von zentraler Bedeutung, wenn es darum geht, die Wiederverwendbarkeit einer Testsuite zu steigern. Zum Einsatz kommen sie vor allem beim Aufruf von Prozeduren. Variablen sind aber auch in vielen anderen Situationen hilfreich.

Abfragen

Das Automatisieren eines Tests wird dann schwierig, wenn sich das *SUT* dynamisch verhält, d.h. wenn sich Dinge bei jedem Programmlauf anders verhalten. Das ist z.B. der Fall, wenn Ids für Datenbankfelder generiert werden, die aktuelle Uhrzeit Eingang in Datensätze findet, etc. Neben seinen Kontrollstrukturen bietet qftestJUI die Möglichkeit, Daten aus Komponenten der Oberfläche des *SUT* auszulesen und in der Testsuite weiter zu verwenden, z.B. als Index für ein Element, oder als Vergleichswert für einen Check.

Ablaufsteuerung

Neben den normalen Sequenzen verfügt qftestJUI über eine Reihe von speziellen Kontrollstrukturen, die der Steuerung des Ablaufs eines Tests dienen. Diese orientieren sich an den gängigen Kontrollstrukturen, wie sie auch in Java vorhanden sind. Außer Schleifen und bedingter Ausführung erbt qftestJUI von Java auch die Art der Fehlerbehandlung mittels Exceptions, die im Fehlerfall bei der Ausführung eines Knotens ”geworfen” werden. Eine Exception bricht die Ausführung eines Testlaufs schrittweise von unten nach oben ab, so lange bis sie durch ein Catch abgefangen wird, oder bis der gesamte Test beendet ist.

Unterelemente komplexer Komponenten

Für die komplexen Swing Komponenten JTable, JTree, JList etc. bietet qftestJUI besondere Unterstützung. So kann z.B. der Inhalt von Tabellenzellen ausgelesen werden, oder ein Mausklick relativ zum Ast eines Baumes angegeben werden, der anhand seines Textes identifiziert wird, unabhängig von seiner Position im Baum.

Erfahrung mit qftestJUI

Das Capture/Replay Testwerkzeug qftestJUI von der Firma Quality First Software (QFS) bietet ein sehr umfangreiches Testpaket mit großem Funktionsumfang zur Testautomatisierung von GUI-Systemtests für Swing basierte Applikationen an. Ein großer Vorteil von qftestJUI ist die vollständige Implementierung in Java und gehört somit zu den wenigen Testtools, die platttformübergreifende Test unter Windows oder diversen Unix Varianten durchführen können. Neben den aktuellen Java Versionen von Sun, IBM und anderen Herstellern, werden durchgängig auch ältere Versionen ab JDK 1.1 unterstützt. Mit qftestJUI hat der Anwender vollen Zugriff auf die Struktur des zu testenden Systems, inklusive komplexer Swing Komponenten wie Bäume und Tabellen. Es klinkt sich direkt in die zu testende Anwendung ein und ist dadurch nicht auf die vom Betriebssystem bereitgestellten Schnittstellen beschränkt. Die aus der Oberfläche gewonnenen Informationen werden mit solchen aus dem Datenmodell der Applikation kombiniert. Dadurch sind Test toleranter gegenüber den unvermeidlichen Änderungen im Verlauf des Software-Entwicklungsprozesses.

Der Schwerpunkt mit qftestJUI liegt bei der Durchführung von Regressionstests. Mit Regressionstests soll durch wiederholte Kontrolle nach kleinen Änderungen alter Funktionen sichergestellt werden, ob die bekannten Fehler wirklich behoben wurden und ob durch die Fehlerbehebung keine neuen Fehler hinzugekommen sind. Folglich muss sich qftestJUI an ein möglicherweise geändertes GUI anpassen können. Im Gegensatz zu Unittests, die zwar unverzichtbar sind, jedoch immer nur kleine Teile des Gesamtsystems isoliert betrachten, testen GUI basierte Tests die Anwendung als Ganzes. Dadurch können viele verschiedene Arten von Fehlern aufgedeckt werden, nicht nur solche an der Oberfläche selbst.

Die intuitive Objekt-basierte Benutzeroberfläche von qftestJUI erleichtert das Erstellen von einfachen Test. Mit „einem Mausklick“ lässt sich die Aufnahme eines manuellen Tests starten. Anschließend kann das aufgezeichnetes Testskript beliebig oft wiederholt und auf geänderten GUIs angewendet werden. Das Hauptaugenmerk von qftestJUI ist nicht das Testen der zugrunde liegenden Swing Implementierung der Fenster und Komponenten. Durch die Konstellation von Events und Checks können mittels einfacher Tastenkombinationen in Handumdrehen Testsequenzen zur Überprüfung der Logik der Anwendung erstellt werden. Mit Hilfe einer Vielzahl von Kontrollelementen wie Bedingungen und Schleifen können diese zu größeren Bausteinen zusammengesetzt werden. Komplexe Testfälle lassen mit Prozeduren, Packages und Bibliotheken übersichtlich strukturieren und Testumgebungen modular aufbauen. Anstatt der fest verdrahteten Texteingaben können auch variable Werte verwendet und die aufgenommene Sequenz in eine Prozedur umgewandelt werden. Auf diesem Weg lassen sich schnell datengetriebene Tests erstellen, die immer wieder dieselbe Sequenz durchlaufen, aber jedes mal mit anderen Daten. Diese können z. B. aus einer Datei oder einer Datenbank gelesen werden. Ein integrierter Debugger hilft bei der Fehlerdiagnose. Durch die schrittweise Ausführung von Tests lassen sich die Ursachen von Problemen genau lokalisieren.

Bei einem fehlgeschlagenen Testlauf ist ein Protokoll ein entscheidendes Hilfsmittel die Fehlerursache herauszufinden. Das Protokoll von qftestJUI enthält für die Fehleranalyse

Zeitstempel, optionale Anmerkungen, Informationen über Variablenexpansion, verschiedene Arten von Meldungen und insbesondere Fehlerinformationen. Aus dem Protokoll können alle relevanten Informationen über einen Testlauf zu einem HTML-/XML-Testreport generiert werden. Mit Hilfe selbst geschriebener XSLT Stylesheets kann der Anwender eigene Reports transformieren. Offenheit und Flexibilität ist dank der Nutzung von XML als zentrales Dateiformat von qftestJUI gewährleistet. Testsuites, Protokolle und Reports basieren auf diesem Industriestandard und sind dank des Textformats hervorragend für die Verwaltung mit gängigen Versionskontrollsystemen geeignet. Fortgeschritten Anwender erhalten über die ausgereifte Skriptesprache Python unmittelbaren Zugriff aus das gesamte Java API der Testanwendung.

Im Vergleich zu dem OpenSource-Produkt JFCUnit ist der Einarbeitungsaufwand durch den Funktionsumfang relativ größer. Aber durch die sehr ausführliche Dokumentation in HTML oder PDF, die mitgeliefert wird, fällt der Einstieg erheblich leichter. Mithilfe des Tutorials werden die verschiedenen Komponenten und Funktionen vorgestellt und vermittelt innerhalb kürzester Zeit die ersten notwendigen Schritten zur Erstellung seiner eigener Testsuite. Das Benutzerhandbuch beschreibt die grundlegenden Funktionen von qftestJUI und erweitert durch seinen gut strukturierten und verständlichen Referenzteil die Dokumentation. Schnell und informativ wird in die Grundlagen von qftestJUI einführt.

Das Testwerkzeug hat bei der Einführung seinen bestimmten Preis. Das kommerzielle Angebot richtet sich aber ausschließlich an Firmenkunden. Während meiner Arbeit konnte ich durch wiederholtes Anfordern der vier Wochen gültige Evaluationslizenz alle Funktionen von qftestJUI frei schalten und nutzen. Bei Problemen und Fragen stand der qftestJUI-Support sieben Tage in der Woche zur Verfügung. Innerhalb kürzester Zeit bekam ich hilfreiche Rückantwort.

Seit der Markteinführung konnte QFS bereits mehr als 100 Kunden aus allen Teilen der Welt für sein Produkt qftestJUI gewinnen. In der näheren Zukunft plant QFS Erweiterungen zu qftestJUI herauszubringen. Es ist das Ziel auf Basis ihres Frameworks Werkzeuge aus dem gesamten Bereich der Software-Qualitätssicherung zu liefern.

Die Kombination von Unittest und funktionalen Testen mit qftestJUI ist für das Unternehmen ein weiteres Bestreben, indem die Applikation als Ganzes durch qftestJUI getrieben und so eine realistische Umgebung für komplexe Unittests aufgebaut wird. Dies kann zu äußerst mächtigen Tests führen, die weit darüber hinaus gehen, was mit Unit- oder GUI Tests alleine möglich ist. Durch der Jython Schnittstelle ist die Integration von Unittests in qftestJUI bereits jetzt erreichbar. Für QFS ist aber das Thema so interessant, dass sie hier eine noch wesentlich weitergehende Unterstützung anstreben.

6 Zusammenfassung

Testautomatisierung ist die Verwendung eines Testwerkzeugs, um Tests zu erleichtern. Es ist ein Hilfsmittel, das eine oder mehrere Testaktivitäten unterstützt. Auf der einen Seite verlangt Testautomatisierung einen strukturierten Testprozess. Auf der anderen Seite hilft Testautomatisierung bei der Einrichtung eines strukturierten Testprozesses. Testwerkzeuge haben sowohl Vor- als auch Nachteile. Die Auswahl des richtigen Werkzeuge ist die Grundlage für den Erfolg der Testautomatisierung. Der Prozess der Werkzeug-Auswahl muss mit großer Umsicht erfolgen. Testwerkzeuge können in jeder Phase des Testprozesses zum Einsatz kommen. Kommerzielle Testwerkzeuge für beinahe jede Testaktivität sind wie im Kapitel 4 ersichtlich am Markt erhältlich.

Der Aufwand zur Automatisierung hat sich besonders bewährt und bezahlt gemacht, bei der Durchführung wiederholender Aufgaben und deren anschließenden Auswertung. Dies führt zu höheren Zuverlässigkeit der Testaktivität und zu höherer Zufriedenheit im Testteam, was wiederum zu höherer Produktivität führt. Es lässt sich darüber hinaus sicherstellen, dass die Tests regelmäßig, vollständig, konsequent und regelmäßig durchgeführt werden. Fehler können frühzeitig erkannt und beseitigt werden. Die Testautomatisierung ist vor allem bei den heutigen Testmethoden abhängig von der Flexibilität, Wartbarkeit und der Lebensdauer des zu testenden Produkt. Der Kauf eines kommerziellen Werkzeugs mit einem großem Umfang von Funktionalität ist nicht billig. Für kleinere Anwendungen oder Projekte sind die kostengünstigen und einfach zu Hand habende OpenSource Testtools geeignet. Die Gründe zur Automatisierung hier nochmals kurz zusammengefasst:

- Herabsetzung des manuellen Aufwand,
- Herabsetzen der Testkosten,
- Erweiterung der Testüberdeckung,
- Reproduzierbarkeit und
- Qualitätssicherung.

Es ist ungewiss, wohin sich die Testautomatisierung weiterentwickeln wird. In den letzten Jahren hat sich in diesem Bereich einiges getan und es wird in der Qualitätssicherung ein wichtiger Bestandteil bleiben. Wer in diese Branche einsteigt, hat gute Aussichten auf Jobangebote. Jede Woche erfährt man aus den Zeitungen von fehlerhaften Softwareprodukten oder erneuten Rückholaktionen mancher bekannten Autohersteller und ihrer Zulieferer. Der Aufwand für das Testen von Softwareprodukten nimmt schon 40 bis 50% der Gesamtzeit und der Gesamtkosten ein. Durch die Testautomatisierung erhoffen sich alle eine bessere Qualitätssicherung und die Reduktion der Kosten.

Es gibt eine große Anzahl von Testwerkzeugen, welche die Testdurchführung automatisieren. Jedoch gibt es kaum Werkzeuge, die den Tester bei der Testfallerstellung unterstützen. Dafür müssten automatisch verwertbare Daten vorliegen. In dieser Richtung gibt es bisher nur relativ wenige Forschungsansätze. Es gibt auch noch keine bekannten Bestrebungen ein kommerzielles Testtool zu entwickeln. Im Moment bedeutet die Erstellung von Testfällen erhöhte Aufwand eine

Investition. Der Erfolg und die Wirkung des Einsatzes sind jedoch nicht immer offensichtlich, weshalb grade dieser Punkt ein Argument gegen die Verwendung der Testautomatisierung ist. Deshalb sollte gerade die Aufgabe der Testfallerstellung und der Testdatengewinnung im Fokus der zukünftigen Entwicklungen von Testautomatisierungswerkzeugen liegen. Kann in diesem Bereich ebenfalls eine Automatisierung verwirklicht werden, so amortisieren sich die Kosten der Testautomatisierung schneller. Der Einsatz bei kurzlebigen Softwareprojekten könnte somit ebenfalls rentabel werden.

Umsetzung in der Praxis

Es stellt sich die Frage, welches der beiden evaluierten Produkte kann in den Labor-Projekten während des Studiums integriert werden. Abgesehen von den gegebenen Erwerbskosten ist eines der Hauptkriterien der Einarbeitungsaufwand, der vom Funktionsumfang und der Dokumentation abhängig ist. Des weiteren gehört der Lernfaktor in Betracht gezogen. Die Fachhochschule Esslingen setzt im Studiengang Softwaretechnik und Medieninformatik verstärkt auf den objektorientierten Ansatz durch die Programmiersprachen C++ und Java.

Beide Testframeworks sind in Java implementiert und fertigen ihre Tests in derselben Programmiersprachen, wodurch man vollen Zugriff auf die gesamte Anwendung erhält. Zusätzliche Verifikationen und Einstellungen können vorgenommen werden.

Das Testframework JUnit hat sich in den letzten Jahren als der Standard für automatische Tests in Java herausgestellt und unterstützt Extreme Programming. JUnit wird in unzähligen Projekten erfolgreich benutzt und viele Entwicklungsumgebungen haben eine direkte JUnit-Unterstützung. Aufgrund der breiten Akzeptanz von JUnit und der Erweiterung JFCUnit, sollte über eine Integration in einigen Java-Projekten nachgedacht werden. Die Implementierung der Testfälle müssen größtenteils selbst programmiert werden und können durch die XML-Erweiterung mit der Record-Funktion deutlich vereinfacht werden. Durch die Nutzung von JFCUnit geht das Schreiben von Tests projektbegleitend in die normale Arbeitsweise über und mit der Übung verringert sich auch der Aufwand für die Erstellung der Testfälle. Somit könnten auch direkt mit der selben Entwicklungsumgebung wie sie für die Anwendungsentwicklung benutzt wird, Tests geschrieben werden. Mit JFCUnit lassen sich einfache funktionale Tests abbilden. Trotz des zur Zeit geringen Funktionsumfanges kann JFCUnit teilweise mit geringen Erweiterungen zum Test von kleinen Komponenten sinnvoll eingesetzt werden. JFCUnit hat im Gegensatz zu qfestJUI keine so ausgereifte und ausführliche Dokumentation. Die beigefügten Beispiele und das Help-Forum sind aber bei mancher Problemstellung eine große Hilfe. Es gibt auch schon zahlreiche IT-Bücher, die sich mit Testautomatisierung und dem Testframework JUnit und JFCUnit auseinander setzen. Zum besseren Verständnis wäre es zum Vorteil sich mit JUnit zuerst auseinander zu setzen.

Die Benutzung vom Capture&Replay-Programm qftestJUI hilft vor allem die Testzeiten bei Regressionstests zu verkürzen. Die Erstellung einer Testsuite und die Einarbeitung in die ersten notwendigen Schritte wird durch das Tutorial in kürzester Zeit unterstützt. Die manuelle

Anpassung der aufgenommenen Testskripte müssen bei qftestJUI auch vorgenommen werden, da zusätzliche Prüfungen und Fehlerbehandlungen in die Skripte integriert werden müssen. Gerade dieser Umstand bringt einen erheblichen Mehraufwand bei Testtools im Vergleich zum manuellen Testen mit sich. Der Vorteil von qftestJUI gegenüber JFCUnit ist die gute übersichtliche intuitive Benutzeroberfläche, die eine Testsuite überschaubar macht und eine umfangreiche Funktionalität anbietet.

Die JUnit-Familie gehört zu den wenigen Testwerkzeugen die White-Box-Testverfahren unterstützen. JUnit ist ein Unittest-Tool, das hauptsächlich dem Testen von isolierten Programmeinheiten wie einzelne Methoden, Klassen oder Modulen dient. Es setzt an den öffentlichen Schnittstellen dieser Einheiten an. Trotzdem ist es ein White-Box-Testverfahren, weil es die Möglichkeit und auch die Bereitschaft voraussetzt, bestehenden Code zu ändern, insbesondere um Schnittstellen zu ändern bzw. diese überhaupt erst einzuführen um eine bestimmte Testbarkeit zu erreichen. Capture/Replay-Werkzeuge unterstützen nur das Black-Box-Testverfahren, welches die verborgene Funktionalität der getesteten Anwendung gar nicht bzw. indirekt testen kann.

White-Box-Test werden von den gleichen Programmierern entwickelt wie das zu testende System selbst. Der den Test entwickelnde Programmierer hat also Kenntnisse über das zu testende System. Kurzfristig sind sie kostengünstiger, zeigen aber in der Praxis allerdings eine äußerst hohe Durchlässigkeit für Fehler. Black-Box-Tests werden stattdessen von Programmierern und Testern entwickelt, die keine Kenntnisse über den inneren Aufbau des zu testenden Systems haben. In der Praxis werden Black-Box-Tests meist von speziellen Test-Abteilungen oder Test-Teams entwickelt. Black-Box-Tests decken besonders viele Fehler auf, erweisen sich in der Praxis aber zum einen als organisatorisch aufwändig und zum anderen manchmal auch als sozial unverträglich wegen eventueller Spannungen zwischen den Test- und den Entwicklungsabteilungen.

Automatisierte Usability-Tests

Eine gute Benutzungsoberfläche muss nicht nur eindeutig zu überprüfende Eigenschaften aufweisen, sondern ist gleichzeitig unscharfen Bewertungskriterien unterworfen wie Ergonomie, intuitive Benutzung und Ästhetik. Die Gebrauchstauglichkeit eines Produktes – Usability- wird in der ISO 9241 Teil 10 & 11 definiert. Diese europäische Norm trägt im Deutschen den Titel „Ergonomische Anforderungen für Bürotätigkeit mit Bildschirmgeräten“. Entscheidende Faktoren für die Usability eines Produktes sind zusammengefasst die Effizienz, die Effektivität und die Zufriedenheit. Im Klartext geht es darum, gesundheitliche Schäden beim Arbeiten am Bildschirm zu vermeiden und dem Benutzer seine Aufgaben zu erleichtern. Zu den Normen gibt es noch etliche Guidelines bezüglich User Interface Design. Durchführung von automatischen Usabilitytests könnte, nach Aussagen einiger Fachleute im Bereich Testautomatisierung, beim derzeitigen Stand der Testtools und Strategien die Einhaltung dieser vorgegebenen Richtlinien nur statisch überprüfen. Vieles im Bereich Usability ist subjektiv und die Definition in ihrer Formulierung sehr abstrakt. Der Nutzer reagiert meist anders, als sich der Entwickler das

gedacht hat. In der Softwareentwicklung sind nach meinen Recherchen und Befragungen keine entsprechende Programme vorhanden. Es gibt Programme, wie zum Beispiel Morae, die es ermöglichen, wichtige Erkenntnisse über die Erfahrungen eines Benutzers durch die Erfassung mittels eines Recorders im Hintergrund der kompletten Interaktion zwischen Menschen und Computer zu gewinnen [TSmith]. Es werden die Benutzer- und Systemdaten synchronisiert für Usability-Analysen von Software, Websites, Intranets und e-Business-Anwendungen aufgenommen.

Im WebDesign Bereich gibt es automatische Testtools, die nur die Aspekte der Barrierefreien Informationstechnik-Verordnung (BITV) oder der Web Accessibility Guidelines (WAI) 1.0 überprüfen. Zum Beispiel bietet das World-Wide-Web-Consortium (W3C) Online-Hilfsmittel zum Testen einer Seite hinsichtlich der Einhaltung der W3C-Standards an, sowohl hinsichtlich der Konformität von (X)HTML-Standards als auch zur Einhaltung der CSS-Standards. Die meisten Bedingungen, die eine barrierefreie Seite erfüllen muss, sind jedoch nicht oder nur teilweise automatisch überprüfbar. Das Aktionsbündnis für barrierefreie Informationstechnik (AbI) arbeitet zur Zeit an Testempfehlungen, die es ermöglichen die wenigsten automatisch prüfbaren Punkte auf einer gemeinsamen Grundlage für Tests zu vereinheitlichen und nachvollziehbar durchführen zu können.

Die Erstellung einer Website nach den anerkannten Kriterien der Web Usability bewirkt Vorteile für den Nutzer und den Anbieter von Informationen, Dienstleistungen oder Produkten gleichermaßen. Welchen Wert hat die Usability? [TSmith]

Erhöhte Produktivität

“Ein durchschnittliches mittelgroßes Unternehmen könnte 5 Millionen US $ pro Jahr durch die gesteigerte Produktivität der Mitarbeiter gewinnen, wenn das Intranet-Design auf das Niveau gebracht würde, welches dem obersten Viertels einer unternehmensübergreifenden Studie über Intranet-Usability entspricht. Die Rendite im vergleich zum Investition? Eintausend Prozent oder mehr.&” (Nielsen, 2002)

Allgemeiner Wert

“Eine Studie schätzte, dass Verbesserungen bei der Kundenerfahrung die Zahl der Käufer um 40 % und den Auftragsumfang um 10% erhöht.&” (Creative Good, 2000)

Gesteigerte Verkaufszahlen

“Die durchschnittliche UI hat etwa 40 Mängel. Werden die 20 einfachsten davon korrigiert, steigert sich die Usability um durchschnittlich 50%. Der große gewinn entsteht jedoch, wenn Usability von Anfang an mit einbezogen wird. Dies kann zu einer Effektivitätssteigerung von über 700% führen.&.” (Landaer, 1995)

Schlechte nachvollziehbare Strukturen, missverständliche Navigationselemente oder technische Unzulänglichkeiten lassen den Kauf von Softwareprodukten und dem Besuch im Netz im Misserfolg enden.

7 Literaturverzeichnis

[Bin99] Robert V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools. Reading, MA : Addison-Wesley, 1999

[ct03] Dr. Andreas Elting, Dr. Walter Huber, Schnellverfahren – Mit Extreme Programming immer im Plan?, c't 03/2001

[ct04] c't-Newsticker "Unglückliche Geldvermehrung" , bb/25.05.04; http://www.heise.de/newsticker/meldung/47666

[Cem97] Cem Kaner, Improving the Maintainability of Automated Test Suites, Paper Presented at Quality Week 1997

[Dij72] Edsger W. Dijkstra: Notes on Structured Programming. In: O. J. Dahl, (Hrsg.) ; E. W. Dijkstra (Hrsg.) ; C. A. R.Hoare (Hrsg.): Structured Programming. London : Academic Press, 1972

[Ginz00] Prof.Dr. Martin Glinz, Universität Zürivh, KV Software Engineering

[Glat04] Christian Glatschke, Automatisierung dse Tests von Java-Swing-GUIs, JavaSpektrum, 06/2004

[Gos98] Interview mit James Gosling vom 24. März 1998 http://java.sun.com/javaone/javaone98/keynotes/gosling/transcript_gosling.html

[IESE04] Ove Armbrust, Michael Ochs, Björn Snoek, Stand der Praxis von Software-Test und deren Automatisierung – Fraunhofer Institut Experimentelles Software Engineering

[ISO9126] International Organization for Standardization, www.iso.org

[ix03] Thomas J. Heistracher, Michael Thalmeier, Tobias Witek, Stefan Zeppetzauer , GUI-Testwerkzeuge im Vergleich, ix Magazine 10/2003

[Java02] Stefan Middendorf, Reiner Singer, Jörn Heid, Java™ - Programmierhandbuch und Referenz für die Java™-2-Plattform, Standard Edition, 3. Auflage 2002 http://www.dpunkt.de/java/Programmieren_mit_Java/Oberflaechenprogrammierung/91.html

[jfc04] http://jfcunit.sourceforge.net

[Kirs03] Christian Kirsch, Test it yourself - Frei verfügbarer GUI-Tester, ix Magazine 10/2003

[Krueg03] Guido Krüger, Handbuch der Java-Programmierung 3.Auflage, Kaptiel 35.1.2 Eigenschaften von Swing, Addison Wesley, Version 3.0.1 http://dufo.tugraz.at/mirror/hjp3/index_c.html

[LE05] Olga Kolov, Frank Müller; TDD in der Praxis, Linux Enterprise 01/2005, http://www.linuxenterprise.de/itr/online_artikel/psecom,id,652,nodeid,9.html

[McC93] S. McConnell, Code Complete., Microsoft Press, 1993

[Mie03] Andreas Mieth, Projekterfahrung mit Testautomatisierung; ix Magazine 10/2003

[Muen98] Westfälische Wilhelms-Universität Münster, Wirtschaftsinformatik und Informationsmanagement; http://www-wi.uni-muenster.de/is/vorlesungen/SE-I/Winter1998/SW-Qualit%C3%A4t-4.pdf.

[Myers79] Glenford J. Myers, Methodisches Testen von Programmen, Oldenbourg, 1979

[Pol00] Martin Pol, Tim Koomen, Andreas Spillner: "Management und Optimierung des Testprozesses: ein praktischer Leitfaden für Testen von Software, mit TPI und TMap", dpunkt.verlag, 2000,

[Post92] R.M. Poston; M.P.Sexton: "Evaluating and selecting testing tools", IEEE Software, Volume: 9 Issue: 3 , May 1992, Pages: 33 -42.

[QBench] Homepage QBench, Stand der Technik, http://www.qbench.de/QBench/CMS/projektbeschreibung/part2_de

[Ri97] E. H. Riedemann, Testmethoden für sequentielle und nebenläufige Software-Systeme. Teubner Verlag, Stuttgart, 1997

[Stein02] Michael Stein 14.12.2002 ; http://www.raumfahrer.net/raumfahrt/raketen/ariane5_fehlstart.shtml

[Stein97] Uwe Steinmüller, Let it swing, www.heise.de/ix/artikel/1997/10/132/

[SW02] H.M. Sneed; M. Winter: Testen objekorientierter Softwar*e*. Hanser Verlag, 2002

[Spg00] Elfriede Dustin, Jeff Rashka, John Paul, Software automatisch testen – Verfahren, Handhabung und Leistung, Springer, 2000

[TSmith] http://de.techsmith.com/products/morae/usability.asp

[Ze02] A. Zeller, Software-Test: Strukturtest. Universität des Saarlandes, Softwaretechnik II, SS 2002, http://www.st.cs.uni-sb.de/edu/se2/

Weitere Quellen wurden von mir beim Erstellen dieser Studienarbeit verwendet:

Ernest Wallmüller, Software-Qualitätssicherung in der Praxis, Carl Hanser Verlag, 1990

Karol Frühauf, Jochen Ludewig, Helmut Sandmayr, Software-Prüfung – Eine Anleitung zum Test und zur Inspektion, 2.Auflage Teubner Verlag & vdf Hochschulverlag AG an der ETH Zürich,1995

Christian Ullenboom, Java ist auch eine Insel - Programmieren für die Java 2-Plattform in der Version 1.4, 3.Auflage Galileo Computing, 2003
http://www.galileocomputing.de/openbook/javainsel3/

Guido Krüger, Handbuch der Java-Programmierung 3.Auflage, Addison Wesley, Version 3.0.1
http://dufo.tugraz.at/mirror/hjp3/index_c.html

Joachim Goll, Cornelia Weiß, Frank Müller, Java als erste Programmiersprache – Vom Einsteiger zum Profi, 3.Auflage Teubner Verlag, 2001

Auflistung von weiteren Testwerkzeugen:

http://www.imbus.de/testlabor/tool-list.html
http://opensourcetesting.org/functional.php
http://www.qalinks.com/Tools/Automated_Testing/
http://www.testingfaqs.org/
http://www.softwareqatest.com/qatweb1.html#JAVA

Usability:
http://www.sozialnetz-hessen.de/ca/p h/het/
http://usability.is.uni-sb.de//werkzeuge/wu_index.php
http://wob11.de/loesungen/tippstools.html

8 Glossar

Akzeptanztests	Test des Kunden, der das Softwaresystem in Auftrag gegeben hat. Es wird an einem laufenden System geprüft, ob es den im Auftrag beschriebenen Anforderungen erfüllt. Werden auch während der Entwicklung zu geschrieben, um zu sehen wie weit der Forschritt ist und welche Eigenschaften noch implementiert werden müssen. Auch Betatest genannt.
API	Application Programming Interface
AWT	Abstract Window Toolkit
Container	sind Oberflächenelemente, die Komponenten enthalten und gruppieren können. Sie stellen jedoch auch selbst wieder Komponenten dar und können somit auch in anderen Container enthalten sein. Ein Beispiel für einen Container ist ein Fenster.
Entwicklungstest	Test durch den Entwickler während der Realisierung.
Fehlerkritikalität	siehe Kritikalität
Framework	Ein Framework ist eine abstrakte Sammlung von Klassen, Interfaces und Mustern mit dem Ziel eine Gruppe von Problemen durch eine flexible und erweiterbare Architektur zu lösen
Funktionstest	Test auf fachliche Korrektheit, das Zusammenwirken mit Nachbarsystemen und Sicherheit. GUI Capture/Replay Testtools werden vor allem für funktionale Tests eingesetzt. Sie können das System nur als ganzes bedienen.
GUI	Graphical User Interface
IESE	Institut Experimentelles Software Engineering
JDK	Java Development Kit
JFC	Java Foundation Classes
JRE	Java Runtime Environment
JVM	Java Virtuelle Maschine
Komponenten	sind Oberflächenelemente, die in eine Benutzerschnittstele eingefügt werden. Beispiel für Komponenten sind Buttons (Schaltflächen), Labels (Beschriftungen) oder Menüs.
Kritikalität	Zustand eines Kernreaktors, bei dem die zur fortgesetzten Kernspaltung notwendige Kettenreaktion nicht mehr selbst erlischt. „bedrohlich“
Lasttest	Prüfen die Antwortzeiten unter Hochlast und Überlastungsverhalten.
Layout-Manager	sind selbst nicht an der Oberfläche sichtbar – legen fest, wie die Komponenten eines Container angeordnet werden.
MVC	Model-View-Control-Konzept
native	betriebssystemeigene
Performancetest	Untersuchung von Zeitverhalten und Ressourcen-Verbrauch.
Regressionstest	Wiederholte Kontrolle nach kleinen Änderungen alter Funktionen soll sicherstellen, ob die bekannten Fehler wirklich behoben wurden und ob durch die Fehlerbehebung keine neuen Fehler hinzugekommen sind.

Robustheitstest	Test auf Fehlertoleranz, Wiederherstellbarkeit des Normalzustands nach Fehlern und Reife des Systems im Dauerbetrieb.
„Swing“	Als 1997 in San Francisco auf der JavaOne die neuen Komponenten vorgestellt wurden, entschied sich Georges Saab, ein Mitglied des JFC-Teams, für Musik parallel zur Präsentation. Dies war gerade Swing-Musik, denn der Entwickler glaubte, dass sie wieder in Mode käme. So wurden auch die neuen grafischen Elemente im Paket mit dem Namen Swing abgelegt. Obwohl der Name offiziell dem Kürzel JFC weichen musste, war er doch so populär, dass er weiter bestehen blieb.
SUT	System Under Test
XML	Extended Markup Language