FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Entwurf eines Kriterienkatalogs und Evaluierung von State-of-the-Art Open-Source Testautomatisierungswerkzeugen in Bezug auf Integrationstauglichkeit, Funktionalität und Flexibilität in realen, komplexen Testprozessen

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Christina Zrelski
Matrikelnummer 0725816

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Roland Breiteneder

Wien, 15. April 2014

(Unterschrift Verfasser/In) (Unterschrift Betreuung)

Entwurf eines Kriterienkatalogs und Evaluierung von State-of-the-Art Open-Source Testautomatisierungswerkzeugen in Bezug auf Integrationstauglichkeit, Funktionalität und Flexibilität in realen, komplexen Testprozessen

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Christina Zrelski
Matrikelnummer 0725816

ausgeführt am
Institut für Rechnergestützte Automation
Forschungsgruppe Industrial Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Roland Breiteneder

Wien, 15. April 2014

Eidesstattliche Erklärung

Christina Zrelski
Herzgasse 60/7/9, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

(Ort, Datum) (Unterschrift Verfasser/In)

Kurzfassung

Die Disziplin des Software-Testens hat in den vergangenen 30 Jahren stetig an Bedeutung gewonnen und ist heute wichtiger Bestandteil jedes Softwareprojekts. Aus Effizienz- und Kostengründen wird dabei häufig der Einsatz von Testwerkzeugen zur (Teil-)Automatisierung wiederkehrender und zeitaufwendiger Tätigkeiten angestrebt. Die Sicherung der Qualität grafischer Benutzeroberflächen ist besonders wichtig, da diese eine zentrale Rolle in der Interaktion zwischen einem Benutzer und einem Softwaresystem einnehmen. Die Erstellung von Tests gestaltet sich auf diesem Gebiet allerdings komplex und Ansätze zur Automatisierung von Testabläufen befinden sich oft erst im Anfangsstadium. Für das automatisierte Testen der Benutzeroberfläche javabasierter Desktop-Applikationen fehlen Informationen über die Verfügbarkeit, Praxistauglichkeit und den Reifegrad bestehender Werkzeuge.

Diese Arbeit reduziert das angesprochene Wissensdefizit, indem sie sechs am Markt verfügbare, als Open-Source lizensierte Werkzeuge zum automatisierten Testen von Benutzeroberflächen javabasierter Rich-Client-Applikationen untersucht und vergleicht. Grundsätzlich gilt: Damit die Praxistauglichkeit bestehender Werkzeuge beurteilt werden kann, müssen diese anhand eines konkreten Fallbeispiels erprobt werden. Im Rahmen dieser Arbeit wurde hierfür eine bestehende Event-Management-Applikation ausgewählt, die aufgrund ihres Umfangs, ihrer Komplexität sowie der verwendeten Technologien als repräsentativ anzusehen ist. Neben dem Funktionsumfang der Werkzeuge werden insbesondere auch nicht-funktionale Qualitätskriterien wie Integrationsfähigkeit, Robustheit und Wartbarkeit der Testskripts betrachtet.

Zu Beginn werden die theoretischen Grundlagen für die nachfolgende praktische Untersuchung geschaffen, indem das Konzept der grafischen Benutzeroberfläche diskutiert und in die Disziplin des Software-Testens eingeführt wird. Im Anschluss daran wird im Rahmen einer Analysephase ein auf das bestehende Softwareprojekt abgestimmter Kriterienkatalog entworfen, der alle projekt-spezifischen und allgemeinen Anforderungen an ein Testwerkzeug abbildet. Dieser wird als Basis für die Auswahl der sechs näher zu betrachtenden Testwerkzeuge FEST, Marathon, Abbot, Sikuli, Robot Framework sowie WindowTester Pro herangezogen.

Im Hauptteil der Arbeit werden die Ergebnisse der prototypischen Implementierung von jeweils fünf repräsentativen Testfällen präsentiert. Dabei wird die Funktionsweise jedes Werkzeugs am Beispiel eines Testskripts erläutert, die positiven Aspekte sowie die identifizierten Problemstellen werden aufgezeigt und Lösungsansätze für diese vorgestellt. Generell ist erkennbar, dass keines der Werkzeuge frei von Problemen und somit ohne entsprechende Adaption für einen Praxiseinsatz tauglich ist. Die Ergebnisse der in dieser Arbeit durchgeführten Untersuchung sind aufgrund des repräsentativen Charakters des Fallbeispiels auch für unerfahrene Tester oder kleine Unternehmen mit wenig Budget für kommerzielle Werkzeuge von hoher Relevanz und können bei ähnlich aufgebauten Projekten als Leitfaden für die Auswahl eines geeigneten Werkzeugs dienen.

Schlüsselwörter

Softwaretest, GUI-Test, Automatisierung, Java Swing, Qualitätssicherung

Abstract

The software testing discipline has gained increasing importance over the past 30 years and is nowadays a vital part in every software project. To enhance efficiency and simultaneously reduce costs effort is put in the automation of recurring manual software testing tasks. The graphical user interface of a system plays a crucial role in establishing interaction between the system's underlying functions and the user. Despite this fact, testing the graphical user interface is a complex and time-consuming task and automation strategies are still in an initial stage. Only rare information on the availability and maturity level of test automation tools for Java based rich client systems and on their suitability for practical application is available.

This thesis overcomes this deficiency in knowledge by investigating six actively developed test automation tools, which are licensed as open source and are suitable for testing the graphical user interface of Java based desktop applications. To perform this investigation, the application of the selected tools within a representative example project is required. Due to its scope, level of complexity as well as usage of state of the art technologies the rich client of an existing event management system is chosen for this task. Object of study is the functional scope of the tools as well as non-functional quality parameters such as integrability, robustness and maintainability of the test scripts. Additionally, their suitability for practical application on the existing software project is analyzed.

First, background information on the concept of graphical user interfaces as well as the discipline of software testing is provided. In order to select the best candidates, a criteria catalog containing general and project specific requirements for test automation tools is developed and evaluated. Following this, six automation tools, namely FEST, Marathon, Abbot, Sikuli, Robot Framework and WindowTester Pro, are chosen for further investigation.

In the main part of this thesis a feasibility study is conducted in which the functioning of each test automation tool is analyzed by implementing five representative test cases from the existing software project. Furthermore, advantages as well as weaknesses of each tool are discussed and possible improvements are introduced. It is concluded, that none of the tools investigated is ready for practical application without extensive adaption work prior to usage.

The results obtained from this feasibility study are due to the representative character of the chosen example project also highly relevant for inexperienced testers or small companies that cannot afford licenses for commercial test automation tools. They can serve as a decision guideline when introducing test automation in similar projects.

Keywords

Software Testing, GUI-Testing, Automation, Java Swing, Quality Management

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Glossar

Callback - eine Funktion innerhalb eines Softwaresystems, die einem anderen, meist zugrundeliegenden System bekannt ist und von diesem genutzt wird, um das Softwaresystem über das Eintreten einer bestimmten Aktion (z.B. Eingabe des Benutzers) zu informieren [60]. 5

Computer Vision - ein in den 1960er-Jahren entstandener Forschungsbereich, der Maschinen die visuelle Erfassung ihrer Umwelt unter Verwendung von Algorithmen der digitalen Bildverarbeitung zugänglich machen soll [35]. 44

Continuous Integration - ein Ansatz in der Softwareentwicklung, bei dem Entwickler ihren Code mehrmals täglich in ein gemeinsames Repository integrieren. Jeder Commit triggert einen automatischen Build, in dessen Rahmen auch (Regressions-)Tests ausgeführt werden [17]. 23, 29, 40, 46, 47, 87

Debugging - eine Tätigkeit, bei der die Fehlerursachen lokalisiert, analysiert und behoben werden. Dabei kommen häufig Tools (sog. Debugger) zum Einsatz, mit deren Hilfe der Programmcode schrittweise durchlaufen werden kann [27]. 11

Inspektion - eine statische Testtechnik zur Analyse von Projektartefakten, häufig von Designdokumenten oder Programmcode. Es handelt sich dabei um die formalste Form des Reviews. Ziel ist die Überprüfung, ob alle Standards und Richtlinien erfüllt und keine Widersprüche mit anderen Projektdokumenten gegeben sind [27, 74]. 15

Look&Feel - bezeichnet das Erscheinungsbild und die Bedienungseigenschaften einer Benutzeroberfläche [6]. 7–9, 28, 39, 63, 66

Platzhalter - simuliert im Rahmen eines Tests zum Zwecke der Isolation die Funktionalität einer Komponente, die eine Abhängigkeit zum im Rahmen eines Tests stehenden Systemteil besitzt [27]. 12, 13

Review - ein Sammelbegriff für verschiedenste Techniken (wie z.B. Walkthrough, Codeinspektion) zur Analyse, die meist im Rahmen von Meetings oder in Kleingruppen Projektartefakte untersuchen, mit dem Ziel, Abweichungen von der Spezifikation zu identifizieren und verbesserungswürdige Aspekte des Artefakts zu identifizieren [74]. 11, 19, 20

Scrum - ein agiles, iteratives Vorgehensmodell in der Softwareentwicklung. Die Anforderungen an das System werden dabei inkrementiell im Rahmen sogenannter Sprints realisiert. Die in einem Sprint umzusetzenden Funktionen sind im Sprint Backlog verzeichnet [24]. 28

Stakeholder - alle Personen oder Personengruppen, die in ein Softwareprojekt involviert sind (intern sowie extern) und gewisse, oftmals widersprüchliche Interessen in diesem verfolgen. Beispiele für Stakeholder sind: Kunde, Benutzer, Entwickler, Tester [24]. 17, 19

Testtreiber - ersetzen Komponenten eines Systems im Rahmen eines Tests zum Zwecke der Isolation und übernehmen den Aufruf und die Ansteuerung des zu testenden Systemteils. In manchen Fällen stellen Testtreiber zusätzlich Eingabedaten bereit und überwachen die Ausführung des Programms [74]. 12, 13

Walkthrough - die schrittweise Präsentation eines Projektartefakts durch den Autor. Ziel dabei ist, gemeinsames Verständnis für die erstellten Inhalte zu schaffen und dem Autor Feedback und Verbesserungsvorschläge zu geben [74]. 15

Abkürzungen

API Application Programming Interface. 7, 9, 27, 37, 42, 43, 45, 47, 53, 63, 65–67, 79, 84

AWT Abstract Window Toolkit. 7–9, 42

CLI Command Line Interface. 4

CSV Comma-Separated Values. 41, 54, 58, 60

EPL Eclipse Public License. 42, 47

GUI Graphical User Interface. 4–7, 9, 25–27, 30, 31, 33, 37, 39, 40, 42–45, 47, 48, 52, 53, 55–57, 59, 61–63, 67, 69–72, 74–76, 78, 83–86

HTML Hypertext Markup Language. 39, 46, 60, 61, 74

HUI Haptic User Interface. 5

IEEE Institute of Electrical and Electronics Engineers. 10

ISO International Organization for Standardization. 3, 4

ISTQB International Software Testing Qualifications Board. 11, 14, 17

JAR Java Archive. 42, 58, 72, 75

JFC Java Foundation Classes. 6, 7

LGPLv2 GNU Lesser General Public License Version 2.0. 40

MIT Massachusetts Institute of Technology. 43

MVC Model-View-Controller. 8, 9

NUI Natural User Interface. 4

OCR Optical Character Recognition. 84, 87

RCP Rich Client Platform. 47

SWT Standard Widget Toolkit. 42, 47

TDD Test-Driven Development. 12, 27, 42

TSV Tab Separated Values. 46, 72

UML Unified Modeling Language. 48, 51

VUI Voice User Interface. 5

XML Extensible Markup Language. 42, 45, 46

1 Einleitung

1.1 Problemstellung und Motivation

Der Gedanke der Automatisierung von Tätigkeiten und Abläufen zur Steigerung von Produktqualität und Effizienz ist in technischen Disziplinen keineswegs neuartig. Auch im Bereich des Software-Testens war das Streben nach Automatisierung ein wichtiger Wegbereiter für seinen heutigen Stellenwert und hat wesentlich zum derzeitigen Stand der Testtechnologie beigetragen. Der Einsatz von Werkzeugen zur Unterstützung und teilweisen Automatisierung verschiedenster Aufgabenbereiche des Software-Testens wird aufgrund der Größe und Komplexität heutiger Softwareprojekte immer bedeutsamer [71].

Gleichzeitig ist die Verfügbarkeit von qualitativ hochwertigen Open-Source-Software-Produkten in den letzten 15 Jahren kontinuierlich gewachsen [43]. Software-Entwickler und Tester sind mit einer enormen Produktvielfalt konfrontiert und nicht immer liegen Produktempfehlungen für konkrete Problemstellungen und Einsatzzwecke vor.

Während sich für das Testen von isolierten Komponenten oder Weboberflächen bereits sehr viele Frameworks (z.B. JUnit, TestNG, Selenium) etabliert haben [5], existiert verhältnismäßig wenig Wissen über Werkzeuge zum automatisierten Testen der Benutzeroberfläche von Rich-Client-Applikationen. Insbesondere fehlen Informationen darüber, welche als Open-Source lizensierten Testtools für javabasierte Desktop-Anwendungen verfügbar sind, wie aufwendig sich ihre Integration, Konfiguration und Verwendung gestaltet und wie gut diese hinsichtlich funktionaler und nicht-funktionaler Eigenschaften wie Wartbarkeit, Robustheit oder Usability abschneiden. Die Sicherstellung einer hohen Qualität ist gerade bei Desktop-Applikationen besonders wichtig, da diese im Gegensatz zu webbasierten Systemen auf dem lokalen Gerät des Kunden und somit außerhalb der Reichweite des Herstellers liegen. Das Beheben von Fehlern ist demzufolge mit hohem Aufwand und Kosten verbunden.

Die Voraussetzung für die Einschätzung der Praxistauglichkeit verfügbarer Testwerkzeuge ist ihre Anwendung auf ein konkretes und repräsentatives Fallbeispiel. Aufgrund seines Umfangs, seiner Komplexität sowie der zur Realisierung verwendeten Technologien bildet ein bestehendes Event-Management-System das für diese Arbeit ausgewählte Fallbeispiel. Konkret handelt es sich dabei um einen umfangreichen Java-Swing-Client, der eine offlinefähige Einheit eines größeren, verteilten Event-Management-Systems darstellt und dessen Qualität zur Zeit nur durch manuelles Testen gesichert wird. Ein Testwerkzeug stellt nur dann einen Nutzen dar, wenn es mit dem vorhandenen Testprozess und der vorliegenden Projektinfrastruktur harmoniert.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, Aussagen über die Praxistauglichkeit bestehender Open-Source-Werkzeuge für das automatisierte Testen javabasierter Rich-Client-Applikationen zu gewinnen. Neben dem Funktionsumfang sind auch die nicht-funktionalen Aspekte der Werkzeuge, insbesondere ihre Integrationsfähigkeit, ihr Bedienkomfort sowie die Wartbarkeit und Robustheit der erstellten Testskripts Gegenstand der Untersuchung. Im Zentrum steht dabei die Beantwortung der Frage, ob

es für die Benutzeroberfläche des bestehenden Softwaresystems, einen zufriedenstellenden Ansatz zur Testautomatisierung gibt. Aufgrund des repräsentativen Charakters des gewählten Fallbeispiels stellt die Lösung dieser Fragestellung einen Mehrwert für andere Softwareprojekte mit ähnlichem Umfang, Aufbau und Grad an Komplexität dar und die Erkenntnisse können als Leitfaden für die Auswahl eines geeigneten Werkzeugs herangezogen werden.

Zur Erreichung dieses Ziels kommen sowohl theoretische als auch praktische Methoden zur Anwendung. Der theoretische Aspekt umfasst die Analyse, welche Anforderungen und Kriterien ein Testautomatisierungswerkzeug erfüllen muss, damit es einen Nutzen für das bestehende Softwareprojekt und seinen Testprozess darstellt. Der praktische Teil der Arbeit gliedert sich in zwei Bereiche und beginnt mit der Durchführung einer Evaluierung, in welcher am Markt verfügbare, potenziell geeignete Werkzeuge identifiziert, kategorisiert und unter Zuhilfenahme der erstellten Kriterien bewertet werden. Ein wichtiges Zwischenziel ist die Einschränkung der Auswahl auf sechs vielversprechende Werkzeuge.

Die prototypische Implementierung von fünf repräsentativen Testfällen mit den ausgewählten Werkzeugen bildet den zweiten praktischen Teil. Als Resultat werden Erkenntnisse hinsichtlich funktionaler und nicht-funktionaler Eigenschaften erwartet, die letztendlich eine Aussage über die Praxistauglichkeit der Werkzeuge ermöglichen und somit die Beantwortung der zentralen Frage dieser Arbeit zulassen sollen. Die anhand des Fallbeispiels erarbeiteten Resultate sollen insofern von generellem Nutzen sein, als sie für ähnliche Softwareprojekte Geltung besitzen.

1.3 Aufbau der Arbeit

Kapitel 2 thematisiert das Konzept der grafischen Benutzeroberfläche. Zu Beginn wird ein Überblick über deren Entstehungsgeschichte sowie technischen Funktionsweise vermittelt. Im Anschluss werden gängige javabasierte Technologien – insbesondere Java-Swing – zur Erstellung von grafischen Benutzeroberflächen vorgestellt. In Kapitel 3 steht das Thema Software-Testen im Mittelpunkt. Neben den Teststufen und Testarten werden Testmanagement-Tätigkeiten, der fundamentale Testprozess sowie Ansätze zur Testautomatisierung beleuchtet.

Die Beschreibung des Umfelds der ausgewählten Applikation sowie der Entwurf eines darauf zugeschnittenen Kriterienkatalogs wird in Kapitel 4 abgehandelt. Im Anschluss daran werden in Kapitel 5 die Resultate der Auswertung der zuvor definierten Kriterien präsentiert und die Auswahl der näher zu betrachtenden Testautomatisierungswerkzeuge begründet. Weiters werden Informationen zum Aufbau und der Funktionsweise der ausgewählten Werkzeuge bereitgestellt.

Kapitel 6 betrachtet die Praxistauglichkeit der ausgewählten Testautomatisierungswerkzeuge und dokumentiert die durch prototypische Implementierung von fünf repräsentativen Testfällen gewonnenen Erkenntnisse. Abschließend erfolgt in Kapitel 7 die Beurteilung und der Vergleich der analysierten Werkzeuge, wobei sowohl die Eignung für das bestehende Projekt als auch die generelle Einsetzbarkeit sowie die Relevanz der Ergebnisse für ähnliche Softwareprojekte betrachtet wird.

Kapitel 8 fasst die zentralen Aspekte dieser Arbeit zusammen und beschreibt mögliche zukünftige Fragestellungen.

2 Grundlagen zu grafischen Benutzeroberflächen

Dieses Kapitel stellt das Konzept der grafischen Benutzeroberfläche vor. Im Rahmen der Begriffsklärung wird Verständnis für die Entstehungsgeschichte und Bedeutung grafischer Benutzeroberflächen geschaffen, um im Anschluss die technische Funktionsweise von Fenstersystemen näher zu erläutern. Ein weiterer Abschnitt betrachtet Swing als eine gängige Technologie zur Erstellung grafischer Benutzeroberflächen für objektorientierte, java-basierte Desktop-Applikationen. Insbesondere werden die Funktionsweise sowie Besonderheiten von Swing näher beleuchtet.

2.1 Begriffsklärung und Evolution

Zu Beginn scheint eine detaillierte Betrachtung des allgemeinen Begriffs Benutzerschnittstelle sinnvoll, bevor im weiteren Verlauf näher auf das Konzept der grafischen Benutzeroberfläche eingegangen wird. Die International Organization for Standardization (ISO) definiert den Begriff wie folgt [50]:

> *„[Die Benutzerschnittstelle umfasst] alle Bestandteile eines interaktiven Systems (Software oder Hardware), die Informationen und Steuerelemente zur Verfügung stellen, die für den Benutzer notwendig sind, um eine bestimmte Arbeitsaufgabe mit dem interaktiven System zu erledigen.“*

Preim [60] erläutert weiters:

> *„Die Benutzerschnittstelle steht als Mittler zwischen einem Anwendungsprogramm und einem Benutzer; sie macht die Funktionen der Anwendung zugänglich.“*

Alternativ wird die Benutzerschnittstelle auch als Benutzer-, Bedien- oder Softwareoberfläche bezeichnet. Sie übernimmt gemäß dem oben angeführten Zitat die komplexe Aufgabe der Vermittlung zwischen Benutzer und Softwaresystem durch die Bereitstellung von Interaktionsmöglichkeiten. Im Wesentlichen werden dabei die Aktionen des Benutzers auf die korrespondierenden Funktionsaufrufe der Software abgebildet und deren Rückgabewerte so aufbereitet, dass sie für den Benutzer wahrnehmbar und verständlich sind (siehe auch Abb. 2.1). Meist erfolgt die Aufbereitung der Informationen für den Benutzer in visueller oder akustischer Form [9].

Blickt man auf die Anfänge der Softwareentwicklung zurück, lässt sich erkennen, dass wechselseitige und einfache Interaktion nicht immer ein zentrales Kriterium von Benutzerschnittstellen war. Nachfolgende Auflistung liefert einen Überblick darüber, wie sich Benutzerschnittstellen zu dem entwickelt haben, was sie dank ihrer grafischen Oberfläche heute sind: hochgradig interaktiv und verhältnismäßig intuitiv zu bedienen.

Batch Interface: Die erste Generation von Schnittstellen ermöglichte lediglich die Zuführung von Lochkarten und die Ausgabe der Resultate über Zeilendrucker. Die auszuführenden

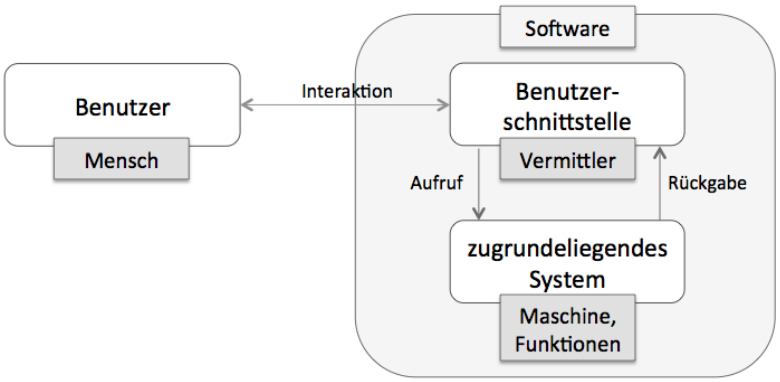


Abbildung 2.1: Die vermittelnde Rolle der Benutzterschnittstelle
angelehnt an [9]

Kommandos wurden vorab spezifiziert, eine tatsächliche Interaktion zwischen Benutzer und System war nicht gegeben [48].

Command Line Interface (CLI): Die Eingabe von Befehlen über die Kommandozeile kann als Beginn der Mensch-Maschine-Interaktion angesehen werden. Die Befehle entstammen dabei dem Vokabular einer Kommandozeilensprache. Sofern Kenntnisse über diese vorliegen, ist eine effektive und flexible Verwendung des zugrundeliegenden Systems möglich. Gleichzeitig ist die Eingabe von Befehlen durch die Verfügbarkeit einer Vielzahl an zusätzlichen Befehlsparametern für Laien unübersichtlich und fehleranfällig [39]. Des Weiteren ermöglichen CLIs nur eine eindimensionale Form der Interaktion, d.h., der Benutzer kann mit dem System nur über die gerade aktuelle Eingabezeile in Kontakt treten [48]. CLIs finden auch heute noch Einsatz (z.B. Unix Shell) [9].

Graphical User Interface (GUI): Gängige Softwaresysteme der heutigen Zeit weisen nahezu immer eine grafische Oberfläche auf. Obwohl erste Ansätze bereits in den 1960ern aufkamen, gelang der große Durchbruch im Einsatz von grafikbasierten Benutzeroberflächen erst gute 20 Jahre später [48]. Ausschlaggebend für deren Entstehung war das Aufkommen hochauflösender Bildschirme und die Erfindung der Maus als Navigationsgerät. Durch den Einsatz von Objekten werden Analogien zur realen Welt geschaffen, die dem Benutzer die Bedienung erleichtern sollen [39].

Grafische Benutzeroberflächen ermöglichen dem Benutzer eine direkte Manipulation der am Bildschirm sichtbaren Objekte, deren Repräsentation sich abhängig von den Aktionen des Benutzers laufend aktualisiert. Direkte Manipulation bedeutet gemäß Definition der ISO [51] das Verschieben oder Verändern der physikalischen Eigenschaften oder Werte der Objekte unter Verwendung der Maus. GUIs werden auch als WIMP (Windows, Icons, Menus, Pointing Device) oder WYSIWYG (What You See Is What You Get) Systeme bezeichnet. Die Abkürzung WIMP verdeutlicht die wichtigsten Elemente grafischer Benutzeroberflächen – nämlich Fenster, Menüs, Icons und die Maus als Navigationsgerät [48].

Natural User Interface (NUI): Aktuelle Trends fokussieren auf den Ausbau des GUI-Konzepts auf weitere Dimensionen der Interaktion, wie Touch, Sound, Sprache oder Bewegung. Grafische Oberflächen sollen menschliche Handlungen besser verstehen und eine ausdrucksstärkere Kommunikation ermöglichen, die an die jeweiligen Bedürfnisse des aktuellen Benutzers angepasst ist [39]. Chlebek [9] prägt zusätzlich die zwei Begriffe Voice User Interface (VUI),

also die Steuerung durch Sprache, und Haptic User Interface (HUI), also die Fähigkeit des Systems, auf Bewegungen des Benutzers zu reagieren [9].

2.2 Technische Funktionsweise

Die GUI eines Softwaresystems benötigt hardwarenahe Mechanismen, um die Aktionen des Benutzers über Maus oder Tastatur zu registrieren und das Zeichnen der Komponenten am Bildschirm zu veranlassen. Diese Mechanismen müssen nicht für jede grafische Benutzeroberfläche von Grund auf neu implementiert werden, sondern werden von einem Fenstersystem (auch: Window-System) für alle Applikationen bereitgestellt. Ein Fenstersystem ist also eine Abstraktionsschicht, welche die Ausgabe von applikationsspezifischen GUIs am Bildschirm sowie die Erkennung der durch den Benutzter ausgelösten Ereignisse übernimmt. Zusätzlich stellt ein Fenstersystem Bibliotheken (sog. Toolkits) häufig benötigter GUI-Komponenten (wie z.B. Button, Textfeld, Dateisystem-Dialog, ...) für Softwareentwickler zur Verfügung [37].

Abbildung 2.2 veranschaulicht die Rolle und Komponenten eines Fenstersystems:

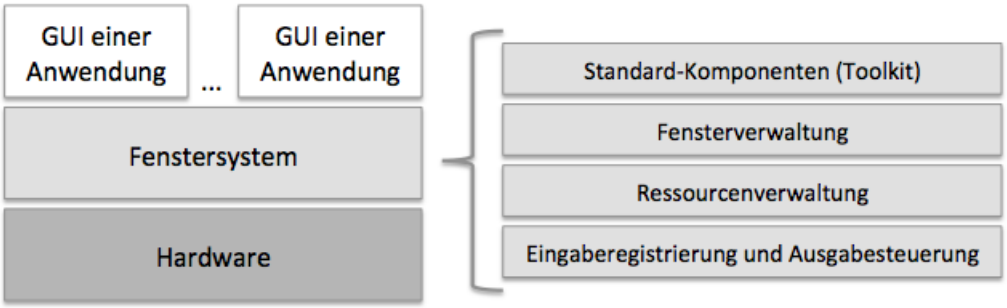


Abbildung 2.2: Rolle und Komponenten eines Fenstersystems
angelehnt an [37]

Das zugrundeliegende Konzept eines Fenstersystems ist die Aufteilung des Bildschirms in rechteckige Segmente, sogenannte Fenster [56]. In diesen Fenstern erfolgt die Ausgabe der GUIs. Fenster sind beliebig steuer- und skalierbar und zeichnen sich durch ihre Fähigkeit zur Interprozesskommunikation aus, worunter der Transfer von Daten aus einem Fenster in ein anderes verstanden wird [60]. Fenster besitzen weiters einen hierarchischen Aufbau, d.h., ein Fenster wird intern in Unterfenster strukturiert, welche die Komponenten der dargestellten GUI repräsentieren. Das Fenster auf oberster Ebene wird auch als Top-Level-Fenster bezeichnet [56].
Klingert [37] nennt als zusätzliche Aufgabe des Fenstersystems die Bereitstellung, Verwaltung und Synchronisation jener Ressourcen (z.B. Bildspeicher, Zeichenflächen, Eingabegeräte), die im Zuge der Interaktion von mehreren GUIs benötigt werden.

Fenstersysteme weisen eine asynchrone und eventbasierte Arbeitsweise auf. Sobald der Benutzer eine Aktion auf der Maus oder Tastatur durchführt, wird ein entsprechendes Event (z.B. Klick, Doppelklick, Loslassen, Ziehen) ausgelöst. GUI-Komponenten registrieren durch Angabe von Callback-Funktionen im Fenstersystem, über welche Events sie informiert werden wollen. Abhängig vom Event-Typ und der Komponente, in dem ein Event aufgetreten ist, ermittelt das Fenstersystem dann die aufzurufende Funktion. Grundsätzlich gibt es drei Auslöser von Events: Zeigegeräte (z.B. Maus), Tastatur und interne Aktionen durch Manipulation der Fenster (z.B. Schließen, Öffnen, Aktualisieren) [60].

Abbildung 2.3 verdeutlicht exemplarisch den hierarchischen Aufbau eines Fensters, indem ein einfaches Programmfenster in seine einzelnen GUI-Komponenten zerlegt wird. Das Top-Level-Fenster ist, basierend auf den darin enthaltenen Komponenten, in drei direkte Subfenster (F1, F2, F3) unterteilt. Das Fenster F1 beinhaltet wiederum drei separate Komponenten (F4, F5, F6).

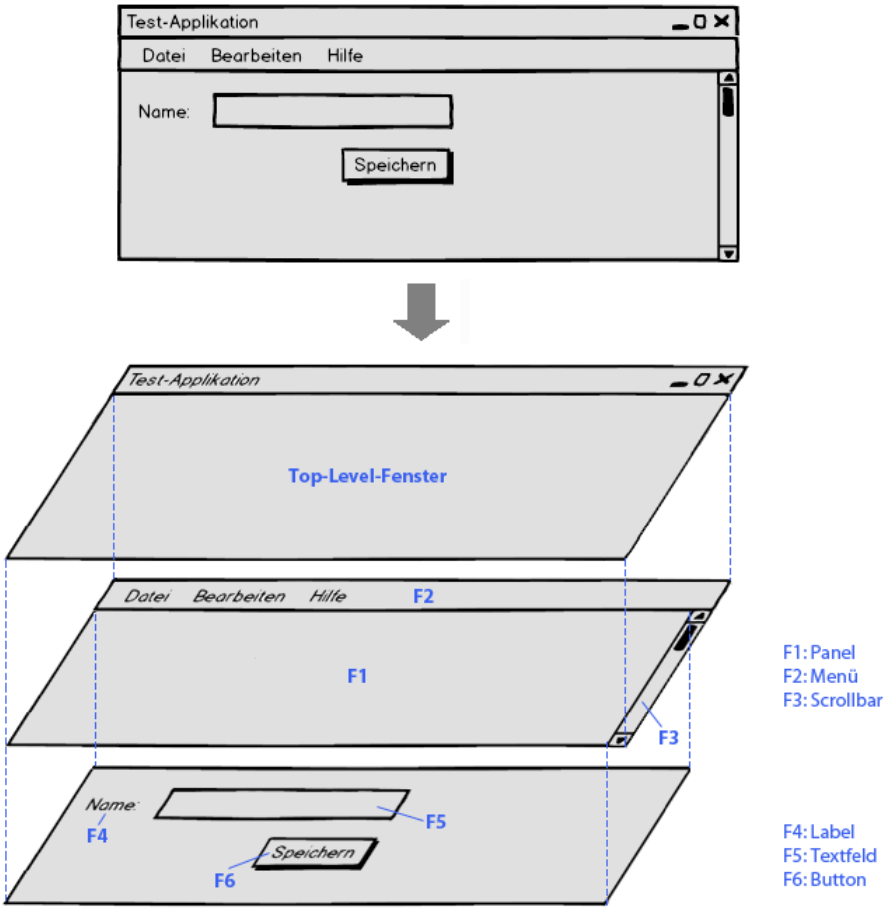


Abbildung 2.3: Hierachischer Aufbau eines Fensters

Üblicherweise sind Fenstersysteme Bestandteil des Betriebssystems, beispielsweise kommt bei allen Unix-basierten Betriebssystemen das X-Window-System zum Einsatz [28].

2.3 Einführung in die Java Foundation Classes

Ein objektorientierter Ansatz eignet sich gut für die Entwicklung eventbasierter grafischer Benutzeroberflächen, da er die Kapselung der GUI-Komponenten in autonome Objekte ermöglicht. Die Programmiersprache Java bietet zusätzlich den Vorteil der Plattformunabhängigkeit und stellt ein Bündel an Bibliotheken zur Entwicklung grafischer Benutzeroberflächen bereit [34]. Diese werden als Java Foundation Classes (JFC) bezeichnet und beinhalten gemäß Geary [22]:

- Abstract Window Toolkit (AWT): Das AWT ist der älteste Bestandteil der JFC und bildet die zugrundeliegende Infrastruktur für die übrigen Bibliotheken. Auf die Bedeutung und Funktionsweise des AWT wird in nachfolgendem Abschnitt 2.4 näher eingegangen.

- Swing: Java Swing ist eine Weiterentwicklung des AWT um zusätzliche, leichtgewichtige Komponenten, wodurch ein plattformunabhängiges Look&Feel realisierbar wird. Auf den Aufbau und die Funktionsweise von Swing wird in nachfolgendem Abschnitt 2.3.2 näher eingegangen.
- Java2D API: Diese Sammlung von Klassen und Funktionen vereinfacht das Zeichnen von zweidimensionalen Grafiken.
- Java Accessibility API: Diese Bibliothek dient als Erweiterung zu Swing und unterstützt die Erstellung barrierefreier Benutzeroberflächen.

2.3.1 Abstract Window Toolkit (AWT)

Das AWT ist als ältester Bestandteil der JFC in den Anfängen der javabasierten GUI-Entwicklung entstanden und hatte ursprünglich zum Ziel, durch Bereitstellung einiger weniger Komponenten die Erstellung einfacher Benutzeroberflächen mit geringem Funktionsumfang zu unterstützen. Heute wird es kaum noch zur Entwicklung von GUIs eingesetzt, sondern bildet die Grundlage und Infrastruktur für Java Swing [22].

Die große Herausforderung bei der Entwicklung des AWT lag im Streben nach Plattformunabhängigkeit. Die mittels AWT erstellten GUIs sollten also nach dem *Write Once, Run Everywhere*-Prinzip unabhängig von der zugrundeliegenden Hardware und dem im Betriebssystem eingesetzten Fenstersystem auf allen Plattformen funktionsfähig sein. Pepper bezeichnet den AWT auch als den *„kleinsten gemeinsamen Nenner“* der Toolkits aller Fenstersysteme und begründet damit seine Kompaktheit und die Beschränkung auf nur wenige Komponenten [56].

Das AWT kann als Abstraktionsschicht zwischen dem nativen Fenstersystem des Betriebssystems und der grafischen Benutzeroberfläche einer Java-Applikation betrachtet werden. Die vom AWT zur Verfügung gestellten Komponenten besitzen allerdings keine eigenständige Implementierung. Stattdessen delegieren sie ihre Darstellung am Bildschirm und die Event-Erkennung an die entsprechende Komponente des nativen Fenstersystems [49]. Die Schnittstellen, die eine einheitliche Kommunikation zwischen AWT und nativen Komponenten ermöglichen, werden als Peers bezeichnet. Abbildung 2.4 visualisiert diesen peer-basierten Ansatz. Soll also eine neue Plattform unterstützt werden, muss für alle Komponenten des AWT eine Peer-Implementierung für das jeweilige Fenstersystem erstellt werden [42].

Als Konsequenz dieses Realisierungskonzepts erhält eine mit dem AWT erstellte GUI das Look& Feel des jeweiligen Betriebssystems. Dies führt zur Problematik, dass trotz der Plattformunabhängigkeit der Programmiersprache keine vollständige Konsistenz zwischen den verschiedenen Plattformen gegeben ist [49].

Nach Geary [22] hat dieser peer-basierte Ansatz zwar die Entwicklungszeit des AWT erheblich verkürzt, es mussten jedoch auch einige Nachteile in Kauf genommen werden. Einerseits gibt es für den Entwickler einer GUI keine Möglichkeit, das Look&Feel der Komponenten einfach anzupassen oder zu erweitern, andererseits weist die Delegation an das native Fenstersystem bei einer großen Anzahl an Komponenten eine schlechte Skalierbarkeit auf.

2.3.2 Swing

Bei Swing handelt es sich um ein in der Praxis weit verbreitetes, umfangreiches Toolkit, das auf der Infrastruktur des AWT aufbaut sowie dessen Konzepte und Komponenten überarbeitet [34]. Im Gegensatz zum AWT sind die Komponenten bei Swing leichtgewichtig, d.h., bis auf

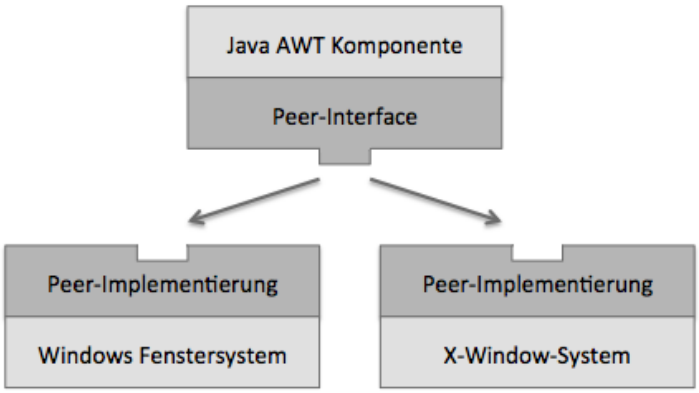


Abbildung 2.4: Kommunikation zum nativen Fenstersystem durch die Nutzung von Peers
angelehnt an [42]

wenige Ausnahmen vollständig in Java implementiert. Swing regelt die Erkennung von Low-Level-Ereignissen und die Ausgabe der Komponenten am Bildschirm eigenständig, anstatt diese Aufgaben an das native Fenstersystem zu delegieren. Durch diese Eigenständigkeit kann ein betriebssystemunabhängiges, konsistentes Look&Feel über die Plattformgrenzen hinweg garantiert und die Flexibilität und Portabilität von Applikationen erhöht werden [49].

Die Ausnahme bilden jene Komponenten, die in Swing zur Erstellung eines Top-Level-Fensters eingesetzt werden können. Sie verbleiben peer-basiert, weil jede leichtgewichtige Komponente in eine schwergewichtige eingebettet sein muss [22]. Neben der Implementierung zusätzlicher Komponenten war auch die Adaption der bereits bestehenden AWT-Komponenten erforderlich. Um eine Namenskollision zwischen den ursprünglichen Komponenten und ihren neuen, meist leichtgewichtigen Varianten zu vermeiden, erhielten alle in Swing implementierten Komponenten ein vorangestelltes J (z.B. `JButton`, `JTextField`) [56].

Nachfolgende Abbildung 2.5 verdeutlicht die Beziehung zwischen dem AWT und Swing. Im Sinne des objektorientierten Vererbungskonzepts leiten sich alle Swing-Komponenten von den AWT-Klassen `Container` und `Component` ab, auch Grafiken, Farben, Schriftsätze und Ansätze zur Positionierung der Komponenten (Layout-Manager) finden in Swing Wiederverwendung [22].

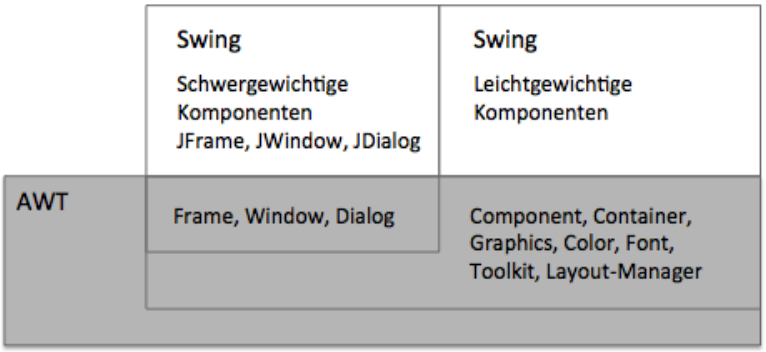


Abbildung 2.5: Beziehung zwischen Swing und AWT
angelehnt an [22]

Zur logischen Trennung von Daten, Darstellung und Verhalten setzt Swing auf eine Model-View-Controller (MVC)-Architektur. Dieser Ansatz reduziert Abhängigkeiten und sorgt somit für einen höheren Grad an Wiederverwendbarkeit und Austauschbarkeit [56]. In seiner klassischen Form trennt das MVC-Architekturmodell gemäß seiner Bezeichnung die folgenden drei Aspekte [11]:

Model: Das Model speichert die inhaltlichen Aspekte einer Komponente und stellt Methoden zur Manipulation dieser bereit. Darunter sind abhängig vom Komponententyp Daten und/oder Zustände zu verstehen. Ein Menü würde also beispielsweise die Menüeinträge im Model ablegen, während eine Scrollbar die Position des Schiebereglers speichern würde.

View: Der View ist für die visuelle Repräsentation der Komponente und ihrer im Model abgelegten Daten am Bildschirm verantwortlich. Er erkennt weiters, wenn Events im Zuständigkeitsbereich der Komponente auftreten und reicht diese, falls relevant, an den Controller weiter.

Controller: Der Controller steuert das Verhalten der Komponente durch Entgegennahme und Verarbeitung der im View ausgelösten Events. Er legt weiters fest, auf welche Events eine Komponente grundsätzlich reagieren kann. Zusätzlich veranlasst er je nach eingetretenem Event die Aktualisierung von Model und/oder View.

Geary [22] weist darauf hin, dass Swing eine abgeänderte Variante der MVC-Architektur für seine Komponenten verwendet, in der View und Controller zu einer Einheit, dem *UIDelegate*, zusammengefasst sind. Der UIDelegate einer Komponente übernimmt also die Erkennung und Verarbeitung der Events sowie das Zeichnen der Komponente am Bildschirm und repräsentiert somit das zentrale Gegenstück zu den Peer-Implementierungen beim AWT. Die geringe Kopplung zum Model ermöglicht den Austausch des UI-Delegates zur Laufzeit. Der Benutzer kann also (sofern vom Entwickler bereitgestellt) bei laufender Anwendung das Look&Feel der Benutzeroberfläche wechseln.

Das Swing Toolkit bietet dem Entwickler standardmäßig einige Look&Feels, unter anderem ist es auch möglich, die Applikation das jeweilig betriebssystemspezifische Aussehen annehmen zu lassen. Durch Bereitstellung einer offenen API wird neben der Erweiterung der bestehenden auch die Programmierung neuer Look&Feels für GUI-Entwickler ermöglicht [16].

3 Grundlagen zum Testen von Software

Dieses Kapitel gibt eine Einführung in das Software-Testen und erläutert wichtige Konzepte dieses Themengebiets. Nach Vorstellung der Teststufen und Testtypen werden insbesondere die Aufgabenbereiche des Testmanagements sowie der fundamentale Testprozess erläutert. Ein weiterer Unterpunkt adressiert das Thema Testautomatisierung und die damit verbundenen Potentiale und Risiken. Abschließend wird der Fokus aufgrund der hohen Relevanz für nachfolgende Kapitel noch auf das Testen von Benutzeroberflächen gelegt.

3.1 Bedeutung und Begriffsklärung

Der Stellenwert des Software-Testens hat sich in den vergangenen 30 Jahren stark gewandelt. Was zu Beginn als eine unstrukturierte, dem Entwickler nach Abschluss der Programmierarbeit überlassene Tätigkeit betrachtet wurde, hat sich mittlerweile als Expertendisziplin etabliert. Dieser Wandel wurde durch Ereignisse in der Vergangenheit bestärkt (wie z.B. Präzisionsfehler im amerikanischen Raketenabwehrsystem, Fehlfunktion eines Bestrahlungsgeräts für Krebspatienten oder Komplettausfall der Tokioter Börse nach Neuinstallation eines Systems), bei denen kritische Softwarefehler im Endprodukt hohe Sach- oder sogar Personenschäden verursacht hatten [10, 59].

Der Begriff Software-Testen wird im Glossar für Software Engineering Terminologie des Institute of Electrical and Electronics Engineers (IEEE) wie folgt definiert [27]:

> *„[Testing is] an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“*

Graham u. a. [23] verstehen unter Software-Testen eine Reihe von Tätigkeiten, die sich auf den gesamten Entwicklungszyklus eines Softwaresystems erstrecken und die Management und Planung erfordern. Diese Tätigkeiten umfassen im Gegensatz zu oben angeführter Definition nicht nur das Ausführen von Programmcode, sondern auch die Prüfung von Spezifikationsdokumenten wie Anforderungskatalogen, Designentwürfen oder Trainingsmaterialien. Weiters spezifiziert die Begriffsdefinition nach Graham u. a. die folgenden drei Ziele, die durch Testen angestrebt werden:

1. Testen soll ermitteln, ob ein Softwareprodukt den spezifizierten Anforderungen entspricht. Dies wird oftmals als Verifikation bezeichnet.
2. Testen soll überprüfen, ob ein Softwareprodukt seinen intentionierten Zweck erfüllt, d.h., ob es den Erwartungen der Benutzer entspricht und ob es diese bei der Durchführung ihrer Tätigkeiten unterstützt. Diese Überprüfung wird oftmals als Validierung bezeichnet.

3. Durch Testen sollen Fehler im Softwareprodukt (möglichst frühzeitig) identifiziert werden. Verständnis und Behebung von Fehlern erhöhen nicht nur die Qualität der Software, sondern ermöglichen eine stetige Verbesserung des Entwicklungsprozesses.

Folgende Definition im Glossar des International Software Testing Qualifications Board (ISTQB) [74] vereint Bedeutung, Strategien und Ziele des Software-Testens kurz und prägnant und deckt sich inhaltlich mit der Darstellung von Graham u. a.:

> *„[Testing is] the process of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.“*

In dieser Definition wird zwischen statischem und dynamischem Testen differenziert. Unter statischem Testen versteht man das Überprüfen eines Teils der Software oder damit verbundener Artefakte im Rahmen von Reviews oder Analysen. Wesentlich ist, dass die Software dabei nicht ausgeführt wird. Im Gegensatz dazu bedeutet dynamisches Testen das Durchführen von Tests im laufenden (Teil-)System [74].

Myers [46] wiederum definiert das Auffinden von Fehlern als primäres Ziel des Software-Testens:

> *„program testing is [. . .] the destructive process of trying to find the errors in a program.“*

Destruktiv bedeutet an dieser Stelle, dass im Rahmen des Testens immer davon ausgegangen werden muss, dass das Softwareprodukt Fehler enthält. Dies zeigt sich auch daran, dass ein Test genau dann als erfolgreich eingestuft wird, wenn er einen Fehler gefunden hat [46]. Software-Testen soll also das Vorliegen von Fehlern aufzeigen, kann jedoch niemals die Korrektheit des Programms bzw. die Abwesenheit von Fehlern garantieren. Ebenso ist das vollständige Testen eines Systems im Allgemeinen praktisch unmöglich, da es die Eingabe und Kombination aller möglichen Werte sowie das Durchlaufen aller möglichen Programmzustände erfordern würde. Wichtig ist daher die Konstruktion von ausdrucksstarken Tests, sodass die Wahrscheinlichkeit, dass durch Testen Fehler gefunden werden, maximiert wird [24]. Bedeutend ist in diesem Zusammenhang allerdings, dass die Tätigkeit des Testens strikt von der des Debuggings abgegrenzt werden muss. Dies bedeutet, dass die Lokalisierung und Behebung der identifizierten Fehler klar außerhalb des Test-Aufgabenbereichs liegt [10].

Im Rahmen dieser Begriffsklärung scheint es weiters sinnvoll, den Begriff *Testfall* näher zu betrachten. Unter einem Testfall versteht man gemäß ISTQB [74] die Summe an Informationen, die zum Überprüfen eines konkreten Ablaufs im System oder eines mit der Software verbundenen Artefakts benötigt werden. In der Regel besteht ein Testfall aus einem Set an Eingabedaten, der Spezifikation von Vor- und Nachbedingungen sowie der Festlegung der erwarteten Ergebnisse.

3.2 Teststufen

Bereits im Zuge der Begriffsklärung wurde angesprochen, dass in allen Phasen der Softwareentwicklung – von der Anforderungsanalyse bis hin zur Inbetriebnahme eines Systems – Bedarf an Tätigkeiten aus der Disziplin des Software-Testens besteht. Das V-Modell (siehe Abb. 3.1)

bietet eine Zuordnung der Arbeitsergebnisse der Entwicklungsphasen zu den korrespondierenden Aktivitäten des Software-Testens. Dieses Vorgehensmodell soll besonders auf die Relevanz von frühzeitigem Testen für den erfolgreichen Verlauf eines Projekts aufmerksam machen. Durch Auffinden von Fehlern in frühen Entwicklungsphasen sollen Kosten gespart und das Risiko einer Nicht-Erreichung der Projektziele reduziert werden [23].

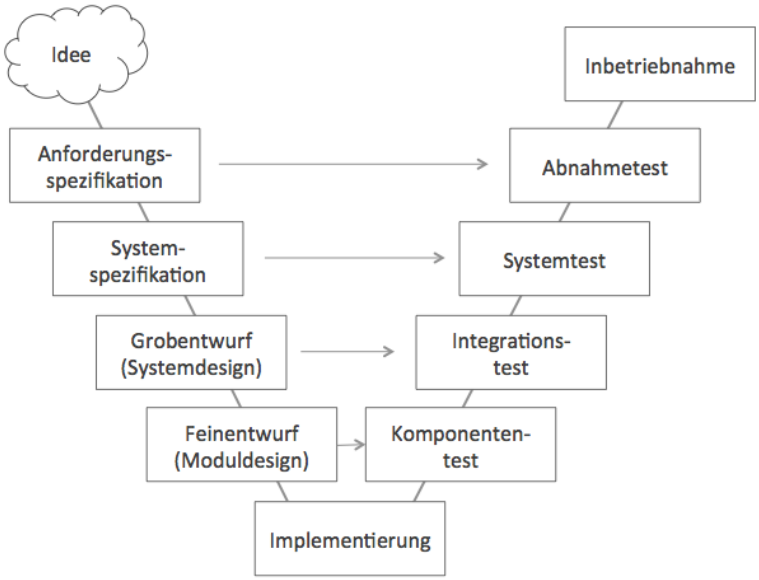


Abbildung 3.1: Darstellung der Teststufen im V-Modell
angelehnt an [23]

Im V-Modell werden die verschiedenen Testtätigkeiten und -aufgaben gemäß ihrer Ziele zu den folgenden vier Teststufen zusammengefasst [10, 23]:

Komponententest: Diese Teststufe wird oftmals auch als Unit- oder Modultest bezeichnet. Es handelt sich dabei um low-level Tests, die die kleinsten Einheiten (also Module oder Klassen) eines Systems isoliert prüfen sollen. Isolation ist ein besonders wichtiges Kriterium, da nur durch sie gefundene Fehler eindeutig der getesteten Komponente zugeordnet werden können. Zur Erreichung von Isolation werden oftmals Testtreiber und Platzhalter eingesetzt, die die Funktionalität von abhängigen Komponenten simulieren und eine vereinfachte Ausführung der zu testenden Komponente ermöglichen.

Komponententests werden üblicherweise vom Entwicklerteam durchgeführt und dienen der Überprüfung von funktionalen und nicht-funktionalen Eigenschaften. Eine besondere Form des Komponententests wird als Test-Driven Development (TDD) bezeichnet. Das Besondere hierbei ist, dass die Tests vor der Entwicklung der zu testenden Komponente implementiert werden.

Integrationstest: Im Rahmen des Integrationstests wird die Interaktion zwischen bereits isoliert getesteten Komponenten überprüft. Es gilt also festzustellen, ob das Zusammenspiel mehrerer Komponenten über deren bereitgestellte Schnittstellen reibungslos funktioniert. Ausschlaggebend für die Vorgehensweise beim Integrationstest ist die bei der Integration gewählte Strategie. Diese legt fest, wie und in welcher Reihenfolge die Komponenten zu einem Gesamtsystem zusammengeführt werden.

Cleff [10] unterscheidet hier zwischen inkrementeller und nicht-inkrementeller Vorgehensweise. Unter inkrementell ist zu verstehen, dass immer nur eine einzige oder einige wenige neue Komponenten hinzugefügt werden. Dies hat den Nachteil, dass sehr viele Testtreiber und Platzhalter für die noch nicht integrierten Komponenten benötigt werden. Bei nicht-inkrementellen Strategien werden alle Komponenten gleichzeitig integriert. Dies reduziert zwar den Aufwand für die Entwicklung von Platzhaltern, führt jedoch dazu, dass im Falle eines Fehlers nur noch sehr schwer lokalisiert werden kann, welche Komponente der Verursacher ist.

Grechenig u. a. [24] kategorisieren Integrationsstrategien in horizontal und vertikal, abhängig davon, ob schichtenweise oder funktionsorientiert vorgegangen wird.

Systemtest: Der Systemtest ist unternehmensintern die abschließende Teststufe bei der Entwicklung eines Softwareprodukts und hat die Überprüfung des integrierten Gesamtsystems zum Ziel. Die Testfälle werden dabei unter Zuhilfenahme der Systemspezifikation, d.h., aus Anwendungsfällen, Geschäftsprozessen, oder ähnlichen Dokumenten abgeleitet. Der Systemtest sollte sowohl funktionale als auch nicht-funktionale Eigenschaften überprüfen.

Wichtig ist der Einsatz einer kontrollierten Testumgebung, die möglichst ähnlich zur späteren Produktionsumgebung sein sollte, um die Maskierung umgebungsspezifischer Fehler zu vermeiden. Die Konfiguration einer solchen Testumgebung kann sich als sehr aufwendig herausstellen, da Teile der Systemlandschaft des Kunden mitabgebildet werden müssen, um auch das Zusammenspiel der entwickelten Software mit bestehenden Systemen zu prüfen.

Abnahmetest: Der Abnahmetest ist eine besondere Form des Systemtests und wird im Rahmen der Abnahme des Softwareprodukts meist durch den Kunden oder einen zukünftigen Benutzer des Systems durchgeführt. Im Vordergrund steht an dieser Stelle nicht mehr die Qualitätssicherung, sondern das Aufzeigen, dass die vertraglich vereinbarten Leistungen vom Auftragnehmer zufriedenstellend umgesetzt wurden und dass das System entsprechend benutzerbar gestaltet ist.

Bei Standard-Softwareprodukten für die breite Masse findet aufgrund des Fehlens eines Auftraggebers keine formale Abnahme statt. Alternativ kann anstelle des Abnahmetests ein Alpha- bzw. Beta-Test durchgeführt werden. Im Rahmen des Alpha-Tests verwenden potenzielle Kunden das System am Standort des Herstellers. Im Gegensatz dazu sieht der Beta-Test den Einsatz des Produkts bei ausgewählten Kunden vor. Durch Alpha- bzw. Beta-Tests soll das Feedback potenzieller Kunden eingeholt und die Akzeptanz des neuen Systems geprüft werden.

In der Praxis existieren verschiedene Varianten des V-Modells, die sich in der Benennung und Anzahl der Entwicklungs- und Teststufen geringfügig unterscheiden können. Gemäß Graham u. a. ist das vierstufige V-Modell das am häufigsten vertretene. Die verschiedenen Ausprägungen lassen sich damit begründen, dass abhängig von Projekt, Produkt und Entwicklungsprozess unterschiedliche Prioritäten vorliegen und somit das Hinzufügen oder Weglassen bestimmter Testphasen erforderlich sein kann [23]. So stellt Franz [20] dem Komponententest eine weitere Teststufe voraus, den **Klassentest**. Die Durchführung dieser Stufe liegt vollständig beim Entwickler, ist eher informell und soll die Funktionalität einzelner Klassen prüfen.
Boehm [7] hingegen fasst Integrations- und Systemtest zu einer Teststufe zusammen und führt dafür zur Prüfung gegen die Anforderungen eine zusätzliche Stufe, den **Betriebstest**[1], ein. Gemäß ISTQB [74] versteht man darunter das Testen des Gesamtsystems in der Produktionsumgebung.

[1] Eine detailliertere Beschreibung dieser Stufe ist vom Autor in [7] nicht gegeben.

3.3 Testtypen

Um umfassende Qualitätssicherung zu betreiben und Fehler in unterschiedlichen Bereichen eines Softwaresystems zu identifizieren, werden verschiedenartige Vorgehensweisen beim Testen benötigt [78]. Diese setzen unterschiedliche Testschwerpunkte und werden in der Literatur als Testtypen bezeichnet. Oftmals wird auch das Synonym Testarten verwendet [10].

Cleff definiert den Begriff wie folgt [10]:

> *„Testtypen stellen eine Bündelung verschiedener Testtechniken dar und beschreiben die grundlegende Art und Weise, wie getestet wird."*

Ein Testtyp ermöglicht somit, bestimmte Aspekte – also konkrete Qualitätsmerkmale – einer Komponente oder eines Systems zu überprüfen. Ein Testtyp kann dabei Anwendung innerhalb mehrerer Teststufen finden [74].

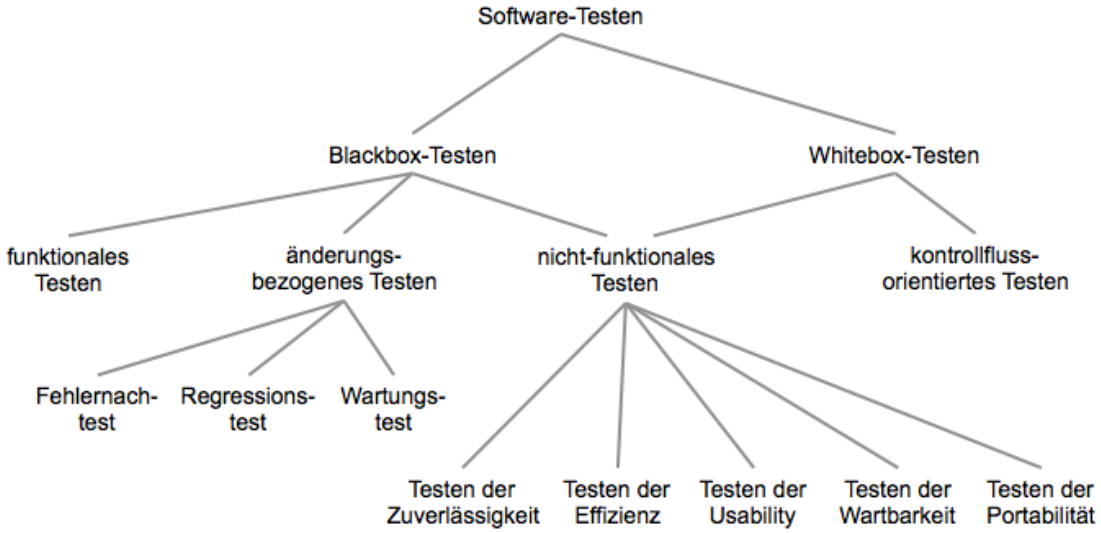


Abbildung 3.2: Kategorisierung der Testtypen

Abbildung 3.2 gibt einen Überblick über die verschiedenen Testtypen und kategorisiert diese auf unterschiedlichen Abstraktionsebenen. Es gilt anzumerken, dass es sich bei dieser Abbildung um keine vollständige und detaillierte Auflistung aller Testtypen handelt, da an dieser Stelle lediglich eine Abgrenzung zwischen den im Rahmen dieser Arbeit relevanten Begriffen geschaffen werden soll. Nachfolgend wird im Text auf die visualisierten Begriffe näher eingegangen.

Auf einem höheren Abstraktionslevel kann zwischen Blackbox- und Whitebox-Testverfahren differenziert werden:

Blackbox-Verfahren: Im Rahmen von Blackbox-Tests wird das System als eine schwarze Box angesehen, die zuvor definierte Eingaben in Ausgaben überführt. Die interne Struktur und der Programmcode sind dabei nicht bekannt. Zur Ableitung von Testfällen werden ausschließlich

Spezifikationsdokumente herangezogen [21]. Aus diesem Grund bezeichnet Cleff Blackbox-Verfahren auch als spezifikationsgetriebenes Testen. Der Begriff Blackbox-Testen umfasst viele verschiedene Testtypen, die sich in funktionale, nicht-funktionale und änderungsbezogene Tests untergliedern lassen [10]. Diese werden in nachfolgenden Unterkapiteln separat betrachtet.

Whitebox-Verfahren: Beim Whitebox-Testing – oftmals auch Glassbox-Testing oder strukturgetriebenes Testen – besitzt der Tester Wissen über die interne Struktur eines Systems und hat Einblick in den Programmcode. Dies unterstützt einerseits ein fokussierteres Vorgehen und ermöglicht andererseits, Fehler im Code zu finden sowie ungenutzen oder unperformanten Code zu identifizieren. Whitebox-Verfahren sind allerdings zeitaufwendig und deshalb mit hohen Kosten verbunden [21].

Zur Realisierung von Whitebox-Tests werden verschiedene Arten von Analysen – wie Code-Inspektionen, Walkthroughs oder formale Verifikationsansätze – eingesetzt. Im Fokus steht dabei der Kontrollfluss eines Programms, welcher zur Bestimmung der Testüberdeckung benötigt wird. Es geht also um die Fragestellung, wie viele und welche Testfälle notwendig sind, um einen bestimmten prozentuellen Anteil an Strukturelementen im Rahmen der Tests zu durchlaufen. Je nachdem, wie tiefgreifend die Überdeckung angestrebt wird, sind unter Strukturelementen Anweisungen, Bedingungen oder Pfade zu verstehen [10, 21].

3.3.1 Funktionales Testen

Der Referenzliteratur ist nicht eindeutig zu entnehmen, ob Whitebox-Testing als Teilbereich des funktionalen Testens einzuordnen ist. Während manche Autoren (so z.B. Grechenig u. a. [24]) sowohl struktur- als auch spezifikationsbezogene Ansätze als funktionales Testen einstufen, ordnet Cleff [10] funktionale Tests eindeutig den Blackbox-Verfahren zu. Eine ähnliche Differenzierung wird auch in [78] vorgenommen. Im Rahmen dieser Arbeit wird funktionales Testen daher ebenfalls als spezifikationsgetriebene Testart angesehen.

Funktionales Testen beschäftigt sich also mit der Frage, ob der im Fokus des Tests stehende Systemteil die spezifizierte Funktionalität aufweist. Nach Cleff geht es hierbei sowohl um den fachlichen als auch um den technischen Funktionsumfang. Der fachliche Funktionsumfang bezieht sich auf die vom Auftraggeber spezifizierten Anforderungen an das System, während der technische Funktionsumfang alle Spezifikationsdokumente, die im Rahmen der Analyse- oder Designphase durch den Auftragnehmer erstellt wurden, umfasst. Aus welchem Spezifikationsdokument die Testfälle abgeleitet werden, ergibt sich aus der jeweiligen Teststufe, in welcher der funktionale Test angewandt wird [10]. Bei einem Komponententest würde demnach gemäß des V-Modells der Feinentwurf als Basis herangezogen werden.

3.3.2 Nicht-funktionales Testen

Diese Art des Testens beschäftigt sich mit der Überprüfung der nicht-funktionalen Qualitätsattribute einer Software und dient der Bewertung ihrer Leistungsfähigkeit. Insbesondere geht es um die Beantwortung der Frage, wie gut oder wie schnell ein Softwaresystem seine Funktionen erfüllt [10, 23]. Das Testen nicht-funktionaler Systemeigenschaften umfasst gemäß [29] die Kategorien Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Portabilität. Für jede dieser Kategorien sind weitere Unterteilungen vorgesehen. Weiters liegen auch verschiedenste Ansätze vor, um die Erfüllung dieser Qualitätseigenschaften durch Tests zu überprüfen.

Für die Bewertung der Wartbarkeit und Portabilität ist jedoch auch die Inspektion des Programmcodes erforderlich, weshalb hierfür auch strukturgetriebene Testarten zum Einsatz kommen [10].

Das Testen von nicht-funktionalen Eigenschaften besitzt nur relativ wenig Relevanz für diese Arbeit, daher werden die verschiedenen nicht-funktionalen Testarten an dieser Stelle nicht näher ausgeführt. Umfangreiche Erläuterungen dazu finden sich in [20, 23, 29].

3.3.3 Änderungsbezogenes Testen

Änderungsbezogenes Testen dient der Sicherstellung, dass aufgrund der Durchführung von Änderungen oder Behebung von Fehlern keine unerwünschten Seiteneffekte im System entstanden sind und dass bekannte Fehler ordnungsgemäß behoben wurden [10]. Die folgenden drei Testarten werden in die Kategorie der änderungsbezogenen Tests eingeordnet:

Fehlernachtest: Der Fehlernachtest prüft die erfolgreiche Behebung eines Fehlers, der in einem früheren Testlauf identifiziert wurde. Dabei werden in der Regel nur jene Testfälle erneut ausgeführt, die aufgrund des Fehlers fehlgeschlagen sind [10].

Regressionstest: Auch im Rahmen von Regressionstests werden keine neuen Testfälle spezifiziert, sondern bereits vorhandene wiederausgeführt. Regressionstests können in beliebigen Teststufen eingesetzt werden. Durch sie soll sichergestellt werden, dass die Behebung eines Fehlers keine neuen Fehler in anderen Systemteilen hervorgerufen und ein behobener Fehler keine anderen Fehler maskiert hat. Regressionstests werden aber auch nach funktionalen Änderungen im System durchgeführt. In diesem Fall dienen sie der Überprüfung, dass die unveränderten, bereits bestehenden Systemteile nach wie vor wie gewünscht arbeiten. Regressionstests sind auch dann sinnvoll, wenn sich die zugrundeliegende technische Infrastruktur verändert hat (z.B. Austausch der Datenbank oder Upgrade auf eine neue Browserversion) [20].
Während sich der Fehlernachtest also mit den lokalen Auswirkungen eines Fehlers befasst, ist der Regressionstest modulübergreifend und adressiert mehrere Systemteile oder das Gesamtsystem.

Wartungstest: Ein Wartungstest wird ausgeführt, um Änderungen am bereits ausgelieferten System oder dessen technischer Infrastruktur zu testen. Er findet also immer in der Produktionsumgebung statt und wird nach Behebung von gemeldeten Fehlern, Adaptierungen im Produktumfeld oder Verbesserung von funktionalen oder nicht-funktionalen Eigenschaften angewandt [10, 74].

3.4 Testmanagement

Software-Testen ist in größeren Projekten ein komplexes und umfangreiches Tätigkeitsfeld und kann als eigenes Subprojekt angesehen werden. Konkrete Zielsetzungen sowie eine systematische Vorgehensweise werden also zur Bewältigung dieser Komplexität benötigt. Es ergibt sich daraus die Notwendigkeit einer unabhängigen Instanz, die für das Management der Testtätigkeiten zuständig ist [23, 59].

Sneed, Baumgartner und Seidl [71] bezeichnen Testmanagement als das Projektmanagement der Testaktivitäten. Sie plädieren für ein eigenständiges und unabhängiges Testmanagement, das mit

dem Projektstart einzurichten ist und das sowohl an das Projekt- als auch an das Qualitätsmanagementteam zu berichten hat. Es wird weiters darauf hingewiesen, dass die Einrichtung ebendieser Managementinstanz oftmals aus den folgenden Gründen vernachlässigt wird:

- Testmanagement wird als Aufgabenbereich des Projektmanagements angesehen.
- Testmanagement wird als Verantwortung des Qualitätsmanagements betrachtet.
- Das Bewusstsein von der hohen Relevanz des Software-Testens und damit verbunden des Managements der Tätigkeiten fehlt vollständig.

Testmanagement sollte keinesfalls als Aufgabe des Projektmanagers betrachtet werden, da in diesem Fall die angestrebten Ziele miteinander in Konflikt stehen. Der Projektmanager sieht sich primär für die Einhaltung der Termine und Kosten gegenüber dem Kunden und anderen Stakeholdern verantwortlich, während der Testmanager auf die Sicherung der Softwarequalität fokussieren sollte [71].
Professionelles Testmanagement hat nach Pinkster u. a. [59] eine klar definierte und akzeptierte Rolle innerhalb einer Organisation und innerhalb von Projekten. Es ist mit einem eigenen Budget ausgestattet und neben dem Management der Testaktivitäten für die Kommunikation der Ergebnisse und Fortschritte nach außen verantwortlich.

Der Glossar des ISTQB [74] definiert die wichtigsten Aufgabenbereiche des Testmanagements wie folgt:

> *„[Test Management is] the planning, estimating, monitoring and control of test activities, typically carried out by a test manager."*

Die in obiger Definition genannten fachlichen Aufgabenfelder des Testmanagers finden sich im fundamentalen Testprozess, der in nachfolgendem Unterkapitel 3.5 vorgestellt wird, wieder. Insbesondere die Phasen Planung, Steuerung und Berichterstattung sowie die kontinuierliche Überwachung aller anderen ausgeführten Testtätigkeiten liegen in der Verantwortung des Testmanagers. Sie werden an dieser Stelle nicht näher ausgeführt, sondern können in nachfolgendem Unterkapitel im Detail nachgelesen werden.

Ein Testmanager muss neben den fachlichen und testspezifischen Fähigkeiten auch soziale Kompetenzen aufweisen. Er fungiert als Motivator und Ansprechpartner für das Testteam und ist auch zu Projektbeginn mit dessen Zusammensetzung betraut. Damit verbunden ist die Spezifikation der benötigten Fähigkeiten und Skills und die Besetzung der offenen Positionen mit den besten dafür verfügbaren Personen. Es gilt dabei, einen guten Mittelweg zwischen fachlichen und sozialen Kompetenzen der Teammitglieder zu wählen und Rahmenbedingungen für einen erfolgreichen und konstruktiven Verlauf der Testaktivitäten bereitzustellen [71].

3.5 Fundamentaler Testprozess

Die verschiedenen Teststufen weisen einen ähnlichen Ablauf immer wiederkehrender Testaktivitäten auf. Diese Aktivitäten reichen von der Planung und Steuerung der Tätigkeiten über das Design der Testfälle, die Testausführung bis hin zur Auswertung und Dokumentation der Ergebnisse. Sie unterscheiden sich zwischen den Teststufen lediglich in Form und Umfang [23].

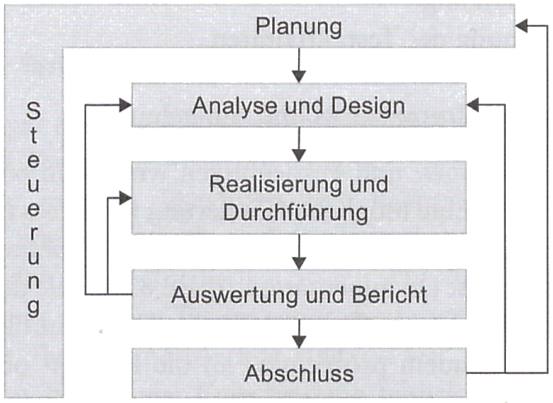


Abbildung 3.3: Fundamentaler Testprozess
entnommen aus [69]

Der Ablauf dieser wiederkehrenden Aktivitäten ist im fundamentalen Testprozess dokumentiert (siehe Abbildung 3.3) und kann je nach Projekt und angewandtem Entwicklungsmodell vom Testmanager an die gegebenen Herausforderungen angepasst werden. Grundsätzlich ist eine sequentielle Abarbeitung der Aktivitäten vorgesehen, projektspezifisch können sich diese aber auch überlappen, parallel stattfinden oder iterativ angewandt werden [69].

In den nachfolgenden Unterkapiteln wird näher auf die in Abbildung 3.3 dargestellten Aktivitäten des fundamentalen Testprozesses eingegangen.

3.5.1 Planung und Steuerung

Die Tätigkeiten dieser Phase liegen in der Verantwortung des Testmanagers. Die Planung dient der Festlegung der Rahmenbedingungen für das Testen in Form eines Testkonzepts und ist eine in sich abgeschlossene Phase. Im Gegensatz dazu begleitet die Steuerung den gesamten Testprozess und beinhaltet die fortlaufende Überwachung des Testfortschritts entsprechend der Planung und die Gegensteuerung im Falle einer Abweichung [69].

Mit der **Planung** sollte nach Spillner u. a. so früh wie möglich begonnen werden, optimalerweise parallel zur Planung der Entwicklungstätigkeiten [72]. Zu Beginn muss ein grundlegendes Verständnis für das Projekt, dessen Risiken und Ziele geschaffen werden, um anschließend eine darauf abgestimmte Vorgehensweise für das Testen ableiten zu können. Sämtliche Erkenntnisse dieser Phase werden in Form eines Testkonzepts dokumentiert. Gemäß Graham u. a. [23] ergeben sich die folgenden Aufgabenbereiche innerhalb der Planungsphase:

- Identifikation der organisatorischen und technischen Risiken sowie der Testobjekte
- Definition der Vorgehensweise beim Testen und Priorisierung der Testobjekte und -tätigkeiten
- Planung der benötigten Ressourcen aus technischer (notwendige Hard- und Software für die Testumgebung) und menschlicher (Zusammensetzung des Testteams) Sicht
- Erstellung eines Zeitplans und Festsetzung von Start- und Enddaten für die verschiedenen Tätigkeiten

- Festlegung der Ausgangskriterien: Diese Kriterien dienen als Entscheidungsgrundlage, wann mit dem Testen aufgehört werden kann. Sie sind je nach Projekt unterschiedlich und definieren beispielsweise, welche Testfälle zwingend durchlaufen müssen oder welcher Überdeckungsgrad mindestens erreicht werden muss. Weitere Beispiele für Ausgangskriterien finden sich in [23]. Neben den Ausgangskriterien werden oftmals auch Abbruchbedingungen festgelegt. Diese legen fest, welche gravierenden Fehler oder Probleme ein vorzeitiges Beenden des Testens erfordern [10].

Die **Steuerung** ist eine Management-Tätigkeit, die immer präsent ist und parallel zu allen anderen Aktivitäten abläuft. Einerseits dient sie der Erhebung und Kontrolle des Testfortschritts, andererseits der frühzeitigen Identifikation von Planabweichungen und eintretenden Risiken. Dieses Wissen ermöglicht, wirksame Maßnahmen zur Gegensteuerung zu setzen oder die bestehenden Pläne zu überarbeiten [72]. Zur Erhebung des Testfortschritts werden dabei Statusberichte herangezogen oder regelmäßige Statusmeetings einberufen. Mögliche Aktionen zur Gegensteuerung liegen in der Verlagerung der Prioritäten oder der Abänderung von Zeit- oder Ressourcenplanung [69]. Graham u. a. [23] nennt als zusätzliche Aufgabe der Steuerung die Bereitstellung von Informationen an die verschiedenen Stakeholder des Projekts, insbesondere an das Projektmanagement.

Seidl, Baumgartner und Bucsics [69] betten auch Tätigkeiten zur Testautomatisierung in den Testprozess ein. Insbesondere gilt es in der Planungsphase zu entscheiden, ob Aufgaben automatisiert werden sollen und wenn ja, in welchen Teststufen und in welchem Ausmaß die Automatisierung einen Nutzen für das Projekt darstellt. Sofern sich der Einsatz von Testautomatisierung lohnt, muss diese bei oben angeführten Planungsaufgaben stets mitberücksichtigt werden. Hinzu kommt die zusätzliche Aufgabe, Werkzeuge und Frameworks zur Umsetzung der Automatisierung auszuwählen. In diesem Zusammenhang muss auch entschieden werden, ob kommerzielle, selbstentwickelte oder Open-Source-Werkzeuge zum Einsatz kommen sollen.
Weiters muss im Rahmen der Steuerung auch regelmäßig auf den Status der Testautomatisierung geachtet werden. Besonders wichtig ist, die technische Machbarkeit sowie Kosten-Nutzen-Faktoren der eingesetzten Werkzeuge zu überwachen.

3.5.2 Analyse und Design

Diese Prozessphase umfasst hauptsächlich Vorbereitungs- und Reviewtätigkeiten, damit frühzeitig Fehler in Projektdokumenten gefunden und qualitativ hochwertige Testfälle und Testumgebungen entworfen werden können. Die folgende Vorgehensweise wird dabei vorgeschlagen:

Qualitätssicherung der Testbasis: Unter der Testbasis versteht man die in der jeweiligen Teststufe herangezogenen Spezifikations- und Projektdokumente (wie z.B. Anforderungskatalog, Designdokumentation oder Risikoanalyse). Mithilfe von Reviews sollen ungenaue Angaben, Widersprüche und Fehler in der Testbasis identifiziert werden, sodass diese nicht in nachfolgende Arbeitsschritte übernommen werden [72]. Insbesondere muss auch die Testbarkeit der Testbasis geprüft werden. Graham u. a. [23] bringen dazu das folgende Beispiel: Die Anforderung „Das System muss schnell antworten“ ist zu unspezifisch und daher nicht testbar. Sehr wohl testbar ist allerdings „Das System muss eine Antwortzeit von fünf Sekunden aufweisen, wenn nicht mehr als 20 Benutzer gleichzeitig aktiv sind“.

Identifikation der Testbedingungen: Eine Testbedingung charakterisiert eine Einheit innerhalb eines (Teil-)Systems (also eine Funktion, Transaktion oder ein Qualitätsattribut) und kann

durch einen oder mehrere Testfälle geprüft werden [74]. Testbedingungen legen unter Berücksichtigung der Testbasis fest, was und wie intensiv getestet werden soll. Der Detailliertheitsgrad der Testbedingungen ist abhängig von den identifizierten Risiken und der Konkretheit der Testbasis. Liegt diese nur abstrakt vor, so können auch die Testbedingungen nur sehr allgemein gehalten werden. Ist hingegen bei einem Aspekt des Systems ein besonders hohes Risiko gegeben, sollten die korrespondierenden Testbedingungen entsprechend detaillierter ausfallen [72].

Entwurf der Testfälle: In diesem Schritt geht es um die Ableitung von Blackbox-Testfällen aus den Testbedingungen unter Zuhilfenahme der Testbasis. Wichtig ist die Auswahl repräsentativer Testfälle, welche die risikoreichen Aspekte weitgehend abdecken [23]. Für jeden Testfall gilt es also die Vorbedingungen, Nachbedingungen, Eingabedaten und Sollergebnis bzw. -verhalten zu spezifizieren. Jeder Testfall sollte weiters umfangreich dokumentiert werden. Die Dokumentation ist so zu gestalten, dass Testfälle reproduzierbar, nachprüfbar und rückverfolgbar gegenüber den Anforderungen sind [72].

Konzeption der Testumgebung: Die für die Umsetzung und Ausführung der Testfälle benötigte Testumgebung muss geplant und mit dem Aufbau der Infrastruktur begonnen werden [72].

Wird Testautomatisierung angestrebt, so muss dies bei der Konzeption der Testumgebung berücksichtigt und in dieser Prozessphase mit der Adaption und Konfiguration der ausgewählten Testwerkzeuge begonnen werden. Zusätzlich ist bei der Ableitung der Testfälle eine Entscheidung zu treffen, welche Testfälle automatisiert werden sollen. Im Anschluss daran ist die technische Umsetzbarkeit dieser Testfälle mit den ausgewählten Werkzeugen zu prüfen [69].

3.5.3 Realisierung und Durchführung

Diese Phase des Testprozesses hat die Implementierung der geplanten Testfälle sowie das eigentliche Testen des in dieser Teststufe relevanten Systemteils durch das Ausführen der Testfälle zum Ziel. Der folgende Ablauf ergibt sich:

Abschluss der vorbereitenden Tätigkeiten: Der in der vorhergehenden Prozessphase begonnene Aufbau der Testinfrastruktur muss abgeschlossen und qualitätsgesichert werden. Durch Prüfen der Infrastruktur wird verhindert, dass die Ursache für das Fehlschlagen von Tests in einer falsch konfigurierten Testumgebung liegt. Es kann auch zielführend sein, spezielle Testfälle für das Prüfen der Testumgebung einzusetzen. Neben der Testinfrastruktur muss auch der Entwurf der Testfälle und der zugehörigen Testdaten abgeschlossen werden [72].

Realisierung der Testfälle: Zur Erhöhung der Effizienz ist eine Gruppierung logisch zusammengehöriger Testfälle zu Test-Suiten empfehlenswert [23]. Ziel dabei ist, eine sinnvolle Ausführungsreihenfolge festzulegen. Die Schwierigkeit bei der Erstellung von Test-Suiten liegt in den Abhängigkeiten, die die Testfälle untereinander aufweisen können. Es gilt also darauf zu achten, dass die Testfälle so angeordnet werden, dass die Nachbedingung des einen mit der Vorbedingung des anderen Testfalls vereinbar ist [10].
Wurde der Einsatz von Werkzeugen und Frameworks beschlossen, muss an dieser Stelle die Implementierung der geplanten Testfälle mit den jeweiligen Tools erfolgen. Durch den anschließenden Einsatz von Code-Reviews sollte der erstellte Code überprüft werden [69].

Testdurchführung: Im Mittelpunkt steht die manuelle oder automatisierte Ausführung der Testfälle und Test-Suiten. Besonders wichtig ist die Überprüfung, ob das tatsächliche Ergebnis

oder Verhalten mit dem spezifizierten übereinstimmt. Hier muss sorgfältig vorgegangen werden, damit keine bestehenden Fehler übersehen werden und korrektes Verhalten nicht fälschlicherweise als Fehlverhalten verzeichnet wird. Unabhängig davon, ob die Testfälle manuell oder automatisiert ausgeführt werden, ist der gesamte Testdurchführungsprozess detailliert zu protokollieren. Dadurch soll nachvollziehbar werden, welcher Testfall wann, durch wen, mit welchem Ergebnis und gegen welche Version der zu testenden Software durchgeführt wurde. Schlägt ein Test fehl, so muss ein entsprechendes Reporting erfolgen und die Fehlerkorrektur eingeleitet werden [72].

Überprüfung der Fehlerkorrektur: Nach Behebung der eingemeldeten Fehler sollten Fehlernachtests sowie Regressionstests durchgeführt werden, um die erfolgreiche Behebung des Fehlers und das Funktionieren des Restsystems sicherzustellen [23].

3.5.4 Auswertung und Bericht

Um entscheiden zu können, ob das Testen innerhalb einer Teststufe oder im Gesamten abgeschlossen werden kann, muss eine Überprüfung der in der Planungsphase definierten Ausgangskriterien durch den Testmanager erfolgen. Dafür ist es erforderlich, konkrete Zahlenwerte oder Metriken anhand der Testprotokolle zu berechnen. Sind die Ausgangskriterien nicht erfüllt, obliegt es dem Testmanager, über die weitere Vorgehensweise zu entscheiden. Mögliche Aktionen sind die Fortsetzung des Testens durch Design und Implementierung zusätzlicher Testfälle oder die Aufweichung der Ausgangskriterien.

Zusätzlich sind die Ergebnisse und Erkenntnisse für jede Teststufe zu dokumentieren und ein abschließender Gesamtbericht ist zu erstellen. Ein solcher Testbericht fasst die Aktivitäten und Resultate der vergangenen Phasen zusammen, listet die identifizierten Abweichungen und gibt eine Bewertung der Qualität des getesteten (Teil-)Systems ab [72]. Wurde die Testdurchführung automatisiert, so kann sich die Analyse und Bewertung der Ergebnisse weitaus aufwendiger gestalten, da derartige Tools oftmals eine Vielzahl an Aufzeichnungen und Daten produzieren [69].

3.5.5 Abschluss

Ist die Testphase in einem Projekt und sind somit alle Testaktivitäten abgeschlossen (oder gewollt abgebrochen), verbleiben noch die folgenden drei Aufgaben:

Archivierung der Testmittel: Der Begriff Testmittel umfasst sämtliche während der vergangenen Prozessphasen erstellten Artefakte, so die Planungsdokumente, Berichte, Testfälle und insbesondere die aufgebaute Testinfrastruktur samt den eingesetzten Werkzeugen und Frameworks. Die Testmittel müssen vollständig archiviert werden, da sie möglicherweise zum Testen von Änderungen – darunter fallen sowohl Fehlerkorrekturen als auch Kundenwünsche – erneut benötigt werden [72]. Auch eine Wiederverwendung mancher Testmittel in zukünftigen Projekten ist denkbar [23].

Übergabe: Üblicherweise geht die Verantwortung für die Weiterentwicklung eines Systems nach Testende auf ein Wartungs- oder Supportteam über. Dazu müssen sowohl das erstellte System als auch die Testmittel an diese Einheit übergeben werden. Im Falle noch offener Fehler gilt weiters zu vereinbaren, wann deren Behebung im Produktivsystem zu erfolgen hat [10].

Rückblick und Erfahrungswert: Abschließend sollte Resümee über das vergangene Testprojekt gezogen werden. Die Vorgehensweise soll kritisch hinterfragt, Best-Practices und Lessions-

Learned für spätere Projekte dokumentiert werden. Ziel ist, im Laufe der Zeit umfangreiches Wissen zu sammeln, Erfahrungen an andere weiterzugeben und somit im Unternehmen langfristig einen qualitativ hochwertigen Testprozess zu etablieren [69].

3.6 Testautomatisierung

In großen Softwareprojekten ist manuelles Testen sehr zeitintensiv und verursacht dadurch hohe Kosten. Die einzige Möglichkeit, Ausgaben zu sparen, ohne dabei den Umfang des Testens zu reduzieren, liegt in der Steigerung der Effizienz, was bedeutet, dass in derselben Zeit mehr Tätigkeiten erledigt werden müssen [71]. Der Begriff Testautomatisierung bezeichnet den Einsatz von Werkzeugen zur Unterstützung der Testaktivitäten [74]. Maschinen übernehmen also Aufgaben, die ansonsten manuell von Testern durchgeführt werden, und tragen somit zur Effizienzsteigerung bei. Testautomatisierung kann in verschiedenen Phasen des Testprozesses und in beliebigen Teststufen zum Einsatz kommen.

Häufig wird Testautomatisierung nur im Sinne der automatisierten Ausführung von Testfällen und Test-Suiten verstanden. Testautomatisierung umfasst jedoch auch Ansätze zur werkzeuggestützten Testfallerstellung, Datengenerierung und Ergebnisauswertung. Immer häufiger werden auch Testmanagement-Tätigkeiten durch Werkzeuge unterstützt. Testautomatisierung ist dabei immer als eine Bereicherung zu sehen, keinesfalls jedoch als Ersatz von Expertenwissen [69].

Cleff [10] definiert als Kernziel der Testautomatisierung die Reduktion wiederkehrender und oftmals zu wiederholender manueller Aufgaben. Zusätzlich wird durch die Automatisierung die Wahrscheinlichkeit menschlichen Versagens reduziert. Der Autor weist weiters darauf hin, dass manche Tätigkeiten (wie Lasttests oder die Berechnung der Überdeckungsgrade) überhaupt erst durch den Einsatz von Werkzeugen ermöglicht werden. Obwohl Testautomatisierung als Effizienzsteigerung angesehen wird, muss an dieser Stelle erwähnt werden, dass dies langfristig zu sehen ist. Wird in einem Projekt oder Unternehmen erstmalig eine Automatisierung angestrebt, ist mit hohen Initialkosten zu rechnen. Diese ergeben sich sowohl aus Anschaffungs- oder Lizenzkosten als auch aus der Notwendigkeit von Schulungen und der Anpassung sowie Integration in bestehende Testumgebungen.

3.6.1 Entwicklung automatisierter Testfälle

Der Einsatz von Automatisierung rentiert sich nur, wenn für Stabilität der Testfälle gesorgt ist. Stabil bedeutet in diesem Zusammenhang, dass ein Testfall ausschließlich dann fehlschlägt, wenn er eine Abweichung vom spezifizierten Systemverhalten oder erwarteten Ergebnis detektiert.

Auch die Verwendung von professionellen, extern produzierten Testwerkzeugen ist kein Garant für deren Fehlerfreiheit. Durch die Adaption und Konfiguration der Tools besteht weiters die potenzielle Gefahr der Fehlereinführung. Aus diesem Grund ist die Qualitätssicherung der mit solchen Werkzeugen erstellten Testfälle wichtig. Dadurch kann einerseits sichergestellt werden, dass das Testwerkzeug wie gewünscht arbeitet, andererseits, dass zwischen Design und Umsetzung der Testfälle keine Übertragungsfehler unterlaufen sind. Abbildung 3.4 demonstriert, wie bei der Entwicklung neuer Testfälle mit einem Automatisierungswerkzeug vorgegangen werden sollte. Nach der Erstellung muss jeder Testfall testweise ausgeführt werden. Dabei ist die zwei- oder mehrmalige Ausführung sinnvoll, um auch die Wiederholbarkeit des Testfalls zu prüfen. Erst wenn die korrekte Funktionsweise des Testfalls sichergestellt ist, sollte er in das Set an produktiv eingesetzten Testfällen eingegliedert werden [69].

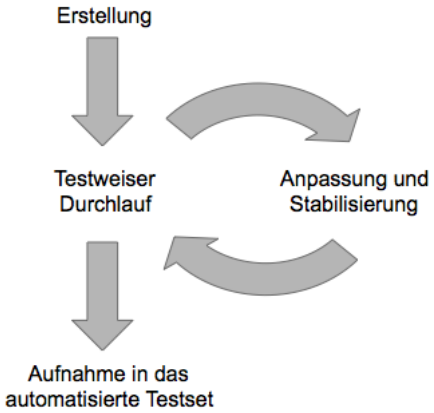


Abbildung 3.4: Vorgehensweise bei der Entwicklung neuer automatisierter Testfälle angelehnt an [69]

3.6.2 Ausführungsarten

Seidl, Baumgartner und Bucsics [69] differenzieren automatisierte Tests nach ihrer Ausführungshäufigkeit in die folgenden drei Kategorien:

Ständige Ausführung: Hierbei handelt es sich meist um Regressionstests auf Modultestebene. Üblicherweise wird ein besonders repräsentatives Subset der verfügbaren Unit-Tests für die ständige Ausführung bestimmt. Unter ständig ist zu verstehen, dass diese Tests regelmäßig in kurzen Zeitabständen, beispielsweise nach jedem Build, ausgeführt werden. Das Konzept der ständigen Ausführung lässt sich daher gut mit Continuous Integration kombinieren. Üblicherweise prüfen ständig ausgeführte Tests nicht die neu hinzugefügte Funktionalität, sondern geben dem Entwickler unmittelbares Feedback, ob seine Änderungen bestehende Teile des Systems beschädigt haben. Diese Ausführungsart eignet sich eher weniger für umfangreiche System- oder Integrationstests, da diese eine zu lange Laufzeit aufweisen.

Häufige Ausführung: Automatisierte Tests dieser Art werden zwar auch regelmäßig, jedoch in längeren Zeitabständen (z.B. alle 3 Tage) und nicht nach jedem Build ausgeführt. Auch hier kommen hauptsächlich Regressionstests zum Einsatz, entweder um das gesamte Set an Modultests auszuführen oder umfangreichere und ressourcenintensivere Integrations- oder Systemtests laufen zu lassen.

Fallweise Ausführung: Es gibt auch automatisierte Tests, für deren Ausführung manuelle Vorarbeit erforderlich ist, so zum Beispiel die Installation des Testobjekts oder händisches Deployment einer Applikation. Tests, die dieser Kategorie angehören, können aufgrund des damit verbundenen Aufwands nicht in regelmäßigen und kurzzeitigen Intervallen ausgeführt werden. Ziel ist, diese Tests zumindest vor jeder Release oder anderen bedeutsamen Zeitpunkten im Verlauf des Projekts durchzuführen. Hauptsächlich handelt es sich hierbei um Tests auf Integrations- oder Systemtestebene.

In obiger Kategorisierung lässt sich erkennen, dass im Zusammenhang mit Testautomatisierung hauptsächlich Regressionstests zum Einsatz kommen. Im Gegensatz zu manuellem Testen sind automatisierte Tests also eine eher konstruktive Tätigkeit, da sie sicherstellen sollen, dass bereits bekannte Fehler nicht erneut eingeführt werden und die bestehende Funktionalität aufrechterhalten bleibt [4].

3.6.3 Risiken und Gefahren beim Einsatz von Testautomatisierung

Berner, Weber und Keller [4] beschreiben ihre Erfahrungen mit Testautomatisierung in der Praxis und zeigen die damit verbundenen Gefahren und Risiken auf. Besonders betont wird dabei die Gefahr der Verwahrlosung von Testfällen und der Testumgebung. Beim Einsatz von Testautomatisierung bedarf es der regelmäßigen Ausführung, Überarbeitung und Wartung der Testfälle und deren Testumgebung. Werden die Testfälle nicht parallel zur Systementwicklung adaptiert, veralten diese, werden inkonsistent und schlagen infolgedessen bei ihrer Ausführung fehl. Abbildung 3.5 zeigt den Verlauf der Verwahrlosung in vier Phasen:

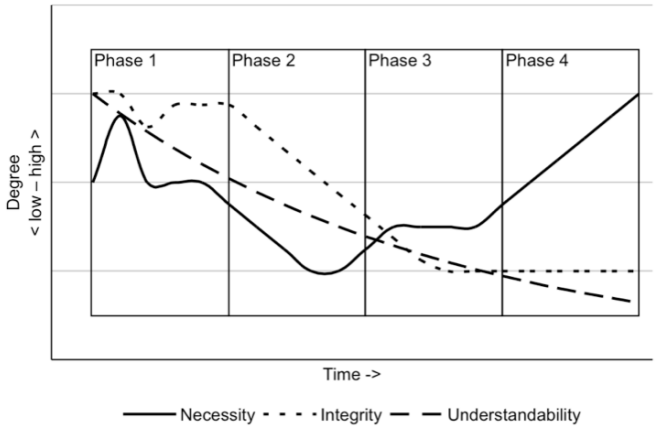


Abbildung 3.5: Verwahrlosung bei Nicht-Wartung der automatisierten Tests
entnommen aus [4]
Die durchgehende Linie beschreibt die Relevanz von Testautomatisierung mit fortschreitendem Entwicklungsstadium des Systems. Die kurz strichlierte Linie stellt die Funktionsfähigkeit der Testfälle dar. Die lang strichlierte Linie steht für die Nachvollziehbarkeit und Verständlichkeit der Testfälle.

Im Folgenden werden die vier Phasen kurz erläutert:

1. **Phase 1:** Die Implementierung automatisierter Testfälle gemäß dem Design wird analog zum Entwicklungsbeginn des Systems durchgeführt. In dieser Phase liegen nur wenige Testfälle vor, diese sind aber aktuell, funktionell und für das Testteam nachvollziehbar. Sie stellen dadurch einen Nutzen für das Projekt dar.
2. **Phase 2:** Die Tester sind nun gut in die verwendeten Werkzeuge eingearbeitet und die Menge an implementierten Testfällen wächst rasch. Die Motivation liegt jedoch eher in der Erstellung neuer statt in der Wartung bereits bestehender Testfälle. Das Scheitern veralteter Testfälle wird ignoriert, das Verständnis der Ursachen für das Fehlschlagen sinkt.
3. **Phase 3:** Die Notwendigkeit an automatisierten Tests steigt, da zu diesem Zeitpunkt im Projektverlauf die fertigen Systemteile integriert werden und somit die Stabilität der Schnittstellen sichergestellt werden muss. Die Verständlichkeit und Funktionalität der automatisierten Testfälle ist so gering, dass keine effektive und effiziente Ausführung mehr möglich ist. Zu diesem Zeitpunkt ist es oftmals nicht länger wirtschaftlich oder zumindest sehr ressourcenintensiv, die veralteten und inkonsistenten Testfälle und deren Testumgebung wiederherzustellen.
4. **Phase 4:** Die Automatisierung ist gescheitert, die zu einem früheren Zeitpunkt erstellten Testfälle sind nutzlos, weil sie entweder fehlschlagen oder ihre Intention nicht mehr nachvoll-

ziehbar ist. Gerade zu diesem Zeitpunkt im Projekt würden funktionsfähige automatisierte Regressionstests aufgrund ihrer Wiederholbarkeit eine hohe Kostenersparnis bringen.

Cleff [10] diskutiert ebenfalls die Risiken, die der Einsatz von automatisierten Tests mit sich bringt. Falsche Erwartungen in Bezug auf den Funktionsumfang der Werkzeuge, blindes Vertrauen in diese oder eine Fehleinschätzung der zur Etablierung von Testautomatisierung benötigten Ressourcen können zum Scheitern des Testprojekts führen oder die Kosten ungeahnt in die Höhe treiben.

3.6.4 Vorgehensweise bei der Auswahl und Einführung eines Testwerkzeugs

Scheffler [68] nennt als Voraussetzung für die erfolgreiche Einführung eines Testautomatisierungswerkzeugs in einem Projekt das Vorhandensein eines funktionierenden und strukturierten Testprozesses, in welchem die Zielsetzungen und Aktivitäten des Testens klar definiert sind. Gemäß dem Autor ist die Vorgehensweise bei der Auswahl und Integration eines neuen Werkzeugs zweistufig und gliedert sich in eine Entscheidungs- sowie eine Einführungsphase. Die Entscheidungsphase umfasst Analyse- und Evaluierungstätigkeiten und hat zum Ziel, das für das Projekt und seinen bestehenden Testprozess am besten geeignete Automatisierungswerkzeug aufzufinden. Die darauf folgende Einführungsphase dient der Vorbereitung der Testumgebung sowie der Installation, Konfiguration und Inbetriebnahme des ausgewählten Werkzeugs.

Nachfolgende Auflistung beschreibt die von Scheffler genannten Tätigkeiten der Entscheidungsphase und ergänzt diese um die von Graham u. a. [23] empfohlenen Schritte bei der Einführung von Testautomatisierung in einem Unternehmen:

- Erkennen des Verbesserungspotenzials: Die Analyse des bestehenden Testprozesses eines Unternehmens oder Projekts dient der Identifikation jener Bereiche, die vom Einsatz eines Werkzeugs zur Unterstützung der Testtätigkeiten profitieren würden.
- Festlegung und Priorisierung von Anforderungen: Die Erstellung eines objektiven Kriterienkatalogs hat die Offenlegung der projekt- und unternehmensspezifischen Anforderungen an ein Testwerkzeug zum Ziel. Es wird empfohlen, die festgelegten Kriterien zu priorisieren und auf deren Basis geeignete Werkzeuge im Rahmen einer Evaluierung zu ermitteln.
- Beurteilung der Eignung: Durch praktische Anwendung der in Frage kommenden Werkzeuge gilt es, ihre Eignung für das Projekt und seine Infrastruktur zu beurteilen und die erforderlichen Anpassungen festzustellen. Die Auswahl eines der Werkzeuge stellt die abschließende Tätigkeit der Entscheidungsphase dar.

Graham u. a. [23] empfiehlt im Anschluss an die zuvor gelisteten Schritte weiters, Dienstleistungsanbieter zu vergleichen, den notwendigen Schulungsbedarf im Testteam zu ermitteln sowie eine Gegenüberstellung von Kosten und Nutzen durchzuführen.

3.7 Testen von Benutzeroberflächen

Die grafische Benutzeroberfläche stellt das Bindeglied in der Interaktion zwischen Mensch und Maschine dar. Neben einer ansprechenden Gestaltung ist vor allem deren korrekte Arbeitsweise für die Bedienung des zugrundeliegenden Systems entscheidend. Daher sollte auch im Zuge des Software-Testens die Überprüfung der GUI keinesfalls vernachlässigt werden [66]. Das Testen von

grafischen Benutzeroberflächen gestaltet sich als komplexe Herausforderung, unabhängig davon, ob funktionale oder nicht-funktionale (wie z.B. Usability oder Performance) Eigenschaften überprüft werden sollen [3].

GUI-Testen als funktionales Testen verfolgt das Ziel, durch Bedienung der Benutzeroberfläche die Funktionsfähigkeit des Gesamtsystems aus der Sicht potenzieller Anwender zu überprüfen. Es handelt sich also nicht um einen isolierten Test der Elemente der Benutzeroberfläche [45]. Es gilt unter anderem festzustellen, ob die Applikation die auftretenden Events und Benutzeraktionen korrekt verarbeitet und die korrespondierenden Reaktionen eintreten [25].

Funktionales GUI-Testen ist nach Memon [41] im Vergleich zu anderen Testarten verhältnismäßig arbeitsintensiv. Die eingesetzten Techniken sind oftmals unstrukturiert, werden ad-hoc und großteils manuell ausgeführt. Ihr Erfolg ist stark abhängig vom Einfallsreichtum und der Erfahrenheit des Testers. Bestehende Ansätze im Software-Testen eignen sich oftmals nur schlecht für das Testen grafischer Benutzeroberflächen, da es nicht zielführend ist, erst am Ende des Tests Ergebniswerte zu vergleichen. Vielmehr muss kontinuierlich nach jeder Interaktion geprüft werden, ob die Screens mitsamt den GUI-Elementen korrekt geladen wurden. Ansätze zur Automatisierung von GUI-Tests befinden sich oft erst im Anfangsstadium.

Ruiz und Price erörtern in [66, 67], warum die Erstellung und Automatisierung von GUI-Tests eine so große Herausforderung darstellt:

- Grafische Benutzeroberflächen weisen einen hohen Grad an Komplexität auf. Dies liegt daran, dass sie versuchen, die Komplexität des zugrundeliegenden Systems vor dem Benutzer zu verbergen und diesem eine möglichst einfache und intuitive Bedienung zu ermöglichen. Sie sind also für die Verwendung durch den Menschen gedacht und eignen sich nur schlecht zur Ansteuerung durch Maschinen und somit für automatisiertes Testen.
- Das Konzept der Isolation in Form von Unit-Tests ist für das Testen von GUIs nicht anwendbar, da eine Einheit auf der Benutzeroberfläche aus mehreren zusammenarbeitenden Komponenten besteht, die wiederum eine Vielzahl an zugrundeliegenden Elementen und Klassen besitzen können.
- Durch den eventgetriebenen Charakter grafischer Benutzeroberflächen ist es erforderlich, dass automatisierte GUI-Tests jene Events simulieren, die sonst durch den Benutzer ausgelöst werden. Die programmatische Simulation von Events ist langwierig und fehleranfällig.
- Die Entscheidung, wann mit dem Testen aufgehört werden kann, gestaltet sich schwierig. Zum einen eröffnet die Benutzeroberfläche eine sehr große Anzahl potenziell möglicher Interaktionsabläufe, die nicht alle durch Tests abgedeckt werden können. Zum anderen ist der Überdeckungsgrad nicht aussagekräftig, da die Menge an ausgeführten Interaktionen nicht zwangsläufig im direkten Verhältnis zu durchlaufenden Zeilen des Programmcodes steht.
- Bei grafischen Benutzeroberflächen ist das Layout mit der Funktionalität verschmolzen. Funktionale GUI-Tests sollten allerdings möglichst robust sein, d.h. Layout- und Designänderungen sollten keinerlei Einfluss auf den erfolgreichen Durchlauf der Testfälle haben. Es muss also eine verlässliche Methode zur Identifikation von GUI-Elementen (wie z.B. Buttons oder Textfelder) ohne Zuhilfenahme ihrer grafischen Position geben.

3.7.1 Techniken zur Umsetzung

Die zwei gängigsten Ansätze zur Erstellung (teil-)automatisierter GUI-Tests liegen im Einsatz eines Capture/Replay-Werkzeugs oder in der skriptbasierten Ansteuerung der Benutzeroberfläche [67]. Reminnyi [62] nennt zusätzlich zu diesen auch noch datengetriebene, schlüsselwortgesteuerte und modellbasierte Techniken. All diese Ansätze haben gemeinsam, dass ihnen ausführbare Skripte zugrunde liegen, sie unterscheiden sich lediglich in der Erstellung, Aufbereitung und Strukturierung dieser [23].

Capture/Replay: Capture/Replay – auch Capture/Playback [38] – bezeichnet Werkzeuge zur Aufzeichnung und wiederholten Wiedergabe aller Events, die durch Interaktion mit der Benutzerschnittstelle eines Systems entstehen. Im Rahmen der Aufzeichnung muss ein menschlicher Benutzer die Interaktionen manuell ausführen, die Wiedergabe kann dann vollkommen automatisiert erfolgen [57].

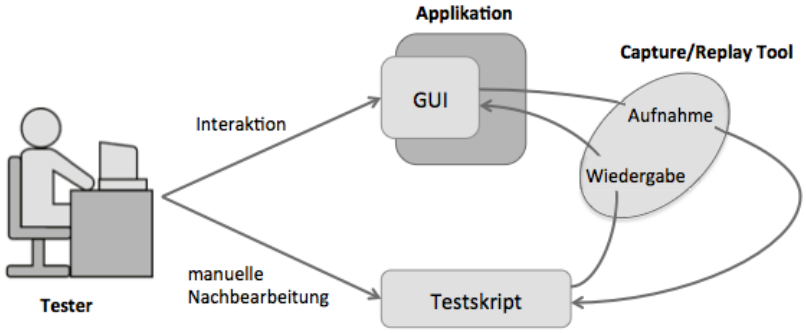


Abbildung 3.6: Funktionsweise eines Capture/Replay-Werkzeugs
angelehnt an [53]

Nach Ostrand u. a. ermöglichen Capture/Replay-Werkzeuge in der Regel neben der Aufnahme und Wiedergabe auch die Nachbearbeitung und Wartung aufgezeichneter Skripte mittels meist herstellerspezifischer Skriptsprachen (siehe auch Abbildung 3.6). Die Verwendung von Capture/Replay gestaltet sich für den Tester einfach, ist jedoch bei einer großen Anzahl zu erstellender Testfälle zeitaufwendig [53].

Ob sich der Einsatz dieser Technik lohnt, ist abhängig vom Entwicklungsstand der Benutzeroberfläche. Für eine zwischenzeitliche Version während der aktiven Entwicklung ist der Wartungsaufwand möglicherweise unverhältnismäßig hoch, da bereits kleine Änderungen eine Überarbeitung oder im schlimmsten Falle die Neuerstellung der betroffenen Skripte erfordern können. Als Grund dafür ist zu nennen, dass die erstellten Skripte einerseits einen zu geringen Abstraktionsgrad haben, andererseits nicht modular aufgebaut sind. Das Fehlen von Modularität führt dazu, dass wiederkehrende Aktionen in jedem Skript enthalten sind und somit separat gewartet werden müssen [57].

Skriptbasierte Testfallerstellung: Dieser Ansatz unterstützt im Gegensatz zu Capture/Replay das Konzept von TDD, da die GUI für die Erstellung der Testfälle nicht notwendigerweise implementiert sein muss. Es werden dabei ausführbare Skripte in einer ausgewählten Programmier- oder Skriptsprache entwickelt, bestehend aus einzelnen Testfällen oder Test-Suiten. Im Rahmen der Implementierung der Testfälle können also alle Vorzüge einer objektorientierten Programmiersprache wie Generalisierung oder Modularisierung genutzt werden, was zu

einem höheren Grad an Wiederverwendbarkeit beiträgt und somit die Wartung einfacher gestaltet [67].

Es muss jedoch auf einen möglichst hohen Level an Abstraktion geachtet werden, da ansonsten ähnlich zum Capture/Replay-Ansatz bereits bei kleinen Änderungen Anpassungen notwendig werden. Üblicherweise erfolgt die Implementierung der Testfälle unter Verwendung eines Application Programming Interface (API), über das die GUI-Elemente angesteuert und kontrolliert werden können. Es ist zu berücksichtigen, dass für die erfolgreiche Ansteuerung ein stabiles Auffinden der GUI-Elemente unbedingt erforderlich ist [73].

Abschließend ist festzuhalten, dass programmatisch erstellte Testskripte in der Regel einfacher und kostengünstiger zu warten sind, sich deren Erstellung allerdings weitaus schwieriger gestaltet und fundierte Programmierkenntnisse erfordert [67].

Unabhängig vom gewählten Erstellungsansatz kann eine datengetriebene Vorgehensweise zur Anwendung kommen [69]. Die Kernidee liegt dabei in der separaten Abspeicherung der Testdaten, also der Eingabewerte und erwarteten Ausgaben zu einem Testfall. Durch diese Vorgehensweise wird Unabhängigkeit zwischen dem Testskript und den Testdaten erreicht. Ein datengetriebener Ansatz rentiert sich besonders dann, wenn dieselbe Abfolge an Interaktionen mit einer großen Menge an unterschiedlichen Daten mehrmals durchlaufen werden soll.

Beim schlüsselwortgetriebenen Testen werden neben den Testdaten zusätzlich auch Schlüsselwörter separat abgespeichert. Diese stehen für konkrete Aktionen (z.B. Klick auf einen Button) und erlauben, die im Rahmen eines Tests auszuführenden Anweisungen auch ohne Programmierkenntnisse festzulegen. Das Set an verfügbaren Schlüsselwörtern sowie deren Bedeutung und Nutzungskontext müssen dem Tester bekannt sein [23].

Modellbasierte Techniken wurden nur der Vollständigkeit halber erwähnt, werden aber aufgrund der geringen Relevanz für diese Arbeit an dieser Stelle nicht näher ausgeführt. Ausführliche Informationen dazu finden sich in [23, 69, 73].

4 Analyse

Dieses Kapitel vermittelt einen Überblick über jene javabasierte Applikation, die ein repräsentatives Fallbeispiel für die Untersuchung bestehender Testautomatisierungswerkzeuge darstellt und auf deren Basis in einem späteren Arbeitssschritt ihre Evaluierung stattfinden soll. Nach Vorstellung des Entwicklungs- und Testprozesses des Projekts wird die Relevanz dieser Arbeit begründet und abschließend ein Kriterienkatalog erarbeitet, der sowohl projektspezifische als auch allgemeine Anforderungen an die Testautomatisierung einer Rich-Client-Applikation definiert. Die für dieses und nachfolgende Kapitel gewählte Vorgehensweise orientiert sich dabei an dem in Abschnitt 3.6.4 erläuterten Prozess zur Einführung eines neuen Testautomatisierungswerkzeugs in einem Projekt.

4.1 Umfeldbeschreibung der bestehenden Applikation

Die Fragestellung dieser Arbeit wird anhand eines repräsentativen Fallbeispiels untersucht. Dabei handelt es sich um ein umfangreiches Softwareprojekt, das sich mit der (Weiter-)Entwicklung eines javabasierten Systems zum Event-Management von Sportveranstaltungen wie Turnieren oder Seminaren befasst und das aufgrund seiner Größe und Komplexität sowie der verwendeten Technologien besonders gut zur Bewertung der Praxistauglichkeit bestehender Testwerkzeuge geeignet scheint. Die Applikation unterstützt sämtliche Aktivitäten der Planung, Durchführung und Nachbereitung von Sportereignissen und wird derzeit in über 130 Ländern weltweit eingesetzt.

Das Gesamtsystem besitzt eine modulare Architektur und besteht aus den folgenden drei Hauptkomponenten: einer umfangreichen Rich-Client-Anwendung, einer Web-Applikation sowie einer mobilen App. Die Kernfunktionalität stellt der offlinefähige Java-Swing-Client zur Verfügung, welcher je nach Bedarf um zusätzliche Module auf einem Plugin-basierten Ansatz erweitert werden kann. Offlinefähigkeit ist ein besonders wichtiges Kriterium, da der Rich-Client auch an Veranstaltungsorten ohne Verfügbarkeit des Internets verwendbar sein muss. Dennoch bestehen potentielle Abhängigkeiten zwischen dem Client und den im Web befindlichen Komponenten des Gesamtsystems, da die gemeinsame Datenbasis ausgetauscht und synchronisiert werden muss. Je nach Konfiguration kann eine interne, eine lokal vernetzte oder eine im Internet befindliche Datenbank zum Einsatz kommen.

Aufgrund seiner globalen Verbreitung sind alle Komponenten des Systems internationalisiert und lokalisiert, es ist also die Verwendung in verschiedenen Sprachen, Kulturen und Zeitzonen möglich. Der Swing-Client kann weiters auf allen Plattformen genutzt werden und bietet dem Benutzer die Wahl zwischen verschiedenen Look&Feels.

4.1.1 Entwicklungsprozess

Das Projektteam besteht insgesamt aus zwölf Personen und verfolgt einen agilen Entwicklungsansatz, der sich am Scrum-Vorgehensmodell orientiert. Ein Großteil des bisherigen Entwicklungsaufwands wurde in die Erstellung der Client-Applikation und seiner Erweiterungsmodule investiert (ca. 55%), während für die Web-Anwendung nur mäßig viel (ca. 35%) und für die mobile App bisher noch wenig Zeit (ca. 10%) aufgewandt wurde.

Die laufende Weiterentwicklung und Verbesserung des bestehenden Gesamtsystems erfolgt in 14-Tagen-Sprints. Welche Funktionen in den jeweiligen Sprint-Backlog aufgenommen werden, ergibt sich durch die Einholung und anschließende Priorisierung des Feedbacks der Kunden.

Zur Programmierung wird die Open-Source-Entwicklungsumgebung Eclipse[1] eingesetzt. Auch bei allen anderen im Rahmen dieses Projekts verwendeten Tools oder Frameworks wird aus Gründen der Kostenreduktion und Erweiterbarkeit auf Open-Source-Lösungen gesetzt. Zur Versionsverwaltung des erstellten Codes wird ein zentrales git-Repository[2] herangezogen, welches an den Continuous Integration-Server Jenkins[3] angebunden ist. Somit können täglich automatisierte Builds der aktuellen Version erstellt werden.

4.1.2 Testprozess

Die Vorgehensweise beim Testen orientiert sich in diesem Projekt am fundamentalen Testprozess, der in Abschnitt 3.5 näher erläutert wurde. Dieser ist projektspezifisch auf die agile Vorgehensweise des Scrum-Entwicklungsmodells abgestimmt und fällt aufgrund des eher kleinen Teams weniger formal aus. Der Fokus liegt also mehr auf der Konzeption und Umsetzung der Testfälle und weniger auf der Erstellung umfangreicher Dokumente. In jedem Sprint werden alle Phasen des Testprozesses durchlaufen.

Das Testteam besteht aus fünf Personen, wobei die Verantwortlichkeiten folgendermaßen aufgeteilt sind: jeweils zwei Personen sind für das Testen von Client- und Web-Applikation zuständig, eine Person für das Testen der mobilen App. Ansätze zur Testautomatisierung sind komponentenabhängig unterschiedlich stark vertreten. Das Testen der mobilen App geschieht vollständig automatisiert durch Verwendung des Tools Cucumber[4]. Auch die Webapplikation setzt zu einem hohen Grad auf automatisiert ausführbare Tests, wobei hierfür Cucumber sowie das plattformunabhängige Browser-Testframework Selenium[5] zum Einsatz kommen. Der Swing-Client und seine clientbasierten Zusatzmodule werden zurzeit ausschließlich durch manuelle Tests geprüft.

Für jeden Sprint wird ein separater Testplan erstellt, in dem die Aufgaben und Verantwortlichkeiten spezifiziert werden. Die Tester sind gut in den Entwicklungsprozess integriert und wichtiger Bestandteil der Projektmeetings. Sie sind mit der Konzeption, Implementierung und Durchführung der Testfälle betraut, wobei abhängig vom Grad der Automatisierung mit den Testtätigkeiten parallel zur Entwicklung oder erst im Anschluss an diese begonnen werden kann. Vor allem im Rahmen des Testens der Swing-Client-Applikation fällt ein Großteil der Arbeit erst gegen Ende eines Sprints an.

Zur Unterstützung der Testtätigkeiten kommt ein Testmanagement-Tool zum Einsatz, konkret fiel in diesem Projekt die Wahl auf das Open-Source-Werkzeug Squash[6]. Testfälle können in diesem Tool bereits in einem frühen Stadium der Entwicklung spezifiziert werden, die tatsächliche Verlinkung zur Implementierung kann zu einem späteren Zeitpunkt erfolgen. Testfälle werden in diesem Projekt abhängig von der zu testenden Funktionalität entweder neu erstellt oder aus vorhergehenden Sprints wiederverwendet. Zusätzlich bietet dieses Testmanagement-System die Möglichkeit zur Interaktion mit Jenkins und somit zur automatisierten Ausführung der mit Selenium sowie Cucumber erstellten Testfälle.

[1] http://www.eclipse.org
[2] http://git-scm.com
[3] http://jenkins-ci.org
[4] http://cukes.info
[5] http://www.seleniumhq.org
[6] http://www.squashtest.org

4.2 Motivation und Themenrelevanz

Ziel dieser Arbeit ist es, umfassende Erkenntnisse hinsichtlich verfügbarer Open-Source-Werkzeuge und praktikabler Lösungsansätze für die Automatisierung des GUI-basierten Testens von Rich-Client-Applikationen zu erlangen. Die Relevanz dieses Themas lässt sich aus zwei Blickwinkeln betrachten:

Steigende Verfügbarkeit von Open-Source-Produkten: Die Open-Source-Gemeinschaft hat in den vergangenen 15 Jahren kontinuierlich an Bedeutung gewonnen. Trotz der Tatsache, dass bei Open-Source-Projekten viele Personen in einem ungeplanten, strukturlosen Entwicklungsprozess über hohe geografische Distanzen zusammenarbeiten, entstehen Produkte, deren Qualität und Funktionsumfang an die traditionell entwickelter, kommerzieller Software heranreichen [43]. Mittlerweile existiert eine unüberschaubare Anzahl an Open-Source-Lösungen für die verschiedensten Problemstellungen und Anwendungsgebiete, alleine auf dem Gebiet des funktionalen Testens listet Aberdour [1] insgesamt 128 verschiedene Werkzeuge.

Gerade aufgrund der steigenden Bedeutung und der enormen Produktvielfalt ist es wichtig, bestehende Open-Source-Lösungen konkreter Nutzungskontexte hinsichtlich ausgewählter Qualitätskriterien zu prüfen. Erkenntnisse über die Qualität und Praxistauglichkeit von Open-Source-Werkzeugen für das GUI-basierte Testen von Rich-Client-Applikationen können Testbeauftragten in Projekten, wo kein Budget für teure Lizenzen vorliegt, als Entscheidungshilfe dienen. Dieser Nutzungskontext ist besonders deshalb von Interesse, da vorangegangene Evaluierungen auf diesem Gebiet entweder aus heutiger Sicht veraltete Produktversionen betrachten oder sich nur auf den Vergleich einiger weniger Werkzeuge beschränkt haben. So werden von Nedyalkova und Bernardino [47] nur fünf bestehende Open-Source-Testwerkzeuge analysiert, während die relativ umfangreiche Arbeit von Jovic u. a. [32] bereits vor über drei Jahren durchgeführt wurde.

Mehrwert für reales Projekt: Aus der Umfeldbeschreibung des bestehenden Softwaresystems geht hervor, dass beim Swing-Client die Vorgehensweise beim Testen trotz der Tatsache, dass er die wichtigste und umfangreichste Komponente darstellt, durchaus Verbesserungspotenzial aufweist. Wiederholtes manuelles Testen ist zeit- und ressourcenintensiv, zusätzlich können Fehler im Produkt durch menschliche Unachtsamkeit übersehen werden. Letztere Problematik wird bei diesem Projekt noch zusätzlich verstärkt durch den hohen Zeitdruck, unter dem die manuellen Tests am Ende jedes Sprints durchgeführt werden müssen. Hinzu kommt, dass durch das späte Testen keinerlei Zeit für die Behebung der gefundenen Fehler innerhalb des laufenden Sprints verbleibt und sich die Abnahme des betroffenen Features somit auf den nächsten Sprint verschiebt.

Umfangreiche und hochwertige Qualitätssicherung ist bei einer Client-Applikation besonders wichtig, da die Auslieferung einer fehlerbehafteten Version weitaus drastischere Konsequenzen als beispielsweise bei einer Webapplikation haben kann. Eine solche liegt üblicherweise an einem oder einigen wenigen zentralen Servern und kann mit relativ geringem Aufwand durch eine neue Version ausgetauscht werden. Im Falle des global verbreiteten Swing-Clients liegt das Installieren einer neuen Version außerhalb der Reichweite des Herstellers, da die Software lokal am System der jeweiligen Kunden läuft. Eine Rückhol- oder Updateaktion ist somit mit hohen Kosten verbunden.

Um langfristig Ressourcen einzusparen und einen höheren Level an gesicherter Qualität in der Rich-Client-Komponente des als Fallbeispiel gewählten Systems zu erhalten, bietet sich der Einsatz eines Werkzeugs an, das die Umsetzung von GUI-Testfällen unterstützt

und diese automatisiert ausführen kann. Wird im Rahmen dieser Arbeit ein geeignetes Testwerkzeug für die bestehende Rich-Client-Applikation aufgefunden, so kann aufgrund ihres repräsentativen Charakters davon ausgegangen werden, dass dieses Werkzeug auch für andere reale Softwareprojekte, die einen ähnlichen Testprozess verfolgen bzw. ähnliche Technologien aufweisen, einen Mehrwert darstellt.

4.3 Kriterienkatalog

Damit eine fundierte Evaluierung der potenziell in Frage kommenden Werkzeuge durchgeführt werden kann, ist die Ausarbeitung eines Kriterienkatalogs erforderlich. Dieser soll einerseits identifizieren, welche Anforderungen und Einschränkungen sich aus dem bestehenden Softwareprojekt ergeben und andererseits definieren, welche allgemeinen Eigenschaften ein Werkzeug erfüllen muss, um in die nähere Auswahl aufgenommen zu werden.

Nachfolgende Aufzählung listet die Anforderungen aus projektspezifischer Sicht:

1. **Open-Source:** Wie bereits in Abschnitt 4.1.1 angesprochen, wird im Entwicklungsprozess des Projekts ausschließlich auf freie, quellcodeoffene Produkte gesetzt. Diese Anforderung gilt auch für ein Testautomatisierungswerkzeug.
2. **Swing-Unterstützung:** Da die Benutzeroberfläche der Client-Komponente des Projekts mit Swing implementiert ist, muss das Testautomatisierungswerkzeug Funktionen zur Erstellung und Ausführung von GUI-Tests für Swing-Applikationen bereitstellen.
3. **Integrationstauglichkeit:** Das Werkzeug muss in den bestehenden Testprozess und die Toollandschaft des Projekts integriert werden können. Darunter ist einerseits die Verlinkung und Einbindung der Testfälle in Squash, andererseits die von Jenkins angesteuerte, automatisierte Ausführung der Testfälle im Rahmen der Erstellung eines neuen Builds zu verstehen. Es sollte keinerlei menschliches Eingreifen (z.B. Starten des Tools) zur Durchführung der Tests erforderlich sein.
4. **Datengetriebenes Testen:** Das Werkzeug sollte die im Rahmen der Ausführung von Testfällen benötigten Eingabe- und Ausgabedaten aus einer separaten Testdatenbank entnehmen können. Es soll also ein datengetriebener Ansatz verfolgt werden (siehe dazu Abschnitt 3.7.1).
5. **Einfache Erstellung:** Nach Möglichkeit sollte das Testautomatisierungswerkzeug keine Programmierkenntnisse in einer herstellerspezifischen, proprietären Skriptsprache voraussetzen. Für die Erstellung von Regressionstests wäre ein Capture/Replay-Ansatz optimal, dessen aufgezeichnete Skripte in einer weit verbreiteten Sprache (z.B. Java) erweitert bzw. angepasst werden können. Für das Testen neuer Funktionalität innerhalb eines Sprints scheint ein skriptbasierter Ansatz sinnvoll, um bereits parallel zur Entwicklung Testfälle umsetzen zu können.

Zusätzlich wurden die nachfolgenden allgemeinen Kriterien definiert:

1. **Aktive Weiterentwicklung:** Optimal wäre es, wenn das Testautomatisierungswerkzeug aktiv weiterentwickelt und Support-Möglichkeiten angeboten würden. Dies ist wichtig, damit die Lösung langfristig eingesetzt werden kann und sichergestellt ist, dass auch neuere oder

zukünftige Swing-Versionen unterstützt werden. Gleichzeitig sind betreute Foren oder andere Support-Formen wichtig, damit man im Falle komplexer Aufgabenstellungen mit Benutzern oder Entwicklern in Kontakt treten kann.

2. **Dokumentationsmaterialien:** Ein weiteres entscheidendes Kriterium ist die Verfügbarkeit von Dokumentation, Tutorials sowie Codebeispielen. Diese werden benötigt, um einen Überblick über ein Werkzeug zu erhalten und entscheiden zu können, ob es die definierten Kriterien erfüllt. Weiters ist das Vorliegen einer qualitativ hochwertigen Dokumentation ausschlaggebend für ein reibungsloses, schnelles Setup sowie die korrekte Anwendung.

3. **Plattformunabhängigkeit:** Das Werkzeug sollte unabhängig vom zugrundeliegenden Betriebssystem sein, sodass das Entwicklungs- bzw. Testteam nicht an bestimmte Plattformen gebunden ist und die für die Erstellung eines Testfalls eingesetzte Plattform nicht zwingend jene der Ausführung sein muss.

4. **Fehleranalyse:** Das Werkzeug sollte den Tester bei der Fehleranalyse unterstützen. Dies bedeutet, dass im Falle fehlgeschlagener Tests Informationen bereitgestellt werden, in welcher Zeile der Ausführung des Tests die Abweichung aufgetreten und inwiefern das tatsächliche Ergebnis vom erwarteten abgewichen ist (dies kann nicht nur schriftlich, sondern auch in Form eines Screenshots erfolgen). Es muss für die Testperson nachvollziehbar werden, warum ein Test fehlgeschlagen ist. Im Idealfall wird das Debuggen eines Testfalls unterstützt.

5. **Reporting:** Nach Durchlauf eines oder mehrerer Tests soll ein High-Level-Report verfügbar sein, der einen Überblick über die getesteten Artefakte liefert und die Anzahl an erfolgreichen/fehlgeschlagenen Testfällen aufzeigt. Optimal ist ein Report in XML- oder HTML-Format.

6. **Reifegrad:** Der Reifegrad macht eine Aussage darüber, wie lange ein Werkzeug bereits am Markt verfügbar und wie hoch sein Bekanntheitsgrad ist. Ein Werkzeug, das schon über Jahre hinweg entwickelt wird, läuft möglicherweise stabiler, besitzt mehr Dokumentation und Funktionsumfang und es liegen häufig Erfahrungsberichte von anderen Testern vor.

Die Bezeichnung für die Kriterien vier bis sechs wurde aus dem Kriterienkatalog der Quality First Software GmbH [61] entnommen. Dieser Katalog definiert eine umfangreiche Sammlung an Kriterien, die jedoch großteils erst nach Anwendung eines Werkzeugs beurteilt werden können (Bewertung unterschiedlicher Aspekte der Benutzerfreundlichkeit, Testdurchführung und Installation).

5 Evaluierung

Dieses Kapitel dokumentiert die Vorgehensweise bei der Evaluierung verfügbarer Open-Source-Testautomatisierungswerkzeuge. Insbesondere werden die Ergebnisse der Auswertung des im vorhergehenden Kapitel definierten Kriterienkatalogs präsentiert und die Wahl der Werkzeuge begründet. Abschließend werden Informationen hinsichtlich Entstehung, Aufbau sowie Funktionsweise der ausgewählten Werkzeuge bereitgestellt.

5.1 Marktanalyse

Den Beginn der Evaluierung bildete die Durchführung einer zweistufigen Marktanalyse. In einem ersten Schritt wurde das Ziel verfolgt, einen Überblick über die am derzeitigen Markt verfügbaren Werkzeuge zum automatisierten Testen von javabasierten Rich-Client-GUIs zu erhalten. Anschließend wurde die Auswertung der in Abschnitt 4.3 definierten Kriterien auf den identifizierten Werkzeugen angestrebt. Eine solche Marktanalyse war erforderlich, damit eine fundierte, nachvollziehbare Entscheidung getroffen werden konnte, welche der Werkzeuge prototypisch eingesetzt werden sollen.

Unter Zuhilfenahme der Webseiten von Aberdour [1], Java-Source [52] sowie Wikipedia [76] konnten 23 potenziell geeignete Werkzeuge ermittelt werden. Die nachfolgenden Abschnitte 5.1.1 und 5.1.2 dokumentieren die nähere Analyse dieser Werkzeuge und die Vorgehensweise bei der Auswertung des Kriterienkatalogs.

5.1.1 Recherche und Informationsaufbereitung

Für jedes der 23 zuvor identifizierten Werkzeuge wurde eine ausführliche Recherche durchgeführt, welche der Sammlung und Visualisierung von Informationen zu den verschiedenen Anforderungen des Kriterienkatalogs diente. Im Zuge dieser Recherche galt es einerseits, die Repräsentation der Werkzeuge im Web (Produktbeschreibung, Produktdokumentation, Foren, Projekt- sowie Code-Repositories), andererseits Resultate vorangegangener wissenschaftlicher Evaluierungen und Abhandlungen (siehe [32, 47, 36]) zu betrachten. Die gesammelten Informationen zu den Werkzeugen wurden in einer Matrix (vertikal: Auflistung der Tools, horizontal: Auflistung der Kriterien) visualisiert und können dem Anhang (siehe A.1) entnommen werden.

5.1.2 Kriterienauswertung

Auf Basis der erstellten Matrix wurde im nächsten Schritt eine Auswertung der Kriterien für die einzelnen Werkzeuge durchgeführt. Zunächst wurde hierfür ein Bewertungsschema für die Kriterien definiert, welches in nachfolgender Tabelle 5.1 abgebildet ist.

Anschließend wurden acht der 23 betrachteten Werkzeuge vorzeitig eliminiert, da sie bereits grundsätzliche Anforderungen (wie Open-Source bzw. Java-Swing-Unterstützung) nicht erfüllten und somit eine weitere Auswertung der übrigen Kriterien nicht zweckmäßig schien.

Symbol	Bedeutung
✓+	das Kriterium ist besonders herausragend erfüllt
✓	das Kriterium ist vollständig erfüllt
~	das Kriterium ist teilweise oder nur unter bestimmten Bedingungen erfüllt
?	es liegen zu wenig oder keine Informationen zu diesem Kriterium vor
×	das Kriterium ist nicht erfüllt

Tabelle 5.1: Schema zur Auswertung der Kriterien

Nachfolgende Tabelle 5.2 stellt die Resultate der Auswertung der projektspezifischen Kriterien (Open-Source, Swing-Unterstützung, Integrationstauglichkeit, datengetriebenes Testen, einfache Erstellung) für die verbleibenden 15 Werkzeuge gemäß dem zuvor definierten Bewertungsschema dar. Aus Gründen der einfacheren Auswertbarkeit wurde das Kriterium „einfache Erstellung“ in die drei Aspekte „Capture-Replay-Ansatz“, „skriptbasierte Testfallerstellung“ sowie „standardisierte Sprache“ unterteilt.

	Open-Source	Swing-Unterstützung	Datengetrieben	Integrationstauglichkeit	Capture-Replay	Skriptbasierte Erstellung	Standardisierte Sprache	✓	~	?	Ranking
FEST	✓	✓	✓	✓	×	✓	✓	6	0	0	1.
Marathon	✓	✓	✓	✓	✓	×	✓	6	0	0	1.
UISpec4J	✓	✓	✓	✓	×	✓	✓	6	0	0	1.
Abbot	✓	✓	~	~	✓	✓	✓	5	2	0	2.
jfcUnit	✓	✓	~	?	✓	✓	✓	5	1	1	3.
Maveryx	✓	✓	✓	?	×	✓	✓	5	0	1	4.
Sikuli	✓	✓	?	✓	×	✓	✓	5	0	1	4.
Robot Framework	✓	✓	✓	✓	×	✓[1]	×	5	0	0	5.
WindowTester Pro	✓	✓	?	~	✓	×	✓	4	1	1	6.
Pounder	✓	✓	?	~	✓	×	✓	4	1	1	6.
Eclipse Jubula	✓	✓	✓	~	✓[2]	×	×	4	1	0	7.
Jemmy	✓	✓	?	?	×	✓	✓	4	0	2	8.
Jacareto	✓	✓	?	?	✓	×	?	3	0	3	9.
GTT	✓	✓	?	?	✓	×	?	3	0	3	9.
GUITAR	✓	✓	×	?	✓	×	×	3	0	1	10.

Tabelle 5.2: Auswertung der projektspezifischen Kriterien

[1] Alternative Funktionsweise: Testfälle werden schlüsselwortgetrieben erstellt
[2] Alternative Funktionsweise: Testfälle werden per Drag&Drop aus vordefiniertem Pool gezogen

Die Einträge in der Tabelle sind absteigend nach Ranking sortiert. Dieses macht darüber eine Aussage, wie gut ein Werkzeug in Bezug auf die projektspezifischen Kriterien abgeschnitten hat und ergibt sich aus den grau eingefärbten Spalten, welche die Anzahl an ✓, ~ und ? pro Zeile listen.

Im Anschluss wurden auch die allgemeinen Kriterien (aktive Weiterentwicklung, Verfügbarkeit von Dokumentation, Plattformunabhängigkeit, Fehleranalyse, Reporting, Reifegrad) ausgewertet. Dazu wurde erneut, sofern anwendbar, auf das zuvor definierte Bewertungsschema zurückgegriffen. Jene Kriterien, die sich nicht auf dieses Schema abbilden ließen, wurden in textbasierter Form angeführt. Nachfolgende Tabelle 5.3 listet die Resultate dieser Auswertung:

	Aktive Entwicklung	Dokumentation	Plattformunabhängigkeit	Fehleranalyse	Reporting	Reifegrad	✓[1]	~	?
FEST	✓	✓+	✓	✓+	✓	2007	7	0	0
Marathon	✓	✓+	✓	✓	✓	2002	6	0	0
UISpec4J	?	✓	✓	✓	✓	2009	4	0	1
Abbot	✓	~	✓	✓	✓	2003	4	1	0
jfcUnit	×	×	✓	~	?	2001	1	1	1
Maveryx	?	✓+	✓	✓+	✓	2011	6	0	1
Sikuli	✓	✓	✓	?	×	2010	3	0	1
Robot Framework	✓	✓+	✓	✓	✓	2008	6	0	0
WindowTester Pro	✓	✓+	✓	✓	✓	2012	6	0	0
Pounder	×	~	?	~	?	2002	0	2	2
Eclipse Jubula	✓	✓+	✓	✓	✓	2011	6	0	0
Jemmy	✓	×	✓	✓	×	2009	3	0	0
Jacareto	?	~	✓	✓	✓	2005	3	1	1
GTT	?	×	?	?	?	2006	0	0	4
GUITAR	✓	✓+	~	✓	?	2009	4	1	1

Tabelle 5.3: Auswertung der allgemeinen Kriterien

Für diese Tabelle wurde bewusst dieselbe Sortierung wie für Tabelle 5.2 gewählt, die Reihenfolge der Werkzeuge in der Tabelle sagt also nichts über deren Qualität in Bezug auf die ausgewerteten Kriterien aus.

[1] ✓+ zählt wie zwei einzelne ✓und wird aus Gründen der Übersichtlichkeit im Wert für ✓mitaddiert

5.2 Auswahl der Werkzeuge

Im Anschluss an die Hintergrundrecherche und Kriterienauswertung galt es, sechs der 15 analysierten Werkzeuge für die spätere Verwendung auszuwählen. Um einen optimalen Lösungsansatz für das bestehende reale Projekt zu erarbeiten, schien es sinnvoll, bei der Festlegung der Werkzeuge besonders auf die Erfüllung der projektspezifischen Kriterien sowie auf Resultate vorangegangener Evaluierungen zu achten. Die Auswertung der allgemeinen Kriterien wurde als zusätzliche Hilfestellung für den Entscheidungsprozess betrachtet.

Nachfolgende Auflistung dokumentiert und begründet die Wahl der Werkzeuge:

FEST: Die Entscheidung für FEST wurde getroffen, da dieses Werkzeug im Ranking von Tabelle 5.2 den ersten Platz erreichte und zusätzlich bei der Auswertung der allgemeinen Kritierien überdurchschnittliche Ergebnisse erzielte. Weiters wird FEST auch in der Evaluierung von Kleuker [36] als empfehlenswerte Software bezeichnet.

Marathon: Auch Marathon konnte alle projektspezifischen Anforderungen erfüllen und schnitt gut in der Bewertung der allgemeinen Kriterien ab. Die Auswahl von Marathon schien weiters besonders deshalb interessant, da hier bei vorangegangenen Evaluierungen sehr unterschiedliche Ergebnisse erbracht wurden. So erzielt Marathon (Version 2.0b4) in der Arbeit von Jovic u. a. [32] – durchgeführt 2010 – äußerst negative Resultate. Von acht mit diesem Werkzeug durchgeführten Aufgabenstellungen konnte nur eine erfolgreich absolviert werden. Dieses Ergebnis veranlasste die Autoren, Marathon nicht weiter zu analysieren. Weitaus positiver fällt die Analyse von Kleuker [36] – ausgeführt 2011 – aus. Gemäß dem Autor überzeugt Marathon besonders durch seine einfache und intuitive Verwendung sowie stabile Funktionalität. Zusätzlich wird darauf hingewiesen, dass die betrachtete Version (3.1.1) über einen neuen Capture-Modus verfüge, wodurch nun auch komplexere Swing-Komponenten erfasst werden könnten. Eine weitere Evaluierung von Nedyalkova und Bernardino [47] aus 2013 bestätigt, dass Marathon sehr benutzerfreundlich, einfach und gut dokumentiert ist. Gleichzeitig wird aber auch aufgezeigt, dass Marathon viele Events bzw. Swing-Komponenten nach wie vor nicht optimal verarbeiten kann.
Betrachtet man die Jahreszahlen, in denen die jeweiligen Untersuchungen ausgeführt wurden, ist zu vermuten, dass die konträren Resultate ihren Ursprung in der Analyse unterschiedlicher Versionen nahmen und sich die Qualität von Marathon über die letzten Jahre verbessert hat. Unter Berücksichtigung dieser Vermutung ist eine neuerliche Betrachtung vor allem deshalb besonders interessant, da im November 2013 eine neue Version von Marathon veröffentlicht wurde [33].

Abbot: Dieses Werkzeug wurde in die Auswahl aufgenommen, da es als eines der wenigen Werkzeuge die Besonderheit aufweist, sowohl Capture-Replay als auch eine skriptbasierte Vorgehensweise zur Erstellung der Testfälle zu unterstützen. Es kann somit später ein Vergleich der beiden Konzepte durchgeführt werden, um die Vor- und Nachteile beider Ansätze gegenüberzustellen. Neben Abbot bietet nur jfcUnit, das im Ranking Platz drei erreichte, ebenfalls beide Erstellungsformen an. Von der Auswahl dieses Werkzeugs wurde aber aufgrund der schlechten Ergebnisse in Tabelle 5.3 sowie einheitlich negativer Bewertungen vorangegangener Evaluierungen der letzten sieben Jahre abgesehen.

Sikuli und Robot Framework: Platz vier im Ranking belegten Maveryx sowie Sikuli. Obwohl Maveryx im direkten Vergleich bei den allgemeinen Kriterien deutlich besser abschnitt, wurde an dieser Stelle die Verwendung von Sikuli bevorzugt. Dies lässt sich damit begründen, dass

Sikuli einen einzigartigen Ansatz zur Erstellung von Testfällen verfolgt, der auf Verwendung einer Screenshot-Technik basiert. Zusätzlich bietet Sikuli gemäß Moroz [44] die Möglichkeit zur Kopplung mit dem Robot Framework, wodurch es um datengetriebene Testansätze und Reporting-Funktionen erweitert werden kann.

WindowTester Pro: Als letztes Werkzeug wurde WindowTester Pro gegenüber UISpec4J bevorzugt, obwohl letzteres ebenfalls einen ersten Platz im Ranking belegte. Grund dafür war, dass WindowTester Pro einen Capture-Replay-Ansatz verfolgt (im Gegensatz zu UISpec4J), und mit FEST, Abbot und Sikuli ohnehin schon drei Werkzeuge mit skriptbasiertem Erstellungsansatz zur Betrachtung ausgewählt waren. Ein weiteres Argument gegen UISpec4J war, dass nicht eindeutig bestimmt werden kann, ob dieses Tool noch aktiv weiterentwickelt und supportet wird.

5.3 Beschreibung der gewählten Werkzeuge

Die nachfolgenden Abschnitte 5.3.1 bis 5.3.6 bieten einen Einblick in die im vorhergehenden Schritt ausgewählten Werkzeuge und deren Funktionsweise.

5.3.1 FEST

Der Grundstein für FEST wurde in einem Open-Source-Projekt der Entwickler und Forscher Ruiz und Price im Jahr 2007 gelegt. Das Projekt mit der Bezeichnung testng-abbot verfolgte ursprünglich die Idee, die von Abbot (siehe Abschnitt 5.3.3) bereitgestellte API zur skriptbasierten Erstellung von GUI-Tests an das Testframework TestNG[1] zu koppeln, das gemäß den Autoren zum damaligem Zeitpunkt weitaus mächtiger als sein Konkurrent JUnit[2] war und unter anderem die folgenden zusätzlichen Funktionen bot [67]:

- Erstellung parametrisierter Tests
- Verwendung eines datengetriebenen Ansatzes
- Vielfältigere Möglichkeiten zur Gruppierung von Testfällen in Bezug auf deren Ausführung

Nach Einstellung der aktiven Weiterentwicklung von testng-abbot wurden die gewonnenen Erkenntnisse für die Realisierung von FEST eingesetzt [64]. Bei FEST (Fixtures for Easy Software Testing) handelt es sich um eine Sammlung von Java-Bibliotheken zur vereinfachten Erstellung von funktionalen GUI-Tests. Das Projekt ist unter der Apache Lizenz Version 2.0 veröffentlicht und besteht aus neun aktiven Mitgliedern.

Die Hauptbestandteile von FEST sind zwei Bibliotheken zur skriptbasierten Erstellung von GUI-Tests für Swing- bzw. JavaFX[3]-Applikationen. Das Swing-Modul kann dabei nicht nur zum Testen von Desktop-Anwendungen sondern auch für die Überprüfung von Java-Applets eingesetzt werden. Das JavaFX-Modul wurde erst zu einem späteren Zeitpunkt entwickelt und hatte zum Ziel, robustes Testen von Benutzeroberflächen, die mit JavaFX erstellt wurden, zu ermöglichen [63].

[1] http://testng.org/doc/index.html
[2] http://junit.org
[3] http://docs.oracle.com/javafx/

Zusätzlich werden die folgenden drei Bibliotheken in FEST bereitgestellt, die als Ergänzung der Basismodule angesehen werden können und diese um spezielle Funktionen erweitern [15]:

- **EasyMock-Modul:** Verbessert die bestehende Bibliothek EasyMock[4], sodass die Entstehung von mehrfach vorhandenem, identischem Programmcode reduziert wird. EasyMock kann für die Erstellung von isolierten Tests bzw. für die Schaffung von Platzhaltern eingesetzt werden. Die Weiterentwicklung dieses Moduls wurde allerdings eingestellt. Stattdessen wird geraten, bei Bedarf auf die alternative Bibliothek Mockito[5] umzusteigen.
- **Assertions-Modul:** Beinhaltet Funktionen zur Spezifikation von Zusicherungen.
- **Reflection-Modul:** Bietet eine typsichere und einfachere Anwendung von Java Reflection. Funktionen dieser Bibliothek werden dann benötigt, wenn spezielle plattformspezifische Eigenschaften einer Komponente beim Testen einer Swing-Applikation ausgelesen werden müssen.

Die Entwickler von FEST verfolgten bei der Realisierung der Module ein Konzept, das in der Literatur als Fluent Interfaces bezeichnet wird. Die Grundidee ist die Erschaffung einer domainspezifischen Sprache innerhalb einer Programmiersprache (im Falle von FEST: Java). Das Ziel dabei ist, möglichst nahe an eine natürliche Sprache heranzukommen, sodass Code auch ohne Programmierkenntnisse nachvollzogen und intuitiv gelesen werden kann. Sehr häufig werden Fluent Interfaces durch das Aneinanderketten von Methoden (Method Chaining) realisiert [65]. Nachfolgendes Code-Beispiel 5.1 demonstriert die im Assertions-Modul von FEST bereitgestellte domainspezifische Sprache:

```
1 @Test
2 public void testFriendsOfAPerson(){
3
4   List<Person> friends = p1.getFriends();
5
6   assertThat(friends).hasSize(3).onProperty("name")
7       .contains("Hans").excludes("Maria");
8 }
```

Quellcode 5.1: Testfall unter Verwendung des FEST Assertions-Moduls mit Fluent Interfaces

Der in Code-Beispiel 5.1 abgebildete Testfall überprüft, ob die zurückgelieferte Liste genau drei Elemente enthält und ob gewisse Eigenschaften für die Elemente in der Liste zutreffen. Zum Vergleich zeigt Code-Beispiel 5.2 dieselbe Überprüfung mit klassischem JUnit ohne Verfügbarkeit einer domänspezifischen Sprache:

```
1 @Test
2 public void testFriendsOfAPerson(){
3
4   List<Person> friends = p1.getFriends();
5   assertThat(3, friends.size());
6   boolean foundHans = false;
7
8   for(Person p: friends){
```

[4] http://easymock.org
[5] https://code.google.com/p/mockito/

```
9       String name = p.getName();
10
11      if("Maria".equals(name)){
12        fail();
13      } else if("Hans".equals(name)){
14        foundHans = true;
15      }
16    }
17
18    assertTrue(foundHans);
19  }
```

Quellcode 5.2: Klassischer Testfall ohne die Verwendung von Fluent Interfaces

Es lässt sich sehr schnell erkennen, dass Code-Beispiel 5.2 nicht nur umfangreicher, sondern auch weniger intuitiv zu lesen ist.

Aufgrund der Relevanz für diese Arbeit wird nachfolgend nur auf das Swing-Modul näher eingegangen. Gemäß der FEST-Dokumentation [63] besitzt dieses den folgenden Funktionsumfang:

- Simulation von Benutzer-Interaktionen in der GUI der zu testenden Applikation. FEST kann dabei sämtliche Maus-, Tastatur- sowie Drag&Drop-Ereignisse im zugrundeliegenden Betriebssystem auslösen.
- FEST verfolgt einen robusten Ansatz zur Identifikation der GUI-Komponenten. Dem Tester werden verschiedene Look-Up-Konzepte bereitgestellt, die das Auffinden per Komponententyp, Name sowie regulärer Ausdrücke möglich machen. Änderungen am Layout oder am Look&Feel der Applikation sollten bereits erstellte Testfälle nicht beeinträchtigen.
- FEST unterstützt das Testen aller standardmäßig in Java verfügbaren Swing-Komponenten.
- Mit FEST erstellte GUI-Tests können wahlweise in JUnit- oder TestNG-Testmethoden eingebettet werden. Dazu erweitert das Swing-Modul beide Frameworks um zusätzliche Annotationen und Klassen. Es wird also dem Tester überlassen zu entscheiden, welches der genannten Frameworks besser für die jeweilige Situation geeignet ist.
- Erstellung von HTML-Reports nach Ausführung der Testfälle. In diese Reports wird im Falle eines fehlgeschlagenen Tests ein Screenshot eingebettet, der den Desktop bei Eintritt der Abweichung abbildet. Ein solcher Screenshot soll dem schnelleren Auffinden der Ursache des Fehlschlagens dienen.

Da es sich bei den erstellten Testfällen um erweiterte Varianten von JUnit- bzw. TestNG-Tests handelt, können diese außerdem direkt in Eclipse bzw. über ein Ant[6]-Skript ausgeführt werden. Zum Einsatz von FEST wird lediglich eine Java Entwicklungsumgebung in Version 1.5 oder höher vorausgesetzt. Sämtliche Module können entweder von Hand heruntergeladen und in ein Projekt eingebunden oder über das zentrale Maven-Repository[7] bezogen werden. Die gepackten Versionen werden von Google Code[8] gehostet, der Source-Code der jeweiligen Module ist über öffentliche git-Repositories zugänglich.

[6] http://ant.apache.org
[7] http://search.maven.org
[8] https://code.google.com/p/fest/

5.3.2 Marathon

Marathon ist ein Werkzeug zur Erstellung und Ausführung von funktionalen GUI-Tests ausschließlich für Swing-Applikationen, die mit Java 1.5 oder einer höheren Java-Version implementiert wurden. Das Open-Source-Projekt wurde Anfang 2002 ins Leben gerufen, der Source-Code ist dabei unter der GNU Lesser General Public License Version 2.0 (LGPLv2) lizensiert. Im Jahr 2006 spezialisierte sich das indische Unternehmen Jalian Systems auf Support und Consulting für Marathon und begann in einem weiteren Schritt die Entwicklung der kommerziellen Variante MarathonITE (Integrated Testing Environment). Diese baut auf Marathon auf und bietet neben professionellem Support zahlreiche zusätzliche Funktionen [31].

Marathon verfolgt einen Capture/Replay-Ansatz für die Erstellung und Wiedergabe der Testskripts und umfasst gemäß seiner Dokumentation [30] die folgenden Funktionen:

- Aufzeichnung eines Skripts durch Verwendung einer GUI-basierten Recorder-Applikation (siehe Abbildung 5.1). Die erstellten Testfälle können dann entweder als Ruby[9]- oder als Python[10]-Skript abgespeichert werden. Im Rahmen der Aufzeichnung können ebenfalls Zusicherungen definiert werden, die das erwartete Ergebnis spezifizieren und bei späterer Ausführung prüfen, ob dieses vom tatsächlichen Resultat abweicht.

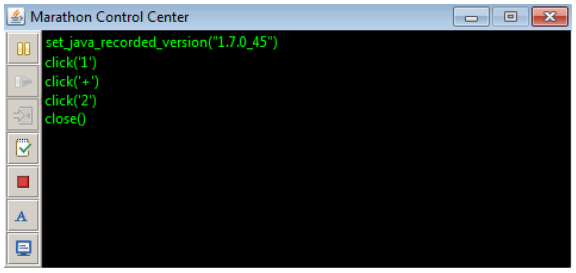


Abbildung 5.1: GUI-basierter Recorder zur Aufzeichnung der Benutzerinteraktionen

- Wiedergabe der aufgezeichneten Skripts. Marathon stellt dabei unterschiedliche Ausführungsmöglichkeiten bereit:
  - Marathon Test-Runner: Ermöglicht das Ausführen eines oder mehrerer Skripts über die Benutzeroberfläche von Marathon.
  - Kommandozeile: Mit Marathon erstellte Tests können auch über die Kommandozeile ausgeführt werden. Durch die Angabe zusätzlicher Parameter können spezielle Ausführungseinstellungen gewählt werden.
  - Ant-Build: Diese Variante ist speziell dann gedacht, wenn Skripts von einem externen Continuous Integration-Server ausgeführt werden sollen.
- Nachbearbeitung von bereits vorhandenen Skripts in Ruby oder Python.
- Erstellung wiederverwendbarer Module: Bestehende Skripts können in mehrere Module aufgespalten werden. Dies ermöglicht die Wiederverwendung eines Moduls in mehreren Testfällen.

[9] http://www.ruby-lang.org/de/
[10] http://www.python.org

- Erstellung datengetriebener Tests: Testskripts können im Rahmen der Nachbearbeitung so modifiziert werden, dass die Ein- und Ausgabewerte einer externen Datenquelle entnommen werden. Die Datendatei muss dazu im CSV-Format vorliegen. Eine automatisierte Auslagerung in eine separate Testdatendatei bereits während der Skriptaufzeichnung wird nur in der kommerziellen Variante bereitgestellt. Marathon ermöglicht dem Tester allerdings, bestimmte Teile des Skripts für das mehrmalige Durchlaufen mit verschiedenen Testdaten (diese Funktion wird in Marathon als Looping bezeichnet) festzulegen.

- Semi-automatisiertes Testen durch Verwendung von Checklisten: Im Zuge der Skriptaufzeichnung können Checklisten spezifiziert und Checkpoints im Skript definiert werden. Bei der Wiedergabe wird das Skript bis zum jeweils nächsten Checkpoint automatisiert ausgeführt, anschließend muss der nächste Punkt der Checkliste manuell bestätigt werden.

Nachfolgende Abbildung 5.2 zeigt die Benutzeroberfläche von Marathon. Der *Navigator* stellt alle Dateien des aktuellen Testprojekts (Testfälle, Testdaten, Reports, . . .) als Baumstruktur dar. Der *Text Editor* unterstützt Syntax-Highlighting und erlaubt die manuelle Nachbearbeitung der bestehenden Testskripts. Sämtliche Resultate der Testausführung werden im unteren Programmfensterbereich ausgegeben (*Ergebnis- und Fehlerausgabe*).

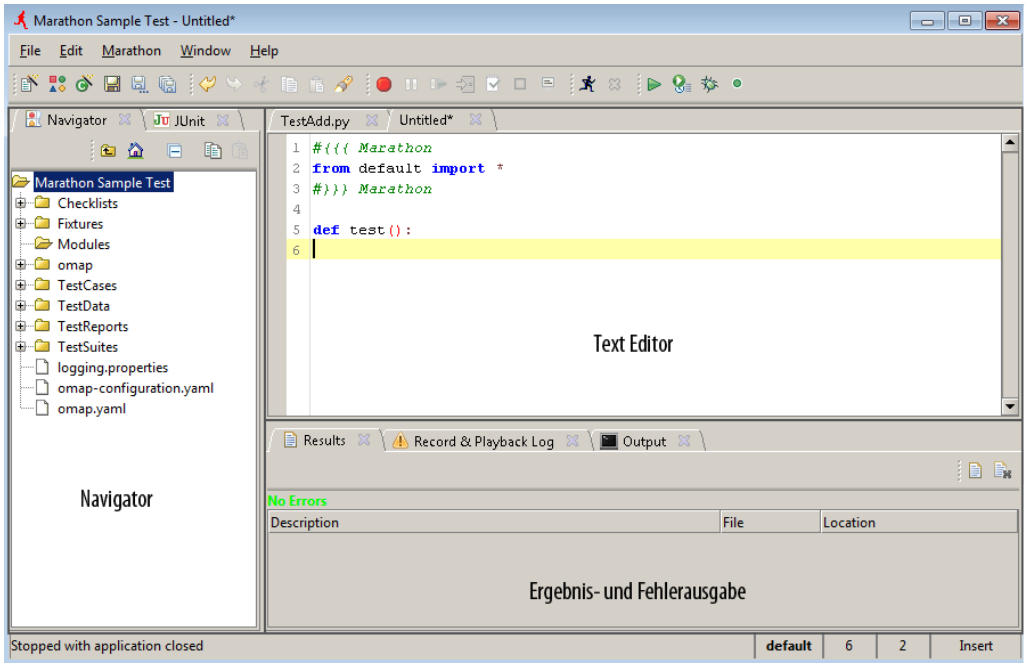


Abbildung 5.2: Marathon-Benutzeroberfläche

Neben zahlreichen kleineren Erweiterungen wurden bislang drei Hauptversionen der Software hervorgebracht: Marathon (2002), Marathon v2 (2008) sowie Marathon v3 (2011). Die derzeit aktuelle Version lautet 3.3.8.2 und wurde erst Ende November 2013 bereitgestellt [33]. Für die Verwendung von Marathon ist lediglich die Installation der Java Entwicklungsumgebung in Version 1.5 oder höher erforderlich, die Ausführung von Ruby- bzw. Python-Code wird durch Mitauslieferung von JRuby und Jython realisiert. Der Source-Code von Marathon ist seit Version 3 über ein öffentliches git-Repository zugänglich. Für Support-Fragen an die Open-Source-Gemeinschaft steht eine Google Group zur Verfügung.

5.3.3 Abbot

Bei Abbot handelt es sich um ein Werkzeug sowie eine Java-API zur Erstellung und Ausführung von funktionalen GUI-Tests für javabasierte Applikationen. Jener Teil des Projekts, der Funktionen zum Testen von SWT[11]- bzw. JFace[12]-Anwendungen bereitstellte, wurde 2009 in ein separates Projekt ausgegliedert und wird somit nicht mehr unter der Bezeichnung Abbot weiterentwickelt. Das Open-Source-Projekt Abbot – Abkürzung für „A better bot“ – wurde 2003 gegründet, beschränkt sich nun nur noch auf Möglichkeiten zum Testen von AWT- sowie Swing-GUIs und besitzt nach derzeitigem Stand 17 aktive Projektmitglieder. Der Quellcode ist unter der Eclipse Public License (EPL) lizensiert.

Abbot hat die Besonderheit, sowohl einen Capture/Replay-Ansatz als auch die skriptbasierte Erstellung von Testfällen zu unterstützen. Letzteres ist besonders dann relevant, wenn mit der Umsetzung der Tests bereits während der Implementierung begonnen werden soll oder wenn testgetrieben (TDD) vorgegangen wird. Die skriptbasierte Nutzung erfolgt über eine Java-API, die als Erweiterung zum JUnit-Framework angesehen werden kann. Die Erstellung neuer Testfälle passiert also innerhalb klassischer JUnit-Methoden und auch für die Überprüfung von erwartetem und tatsächlichem Ergebnis werden die von JUnit bereitgestellten Zusicherungen genutzt.

Die Abbot-API stellt dem Tester dabei High-Level-Funktionen bereit, sodass dieser sich nicht mit der Simulation betriebssystemnaher Events wie Mausbewegung oder Drücken einer Taste befassen muss. Sie bietet spezielle `Tester`-Klassen an, die jeweils alle auf einer bestimmten GUI-Komponente simulierbaren Aktionen kapselt. Für jede standardmäßig in Swing verfügbare Komponente liegt eine solche Tester-Klasse vor. Zur Realisierung dieser Abstraktion nutzt Abbot intern die Klasse `java.awt.Robot`, die es erlaubt, Low-Level-Events des Betriebssystems zu simulieren.

Die Capture/Replay-Funktionalität wird durch den Skript-Editor Costello (siehe Abbildung 5.3) bereitgestellt. Dieser ist ebenfalls in Java implementiert und bietet Möglichkeiten zur Aufzeichnung, Wiedergabe und Überarbeitung von Testskripts.

Zur Ausführung der zu testenden Applikation in Costello muss der Pfad zum JAR-Archiv sowie die ausführbare Klasse spezifiziert werden (*Launch Informationen*). Während der Aufzeichnung werden im Bereich *Komponentenhierarchie* sämtliche Swing-Komponenten der zu testenden Applikation in einer Baumstruktur geladen. Durch Auswahl einer Komponente aus der Hierarchie werden deren spezifische Eigenschaften im Fenster rechts unten (*Eigenschaften der Komponenten*) angezeigt. Sie können zur Spezifikation neuer Zusicherungen verwendet werden.

Costello kann also auch eingesetzt werden, um Informationen zur Komponentenhierarchie einer Benutzeroberfläche zu erhalten oder ein Video zur Demonstration der Funktionsweise einer Applikation aufzuzeichnen. Intern baut Costello auf der Abbot-API auf und weist somit denselben Funktionsumfang wie der skriptbasierte Erstellungsansatz auf. Bei der Ausführung eines mit Costello erstellten Skripts werden also ebenfalls High-Level-Aktionen durch Verwendung der Tester-Klassen simuliert.

Die mit Costello erstellten Skripts können als XML-Datei exportiert und unter Verwendung der Abbot-APIs in JUnit-Test-Suites eingebunden werden. Damit jedoch eine vollständige Unabhängigkeit zur Testumgebung und den genutzten Frameworks besteht, beinhaltet die XML-Datei alle für die Ausführung relevanten Informationen. Die Verwendung von XML zur Spezifikation der Testskripts hat einen entscheidenden Vorteil gegenüber einer Programmiersprache: Skripts können

[11] http://www.eclipse.org/swt/
[12] http://wiki.eclipse.org/JFace

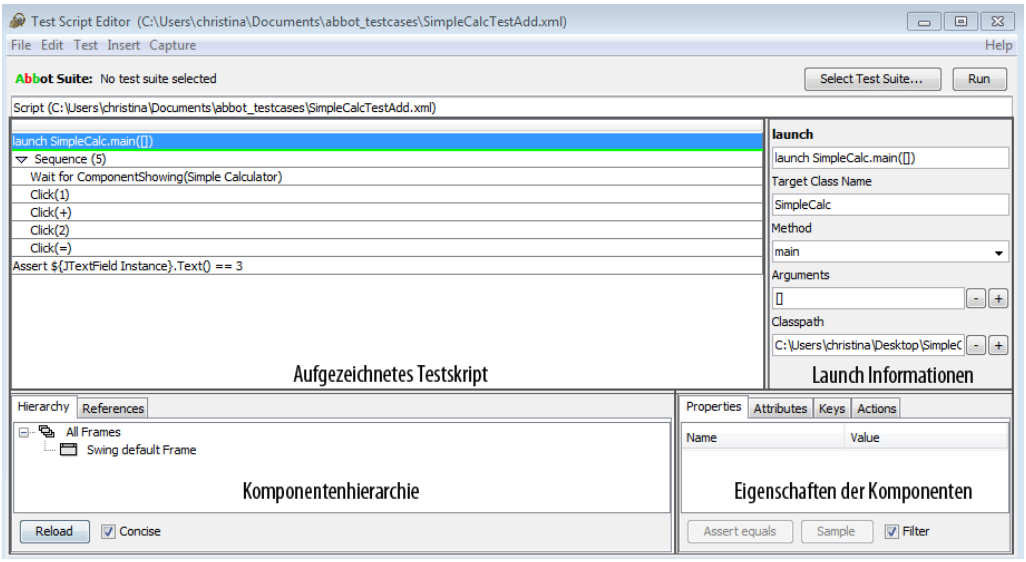


Abbildung 5.3: Skript-Editor Costello

dynamisch verändert werden und erfordern keine erneute Kompilierung [75].

Wall [75] nennt die folgenden Besonderheiten von Abbot:

- Abbot ermöglicht die Erstellung zuverlässiger, automatisiert ausführbarer GUI-Tests.
- Mit Abbot erstellte Testfälle sind unabhängig von Größe, Aussehen und Position einer Komponente und sind somit robust gegenüber Plattformwechsel und geringfügigen Layout-Änderungen.
- Einfache Benutzbarkeit der bereitgestellten API durch Abstraktion von Low-Level-Events.
- Die API ist auf Erweiterbarkeit ausgelegt und kann mit neuen Tester-Klassen für projektspezifische GUI-Komponenten bereichert werden.

Sämtliche Projektdateien sowie der Source-Code von Abbot werden auf SourceForge[13] gehostet, die aktuellste Version (1.3.0) wurde im August 2013 veröffentlicht.

5.3.4 Sikuli

Das Open-Source-Projekt Sikuli nahm seinen Ursprung in der Forschungsgruppe für User Interface Design am Massachusetts Institute of Technology (MIT), demnach ist auch der Source-Code der Software unter die MIT-Lizenz gestellt. Seit Hauptversion SikuliX wurde die Verantwortung für die Weiterentwicklung und Wartung des Projekts an Raimund Hocke sowie eine 19 Mitglieder umfassende Open-Source-Gemeinschaft übertragen. Sikuli ist plattform- und applikationsunabhängig und kann somit für die Erstellung von GUI-Tests für beliebige Software eingesetzt werden, unter anderem auch für das Testen mobiler Anwendungen wie iOS-, Midlet- oder Android-Applikationen [70].

[13] http://sourceforge.net/projects/abbot/files/abbot/1.3/

Der von Sikuli gewählte Ansatz zur Erstellung von Testfällen unterscheidet sich stark von dem der anderen betrachteten Werkzeuge. Die Grundidee orientiert sich an einem Phänomen der zwischenmenschlichen Kommunikation, nämlich der visuellen Referenzierung umliegender Objekte, um einer Aussage oder Geste Bedeutung zuzuordnen (z.B. zeigen auf den Salzstreuer, wenn man diesen gereicht haben möchte). Sikuli überträgt dieses Konzept auf den Bereich des Software-Testens, indem es die visuelle Gestaltung von Testfällen durch die Einbettung von Screenshots in diese ermöglicht. Anstelle der programmatischen Identifikation einer GUI-Komponente über deren eindeutigen Namen, deren Position oder reguläre Ausdrücke wird diese in Sikuli durch einen Screenshot spezifiziert [77].

Nachfolgende Abbildung 5.4 zeigt ein einfaches Sikuli-Testskript, das eine Applikation namens „CleanMyMac“ aufruft, anschließend den Computer durchsucht und nach Abschluss dieser Aktion die Enfernung unbenötigter Dateien bestätigt.

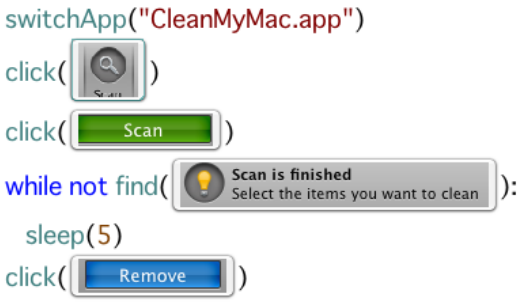


Abbildung 5.4: Einfaches Sikuli-Testskript
entnommen von [70]

Sikuli bietet dem Tester ein Set an Funktionen, die bestimmte Aktionen (wie z.B. Klicken, Auffinden, Eingabe von Text) mit einem Screenshot verknüpfen können. Zum Zeitpunkt der Ausführung eines Testskripts werden die Screenshots analysiert und auf deren Basis die GUI-Elemente in der zu testenden Applikation identifiziert. Zur Realisierung dieses Konzepts wurden Verfahren aufgegriffen, die üblicherweise im Bereich der Computer Vision zum Einsatz kommen. Diese werden als Template Matching bzw. Invariant Feature Voting bezeichnet und haben den Vorteil, dass sie die Übereinstimmung zwischen zwei Referenzbildern unabhängig von deren Größe, Farbe sowie Orientierung bestimmen können. Es ist also nicht zwingend erforderlich, dass der im Testskript eingebettete Screenshot vollständig mit den tatsächlichen Eigenschaften der GUI-Komponente in der Applikation übereinstimmt, wodurch die Robustheit der Sikuli-Tests gegenüber geringfügigen Layout-Änderungen gewährleistet wird.

Sikuli besteht aus den folgenden beiden Hauptkomponenten [77]:

- **Sikuli-Search:** ermöglicht den Benutzern eine bildbasierte Online-Suche.
- **Sikuli-Script:** ermöglicht die Erstellung von Testfällen über die bereits vorgestellte, auf Screenshots basierende Technik.

Für die Erstellung, Ausführung sowie Überarbeitung der Testskripts wird eine eigene Entwicklungsumgebung (SikuliX IDE) zur Verfügung gestellt. Diese bietet Funktionen zum Aufnehmen und Einbetten eines neuen Screenshots an einer ausgewählten Position im Testskript und kann die erzeugten Bilder auch sichtbar für den Tester – ähnlich zu Abbildung 5.4 – innerhalb der Entwicklungsumgebung darstellen. Als Skriptsprache fungiert Python, das um spezielle Befehle erweitert wurde. Die erstellte Bibliothek baut auf einer ebenfalls als Teil des Sikuli-Projekts entwickelten

Java-API auf. Alternativ kann anstelle der Python-Bibliothek auch direkt die Verwendung der Java-APIs erfolgen, allerdings müssen die Screenshots dann manuell erstellt und in Form einer Pfadangabe zur Bilddatei in den Java-Code eingebettet werden.

Die Verwendung von Sikuli ist jedoch nicht auf die mitgelieferte Entwicklungsumgebung beschränkt. Sowohl die Integration in Eclipse oder Netbeans als auch die Ausführung der Testskripts über die Kommandozeile werden unterstützt. Zusätzlich können Testfälle als Unit-Tests deklariert werden, die durch Nutzung von Jython zu JUnit kompatibel sind. Sikuli ermöglicht weiters die Modularisierung von Testfällen [70].

Chang, Yeh und Miller [8] weisen in ihrer Arbeit aus dem Jahr 2010 auf die folgenden zwei Beschränkungen von Sikuli hin:

- Es gibt keine Möglichkeit, unerwartetes Verhalten in jenen Bereichen der Benutzeroberfläche zu erkennen, die nicht in einem der Screenshots enthalten sind.
- Sikuli testet ausschließlich die Korrektheit der visuellen Repräsentation einer Benutzeroberfläche, nicht jedoch die interne Funktionsweise eines Systems.

5.3.5 Robot Framework

Bei Robot handelt es sich um ein unter der Apache Version 2.0 lizensiertes Open-Source-Framework zur Erstellung und Ausführung automatisierter Akzeptanztests für beliebige Applikationen. Das Projekt wurde 2008 ins Leben gerufen und seine Weiterentwicklung wird seitdem laufend vorangetrieben, die Projektgemeinschaft ist nach wie vor im Wachsen begriffen. Das Projekt wird zusätzlich durch das Nokia Siemens Network (NSN) finanziell unterstützt.

Robot verfolgt einen generischen Ansatz, bei dem Testfälle in tabellarischer Form über die Verwendung von Schlüsselwörtern spezifiziert werden. Das Framework stellt grundsätzlich nur eine beschränkte Menge an Schlüsselwörtern bereit, bietet jedoch eine Schnittstelle, um externe Test-Bibliotheken einzubinden und somit domainspezifische Schlüsselwörter verfügbar zu machen. Gemäß Dokumentation des Frameworks kann zwischen den folgenden Arten von Schlüsselwort-Bibliotheken unterschieden werden [18]:

- **Standard-Bibliotheken:** Alle Bibliotheken dieser Kategorie sind standardmäßig im Robot Framework bereitgestellt und die darin enthaltenen Schlüsselwörter können ohne weitere Konfiguration für die Erstellung von Testfällen verwendet werden. Standard-Bibliotheken stellen beispielsweise Schlüsselwörter für die Verarbeitung von XML-Dateien, für die Manipulation von Zeichenketten sowie für betriebssystemnahe Aufgaben bereit.
- **Externe Bibliotheken:** Diese Kategorie umfasst eine große Anzahl an Bibliotheken, die zwar separat eingebunden werden müssen, jedoch offiziell auf der Robot-Webseite gelistet sind. Es ist also sichergestellt, dass diese reibungslos funktionieren. Externe Bibliotheken ermöglichen das Testen spezifischer Domänen wie beispielsweise mobile Apps (Android, iOS), Webseiten, Java-GUIs oder datenbanknahe Applikationen.
- **Projektspezifische Bibliotheken:** Des Weiteren ist auch die Einbindung selbstimplementierter oder im Internet verfügbarer Bibliotheken möglich. Es wird jedoch auf der offiziellen Robot-Webseite keine jeweilig spezifische Beschreibung bzw. Hilfestellung bereitgestellt. Zusätzlich besteht die Einschränkung, dass nur Bibliotheken, die in Java oder Python implementiert sind, eingebunden werden können.

Das Robot Framework ist in Python implementiert und bietet die folgenden Funktionen:

- Erstellung datengetriebener Testfälle durch Nutzung von Platzhaltern statt realen Daten.
- Einsatz eines grafischen Editors (RIDE) zur Spezifikation und Verwaltung der Testdaten.
- Generierung von Logs und Reports im XML-Format.
- Benutzerspezifische Definition neuer Schlüsselwörter auf Basis bereits bestehender.

Nachfolgende Abbildung 5.5 zeigt die modulare Arbeitsweise von Robot und demonstriert die strikte Trennung zwischen Testfällen, Framework sowie Bibliotheken.

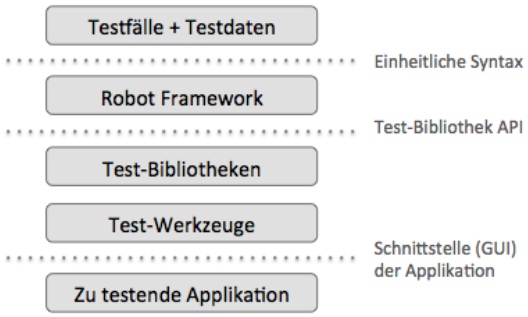


Abbildung 5.5: Modulare Arbeitsweise von Robot
angelehnt an [18]

Die in tabellarischer Form spezifizierten Testfälle und -daten müssen einer strikten Syntax folgen und können in eine HTML-, Text- oder Tab Separated Values (TSV)-Datei eingebettet werden. Tabelle 5.4 beinhaltet einen in Robot spezifizierten einfachen Testfall. Dieser dient der Überprüfung, ob ein neu erstellter Benutzer sich bei Angabe der korrekten Benutzerdaten erfolgreich einloggen kann. Der Text in der ersten Spalte (TestCase) repräsentiert die textuelle Beschreibung, die zweite Spalte (Action) enthält die Schlüsselwörter, welche die auszuführenden Aktionen indizieren.

TestCase	Action	Argument1	Argument2
Erzeugung eines validen Users und Login	Create Valid User	fred	abc
	Attempt to Login with Credentials	fred	abc
	Status Should be	Logged In	

Tabelle 5.4: Beispielhafter Testfall in Robot
angelehnt an [19]

Robot unterstützt die Ausführung von Testfällen über die Kommandozeile. Weiters befasst sich das eigenständige Open-Source-Projekt RobotAnt mit der Einbindung von Robot-Testfällen in Ant-Build-Skripts [19]. Der Continuous Integration-Server Jenkins bietet außerdem ein eigenes Robot Framework-Plugin an, das die Ausführung mit Robot erstellter Tests erleichtert und die Ergebnisse auswerten kann. Das Plugin wird nach wie vor weiterentwickelt, die letzte Version wurde im November 2013 bereitgestellt [58].

Die Hauptmotivation für die nähere Betrachtung des Robot Frameworks ergibt sich aus einem umfassenden Artikel von Moroz [44], in dem die Kombinationsmöglichkeit des Sikuli-Werkzeugs

mit Robot diskutiert wird. Ein mit Sikuli erstelltes Testskript kann gemäß dem Autor in eine Test-Bibliothek umgewandelt werden. Demnach können die Vorteile des Sikuli-Ansatzes mit der schlüsselwort- sowie datengetriebenen Funktionsweise von Robot verbunden und zusätzlich die Erstellung von Reports sowie die reibungslose Einbindung in Jenkins erzielt werden.

5.3.6 WindowTester Pro

Bei WindowTester Pro handelte es sich ursprünglich um eine kommerzielle Software, die zusammen mit WindowBuilder, einem Werkzeug zur Erstellung von Java-GUIs, sowie dem Code-Analyse-Tool CodePro AnalytiX von dem amerikanischen Unternehmen Instantiations Inc. entwickelt und vertrieben wurde. 2010 erfolgte eine Übernahme der genannten Produkte durch Google und in einem weiteren Schritt wurde der Source-Code von WindowTester Pro im März 2012 unter der EPL Version 1.0 als Open-Source zugänglich gemacht [12, 55].

WindowTester Pro ist als Eclipse-Plugin[14] erhältlich und kann zum automatisierten Testen von javabasierten Rich-Client-Applikationen (Swing, SWT, RCP, Eclipse-Plugins) eingesetzt werden. Voraussetzung für die Nutzung ist das Importieren des zu testenden Projekts in Eclipse. WindowTester Pro verfolgt einen Capture/Replay-Ansatz, die aufgezeichneten Interaktionen können über einen mitgelieferten Codegenerator als standardkonforme JUnit-Testfälle exportiert werden [40].

WindowTester Pro umfasst dabei die folgenden Komponenten [13]:

- **Recording Console:** Besitzt eine grafische Oberfläche, die Funktionen zum Starten, Stoppen sowie Pausieren von Aufzeichnungen bereitstellt und dem Tester unmittelbares Feedback über die verarbeiteten Aktionen gibt. Es ermöglicht deren Verwaltung und Modularisierung sowie die Definition von Zusicherungen. Nachfolgende Abbildung 5.6 zeigt die Recording Console, die in Eclipse als eigener View angezeigt wird:

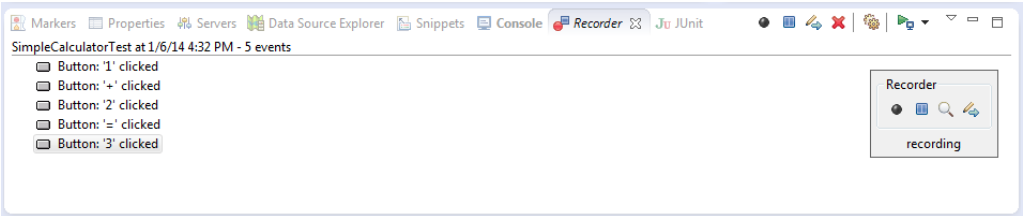


Abbildung 5.6: WindowTester Pro Recording Console

- **Code Generation Wizard:** Die aufgezeichneten Aktionen können unter Verwendung des Code Generation Wizards als JUnit-Testfälle exportiert werden.
- **Code Coverage:** Diese Komponente evaluiert den Überdeckungsgrad der mit WindowTester Pro erstellten Testfälle und macht somit eine Aussage über deren Effektivität.

WindowTester Pro stellt weiters eine umfangreiche API zur Verfügung, mit der neue Testfälle auch von Hand implementiert bzw. bereits bestehende überarbeitet werden können. Auch die automatisierte Ausführung im Zuge von Continuous Integration ist gegeben, da es sich, wie bereits erwähnt, um konforme JUnit-Tests handelt. Allerdings erfordert WindowTester Pro eine grafische Benutzeroberfläche [13].

[14] Link zur Eclipse Marketplace Seite: https://marketplace.eclipse.org/content/windowtester-pro-gui-tester

6 Machbarkeitsstudie und prototypische Testfallimplementierung

Dieses Kapitel evaluiert die Praxistauglichkeit der ausgewählten Testautomatisierungswerkzeuge durch prototypische Implementierung von fünf repräsentativen Testfällen. Im ersten Abschnitt erfolgt die Beschreibung der für die Ableitung der Testfälle herangezogenen Anwendungsfälle der bestehenden Applikation. Anschließend werden die Vorgehensweise bei der Umsetzung der Testfälle sowie die positiven Aspekte und Problemstellen der einzelnen Werkzeuge erläutert.

6.1 Beschreibung der ausgewählten Anwendungsfälle

Aus dem Funktionsumfang des bestehenden Softwaresystems wurden insgesamt fünf Anwendungsfälle für die prototypische Umsetzung von Testfällen entnommen. Diese Szenarien wurden so gewählt, dass sie verschiedene Bereiche des Systems abdecken und Interaktion mit vielen unterschiedlichen GUI-Komponenten erfordern, wodurch eine fundiertere Einschätzung der Praxistauglichkeit des jeweiligen Testtools ermöglicht werden sollte. Das in Abbildung 6.1 dargestellte UML-Diagramm liefert einen Überblick über die gewählten Anwendungsfälle:

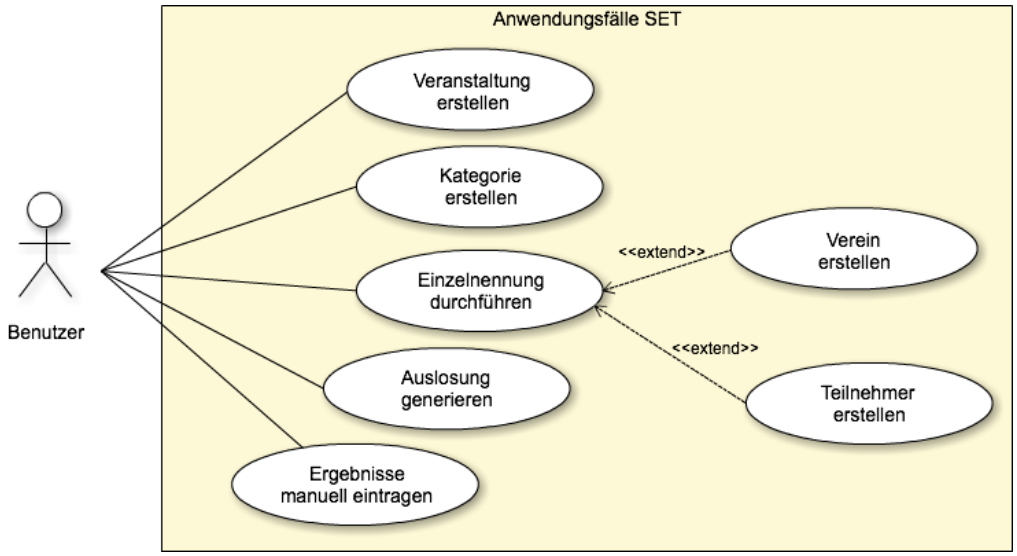


Abbildung 6.1: Ausgewählte Anwendungsfälle des bestehenden Softwaresystems

Nachfolgende Tabellen 6.1 bis 6.7 beinhalten Beschreibungen zu den in Abbildung 6.1 dargestellten Anwendungsfällen. Auf alternative Abläufe sowie Fehlerfälle wurde in den Beschreibungen bewusst verzichtet, da diese keine Relevanz für die Umsetzung der Testfälle besitzen. Voraussetzung für alle Anwendungsfälle ist, dass die Applikation gestartet wurde und die Eingabe der korrekten Login-Daten erfolgt ist.

Bezeichnung:	#1 Veranstaltung erstellen
Kurzbeschreibung:	Ermöglicht das Erstellen einer neuen Sportveranstaltung durch Angabe der erforderlichen Daten (Veranstaltungsname, Beginn, Ende, Nennfrist, Land, Währung, Modus sowie Typ).
Vorbedingungen:	Der Benutzer befindet sich im Fenster *Veranstaltungsdaten.* Eine Veranstaltung mit diesem Namen existiert noch nicht.
Nachbedingungen:	Die Veranstaltung wurde gemäß den eingegebenen Daten erstellt und ist im Hauptfenster geöffnet.
Standardablauf:	(1) Eingabe der erforderlichen Daten für Veranstaltungsname, Beginn, Ende, Nennfrist, Land, Währung (optional), Modus (optional) sowie Typ (optional). (2) Bestätigung der eingegebenen Daten durch Klick auf den *Weiter*-Button.

Tabelle 6.1: Anwendungsfallbeschreibung für *Veranstaltung erstellen*

Bezeichnung:	#2 Kategorie erstellen
Kurzbeschreibung:	Ermöglicht das Erstellen einer neuen Kategorie durch Angabe der erforderlichen Daten (Bezeichnung, untere/obere Altersgrenze sowie Geschlecht) und das Hinzufügen dieser zu einer bestehenden Veranstaltung.
Vorbedingungen:	Die Veranstaltung, für die eine neue Kategorie erstellt werden soll, existiert bereits und wurde im Hauptfenster geöffnet. Eine Kategorie mit diesem Namen existiert noch nicht.
Nachbedingungen:	Die neue Kategorie wurde gemäß den eingegebenen Daten erstellt und liegt in der Baumansicht des Hauptfensters unter *Kategorien dieser Veranstaltung* vor.
Standardablauf:	(1) Klick im Menü auf *Datei/Kategorien dieser Veranstaltung* oder Rechtsklick in Baumansicht auf *Kategorien dieser Veranstaltung*. (2) Klick auf *Kategorie hinzufügen*. (3) Eingabe der gewünschten Daten für Bezeichnung, untere/obere Altersgrenze sowie Geschlecht (optional). (4) Klick auf *speichern*. (5) Bestätigung, dass die Kategorie verwendet werden soll.

Tabelle 6.2: Anwendungsfallbeschreibung für *Kategorie erstellen*

Bezeichnung:	#3 Einzelnennung durchführen
Kurzbeschreibung:	Ermöglicht die Registrierung eines neuen Teilnehmers eines Vereins für eine ausgewählte Kategorie einer Veranstaltung. Verein sowie Teilnehmer können im Rahmen der Nennung neu angelegt werden.
Vorbedingungen:	Die Veranstaltung ist im Hauptfenster geöffnet. Die Kategorie, für die eine Nennung durchgeführt werden soll, existiert bereits. Es wurde noch keine Nennung für diesen Teilnehmer für diese Kategorie in dieser Veranstaltung durchgeführt.
Nachbedingungen:	Die Nennung für einen bestimmten Teilnehmer eines Vereins wurde gespeichert und ist in der Baumansicht des Hauptfensters unter *Nennungen/Einzel / Team Nennungen* sichtbar.

Standardablauf:	(1) Klick im Menü auf *Datei/Nennungen/Einzelstarter* oder Rechtsklick in der Baumansicht auf *Nennungen/Nennung Einzelstarter*. (2) Auswahl eines Vereins. [Extension Point: #3.1] (3) Auswahl des Teilnehmers. [Extension Point: #3.2] (4) Auswahl der Kategorie. (5) Klick auf *Nennung bestätigen*.

Tabelle 6.3: Anwendungsfallbeschreibung für *Einzelnennung durchführen*

Bezeichnung:	#3.1 Verein erstellen
Kurzbeschreibung:	Ermöglicht das Erstellen eines neuen Vereins durch Angabe der erforderlichen Daten (Bezeichnung, Kürzel, Nation, Verband).
Vorbedingungen:	Der Benutzer befindet sich im Fenster *Verein Manager*.
Nachbedingungen:	Der neu erstelle Verein wurde gespeichert und kann in #3 ausgewählt werden.
Standardablauf:	(1) Eingabe der erforderlichen Daten für Bezeichnung und Kürzel. (2) Klick auf *Nation suchen* und Auswahl der Nation. (3) Klick auf *Landesverbände* und Auswahl des Landesverbandes (optional). (4) Klick auf *speichern*.

Tabelle 6.4: Anwendungsfallbeschreibung für *Verein erstellen*

Bezeichnung:	#3.2 Teilnehmer erstellen
Kurzbeschreibung:	Ermöglicht das Erstellen eines neuen Teilnehmers durch Angabe der erforerlichen Daten (Vorname, Nachname, Verein, Geburtsdatum, Gewicht, Größe, Geschlecht, Kyu/Dan).
Vorbedingung:	Der Benutzer befindet sich im Fenster *Teilnehmer Manager*.
Nachbedingung:	Der Teilnehmer wurde gespeichert und kann in #3 ausgewählt werden.
Standardablauf:	(1) Eingabe der erforderlichen Daten für Vorname, Nachname, Geburtsdatum, Gewicht, Größe, Geschlecht (optional), Kyu (optional) oder Dan (optional). (2) Klick auf *Vereinsnummer suchen* und Auswahl des Vereins. (3) Klick auf *speichern*.

Tabelle 6.5: Anwendungsfallbeschreibung für *Teilnehmer erstellen*

Bezeichnung:	#4 Auslosung generieren
Kurzbeschreibung:	Ermöglicht, eine Auslosung der registrierten Teilnehmer innerhalb einer bestehenden Veranstaltung und Kategorie gemäß definierter Kriterien zu generieren.
Vorbedingungen:	Veranstaltung, Kategorie sowie Nennungen sind bereits vorhanden und die Veranstaltung, für die eine Auslosung generiert werden soll, ist im Hauptfenster geöffnet.
Nachbedingungen:	Eine Auslosung wurde generiert und abgespeichert. Sie liegt in der Baumansicht des Hauptfensters als neuer Eintrag unter *Auslosung, Mitschrift, Punktetabelle.../Auslosung* vor.

Standardablauf:	(1) Klick im Menü auf *Datei/Auslosen*. (2) Auswahl der Kategorie und Bestätigen der Auswahl. (3) Festlegung der Auslosungskriterien. (4) Schließen der generierten Auslosung und Bestätigung, dass sie gespeichert werden soll.

Tabelle 6.6: Anwendungsfallbeschreibung für *Auslosung generieren*

Bezeichnung:	#5 Ergebnisse manuell eintragen
Kurzbeschreibung:	Ermöglicht, die Resultate (die Platzreihung) eines Turniers händisch einzutragen.
Vorbedingungen:	Veranstaltung, Kategorie sowie Nennungen sind bereits vorhanden und die Veranstaltung, für die eine Auslosung generiert werden soll, ist im Hauptfenster geöffnet.
Nachbedingungen:	Die manuell eingetragenen Ergebnisse sind abgespeichert und in der Baumansicht des Hauptfensters als neuer Eintrag unter *Ergebnis* sichtbar.
Standardablauf:	(1) Klick im Menü auf *Datei/Ergebnisliste/Erstellen* oder Rechtsklick in Baumansicht auf *Ergebnis/Ergebnisliste erstellen / Bearbeiten*. (2) Auswahl einer Kategorie. (3) Klick auf *Bearbeiten*. (4) Vergabe der Plätze durch Auswahl der genannten Teilnehmer aus der jeweiligen Combobox. (5) Klick auf *speichern*.

Tabelle 6.7: Anwendungsfallbeschreibung für *Ergebnisse manuell eintragen*

Das Starten der Applikation ist ein komplexer Vorgang, der sich über mehrere Programmfenster erstreckt und dem Benutzer verschiedene Optionen zur Konfiguration bereitstellt. Der für die Implementierung der Testfälle relevante Ablauf ist in Abbildung 6.2 in Form eines UML-Aktivitätsdiagramms abgebildet.

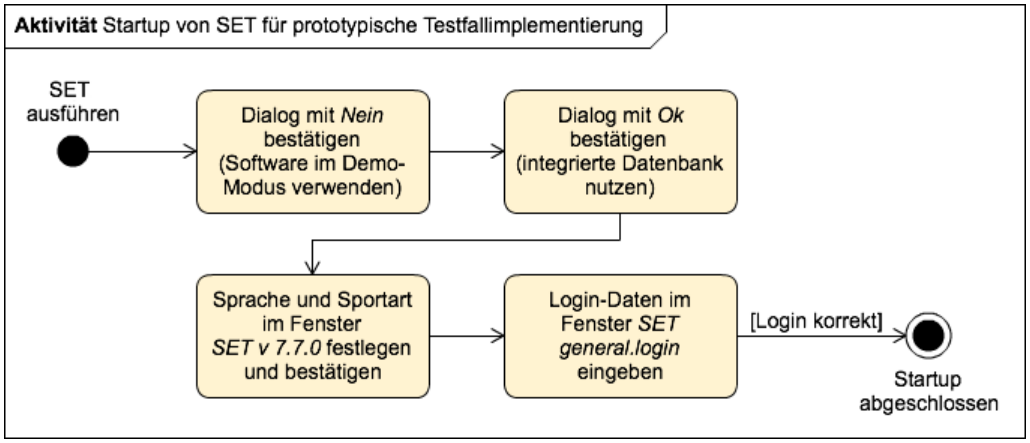


Abbildung 6.2: Startvorgang und Setup der Applikation

Zu Beginn gilt es festzulegen, ob die Software im Demo-Modus verwendet werden soll oder ob eine gültige Lizenz vorliegt. Der Demo-Modus hat die Einschränkung, dass zum Speichern sämtlicher Datensätze nur eine interne, lokal verfügbare Datenbank herangezogen werden kann. Diese Option ist für die prototypische Umsetzung der Testfälle jedoch vollkommen ausreichend. Nach optionaler Einstellung der Anzeigesprache werden in einem weiteren Programmfenster die Login-Daten abgefragt. Nach erfolgreicher Eingabe dieser ist der Startvorgang der Applikation abgeschlossen und der Benutzer kann die gewünschten Tätigkeiten durchführen.

6.2 Besonderheiten der bestehenden Applikation

Einige Implementierungsdetails der zu testenden Applikation haben die Umsetzung der Testfälle beeinflusst und werden daher an dieser Stelle näher ausgeführt:

- Die Applikation macht Gebrauch von der SwingX-Bibliothek[1]. Dieses Paket erweitert das Standard-Toolkit von Swing um zusätzliche, häufig benötigte GUI-Komponenten.
- Bei der Implementierung der Applikation wurde auf die Vergabe eindeutiger Bezeichner (kann in Swing über die Methode `setName` erfolgen) für die GUI-Komponenten großteils verzichtet. Dies hat zur Folge, dass namenlose Komponenten im Rahmen von automatisierten GUI-Tests nur über deren Position bzw. Beschriftung in der Benutzeroberfläche angesprochen werden können.
- Die Applikation besitzt zwar ein Hauptfenster, sehr viel Interaktion läuft jedoch in zusätzlichen Programmfenstern ab, die sich je nach Aktion im Hauptfenster öffnen. Die Titel dieser zusätzlichen Fenster sind nicht immer applikationsweit eindeutig.
- Der Titel des Hauptfensters wird dynamisch zusammengesetzt und beinhaltet Datum und Uhrzeit des Zeitpunkts der Programmausführung.
- Sämtlicher Text der Benutzeroberfläche liegt in separaten Sprachdateien internationalisiert vor. Die gewünschte Sprache kann im Zuge des Startvorgangs gewählt oder zu einem späteren Zeitpunkt im Hauptfenster gewechselt werden.

6.3 Prototypische Implementierung der Testfälle

Die im praktischen Teil dieser Arbeit implementierten Testfälle wurden den Standardabläufen der zuvor definierten Anwendungsfälle entnommen. Zusätzlich wurde ein weiterer Testfall definiert, der den in Abbildung 6.2 dargestellten Startvorgang der Applikation automatisieren und als Basis für alle nachfolgenden Testfälle fungieren soll.

Vor Beginn der Umsetzung der Testfälle wurde eine mit sinnvollen Testdaten befüllte Testdatenbank erstellt, die vor jedem Testlauf zurückgesetzt wurde. Diese garantierte die Unabhängigkeit zwischen den einzelnen Testfällen und sorgte für die Erfüllung ihrer Vorbedingungen (siehe Zeile *Vorbedingungen* in den Tabellen 6.1 bis 6.7).

Nachfolgende Abschnitte dokumentieren die Vorgehensweise bei der Implementierung der Testfälle sowie deren Ergebnisse für jedes der in Kapitel 5.3 ausgewählten Testwerkzeuge.

[1] http://download.java.net/javadesktop/swinglabs/releases/0.8/docs/api/org/jdesktop/swingx/package-summary.html

6.3.1 FEST

Mit FEST konnten insgesamt vier der fünf Testfälle erfolgreich und vollständig implementiert werden, bei Testfall #5 scheiterte die Umsetzung aufgrund der Anwesenheit zweier Fenster mit identischem Titel. Die von FEST bereitgestellten Klassen sowie deren Methoden tragen eindeutige und ausdrucksstarke Bezeichnungen, sodass eine schnelle Einarbeitung in die API gewährleistet ist. Das Konzept der Fluent Interfaces sorgt weiters für einen hohen Grad an Lesbarkeit.

Tool-Setup und Automatisierung des Startvorgangs

Das Setup von FEST gestaltete sich einfach und reibungslos. Nach der Entscheidung, das Testframework JUnit für die Einbettung der FEST-Testfälle zu verwenden, mussten lediglich die JUnit-basierten Versionen der FEST-Bibliotheken zum Build-Path des bestehenden Projekts hinzugefügt werden.

Zur besseren Modularisierung der Testfälle wurde das in Abbildung 6.3 dargestellte Vererbungskonstrukt für die Testklassen entworfen. Die `setup`-Methode der `BaseTest`-Klasse beinhaltet sämtlichen Code, der für die Automatisierung des Startvorgangs der Applikation benötigt wird. Durch Verwendung der JUnit-Annotation `@Before` wird weiters veranlasst, dass diese Methode automatisch vor jedem Test ausgeführt wird.

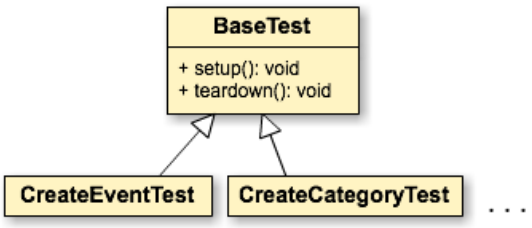


Abbildung 6.3: Vererbungshierarchie für FEST-Testfälle

Das Automatisieren des Startvorgangs mit FEST funktionierte nicht vollständig. Probleme bereiteten die beiden Dialoge, welche Bestandteil der ausführbaren Klasse sind und die weitere Ausführung des Programms solange blockieren, bis eine Benutzerinteraktion erfolgt ist. FEST kann allerdings erst im Anschluss an die vollständige Abarbeitung der `main`-Methode Interaktionen simulieren. Abhilfe schaffte die Einführung eines Test-Modus durch Abänderung der ausführbaren Klasse. Im Test-Modus wird das Anzeigen der betroffenen Dialoge verhindert.

Exemplarischer Testfall

Nachfolgendes Listing 6.1 zeigt einige Ausschnitte aus der Implementierung des Testfalls *#3 Einzelnennung durchführen*. Es ist zu erkennen, dass die Klasse `FrameFixture` im Zusammenhang mit dem Auffinden eines neuen Fensters sehr häufig Verwendung findet. FEST stellt eine große Anzahl solcher `Fixture`-Klassen bereit. Sie kapseln sämtliche Funktionen zur Simulation und Verifikation von Benutzerinteraktionen mit einer bestimmten GUI-Komponente.

Weiters ist das bereits in Abschnitt 5.3.1 angesprochene Konzept der Fluent Interfaces sehr stark vertreten, was sich sowohl beim Aneinanderketten von Interaktionen (Zeile 6 und 11) als auch bei

der Verifikation der Ergebnisse (Zeilen 21-23) erkennen lässt.

```
1  @Test
2  public void addEntry() {
3
4    //(1) aus Standardablauf von Anwendungsfall #3
5    FrameFixture mainFrame = findWindow("\\(c\\)sportdata GmbH .*");
6    mainFrame.menuItemWithPath("Datei", "Nennungen",
7        "Einzelstarter").click();
8
9    //(2) aus Standardablauf von Anwendungsfall #3
10   FrameFixture entryFrame = findWindow("Nennung Einzelstarter");
11   entryFrame.button(JButtonMatcher.withText("Verein hinzufügen")).click();
12
13   //(1) aus Standardablauf von Anwendungsfall #3.1
14   FrameFixture addClubFrame = findWindow("Verein Manager");
15   enterTextByIndex(addClubFrame, 1, clubName);
16   enterTextByIndex(addClubFrame, 2, clubShortName);
17
18   ...
19
20   //Verifikation, ob Einzelnennung korrekt durchgeführt wurde
21   assertThat(competitorsFrame.table().rowCount()).isEqualTo(1);
22   String[][] contents = competitorsFrame.table().selectRows(0).contents();
23   assertThat(contents[0]).hasSize(8).isEqualTo(expectedResult);
24 }
25
26 private FrameFixture findWindow(String windowTitle) {
27   return WindowFinder.findFrame(
28       FrameMatcher.withTitle(windowTitle)).using(robot());
29 }
```

Quellcode 6.1: Ausschnitt aus dem FEST-Testskript *#3 Einzelnennung durchführen*

Ebenfalls erwähnenswert ist das Auffinden des Hauptfensters in Zeile 5 durch Verwendung eines regulären Ausdrucks, wodurch die Problematik mit dem dynamischen Fenstertitel gelöst wird. Auf die Methode `enterTextByIndex` wird an dieser Stelle nicht näher eingegangen, dies wird jedoch zu einem späteren Zeitpunkt nachgeholt.

Support für datengetriebenes Testen

Seit Version 4 unterstützt JUnit und infolgedessen auch FEST das Erstellen und Ausführen datengetriebener Tests. Zur Nutzung dieser Funktion sind die folgenden Adaptionen an einer JUnit-Testklasse erforderlich:

- Annotation der Testklasse mit `@RunWith(Parameterized.class)`.
- Implementierung einer Methode, die mit `@Parameters` annotiert wird und für das Bereitstellen der Daten zuständig ist.
- Implementierung eines Konstruktors, der die bereitgestellten Daten entgegennimmt.

Nachfolgendes Listing 6.2 zeigt die relevanten Aspekte eines solchen parametrisierten Testfalls. Wie in Zeile 12 des Listings zu sehen ist, wurden die Daten für die FEST-Testfälle über eine CSV-Datei bereitgestellt. Jede Zeile dieser Datei entspricht einem separaten Testdurchlauf. Somit wird es möglich, denselben Testfall je nach Belieben mehrmals mit unterschiedlichen Datensätzen auszuführen. Die Größe der `Object`-Arrays entspricht dabei der Anzahl der Daten in einer Zeile und muss mit der Anzahl und Reihenfolge der im Konstruktor erwarteten Parameter übereinstimmen.

Alternativ könnten die Daten auch direkt in Java definiert oder von beliebigen anderen Dateien oder Datenbanken eingelesen werden.

```
@RunWith(Parameterized.class)
public class AddEntryTest extends BaseTest {
  private String clubName;
  private String clubShortName;

  public AddEntryTest(String clubName, String clubShortName) {
    ... // Setzen der korrespondierenden Objektvariablen
  }

  @Parameters
  public static Collection<Object[]> data() {
    return CSVDataLoader.load("testdata.csv");
  }
}
```

Quellcode 6.2: Datengetriebener JUnit4-Testfall

Positive Aspekte

In nachfolgender Aufzählung werden jene Aspekte von FEST näher erläutert, die bei der Umsetzung der Testfälle positiv aufgefallen sind:

- Das Assertions-Modul von FEST erweitert die bestehenden JUnit-Zusicherungen um wichtige und hilfreiche Methoden. Die Verwendung von Fluent Interfaces ermöglicht zusätzlich, in einer Anweisung gleich mehrere, inhaltlich zusammengehörige Überprüfungen umzusetzen. So kann beispielsweise das Übereinstimmen der Länge sowie der Inhalte eines Arrays gemeinsam geprüft werden, wodurch doppelter Code vermieden wird.
- FEST kann Interaktionen mit den Komponenten der SwingX-Erweiterung mit gewissen Einschränkungen simulieren. Verwendet werden muss dabei die `Fixture`-Klasse jener Standard-Swing-GUI-Komponente, von der die SwingX-Komponente abgeleitet wurde. Für die Klasse `JXTable` der SwingX-Bibliothek kommt also die `JTableFixture` zur Anwendung. Simuliert werden können allerdings nur jene Interaktionen, die auch in der Standard-Swing-Komponente verfügbar sind.
- Die Verwendung von regulären Ausdrücken kann nicht nur für die Identifikation von Fenstern sondern für beliebige GUI-Komponenten erfolgen.

Problemstellen und Lösungsansätze

Im Zuge der Umsetzung der Testfälle ergaben sich die folgenden Probleme:

- **Identifikation per Index:** FEST sieht aufgrund der Robustheit dieses Ansatzes eine Identifikation der GUI-Komponenten über deren eindeutige Namen vor. Da diese in der bestehenden Applikation fehlen, können GUI-Elemente desselben Typs nicht immer eindeutig voneinander unterschieden werden, was besonders die Umsetzung der Testfälle *#1* bis *#3* beeinträchtigte.

  Abbildung 6.4 verdeutlicht dieses Problem, indem sie den Ausschnitt eines Fensters von Anwendungsfall *#3* zeigt, in das drei leere Textfelder eingebettet sind. Die Identifizierung dieser Textfelder erfordert die Verwendung ihrer Indices, also der relativen Position eines Textfelds

gegenüber den restlichen Textfeldern (das Textfeld rechts von *Verein* kann beispielsweise über den Index 1 angesprochen werden).

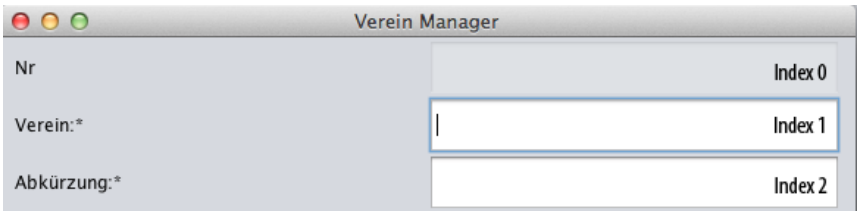


Abbildung 6.4: Ausschnitt aus der bestehenden Applikation, der die Notwendigkeit der Identifikation per Index verdeutlicht

Die Selektion einer Komponente per Index wird von FEST standardmäßig nicht unterstützt und erforderte somit die Erweiterung der bestehenden `FrameFixture`-Klasse. Für jede GUI-Komponente, die das Ansprechen per Index ermöglichen sollte, wurde eine zusätzliche Methode implementiert. Listing 6.3 zeigt exemplarisch die neu erstellte Methode für die Selektion eines Textfeldes per Index.

```
1  public class CustomizedFrameFixture extends FrameFixture {
2    ...
3
4    public JTextComponentFixture textBox(final int index) {
5
6      //Auffinden aller Textfelder innerhalb des aktuellen Fensters
7      Collection<JTextComponent> results = finder().findAll(...);
8
9      //Rückgabe des Textfelds mit gewünschtem Index
10     Iterator<JTextComponent> iterator = results.iterator();
11     for(int i = 0; iterator.hasNext(); i++){
12       if(i == index){
13         return new JTextComponentFixture(robot, iterator.next());
14       }
15     }
16
17     return null;
18   }
19 }
```

Quellcode 6.3: Erweiterung der `textBox`-Methode um Selektion per Index

Die Methode `enterTextByIndex` aus Listing 6.1 kapselt die Verwendung dieser neu erstellten Methode und ermöglicht, den angegebenen Text in das Textfeld am spezifizierten Index im Fenster `parent` einzufügen.

```
1  private void enterTextByIndex(FrameFixture parent, int i, String t){
2
3    //Erzeugen eines neuen Objekts von CustomizedFrameFixture
4    CustomizedFrameFixture temp =
5        new CustomizedFrameFixture(robot(), parent.target);
6
7    //Einfügen des Textes t in Textfeld mit Index i
8    temp.textBox(i).enterText(t);
9  }
```

Quellcode 6.4: Implementierung der Methode `enterTextByIndex`

- **Abstraktion in Tabellen:** FEST unterstützt die Selektion einer Zelle in einer Tabelle nur durch Angabe der Zeilen- sowie Spaltennummer. Dieser Ansatz ist nicht sehr robust, da sich diese numerischen Werte durch Umsortierung der Tabelle und/oder Hinzufügen/Entfernen von Einträgen ändern können.

  Da in jedem Testfall die Verarbeitung von Tabelleneinträgen vorkommt, wurde eine zusätzliche Methode implementiert, die das Auswählen einer Zelle auf einem höheren Abstraktionsniveau erlaubt. Diese Methode nimmt den Spaltennamen sowie den Zellinhalt als Parameter entgegen und identifiziert auf deren Basis dynamisch die richtige Zeilen- sowie Spaltennummer.

- **Eingeschränkte Erweiterbarkeit:** Zur Simulation von jenen Interaktionen mit SwingX-Komponenten, die nicht in den entsprechenden Standard-Swing-Komponenen vorgesehen sind, wäre das Erstellen zusätzlicher `Fixture`-Klassen durch Ableitung bestehender Fixtures erforderlich. Dies ist in FEST nicht vorgesehen, da bei der Deklaration von Methoden und Variablen durchgehend der Sichtbarkeitsmodifikator `private` verwendet wurde und viele Klassen oder Variablen mit dem Attribut `final` versehen sind.

- **Übereinstimmende Fenstertitel:** Bei der Umsetzung von Testfall *#5* weisen die Fenster in Schritt *(2)* sowie *(4)* des Standardablaufs den identischen Titel *Ergebnisliste erstellen* auf. Hinzu kommt, dass beide Fenster am Bildschirm geöffnet sind. Es liegen nicht genügend Informationen vor, um die Fenster mit FEST voneinander zu differenzieren, weshalb die Umsetzung dieses Testfalls scheiterte.

Wartbarkeit

Die Wartbarkeit der erstellten Tests ist stark von den Programmierkenntnissen des Testers abhängig. Sofern bestehenden Programmierparadigmen und Best Practices gefolgt wird, können mit FEST gut wartbare Tests erstellt werden. Besonders wichtig für eine hohe Wartbarkeit ist, häufig benötigte Konstrukte in eine gemeinsame Elternklasse auszulagern und somit mehrfach vorhandenen Code zu reduzieren.

Für das Schreiben sowie Warten der Tests ist die Kenntnis des Programmcodes der zu testenden Applikation erforderlich, da für die jeweiligen GUI-Komponenten die richtigen `Fixture`-Klassen ausgewählt werden müssen. Würde man für die Identifikation der Komponenten deren eindeutigen Namen heranziehen, müssten diese dem Tester ebenfalls bekannt sein bzw. sollten diese ausreichend (z.B. in den Testfallbeschreibungen) dokumentiert sein.

6.3.2 Marathon

Mit Marathon konnten alle fünf Testfälle vollständig umgesetzt werden. Im Zuge der Implementierung musste jedoch im Hauptfenster ausschließlich auf die Verwendung des Menüs zurückgegriffen werden, da die Wiedergabe von Interaktionen mit den SwingX-Komponenten (vor allem der Baumansicht, die eine Instanz von `JXTree` ist) nicht einwandfrei funktionierte. Zur Aufzeichnung der Interaktionen wurde die bereitgestellte Recorder-Applikation (siehe Abschnitt 5.3.2) verwendet, die Nachbearbeitung und Ausführung der Testskripts erfolgte über die Benutzeroberfläche von Marathon. Zusätzlich wurde auch die Ausführung über die Kommandozeile analysiert.

Tool-Setup und Automatisierung des Startvorgangs

Das Setup von Marathon verlief problemlos und gemäß seiner Dokumentation. Nach dem Download des ZIP-Archivs musste dieses lediglich entpackt und zur *Path*-Systemvariable hinzugefügt werden. Bevor mit der Automatisierung des Startvorgangs begonnen werden konnte, musste ein neues Testprojekt durch Angabe der folgenden Informationen konfiguriert werden:

- Name und Speicherort des neuen Projekts
- Ausführungseigenschaften der zu testenden Applikation (Pfad zur JAR-Datei, Name der ausführbaren Klasse). Die JAR-Datei der zu testenden Applikation muss innerhalb ihrer Projektstruktur ausgeführt werden, da sie Abhängigkeiten zu anderen Ordnern dieser besitzt. Marathon ermöglicht die Spezifikation eines *Working Directories*, wodurch die Ausführung der Applikation innerhalb ihrer Projektstruktur sichergestellt wird.
- Auswahl der Skriptsprache für die Aufzeichnung der Testskripts (an dieser Stelle wurde Python als Skriptsprache gewählt).

Nach erfolgreicher Konfiguration des neuen Marathon-Projekts wurden einige Ordner sowie Dateien automatisch erstellt:

- **Fixtures:** Eine *Fixture* definiert Ausführungseigenschaften für einen oder mehrere Testfälle. Bei Erstellung eines neuen Projekts wird eine Standard-Fixture generiert, die bei Bedarf adaptiert werden kann. Eine Fixture enthält außerdem `setup`- sowie `teardown`-Methoden, die vor bzw. nach jedem Testskript, das die entsprechende Fixture verwendet, ausgeführt werden.
- **Modules:** Ein *Module* kapselt Funktionen und kann innerhalb der Testskripts referenziert werden. Die Erstellung solcher Module trägt somit zur Codewiederverwendung bei und ermöglicht die Modularisierung häufig benötigter Interaktionen.
- **TestCases:** In diesem Ordner werden später die Testskripts erstellt.
- **TestData:** In diesem Ordner kann später eine beliebige Anzahl an CSV-Dateien mit Testdaten abgelegt werden.

Der Startvorgang der Applikation konnte mit Marathon vollständig und ohne Probleme durch Aufzeichnung der entsprechenden Interaktionen abgebildet werden. Im Anschluss wurde das Skript in ein eigenes Modul ausgelagert und konnte somit in alle nachfolgenden Testskipts referenziert werden.

Exemplarischer Testfall

Nachfolgendes Listing 6.5 beinhaltet einige Ausschnitte aus der Implementierung des Testfalls *#3 Einzelnennung durchführen*. Die Zeilen 1 bis 5 werden als Header bezeichnet und sind wichtiger Bestandteil jedes Marathon-Skripts. Sie legen fest, welche Fixture (Zeile 2) verwendet sowie welche Module (in diesem Fall die `login`-Funktion des Login Moduls) importiert werden sollen. Zusätzlich kann über den Befehl `use_data_file` (Zeile 4) der Name einer Testdatendatei spezifiziert werden, sofern es sich um ein datengetriebenes Testskript handelt.

```
1  #{{{ Marathon
2  from default_with_cleanup import *
3  from Modules.LoginModule import login
4  use_data_file('TestData_AddIndividualEntry.csv')
5  #}}} Marathon
6
7  def test():
8    #Ausführung des Login Moduls
9    if window('Achtung!'):
10     login()
11   close()
12
13   #(1) aus Standardablauf von Anwendungsfall #3
14   if window('/\(c\)sportdata GmbH .*'):
15     select_menu('Datei>>Nennungen>>Einzelstarter')
16
17     #(2) aus Standardablauf von Anwendungsfall 3
18     if window('Nennung Einzelstarter'):
19       click('Verein hinzuf\xfcgen')
20     close()
21
22   close()
23
24   #(1) aus Standardablauf von Anwendungsfall #3.1
25   if window('Verein Manager'):
26     select('Verein:*', club_name)
27     select('Abk\xfcrzung:*', short_club_name)
28
29   ...
30
31   #Verifikation, ob Einzelnennung korrekt durchgeführt wurde
32   if window('Nennungen'):
33     assert_p('o', 'Text', club_name+'('+short_club_name+')', '{0, Verein}')
34     assert_p('o', 'Text', lastname+' '+firstname, '{0, Name}')
35   close()
```

Quellcode 6.5: Ausschnitt aus dem Marathon-Testskript *#3 Einzelnennung durchführen*

In Zeile 10 wird die `login`-Funktion des Login Moduls aufgerufen und somit der Startvorgang der Applikation initiiert. Im Recorder kann auf alle bereits bestehenden Module zugegriffen werden, sodass die bereits vorhandenen Interaktionen nicht erneut aufgezeichnet werden müssen.

Aufgrund des dynamischen Fenstertitels im Hauptfenster musste die Aufzeichnung in Zeile 14 manuell nachbearbeitet werden, sodass anstelle des vollständigen Titels ein regulärer Ausdruck Anwendung findet.

In den Zeilen 33 und 34 ist besonders das erste Argument der `assert_p`-Funktion interessant. Marathon fertigt im Zuge des Recordings sogenannte *Object-Maps* an. Dabei handelt es sich um Dateien, in denen sämtliche Eigenschaften aller GUI-Komponenten, mit denen während der Aufzeichnung eine Interaktion stattgefunden hat, in Form von *(Key, Value)*-Paaren abgelegt werden. Jede Komponente erhält dabei einen zufällig zugewiesenen eindeutigen Namen (im Falle des Listings ist dieser ein *o*), über den sie im Testskript referenziert werden kann. Im Rahmen der Skriptwiedergabe werden die Object-Maps zur Identifikation der GUI-Komponenten am Bildschirm herangezogen.

Nachfolgendes Listing 6.6 zeigt den Aufbau der Object-Map für die im vorhergehenden Listing referenzierte Komponente *o*. Marathon differenziert dabei zwischen für die Identifikation einer GUI-Komponente zwingend erforderlichen (`componentRecognitionProperties`) und zu Informationszwecken mitgespeicherten (`generalProperties`) Eigenschaften. Welche Eigenschaften in der Object-Map abgelegt werden und welche Priorität ihre Erfüllung bei der Identifikation der GUI-Komponenten hat, kann in einer separaten Konfigurationsdatei festgelegt werden.

```
!!net.sourceforge.marathon.objectmap.OMapComponent
  componentRecognitionProperties:
    - {method: equals, name: fieldName, value: o}
    - {method: equals, name: type, value: org.jdesktop.swingx.JXTable}
  generalProperties:
    - {name: position, value: '[x=151,y=141]'}
    - {name: type, value: org.jdesktop.swingx.JXTable}
    - {name: enabled, value: 'true'}
    - {name: instanceOf, value: javax.swing.JTable}
    - {name: size, value: '[width=978,height=575]'}
  name: o
```

Quellcode 6.6: Marathon Object-Map einer Tabelle mit Namen *o*

Support für datengetriebenes Testen

Das Erstellen datengetriebener Tests mit Marathon erforderte die manuelle Nachbearbeitung der aufgezeichneten Testskripts. Dabei mussten die tatsächlichen Daten in eine neue CSV-Datei ausgelagert und im Testskript durch entsprechende Platzhalter-Variablen ersetzt werden.

Positive Aspekte

Nachfolgend werden jene Aspekte gelistet, die in Marathon besonders gut umgesetzt sind:

- Marathon verfügt über ein umfangreiches und sehr gut strukturiertes Benutzerhandbuch. Somit wird eine schnelle Einarbeitung gewährleistet und ein fundiertes Verständnis der Funktionsweise von Marathon möglich.
- Die Recorder-Applikation läuft sehr stabil und ermöglicht aufgrund der Wiederverwendbarkeit bestehender Module ein rasches Aufzeichnen neuer Testskripts.
- Bei jeder Ausführung der Testskripts wird automatisch ein Report im HTML-Format erstellt. Nachfolgende Abbildung 6.5 zeigt einen solchen Report, bei dem alle Testskripts erfolgreich durchgelaufen sind:

SET_Testing - Results

Generated on **Wednesday Jan 22 16:40:9 2014**

Summary

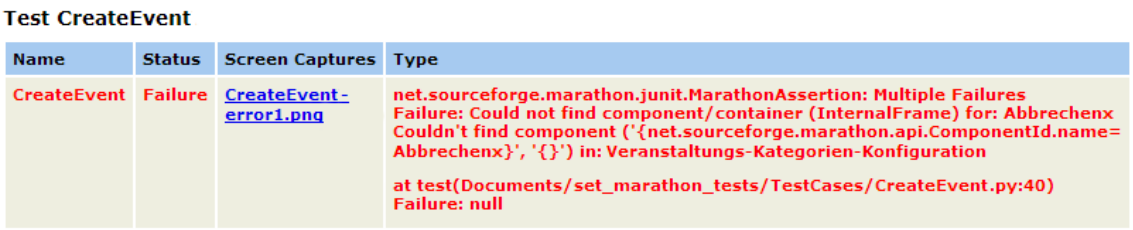
Tests	Failures	Errors	Success rate	Time
3	0	0	100.00%	NaN

Packages

Name	Tests	Failures	Errors	Time(s)
AddIndividualEntry	1	0	0	NaN
CreateCategory	1	0	0	NaN
CreateEvent	1	0	0	NaN

Abbildung 6.5: HTML-Report nach erfolgreichem Durchlauf der Marathon-Testskripts

Tritt während der Ausführung eines Testskripts ein Fehler auf, wird automatisch ein Screenshot erstellt und in den Report eingebettet. Weiters wird die Fehlerursache sowie die Zeile des Testskripts, in welcher der Fehler ausgelöst wurde, angegeben. Abbildung 6.6 zeigt einen Ausschnitt eines Reports, bei dem die Ausführung eines Testskripts fehlgeschlagen ist. Dem Report kann entnommen werden, dass der Button mit der Bezeichnung *Abbrechenx* im Fenster *Veranstaltungs-Kategorien-Konfiguration* nicht gefunden werden konnte und dass sich die entsprechende Anweisung in Zeile 40 des Testskripts befand.



Test CreateEvent

Name	Status	Screen Captures	Type
CreateEvent	Failure	CreateEvent-error1.png	net.sourceforge.marathon.junit.MarathonAssertion: Multiple Failures Failure: Could not find component/container (InternalFrame) for: Abbrechenx Couldn't find component ('{net.sourceforge.marathon.api.ComponentId.name=Abbrechenx}', '{}') in: Veranstaltungs-Kategorien-Konfiguration at test(Documents/set_marathon_tests/TestCases/CreateEvent.py:40) Failure: null

Abbildung 6.6: HTML-Report nach Auftreten eines Fehlers bei der Ausführung eines Marathon-Testskripts

Problemstellen und Lösungsansätze

Nachfolgende Aufzählung listet sämtliche Probleme, die sich bei der Umsetzung der Testfälle in Marathon ergaben:

- **Zeichenkodierung:** Marathon kann standardmäßig nur mit dem englischen Zeichensatz umgehen. An jenen Stellen, wo Beschriftungen in der Benutzeroberfläche oder eingegebener Text regionale Sonderzeichen (z.B. die deutschen Umlaute) enthielten, brach das Testskript bei der Wiedergabe mit einer Fehlermeldung ab. Abhilfe schaffte das explizite Festlegen einer alternativen Zeichenkodierung im Header jedes Testskripts. Dennoch konnten aufgezeichnete Sonderzeichen nicht als solche im Testskript dargestellt werden, was in Listing 6.5 in den Zeilen 19 sowie 27 zu erkennen ist.
- **SwingX-Unterstützung:** Marathon kann keine High-Level-Interaktionen mit vielen Komponenten der SwingX-Erweiterung erfassen. Stattdessen werden sogenannte Roh-Events aufgezeichnet, also die Operationen mit Maus und Tastatur an konkreten *(X, Y)*-Koordinaten. Diese Art der Event-Verarbeitung hat negative Auswirkungen auf die Robustheit der Testskripts, da die Koordinaten von der Bildschirmauflösung, der Größe des Applikationsfensters sowie dem Layout der GUI-Komponenten zum Zeitpunkt der Aufzeichnung abhängig sind. Neben der Robustheit wird auch die Lesbarkeit stark beeinträchtigt, wie der folgende Ausschnitt aus dem Testskript für Testfall *#1* in Listing 6.7 zeigt:

```
1 click('popupButton')
2 click('org.jdesktop.swingx.JXMonthView_0', 7, 154, 16)
3 click('org.jdesktop.swingx.JXMonthView_0', 113, 62)
```

Quellcode 6.7: Aufzeichnung der Roh-Events mit Marathon

Der dargestellte Code selektiert ein Datum in einem `DatePicker`. Zur besseren Nachvollziehbarkeit zeigt Abbildung 6.7 den zugehörigen Ausschnitt der Benutzeroberfläche. Die

Beschriftung (1. bis 3.) innerhalb der Abbildung stellt eine Verbindung zu den Zeilennummern des Listings her. Nach Öffnen des `DatePickers` (1.) wird der Vorwärts-Button an Position (154, 16) sieben Mal betätigt (2.) und anschließend ein Datum (3.) durch einen einzelnen Klick ausgewählt.

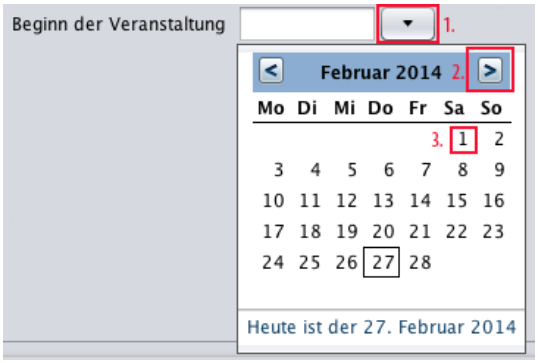


Abbildung 6.7: Ausschnitt aus der bestehenden Applikation, der die Interaktion mit einem `DatePicker` der SwingX-Erweiterung verdeutlicht

Weiters verursacht die Aufzeichnung der Roh-Events in diesem Beispiel funktionale Fehler. Der `DatePicker` zeigt nach dem Öffnen immer den aktuellen Monat an, demnach werden nicht zwingend sieben Klicks benötigt, um zum gewünschten Datum zu gelangen.

Um die Roh-Aufzeichnung von Events in nicht unterstützten GUI-Komponenten zu verhindern, müsste Marathon um zusätzliche Klassen erweitert werden. Diese werden als `Resolver` bezeichnet und definieren spezifische Interaktionen für eine bestimmte GUI-Komponente.

- **Abstraktion in Tabellen:** Marathon zeichnet bei Interaktionen mit Tabellen die Zeilennummer anstelle des Zellinhalts auf (siehe Listing 6.8). Dieser geringe Abstraktionsgrad erfordert die Anpassung des Skripts bei Änderung der Anzahl und/oder Sortierung der Einträge in der Tabelle.

```
1 select('k', 'rows: [15], columns: [Bezeichnung]')
```

Quellcode 6.8: Auswahl der Zeile 15 in einer Tabelle mit Marathon

Die Zeilennummer kann zwar ebenfalls in die Testdatendatei ausgelagert werden, erfordert aber weiterhin das Wissen des Testers, in welcher Zeile sich der auszuwählende Inhalt befindet.

- **Manuelle Nachbearbeitung:** Die Recorder-Applikation erstellt nicht immer fehlerfrei ausführbare Testskripts und macht somit ihre manuelle Nachbearbeitung erforderlich. Vor allem beim Öffnen/Schließen mehrerer aufeinanderfolgender Fenster sowie bei Interaktionen mit Tabellen kam es bei der Aufzeichnung der Testfälle vor, dass Events in der falschen Reihenfolge und/oder mehrfach erfasst wurden.

Wartbarkeit

Die Wartbarkeit der mit Marathon erstellten Testskripts ist eher gering. Obwohl es sich um ein Capture/Replay-Werkzeug handelt, muss ein Tester fundierte Programmierkenntnisse und zumin-

dest rudimentäre Erfahrung mit Python besitzen, damit bestehende Testskripts nachvollzogen und gegebenenfalls angepasst werden können. Zusätzlich verursacht die Erweiterung bestehender Skripts (z.B. bei Hinzukommen eines zusätzlichen Textfeldes) einen hohen Zeitaufwand, da eine Anpassung der Object-Maps erforderlich ist, was ausschließlich während des Capturing-Vorgangs erfolgen kann. Somit muss der gesamte Interaktionsablauf des betroffenen Testskripts neu durchgespielt werden.

Besonders negativen Einfluss auf die Wartbarkeit hat auch die zufällige Wahl der Bezeichner für die Object-Maps. So ist der Typ einer Komponente für den Tester nicht immer auf den ersten Blick erkennbar. Da weiters für jede Komponente eine eigene Object-Map-Datei angelegt wird, gestaltet es sich schwierig, die zu einem Bezeichner gehörige Object-Map schnell aufzufinden.

Auf die Aufzeichnung von Roh-Events sollte auch aus dem Blickwinkel der Wartbarkeit verzichtet werden. Einerseits ist für einen Tester zu einem späteren Zeitpunkt vollkommen unklar, welche Aktionen die entsprechenden Anweisungen repräsentieren, andererseits ist aufgrund der geringen Robustheit solcher Anweisungen eine häufige Wartung der Testskripts vorhersehbar.

6.3.3 Abbot

Wie bereits in Abschnitt 5.3.3 erwähnt, verfügt Abbot sowohl über ein Capture-Replay-Werkzeug als auch über einen skriptbasierten Erstellungsansatz. Aufgrund dieser Besonderheit war vorgesehen, alle Testfälle mit dem Skript-Editor Costello sowie unter Verwendung der Java-API umzusetzen, um einen Vergleich der beiden Konzepte durchführen zu können. Das Zusammenspiel zwischen Costello und der zu testenden Applikation verursachte jedoch bereits beim Startvorgang gravierende Probleme, weshalb kein einziger Testfall umgesetzt werden konnte. Die bereitgestellte API ist eine JUnit-Erweiterung und ermöglichte die vollständige Implementierung der ersten drei Testfälle.

Tool-Setup und Automatisierung des Startvorgangs

Das Setup von Abbot gestaltete sich einfach und reibungslos. Wichtig war, beim Download des ZIP-Archivs die neueste Version 1.3.0 auszuwählen, da alle vorhergehenden Versionen noch keine Unterstützung für Java 7 boten.

Costello: Die erste Hürde bei der Aufzeichnung des Startvorgangs war, dass Costello das von der Applikation standardmäßig angewandte Nimbus-Look&Feel nicht unterstützte. Erwartet wurde stattdessen gemäß der Fehlermeldung das entsprechende Look&Feel des zugrundeliegenden Betriebssystems. Die zu testende Applikation erlaubt das Ändern der Look&Feel-Einstellungen über eine separate Konfigurationsdatei und ermöglichte somit die Lösung des Problems. Jedoch verursachte auch die Aufzeichnung der weiteren Schritte des Startvorgangs nicht nachvollziehbare Fehlermeldungen und infolgedessen die sehr eingeschränkte Verfügbarkeit der Costello-Funktionen. An dieser Stelle wurde deshalb entschieden, Costello nicht weiter zu betrachten, da die Praxistauglichkeit des Werkzeugs ohnehin nicht gegeben war.

API: Zur Verwendung der Abbot-API musste diese sowie das JUnit-Framework zum Build-Path des Projekts hinzugefügt werden. Zur besseren Modularisierung der Testfälle wurde analog zur Vorgehensweise bei FEST eine Vererbungshierarchie definiert. Außerdem musste bei der Automatisierung des Startvorgangs erneut das Erscheinen der beiden Dialoge unterbunden werden, da auch Abbot auf die vollständige Abarbeitung der `main`-Methode wartet, bevor andere Interaktionen mit der Benutzeroberfläche simuliert werden können.

Exemplarischer Testfall

Nachfolgendes Listing 6.9 zeigt einige Ausschnitte aus der Implementierung des Testfalls *#3 Einzelnennung durchführen*. Die Simulation von Benutzerinteraktion geschieht durch die bereits in Abschnitt 5.3.3 erwähnten `Tester`-Klassen (Zeile 41), die für jede standardmäßig in Swing verfügbare GUI-Komponente bereitgestellt werden. Die Methode `clickButton` (Zeile 29 bis 43) zeigt, wie das Auffinden von und Interagieren mit Elementen der Benutzeroberfläche funktioniert. Für jede Komponente muss ein neuer `Matcher` implementiert werden, der die Kriterien für ihre Identifikation auswertet. Im Falle des Buttons ist eine Übereinstimmung genau dann gegeben, wenn eine Komponente eine Instanz von `JButton` ist und die spezifizierte Beschriftung aufweist.

```
1  @Test
2  public void test_addEntry() {
3
4    //(1) aus Standardablauf von Anwendungsfall #3
5    JFrame mainWindow = findWindow("\\(c\\)sportdata GmbH .*");
6    clickMenu(mainWindow, "Datei|Nennungen|Einzelstarter");
7
8    //(2) aus Standardablauf von Anwendungsfall #3
9    JFrame addEntryFrame = findWindow("Nennung Einzelstarter");
10   clickButton(addEntryFrame, "Verein hinzufügen");
11
12   //(1) aus Standardablauf von Anwendungsfall #3.1
13   Frame clubManagerFrame = findWindow("Verein Manager");
14   enterText(clubManagerFrame, 1, "Sportverein");
15   enterText(clubManagerFrame, 2, "SV");
16
17   ...
18
19   //Verifikation, ob Einzelnennung korrekt durchgeführt wurde
20   JXTable table = findTable(competitorsFrame);
21   assertEquals(1, table.getRowCount());
22   assertEquals("Sportverein(SV)", table.getStringAt(0, 1));
23 }
24
25 private JFrame findWindow(String windowTitle) {
26   return (JFrame) getFinder().find(new WindowMatcher(windowTitle, true));
27 }
28
29 private void clickButton(Container parent, final String buttonText) {
30
31   //Auffinden der gesuchten Komponente (JButton)
32   JButton b = (JButton) getFinder().find(parent, new Matcher() {
33     @Override
34     public boolean matches(Component c) {
35       if(!(c instanceof JButton)) return false;
36       return ((JButton) c).getText().equals(buttonText);
37     }
38   });
39
40   //Durchführen der Interaktion (einfacher Klick)
41   JButtonTester buttonTester = new JButtonTester();
42   buttonTester.actionClick(b);
43 }
```

Quellcode 6.9: Ausschnitt aus dem Abbot-Testskript *#3 Einzelnennung durchführen*

Wie aus dem obigen Listing klar hervorgeht, werden für die Implementierung einer einfachen Interaktion bereits über zehn Zeilen Code benötigt. Zur Erhaltung der Lesbarkeit der Tests und zur Verhinderung von mehrfach identischem Code wurde jede benötigte Interaktionsform (`enterText`, `clickMenu`, `clickTable`, ...) in eine separate Methode ausgelagert.

Zur Verifikation (Zeile 20 bis 22) der Ergebnisse werden die von JUnit bereitgestellten `assert`-Methoden verwendet.

Support für datengetriebenes Testen

Es wurde keine Möglichkeit zur Erstellung datengetriebener JUnit-Tests mit Abbot gefunden. Die Annotation der Testklassen (analog zu FEST) mit `@RunWith(Parameterized.class)` verursachte Abbot-spezifische Fehlermeldungen bei der Ausführung. Eine entsprechende Anfrage an die Abbot-Mailingliste wurde nicht beantwortet.

Positive Aspekte

Die folgenden positiven Eigenschaften von Abbot sind erwähnenswert:

- Ist der hohe Initialaufwand für das Kapseln sämtlicher Interaktionsformen in separate Methoden einmal investiert, können mit Abbot sehr rasch neue Testfälle erstellt werden. Zusätzlich weisen die Testfälle dadurch einen sehr hohen Abstraktionsgrad auf, dass sie die tatsächliche Interaktion delegieren und somit unabhängig gegenüber Änderungen der Abbot-API sind.
- Abbot stellt die Klasse `Strings` zur Internationalisierung von Testfällen durch Verwendung von *i18n*-Sprachdateien bereit. Dadurch kann die Unabhängigkeit der Testfälle von der Sprache der Benutzeroberfläche erzielt werden. Listing 6.10 zeigt die Konfiguration (Zeile 2 und 3) und Erstellung (Zeile 6) internationalisierter Testfälle.

```
1 //Einbinden des Bundles in der setup-Methode
2 Strings.addBundle(ResourceBundle.getBundle("abbot/mybundle",
3     Locale.GERMAN));
4
5 //Verwendung des Bundles in einem Testfall
6 clickButton(eventWindow, Strings.get("eventwindow.nextbutton"));
```

Quellcode 6.10: Internationalisierung eines Abbot Testfalls

Problemstellen und Lösungsansätze

Bei der Implementierung der Testfälle wurden die folgenden Schwachstellen in Abbot identifiziert:

- **Veraltete JUnit-Version:** Die Abbot-API erweitert die Klasse `TestCase` des JUnit-Frameworks, welche vor allem in Version 3 eine zentrale Rolle bei der Deklaration von Testmethoden spielte. Dies lässt vermuten, dass die interne Funktionsweise von Abbot nach wie vor auf Version 3 des Frameworks aufbaut, auch wenn zur Ausführung der Testskripts bereits JUnit 4 verwendet wird. Dies ist vor allem deshalb verwunderlich, weil JUnit 4 gemäß Ottinger [54] bereits vor acht Jahren veröffentlicht wurde. Die angestellte Vermutung wird zusätzlich bestätigt durch die Tatsache, dass den Testmethoden der Text `test` vorangestellt werden muss. Außerdem würde dies erklären, warum die Annotation der Testklassen mit `@RunWith(Parameterized.class)` zu Fehlermeldungen führte.
- **Fehlerbehaftete Implementierung:** Die Implementierung der Abbot-API ist nicht gänzlich frei von Fehlern. Immer wieder kommt es zu einer `ClassNotFoundException`, wenn Einträge von Tabellen, Listen oder Comboboxen über ihren Namen und nicht den Index identifiziert werden sollen. Listing 6.11 zeigt, wie dieses Problem behoben werden konnte.

```
1  JComboBox c = (JComboBox) getFinder().find(...);
2  JComboBoxTester comboboxTester = new JComboBoxTester();
3
4  //verursacht Exception
5  comboboxTester.actionSelectItem(c, "AUSTRIA");
6
7  //Workaround durch Suchen des korrespondierenden Indices
8  for(int i = 0; i < c.getItemCount(); i++){
9    if(c.getItemAt(i).equals("AUSTRIA")){
10     comboboxTester.actionSelectIndex(c, i);
11     break;
12   }
13 }
```

Quellcode 6.11: Identifikation von Einträgen per Name

Ebenfalls fragwürdig ist die Implementierung der Klasse `MultiMatcher`. Während ein normaler `Matcher` wie in Listing 6.9 exakt eine Komponente auffinden kann, ermöglicht ein `MultiMatcher` beliebig viele Komponenten gleichzeitig auszuwählen (beispielsweise alle Buttons in einem Fenster). Für das Abspeichern der aufgefundenen Komponenten verwendet die `MultiMatcher`-Klasse intern eine Datenstruktur mit Hashfunktion, wodurch die Anordnungsreihenfolge der Komponenten im Fenster verloren geht.

- **Dokumentation:** Vor allem der skriptbasierte Ansatz von Abbot ist nahezu undokumentiert. Es gibt kein Benutzerhandbuch, lediglich einige Anmerkungen in der Spezifikation der API. Teilweise fehlen notwendige Informationen für die effektive Nutzung der bereitgestellten Klassen und Methoden. So war es erforderlich, den Code mit einem Debugger zu durchlaufen, damit herausgefunden werden konnte, welches Trennzeichen für Pfadangaben in Menüs verwendet werden muss.
- **Look&Feel-Wechsel:** Obwohl Abbot mit der Unabhängigkeit seiner Tests vom Aussehen der Benutzeroberfläche wirbt, scheinen gewisse Abhängigkeiten zum Look&Feel der Komponenten zu bestehen. Die Umsetzung von Testfall *#4* scheiterte, da für die Darstellung der generierten Auslosung ein anderes Look&Feel zum Einsatz kommt.
- **Übereinstimmende Fenstertitel:** Wie schon bei FEST waren auch bei Abbot die übereinstimmenden Fenstertitel die Ursache für das Scheitern der Umsetzung von Testfall *#5*.

Wartbarkeit

Die Wartbarkeit der mit Abbot erstellen Testskripts wird maßgeblich von dem gewählten Design der Testklassen und infolgedessen von den Programmiererfahrungen des Testers beeinflusst. Besonders positiv auf die Wartbarkeit wirkt sich das zuvor angesprochene Auslagern häufig benötigter Interaktionsformen in separate Methoden aus, weil dadurch mehrfach vorhandener Code reduziert und der Abstraktionsgrad der Testmethoden erhöht wird.

Die Internationalisierung der Tests fördert die Wartbarkeit ebenfalls, da im Idealfall direkt die Sprachdateien der zu testenden Applikation eingesetzt werden können und somit das Ändern von Beschriftungen in der Benutzeroberfläche oder der Wechsel der Sprache keine Anpassung der Testskripts erforderlich macht.

6.3.4 Sikuli

Wie bereits in Abschnitt 5.3.4 erörtert, verfolgt Sikuli einen auf Screenshots basierenden GUI-Testing-Ansatz, wodurch Technologieunabhängigkeit und die einfache Erstellung der Testfälle gewährleistet werden soll. Mit Sikuli konnten zwar alle fünf Testfälle umgesetzt werden, es mussten jedoch an mehreren Stellen Einschränkungen in Kauf genommen bzw. Lösungsansätze für Probleme entworfen werden. Diese werden an späterer Stelle näher ausgeführt.

Tool-Setup und Automatisierung des Startvorgangs

Die Installation und Konfiguration von Sikuli gestaltete sich komplex. Nach Ausführung der dem Download beigelegten Setup-Batchdatei können unterschiedliche Installationspakete gewählt werden, abhängig davon, ob Python, Jython oder Java genutzt werden soll bzw. welche Entwicklungsumgebung (SikuliX IDE, Eclipse, Netbeans) bevorzugt wird. Die Pakete sind teilweise exklusiv und die Auswahl der richtigen Optionen stellt eine Herausforderung für einen unerfahrenen Benutzer dar.

Für die Umsetzung der Testfälle schien es aufgrund der nachfolgend genannten Aspekte sinnvoll, die Java-basierte Variante von Sikuli in Kombination mit JUnit zu verwenden:

- Kein Einarbeitungsaufwand in eine neue Entwicklungsumgebung und Programmiersprache.
- Gleiche Programmiersprache für Applikation und zugehörige Tests.
- Integrationsfähigkeit der JUnit-Testklassen in das bestehende Eclipse-Projekt.
- Wiederverwendung des bereits bei FEST und Abbot bewährten Vererbungskonstrukts (siehe Abbildung 6.3).
- Identischer Funktionsumfang von Java-API und Python-Variante.

Zur Verwendung der Java-Variante musste lediglich der Installationsordner von Sikuli zum Build-Path des Projekts hinzugefügt werden. Die Automatisierung des Startvorgangs verlief problemlos. Erneut musste Gebrauch vom bereits bei FEST und Abbot erwähnten Test-Modus gemacht werden.

Exemplarischer Testfall

Nachfolgendes Listing 6.12 zeigt Ausschnitte aus der Implementierung von Testfall *#3 Einzelnennung durchführen*. Über die `focus`-Methode der Klasse `App` kann der Fokus auf ein am Bildschirm sichtbares Fenster gelegt werden. Besonders wichtig für das Wechseln des Fokus sind die `wait`-Anweisungen (Zeile 9 sowie 14). Diese sollen garantieren, dass das Fenster vor dem Fokuswechsel bereits am Bildschirm sichtbar ist.

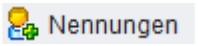
Das Problem des dynamischen Titels im Hauptfenster konnte hier nicht durch Verwendung eines regulären Ausdrucks gelöst werden. Sikuli unterstützt bislang keine regulären Ausdrücke, gemäß Hocke [26] soll diese Schwachstelle jedoch mit Release 1.2.0 Anfang 2015 behoben werden. Als temporäre Lösung wird stattdessen der vollständige dynamische Titel über eine Hilfsmethode (`getWindowTitle`) ermittelt.

```
1  @Test
2  public void addEntry(){
3
4    //(1) aus dem Standardablauf von Anwendungsfall #3
5    App.focus(getWindowTitle());
6    screen.type("d", Key.ALT); //Klick auf Datei
7    screen.click(new Pattern("UC3_menu_entries.png"));
8    screen.click(new Pattern("UC3_menu_singleentry.png"));
9    screen.wait(1.0);
10
11   //(2) aus Standardablauf von Anwendungsfall #3
12   App.focus("Nennung Einzelstarter");
13   screen.click(new Pattern("UC3_button_addclub.png"));
14   screen.wait(1.0);
15
16   //(1) aus Standardablauf von Anwendungsfall #3.1
17   App.focus("Verein Manager");
18   screen.type(new Pattern("UC3_textbox_clubname.png"), "Sportverein");
19   screen.type(new Pattern("UC3_textbox_clubnameshort.png"), "SV");
20
21   ...
22
23   //Verifikation, ob Einzelnennung korrekt durchgeführt wurde
24   assertNotNull(screen.exists(pattern("UC3_entrytable.png")));
25 }
```

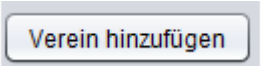
Quellcode 6.12: Ausschnitt aus dem Sikuli-Testskript *#3 Einzelnennung durchführen*

Zur Verifikation des Ergebnisses kann die JUnit-Zusicherung `assertNotNull` in Kombination mit der `exists`-Methode von Sikuli genutzt werden. Letztere stellt sicher, dass der spezifizierte Screenshot in dieser Form existiert und liefert andernfalls `null`.

Zur besseren Verständlichkeit des obigen Listings werden in der nachfolgenden Abbildung 6.8 einige der verwendeten Screenshots dargestellt. Diese wurden der zu testenden Applikation händisch vor Erstellung des Testskripts entnommen. Aufgrund der großen Anzahl an Screenshots muss besonders auf die ausdrucksstarke Benennung der Dateien geachtet werden.



(a) UC3_menu_entries



(b) UC3_button_addclub

Name	Verein	Geburtsdat...	Geschle...	Gewicht /...	Grösse / ...	Kyu	Dan
Mustermann Heinz	Sportverein(SV)	2000-05-05	m	75	190	0	0

(c) UC3_entrytable

Abbildung 6.8: Im Testskript verwendete Screenshots

Support für datengetriebenes Testen

Da es sich bei den Testskripts um JUnit 4 Testklassen handelt, ist die Erstellung datengetriebener Tests gemäß der bei FEST angesprochenen Vorgehensweise technisch möglich. Die Spezifikation beliebiger Testdaten in einer separaten Datei steht allerdings in Widerspruch zu dem Screenshot-basierten Ansatz von Sikuli, da – wie in Zeile 24 in Listing 6.12 bzw. Abbildung 6.8c zu sehen – auch die erwarteten Ergebnisse in Form von Screenshots definiert werden. Ändern sich also die Eingabedaten, müssten die zugehörigen Screenshots ausgetauscht werden, was einen erheblichen Mehraufwand für den Tester darstellt.

Positive Aspekte

Folgende positive Eigenschaften von Sikuli können genannt werden:

- Sikuli ermöglicht mittels einer Highlight-Funktion, die anhand der Screenshots identifizierten Objekte am Bildschirm zu markieren. Dadurch kann der Tester den Interaktionsablauf bei der Testausführung mitverfolgen und bei Fehlern nachvollziehen, wo das Problem liegt. Es handelt sich dabei um eine Art visuellen Debugger. Die Highlight-Funktion sollte nur bei Bedarf aktiviert werden, da sie die Ausführung der Anweisungen verlangsamt.
- Das Support-Forum wird aktiv betreut und innerhalb weniger Stunden beantwortet.
- Es werden keine umfassenden Programmierkenntnisse benötigt, da nur einige wenige Anweisungen zur Verfügung stehen. Weiters ist es für den Tester auch nicht erforderlich, Wissen über die interne Funktionsweise der Applikation oder den Typ der verwendeten GUI-Komponenten zu besitzen.

Problemstellen und Lösungsansätze

Beim Arbeiten mit Sikuli kam es zu den folgenden Problemen:

- **Fokusfehler bei Textfeldern:** Das Zusammenspiel von Sikuli und der Simulation von Eingaben in Textfelder der Applikation funktionierte nicht reibungslos. Betroffen waren alle Programmfenster, die mehr als ein Textfeld enthielten. Das gewünschte Textfeld wurde zwar korrekt identifiziert (dies wurde durch Einsatz der Highlight-Funktion geprüft), der Text aber dennoch in das oberste Textfeld eines Fensters eingegeben. Abbildung 6.9 visualisiert das Problem. Der blaue Rahmen entspricht der farbigen Markierung der Highlight-Funktion, das gewünschte Datum (rot umrandet) wird allerdings im Textfeld darüber eingegeben.

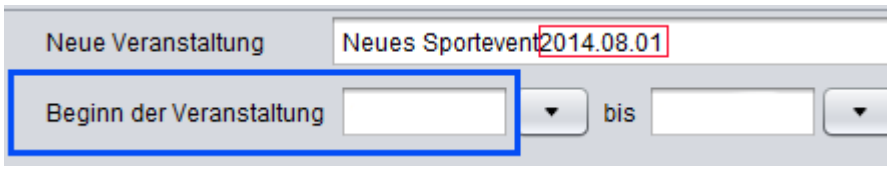


Abbildung 6.9: Fokusproblem bei Eingabe von Text in Textfelder

  Eine Diskussion mit dem Projektleiter Raimund Hocke im offiziellen Support-Forum brachte leider keinen Erkenntnisgewinn. Das Auftreten des Problems konnte durch Simulieren eines Tastendrucks auf die TAB-Taste und somit das explizite Setzen des Fokus in das nächste auszufüllende Textfeld unterbunden werden. Diese Lösung ist jedoch nicht praxistauglich, da sie die Robustheit der Tests gegenüber Änderungen in der Anordnung der Elemente beeinträchtigt.

- **Zeichensatz:** Sikuli kann nur mit den Zeichen des Standard-US-Zeichensatzes umgehen. Das Verwenden von Sonderzeichen wie Umlauten in der `type`-Methode verursachte eine `IllegalArgumentException`. Da für dieses Problem keine Lösung gefunden werden konnte, wurde das Eingeben von Sonderzeichen vermieden.

- **Einschränken der Suchregion:** Aus Performanzgründen wird dem Tester in der Sikuli-Dokumentation die Festlegung von Regionen nahegelegt. Diese schränken den von Sikuli beim Auffinden von Screenshots berücksichtigten Bereich auf ein Rechteck beliebiger Größe ein. Häufig wird das aktuell fokussierte Fenster als Region gewählt, was in nachfolgendem Listing 6.13 dargestellt wird:

```
1 App eventwindow = App.focus("Veranstaltungsdaten");
2 Region region = eventwindow.window();
3 region.click(new Pattern("UC3_textbox_eventname.png"));
```

Quellcode 6.13: Einsatz von Regionen zur Einschränkung des Suchbereichs

Abbildung 6.10 zeigt das Problem bei der Verwendung von Regionen für die Erstellung der Testfälle. Der blaue Rahmen entspricht den Regionsgrenzen, welche durch Verwendung der Highlight-Funktion sichtbar gemacht wurden. Es ist leicht zu erkennen, dass nur ein kleiner Bereich des Gesamtfensters erfasst wurde. Aufgrund dieses Problems wurde auf die Verwendung von Regionen weitgehend verzichtet.

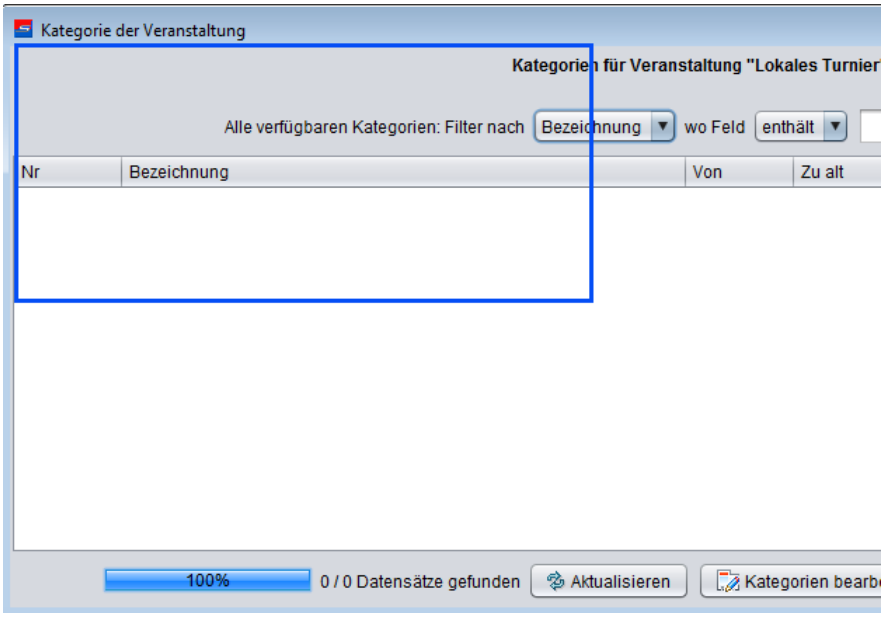


Abbildung 6.10: Fehlfunktion bei Definition der Region

- **Ähnlichkeit**: Befinden sich mehrere GUI-Komponenten gleichen Typs und mit ähnlicher Beschriftung im selben Fenster, hat Sikuli Probleme mit deren Identifikation. Dies hat damit zu tun, dass standardmäßig zwischen dem Screenshot und einem Element am Bildschirm nur eine Übereinstimmung von 70% gegeben sein muss. Listing 6.14 zeigt, wie die erforderliche prozentuelle Ähnlichkeit erhöht werden kann:

```
1 screen.find(new Pattern("UC5_combo_firstplace.png").similar(0.99f));
```

Quellcode 6.14: Setzen der erforderlichen Übereinstimmung auf 99%

Das Setzen einer 99%-igen Übereinstimmung war für die Umsetzung von Testfall *#5* Standardablauf *(4)* erforderlich. Wie in Abbildung 6.11 zu sehen, enthält das zugehörige Applikationsfenster vier Comboboxen, die sich nur in der Nummerierung der Plätze unterscheiden.

Abbildung 6.11: Ausschnitt aus Testfall *#5* Standardablauf *(4)*

Besonders problematisch war der Umgang mit den unteren beiden Comboboxen in der Abbildung, da diese vollständig identisch sind. Durch Verwendung der `matchAll`-Methode konnten diese zwar identifiziert werden, allerdings wurde die chronologische Reihenfolge nicht beibehalten. Die einzige Lösung bestand im Vergleich der Y-Koordinaten, was jedoch erneut keine besonders robuste Lösung darstellt.

- **Plattformabhängigkeit:** Die zu testende Applikation weist auf verschiedenen Plattformen geringfügige Unterschiede (Schriftart, Abrundung von Buttons, Top-Level-Fenster) auf. Wenn auch für einen menschlichen Betrachter nicht besonders auffällig, reichen diese Unterschiede bei Sikuli aus, um die Identifikation von GUI-Komponenten zu verhindern. Ein auf einem alternativen Betriebssystem aufgezeichneter Screenshot findet in der Applikation keine Wiedererkennung. Dies hat als Konsequenz, dass für die Aufzeichnung der Screenshots und Wiedergabe der Testskripts dieselbe Plattform verwendet werden muss.

Wartbarkeit

In Sikuli beschränkt sich der Begriff Wartbarkeit nicht ausschließlich auf die erstellten Testskripts, sondern umfasst auch die damit in Verbindung stehenden Screenshots. Der Code der Testskripts ist an sich gut wartbar, da er einen geringen Umfang aufweist und aus aussagekräftigen, für sich sprechenden Methodenaufrufen besteht. Die Wartung der Screenshots gestaltet sich weitaus aufwendiger. Layout-Änderungen in der Applikation können die Erneuerung aller Screenshots erforderlich machen, was mit einem hohen Zeitaufwand verbunden ist. Auch das Hinzufügen neuer Anweisungen zum Testskript oder das Ändern der Eingabedaten ist mit der Erstellung zusätzlicher Screenshots verbunden.

6.3.5 Robot Framework

Die ursprüngliche Motivation für die Verwendung des Robot Frameworks lag in der Kopplungsmöglichkeit an Sikuli durch Umwandlung der mit Sikuli erstellten Testskripts in eine projektspezifische Robot-Bibliothek (siehe Abschnitt 5.3.5). Dadurch sollten die Testskripts um einen datengetriebenen Ansatz und Reporting-Funktionen erweitert werden.

Nach Abschluss der Implementierungstätigkeiten mit Sikuli wurde jedoch erkannt, dass die Kopplung an das Robot Framework nicht zweckmäßig schien und keine zusätzlichen Vorteile bringen würde. Grund dafür waren die großen Abhängigkeiten der Sikuli-Skripts zu den Inhalten der

Screenshots, die somit eine Nutzen bringende Auslagerung der Testdaten verhinderten. Hinzu kam, dass durch die Verwendung der Java-basierten Variante von Sikuli die Fehleranalyse und die Möglichkeit zur Erstellung von Reports bereits durch das JUnit-Framework bzw. Ant abgedeckt wurden.

Aufgrund ihrer vielversprechenden Beschreibung wurde stattdessen beschlossen, die externe Test-Bibliothek *Swing Library* näher zu betrachten und für die Umsetzung der Testfälle in das Robot Framework einzubinden. Die Swing Library stellt sämtliche Schlüsselwörter bereit, die für das Testen einer Swing-Applikation benötigt werden. Mit dieser Bibliothek konnten insgesamt vier der fünf Testfälle erfolgreich implementiert werden.

Tool-Setup und Automatisierung des Startvorgangs

Das Setup von Robot Framework sowie die Einbindung der Swing Library gestalteten sich aufwendig, da zuvor die Installation einer Python- oder Jython-Entwicklungsumgebung sowie das Anpassen einiger Systemvariablen erforderlich waren.

Alternativ kann das Robot Framework auch ohne Installation über eine Standalone-JAR-Variante genutzt werden, was bei vorhandener Java-Entwicklungsumgebung erheblich weniger Konfigurations- und Installationsaufwand verursacht und denselben Funktionsumfang aufweist. Allerdings gestaltet sich bei der Standalone-Version die Einbindung externer Bibliotheken schwieriger.

In beiden Fällen ist es jedoch erforderlich, die Swing Library sowie die JAR-Datei der zu testenden Applikation temporär vor jeder Ausführung oder dauerhaft in die *Classpath*-Systemvariable aufzunehmen.

Bei der Automatisierung des Startvorgangs bereitete erneut das Auftreten der beiden Dialoge Probleme, weshalb wiederum der bereits bewährte Test-Modus zum Einsatz kam. Das Robot Framework ermöglicht die Spezifikation sogenannter Resource-Dateien. Diese haben einen ähnlichen Aufbau wie normale Testskripts, besitzen aber den Vorteil, dass sie von anderen Testskripts referenziert und ausgeführt werden können. Die Anweisungen zur Automatisierung des Startvorgangs wurden daher in eine solche Resource-Datei ausgelagert.

Exemplarischer Testfall

Tabelle 6.8 bildet einen Ausschnitt des mit Robot umgesetzten Testfalls *#3 Einzelnennung durchführen* im TSV-Format ab. Die zwei wesentlichen Bestandteile des Testskripts sind dabei die Einstellungen (`Setting`) sowie die Spezifikation der Testfälle (`TestCases`). In den Einstellungen erfolgt das Laden der zuvor angesprochenen Resource-Datei (Zeile 2) und im Anschluss daran die Ausführung des automatisierten Startvorgangs (Zeile 3) als Setup vor jedem Testfall.

Ein Testfall besteht immer aus einer textuellen Beschreibung (Zeile 6) und einer beliebigen Anzahl an Zeilen, die jeweils ein Schlüsselwort (Spalte `Action`) gefolgt von einer Menge an Argumenten beinhalten. Wie in Zeile 7 zu sehen ist, unterstützt Robot Framework auch die Verwendung regulärer Ausdrücke, wodurch das Problem mit dem dynamischen Fenstertitel gelöst werden konnte.

Die Identifikation eines GUI-Elements kann entweder über dessen Index (siehe Zeile 8/9 oder 13/14), seine in der Benutzeroberfläche sichtbare Beschriftung (z.B. beim Klick auf den Button in Zeile 11) oder durch den eindeutigen Namen, der im Programmcode für das Element gesetzt wurde, erfolgen.

```
1  *Setting*      *Value*
2  Resource       resource.txt
3  Test Setup     Login With Correct Credentials
4
5  *TestCases*    *Action*                 *Argument*                  *Argument*              *Argument*
6  Add Entry
7                 Select Window            regexp=.*sportdata GmbH.*
8                 Select Tree Node         0                           ${TC3_EVENT}|Nennungen
9                 Select From Popup Menu   0                           Nennung Einzelstarter
                  On Selected Tree Nodes
10                Select Window            Nennung Einzelstarter
11                Push Button              Verein hinzufügen
12                Select Window            Verein Manager
13                Type Into Text Field     1                           ${TC3_CLUBNAME}
14                Type Into Text Field     2                           ${TC3_CLUBNAME_SHORT}
15                ...
16                ${TC3_ROWCOUNT}=         Get Table Row Count         0
17                Should Be Equal          ${1}                        ${TC3_ROWCOUNT}
18                ${TC3_RESULTROW}=        Get Table Row Values        0                       0
19                ${TC3_C0}=               Get From List               ${TC3_RESULTROW}        0
20                Should Be Equal          ${TC3_C0}                   ${TC3_NAME}
```

Tabelle 6.8: Ausschnitt aus dem Robot-Testskript *#3 Einzelnennung durchführen*

Besonders interessant sind auch die Zeilen 16 bis 20, welche exemplarisch einige Zusicherungen zeigen. Bevor ein Vergleich zwischen erwartetem und tatsächlichem Ergebnis stattfinden kann, muss das tatsächliche Ergebnis in einer temporären Variable zwischengespeichert werden (siehe Zeile 16, 18 und 19).

Support für datengetriebenes Testen

Das Robot Framework ist auf datengetriebenes Testen ausgelegt. Die Testdaten können dabei entweder in einer separaten Python-Datei oder innerhalb einer bestehenden Resource-Datei spezifiziert werden. Handelt es sich um eine Python-Datei, muss diese zusätzlich – ähnlich wie eine Resource-Datei – über das Schlüsselwort `Variables` in den Einstellungen eingebunden werden. Nachfolgende Tabelle 6.9 zeigt den für die Spezifikation der Testdaten relevanten Ausschnitt aus einer Resource-Datei:

```
1  *Variable*                *Value*
2  ${TC3_EVENT}              Deutsche Meisterschaft
3  ${TC3_CLUBNAME}           Sportverein
4  ${TC3_CLUBNAME_SHORT}     SV
```

Tabelle 6.9: Datengetriebener Ansatz innerhalb einer Robot Resource-Datei

Der Variablenname wird dabei immer in geschwungenen Klammern und mit vorangestelltem $-Zeichen spezifiziert und kann im Testskript anstelle der tatsächlichen Daten angegeben werden (siehe vorhergehende Tabelle 6.8 in Zeile 8, 13 bzw. 14).

Alternativ zeigt das nachfolgende Listing 6.15 eine in Python erstellte Variablen-Datei.

```
1 TC3_EVENTNAME = "Deutsche Meisterschaft"
2 TC3_CLUBNAME = "Sportverein"
3 TC3_CLUBNAME_SHORT = "SV"
```

Quellcode 6.15: Datengetriebener Ansatz durch separate Python-Datei

Der wesentliche Vorteil einer Variablen-Datei ist, dass diese bei Bedarf erst bei der Ausführung eines Testskripts in der Kommandozeile über die zusätzliche Option `--variablefile <name>.py` spezifiziert werden kann. Somit wird es möglich, dasselbe Testskript ohne dessen Modifikation mit unterschiedlichen Testdatensätzen zu durchlaufen.

Positive Aspekte

Nachfolgend werden die besonders gut umgesetzten Aspekte des Robot Frameworks diskutiert:

- Das Robot Framework und auch die Swing Library verfügen über eine ausgezeichnete und umfangreiche Benutzerdokumentation. Diese Dokumente sind zusätzlich für jede Produktversion separat erhältlich, sodass sie nur jene Inhalte enthalten, die für die jeweilige Version auch tatsächlich zutreffend sind. Weiters wird für jede standardmäßig verfügbare oder externe Test-Bibliothek eine Schlüsselwortdokumentation bereitgestellt. Dabei handelt es sich um eine kompakte Webseite, die alle verfügbaren Schlüsselwörter mitsamt ihren erwarteten Argumenten beschreibt.
- Für die Erstellung von Testskripts sind keinerlei Programmierkenntnisse erforderlich, zusätzlich muss kein internes Wissen über die zu testende Applikation vorliegen, sofern der Typ der GUI-Elemente über die Benutzeroberfläche ersichtlich ist.
- Bei jeder Ausführung eines Testskripts über die Kommandozeile werden automatisch ein Report sowie eine Log-Datei im HTML-Format angefertigt. Der Report enthält nur High-Level-Informationen darüber, welche Tests ausgeführt wurden und ob diese erfolgreich waren. Abbildung 6.12 zeigt einen Ausschnitt eines Reports:

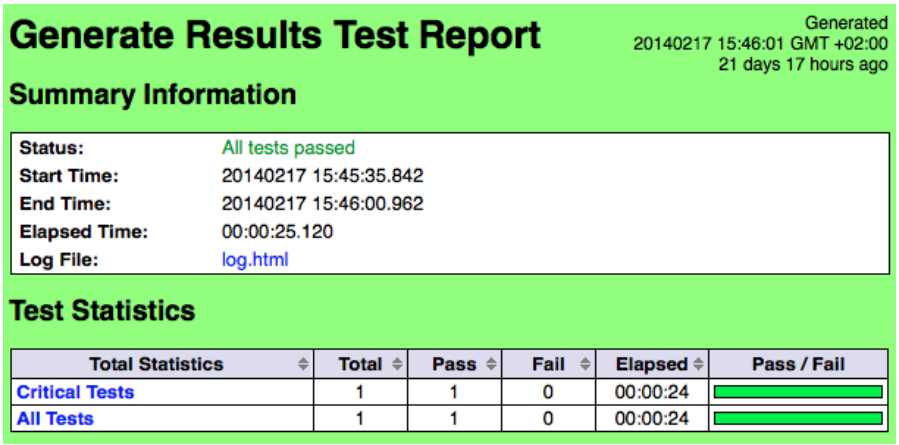


Abbildung 6.12: Ausschnitt einer von Robot Framework erstellten Report-Datei

Im Gegensatz dazu enthält die Log-Datei detaillierte Informationen über die ausgeführten Anweisungen. Im Falle eines Fehlers kann somit nachgesehen werden, bei welchem Schlüsselwort dieser aufgetreten ist und was die Ursache dafür war. Abbildung 6.13 zeigt eine Log-Datei eines fehlgeschlagenen Tests, bei dem der Button mit der Bezeichnung *Not Existing* nicht gefunden werden konnte.

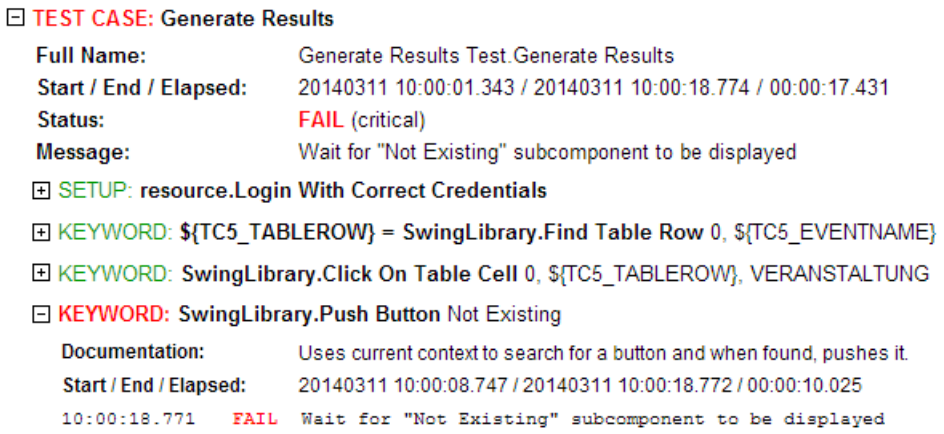


Abbildung 6.13: Ausschnitt einer von Robot Framework erstellten Log-Datei

- Der generische Ansatz des Robot Frameworks gibt dem Tester einen hohen Grad an Flexibilität und Anpassbarkeit. Durch nur eine einzelne Anweisung in den Einstellungen eines Testskripts können neue Bibliotheken eingebunden und somit ein Set an zusätzlichen Funktionen genutzt werden.
- Das Swing Library-Projekt ist gut dokumentiert. Das Auschecken des Source-Codes von Github, das Einbinden des Projekts in Eclipse und anschließendes Packen einer neuen JAR-Version über Maven funktionierte reibungslos und gemäß der bereitgestellten Anleitung.

Problemstellen und Lösungsansätze

Beim Einsatz des Robot Frameworks bzw. der Swing Library entstanden die folgenden Probleme:

- **Reguläre Ausdrücke:** Das Verwenden regulärer Ausdrücke funktioniert nur im Zusammenhang mit den Schlüsselwörtern `Select Window` bzw. `Select Dialog`. Für Buttons oder andere GUI-Komponenten können keine regulären Ausdrücke spezifiziert werden, was bei Testfall *#4* zu Problemen führte. Dort beinhaltete ein Button mehrere Leerzeichen vor bzw. nach der eigentlichen Beschriftung. Als Lösung konnte der betroffene Button nur über seinen Index angesprochen werden.
- **JFrame-Limitierung:** Das Schlüsselwort `Select Window` ist auf das Erkennen von Programmfenstern vom Typ `javax.swing.JFrame` beschränkt. Bei Testfall *#2* und *#3* wurde für die Erstellung einiger Fenster jedoch die Klasse `java.awt.Frame` instanziert, weshalb diese über die Swing Library nicht angesprochen werden konnten. Zur Lösung des Problems musste der Source-Code der Swing Library ausgecheckt und adaptiert werden. Von den Änderungen betroffen waren die Klassen `FrameOperator` sowie `ByNameOrTitleFrameChooser`. Listing 6.16 zeigt grün hinterlegt die Anpassungen, die in der Klasse `FrameOperator` erforderlich waren.

```
1  public class FrameOperator
2    //zuvor JFrameOperator
3    extends org.netbeans.jemmy.operators.FrameOperator
4    implements ComponentWrapper {
5
6    ...
7    private static ComponentChooser createRegExpChooser(String title) {
8      //zuvor JFrameFinder
9      return new FrameFinder(...);
10   }
11 }
```

Quellcode 6.16: Änderungen in der Klasse `FrameOperator`

In der Klasse `ByNameOrTitleFrameChooser` musste die Methode `titleMatches` erweitert werden, sodass nun auch Instanzen von `Frame` verarbeitet werden können (siehe Listing 6.17).

```
1  public class ByNameOrTitleFrameChooser implements ComponentChooser {
2    ...
3    private boolean titleMatches(Component c) {
4
5      if(c instanceof JFrame || c instanceof Frame) {
6
7        String title = (c instanceof JFrame) ?
8            ((JFrame) c).getTitle() : ((Frame) c).getTitle();
9
10       return ObjectUtils.nullSafeEquals(title, expectedNameOrTitle);
11     }
12
13   }
14 }
```

Quellcode 6.17: Änderungen in der Klasse `ByNameOrTitleFrameChooser`

- **Fokussieren von JOptionPanes:** Die Swing Library stellt keine Schlüsselwörter zur Selektion von GUI-Komponenten, die eine Instanz von `javax.swing.JOptionPane` sind, bereit. Dieses Problem konnte durch Anpassung der Swing Library nicht behoben werden, da bereits das der Bibliothek zugrundeliegende Werkzeug Jemmy Probleme bei Interaktionen mit diesen Dialogen verursacht (siehe [2]). Dies verursachte das Scheitern von Testfall *#4*, bei dem eine Instanz von `JOptionPane` geschlossen werden musste.

Wartbarkeit

Die mit dem Robot Framework bzw. der Swing Library erstellten Testskripts weisen eine sehr gute Wartbarkeit auf. Die Schlüsselwörter sind aussagekräftig und auch ohne das Schreiben von Kommentaren nachvollziehbar. Einem Tester sollte es innerhalb kürzester Zeit möglich sein, sich in das Framework sowie bestehende Testskripts einzuarbeiten und Ergänzungen oder Änderungen schnell umzusetzen.

Durch die Auslagerung der Testdaten in separate Variablen- oder Resource-Dateien können diese unabhängig von den Skript-Anweisungen gewartet werden, sofern die Variablennamen gleich bleiben.

Einen schlechten Einfluss auf die Häufigkeit der Wartung haben die Verwendung von Indices sowie Beschriftungen aus der Benutzeroberfläche zur Identifikation der GUI-Elemente. Hier sollte, sofern in der zu testenden Applikation vorhanden und dem Tester bekannt, auf die eindeutigen Bezeichner zurückgegriffen werden.

6.3.6 WindowTester Pro

Mit WindowTester Pro konnten vier der fünf Testfälle erfolgreich implementiert werden. Zur Erstellung der Testskripts wurden die Interaktionen zuerst mit der bereits in Abschnitt 5.3.6 vorgestellten Recording Console aufgezeichnet und anschließend unter Verwendung des Code Generation Wizards als JUnit-Testklassen exportiert. Im Anschluss war die manuelle Anpassung des generierten Codes sowie die Erweiterung der WindowTester Pro-Implementierung an zahlreichen Stellen erforderlich, damit die fehlerfreie Wiedergabe der Testskripts sowie eine gute Modularisierung des Codes sichergestellt werden konnte.

Tool-Setup und Automatisierung des Startvorgangs

Die Installation von WindowTester Pro gestaltete sich sehr einfach und komplikationslos. Das Plugin konnte durch Angabe der Update-Seite direkt innerhalb von Eclipse bezogen werden und benötigte keine weitere Konfiguration. Auch die Aufzeichnung des Startvorgangs mit WindowTester Pro verursachte keine Probleme, von dem Testmodus zur Unterbindung der Dialoge musste nicht Gebrauch gemacht werden.

Nach Export der Aufzeichnung als JUnit-Testklasse wurde diese restrukturiert, sodass erneut das bewährte Vererbungskonstrukt aus Abbildung 6.3 entstand. Der automatisierte Startvorgang wurde also wiederum in eine `setup`-Methode ausgelagert und somit für alle nachfolgenden Testklassen verfügbar gemacht.

Exemplarischer Testfall

Nachfolgendes Listing 6.18 enthält einige Ausschnitte aus der nachbearbeiteten Aufzeichnung des Testfalls *#3 Einzelnennung durchführen*. Bereits am Methodenkopf lässt sich erkennen, dass die generierten Testklassen auf JUnit 3 basieren, da anstelle der `@Test`-Annotation der Methodenname mit der Bezeichnung `test` beginnt.

```
1  public void testAddEntry() {
2    //(1) aus dem Standardablauf von Anwendungsfall #3
3    ui.click(new JMenuItemLocator(JMenuItem.class,
4        "Datei/Nennungen/Einzelstarter",
5        new SwingWidgetLocator(JMenu.class, 0, null)));
6
7    //(2) aus dem Standardablauf von Anwendungsfall #3
8    ui.wait(new WindowShowingCondition("Nennung Einzelstarter"));
9    ui.click(new JButtonLocator("Verein hinzufügen"));
10
11   //(1) aus dem Standardablauf von Anwendungsfall #3.1
12   ui.wait(new WindowShowingCondition("Verein Manager"));
13   ui.enterText(clubName);
14   ui.click(new LabeledTextLocator("Abkürzung:*"));
15   ui.enterText(clubShortName);
16
17   //Verifikation, ob Einzelnennung korrekt durchgeführt wurde
18   IWidgetLocator loc = ui.find(new JXTableItemLocator(new Point(0, 0)));
19   JXTable table = (JXTable) ((IWidgetReference) loc).getWidget();
20
21   assertEquals(1, table.getRowCount());
22   assertEquals(clubName+"("+clubShortName+")", table.getValueAt(0, 1));
23 }
```

Quellcode 6.18: Ausschnitt aus dem WindowTester Pro-Testskript *#3 Einzelnennung durchführen*

Das Auffinden von GUI-Komponenten erfolgt bei WindowTester Pro über sogenannte `Locator`-Klassen (siehe Zeilen 4, 6, 10 und 15). Diese erhalten bei ihrer Instanzierung als Argumente die für die Identifikation eines bestimmten Komponententyps benötigten Informationen. Meist ist die am Bildschirm sichtbare Beschriftung (Zeile 10 und 15) für die eindeutige Identifikation ausreichend, bei komplexeren `Klassen` wie dem `JMenuItemLocator` werden zusätzlich noch Informationen zum Elternelement benötigt.

Die Verifikation der Ergebnisse musste nachträglich von Hand implementiert werden, da mit der Recording Console von WindowTester Pro nur sehr limitierte und verallgemeinerte Zusicherungen erstellt werden können. Während der Aufzeichnung können die GUI-Elemente der Applikation inspiziert und sogenannte *AssertionHooks* eingefügt werden. Abbildung 6.14 zeigt die Inspektion einer Tabelle. Es ist zu erkennen, dass nur Informationen über ihre Sichtbarkeit und/oder Aktivierung zugesichert werden können. In der Benutzerdokumentation ist allerdings ausführlich erklärt, wie die Erstellung von spezifischeren Zusicherungen im Code erfolgen kann.

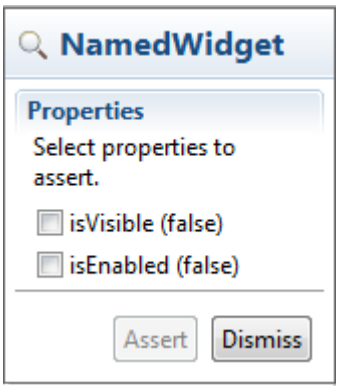


Abbildung 6.14: Inspektion einer Tabelle im Aufzeichnungsmodus

Support für datengetriebenes Testen

Da die von WindowTester Pro generierten Testklassen auf Version 3 des JUnit-Frameworks basieren, kann die bei FEST vorgestellte Vorgehensweise zur Erstellung parametrisierter Tests nicht zum Einsatz kommen. Grundsätzlich wird datengetriebenes Testen somit nicht unterstützt.

Allerdings dokumentiert Dufour [14] einen möglichen Lösungsansatz für dieses Problem. Durch Ableitung der bestehenden JUnit 3 `TestSuite`-Klasse und durch anschließende Implementierung zusätzlicher Methoden kann eine Basis für die Erstellung parametrisierter Tests geschaffen werden. Die entsprechenden Methoden müssen dabei nicht von Grund auf neu geschrieben werden, sondern können teilweise aus der Implementierung von JUnit 4 entnommen werden.

Listing 6.19 zeigt die Struktur einer mit WindowTester Pro erstellten Testklasse, welche die zuvor angesprochene `TestSuite`-Erweiterung verwendet und somit parametrisiert werden kann. Das Herzstück dieses Ansatzes liegt in der statischen `suite`-Methode (Zeile 12 bis 14). Diese signalisiert in JUnit 3, dass die Testmethoden einer Klasse als gemeinsame Test-Suite ausgeführt werden sollen. An dieser Stelle wird die erweiterte `TestSuite`-Klasse eingebunden und somit ein parametrisierter Durchlauf initiiert.

```
1  public class AddEntryTest extends BaseTest {
2    private String clubName;
3    private String clubShortName;
4
5    //identisch zum parametrisierten Konstruktor bei JUnit 4
6    public AddEntryTest(String clubName, String clubShortName) {...}
7
8    //entspricht der bei JUnit 4 mit @Parameters annotierten Methode
9    public static Collection<Object[]> parameters() {...}
10
11   //Instanzierung der erweiterten TestSuite-Klasse
12   public static Test suite(){
13     return new ExtendedTestSuite(AddEntryTest.class, parameters());
14   }
15
16   public void testAddEntry(){...}
17 }
```

Quellcode 6.19: Struktur der parametrisierten JUnit 3 Testklasse

Positive Aspekte

Die folgenden Aspekte von WindowTester Pro finden positive Erwähnung:

- WindowTester Pro vereint Capture/Replay mit einer skriptbasierten Vorgehensweise und ermöglicht somit, die Vorteile beider Ansätze zu nutzen. Während die Recording Console das rasche Aufzeichnen neuer Testskripts zulässt und eine grobe Struktur für die Testklassen vorgibt, erlaubt die händische Nachbearbeitung des Codes dessen Verfeinerung, Modularisierung und Abstrahierung. Da die Exportfunktion standardkonforme JUnit-Tests generiert, ist für einen erfahrenen Java-Tester kein zusätzlicher Einarbeitungsaufwand erforderlich. Dies ist ein klarer Vorteil gegenüber anderen Capture/Replay-Werkzeugen, die ihre eigene Skriptsprache entwerfen oder schlecht lesbaren Code produzieren.
- Sowohl die Funktionen für Aufzeichnung und Export als auch die javabasierte API von WindowTester Pro ist gut dokumentiert. Ein zusätzlicher Pluspunkt ist, dass die gesamte Benutzerdokumentation über die Eclipse Hilfe verfügbar ist.

Problemstellen und Lösungsansätze

In nachfolgender Aufzählung werden die Schwachstellen von WindowTester Pro diskutiert:

- **Umgang mit Fenstern:** Das Warten auf das Öffnen bzw. Schließen von Fenstern wird in den Testskripts durch Instanzen von `WindowShowingCondition` bzw. `WindowDisposedCondition` abgebildet. Häufig wurden diese jedoch in der falschen Reihenfolge bzw. mehrfach generiert oder waren überhaupt überflüssig, wodurch es zu Problemen bei der Wiedergabe kam. Bei jedem der aufgezeichneten Testskipts war daher die manuelle Nachbearbeitung dieser Conditions erforderlich. Die Umsetzung von Testfall *#4* scheiterte an einer `WindowShowingCondition`, die auf das Erscheinen eines Dialogs bis zum Timeout wartete, obwohl dieser bereits am Bildschirm sichtbar war.
- **Zeichensatz:** Standardmäßig kann WindowTester Pro bei der Aufzeichnung von Texteingaben nur Zeichen des US-Zeichensatzes verarbeiten. Werden andere Zeichen getippt, stürzt die Recording Console mit einer Exception ab. Allerdings kann durch Einfügen einer entsprechenden Anweisung (siehe Listing 6.20) in der `setup`-Methode der Testskripts ein alternativer

Zeichensatz festgelegt werden, sodass zumindest die Wiedergabe von Sonderzeichen möglich wird:

```
1 //Setzen eines alternativen Zeichensatzes
2 WT.setLocaleToCurrent();
3
4 //Zurücksetzen auf englischen Zeichensatz
5 WT.resetLocale();
```

Quellcode 6.20: Festlegen eines alternativen Zeichensatzes

- **SwingX-Unterstützung:** WindowTester Pro zeigte Probleme beim Umgang mit Komponenten der SwingX-Erweiterung. Der generierte Code war abhängig vom Typ der Komponente entweder gar nicht kompilierfähig oder scheiterte bei der Ausführung an einer `ComponentNotFoundException`. Die Ursache für dieses Problem liegt in den Konstruktoren der `Locator`-Klassen und wird am Beispiel des `JTableItemLocators` (siehe Listing 6.21) näher erläutert.

```
1  public class JTableItemLocator extends SwingWidgetLocator {
2
3    public JTableItemLocator(Point p) {
4      this(p, null);
5    }
6
7    public JTableItemLocator(Point p, SwingWidgetLocator parent) {
8      this(p, UNASSIGNED, parent);
9    }
10
11   public JTableItemLocator(Point p, int index,
12       SwingWidgetLocator parent) {
13     this(JTable.class, p, index, parent);
14   }
15   ...
16 }
```

Quellcode 6.21: Kaskadierende Konstruktoren der `JTableItemLocator`-Klasse

Wie im Listing zu sehen ist, werden beim Aufruf des „kleinsten“ Konstruktors (Zeile 3) automatisch alle nachfolgenden Konstruktoren mit Defaultwerten befüllt durchlaufen. Der dritte Konstrutor (Zeile 11) spezifiziert, dass die zu suchende Tabelle eine Instanz der `JTable`-Klasse sein muss. In einer eigens bereitgestellten `ExactClassMatcher`-Klasse wird das Übereinstimmen der Klassen während der Skriptausführung geprüft und bei Abweichung die angesprochene `ComponentNotFoundException` geworfen.

Damit auch Tabellen vom Typ `JXTable` verarbeitet werden konnten, war die in Listing 6.22 gezeigte Erweiterung notwendig:

```
1 public class JXTableItemLocator extends JTableItemLocator {
2
3   public JXTableItemLocator(Point p){
4     super(JXTable.class, p, UNASSIGNED, null);
5   }
6 }
```

Quellcode 6.22: Erweiterung der `JTableItemLocator`-Klasse

Diese Erweiterung konnte jedoch nicht in die Recording Console eingebunden werden, wodurch für jede `JXTable` eine manuelle Anpassung der aufgezeichneten Interaktion stattfinden musste. Eine analoge Vorgehensweise musste auch für andere Komponenten der SwingX-Erweiterung angewandt werden.

- **Eingeschränkte Adaptierbarkeit:** WindowTester Pro verwendet als Trennzeichen (z.B. bei Pfadangaben in einem Menü) den „/“. Dem Tester wird keine Möglichkeit zur Änderung dieser Einstellung geboten. Dies ist insofern problematisch, als dieses Zeichen in vielen Beschriftungen der Applikation enthalten ist und WindowTester bei der Wiedergabe an diesen Stellen irrtümlich eine Pfadangabe annimmt. Gelöst werden konnte das Problem durch Ersetzen sämtlicher „/“, die Teil einer Beschriftung waren, durch reguläre Ausdrücke.
- **Abstraktion in Tabellen:** Wie in Zeile 3 von Listing 6.21 zu sehen war, werden Interaktionen mit Tabelleneinträgen bei WindowTester Pro durch Instanzen von `java.awt.Point` beschrieben. Dabei entspricht die X-Koordinate der Zeilen- und die Y-Koordinate der Spaltennummer der zu selektierenden Zelle. Dieser Ansatz ist nicht sonderlich robust, da er nicht unabhängig von Änderungen der Tabelleneinträge und/oder deren Sortierung ist.

Wartbarkeit

Die Wartbarkeit der mit WindowTester Pro erstellten Testskripts ist abhängig davon, wie viel Aufwand in die Nachbearbeitung der exportierten JUnit-Testklassen investiert wurde. Die Modularisierung, Abstrahierung und Dokumentation des Codes im Zuge der initialen Erstellung eines Testskripts trägt unmittelbar zu besser wart- und lesbaren Testklassen bei. Ein weiterer positiver Aspekt von WindowTester Pro ist, dass zukünftige Anpassungen eines Testskripts schnell und ohne neuerliche Aufzeichnung des gesamten Interaktionsablaufs erfolgen können.

Dass die Identifikation der Komponenten ausschließlich über ihre Beschriftung in der Benutzeroberfläche erfolgt, führt unter Umständen dazu, dass eine häufigere Wartung der Testskripts erforderlich wird.

7 Bewertung und Vergleich

Dieses Kapitel diskutiert die im Zuge der Machbarkeitsstudie gewonnenen Erkenntnisse und führt eine abschließende Beurteilung der analysierten Werkzeuge durch. Nach tabellarischer Darstellung ausgewählter Kriterien erfolgt im Anschluss der separate Vergleich der Capture/Replay-Tools sowie der skriptbasierten Automatisierungsansätze. Neben Beurteilung der Eignung der Testwerkzeuge für das bestehende Softwareprojekt werden insbesondere auch die Übertragbarkeit und der allgemeine Nutzen der gewonnenen Erkenntnisse für ähnliche Projekte angesprochen.

7.1 Auswertung der Erkenntnisse

In nachfolgender Tabelle 7.1 werden einige allgemeine Aspekte für jedes Werkzeug beurteilt. Vermittelt werden soll ein Überblick über hauptsächlich nicht-funktionale Kritieren, welche auch für die Auswahl eines geeigneten Testwerkzeugs in ähnlichen Projekten als Entscheidungsgrundlage herangezogen werden können. Erneut wurde das bereits in Tabelle 5.1 definierte Bewertungsschema angewandt.

	FEST	Marathon	Abbot (API)	Sikuli	Robot Framework	Window-Tester Pro
Schnelle Installation	✓	✓	✓	~	~	✓
Schnelle Einarbeitung	~	✓	~	✓	✓	✓
Programmierkenntnisse erforderlich	✓	✓	✓	✓	×	✓
JUnit-Tests möglich	✓	×	✓	✓	×	✓
Gute Lesbarkeit des Codes	✓	~	✓	✓	✓	✓
Geringer Wartungsaufwand	✓	×	✓	~	✓	✓
Internes Wissen über App. erforderlich	✓	×	✓	×	✓	~
Robustheit ggb. Änderungen der App.	✓	×	✓	×	✓	~
Fehlerfreie Funktionsweise[1]	✓	~	~	~	✓	~

Tabelle 7.1: Beurteilung ausgewählter Aspekte der Werkzeuge

[1] Die Beurteilung bezieht sich auf den Umgang mit Standard-Swing-Komponenten

Die Beurteilung erfolgte dabei unter Berücksichtigung des gesamten Funktionsumfangs des Werkzeugs, auch wenn dieser für die Umsetzung der Testfälle aufgrund der Besonderheiten der bestehenden Applikation nicht immer optimal genutzt werden konnte. Beispielsweise erhält FEST für die Robustheit eine positive Bewertung, da die Identifikation der GUI-Elemente über den im Programmcode vergebenen eindeutigen Namen grundsätzlich möglich ist.

7.2 Beurteilung der Werkzeuge

Wie bereits in Abschnitt 3.7.1 erwähnt, eignen sich Capture/Replay-Werkzeuge besonders für die Erstellung von Regressionstests zu einem Zeitpunkt, wo die Hauptentwicklungsphase der zu testenden Software bereits abgeschlossen ist, während ein skriptbasierter Ansatz bereits vor oder während der Implementierung zum Einsatz kommen kann. Aufgrund dieser Tatsache scheint es sinnvoll, die eingesetzten Werkzeuge nach Typ zu differenzieren und getrennt zu beurteilen.

7.2.1 Vergleich der auf Capture/Replay basierenden Werkzeuge

In diesem Abschnitt werden die beiden Werkzeuge Marathon und WindowTester Pro miteinander verglichen. Allgemein kann gesagt werden, dass beide Werkzeuge in der Praxis eher zum Testen von jenen Rich-Client-Applikationen geeignet sind, welche einen geringen Umfang aufweisen, keine Erweiterungsbibliotheken wie zum Beispiel SwingX verwenden und bei welchen sich die Interaktionen auf ein einzelnes Hauptfenster beschränken. Die nachfolgende Tabelle 7.2 stellt jene Vorteile sowie Schwachstellen der beiden Tools gegenüber, die nicht nur für das Fallbeispiel, sondern auch für ähnlich aufgebaute Softwareprojekte Relevanz besitzen:

	Marathon	WindowTester Pro
Vorteile	(+) Erstellung von Zusicherungen während Aufzeichnung (+) Datengetrieben nativ unterstützt (+) Integriertes Reporting (+) Verschiedene Ausführungskonfigurationen durch Fixtures (+) Einbindung vorhandener Module während Aufzeichnung	(+) Skriptbasierte Nachbearbeitung (+) Integration in ein bestehendes Eclipse-Projekt (+) Erweiterung bestehender Klassen durch Vererbung
Nachteile	(-) Nur Standard-Swing-Komponenten (-) Object Maps schlecht für Wart- und Lesbarkeit (-) Erweiterung aufwendig	(-) Nur Standard-Swing-Komponenten (-) Zusicherungen müssen händisch erstellt werden (-) Datengetrieben nicht nativ unterstützt (-) Veraltete JUnit-Version

Tabelle 7.2: Vor- und Nachteile der betrachteten Capture/Replay-Werkzeuge

Relevanz für das gewählte Fallbeispiel

Für das automatisierte Testen der bestehenden Applikation ist aufgrund der fehlenden SwingX-Unterstützung sowie der übrigen erwähnten Schwachstellen keines der beiden Werkzeuge optimal

geeignet. In beiden Fällen wäre die Entwicklung umfangreicher Erweiterungen zur Unterstützung der zusätzlichen Komponenten erforderlich.

Relevanz und Übertragbarkeit der Erkenntnisse für andere Softwareprojekte

Aus dem Inhalt der Tabellen 7.1 und 7.2 lässt sich schließen, dass auch für ähnliche Softwareprojekte die Eignung der Werkzeuge zur Unterstützung des Testens nicht vollständig gegeben wäre. Geringe Wart- und Lesbarkeit von Testskripts, die Verwendung veralteter Versionen von Testframeworks sowie ein Mangel an Robustheit bergen für beliebige Softwareprojekte das hohe Risiko, dass die Einführung von Testautomatisierung nicht den gewünschten langfristigen Nutzen bringt und die aufgewandten Kosten sich nicht rentieren. Sowohl bei Marathon als auch bei WindowTester Pro ist davon auszugehen, dass aufgrund der genannten Einschränkungen nicht nur bei dem untersuchten Fallbeispiel, sondern auch bei anderen, ähnlich komplexen Projekten Aufwand in die Anpassung, Verbesserung und Erweiterung investiert werden müsste. Ist datengetriebenes Testen ein unverzichtbares Kriterium für ein Projekt, scheint die Auswahl von Marathon an dieser Stelle empfehlenswerter.

Zusammenfassend lässt sich also beurteilen, dass bei keinem der betrachteten Capture/Replay-Werkzeuge von einer umfassenden Praxistauglichkeit gesprochen werden kann. Diese Erkenntnis deckt sich mit den Ergebnissen der Untersuchung bestehender Capture/Replay-Werkzeuge von Jovic u. a. [32]. Die Autoren haben fünf bestehende Werkzeuge – unter ihnen auch Marathon und Abbot Costello – analysiert und weisen im Rahmen einer abschließenden Diskussion darauf hin, dass ihre Praxistauglichkeit unter anderem stark durch die folgenden beiden Aspekte beeinträchtigt wird:

- Fehlende Funktionalität: Die Werkzeuge sind nicht in der Lage, den Funktionsumfang von realen Softwareprojekten sowie die Varietät an allgemein verfügbaren GUI-Komponenten abzudecken.
- Probleme bei der robusten Identifikation von GUI-Komponenten: Liegen keine eindeutigen Identifikationsmerkmale über die Komponenten der Benutzeroberfläche vor, greifen die Werkzeuge auf von Layout und Position abhängige Eigenschaften zurück, wodurch die Robustheit der Testskripts und somit ihr langfristiger Einsatz als Regressionstests nicht gegeben ist.

7.2.2 Vergleich der Werkzeuge mit skriptbasiertem Ansatz

In diesem Abschnitt werden die verbleibenden vier Werkzeuge FEST, Abbot, Sikuli sowie Robot Framework diskutiert. Allgemein lässt sich als Erkenntnis festhalten, dass skriptbasierte Ansätze (mit Ausnahme von Sikuli) grundsätzlich mehr Wissen über die interne Funktionsweise der zu testenden Applikation und der verwendeten GUI-Komponenten erfordern.

Zunächst erfolgt eine Gegenüberstellung der Vor- und Nachteile von FEST und Abbot. Die getroffenen Aussagen besitzen wiederum nicht nur für das gewählte Fallbeispiel, sondern auch in Bezug auf andere reale Softwareprojekte Gültigkeit. Es ist erkennbar, dass die beiden Bibliotheken einen sehr ähnlichen Ansatz verfolgen und sich auch in ihrer Struktur nicht besonders unterscheiden. Dies ist nicht weiter verwunderlich, da - wie bereits in Abschnitt 5.3.1 erläutert - das Vorläuferprojekt von FEST eine um TestNG erweiterte Variante von Abbot war. Wie aus Tabelle 7.3 klar hervorgeht, ist FEST gegenüber Abbot besser zur Implementierung von datengetriebenen, skriptbasierten GUI-Tests geeignet. Wird der Einsatz eines skriptbasierten Werkzeugs angestrebt, ist allerdings

zu empfehlen, bereits bei der Programmierung eines Softwaresystems auf die Vergabe eindeutiger Bezeichner für die GUI-Komponenten zu achten und diese beim Erstellen der Tests zu verwenden.

	FEST	Abbot
Vorteile	(+) Partielle SwingX-Unterstützung (+) Fluent Interfaces (+) Datengetrieben durch JUnit 4 (+) Standard-Matcher bereitgestellt	(+) Partielle SwingX-Unterstützung (+) Support für Internationalisierung
Nachteile	(-) Identifikation per Index nicht unterstützt (-) Erweiterung der Bibliothek über Vererbung eingeschränkt	(-) Identifikation per Index nicht unterstützt (-) Veraltete JUnit-Version (-) Datengetrieben nicht unterstützt (-) Kein Benutzerhandbuch (-) Standard-Matcher müssen selbst implementiert werden

Tabelle 7.3: Gegenüberstellung der Vor- und Nachteile von FEST und Abbot

Der praktische Einsatz von Sikuli hat gezeigt, dass dieses Werkzeug noch nicht ausgereift ist und in vielen Punkten Verbesserungspotential besteht. Die Erstellung der Screenshots gestaltet sich zeitaufwendig und die große Abhängigkeit der Testskripts zu diesen schränkt die Flexibilität hinsichtlich austauschbarer Testdaten ein. Zurzeit wird von dem Projektteam an einer OCR-Unterstützung gearbeitet, sodass Elemente am Bildschirm über ihren sichtbaren Text anstelle von Screenshots identifiziert werden können. Bis dieses Feature in stabiler Form verfügbar ist, ist vom Einsatz von Sikuli in umfangreichen Projekten abzuraten.

Ähnlich wie Sikuli verfolgt auch das Robot Framework einen Ansatz, der nicht mit den anderen betrachteten Werkzeugen direkt verglichen werden kann. Besonders stark sticht das Framework durch die Tatsache, dass ein Tester für die Umsetzung von Testfällen keinerlei Programmierkenntnisse besitzen muss, heraus. Überraschend ist weiters, dass sich über die bereitgestellten Schlüsselwörter nahezu beliebige Interaktionen auch mit einer komplexen Benutzeroberfläche simulieren lassen. Nennenswert ist ebenfalls, dass das Robot Framework das einzige skriptbasierte Werkzeug ist, bei dem automatisch sehr umfangreiche Reports und Log-Dateien erstellt werden, während bei allen anderen zusätzlicher Aufwand für die Report-Erstellung über einen Ant-Build eingeplant werden muss.

Relevanz für das gewählte Fallbeispiel

Als Lösungsansatz für das automatisierte Testen der bestehenden Applikation kommen grundsätzlich die FEST Bibliothek bzw. das Robot Framework in Frage. Beide Tools müssten jedoch vor dem Einsatz in der Praxis entsprechend angepasst werden. Beim Robot Framework müssten die Swing Library und Jemmy erweitert werden, sodass die Erkennung von `JOptionPanes` funktioniert. Bei FEST würden mehrere, allerdings weniger aufwendige Adaptionen der Bibliothek zur optimalen Interaktion mit den GUI-Komponenten der SwingX-Erweiterung benötigt werden. Hinzu kommt in beiden Fällen, dass der angesprochene Test-Modus zur Unterbindung der Dialoge beim Startvorgang dauerhaft in die Applikation integriert werden müsste.

Relevanz und Übertragbarkeit der Erkenntnisse für andere Softwareprojekte

Die gewonnenen Erkenntnisse in den Tabellen 7.1 sowie 7.3 lassen vermuten, dass sowohl FEST als auch das Robot Framework einen Nutzen für das Testen ähnlich umfangreicher und komplexer Softwareprojekte darstellen könnten. Dennoch kann diese Vermutung nicht als allgemein gültig betrachtet werden, da der Mehrwert einer Automatisierung von Testtätigkeiten immer im Kontext eines realen konkreten Projekts und seines Testprozesses beurteilt werden muss. Es kann lediglich gesagt werden, dass beide Bibliotheken aufgrund der hohen Robustheit, guten Wart- und Lesbarkeit ihrer Testskripts sowie der geringen Menge an aufgefundenen kritischen Schwachstellen das Potenzial haben, zu einer Steigerung der Effizienz beizutragen.

Der Grad an notwendiger Adaption wäre ebenfalls abhängig von den Charakteristiken des zu testenden Projekts. Im Idealfall, wenn sowohl auf die Vergabe von eindeutigen Bezeichnern für die GUI-Elemente geachtet als auch auf die Verwendung von `JOptionPanes` verzichtet wurde und ausschließlich Komponenten aus dem Standard-Swing-Toolkit verwendet werden, wäre ein Einsatz auch ohne Adaption der gewählten Testautomatisierungsbibliotheken denkbar.

Eine Entscheidung, welche der beiden Testbibliotheken die bessere Wahl darstellt, muss ebenfalls situationsbezogen getroffen werden, da projektspezifische Aspekte wie zum Beispiel der Grad an Programmierkenntnissen im Testteam oder die Notwendigkeit der Integration der Testskripts in Form von Unit-Tests auf jeden Fall mitberücksichtigt werden müssten. Fällt die Wahl auf die FEST-Bibliothek, sollte der vorgestellte oder ein alternativer Lösungsansatz zur Erhöhung der Abstraktion im Umgang mit Tabellen angestrebt werden, da Tabellen eine Darstellungsform sind, die in vielen Applikationen zum Einsatz kommt.

8 Zusammenfassung und Ausblick

Die (Teil-)Automatisierung wiederkehrender und oftmals zu wiederholender manueller Tätigkeiten des Software-Testens dient der langfristigen Steigerung der Effizienz und Qualität unter gleichzeitiger Reduktion der Kosten. Der Einsatz eines Testwerkzeugs rentiert sich jedoch nur dann, wenn seine Praxistauglichkeit gegeben ist und die Stabilität und Robustheit der damit erstellten Testskripts gewährleistet werden kann. Die Auswahl des geeigneten Werkzeugs ist somit entscheidend für den Erfolg eines Automatisierungsvorhabens. Vor allem dann, wenn es eine große Anzahl an am Markt verfügbaren Werkzeugen gibt oder ein Tester noch wenig Erfahrung mit der Automatisierung eines bestimmten Teilbereichs des Software-Testens aufweist, werden für eine zielführende Entscheidungsfindung Informationen über die Funktionalität und Qualität existierender Produkte benötigt.

Diese Arbeit hat die Praxistauglichkeit bestehender Open-Source-Werkzeuge zum automatisierten Testen der Benutzeroberflächen von javabasierten Rich-Client-Applikationen anhand eines Fallbeispiels untersucht. Dabei handelte es sich um ein verteiltes Event-Management-System, das aufgrund seiner Größe, Komplexität sowie der für die Realisierung verwendeten Technologien als repräsentatives Projekt erachtet wurde. Die Relevanz dieser Untersuchung ergab sich aus der Tatsache, dass in diesem Bereich bisher verhältnismäßig wenig Wissen über verfügbare Lösungen vorliegt. Durch Erarbeitung eines Lösungsansatzes zur Testautomatisierung der bestehenden Applikation sollte aufgrund ihres repräsentativen Charakters die Beurteilung der Eignung der Werkzeuge für ähnlich komplexe Systeme diskutiert werden.

Nach Abhandlung der theoretischen Grundlagen von grafischen Benutzeroberflächen und der Disziplin des Software-Testens wurden in einem ersten Schritt das Umfeld der bestehenden Applikation analysiert und projektspezifische sowie allgemein relevante Auswahlkriterien entworfen. Für das Projekt besonders wichtig waren neben der Open-Source-Lizensierung die Unterstützung eines datengetriebenen Ansatzes, die Integrationstauglichkeit in den bestehenden Testprozess sowie die Verwendung einer standardisierten Skriptsprache. Nach Auswertung des definierten Kriterienkatalogs konnte eine Reihung der 23 zuvor identifizierten, potenziell geeigneten Werkzeuge vorgenommen werden.

In einem nächsten Schritt wurden zwei Werkzeuge mit Capture/Replay-Ansatz, nämlich Marathon und WindowTester Pro, die drei auf dem Testframework JUnit basierenden Bibliotheken FEST, Abbot und Sikuli sowie das schlüsselwortgetriebene Robot Framework in die engere Auswahl aufgenommen. Nach der theoretischen Auseinandersetzung mit dem Aufbau, dem Konzept und der Funktionsweise jedes Werkzeugs wurden anschließend jeweils fünf repräsentative Testfälle implementiert. Diese wurden so gewählt, dass sie Interaktionen mit verschiedensten GUI-Komponenten des Standard-Swing- bzw. SwingX-Toolkits enthielten.

Es hat sich gezeigt, dass keines der Werkzeuge frei von Problemen und Schwachstellen ist. Besonders häufige Probleme waren ein zu geringer Abstraktionsgrad beim Umgang mit Tabellen oder Listen, die fehlende oder limitierte Unterstützung des SwingX-Toolkits, eine zu geringe Robustheit der Testskripts, Mängel beim Verarbeiten von Sonderzeichen sowie das häufige Öffnen und Schließen von Fenstern. Besonders schlecht schnitten die Werkzeuge Abbot und Sikuli ab, da mit ihnen zusätzlich keine datengetriebene Vorgehensweise umgesetzt werden konnte und im Falle von Sikuli außerdem starke Abhängigkeiten zwischen den erstellten Screenshots und dem Skriptcode

bestanden. Am ehesten für den praktischen Einsatz qualifizierten sich die FEST-Bibliothek sowie das Robot Framework. Doch auch diese beiden Werkzeuge konnten nicht vollständig überzeugen. Bei FEST mangelte es an der einfachen Erweiterbarkeit der bestehenden Implementierung, beim Robot Framework bestand ein gravierendes, im Rahmen dieser Arbeit jedoch nicht bis ins Detail analysiertes Problem mit der Verarbeitung von `JOptionPanes`, für das erst eine Lösung entworfen und umgesetzt werden müsste.

Die Fragestellung dieser Arbeit – nämlich ob sich ein existierendes Open-Source-Werkzeug zum automatisierten Testen der bestehenden Applikation eignet – muss an dieser Stelle also mit einem *Nein* beantwortet werden. Unabhängig vom gewählten Ansatz (Capture/Replay, skriptbasiert, schlüsselwortgetrieben) wäre die umfangreiche Erweiterung oder Adaption der Werkzeuge erforderlich, damit ihre Praxistauglichkeit für das reale, komplexe Projekt gegeben wäre und ihr Einsatz einen Mehrwert für dieses darstellen würde.

Die mangelnde Wart- und Lesbarkeit, Einschränkungen in der Funktionalität sowie die fehlende Robustheit der Testskripts disqualifizieren die betrachteten Capture/Replay-Werkzeuge auch für den Einsatz in anderen Rich-Client-Softwareprojekten. Die Nichterfüllung dieser Aspekte würde das hohe Risiko des Scheiterns des Automatisierungsvorhabens bergen oder zumindest keinen langfristigen Nutzen für ein Projekt darstellen. FEST sowie das Robot Framework hingegen zeigten großes Potenzial und könnten zur Effizienzsteigerung der Testaktivitäten in ähnlichen Projekten beitragen. Es ist dennoch zu vermuten, dass Aufwand in die Umsetzung der projektspezifisch benötigten Erweiterungen zur Bewältigung der identifizierten Schwachstellen (fehlender Abstraktionsgrad, nur partielle Unterstützung eines Erweiterungstoolkits, Verarbeitung von `JOptionPanes`) investiert werden muss.

Abschließend gilt es zu erwähnen, dass alle betrachteten Werkzeuge nach wie vor aktiv weiterentwickelt werden. Für zukünftige Untersuchungen ist daher zu empfehlen, neue Versionen der Werkzeuge zu betrachten und hinsichtlich Verbesserungen der in dieser Arbeit genannten Problemstellen zu prüfen. Für die Anfang 2015 angekündigte neue Version von Sikuli ist beispielsweise die Unterstützung regulärer Ausdrücke sowie die Texterkennung mittels OCR geplant. Dieser Ansatz klingt vielversprechend und sollte ab Verfügbarkeit einer stabilen Version auf jeden Fall näher betrachtet werden.

Weiters wurde die Integrationstauglichkeit der Werkzeuge nur theoretisch abgehandelt. Es scheint daher sinnvoll, vor Auswahl und Erweiterung eines konkreten Werkzeugs im Rahmen eines Pilotprojekts zu untersuchen, ob, und wenn ja, wie gut die Einbindung der erstellten Testskripts sowie der durch das Werkzeug generierten Ergebnisse in die bestehende Infrastruktur funktioniert.

Literatur

[1] M. Aberdour. *opensourcetesting.org: open source software testing tools, news and discussion*. URL: http://www.opensourcetesting.org (besucht am 08. 11. 2013).

[2] *Add support for parent-less JDialog boxes to Jemmy V2*. Okt. 2013. URL: https://java.net/jira/browse/JEMMY-31 (besucht am 11. 03. 2014).

[3] P. Baker u. a. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag Berlin Heidelberg, 2008.

[4] S. Berner, R. Weber und R. K. Keller. "Observations and Lessons Learned from Automated Testing". In: *ICSE '05 Proceedings of the 27th international conference on Software engineering* (2005), S. 571–579.

[5] C. Beust und H. Suleiman. *Next generation Java testing: TestNG and advanced concepts*. Addison-Wesley Professional, 2007.

[6] Bibliographisches Institut GmbH, Hrsg. *Duden*. URL: http://www.duden.de/node/847825/revisions/1156262/view (besucht am 31. 10. 2013).

[7] B. W. Boehm. *Verifying and Validating Software Requirements and Design Specifications*. URL: http://csse.usc.edu/csse/TECHRPTS/1979/usccse79-501/usccse79-501.pdf (besucht am 07. 10. 2013).

[8] T.-H. Chang, T. Yeh und R. C. Miller. "GUI Testing Using Computer Vision". In: *CHI '10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2010), S. 1535–1544.

[9] P. Chlebek. *User Interface-orientierte Softwarearchitektur*. 1. Aufl. Friedr. Vieweg & Sohn Verlag, 2006.

[10] T. Cleff. *Basiswissen Testen von Software*. W3L GmbH, 2010.

[11] B. Cole u. a. *Java Swing*. 2. Aufl. O'Reilly, 2002.

[12] Heise Developer, Hrsg. *Google stellt frühere Instantiations-Werkzeuge kostenlos zur Verfügung*. Sep. 2010. URL: http://www.heise.de/developer/meldung/Google-stellt-fruehere-Instantiations-Werkzeuge-kostenlos-zur-Verfuegung-1080239.html (besucht am 19. 12. 2013).

[13] Google Developers, Hrsg. *WindowTester Pro User Guide*. März 2012. URL: https://developers.google.com/java-dev-tools/wintester/html/ (besucht am 18. 12. 2013).

[14] T. Dufour. *Parameterized test case in JUnit 3.x*. Feb. 2009. URL: http://stackoverflow.com/a/512707/3278695 (besucht am 06. 03. 2014).

[15] FEST, Hrsg. *fest: Fixtures for Easy Software Testing*. URL: https://code.google.com/p/fest/ (besucht am 14. 12. 2013).

[16] A. Fowler. *A Swing Architecture Overview: The Inside Story on JFC Component Design*. URL: http://www.oracle.com/technetwork/java/architecture-142923.html (besucht am 03. 11. 2013).

[17] M. Fowler. *Continuous Integration*. Mai 2006. URL: http://www.martinfowler.com/articles/continuousIntegration.html (besucht am 15. 10. 2013).

[18] Robot Framework, Hrsg. *Robot Framework: Generic test automation framwork for acceptance testing and ATDD*. URL: http://robotframework.org (besucht am 16. 12. 2013).

[19] Robot Framework, Hrsg. *robotframework: A generic test automation framework*. URL: https://code.google.com/p/robotframework/ (besucht am 17. 12. 2013).

[20] K. Franz. *Handbuch zum Testen von Web-Applikationen: Testverfahren, Werkzeuge, Praxistipps*. Springer-Verlag Berlin Heidelberg, 2007.

[21] H. Freeman. "Software Testing". In: *IEEE Instrumentation & Measurement Magazine* 5.3 (2002), S. 48–50.

[22] D. M. Geary. *Graphic Java 2.0: Die JFC beherrschen (Swing)*. 3. Aufl. Bd. 2. Markt+Technik, 2000.

[23] D. Graham u. a. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning EMEA, 2008.

[24] T. Grechenig u. a. *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2010.

[25] *GUI Testing*. URL: http://www.appperfect.com/products/application-testing/app-test-gui-testing.html (besucht am 16. 10. 2013).

[26] R. Hocke. *Sikuli 1.2.0 Early2015*. URL: https://launchpad.net/sikuli/+milestone/1.2.0 (besucht am 04. 03. 2014).

[27] "IEEE Standard Glossary of Software Engineering Terminology". In: *IEEE Std 610.12-1990* (1990), S. 1–84.

[28] Institut für Visualisierung und Interaktive Systeme, Universität Stuttgart. *Grundlagen der Interaktiven Systeme: GUI Grundlagen und Toolkits*. URL: http://cumbia.visus.uni-stuttgart.de/ger/research/proj/ito/materials/ (besucht am 29. 10. 2013).

[29] International Organization for Standarization, Hrsg. *ISO/IEC 9126-1: Software Engineering – Product Quality, Part 1: Quality Model*. 2001.

[30] Jalian Systems Pvt. Ltd., Hrsg. *Marathon and MarathonITE Manual*. URL: http://marathontesting.com/wp-content/downloads/marathonite-userguide.pdf (besucht am 11. 12. 2013).

[31] Jalian Systems Pvt. Ltd., Hrsg. *marathonITE: Powerful Tools for Creating Resilient Test Suites*. URL: http://marathontesting.com (besucht am 11. 12. 2013).

[32] M. Jovic u. a. "Automating Performance Testing of Interactive Java Applications". In: *AST '10 Proceedings of the 5th Workshop on Automation of Software Test* (2010), S. 8–15.

[33] D. Karra. *Marathon- GUI Acceptance Test Runner*. URL: http://sourceforge.net/projects/marathonman/ (besucht am 09. 12. 2013).

[34] S. Kersken. *IT-Handbuch für Fachinformatiker: Der Ausbildungsbegleiter*. 6. Aufl. Galileo Computing, 2013.

[35] M. Kleinsteuber und C. Hage. *Computer Vision*. URL: http://www.ldv.ei.tum.de/lehre/computer-vision/ (besucht am 08. 01. 2014).

[36] S. Kleuker. *Computer Software Investigation (CSI) Hochschule Ösnabrück*. Juli 2011. URL: http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/kombiQuWerkzeuge.html (besucht am 28. 11. 2013).

[37] A. Klingert. *Einführung in Graphische Fenstersysteme: Konzepte und reale Systeme*. Springer-Verlag Berlin Heidelberg, 1996.

[38] K. Li und M. Wu. *Effective GUI Test Automation: Developing an Automated GUI Testing Tool*. Sybex Inc., 2005.

[39] W. Liu. "Natural User Interface - Next Mainstream Product User Interface". In: *2010 IEEE 11th International Conference on Computer-Aided Industrial Design & Conceptual Design (CAIDCD)* 1 (2010), S. 203–205.

[40] Eclipse Marketplace, Hrsg. *WindowTester Pro GUI Tester 6.0*. URL: https://marketplace.eclipse.org/content/windowtester-pro-gui-tester (besucht am 19. 12. 2013).

[41] A. M. Memon. "GUI Testing: Pitfalls and Process". In: *Computer* 35 (2002), S. 87–88.

[42] S. Middendorf, R. Singer und J. Heid. *Java Programmierhandbuch und Referenz für die Java-2-Plattform: Einführung und Kernpakete*. 3. Aufl. dpunkt Verlag, 2002.

[43] A. Mockus, R. T. Fielding und J. D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), S. 309–346.

[44] M. Moroz. *How-To: Sikuli and Robot Framework Integration*. Feb. 2011. URL: http://blog.mykhailo.com/2011/02/how-to-sikuli-and-robot-framework.html (besucht am 11. 12. 2013).

[45] M. Moser. *Wann lohnt sich GUI-Testautomatisierung?* URL: http://www.qfs.de/de/info/HandoutsJFS07.pdf (besucht am 16. 10. 2013).

[46] G. J. Myers. *The Art of Software Testing: Second Edition*. 2. Aufl. John Wiley & Sons, Inc., 2004.

[47] S. Nedyalkova und J. Bernardino. "Open Source Capture and Replay Tools Comparison". In: *C3S2E '13 Proceedings of the International C* Conference on Computer Science and Software Engineering* (2013), S. 117–119.

[48] J. Nielsen. *Usability Engineering*. Academic Press, 1994.

[49] P. Niemeyer und J. Peck. *Exploring Java*. 2. Aufl. O'Reilly Media, 1997. Kap. 13. The Abstract Window Toolkit.

[50] Österreichisches Normungsinstitut, Hrsg. *ÖNORM EN ISO 9241-110: Ergonomie der Mensch-System-Interaktion, Teil 110: Grundsätze der Dialoggestaltung*. 2008.

[51] Österreichisches Normungsinstitut, Hrsg. *ÖNORM EN ISO 9241-16: Ergonomische Anforderungen für Bürotätigkeiten mit Bildschirmgeräten, Teil 16: Dialogführung mittels direkter Manipulation*. 2000.

[52] *Open Source Software in Java*. URL: http://java-source.net (besucht am 04. 12. 2013).

[53] T. Ostrand u. a. "A Visual Test Development Environment for GUI Systems". In: *ISSTA '98 Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis* 23 (1998), S. 82–92.

[54] J. Ottinger. *JUnit 4.0 has been released*. Feb. 2006. URL: http://www.theserverside.com/news/thread.tss?thread_id=39048 (besucht am 28. 02. 2014).

[55] K. Parthasarathy. *Announcing WindowTester open source release*. März 2012. URL: http://google-opensource.blogspot.co.at/2012/03/announcing-windowtester-open-source.html (besucht am 18. 12. 2013).

[56] P. Pepper. *Programmieren mit Java: Eine grundlegende Einführung für Informatiker und Ingenieure*. Springer-Verlag Berlin Heidelberg, 2005.

[57] M. Pezzé und M. Young. *Software testen und analysieren: Prozesse, Prinzipien und Techniken*. Oldenbourg Verlag München, 2009.

[58] J. Piironen. *Robot Framework Plugin*. Nov. 2013. URL: https://wiki.jenkins-ci.org/display/JENKINS/Robot+Framework+Plugin (besucht am 17. 12. 2013).

[59] I. Pinkster u. a. *Successful Test Management: An Integral Approach*. Springer, 2004.

[60] B. Preim. *Entwicklung interaktiver Systeme: Grundlagen, Fallbeispiele und innovative Anwendungsfelder*. Springer-Verlag Berlin Heidelberg, 1999.

[61] Quality First Software GmbH, Hrsg. *Checkliste: Anforderungen GUI-Testtool für Java und/oder Web*. URL: http://www.qfs.de/de/info/Checkliste_QF-Test.pdf (besucht am 12. 11. 2013).

[62] O. Reminnyi. *Functional GUI Testing Automation Patterns*. 2013. URL: http://www.infoq.com/articles/gui-automation-patterns (besucht am 17. 10. 2013).

[63] A. Ruiz. *FEST*. Apr. 2011. URL: http://docs.codehaus.org/display/FEST/Home (besucht am 13. 12. 2013).

[64] A. Ruiz. *testng-abbot*. URL: https://code.google.com/p/testng-abbot/ (besucht am 13. 12. 2013).

[65] A. Ruiz und J. Bay. *An Approach to Internal Domain-Specific Languages in Java*. Feb. 2008. URL: http://www.infoq.com/articles/internal-dsls-java (besucht am 14. 12. 2013).

[66] A. Ruiz und Y. W. Price. "GUI Testing Made Easy". In: *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic & Industrial Conference* (2008), S. 99–103.

[67] A. Ruiz und Y. W. Price. "Test-Driven GUI Development with TestNG and Abbot". In: *IEEE Software* 24 (2007), S. 51–57.

[68] O. Scheffler. "Einführung eines Tools zur Testautomatisierung: Zwei Praxisbeispiele aus der Qualitätssicherung". In: *blickpunkte: Das Magazin rund um IT-Themen* (Okt. 2009). URL: http://www.pentasys.ag/Unternehmen/Blickpunkte.

[69] R. Seidl, M. Baumgartner und T. Bucsics. *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken*. dpunkt Verlag, 2012.

[70] Sikuli, Hrsg. *Sikuli Script*. URL: http://www.sikuli.org (besucht am 16. 12. 2013).

[71] H. Sneed, M. Baumgartner und R. Seidl. *Der Systemtest: Von den Anforderungen zum Qualitätsnachweis*. 3. Aufl. Carl Hanser Verlag München, 2012.

[72] A. Spillner u. a. *Praxiswissen Softwaretest - Testmanagement*. dpunkt Verlag, 2011.

[73] M. Utting und B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2007.

[74] E. v. Veenendaal, Hrsg. *Standard Glossary of Terms used in Software Testing*. 2012. URL: http://www.istqb.org/downloads/finish/20/101.html (besucht am 25. 09. 2013).

[75] T. Wall. *Abbot framework for automated testing of Java GUI components and programs*. URL: http://abbot.sourceforge.net/doc/overview.shtml (besucht am 12. 12. 2013).

[76] Wikipedia. *List of GUI testing tools*. Dez. 2013. URL: http://en.wikipedia.org/wiki/List_of_GUI_testing_tools (besucht am 04. 12. 2013).

[77] T. Yeh, T.-H. Chang und R. C. Miller. "Sikuli: Using GUI screenshots for search and automation". In: *UIST '09 Proceedings of the 22nd annual ACM symposium on User interface software and technology* (2009), S. 183–192.

[78] W. Zuser, T. Grechenig und M. Köhle. *Software Engineering mit UML und dem Unified Process*. 2. Aufl. Pearson Studium, 2004.

A Anhang

A.1 Werkzeuge-Matrix

Die nachfolgenden Tabellen beinhalten die gesammelten Informationen zu den 23 identifizierten Testautomatisierungswerkzeugen. Tabelle A.1 enthält die aufgefundenen Daten zu den projektspezifischen Kriterien, während Tabelle A.2 jene zu den allgemeinen Kriterien abbildet. Die Informationen wurden dabei hauptsächlich von den Produktwebseiten, Benutzerhandbüchern sowie Code-Repositories der Werkzeuge entnommen.

	Lizenztyp	Swing-Unterstützung	Datengetrieben	Integrationstausglichkeit	Capture-Replay vs. skriptbasierte Erstellung	Standardisierte Sprache
Abbot	EPL	ja	Costello: unbekannt, API: ja	Costello: unbekannt, API: ja	C/R: Costello, skriptbasiert: JUnit-Erweiterung	XML, Java (JUnit)
FEST	Apache v2.0	ja	sowohl TestNG als auch JUnit unterstützen datengetriebene Tests	ja	skriptbasiert über JUnit oder TestNG	Java (JUnit, TestNG)
jfcUnit	GPL v2.0	ja	XML-Skripts: unbekannt, JUnit: ja	fraglich, Classpath muss angepasst werden	rudimentärer C/R-Modus, wird über spezielle Tags im XML-Testfall gestartet, alternativ manuelle Anfertigung von XML-Skripts, skriptbasiert auch über JUnit	XML, Java (JUnit)
Marathon	LGPL	ja	ja, manuelle Nachbearbeitung der Skripts nötig	ja, über Ant	nur C/R	Ruby, Python
SWTBot	EPL	nein, nur SWT	*nicht mehr näher betrachtet*			
UISpec4J	CPL v1.0	ja	ja, da JUnit	ja, da JUnit	nur skriptbasiert	Java (JUnit, TestNG)

Jacareto	GPL	ja	?	?	nur C/R	?
Jemmy	CDDL v1.0	ja	?	?	nur skriptbasiert	Java
Pounder	LGPL	ja	?	unklar, grundsätzlich können XML-Skripts über JUnit-Erweiterung ausgeführt werden	nur C/R	XML
WindowTester Pro	EPL v1.0	ja	eventuell durch Nachbearbeitung	automatisierte Ausführung über Ant-Build bzw. JUnit, erfordert zwingend UI am Build-Server	nur C/R	Java (JUnit)
Eclipse Jubula (Plugin-Variante)	EPL v1.0	ja	Testdaten als Excel-Datei	nicht klar, ob in Standalone- oder Plugin-Version	weder noch, Testfallerstellung durch Drag&Drop der GUI-Elemente, keine Programmierkenntnisse erforderlich	herstellerspezifisch
GUIDancer	kommerziell			*nicht mehr näher betrachtet*		
fressia project	Apache v2.0	nein		*nicht mehr näher betrachtet*		
Maveryx	GPL v2.0	ja	CSV, XML, TXT, HTML, relationale DB	?	nur skriptbasiert	Java (JUnit)

Twist + Frankenstein	kommerziell	*nicht mehr näher betrachtet*				
Sikuli	MIT	ja	?	ja	nur skriptbasiert, Screenshot-Technik, Verfahren aus Computer Vision, Screenshots in Skript eingebettet	Python oder Java (JUnit)
Robot Framework	Apache v2.0	ja	HTML, TSV, TXT	ja, es gibt eigenes Jenkins-Plugin, XML-basierte Output-Files	weder noch, ist schlüsselwortgetrieben, also Verwendung vordefinierter Keywords	herstellerspezifisch
Arbiter	?	nein, nur Web	*nicht mehr näher betrachtet*			
Avignon	GPL v2.0	nein, nur Web	*nicht mehr näher betrachtet*			
GTT	GPL v2.0	ja	?	?	nur C/R	?
White	?	nein	*nicht mehr näher betrachtet*			
GUITAR	LGPL v3.0	ja	nein	?	vollkommen automatisierte Erstellung und Ausführung, modellbasiert, C/R als Add-On verfügbar	herstellerspezifisch

T-Plan Robot	GPL v2.0	ja	ja, sofern Java Test Skripts verwendet werden	?	beides, die mittels C/R erstellten Skripts sind in proprietärer Sprache, Export zu Java möglich, skriptbasierte Erstellung in Java	beides (proprietär, Java)

Tabelle A.1: Gesammelte Informationen über die verschiedenen Werkzeuge zu den projektspezifischen Kriterien

	aktive Weiterentwicklung	Dokumentation	Plattformunabhängigkeit	Fehleranalyse	Reporting	Reifegrad
Abbot	ja	rudimentär vorhanden, triviale Einführungsbeispiele, Dokumentation is JavaDoc von API	ja (weil in Java implementiert)	Costello: unbekannt, skriptbasiert: wie JUnit	ja, wennVerwendung von Ant	seit 2003
FEST	ja	umfangreich	ja (weil in Java implementiert)	Debugging möglich, fertigt Screenshots an, wenn Testfall fehlschlägt, sonst wie JUnit/TestNG	ja, HTML-Reports mit eingebetteten Screenshots	seit 2007
jfcUnit	nein, letzte Version 2004	einige Beispiele, teilweise veraltet (JUnit 3), keine eigene Dokumentation, nur JavaDoc von API	ja	XML-Skripts: unbekannt, Java: wie klassisches JUnit	?	seit 2001

Marathon	ja letzte Version Nov. 2013	umfangreiche Dokumentaion und Tutorials, gemeinsame Dokumentation mit kommerziellen Variante MarathonITE, teilweise unklar, welche Features in welcher Version verfügbar sind	ja	tabellarisch, ähnlich wie JUnit wird auch bei Python-Skript auf fehlgeschlagene Zeile verwiesen, integrierter Debugger	ja, Report- Datei kann erstellt werden	seit 2002, >13.000 Downloads auf Source- forge
SWTBot	*nicht mehr näher betrachtet*					
UISpec4J	? letzte Version 2011, letzte Commits 2011	rudimentäre Dokumenttion, gute Example-Projekte und Tutorials	ja	wie JUnit, Debugging möglich	ja, wenn Verwendung von Ant	seit 2005, Hauptent- wicklungs- phase 2009-2011
Jacareto	? letzte Version (0.8.1) 2010	rudimentär, zu vielen Punkten wenig Information	ja	ja, Logging und Debugging mit Log4J	ja, Reports als XML-Datei	seit 2005, >19.000 Downloads auf Source- forge
Jemmy	ja, aktiv weiter- entwickelt, jedoch Doku- mentation nicht aktuell gehalten	katastrophal, veraltete Links und Dokumentation, neue Version Jemmy 3 vollkommen undokumentiert	ja	ja, wie JUnit	nein	seit 2009

Pounder	nein, letzte Version 2010	Dokumentation vorhanden, aber nicht allzu umfangreich, rudimentäre Tutorials	?	? internes Statuspanel, das aufgetretene Exceptions anzeigen kann	?	seit 2002, >4000 Downloads auf Source-forge
WindowTester Pro	ja	umfangreich, gut strukturierte Webseite	empfohlen für Windows oder Linux-GTK	ja, wie JUnit	ja, wenn Verwendung von Ant	seit Jänner 2012
Eclipse Jubula (Plugin-Variante)	ja, letzte Plugin-Version Juni 2013	umfangreich, auf Herstellerwebseite nur Dokumentation von kommerzieller Standalone-Version, jedoch großteils featuregleich, nach Installation des Plugins Zugriff in Eclipse auf Online-Doku	ja	ja, erstellt Screenshots bei Fehlschlagen eines Tests	ja, Reports in XML- oder HTML-Format	seit 2011, 1761 Instal-lationen des Plugins
GUIDancer	*nicht mehr näher betrachtet*					
fressia project	*nicht mehr näher betrachtet*					
Maveryx	? letzte Version 2011, Support-Forum noch aktiv	umfangreich, Tutorial und User Guide ist dem Repository beigelegt	ja	ja, Log-File wird erstellt, optional kann die Erzeugung von Screenshots im Falle des Fehlschlagens eines Tests aktiviert werden	ja, XML- bzw. HTML-Reports, inkl. Tabellen, Charts, Statistiken	seit 2011, >15000 Downloads auf Source-forge

Twist + Frankenstein	*nicht mehr näher betrachtet*					
Sikuli	ja	mittelmäßig, mit neuer Version 1.1 (2014) soll Dokumentation verbessert werden	ja	?	nein, nur durch Verwendung von HTML-TestRunner oder Robot Framework	seit 2010
Robot Framework	ja	umfangreich	ja	ja, Log-File als HTML-Datei, diese enthält auch Low-Level Informationen bei Fehlschlagen eines Tests	ja, HTML- bzw. XML-Report	seit 2008 als Open-Source-Projekt
Arbiter	*nicht mehr näher betrachtet*					
Avignon	*nicht mehr näher betrachtet*					
GTT	? letzte Version 2009, kein Support-Forum	katastrophal, auf Webseite gar keine Dokumentation auffindbar, auf Sourceforge nur veraltete Version (2006) dokumentiert und auch hier nur Installationsanleitung und Beschreibung der Buttons im UI	?	?	?	seit 2006, 4700 Downloads auf Source-forge
White	*nicht mehr näher betrachtet*					

GUITAR	ja	umfangreich, Funktionsweise und Setup beschrieben, auch Screencasts auf Youtube bereitgestellt	empfohlen wird Unix-System, Windows erfordert Installation von cygwin	ja, Log-File wird erstellt, enthält auch Exception-Traces	?	seit 2009, >7000 Downloads auf Source-forge
T-Plan Robot [1]	nein, kein professioneller Support, keine aktive Weiter-entwicklung neuer Features	umfangreich, jedoch nicht sehr strukturiert. Unklar, welche Features in Open-Source-Variante verfügbar sind	ja	?	ja, HTML-Reports mit Screenshots	seit 2009, >10000 Downloads auf Source-forge

Tabelle A.2: Gesammelte Informationen über die verschiedenen Werkzeuge zu den allgemeinen Kriterien

[1] T-Plan Robot ist gedacht als Client-Server-Szenario, d.h. zum Testen einer Remote-Applikation, die über einen Virtual Network Computing (VNC)-Server verwendet wird. Aus diesem Grund wurde T-Plan Robot in der Evaluierung nicht näher betrachtet.