Masterarbeit am Fachbereich Mathematik und Informatik
der Freien Universität Berlin

Vergleich der Entwicklungs- und Wartungskosten unterschiedlicher Capture & Replay-Tools beim Testen grafischer Benutzeroberflächen

Autor: Antonia Kresse
akresse@fu-berlin.de
MatNr. 4199997

Eingereicht am: 24. November 2016

1. Betreuer: Prof. Dr. Lutz Prechelt
2. Betreuerin: Prof. Dr. Margarita Esponda-Argüero

Zusammenfassung

Für das Testen grafischer Benutzeroberflächen existieren viele verschiedene Werkzeuge. Ziel dieser Arbeit ist es, eine ökonomisch begründete Aussage im Hinblick auf die Vor- und Nachteile verschiedener Capture & Replay-Tools zu treffen. Dazu erfolgt u. a. ein Vergleich über mehrere Generationen von Capture & Replay-Tools durch die Messung anfallender Entwicklungs- und Wartungskosten von Testfällen. Dabei können im Hinblick auf die Implementierung und den Wartungsaufwand der Testwerkzeuge signifikante Unterschiede ermittelt werden, während die Robustheit der jeweiligen Capture & Replay-Tools dagegen nahezu identisch ist. Angesichts der Relevanz eines effektiven Testwerkzeugs für den Erfolg eines Softwareprojekts sollte es unter Verwendung wissenschaftlich ermittelter Vergleichswerte sorgfältig ausgewählt werden. Diese Arbeit leistet in diesem Gebiet einen wichtigen Beitrag sowohl für praktische Zwecke, als auch die weitere Forschung. Teilergebnisse der vorliegenden Arbeit wurden zur Veröffentlichung angenommen in [KK16].

Abstract

The testing of graphical user interfaces can be performed by many different tools. The aim of this master thesis is a scientific statement regarding the advantages and disadvantages of various capture and replay tools with regard to their use in the economic context. This involves a comparison of the differences between their generations by measuring the incidental development and maintenance costs. The results show significant differences regarding the implementation and maintenance cost of the evaluated tools, whereas their respective test robustness seems to be almost identical. Choosing an effective tool for GUI testing based on scientifically determined comparative values is crucial to the success of a project. This master thesis provides thereby a substantial contribution to both practical purposes and further research. Portions of this work have been accepted for publication in [KK16].

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 Ausgangssituation . . . 7
- 1.2 Motivation . . . 7
- 1.3 Problem . . . 7
- 1.4 Lösungsidee . . . 8
- 1.5 Struktur der Arbeit . . . 8

2 Theoretischer Hintergrund **11**
- 2.1 Grundlagen des Testens . . . 11
- 2.2 Automatisiertes Testen von Software . . . 12
  - 2.2.1 Software Qualität . . . 12
  - 2.2.2 Kostensenkung . . . 13
- 2.3 Capture & Replay . . . 14
  - 2.3.1 Grundidee . . . 14
  - 2.3.2 Generationen von Capture & Replay-Tools . . . 15
    - 2.3.2.1 Erste Generation: Koordinatenbasiert . . . 15
    - 2.3.2.2 Zweite Generation: Komponenten-/Widget-basiert . . . 15
    - 2.3.2.3 Dritte Generation: Bilderkennung (VGT) . . . 17
  - 2.3.3 Weitere Arbeiten auf diesem Gebiet . . . 18

3 Lösungsansatz **21**
- 3.1 Vorgehen . . . 21
- 3.2 TESTONA . . . 22
- 3.3 Verwendete Capture & Replay-Tools . . . 25
  - 3.3.1 QF-Test . . . 25
  - 3.3.2 TestComplete . . . 26
  - 3.3.3 Eggplant . . . 27
- 3.4 Auswahl der Testszenarien . . . 29

4 Durchführung **35**
- 4.1 Erwartungen . . . 35
  - 4.1.1 Erwartete Fehlschläge von 4.1 zu 4.3 . . . 36
  - 4.1.2 Erwartete Fehlschläge von 4.3 zu 4.5 . . . 36
- 4.2 Aufbau der Testumgebung . . . 37
  - 4.2.1 Entwicklungsumgebung . . . 37
  - 4.2.2 Automatisierte Ausführungsumgebung . . . 38
- 4.3 Durchführung mit den einzelnen Tools . . . 39
  - 4.3.1 Eggplant . . . 39
    - 4.3.1.1 Einarbeitung . . . 40
    - 4.3.1.2 Arbeiten mit Grafiken . . . 40
    - 4.3.1.3 Arbeiten mit Variablen . . . 42
    - 4.3.1.4 Testskripte . . . 44
    - 4.3.1.5 MainSuite . . . 47

4.3.1.6 Entwicklungszeit und −probleme 49
4.3.1.7 Fehlschläge von TESTONA 4.1 zu 4.3 51
4.3.1.8 Fehlschläge von TESTONA 4.3 zu 4.5 52
4.3.1.9 Jenkins Ergebnisse 53
4.3.2 QF-Test 54
4.3.2.1 Einarbeitung 54
4.3.2.2 Arbeiten mit Grafiken 54
4.3.2.3 Arbeiten mit Variablen 55
4.3.2.4 Testskripte 56
4.3.2.5 MainSuite 58
4.3.2.6 Entwicklungszeit und −probleme 59
4.3.2.7 Fehlschläge von TESTONA 4.1 zu 4.3 63
4.3.2.8 Fehlschläge von TESTONA 4.3 zu 4.5 63
4.3.2.9 Jenkins Ergebnisse 65
4.3.3 TestComplete 65
4.3.3.1 Einarbeitung 65
4.3.3.2 Arbeiten mit Grafiken 66
4.3.3.3 Arbeiten mit Variablen 66
4.3.3.4 Testskripte 67
4.3.3.5 MainSuite 67
4.3.3.6 Entwicklungszeit und −probleme 69
4.3.3.7 Fehlschläge von TESTONA 4.1 zu 4.3 74
4.3.3.8 Fehlschläge von TESTONA 4.3 zu 4.5 75
4.3.3.9 Jenkins Ergebnisse 77

5 Auswertung 79
5.1 Gegenüberstellung der Tools 79
5.1.1 Implementierung 79
5.1.2 Robustheit 82
5.1.3 Wartungsaufwand 83
5.1.4 Zusammenfassung 85
5.2 Ticketanalyse 85
5.3 Grenzen der Arbeit 88

6 Ausblick 91

7 Fazit 93

Literaturverzeichnis 97

Abbildungsverzeichnis 101

Tabellenverzeichnis 103

Anhang 104

Eidesstattliche Erklärung 130

1 Einleitung

1.1 Ausgangssituation

Grafische Benutzeroberflächen (engl. *graphical user interface*, (GUI)) sind heutzutage allgegenwärtig [XGFC09] und daher ein wichtiger Bestandteil vieler Softwareprodukte. Die zunehmende Bedeutung spiegelt sich auch im Quellcode wieder, so dass sich bis zu 60% des Programmcodes auf die graphische Oberfläche beziehen können [Mem02]. Eine GUI ist das grafische Front-End eines Softwaresystems und oft der einzige für den Endbenutzer sichtbare Interaktionspunkt zwischen ihm und dem Programm. Daher ist nicht nur eine Korrektheit des Programms, sondern auch insbesondere eine fehlerfreie Integration und Funktionalität der dazugehörenden GUI notwendig. Zu den wichtigen Aspekten beim Testen grafischer Oberflächen gehören Testabdeckung, Testfallgenerierung, Testorakel und Regressionstests [QN13]. Im Rahmen dieser Arbeit erfolgt dabei eine Fokussierung auf die Testfallgenerierung mit automatisierten Tests unter der Verwendung von Capture & Replay-Tools.

1.2 Motivation

Das Testen GUI-basierter Anwendungen (engl. *GUI-based applications*,(GAPs)) bedarf eines hohen Arbeits- und Zeitaufwands, da GAPs aus vielen zu testenden Bildern und GUI-Objekten bestehen [XGFC09]. Häufig werden diese Tests manuell oder automatisiert mit Hilfe von Capture & Replay-Tools durchgeführt, um die Interaktionen der Endbenutzer zu simulieren. Der Vorteil automatisierter Tests im Vergleich zu einem manuellen Vorgehen ist deren schnelleres und höher frequentiertes Durchlaufen [BF12]. Ziel dieser Arbeit ist die qualitative Untersuchung der Funktionsweise von drei verschiedenen Capture & Replay-Tools im Hinblick auf deren Robustheit sowie Entwicklungs- und Wartungskosten.

1.3 Problem

In wissenschaftlichen Arbeiten zum Testen von GUIs werden Capture & Replay-Tools häufig gebündelt bzw. undifferenziert aufgeführt. Jedes Tool besitzt jedoch einen eigenen Algorithmus, der in Bezug auf das zu testende System (engl. *system*

under Test, (SUT)) Vor- oder Nachteile besitzen kann. Zudem können bei der Entwicklung von SUTs bereits minimale Änderungen im Quellcode die GUI verändern und somit bis zu 74% der GUI-Tests unbrauchbar machen [XGFC09]. Dabei haben die durchgeführten Änderungen im Quellcode (und damit an den Elementen der GUI) unterschiedliche Auswirkungen auf die verschiedenen Tests der Capture & Replay-Tools, so dass eine differenzierte Betrachtung ebenfalls sinnvoll wäre.

1.4 Lösungsidee

Um den Einsatz von Capture & Replay-Tools differenzierter zu betrachten, wird im Rahmen dieser Arbeit eine qualitative Untersuchung der Funktionsweise der drei verschiedenen Capture & Replay-Tools „QF-Test“, „TestComplete“ und „Eggplant“ auf dem in verschiedenen Versionen vorliegenden SUT TESTONA durchgeführt. Dazu werden für die Ausgangsversion des SUTs TESTONA 4.1 26 verschiedene Testfälle detailliert spezifiziert und anschließend durch die Capture & Replay-Tools implementiert bzw. aufgenommen. Für jeden Test erfolgt dabei die Messung der Entwicklungszeit und die Notation eventuell entstandener Probleme. Die fertig implementierten Tests werden nach ihrer Fertigstellung auf der aktualisierten SUT-Version TESTONA 4.3 ausgeführt und auf ihre Robustheit geprüft. Anschließend erfolgt die Ursachenanalyse der fehlgeschlagenen Tests sowie deren Reparatur. Die so reparierten Tests werden im Anschluss mit den fehlerfreien Testfällen erneut auf der neuesten SUT-Version TESTONA 4.5 ausgeführt, um wiederum Fehlschläge zu inspizieren und zu reparieren. Die im Rahmen dieses Vorgehens gewonnenen Erkenntnisse im Hinblick auf die Robustheit sowie Entwicklungs- und Wartungskosten der verschiedenen Capture & Replay-Tools ermöglichen dann im Ergebnis die angestrebte differenziertere Betrachtung.

1.5 Struktur der Arbeit

Im sich anschließenden zweiten Kapitel „Theoretischer Hintergrund“ werden die Grundlagen des Testens und automatisiertem GUI-Testen im Speziellem sowie die verschiedenen Generationen von Capture & Replay-Tools näher erläutert. Die unter 1.4 dargestellte Lösungsidee wird im dritten Kapitel „Lösungsansatz“ durch die Beschreibung des SUTs, die Auswahl der Testszenarien und die verwendeten GUI-Testing Tools konkretisiert. Das vierte Kapitel „Durchführung“ beschreibt dann, neben den vorher formulierten Erwartungen und dem Aufbau der Testumgebung, den Einsatz der einzelnen Tools sowie die damit verbundenen Herausforderungen. Die gewonnenen Erkenntnisse und Ergebnisse werden anschließend im fünften Kapitel „Auswertung“ vorgestellt, bewertet und u.a. im Hinblick auf vorhandene Grenzen

interpretiert. Neben einem zusammenfassenden Fazit erfolgt zudem abschließend ein Ausblick auf thematisch fortführende Arbeits- bzw. Forschungsmöglichkeiten.

2 Theoretischer Hintergrund

2.1 Grundlagen des Testens

Das Validieren und Verifizieren von Software ist ein essentieller Bestandteil moderner Softwareentwicklung, der 20-50 Prozent der Entwicklungskosten beansprucht und somit einen signifikanten Kostenfaktor darstellt [HS02, EBI06]. Beim Testen eines SUTs werden für die Überprüfung des Front-Ends (den grafischen Oberflächen) und des Back-Ends (dem zugrundeliegenden Quellcode) sowohl im Testentwurf als auch bei der anschließenden Entwicklung verschiedene Vorgehensweisen angewandt: Mit Kenntnis der Quellcodestruktur eines SUTs können Tester für das Back-End Scripte (z.B. in Form von JUnit-Tests) entwickeln, in denen sie die Rückgabewerte von Aufrufen abfangen und mit den erwarteten Werten vergleichen. Im Gegensatz dazu erfolgt das Testen des Front-Ends eines SUTs aus der Sicht von Endbenutzern, also ohne den zugrundeliegenden Quellcode direkt zu betrachten.

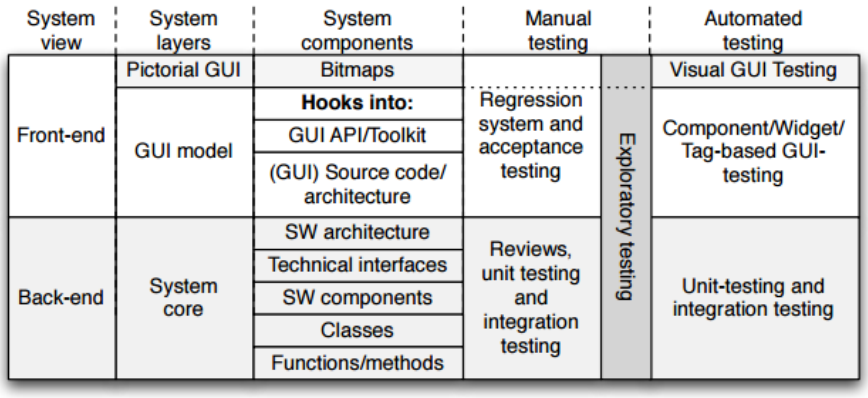

Abbildung 2.1: Theoretisches Modell eines Systems hinsichtlich der Unterscheidung von Front-End und Back-End, sowie deren Möglichkeiten für manuelles und automatisiertes Testen [Alé15]

GUI-Tests, als Teil des Front-Ends eines SUTs, fallen in die Kategorie der Black-box-Tests. Abbildung 2.2 verdeutlicht dazu den Unterschied zwischen Black- und Whitebox-Tests: Während in beiden Tests das Testobjekt mit den Eingabedaten ausgeführt und die Endergebnisse in Form von Ausgabedaten registriert werden, sind sowohl die Bestimmung der Eingangsdaten, als auch die Kenntnis bzw. Unkenntnis der inneren Struktur des Testobjektes während der Testausführung verschieden.

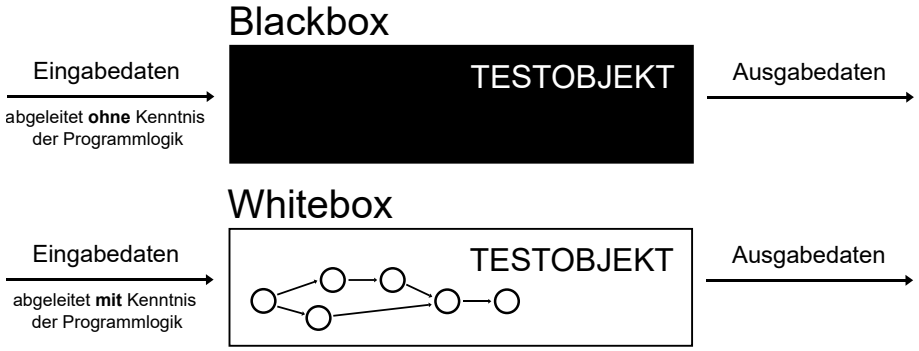

Abbildung 2.2: Unterschiede zwischen Blackbox und Whitebox

Die Grundlage des Blackbox-Testentwurfsverfahren ist die Analyse der Spezifikation: Sei es die Spezifikation von einem Test direkt oder der des SUTs. Die interne Struktur des Testobjektes wird dabei nicht berücksichtigt, so dass GUI-Testern, wie den Endbenutzern, nur die grafische Oberfläche des SUTs als Interaktionsmöglichkeit zur Verfügung steht. Die erwarteten Ergebnisse sind folglich nur auf der GUI zu erkennen: Wird beispielsweise nach dem Bedienen eines Menüs nicht das gewünschte bzw. erwartete Fenster geöffnet, so befindet sich das SUT nicht im spezifizierten Zustand und der dafür implementierte Test sollte fehlschlagen.

2.2 Automatisiertes Testen von Software

Der Einsatz automatisierter Tests in der Industrie beruht hauptsächlich auf dem Wunsch nach einer effektiven Qualitätsverbesserung der Software bei gleichzeitiger Minimierung der beim Testen entstehenden Kosten [RMPM12].

2.2.1 Software Qualität

Automatisierte Tests verbessern durch eine höhere Ausführungsgeschwindigkeit im Vergleich zu manuellen Tests die Qualität einer Software [RMPM12, BWK05]. Die höhere Testfrequenz ermöglicht eine schnellere Aufdeckung von Defekten und die damit verbundene Rückmeldung an die Entwickler. Die schnellere Aufdeckung von Defekten senkt die Entwicklungszeit des Projektes und verringert eine mögliche Ausbreitung des Defekts. Zudem kann eine frühe Fehlererkennung das Auftreten von Synergieeffekten, in denen zwei oder mehr Defekte zusammen zu weiteren Defekten führen, unterbinden und so eine komplexere, kostenintensivere Reparatur präventieren.

Voraussetzung für eine effektive Automatisierung von Tests ist dabei stets eine individuell geringe Testausführungszeit. Dies ist besonders wichtig, wenn die Tests für eine kontinuierliche Integration und Entwicklung der Software benötigt werden [OAB12]. Ein Beispiel ist das Ausführen der Tests nach jedem Commit eines

Entwicklers, wenn dieser neuen Code für die Software geschrieben hat. Die schnelle Ausführung mehrerer hundert Unit-Tests innerhalb weniger Minuten ist einer der Gründe, warum Unit-Tests in der Industrie sehr häufig zum Einsatz kommen [RUCH01, CL02, Ola03].

2.2.2 Kostensenkung

Automatisiertes Testen ermöglicht eine Kostensenkung, da Tests andernfalls manuell mit höherem Zeitaufwand mehrfach ausgeführt werden müssten. Dabei sind allerdings zusätzlich entstehende Kosten zu berücksichtigen, die im Folgenden vorgestellt werden.

Jedes automatisierte Testverfahren benötigt ein Werkzeug, das erworben oder entwickelt werden muss. Des Weiteren müssen die für die Anwendung des Werkzeugs vorgesehenen Tester Schulungen oder Einarbeitungszeit erhalten, um einen effektiven Umgang mit dem Testwerkzeug zu erlernen. Die dafür benötigte Zeit hängt direkt von der Komplexität des Werkzeugs ab [RMPM12] und führt somit zu davon abhängigen Kosten. Zudem haben viele automatisierte Testtechniken bestimmte Voraussetzungen, können also beispielsweise nur für bestimmte Plattformen, Betriebssysteme oder Programmiersprachen verwendet werden [Fin01, SAP06]. Daher empfiehlt sich beispielsweise die Durchführung eines Pilotprojekts, in dem eine Überprüfung der Anwendbarkeit und Flexibilität der Technik auf dem gewünschten SUT stattfindet. Die Ausführung eines Pilotprojekts ist meist mit erheblichen Kosten verbunden, eine entsprechende Vernachlässigung kann allerdings bei Änderungen des Softwareprozesses zu einem späteren Zeitpunkt zu unerwarteten Kosten führen.

Ein weiterer relevanter Kostenpunkt des automatisierten Testens ist die Wartung von Testskripten. Es handelt sich dabei um einen kontinuierlichen Kostenpunkt, der proportional mit der Größe der Testsuite ansteigt. Die Wartung ist zwingend notwendig, um die Tests stets den Anforderungen entsprechend aktuell zu halten [BEF11] und zu garantieren, dass die Tests nur durch Defekte des SUTs fehlschlagen, jedoch nicht durch dessen Änderung. Umfangreiche Änderungen können dabei zu enormen Kosten führen, die im schlimmsten Fall nicht tragbar sind [SAP06].

Für jede durchführbare Testautomatisierungstechnik sollten die Entwicklungs- und Wartungskosten eine Anlagenrentabilität (*engl. return of investment*, (ROI)) bieten. Folglich sollte die Gesamtbetrachtung der Kostenfaktoren wie Erlernbarkeit, Benutzerfreundlichkeit und Wartbarkeit, eine Antwort auf eine mögliche Umsetzbarkeit der Testautomatisierung liefern. Während die Nutzerfreundlichkeit und Erlernbarkeit zwar Berücksichtigung finden, liegt der Fokus dieser Arbeit auf der Robustheit und Wartbarkeit.

2.3 Capture & Replay

2.3.1 Grundidee

Etablierte kommerzielle Produkte für das Testen von GUIs setzen vorwiegend auf eine Aufnahme- und Wiedergabefunktion der jeweiligen Nutzerinteraktionen, indem die Ausführung des SUTs und die Bedienung durch den Nutzer, insbesondere die dabei stattfindenden Aufrufe von Programmbibliotheken- und Systemschnittstellen, überwacht und später wahlweise angepasst und wiedergegeben werden können. Dies erfolgt durch die Verwendung der Aufnahmemöglichkeit (Capture), mit der ein Tester eine ausgeführte Anwendung in einer interaktiven Sitzung aufzeichnen kann. Dabei protokolliert das Capture & Replay-Tool alle Ereignisse wie Tastatur- und Mausevents und notiert sie in einem Testskript (vgl. Abbildung 2.3). Das entstandene Testskript kann anschließend, ohne zusätzliche menschliche Interaktion, erneut abgespielt werden (Replay).

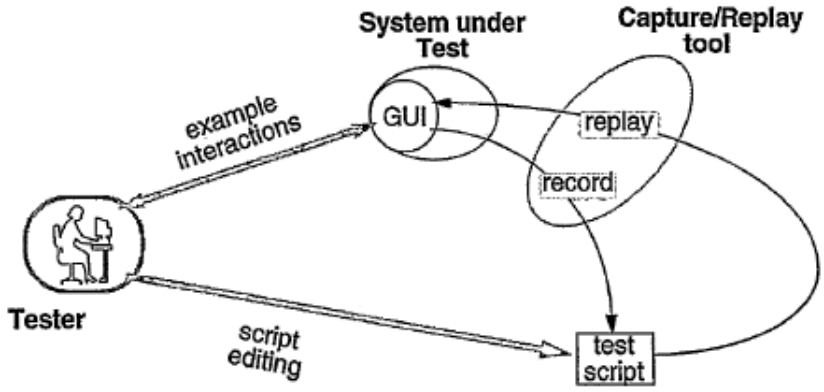

Abbildung 2.3: Funktionsweise von Capture & Replay-Tools aus Ostrand u. a. (1998)

Die Capture & Replay-Tools verbergen dabei einen Großteil der zugrundeliegenden technischen Strukturinformation, die eine graphische Benutzeroberfläche ausmachen. Die Erfassung der Objektinformation und das Aufzeichnen der Testskripte geschieht in einer logischen Repräsentation, so dass die Tester die Implementierungsdetails nicht kennen müssen. Bei Änderungen und Erweiterungen der Bedienoberfläche tritt allerdings gelegentlich das Problem auf, dass die aufgezeichneten Skripte nicht mehr zur GUI passen und eine manuelle Anpassung der Skripte notwendig wird [SL10]. Um ein effizientes Testen grafischer Oberflächen zu gewährleisten, fassen Ostrand u. a. [OAFG98] daher folgende Anforderungen an Capture & Replay-Tools zusammen:

- Alle Bereiche der Funktionalität des Systems sollen abgedeckt werden.

- Jeder Bestandteil der GUI des Systems muss ausgeführt werden: Alle Eingabeformulare, individuellen Entscheidungsmöglichkeiten und Folgen von Nutzerinteraktionssequenzen, die in der Praxis auftreten können.

- Die Tests sollten bei einer Veränderung des Systems wartbar sein und benutzbar bleiben, so dass ein minimaler Arbeitsaufwand entsteht um vormals aufgezeichnete Testskripte auch auf einer neuen Version des Systems laufen zu lassen.

Neben den die Effektivität, Abdeckung und Wartbarkeit betreffenden Kriterien sollten die Tests zudem einen geringen Entwicklungsaufwand und eine hohe Wartbarkeit besitzen.

2.3.2 Generationen von Capture & Replay-Tools

Im Laufe der Zeit haben sich verschiedene Techniken zum Testen von grafischen Oberflächen entwickelt. Alégroth u. a. unterteilen diese in drei Generationen, sortiert nach ihrer chronologischen Entstehung [AGOM15]. Die Unterschiede der Generationen basieren auf ihrer Interaktion mit dem SUT, beispielsweise durch konkrete Koordinaten, das Zugreifen auf GUI-Komponenten oder der Bilderkennung des SUTs. Im Folgenden werden die drei Generationen und ihre Funktionsweise näher erläutert.

2.3.2.1 Erste Generation: Koordinatenbasiert

Die erste Generation von Capture & Replay-Tools beruht auf der Interaktion mit dem SUT über Koordinaten [GXF09]. Beim Aufzeichnen bzw. Schreiben der Interaktionen mit dem SUT werden dementsprechend die Koordinaten für die spätere Wiedergabe in den Skripten gespeichert. Das ausschließliche Verwenden von Koordinaten zum Identifizieren von GUI-Elementen weist jedoch eine hohe Fehleranfälligkeit auf. Bereits minimale Änderungen an der grafischen Oberfläche können zum Fehlschlag eines Tests führen [NRBM14, HS93] und sorgen für kostenintensive Wartungen. Daher hat sich diese Generation in der Praxis nicht durchgesetzt, wird jedoch teilweise noch als Basiskomponente von anderen Testautomatisierungstools verwendet.

2.3.2.2 Zweite Generation: Komponenten-/Widget-basiert

Die zweite Generation von Capture & Replay-Tools interagiert mit bzw. steuert das SUT durch einen direkten Zugriff auf die GUI-Komponenten, was auch als Komponenten- oder Widget-basiertes Testen bezeichnet wird. Dabei kann beispielsweise ein Zugriff auf die GUI-Bibliotheken oder Toolkits erfolgen [SAP06]. Durch

die Interaktion mit den GUI-Komponenten werden robuste Testfälle erzeugt, da ein direkter Zugriff auf die Komponenten über deren IDs, Labels, etc. möglich ist. Das Ansteuern von Komponenten ohne Koordinaten macht die Tests unabhängiger von Auflösungen oder anderen minimal sichtbaren Änderungen. Ändert sich jedoch eine auf der GUI unsichtbare Eigenschaft im Quellcode, so kann dies zum Fehlschlag eines Tests führen. Ein weiterer Vorteil des direkten Zugriffs auf die GUI-Komponenten ist das schnellere Ausführen der Tests, da ein Zugriff ohne das Laden bestimmter Elemente möglich ist.

Die meisten Tools der zweiten Generation unterstützen die Funktionalität von Capture & Replay und sorgen somit für ein einfaches Aufnehmen von Interaktionen mit dem SUT und die damit verbundene Senkung der Kosten in der Testentwicklung. Tester erhalten zudem meist Unterstützung durch eine automatische Verwaltung der Eigenschaften der GUI-Komponenten, da beispielsweise eine manuelle Zuordnung der ID zu einzelnen GUI-Elementen nicht intuitiv möglich ist. Welche Eigenschaften zum Identifizieren der Komponenten verwendet werden, bestimmt jedes Tool (mit Konfigurationsmöglichkeiten durch den Tester) selbst. Ein Beispiel für den Aufbau eines Tests mit einem Capture & Replay-Tool der zweiten Generation ist in Abbildung 2.4 dargestellt.

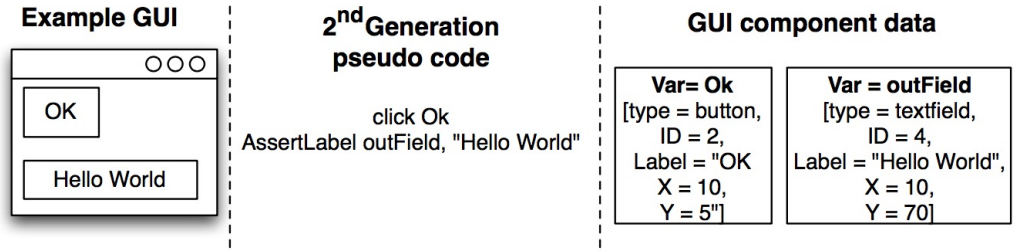

Abbildung 2.4: Aufbau von GUI-Komponenten in Tools der zweiten Generation ([Alé15])

Durch die Interaktion mit GUI-Komponenten über eine tiefere Ebene als die Endbenutzer sichtbare grafische Oberfläche, können die Tests nicht die Überprüfung der vollen Funktionalität der GUI garantieren. Meist erfordert der Zugriff auf die Komponenten bestimmte Programmiersprachen oder GUI-Bibliotheken. Bekannte Bibliotheken wie AWT oder Java Swing werden dabei häufig unterstützt, jedoch können selbstentwickelte GUI-Komponenten ein Problem darstellen, wenn das Testtool auf diese nicht zugreifen kann. In diesem Fall müssen die Komponenten zusätzlich angepasst werden, was bei deren Änderung zu zusätzlichen Wartungskosten führen kann [HS93] [BWK05]. Ein zusätzliches Problem stellen Komponenten dar, die erst während der Laufzeit generiert werden und deren Eigenschaften vorher nicht bekannt sind.

Eine nähere Vorstellung der in dieser Arbeit untersuchten Capture & Replay-Tools der zweiten Generation „QF-Test“ und „TestComplete“ erfolgt in Kapitel 3.3 „Verwendete Capture & Replay-Tools“.

2.3.2.3 Dritte Generation: Bilderkennung (VGT)

Die dritte Generation der Capture & Replay-Tools wird ebenfalls für automatisierte GUI-Tests verwendet und nutzt das Prinzip der Bilderkennung. Daher wird in diesem Zusammenhang auch von visuellem GUI-Testen (engl. *visual GUI Testing*, (VGT)) gesprochen [AFO13]. Im Gegensatz zur zweiten Generation wird in VGT-Tests nicht auf GUI-Komponenten zugegriffen, sondern lediglich eine Bilderkennung der tatsächlichen Grafiken auf dem Monitor angewendet [AFO13, BF12]. Aufgrund dieser Eigenschaft können VGT-Tools GUI-Elemente unabhängig von deren Position identifizieren und sind somit robuster bezüglich Positionsänderungen im GUI-Layout [AFO13, BF12], Änderungen in der API oder dem Quellcode des SUTs. Hinsichtlich grafischer Änderungen (z.B. Größe und Farbe von Komponenten) sind die Tests jedoch sehr anfällig [AFO13, BF12]. Für VGT-Tests ist das SUT eine Blackbox und nicht intrusiv, so dass weder Informationen bezüglich der Programmiersprache, noch des zugrundeliegenden Aufbaus des SUTs zum Erstellen der Tests notwendig sind. VGT-Tools können daher jede Form eines SUTs testen, solange dieses eine grafische Oberfläche besitzt.

VGT-Skripte werden als Szenarien aufgenommen bzw. geschrieben und enthalten Aktionen, wie Mausklicks und Tastaturevents, sowie eine Vielzahl an Bildern. Diese Grafiken werden für den Bilderkennungsalgorithmus verwendet, um Ausschnitte des SUTs aus der Sicht des Endbenutzers in der GUI des SUTs zu finden und mit ihnen zu interagieren. Die Skripte sind meist intuitiv verständlich und benötigen kaum technisches Grundwissen, da die Syntax stark den Interaktionen mit dem SUT zugeordnet ist [YCM09]. Eine vereinfachte Darstellung für den Quellcode eines VGT-Tools ist in Abbildung 2.5 zu betrachten.

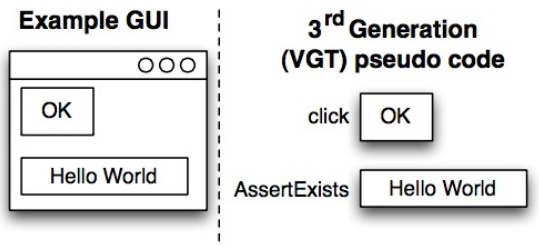

Abbildung 2.5: Aufbau von GUI-Komponenten in Tools der dritten Genetation ([Alé15])

Die Bilderkennung bei der Wiedergabe des Tests wird in zwei Schritte unterteilt: Zunächst wird das aktuelle GUI-Abbild als Bitmap erfasst und zusammen mit dem

zu findenden Bildausschnitt an den Bilderkennungsalgorithmus weitergegeben. Im zweiten Schritt prüft der Algorithmus, ob der Bildausschnitt im Screenshot sichtbar ist. Bei einer Übereinstimmung werden die dazu passenden Koordinaten zurückgegeben, bei einem Fehlschlag ein False oder eine Exception. Der Bilderkennungsalgorithmus ist einer der wichtigsten Bestandteile eines VGT-Tools, da er zu falsch-positiven Testergebnissen führt, sofern er nicht deterministisch ist. In diesem Fall würde ein Vergleich fehlschlagen, obwohl der gesuchte Bildausschnitt sichtbar ist [AFR15]. VGT-Tools nutzen verschiedene Algorithmen für den Bildvergleich, wodurch jeweils unterschiedliche Ergebnisse entstehen können. Die meisten Algorithmen basieren auf einem Vergleich der Ähnlichkeit der Bildausschnitte. Häufig wird eine Ähnlichkeit von 70-80% erwartet, um mit minimalen Änderungen umgehen zu können [YCM09].

Im Vergleich zur zweiten Generation ist die Testausführung bei VGT-Tests meist langsamer, da nur Bildausschnitte vom aktuellen Abbild des SUTs verglichen werden können. Somit kann ein Testschritt erst ausgeführt werden, wenn sein Vorgänger beendet ist und sich die GUI des SUTs aktualisiert hat. Eine nähere Vorstellung des in dieser Arbeit untersuchten Capture & Replay-Tools der dritten Generation „Eggplant“ erfolgt in Kapitel 3.3 „Verwendete Capture & Replay-Tools“.

2.3.3 Weitere Arbeiten auf diesem Gebiet

In ihrem Paper „Coverage Criteria for GUI Testing“ [MSP01] thematisieren Memon et al. Abdeckungskriterien für GUIs, da sich diese signifikant von den Kriterien für Quellcode unterscheiden. Bei einer GUI kann im Gegensatz zum Quellcode nicht geprüft werden, ob alle Zeilen Code einmal durchlaufen werden oder ob jede Schleife oder jeder Zweig einmal ausgeführt wird. Dies liegt unter anderem daran, dass GUI-Elemente auf einer höheren Ebene getestet werden. Da eine grafische Oberfläche viele Elemente enthält, unterteilen Memon et al. die GUI in Komponenten, die anschießend einzeln geprüft werden können. Dabei besteht eine Komponente aus Ereignissen, die unabhängig von Events aus anderen Komponenten sind. Insgesamt stellen Memon et al. zwei Abdeckungskriterien auf: Die intra-komponenten Abdeckungskriterien für Events innerhalb einer Komponente und die inter-komponenten Abdeckungskriterien für Events zwischen Komponenten. Ihre Arbeit beinhaltet zur Unterstützung der Kriterien darüber hinaus auch eine Fallstudie. In meiner Arbeit spielt die Abdeckung aller GUI-Elemente keine entscheidende Rolle, da der Fokus auf der Robustheit und Wartbarkeit der Tests liegt, die durch die Abdeckung nicht beeinflusst werden.

Studien mit einem Fokus auf die ökonomischen Aspekte für die Entwicklung und Wartung von GUI-Tests sind aktuell noch rar. Eine Ausnahme bildet die Arbeit von Khazal und Sigurdsson, die einen Vergleich der Test-Tools „HP QuickTest Professional“ und „HP Unified Functional Testing“ durchführen [KS15]. Mit beiden Werkzeu-

gen werden 20 zwischen beiden Autoren aufgeteilte Testfälle implementiert, so dass jeweils beide mit den Tools Tests erstellt haben. Der Fokus ihrer Arbeit liegt ebenfalls auf der Entwicklungszeit und Robustheit der Tests. Die Robustheit wird dabei durch wiederholte Ausführungen auf dem SUT gemessen, wobei sich die Version des SUT innerhalb der Studie nicht ändert. Aussagen bezüglich der Komplexität des SUTs sowie Details der Testfälle werden in ihrer Arbeit jedoch nicht veröffentlicht.

3 Lösungsansatz

3.1 Vorgehen

Im Rahmen dieser Arbeit erfolgt der Vergleichg der drei verschiedenen Capture & Replay-Tools „QF-Test“, „TestComplete“ und „Eggplant“. Dazu werden mit ihnen jeweils die gleichen Tests bezüglich des SUTs TESTONA [KL10] aufgenommen. Diese Tests werden zunächst für die erste Version TESTONA 4.1 entwickelt und anschließend auf der zweiten TESTONA 4.3 und dritten Version TESTONA 4.5 ausgeführt. Ziel ist dabei ein Vergleich der benötigten Entwicklungs- und Wartungskosten der Capture & Replay-Tools um zu ermitteln welches Tool unter ökonomischen Gesichtspunkten bezüglich eines sich stetig ändernden SUTs am besten geeignet ist. Bedingt durch die für diese Arbeit zur Verfügung stehenden Zeit wird nur ein SUT verwendet, das in Kapitel 3.2 „TESTONA“ detailliert beschrieben wird. Nach Alégroth [AGOM15] sind dabei einzig die GUI-Komponenten des SUTs relevant, so dass die Größe der Systemlogik vernachlässigt werden kann. Daher wird die zugrundeliegende Komplexität des Quellcodes von TESTONA im Rahmen dieser Arbeit nicht weiter thematisiert, da auch die Capture & Replay-Tools ohnehin ausschließlich die GUI von TESTONA überprüfen.

Vor der Entwicklung der Tests für die erste Version des SUTs werden die 26 verschiedenen Tests spezifiziert, so dass möglichst viele verschiedene Komponenten des SUTs Berücksichtigung finden. Eine konkrete Beschreibung dieser Tests erfolgt in Kapitel 3.4 „Auswahl der Testszenarien“, der jeweils gleiche Testablauf für die Capture und Replay-Tools kann in Abbildung 3.1 nachvollzogen werden.

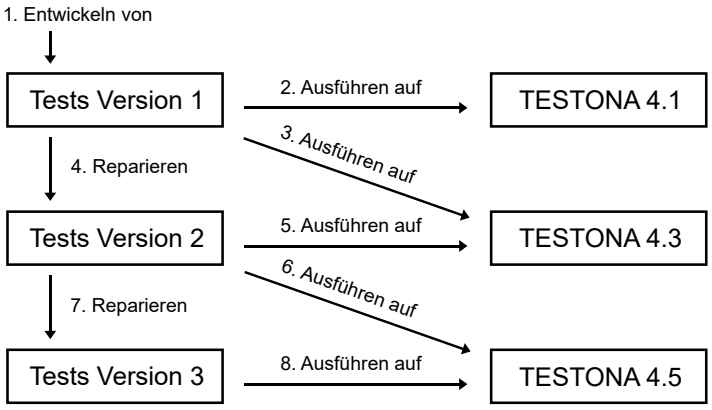

Abbildung 3.1: Testablauf

Nach der Spezifikation der Tests folgt deren Entwicklung für (Schritt 1) sowie die anschließende Ausführung auf (Schritt 2) TESTONA Version 4.1., um ihre Korrektheit und Funktionalität zu überprüfen. Anschließend werden die Tests auf TESTONA 4.3 (Schritt 3) erneut ausgeführt, um die aus den Änderungen zwischen den beiden TESTONA-Versionen resultierenden, fehlgeschlagenen Tests zu ermitteln. Daraufhin erfolgt das Reparieren der fehlgeschlagenen Tests (Schritt 4), wodurch eine zweite Version der Tests entsteht, die daraufhin zur Funktionalitätsüberprüfung auf TESTONA 4.3 (Schritt 5) und zur Fehlschlagsanalyse auf TESTONA 4.5 (Schritt 6) ausgeführt werden. Abschließend werden die Tests noch einmal auf TESTONA 4.5 angepasst bzw. repariert (Schritt 7) und dann abschließend nochmal auf dem SUT ausgeführt (Schritt 8).

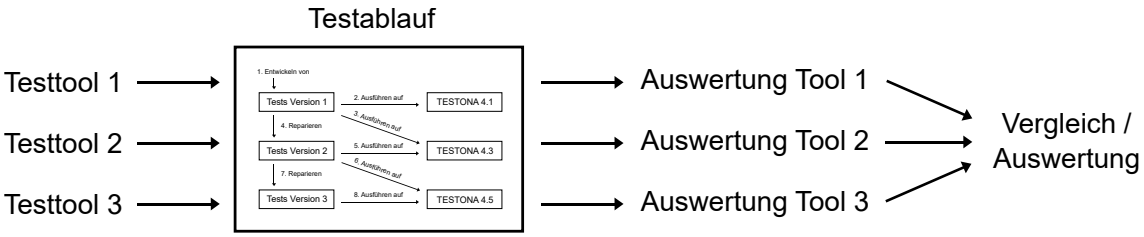

Abbildung 3.2: Vorgehensweise zur Datenerfassung

Während des Testablaufs erfolgt in den verschiedenen Schritten die Betrachtung diverser Aspekte: Zunächst wird beim Entwickeln (Schritt 1) und Reparieren der Tests (Schritt 4 und 7) die benötigte Zeit gemessen und eventuell aufgetretene Probleme dokumentiert. Während der Ausführung (Schritt 2, 3, 5, 6 und 8) erfolgt die Dokumentation und Kategorisierung fehlgeschlagener Tests. Dieser Testablauf wird von allen drei Capture & Replay-Tools vollzogen und liefert anschließend jeweils ein Ergebnis pro Testtool (vgl. Abbildung 3.2). Die erarbeiteten Ergebnisse werden abschließend untereinander verglichen, um Vor- und Nachteile der unterschiedlichen Capture & Replay-Tools aufzuzeigen.

3.2 TESTONA

Das im Rahmen dieser Arbeit verwendete SUT „TESTONA“ ist ein graphischer Editor, der die Anwendung der Klassifikationsbaum-Methode [GG93] unterstützt. Ursprünglich Klassifikationsbaum-Editor genannt, wurde er im Rahmen der Forschungstätigkeiten der Daimler-Benz AG entwickelt [LW00]. Seit 2008 wird das Tool von der „Berner & Mattner Systemtechnik GmbH“ weiter entwickelt und 2014 in TESTONA umbenannt. Aufgrund der Namensänderung und diversen Vorgängerversionen besitzt die erste Version von TESTONA bereits die Versionsnummer 4.1. Um einen signifikanten Abstand zwischen den zu testenden Versionen des SUTs zu gewährleisten, wurden die Versionen 4.1, 4.3 und 4.5 von TESTONA ausgewählt.

Tabelle 3.1 verdeutlicht die Änderungen zwischen den jeweiligen Versionen. Dabei fällt auf, dass der Entwicklungsumfang von Version 4.1 zu 4.3 niedriger ist als von Version 4.3 zu 4.5, was sich auch in den zu erwartenden Fehlschlägen der Tests widerspiegelt.

	4.1 zu 4.3	4.3 zu 4.5
Zeit zwischen Releases	37 Wochen	27 Wochen
Bearbeitete Tickets	22	53
- davon neue Features	10	22
Änderungen LOC		
- geänderte Datein	187	488
- hinzugefügte Zeilen	2814	8177
- gelöschte Zeilen	1936	4504

Tabelle 3.1: Entwicklung von TESTONA

Aufgrund der vielen Interaktionsmöglichkeiten die TESTONA dem Nutzer bietet, ist sowohl die Oberfläche komplex, als auch die Auswahl an GUI-Elementen umfangreich. Durch die enorme Auswahl an diversen Komponenten eignet sich TESTONA hervorragend als SUT. Der Quellcode des Tools ist in Java, wobei für die grafische Oberfläche wird zusätzlich die SWT Library verwendet. Die GUI von TESTONA ist in Abbildung 3.3 dargestellt.

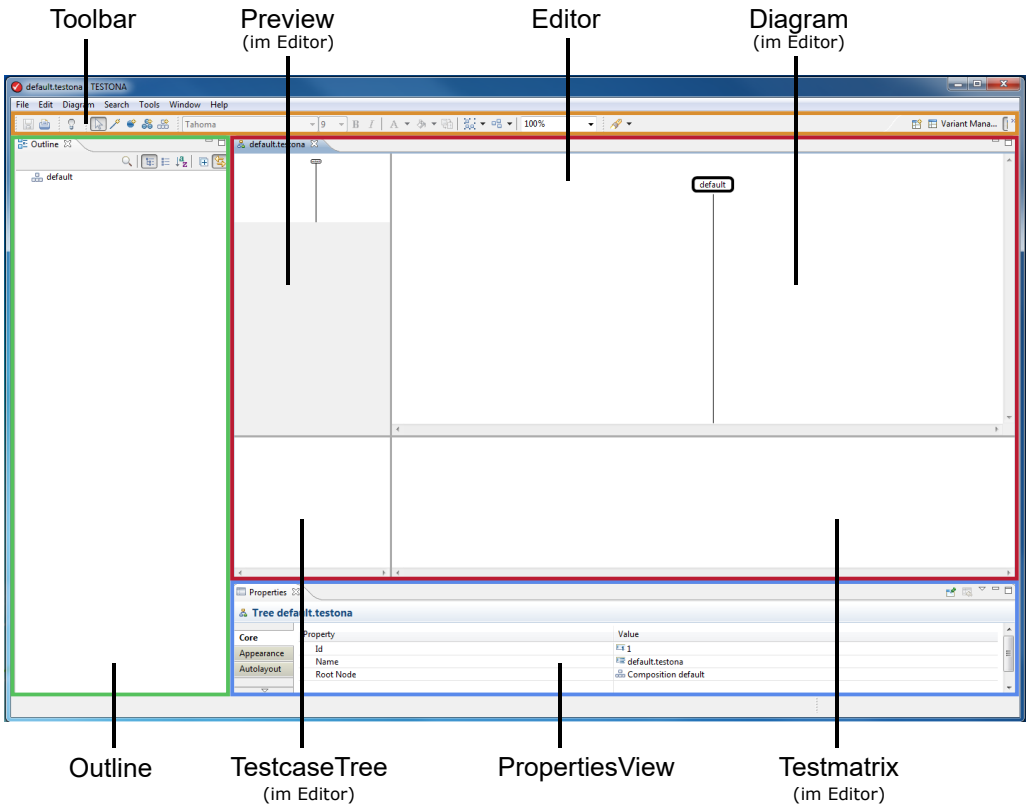

Abbildung 3.3: GUI-Komponenten von TESTONA

Hauptbestandteile der GUI von TESTONA, neben der Menüleiste, sind die Toolbar, der Editor, die Outline und die Properties View. Der **Editor** unterteilt sich dabei erneut in vier verschiedene Komponenten: Die Preview, das Diagramm, den TestcaseTree und die Testmatrix. Im Diagramm können Klassifikationsbäume erstellt werden, indem über Hotkeys, die Toolbar oder das Rechtsklick-Menü Klassifikationen, Kompositionen oder Klassen eingefügt und verbunden werden. Besonders in diesem Bereich der GUI kann der Nutzer viele Drag & Drop Interaktionen ausführen. Um bei umfangreicheren Bäumen eine sinnvolle Übersicht zu gewährleisten, wird der im Diagramm erstellte Klassifikationsbaum in der Preview noch einmal kleiner abgebildet. In der Preview wird durch einen interaktiven Rahmen der im Diagramm aktuell sichtbare Ausschnitt des Klassifikationsbaums markiert. Der TestcaseTree unterhalb der Preview beinhaltet die erstellten Testfälle, Testgruppen oder Testsequenzen in einer hierarchisch verschachtelten Liste. Rechts daneben befindet sich die Testmatrix, in der verschiedene Markierungen für die jeweiligen Testfälle gesetzt werden können. Abbildung 3.4 zeigt die einzelnen Elemente der GUI von TESTONA anhand eines beispielhaften Testdesigns.

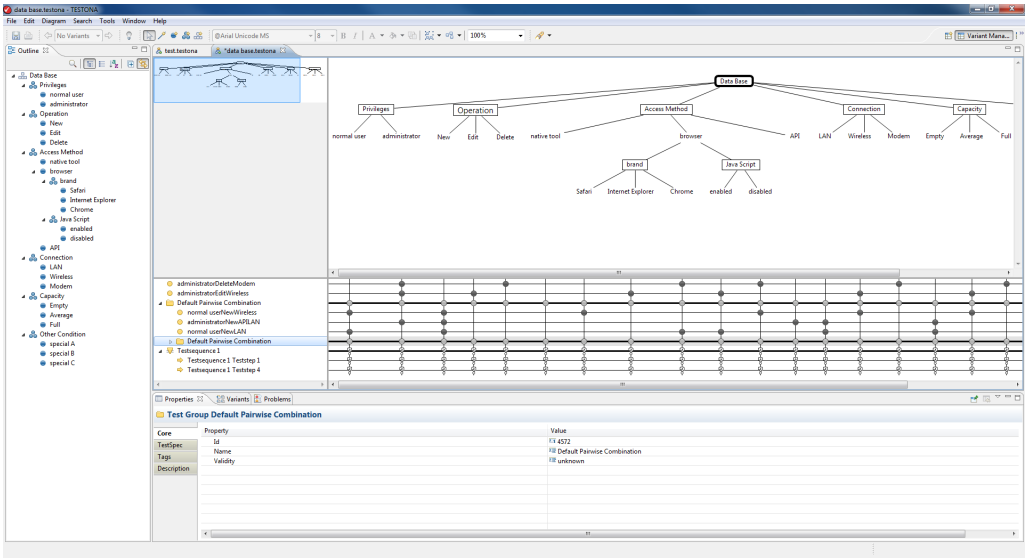

Abbildung 3.4: Exemplarisches Testdesign in der GUI von TESTONA

Die **Outline** stellt den im Diagramm erstellten Baum in hierarchischer Listenform dar. Auch hier ist das Erstellen von Klassifikationen, Kompositionen und Klassen durch verschiedene Drag & Drop Interaktionen möglich. Die Outline besitzt eine eigene Toolbar, über die unter anderem die Darstellung der Baumelemente geändert werden kann. Die im unteren Bereich der GUI verankerte **PropertiesView** bietet verschiedene Möglichkeiten zur Anpassung der Einstellungen, beispielsweise von Testfällen oder Baumelementen. Ein Großteil der Funktionen von TESTONA wird über die standardisierte **Menüleiste** gestartet, die häufig ein jeweils neues Dialogfenster für weitere Funktionsoptionen öffnet. Die unterhalb der Menüleiste

verankerte **Toolbar** enthält verschiedene Buttons, die einen schnelleren Zugriff auf diverse Funktionen (z.B. das Einfügen von Kompositionen und Klassifikationen) ermöglichen.

3.3 Verwendete Capture & Replay-Tools

Im Rahmen dieser Arbeit werden die drei Capture & Replay-Tools „QF-Test“, „TestComplete“ und „Eggplant“ untersucht. Deren Auswahl gründet dabei auf Erfahrungswerten beim Testen des im vorherigen Kapitel 3.2 vorgestellten SUT TESTONA, die im Rahmen der Arbeit beim Entwicklungsdienstleister „Berner & Mattner“ im Verlauf der letzten Jahre entstanden sind. TestComplete, das vor einigen Jahren in einer älteren Version verwendet und dann aufgrund von Problemen ausgewechselt werden musste, sowie QF-Test, welches aktuell bei „Berner & Mattner“ zum Einsatz kommt, vertreten die zweite Generation der Capture & Replay-Tools. Das im Gartner Report 2016 [PSH+16] als gut bewertete Eggplant repräsentiert dagegen den neuen Ansatz der dritten Generation. Im Folgenden werden die einzelnen Capture & Replay-Tools näher vorgestellt.

3.3.1 QF-Test

QF-Test (Version 4.0.10) ist als Capture & Replay-Tool ein GUI-Testwerkzeug zur Erstellung, Ausführung und Verwaltung von automatisierten System- und Lasttests für Java- und Web-Anwendungen mit grafischer Benutzeroberfläche. Das Tool wird seit 1999 von der Quality First Software GmbH (QFS) entwickelt und vertrieben. Laut eigener Aussage von QFS „können mit QF-Test modulare und wiederverwendbare Tests erstellt werden, die durch eine hohe Anwenderfreundlichkeit und niedrigem Preis zu einem hohen Return on Investment (ROI) führen“ [QFT16]. Die Tests können für Java Swing, SWT, Eclipse Plugins, RCP Anwendungen, ULC, Applets, WebStart sowie JavaFX unter den Betriebssystemen Windows und Linux/Unix entwickelt werden. Des Weiteren wird eine zuverlässige Erkennung von komplexen und dynamischen GUI-Komponenten versprochen, so dass die Tests tolerant gegenüber Änderungen an der GUI und damit wartungsarm sind. QF-Test ist in den Sprachen Deutsch und Englisch verfügbar und besitzt jeweils eine passende Dokumentation und Support. Der Preis für eine zeitlich unbeschränkte Entwicklerlizenz liegt bei 1.995€ und ist für eine GUI-Technologie gültig. Bei der Lizenz für eine GUI-Technologie wird zwischen Swing, JavaFX, SWT und Web unterschieden. Die Upgrades und der (beim Neukauf verbindliche) Support kosten für ein Jahr 480€.

Abbildung 3.5 zeigt das Hauptfenster von QF-Test: Auf der linken Seite ist eine Testsuite in Form eines hierarchischen Baums repräsentiert, der Kontrollstrukturen und Testdaten vereint und eine übersichtliche Verwaltung sowie einfache Bedienung

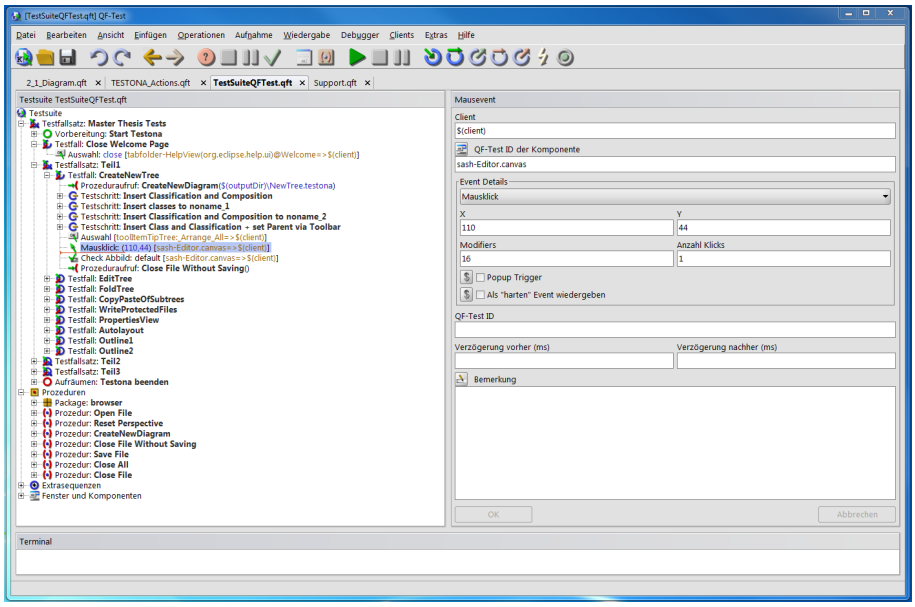

Abbildung 3.5: Grafische Oberfläche von QF-Test

ermöglicht. Auf der rechten Seite befindet sich die Detailansicht mit allen Attributen des aktuell im Baum ausgewählten Knotens. Insgesamt existieren mehr als 60 verschiedene Arten von Knoten, von denen manche nur als Datencontainer fungieren, während andere den Ablauf von Tests steuern oder sogar selbst einen Test darstellen. Nach der Installation von QF-Test muss in einer TestSuite zunächst die Verbindung zum SUT aufgebaut werden. Anschließend kann über die Toolbar mit Hilfe des Record-Buttons ein Test aufgezeichnet und gestoppt werden. Daraufhin erscheint dieser im linken Bereich von QF-Test und kann sofort bearbeitet oder wiedergegeben werden, was eine zügige und einfache Testaufnahme ermöglicht. Nach einem Testdurchlauf kann ein Protokoll geöffnet werden, das sämtliche Schritte des ausgeführten Tests aufführt. Eine fertige Suite kann mit Hilfe des Plugins „Jenkins“ zu jeder Zeit gestartet werden und liefert anschließend detaillierte Ergebnisse und Informationen über den Testlauf, wie beispielsweise Gesamtergebnisse mit Ausführungszeiten, Fehlerübersicht und Fehlerdetails inklusive Bildschirmabbildern zum jeweiligen Fehlerzeitpunkt.

3.3.2 TestComplete

TestComplete (Version 11.31) ist ein Capture & Replay-Tool der zweiten Generation und wird von der Firma „SmartBear Software“ entwickelt. Mit dem Tool können sowohl Desktop-, als auch Web- und Mobileanwendungen getestet werden, was auch im Lizenzerwerb unterschieden wird: Neben einer Einzelplatz-Lizenz der TestComplete-Plattform für 1067€ wird mindestens eines der zusätzlichen Module „Web“ (1067€), „Mobile“ (1391€) und „Desktop“ (1067€) benötigt. TestComplete besitzt, neben dem Arbeiten mit Capture & Replay, auch die Möglichkeit zum Ent-

wickeln von Tests in Python, VisualBasic, JavaScript, C#, C++ und Delphi. Zum Zeitpunkt dieser Arbeit ist das Tool einschließlich Support und Dokumentation nur in englischer Sprache erhältlich.

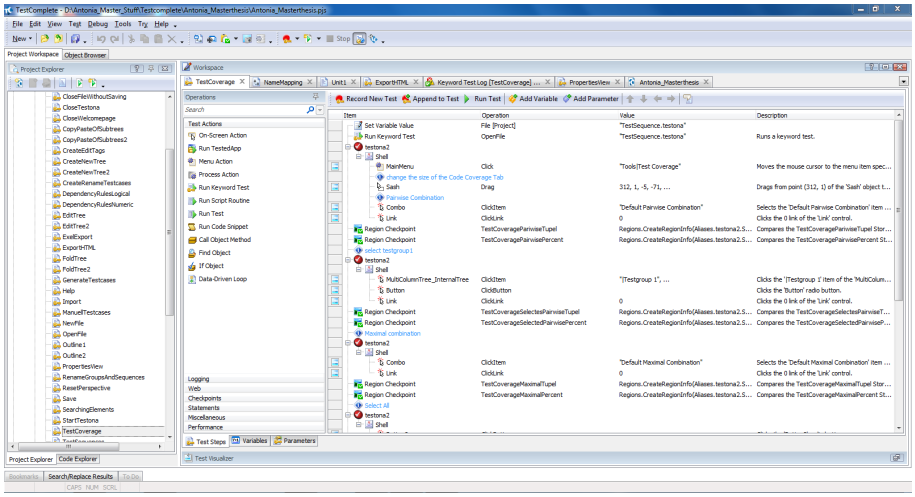

Abbildung 3.6: Grafische Oberfläche von TestComplete

Abbildung 3.6 zeigt das Hauptfenster von TestComplete: Auf der linken Seite befindet sich, ähnlich wie im zuvor vorgestellten QF-Test, eine Testsuite in Form eines hierarchischen Baums. Dieser vereint Kontrollstrukturen mit Testdaten und ermöglicht eine übersichtliche Verwaltung und einfache Bedienung. Neben den Testfällen existiert unten ein Verweis auf die Sammlung der für Checks benötigten Bilder sowie zum Namemapping. Auf der rechten Seite befindet sich der Workspace, in dem pro Tab ein Test, das Protokoll eines Testfalls oder andere Dateien, wie Skripte und Bilder dargestellt werden. Jeder Tab besitzt eine Toolbar, über die ein schneller Zugriff auf die Aufnahme- und Wiedergabefunktion möglich ist. Zusätzlich können über das links verankerte Menü einzelne Aktionen, wie beispielsweise Kommentare, Delays und Checkpoints eingebaut werden. TestComplete greift dabei über ein bei einer Aufnahme automatisch erstelltes oder im Objektbrowser manuell konfiguriertes Namemapping auf die verschiedenen GUI-Komponenten zu.

3.3.3 Eggplant

Eggplant Functional (Version 16.01) ist der für das Testen von grafischen Oberflächen geeignete Teil einer im Jahr 2013 veröffentlichten Testtoolreihe der Firma „Testplant“ [Egg16]. Das Capture & Replay-Tool ermöglicht das Entwickeln von GUI-Tests auf den Betriebssystemen Windows, Linux und Mac OS X, wobei im Gegensatz zu den vorherigen Testtools mindestens zwei Computer benötigt werden. Während eine öffentlich verfügbare Preisliste zum Zeitpunkt dieser Arbeit nicht zur Verfügung stand, konnte auf Nachfrage beim Hersteller der Preis einer Einzelplatz-

Lizenz von 6.499€ pro Jahr inklusive Softwareupdates und Support bei einer Mindestlaufzeit von 12 Monaten ermittelt werden.

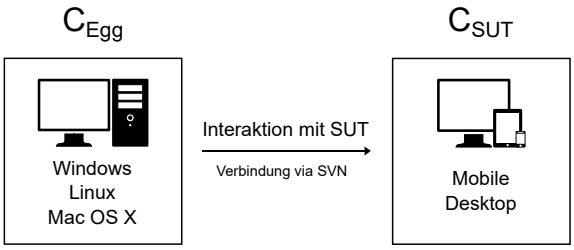

Abbildung 3.7: Zwei-System Model von Eggplant Functional [Egg16]

Abbildung 3.7 visualisiert das „Zwei-System-Modell“ von Eggplant, bei dem ein Computer C_{Egg} mit installiertem Eggplant und ein Computer C_{SUT} mit dem SUT benötigt werden. C_{Egg} verbindet sich über VPN mit C_{SUT} und ermöglicht so das Aufnehmen, Skripten und Wiedergeben von Tests. Dabei kann C_{Egg} stets mit nur einem C_{SUT} verbunden sein. Um Tests auf mehreren Systemen aufzunehmen kann C_{SUT} aber variiert werden.

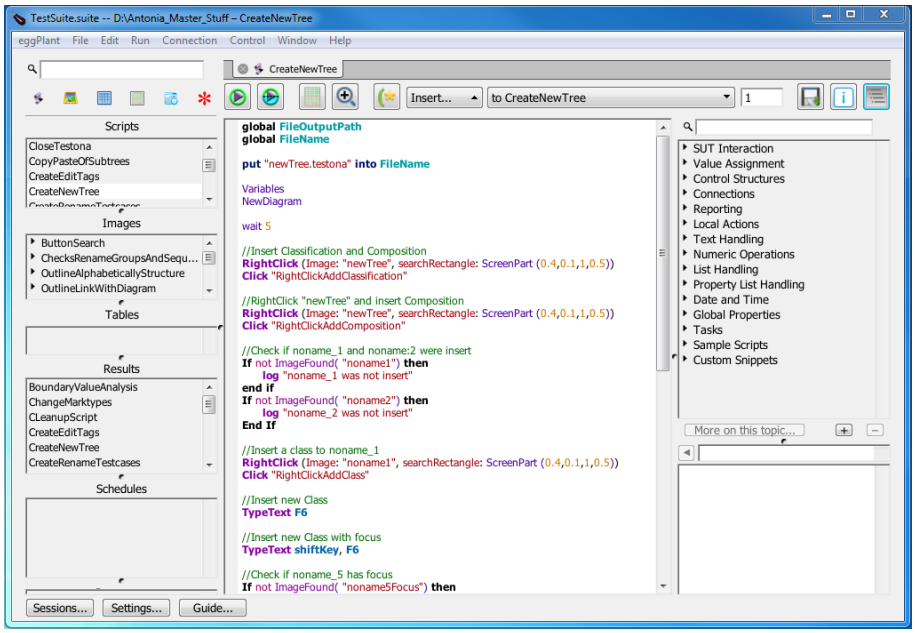

Abbildung 3.8: Grafische Oberfläche F_1 von Eggplant Functional

Die grafische Oberfläche von Eggplant teilt sich dementsprechend in mehrere Fenster auf. Abbildung 3.8 zeigt das Fenster F_1 zur Erstellung und Bearbeitung der Testsuite. F_1 gliedert sich grob in zwei Bestandteile: Auf der linken Seite sind unter anderem die Testscripte, Bilder und Tabellen der Testsuite dargestellt. Die Größe der Darstellung kann dabei verändert werden, so dass wichtigere Bestandteile besser erkennbar sind. Die Tests werden als Liste angezeigt, wobei die Übersicht der Grafiken auch Bäume enthalten kann, wenn mehrere Grafiken zu einer Gruppe

zusammengefasst wurden. Im Fenster auf der rechten Seite ist der Quellcode vom Testskript dargestellt, der wie in einem normalen Editor bearbeitet werden kann.

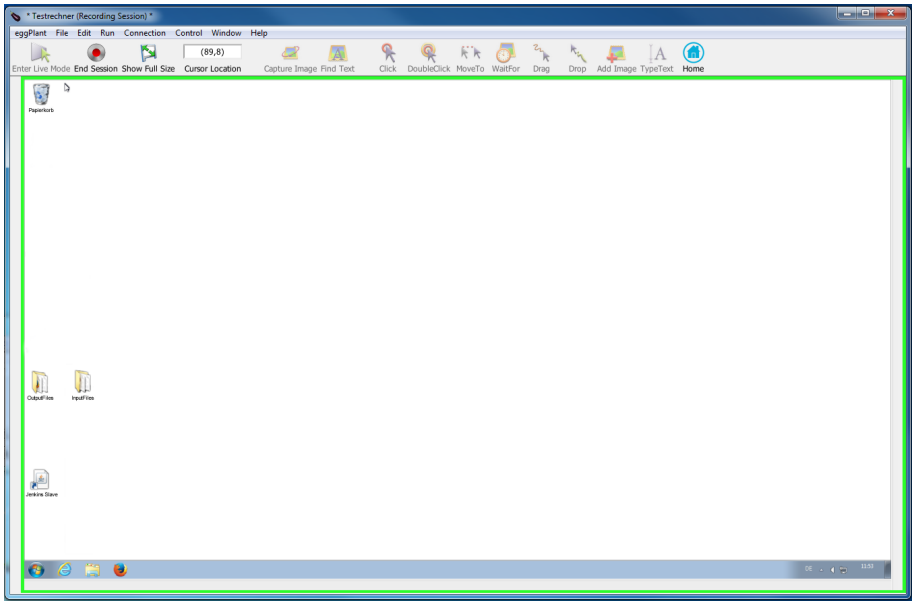

Abbildung 3.9: Grafische Oberfläche F_2 von Eggplant Functional

Das zweite Fenster F_2 zeigt die grafische Oberfläche des SUTs. Über die Toolbar kann ein Test aufgenommen und anschließend in F_1 bearbeitet werden. Abbildung 3.9 zeigt F_2 im Aufnahmemodus, weshalb das Fenster einen grünen Rahmen besitzt und einige Buttons in der Toolbar deaktiviert sind. Im Live-Modus kann mit dem Computer C_{SUT} interagiert oder einzelne Grafiken für Tests aufgenommen werden. Wird ein Test abgespielt, kann die Ausführung über F_2 beobachtet werden, während in einem dritten Fenster F_3 zeitgleich das Protokoll ausgegeben wird, um eine Erkennung eventuell auftretender Fehler zu ermöglichen.

3.4 Auswahl der Testszenarien

Im Rahmen dieser Arbeit habe ich 26 verschiedene Testszenarien entworfen, die ich in drei Gruppen einteile: Basisfunktionen, Testfallerstellung und -bearbeitung sowie weitere Funktionen. Die Testszenarien sind in deutscher Sprache beschrieben, haben jedoch englische Namen, da die meisten Testtools eine englische Sprache verwenden. Innerhalb der Testfälle in den Tools wird ebenfalls Englisch verwendet, so dass Komponenten, Kommentare, etc. einheitlich mit den Befehlen der Tools sind und es nicht zur Vermischung der Sprachen kommt.

Die Tests aus der Gruppe der **Basisfunktionen** testen die grundlegenden Funktionen von TESTONA, wobei die Erstellung und Bearbeitung von Bäumen innerhalb des Diagramms im Fokus steht. In den Tests spiegelt sich die häufige Verwendung der Toolbar, des Rechtsklick-Menüs, sowie der Interaktion im Tree wieder.

Die Gruppe der Basisfunktionen enthält folgende neun Tests, die im Anhang genauer beschrieben werden: CreateNewTree, EditTree, FoldTree, CopyPasteOfSubtrees, WriteProtectedFiles, PropertiesView, Autolayout, Outline1 und Outline2.

Die zweite Testgruppe der **Testfallerstellung und -bearbeitung** interagiert vorwiegend innerhalb der Testmatrix und dem TestcaseTree. Die Darstellung der Testgruppen und -sequenzen innerhalb des TestcaseTrees sorgt für ein häufiges Interagieren mit Baum- und Unterbaumknoten sowie den Elementen der Testmatrix. Diese Gruppe setzt sich aus den folgenden sechs Testszenarien zusammen: CreateRenameTestcases, ChangeMarktypes, GenerateTestcases, ManualTestcases, RenameGroupsAndSequences und TestSequences.

Die dritte Testgruppe **weitere Funktionen** beinhaltet diverse Funktionen, die meist über das Menü aufgerufen werden. Dazu zählen Funktionen, die innerhalb von TESTONA Elemente bearbeiten, aber auch das Importieren und Exportieren von Dateien. Häufig werden in den Tests zusätzliche Dialoge geöffnet, in denen mit GUI-Komponenten wie Dropdown-Menüs, Listenfeldeinträgen, Auswahlkästchen etc. interagiert wird. Diese Gruppe besteht aus den folgenden elf Testfällen: BoundaryValueAnalysis, CreateEditTags, VariantenManagementView, Help, SearchingElements, DependencyRulesLogical, DependencyRulesNumeric, ExportExcel, ExportHTML, Import und Testcoverage.

Eine detaillierte Auflistung der verwendeten GUI-Komponenten innerhalb der Tests ist in Tabelle 3.2 abgebildet. Die Komponenten sind von links nach rechts bezüglich ihrer Vorkommen in den Tests sortiert, was sich auch nahezu in der Häufigkeit der Klicks bzw. Interaktion mit den Elementen widerspiegelt. Dabei habe ich die Häufigkeit der Vorkommen sowie die Interaktion mit den GUI-Komponenten in den Tests für TESTONA 4.1 gemessen, so dass im Hinblick auf die Weiterentwicklung des SUTs minimale Unterschiede für spätere Versionen existieren können.

	Buttons	Menü	Menüleiste	Textfeld	Rechtsklick-Menü	Tree	Toolbar	Dropdown-Menü	Baumknoten	TestcaseTree	Unterbaumknoten	Properties	Listenfeldeintrag	Auswahlkästchen	Radiobutton	Tabs/Reiter	Treetable	Treeview	Tabelle	Schieberegler	Farbübersicht	Link	Combobox	Scrollbalken
CreateNewTree	1	2	2	1	4	9	4																	
EditTree	6	3	2	1	6	11	5					2								1	2			
FoldTree	1	2	2	1	5	8	1									1								
CopyPasteOfSubtrees	4	4	4	2	6	7	2																	
WriteProtectedFiles	4	5	5	2		4	1	1								1								
PropertiesView	5	4	4	3		2	3					6												
Autolayout	5+(15)	3	2	2		3						2								2				
Outline1	6	3	2	1	1	1												4						
Outline2	4	3	2	3	12		2											9						
CreateRenameTestcases	5	3	2	3	5	2				3		2					7							
ChangeMarktypes	6	4	4	1	3			2	1	1	2						10							
GenerateTestcases	16	6	6	3	2			1	2	4					2								3	
ManualTestcases	12	4	4	4		2			2							1								
RenameGroupsAndSequences	5	2	2	5	5				4	5	7													
TestSequences	2	2	2	1	5				8		3													
BoundaryValueAnalyses	10	4	4	8				1																
CreateEditTags	12	4	4	3		6	3						4	1										1
VariantManagementView	10	7	6	2	6	2	4	2	1		1		1			1					1			
Help	7	4	4	2																				
SearchingElements	6	3	3	5	7		2	2		1				5										
DependencyRulesLogical	14	4	4	3			2												3					
DependencyRulesNumeric	11	4	4	2		2	1	3		2					1		1		1					
ExportExcel	10	5	5	3				2	3		2		2	1										
ExportHTML	7	4	4	4				1			1		2	1										
Import	7	2	2	2					1	1	1		1		2									
Testcoverage	1	4	4	1			2	1		1		1		1	1		1					4		
Vorkommen in Tests	26	26	26	26	13	13	13	10	8	8	7	5	5	5	4	4	3	2	2	2	2	1	1	1
Klicks	117	95	89	68	67	59	32	16	22	18	17	13	10	9	6	4	19	13	4	3	3	4	3	1

Tabelle 3.2: Verwendete GUI-Komponenten innerhalb der Testszenarien

Die 26 verschiedenen Testfälle wurden vor ihrer Implementierung von mir spezifiziert. Dazu habe ich alle Klicks detailliert beschrieben, so dass in allen Tools die Tests in der gleichen Ausführung aufgezeichnet werden. Dies ist notwendig, da einige Funktionen in TESTONA auf verschiedene Arten aufgerufen werden können. So kann beispielsweise eine Klasse über einen Shortcut, das Rechtsklick-Menü oder die Toolbar in einem Baum hinzugefügt werden. Die Schritte in der Spezifikation beschreiben entweder Handlungen oder erwartete Zustände, die immer mit dem Symbol „⇒“ gekennzeichnet sind. An diesen Stellen habe ich jeweils, sofern dies möglich war, einen Check als Überprüfung in den Test eingefügt. Eine beispielhafte Spezifikation des Tests „CreateNewTree“ ist im Folgenden aufgeführt.

CreateNewTree

1. Procedure Neue Datei erstellen
2. Rechtsklick auf Root, Add classifications `noname_1` wählen
3. Rechtsklick auf Root, Add Composition `noname_2` auswählen
4. ⇒ Zwei Elemente wurden eingefügt
5. Rechtsklick auf `noname_1`, Add Class
6. mit F6 erneut eine Klasse einfügen
7. ⇒ Es wurden erneut zwei Elemente eingefügt
8. SHIFT+F6 drücken
9. ⇒ Es wurde eine Klasse hinzugefügt, die den Fokus hat
10. ENTF
11. ⇒ Die Klasse wurde wieder entfernt
12. `noname_2` auswählen und mit F7 eine Klassifikation und
13. mit F8 eine Komposition hinzufügen
14. „Create Class“ in Toolbar aktivieren und unterhalb von `noname_1` eine Klasse hinzufügen
15. ⇒ Es wurde eine neue Klasse `noname_7` hinzugefügt
16. „Create Classification“ auswählen und in den Baum klicken `noname_8`
17. „Set Parent“ in Toolbar aktivieren
18. `noname_7` als Kind von `noname_1` zuweisen, dazu zuerst auf `noname_7` und dann auf `noname_1` klicken
19. ⇒ `noname_1` ist mit `noname_7` verbunden
20. Rechtsklick auf `noname_8`, „Set Parent“ klicken und `noname_4` auswählen
21. In Toolbar „Arrange All“ auswählen
22. Erst Close File und dann Don‘t Save klicken

Alle anderen Spezifikationen der Tests befinden sich im Anhang. Im Folgenden ist aber trotzdem für einen besseren Überblick eine knappe Beschreibung für jeden der 26 Tests formuliert.

CreateNewTree: Im Baum werden über das Rechtsklick-Menü, die Toolbar und Shortcuts Elemente in einen neu erzeugten Baum eingefügt und entfernt.

EditTree: Ein fertiger Baum wird geöffnet und dessen Layout geändert. Dabei werden Knoten umbenannt, die Schrift verändert und Elemente eingefärbt.

FoldTree: In einem fertigen Baum werden Teilbäume gefaltet, in die Subtree-Ansicht gewechselt und dort der Baum bearbeitet. Das Falten und Entfalten wird über Doppelklicks oder das Rechtsklick-Menü realisiert.

CopyPasteOfSubtrees: Ein Teil des geöffneten Baumes wird kopiert und sowohl innerhalb des Baumes, als auch außerhalb eingefügt. Letzterer wird durch Drag & Drop hin und her bewegt und bekommt einen neuen Elternknoten zugewiesen.

WriteProtectedFiles: Es wird eine schreibgeschützte Datei geöffnet, der darin enthaltene Baum bearbeitet und anschließend der Versuch unternommen die Änderungen zu speichern. Es wird eine neue Datei erstellt und der gesamte Baum aus der schreibgeschützten Datei durch Copy & Paste eingefügt.

PropertiesView: In der PropertiesView werden Baumelemente umbenannt. Es wird an verschiedenen Stellen geprüft, ob in den Properties die richtigen Werte angezeigt werden.

Autolayout: Über die Properties wird die Darstellung der Bäume geändert. Sowohl die Anordnung als auch der Abstand zwischen den Baumelementen wird dabei geändert.

Outline1: Der geöffnete Baum wird über die Outline durch das Einfügen von Klassen und Klassifikationen bearbeitet. Des Weiteren werden die verschiedenen Darstellungsoptionen geprüft.

Outline2: In diesem Test werden über die Outline Baumelemente umbenannt, sowohl über Shortcuts als auch Rechtsklick-Menü-Optionen ausgeschnitten und an einer anderen Stelle im Baum eingefügt. Außerdem werden weitere Darstellungsoptionen der Outline geprüft.

CreateRenameTestcases: In einer Datei werden Testfälle erstellt, umbenannt und an den geforderten Stellen Markierungen gesetzt.

ChangeMarktypes: Neben dem Setzen von Markierungen erfolgt ein Ändern der Marktypes. In diesem Zusammenhang wird auch der Modus zwischenzeitlich von explizit auf implizit geändert.

GenerateTestcases: In diesem Test werden Testfälle automatisch generiert, wobei überwiegend fertige Regeln verwendet und in ihren Optionen angepasst werden.

ManualTestcases: Im Gegensatz zu GenerateTestcases werden eigene Regeln erstellt und damit Testfälle generiert.

RenameGroupsAndSequences: In diesem Test werden Testgruppen und Testsequenzen sowohl manuell als auch mit der Funktion Advanced Rename umbenannt.

TestSequences: Es wird überprüft, ob die Typen von Testsequenzen, Testgruppen und einzelnen Tests unter bestimmten Bedingungen geändert werden können.

BoundaryValueAnalysis: Die Funktion Boundary Value Analysis (kurz BVA) ermöglicht es, Teilbäume mit bestimmten Werten zu definieren. In diesem Test wird die Funktionalität des BVA geprüft.

CreateEditTags: Dieser Test erstellt und bearbeitet Tags, aktiviert diese bei bestimmen Knoten und prüft, ob diese an allen Stellen korrekt angezeigt werden.

VariantenManagementView: Es erfolgt der Wechsel von der Standardansicht in die zusätzliche Ansicht „Varianten Management View“. Wird die Sicht gewechselt, ändert sich dabei unter anderem der Inhalt der Toolbar. Es werden in diesem Test Varianten erstellt und bearbeitet, so dass sich auch die optische Darstellung von Baumelementen ändern kann.

Help: Der Test prüft verschiedene Aufrufmöglichkeiten der Hilfefunktion und führt in dieser eine Suche aus.

SearchingElements: Nach dem Öffnen der Suchfunktion wird durch Eingabe in ein Textfeld nach verschiedenen Wörtern gesucht. Es erfolgt zudem eine Veränderung der Sucheinstellungen durch ein ausschließliches Suchen nach Testfällen oder Baumelementen sowie dem Berücksichtigen von Namen, Beschreibungen oder Tags.

DependencyRulesLogical: In diesem Test wird der für die Erstellung logischer Abhängigkeitsregeln von Klassen und Klassifikationen zuständige Dialog getestet, sowie in ihm Regeln erstellt und bearbeitet.

DependencyRulesNumeric: Während der vorherige Test die logischen Abhängigkeitsregeln testet, fokussiert sich dieser Test auf die numerischen Abhängigkeitsregeln, die in einem anderen Unterdialog erstellt werden. Dabei werden unter anderem Elemente per Drag & Drop aus dem Baum in die Regel gezogen.

ExportExcel: Dieser Test prüft zunächst den Export für Bilder und anschließend den für Excel. Im letzten Fall werden zusätzlich einige Einstellungen im Dialog angepasst.

ExportHTML: In diesem Test werden die Einstellungen für den HTML-Export gesetzt und getestet. Anschließend wird die Suche innerhalb des Exports mit den Suchwörtern „i“ und „c“ ausgeführt.

Import: Es werden die verschiedenen Importmöglichkeiten geprüft und anschließend eine Excel-Datei importiert.

TestCoverage: Ein neuer Tab wird in den Properties geöffnet. In ihm werden verschiedene Regeln geprüft.

4 Durchführung

4.1 Erwartungen

In diesem Kapitel thematisiere ich die erwarteten Fehler beim Ausführen der Tests. Die formulierten Erwartungen beruhen dabei auf den vom TESTONA-Entwicklerteam bearbeiteten Tickets der verschiedenen Versionen von TESTONA und sind sowohl übersichtlich in Tabelle 4.1 dargestellt als auch im Folgenden näher beschrieben.

	Testona 4.1	Testona 4.3	Testona 4.5
CreateNewTree			
EditTree			
FoldTree			
CopyPasteOfSubtrees			
WriteProtectedFiles			
PropertiesView			
Autolayout			
Outline1			
Outline2			
CreateRenameTestcases			
ChangeMarktypes			ϟ
GenerateTestcases			ϟ
ManuelTestcases			
RenameGroupsAndSequences			ϟ
TestSequences			
BoundaryValueAnalyses			
CreateEditTags			
VariantManagementView			
Help			
SearchingElements		ϟ	ϟ
DependencyRulesLogical			ϟ
DependencyRulesNumeric			ϟ
ExportExcel			ϟ
ExportHTML		ϟ	ϟ
Import		ϟ	
Testcoverage			

Tabelle 4.1: Erwartete Fehlschläge (ϟ) der Testfälle

4.1.1 Erwartete Fehlschläge von 4.1 zu 4.3

Aufgrund der geringen Änderungen von TESTONA Version 4.1 zu Version 4.3 erwarte ich lediglich drei Fehlschläge in den Tests `SearchingElements`, `HTMLExport` und `Import`.

SearchingElements: Bei der Suche nach bestimmten Tags von Elementen ergab die Suche in TESTONA 4.1 keine Treffer, da diese noch nicht implementiert war. Aufgrund deren Implementation in Version 4.3 werden nun Ergebnisse gefunden, so dass mit einem Fehlschlag des Tests gerechnet werden muss.

ExportHTML: Der Export wurde in Version 4.3 um die Möglichkeit „Code Generation“ erweitert. Diese Option befindet sich nicht nur in der Liste der Exportmöglichkeiten und könnte dort zu einem Fehler führen, sondern liefert auch bei der Suche nach dem Buchstaben „c“ einen Treffer, der vorher nicht existierte.

Import: Auch der Import wurde in Version 4.3 um eine neue Option erweitert: Die Option „Testresult“ befindet sich nun am Ende des Baums im Importfenster und könnte bei der Auswahl eines Dateiformats zu einem Fehlschlag führen.

4.1.2 Erwartete Fehlschläge von 4.3 zu 4.5

Aufgrund der signifikanten Veränderungen von TESTONA 4.3 zu Version 4.5 erwarte ich in diesem Testschritt deutlich mehr Fehlschläge. Wie die von Tabelle 3.1 aus Kapitel 3.2 dargestellte Entwicklung von TESTONA zeigt, sind die Anzahl der bearbeiteten Tickets sowie die Änderungen in den LOC ungefähr doppelt so groß. Dies müsste sich in den erwarteten Fehlschlägen widerspiegeln, so dass ich mit den folgenden acht Fehlschlägen rechne.

ChangeMarktypes: Im impliziten Modus sind nun auch „Square Marktypes“ erlaubt. Daher ändern sich die Symbole in der Testmatrix und beim Wechsel vom expliziten zum impliziten Modus. In Version 4.3 wurden diese Symbole in Fragezeichen konvertiert, während sie nun erhalten bleiben. Zudem ist ein Dialog beim Wechsel der Modi entfallen.

GenerateTestcases: Der Algorithmus zum Generieren der „Default Minimal Combination“ hat sich verbessert, so dass weniger Testfälle generiert werden.

RenameGroupsAndSequences: Die Option „Advanced Renaming“ befand sich früher direkt im Rechtsklick-Menü, während sie nun unter dem Menüpunkt „Edit“ zu finden ist.

SearchingElements: Das Icon für die Suche ist in der Toolbar verschoben worden. Es befinden sich nun einige neue Buttons vor der Suche, was zu einem Fehlschlag führen könnte.

DependencyRulesLogical: Beim Erstellen einer neuen Regel gab es im ersten Schritt ein Fenster, das lediglich der Eingabe des Namens diente. Dieser Dialog ist nun entfallen und der Name kann direkt im Fenster für die Regel eingegeben werden.

DependencyRulesNumeric: Ebenso wie im vorherigen Testfall ist das Dialogfenster entfallen. Zudem wurde ein neues Hinweisfenster hinzugefügt, falls beim Generieren von Testfällen keine erzeugt wurden. Dieser Dialog muss an der entsprechenden Stelle zusätzlich geschlossen werden.

ExportExcel: Der Exportdialog hat sich in Version 4.5 im Hinblick auf seine Oberfläche signifikant geändert. So wurden beispielsweise frühere Listen zum Auswählen von Eigenschaften in Checkboxen umgewandelt.

ExportHTML: Der Export wurde durch die „Code Generation Specification“ erweitert. Während die Suche nach dem Buchstaben „c“ in Version 4.3 drei Ergebnisse lieferte, produziert die Suche in der neuen Version nun vier Ergebnisse.

4.2 Aufbau der Testumgebung

4.2.1 Entwicklungsumgebung

Damit alle Testtools die gleichen Voraussetzungen besitzen, habe ich das SUT TESTONA auf einem Computer C_{SUT} unter dem Betriebssystem Windows 7 Enterprise installiert. C_{SUT} war während des Bearbeitungszeitraumes dauerhaft an einen Monitor mit einer Auflösung von 1920x1080 angeschlossen. Veränderungen bezüglich der Auflösung o. Ä. wurden im gesamten Zeitraum nicht vorgenommen. Da QF-Test und TestComplete keinen zusätzlichen Computer zur Aufnahme der Tests benötigen, habe ich beide Tools direkt auf C_{SUT} installiert. Eggplant dagegen verwendet einen zweiten Rechner (vgl. Kapitel 3.3.3), weshalb das Tool von mir auf einem Computer C_1 mit denselben Rahmenbedingungen wie C_{SUT} installiert wurde, so dass TESTONA weiterhin auf C_{SUT} ausgeführt und getestet werden kann. Der Aufbau der Testumgebung ist in Abbildung 4.1 dargestellt

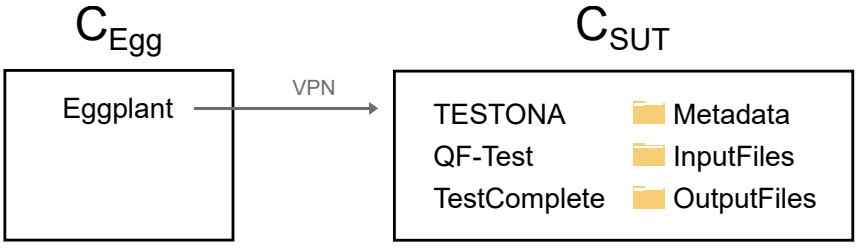

Abbildung 4.1: Aufbau der Testumgebung

Auf C_{SUT} befinden sich u. A. die drei Ordner „Metadata“, „InputFiles“ und „OutputFiles“, die ebenfalls in Abbildung 4.1 dargestellt sind. Im Metadata-Ordner werden Benutzereinstellungen von TESTONA gespeichert. Ändert ein Anwender das

Layout oder Einstellungen in TESTONA, so werden die Änderungen an dieser Stelle gespeichert, so dass nach einem Neustart von TESTONA Änderungen nicht in die Ausgangsform zurückgesetzt werden. Vor jedem Durchlauf der Testsuite muss dieser Ordner gelöscht werden, damit alle Tests unter den gleichen Bedingungen eines neu installierten Programms gestartet werden. Im Ordner „InputFiles“ befinden sich alle Dateien, die innerhalb der Testfälle benötigt werden. Dazu gehören unter anderem Dateien, die geöffnet oder importiert werden sollen. Im Ordner „OutputFiles“ werden alle innerhalb der Testfälle erzeugten Dateien gespeichert. Beispielsweise befinden sich nach einem Durchlauf der Testsuite die durch den Export entstandenen Dateien in diesem Ordner. Nach jedem Durchlauf muss der Ordner geleert werden, da es sonst zu ungewollten Dialogen oder Fehlermeldungen aufgrund bereits existierender Dateien kommt.

4.2.2 Automatisierte Ausführungsumgebung

Alle implementierten Testsuiten werden mit Hilfe des Tools Jenkins [Jen16] automatisiert ausgeführt. Jenkins (ehemals Hudson) ist ein in Java geschriebenes, plattformunabhängiges, frei verfügbares, webbasiertes Software-System zur kontinuierlichen Integration (engl. continuous integration system). Da alle drei Capture & Replay-Tools ein Plugin für Jenkins bereitstellen, bietet sich eine Automatisierung darüber an. Für die Ausführung habe ich die Testsuiten jeweils im SVN hochgeladen und auf C_{SUT} einen Jenkins-Client installiert.

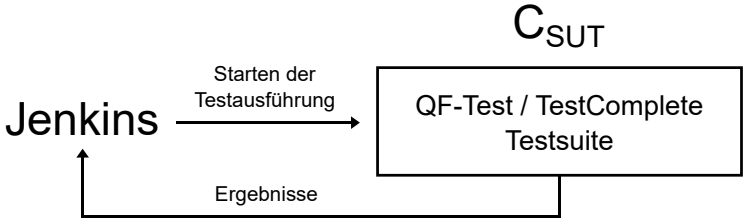

Abbildung 4.2: Automatisierte Testausführung für QF-Test & TestComplete

Abbildung 4.2 visualisiert die Testautomatisierung mit Hilfe von Jenkins für QF-Test und TestComplete. Über die Jenkins-Seite wird der zum Tool gehörende Job getriggert und daraufhin auf C_{SUT} die Testsuite ausgeführt. Nach einer erfolgreichen Ausführung werden die Testergebnisse an Jenkins weitergeleitet und sind auf der Seite vom jeweiligen Job einzusehen. Ein Jenkins-Job teilt sich dabei in die folgenden Schritte auf:

1. Update SVN Testsuite
2. Taskkill TESTONA
3. Delete OutputFiles
4. Delete Metadaten

5. Run TestSuite

Die Ausführung für Eggplant ähnelt der von TestComplete und QF-Test, ist jedoch aufgrund des Zwei-System-Modells nicht vollständig äquivalent. Damit TESTONA weiterhin auf C_{SUT} ausgeführt werden kann, muss Eggplant auf einem anderen Computer C_{Test} installiert werden (vgl. Abbildung 4.3). C_{Test} ist dabei ebenfalls ein Computer auf dem der Jenkins-Client unter Windows 7 Enterprise läuft.

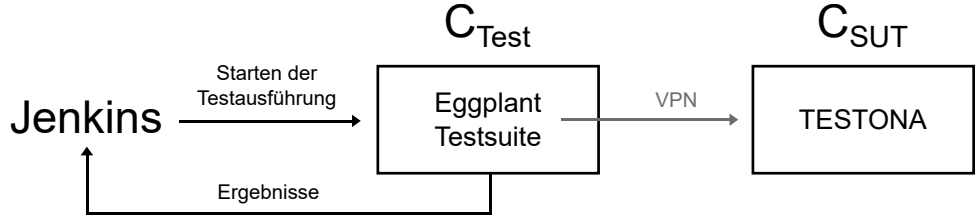

Abbildung 4.3: Automatisierte Testausführung für Eggplant

Der Haupt-Job von Jenkins ist dabei mit Ausnahme eines zusätzlich angelegten Prepare-Jobs äquivalent zu dem von QF-Test und TestComplete. Das Jenkins-Plugin von Eggplant konnte alte Jenkins-Ergebnisse nicht selbstständig entfernen, wodurch sich bei jedem Durchlauf die Liste der Ergebnisse vergrößerte und eine effektive Auswertung unmöglich wurde. Folglich mussten die alten Ergebnisse aus dem Ergebnisordner entfernt werden, was jedoch nicht im Haupt-Job möglich war. Daher habe ich bei Eggplant noch einen „Schritt 0“ bzw. „Prepare“ hinzugefügt, der alte Testergebnisse löscht und dann den Haupt-Job triggert.

4.3 Durchführung mit den einzelnen Tools

In diesem Kapitel wird das in Kapitel 3.1 beschriebene Vorgehen zur Ermittlung der für den Vergleich notwendigen Messwerte durchgeführt und die dabei getätigten Erfahrungen beschrieben. Die Reihenfolge der einzelnen Capture & Replay-Tools habe ich dabei bewusst geplant, um mein vorhandenes Vorwissen beim Umgang mit QF-Test auf ein vergleichbares Niveau zur Einarbeitung mit den beiden anderen Tools anzupassen.

4.3.1 Eggplant

Zu Beginn habe ich mich entschieden, bei der Entwicklung der Tests mit Eggplant zu beginnen. Einerseits hatte Eggplant einen zuvorkommenden Ansprechpartner, der mir kurzfristig eine Lizenz zur Verfügung stellen konnte, andererseits hatte ich noch keine Erfahrung mit einem Tool der dritten Generation und wollte nicht direkt mit einem mir bekannten Capture & Replay-Tool arbeiten, das dadurch einen Vorteil besitzen könnte.

4.3.1.1 Einarbeitung

Bei der Einarbeitung wurde ich von einem Ansprechpartner von Testplant unterstützt, der es mir auch ermöglichte, an einer privaten Vorstellung zum Umgang mit dem Capture & Replay-Tool teilzunehmen. Dabei konnte ich bereits erste Fragen stellen und Eindrücke von Eggplant erhalten, die mir die Einarbeitung im späteren Verlauf erleichterten. Dementsprechend schrieb ich den ersten Test `StartTestona` zügig und musste ihn lediglich im Hinblick auf eine Unabhängigkeit von der Optik des Start-Icons anpassen. Anschließend versuchte ich die Größe des TESTONA-Fensters für alle Tests zu fixieren, um eine konstante Größe aller GUI-Elemente zu gewährleisten. Da Eggplant eine entsprechende Einstellungsmöglichkeit jedoch nicht besitzt, entschied ich mich dazu, TESTONA immer im Vollbildmodus laufen zu lassen.

Ebenso entschied ich mich bereits nach wenigen Versuchen, die Record-Funktion von Eggplant nicht zu verwenden, sondern die Tests eigenhändig zu skripten. Aufgrund der eindeutig benannten Skriptbefehle von Eggplant ist der Quellcode eines Tests einfach zu verfassen und ebenso einfach zu lesen. Da Bilder zur Erkennung der Objekte in beiden Vorgehensweisen aufgenommen und bearbeitet werden müssen, führte meine Entscheidung zu keiner signifikanten Erhöhung des Arbeitsaufwands. Das Schreiben der Skripte ermöglichte es mir zudem, die Tests schrittweise zu implementieren und zu dokumentieren. Einzelne Aktionen konnten somit schnell von mir geprüft und ggf. direkt angepasst werden. Ein schrittweises Erarbeiten der Testschritte vereinfachte die Einarbeitung für mich deutlich, da ich zu jedem Zeitpunkt wusste, was ein Test ausführen sollte.

4.3.1.2 Arbeiten mit Grafiken

Für jede Interaktion mit dem SUT wird von Eggplant eine passende Abbildung benötigt, um die jeweilige Komponente auf dem Bildschirm zu suchen. Daher musste ich zu Beginn der Entwicklung für jeden Klick im SUT ein Bild aufnehmen und benennen, wobei selbst ein einfacher Klick im Menü oder in der Toolbar eine Grafik benötigt. Dabei gilt: Je mehr Elemente aus dem Menü bereits als Grafik existierten, desto schneller kann der Quellcode geschrieben werden, da die jeweils gesuchte Grafik lediglich aufgerufen wird. Dementsprechend benötigen neue Testfälle, die zuvor unbearbeitete Fenster öffnen bzw. prüfen, wieder neue Grafiken und somit mehr Entwicklungszeit für die Testerstellung. Zusammenfassend gilt daher beim Arbeiten in Eggplant: Je größer die vorhandene Grundmenge an Grafiken, desto schneller können Tests entwickelt werden.

Das Anpassen von Grafiken ist in Eggplant sehr intuitiv gestaltet. In einem zusätzlichen Fenster (vgl. Abbildung 4.4) kann ein Bild durch das Ändern des Rahmens angepasst werden. Der äußere Rahmen ist dabei die Originalgröße der Aufnahme,

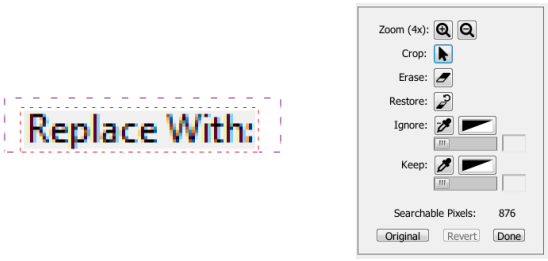

Abbildung 4.4: Bildbearbeitung in Eggplant

während der innere Rahmen den Ausschnitt repräsentiert, nach dem bei einem Aufruf gesucht wird. Je nach Kontext habe ich den Rahmen unterschiedlich groß gewählt und die Grafiken jeweils bestmöglich zurechtgeschnitten, da bereits minimale Änderungen in der Umgebung zu unterschiedlichen Ergebnissen führen können. Bei Elementen, die mit Zahlen enden, habe ich nach der Zahl immer einen geringen Abstand gelassen, um sicher zu gehen, dass dahinter nicht noch eine weitere Ziffer folgt.

Abbildung 4.5: Grafik mit Fokuspunk

Neben der genauen Größe wird zu jeder Grafik auch ein Fokuspunkt gespeichert, der sich zwar in einer anderen Datei befindet, aber ebenfalls im Tool direkt bearbeitet werden kann. Abbildung 4.5 zeigt eine Grafik mit einem Fokuspunkt, der jeweils angibt, wo beim Finden des Ausschnitts geklickt werden soll. Der Punkt muss nicht innerhalb der Grafik liegen, so dass bei Formularen ein Text gesucht und daneben in das Textfeld geklickt werden kann. Für den Fall, dass bei einer Grafik an unterschiedlichen Positionen geklickt werden soll, müssen zwei Grafiken mit unterschiedlichen Punkten erstellt werden.

Bei der Testerstellung entsteht dadurch schnell eine hohe Anzahl an Grafiken, für die der Tester den Überblick behalten muss. Für optisch verschiedene Grafiken mit ähnlicher Funktionalität kann dazu eine „Collection“ erstellt werden. Abbildung 4.6 zeigt beispielhaft die für den Button „Search“ erstellte Collection. Diese musste von mir erstellt werden, da der Button an einer Stelle einen Strich unter dem „S“ aufweist und somit nicht mehr das ebenfalls als Grafik gespeichertes, ursprüngliches Aussehen besitzt. Bei einem Aufruf von „ButtonSearch“ wird dann nach beiden möglichen Darstellungen des Buttons gesucht und auf die passende Grafik geklickt.

Um doppelte Grafiken zu vermeiden und einen besseren Überblick zu behalten, habe ich die folgenden Regeln für die Namensgebung von Grafiken aufgestellt:

- **Allgemein:** „Wo befindet sich das Element“ + „Name des Elements“

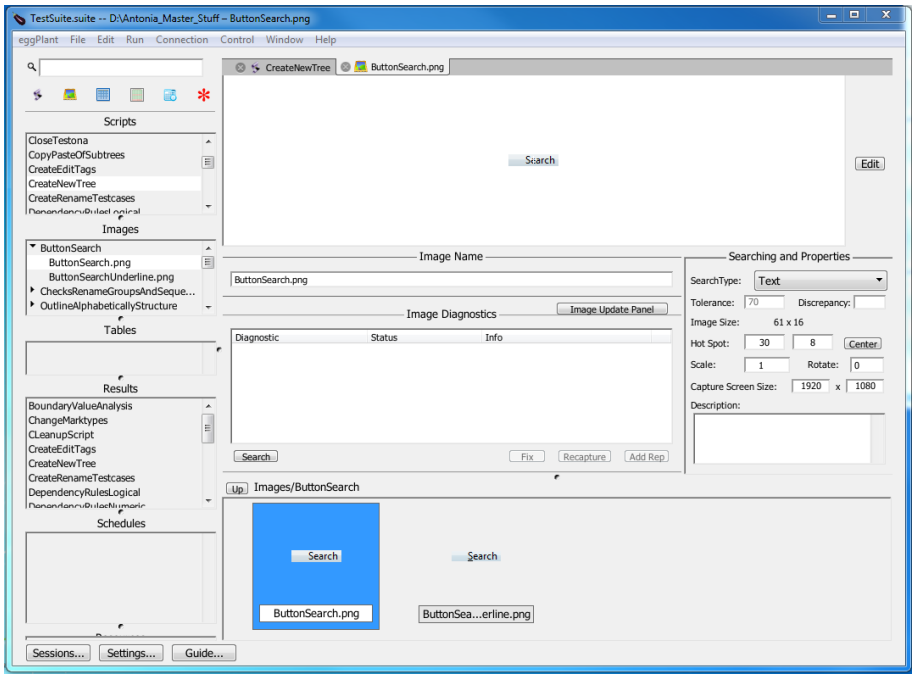

Abbildung 4.6: Grafiken in Eggplant

- **Buttons:** Buttons beginnen immer mit „Button“ + „Name des Buttons“, z.B.: „ButtonOk“, „ButtonSave“, ...
- **Baumelemente im Tree:** Die Standardnamen für Baumelemente wie „noname_1“ bis „noname_x“ bekommen den Namen „nonameZAHL“, außer für sie existiert ein spezifischer „Name“, dann heißen sie „nodeName“
- **Rechtsklick-Menü:** Elemente aus einem Rechtsklick-Menü beginnen immer mit „RightClick“ + „Name des Menüeintrags“, z.B.: „RightClickAddClass“, „RightClickEditRename“, ...
- **Toolbar:** Elemente aus der Toolbar beginnen immer mit „Toolbar“ + „Name des Elements“, z.B.: „ToolbarCreateClass“, „ToolbarArrangeAll“, ...

Für den Fall, dass einzelne Baumknoten in einem Test umbenannt werden mussten, habe ich fast immer denselben Namen gewählt, um unnötige Grafiken zu vermeiden.

4.3.1.3 Arbeiten mit Variablen

Wie in Kapitel 4.2.1 dargestellt, befinden sich Dateien, die innerhalb der Tests geöffnet werden, im Ordner „InputFiles“ und Dateien, die innerhalb der Tests entstehen und gespeichert werden, im Ordner „OutputFiles“. Um den Pfad zu den Ordnern nicht überall als festen String einzugeben, war es notwendig den Pfad in eine Variable auszulagern, um diese ggf nur an einer Stelle anpassen zu müssen.

```
global FileName
global FileOutputPath
global FileInputPath

put "C:\...\Desktop\OutputFiles\ " into FileOutputPath
put "C:\...\Desktop\InputFiles\ " into FileInputPath
```

Listing 4.1: Definition der Variablen in Eggplant

Der Code in Listing 4.1 zeigt das dazu von mir programmierte Skript `Variables`, in dem der Pfad für den Input- und Outputpfad in die Variablen geschrieben wird. Zu Beginn jedes Tests muss daher die benötigte Variable genannt und das Skript `Variables` ausgeführt werden.

```
global FileInputPath
global FileName

put "TestGroup.testona" into FileName

Variables
FileOpen
...
```

Listing 4.2: „Datei öffnen“ in Eggplant

Der Quellcode in Listing 4.2 zeigt den Beginn eines Testfalls, in dem die Datei „TestGroup.testona“ geöffnet werden soll. Dabei wird zunächst der Name der Datei in die Variable „FileName“ geschrieben, so dass er im `FileOpen`-Skript verwendet werden kann. Das Skript zum Öffnen einer Datei ist in Listing 4.3 dargestellt.

```
global FileName
global FileInputPath

Click "MenuFile"
Click (Image:"MenuFileOpen", WaitFor:40)

put FileInputPath into File
put FileName after File

TypeText File
Put EveryImageLocation("ButtonFileOpen") into
   FoundImagesButtonFileOpen
Click item 2 of FoundImagesButtonFileOpen

wait 3
```

Listing 4.3: Test „FileOpen“ in Eggplant

Wie man im Quellcode von Listing 4.3 erkennen kann, wird zunächst im Menü auf `File` und anschließend auf `File Open` geklickt. Da das Laden des Fensters

häufig mehr Zeit benötigt, habe ich ein `Wait` mit einer maximalen Wartezeit von 40 Sekunden integriert. Anschließend wird der Pfad für die neue Datei aus den Variablen `FileInputPath` und `FileName` zusammengesetzt und in die Variable `File` gespeichert. Da `File` lediglich in diesem Test verwendet wird, muss die Variable nicht als global deklariert werden und benötigt keine zusätzliche Spezifikation. Der fertige String wird im Fenster eingegeben und im Anschluss soll auf den `File Open`-Button geklickt werden. Da auf dem Button lediglich `Open` steht und sich der gleiche String mit demselben Hintergrund auch im Titel des Dialogs befindet, habe ich in Zeile 11 alle gefundenen Möglichkeiten für das Abbild von `ButtonFileOpen` in die Variable `FoundImagesButtonFileOpen` geschrieben. Aufgrund der konstanten Arbeitsweise des Suchalgorithmus von oben-links nach unten-rechts wird der Titel zuerst gefunden und anschließend der Button. Demnach wird im Test auf das zweite gefundene Objekt geklickt. Damit die Datei ausreichend Zeit zum Öffnen erhält, habe ich am Ende noch ein `Wait` von drei Sekunden hinzugefügt.

4.3.1.4 Testskripte

Wie bereits in Kapitel 4.3.1.1 beschrieben, begann ich die Testentwicklung mit dem simplen Testfall `StartTestona`, der zunächst nur aus einer Zeile bestand (vgl. Listing 4.4).

```
Click "TestonaIcon"
```

Listing 4.4: Erste Version von „Start Testona“

Auch wenn Eggplant die Möglichkeit besitzt Bildausschnitte auf dem gesamten Bildschirm bzw. Desktop zu finden, entschied ich mich dazu, eine robustere Methode zum Start von TESTONA zu entwickeln, so dass das SUT, auch wenn das Icon verdeckt sein sollte, gestartet werden kann. Die von mir entwickelte Lösung beruht dabei auf dem Start von TESTONA über das Suchfeld des Windows-Menüs und ist in Listing 4.5 dargestellt.

```
//Start Testona
TypeText windowsKey
TypeText "Testona"
TypeText enterKey

wait 20

//If testona is not maximised, click the maximise button
If ImageFound(ButtonMaximise) then
  Click (Image:"ButtonMaximise", WaitFor:10)
End if

//If there is the welcomepage, close it
If ImageFound(TabWelcomePageClose) then
```

```
15    Click (Image:"TabWelcomePageClose", WaitFor:10)
  End if

  ResetPerspective
```

Listing 4.5: Finale Version von „Start Testona“

Der in Listing 4.5 abgebildete Programmcode enthält dabei neben dem Start von TESTONA auch das Maximieren des Fensters sowie das Zurücksetzen der Perspektive. Die Zeilen 9-11 prüfen dazu, ob es einen sichtbaren Maximieren-Button gibt und führen einen Klick auf diesen aus, sollte das Fenster noch nicht maximiert sein. In den Zeilen 14-16 wird geprüft, ob die sogenannte „Welcome-Page“ sichtbar ist: Bei einer Neuinstallation von TESTONA sowie dem Löschen der Metadaten erscheint diese nach dem Start und verschwindet erst durch das Schließen des Tabs. Ohne das Schließen der „Welcome-Page“ wäre ein weiteres Arbeiten mit TESTONA nicht möglich. Anschließend erfolgt das Zurücksetzen der Perspektive, was ich allerdings in ein zusätzliches Testskript ausgelagert habe. Zwischen den einzelnen Befehlen habe ich stets optionale Wartezeiten eingebaut, um einen Fehlschlag durch zu langsames Laden zu verhindern.

Der in Kapitel 3.4 „Auswahl der Testszenarien“ näher vorgestellte Testfall `CreateNewTree` wird im Folgenden als Beispiel für die Erstellung aller 26 Testfälle in Eggplant näher beschrieben. Der dazu von mir erstellte Quellcode für diesen Testfall ist in Listing 4.6 dargestellt.

```
  global FileOutputPath
2 global FileName

  put "newTree.testona" into FileName
  Variables
6 NewDiagram
  wait 5

  //Insert Classification to the root "newTree"
10 RightClick (Image: "newTree", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
  Click "RightClickAddClassification"

  //RightClick "newTree" and insert Composition
14 RightClick (Image: "newTree", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
  Click "RightClickAddComposition"

  //Check if noname_1 and noname_2 were insert
18 If not ImageFound( "noname1") then
    log "noname_1 was not insert"
  end if
  If not ImageFound( "noname2") then
22   log "noname_2 was not insert"
  End If

  //Insert a class to noname_1
```

```
26 RightClick (Image: "noname1", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
   Click "RightClickAddClass"
   //Insert new Class
   TypeText F6
30 //Insert new Class with focus
   TypeText shiftKey, F6

   //Check if noname_5 has focus
34 If not ImageFound( "noname5Focus") then
     log "noname_5 has no focus"
   End If

38 //delete the node
   TypeText deleteKey
   Click (Image: "noname2", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
   //insert classification
42 TypeText F7
   //insert composition
   TypeText F8

46 Click "ToolBarCreateClass"
   Click(Image: "Noname3Below", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
   Click "ToolbarCreateClassification"
   Click (Image: "noname4Below", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
50 Click "ToolbarSetParent"
   Click (Image: "noname7", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
   Click (Image: "noname1", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
   Click (Image: "noname8", searchRectangle: ScreenPart (0.4,0.1,1,0.5))
54 Click (Image: "noname4", searchRectangle: ScreenPart (0.4,0.1,1,0.5))

   Click "ToolbarMouseClick"
   Click "ToolbarArrangeAll"
58 CloseFileWithoutSaving
```

Listing 4.6: Testfall „CreateNewTree“

Die Zeilen 1-7 im von Listing 4.6 dargestellten Quellcode des Tests `CreateNewTree` deklarieren die Werte der Variablen und erstellen eine neue Datei mit dem in Zeile 4 definierten Namen. In den Zeilen 10 und 11 wird zunächst das Rechtsklick-Menü vom Knoten `newTree` geöffnet und dort der Menüeintrag `Add Classification` gewählt. Beim ersten Implementieren des Tests bestand die Zeile 10 noch aus den in Listing 4.7 dargestellten Zeilen.

```
Put EveryImageLocation("newTree") into FoundImagesNewTree
2 Click item 2 of FoundImagesNewTree
```

Listing 4.7: Alte Variante zum Finden von „`newTree`“

Bei einer Videokonferenz mit einem Entwickler von Eggplant erfuhr ich vom Befehl `searchRectangle: ScreenPart` (x_1,y_1,x_2,y_2) mit dem ein bestimmter, prozentualer Teil des Bildschirms angegeben werden kann, so dass der Suchbereich einschränkt wird. Da in diesen Zeilen innerhalb des Testbaumes geklickt werden

sollte, genügte es die Suche auf diesen Bereich zu beschränken, wodurch diese zwar nicht erheblich schneller wurde, aber einen eventuellen Klick auf ein zweites Element verhinderte.

Die folgenden Zeilen 18-20 prüfen, ob die Klassifikation aus den Zeilen 10-11 eingefügt wurde. Da die Namen aufgrund der von mir aufgestellten Namensgebung für Grafiken (vgl. Kapitel 4.3.1.2) für neue Baumelemente durchgängig nummeriert werden, muss lediglich eine Suche nach einem Abbild des aufgenommenen Bildes `noname1` erfolgen. Wird das Bild nicht gefunden, so wird die Fehlermeldung aus Zeile 19 in die Log-Datei geschrieben. Der in Zeile 29 verwendete Befehl `TypeText` kann sowohl zum Schreiben von Text, als auch für Tastendrücke verwendet werden. `TypeText F6` bewirkt einen einfachen Tastendruck der Taste F6. Soll die Taste gedrückt bleiben und erst später losgelassen werden, so verwendet man die Befehle `KeyDown` und `KeyUp`. In meinen Tests habe ich diese Variante außer beim Markieren von Elementen, so selten wie möglich verwendet, da ein Vergessen von `KeyUp` die Testausführung blockieren kann, indem es beispielsweise durch eine dauerhafte gedrückte `SHIFT`-Taste nicht mehr möglich ist Text zu verfassen.

4.3.1.5 MainSuite

Neben den einzelnen Testskripten war für eine automatisierte Ausführung mit Hilfe von Jenkins eine `MainSuite` notwendig. Der Quelltext dieses alle 26 Testfälle ausführenden Skripts ist in Listing 4.8 dargestellt. Dabei plante ich zunächst die Testfälle in die unter Kapitel 3.4 genannten Gruppen „Basisfunktionen“, „Testfallerstellung und – bearbeitung“ und „weitere Funktionen“ einzuteilen und nacheinander mit jeweils einem `StartTestona` und `CloseTestona` zwischendrin auszuführen. Nach Problemen bei deren anschließender Ausführung mit Jenkins und darauffolgender Rücksprache mit dem Support von Eggplant stellte ich den Quellcode unter Beibehaltung der Gruppen um, so dass bei der schrittweisen Ausführung der Testfälle mögliche Fehlschläge in die Variable `MyResult` geschrieben werden und anschließend das `CleanUp`-Skript ausgeführt wird.

```
repeat with each item MyScript of ("Variables","StartTestona","
    CreateNewTree", "EditTree","FoldTree","CopyPasteOfSubtrees","
    WriteProtectedFiles","PropertiesView", "Autolayout","Outline1
    ", "Outline2", "CloseTestona")
  runwithnewresults MyScript
  put the status of the result into MyResult
  if MyResult is "Failure"
    CleanupScript
  end if
end repeat
```

```
repeat with each item MyScript of ("StartTestona","
    CreateRenameTestcases", "ChangeMarktypes","GenerateTestcases"
    ,"ManualTestcases","RenameGroupsAndSequences","TestSequences"
    , "CloseTestona")
  runwithnewresults MyScript
  put the status of the result into MyResult
  if MyResult is "Failure"
    CleanupScript
  end if
end repeat

repeat with each item MyScript of ("StartTestona","
    BoundaryValueAnalysis", "CreateEditTags","
    VariantenManagementView","Help","SearchingElements","
    DependencyRulesLogical", "DependencyRulesNumeric","
    ExportExcel", "ExportHtml","ImportFiles", "TestCoverage", "
    CloseTestona")
  runwithnewresults MyScript
  put the status of the result into MyResult
  if MyResult is "Failure"
    CleanupScript
  end if
end repeat
```

Listing 4.8: MainSuite

Das Listing 4.9 zeigt den Quellcode des `CleanUp`-Skripts. In diesem prüfe ich zunächst ob die Grafik `ButtonCloseActive`, ein aktiver `Schließen`-Button, für eine sichtbare Fehlermeldung erkennbar ist. Soll beispielsweise eine Datei geschlossen werden und es tritt ein Fehler auf, so befindet sich ein neues Fenster im Vordergrund des SUTs und blockiert das Schließen der Datei. Beim Eintreten dieses fehlerhaften Zustands wird die Fehlermeldung in Zeile 2 beendet, die Datei mit Hilfe von `CloseFileWithoutSaving` geschlossen und alles in einem Log protokolliert.

```
1 if (ImageFound("ButtonCloseActive"))
    Click "ButtonCloseActive"
    CloseFileWithoutSaving
  end if
5 log "Cleaning up the SUT"
```

Listing 4.9: `CleanUp`-Script

Mit Hilfe des `CleanUp`-Skripts und weiteren Detailverbesserungen erfolgt ein unterbrechungsfreier Durchlauf der MainSuite trotz evtl. fehlgeschlagener Tests. Anschließend sind fehlgeschlagene Testfälle bei einem Durchlauf mit Jenkins durch deren Protokollierung detailliert nachvollziehbar.

4.3.1.6 Entwicklungszeit und −probleme

Zu Beginn der Testfallerstellung mit Eggplant benötigte ich aufgrund der notwendigen Grafiken viel Zeit für die Erstellung der Tests. Da die Tests häufig in verschiedenen Fenstern arbeiteten, waren in jedem Testfall neue Grafiken zu erstellen, so dass lediglich Buttons, Menüeinträge oder Baumelemente von mir wiederverwendet werden konnten. Das schrittweise Erstellen der Testfälle hatte dabei allerdings den Vorteil, dass ich die volle Kontrolle über den Quellcode besaß und jeweils die Funktionalität überprüfen konnte, um mögliche Fehler direkt zu beheben. Die Spalte „TESTONA 4.1“ in Tabelle 4.2 zeigt die benötigte Entwicklungszeit für alle Testfälle mit Eggplant. Die für die Implementierung der 26 Testfälle benötigte Entwicklungszeit schwankt dabei zwischen 30 und 240 Minuten. Die Zeiten neben den jeweils neuen Elementen sind aber auch von der Größe der Testspezifikation und dem damit verbundenen Aufwand abhängig.

Bei der Implementierung der einzelnen Testfälle wurde ich immer wieder vor Herausforderungen gestellt. So bemerkte ich bei der Entwicklung des Testfalls `CreateNewTree`, dass Eggplant bei der Suche nach einer Grafik einen auszuführenden Klick stets auf das erste gefundene Vorkommen und damit nicht zwingend auf die korrekte Grafik anwendet. Um dieses Problem zu beheben, verwendete ich die in Kapitel 4.3.1.4 vorgestellte Lösung der Zwischenspeicherung aller gleichartigen Grafiken mit anschließender Auswahl des korrekten Vorkommens. Diese Problembehebung habe ich dann im Verlauf der Implementierung nochmals durch die Verwendung einer Funktion zur konkreten Auswahl eines Bildschirmbereichs optimiert. Während der Entwicklung des Testfalls `EditTree` entdeckte ich die Möglichkeit, Eggplant mit nur minimal längerem Zeitaufwand nach einem spezifischen Text suchen zu lassen. Dies ermöglichte es mir, nur einmal vorkommende Wörter als String zu suchen, so dass die von mir erstellte Kollektion an Grafiken zum Testen nicht unnötig gefüllt und damit unübersichtlicher wurde. Bei der Implementierung des Testfalls `FoldTree` stellte ich fest, dass unter Windows 7 pulsierende Buttons existieren, die durch ihre sich ständig verändernde Optik nicht korrekt von Eggplant gefunden werden. Eine Recherche im Handbuch von Eggplant löste das Problem in Form einer Einstellung, so dass Eggplant den Hintergrund der Grafik ausblendete und nur mit der Schrift als Vergleichsmerkmal arbeitete.

Der mit 240 Minuten am meisten Entwicklungszeit benötigende Test `ChangeMarktypes` stellte für mich eine besondere Herausforderung dar. Der enorme Zeitaufwand für die Implementierung entstand dabei nicht beim Erstellen der Grafiken oder Schreiben des Skripts, sondern beim Entwickeln einer effektiven Strategie zum Auswählen des passenden Bildausschnitts in der Testmatrix. Im `ChangeMarktypes`-Test muss in der Testmatrix an bestimmte Stellen geklickt werden, um die Testfälle zu spezifizieren. Da für Eggplant die Testmatrix nur als ganze Grafik sichtbar ist,

Eggplant	TESTONA 4.1	TESTONA 4.3	TESTONA 4.5
CreateNewTree	180 min		
EditTree	180 min		10 min
FoldTree	120 min		
CopyPasteOfSubtrees	120 min		
WriteProtectedFiles	60 min		
PropertiesView	120 min		
Autolayout	150 min		
Outline1	60 min		
Outline2	90 min		
CreateRenameTestcases	120 min		
ChangeMarktypes	240 min		↯ 60 min
GenerateTestcases	120 min		↯ 25 min
ManualTestcases	120 min		
RenameGroupsAndSequences	120 min		↯ 5 min
TestSequences	90 min		10 min
BoundaryValueAnalyses	60 min		
CreateEditTags	90 min		
VariantManagementView	90 min		
Help	30 min		5 min
SearchingElements	120 min	↯ 5 min	↯ 20 min
DependencyRulesLogical	60 min		↯ 20 min
DependencyRulesNumeric	60 min		↯ 10 min
ExportExcel	60 min	10 min	↯ 5 min
ExportHTML	60 min	↯ 15 min	↯
Import	60 min	↯	10 min
Testcoverage	60 min		
StartTestona	20 min		
CloseFileWithoutSaving	20 min		
CloseFileWithSaving	10 min		
OpenFile	20 min		
CloseFile	10 min		
SaveAs	20 min		
VariablesScript	10 min		
Summe	2750 min 0 min	3 30 min	10 180 min

Tabelle 4.2: Entwicklungs- & Reparaturzeit für Eggplant in Minuten

kann nicht direkt auf die einzelnen Elemente in der Matrix zugegriffen werden. Eine Problemlösung, beispielsweise über den Zugriff auf Koordinaten, wäre bei Veränderungen in der Testmatrix zu fehleranfällig gewesen. Beim Versuch den Bildschirm nach einem Vorkommen von Fragezeichen, Kreisen oder den Schnittpunkten der Zeilen und Spalten abzusuchen, entstanden die in Abbildung 4.7 dargestellten Bildausschnitte als einzelne Grafiken, mit denen dann ein Zugriff innerhalb der Testmatrix möglich wurde.

Diese Vorgehensweise ermöglichte einen deutlich robusteren Testfall. Ein Klick auf einen Schnittpunkt benötigte das genaue Auszählen aller Vorkommen und die Ermittlung der passenden Position. Listing 4.10 zeigt einen beispielhaften Ausschnitt

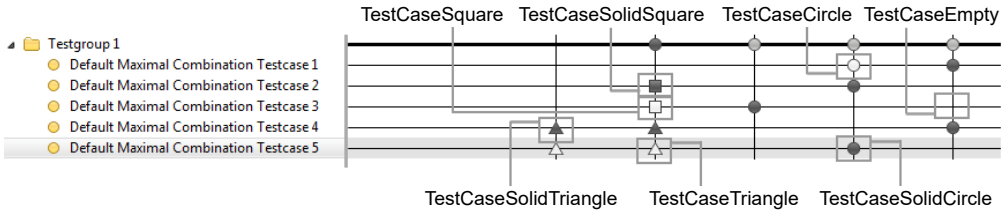

Abbildung 4.7: Grafische Elemente der Testmatrix

aus dem Testfall. In Zeile 3 werden alle Schnittpunkte ohne Symbol gezählt und anschließend auf deren viertes Vorkommen geklickt. Nach dem Einfügen eines bestimmten Marktypes ändert sich die Anzahl der Vorkommen von `TestCaseEmpty`, weshalb sich das vierte Vorkommen nun an einer anderen Stelle befindet.

```
...
//Set mark in testcase 2 at noname_3
Put EveryImageLocation("TestCaseEmpty") into
    FoundImagesTestCaseEmpty
Click  item 4 of FoundImagesTestCaseEmpty
...
```

Listing 4.10: Ausschnitt aus dem Testfall „ChangeMarktypes“

Die größte Herausforderung stellte beim Umgang mit Eggplant das zur Prüfung des Zustands eines Elements notwendige Umdenken dar. So musste ich beispielsweise um zu testen, ob ein Element nicht anklickbar ist, den passenden Bildausschnitt mit einem grau hinterlegten Menüeintrag finden. Tabelle 4.2 belegt die dabei entstandene Robustheit der Tests, da keiner von ihnen für TESTONA 4.1 repariert werden musste.

4.3.1.7 Fehlschläge von TESTONA 4.1 zu 4.3

Bei der Umstellung der Version von TESTONA 4.1 auf 4.3 sind drei Testfälle fehlgeschlagen. Dabei sind zwei der erwarteten Fehlschläge eingetreten, ein Test hat trotz erwartetem Fehlschlag funktioniert und ein Test ist zusätzlich fehlgeschlagen. **SearchingElements:** Die Suche nach Elementen ist fehlgeschlagen, da die Suche nach Attributes-Tags in TESTONA 4.1 nicht implementiert war. Im von Listing 4.11 dargestellten Quellcode musste die erste Zeile für die Anzahl der gefundenen Tags angepasst werden, da die korrekte Anzahl noch nicht bestimmt war.

```
If not (number of items in FoundImagesAttributes equal to 0)
    then
  log "SearchingElements: There are  Attributes Tags shown in
      the search tab."
end if
```

Listing 4.11: Ausschnitt aus dem Testfall „SearchingElements“

ExportExcel: Bei diesem Test wurde von mir kein Fehlschlag erwartet, dennoch ist die Eingabe des Pfads fehlgeschlagen, da das Textfeld ohne entsprechende Anmerkung im Ticket nach links gerutscht und somit der Fokuspunkt nicht mehr im Textfeld war. Der Fehler wurde von mir durch eine Anpassung des Fokuspunkts behoben.

ExportHTML: Obwohl ich für diesen Test einen Fehlschlag erwartet hatte, schlug dieser entgegen meiner Vermutung aus denselben Gründen wie der Test `ExportExcel` fehl. Der Abstand vom Text zum Textfeld für den Pfad bzw. Dateinamen hatte sich geändert und musste aufgrund eines signifikanten Abstands durch das Setzen eines Fokuspunkts in zwei Grafiken von mir behoben werden.

Zusammenfassend ist von drei erwarteten Fehlschlägen nur einer aufgrund der von mir vermuteten Ursache eingetreten. Der Fehlschlag im Test `SearchingElements` war nicht vermeidbar, wohingegen die beiden anderen Fehlschläge durch anders gesetzte Fokuspunkte vermeidbar waren und demnach in die Kategorie Entwicklungsfehler fallen. Mit zwei Änderungen konnte Eggplant hingegen problemlos umgehen und ist dementsprechend nicht fehlgeschlagen. Die Reparatur der drei fehlgeschlagenen Tests dauerte insgesamt 30 Minuten, von denen jedoch mehr als die Hälfte für die eigentliche Fehlersuche verwendet wurde.

4.3.1.8 Fehlschläge von TESTONA 4.3 zu 4.5

Die Umstellung der Version von TESTONA 4.3 auf 4.5 führte zu insgesamt zehn fehlgeschlagenen Tests, von denen drei Fehlschläge von mir zuvor nicht erwartet wurden.

ChangeMarktypes: Grund für den von mir erwarteten Fehlschlag war ein fehlender Dialog, so dass für die Reparatur ein damit verbundener Klick von mir entfernt werden musste. Zudem sind in TESTONA 4.5 beim Wechsel vom expliziten zum impliziten Modus nun `Square Marks` erlaubt, so dass aufgrund der in Kapitel 4.3.1.6 beschriebenen Vorgehensweise die darauffolgenden Vergleiche und Klicks von mir angepasst werden mussten.

GenerateTestcases: Wie von mir erwartet führte der neue Algorithmus der `Default Minimal Combination` zum Fehlschlag des Tests. Da weniger Testfälle generiert wurden, musste die Abfrage der Anzahl der Testfälle angepasst werden. Der neue Algorithmus setzt auch neue Default-Werte in den Generator, so dass auch dieser Vergleich durch ein neues Bild angepasst werden musste. Die dritte Fehlerursache war der neue Dialog für nicht generierte Testfälle, der durch das Einfügen der Zeile `Click "ButtonOk"` von mir behoben werden konnte.

RenameGroupsAndSequences: Da sich der Menüpunkt `Advanced Rename` nicht mehr direkt im Rechtsklick-Menü befindet, sondern in den Unterpunkt Edit verscho-

ben wurde, musste an vier Stellen vor dem Klick noch die Zeile eingefügt werden: `Click (Text:"Edit", searchRectangle: ScreenPart (0.2, 0.4,0.7,0.9))`.

TestSequences: Der Grund für diesen unerwartet fehlgeschlagenen Test war das in TESTONA 4.5 hinzugefügte zusätzliche Anzeigen des Namens eines selektierten Elements in der Statusleiste, wodurch ein Testfall öfter gefunden wurde. Mit einer einfachen Änderung in einer Zeile des Quelltexts konnte ich den Fehler beheben.

Help: Während ich die inhaltliche Überarbeitung der Hilfe berücksichtigte, entstand der Fehler durch mehrere unerwartete Treffer für das Suchwort „Shortcuts“ innerhalb des Tests. Ursprünglich hatte ich nur zwei Ergebnisse erwartet, mit den neuen Kapiteln waren es jedoch 31, so dass lediglich der zu prüfende Text in der Bedingung angepasst werden musste.

SearchingElements: Einer der Hauptgründe für den Fehlschlag des Tests war die verbesserte Suche nach Attributen, so dass ebenfalls lediglich die Bedingung angepasst werden musste. Zudem musste ich den Suchradius beschränken, da ein bereits ausgewählter Testfall bei einem Check ebenfalls gefunden wurde.

DependencyRulesLogical: Ein Fehlschlag von diesem Test war nicht vermeidbar, da der Dialog zum Benennen der Regel weggefallen ist. Neben einer Erweiterung des Quelltexts musste ich eine neue Grafik erstellen.

DependencyRulesNumeric: Beide Ursachen für den Fehlschlag des Tests wurden zuvor von mir erwartet. Dabei konnte ich das Wegfallen des Dialogs zum Benennen der Regel wie im vorherigen Testfall lösen und benötigte für die Warnung, dass keine Testfälle erstellt wurden, nur eine kleinere Anpassung des Testskripts.

ExportExcel: Entgegen meiner Erwartungen und aus anderen Gründen trat nur bei einem der Tests zum Exportieren ein Fehlschlag auf. Ursache des Fehlers war das zu weit nach rechts verschobene `SearchRectangle`, so dass ich diesen Entwicklungsfehler durch das Anpassen eines Wertes beheben konnte.

Import: In TESTONA 4.3 existierte nach einem Import ein Knoten namens `Description`, der in der neuen Version schlicht `A` heißt, so dass lediglich der letzte Vergleich von mir angepasst werden musste.

EditTree: Die Robustheit des Tests musste überarbeitet werden, da bei der automatischen Ausführung mit Jenkins der Test häufiger Probleme mit dem Ändern der Schriftgröße hatte, da diese nicht gefunden wurde. Nach einer Umstellung der Zahlen „9“ und „16“ in Grafiken konnte der Fehler behoben werden.

4.3.1.9 Jenkins Ergebnisse

Eine automatisierte Ausführung mit Hilfe von Jenkins kann gelegentlich zu zufälligen Fehlern führen, da Dateien nicht schnell genug geladen werden. Dennoch sollten solche Fehlschläge minimal auftreten, so dass die Auswertung der Tests einfacher ist. Von sieben Durchläufen der Tests auf TESTONA 4.1 mit einer durch-

schnittlichen Dauer von 48 Minuten ist der Test `ExportExcel` einmalig aufgrund eines nicht identifizierbaren Problems fehlgeschlagen. Bei der anschließenden Ausführung der reparierten Tests auf TESTONA 4.3 trat bei sieben Durchläufen mit einer durchschnittlichen Dauer von 43 Minuten einmalig ein Fehlschlag des Tests `BoundaryValueAnalysis` auf, der allerdings in den folgenden Durchläufen wieder problemlos funktionierte. Die erneut reparierten Tests wurden abschließend auf TESTONA 4.5 mit einer Durchschnittszeit von 45 Minuten neunmal ausgeführt. Dabei schlug der Test `ExportExcel` ohne erkennbaren Grund gleich zweimal fehl, funktionierte dann beim jeweils nächsten Durchlauf allerdings wieder.

4.3.2 QF-Test

Nach der Durchführung mit Eggplant habe ich die Tests mit QF-Test aufgenommen. Ursprünglich plante ich als zweites mit TestComplete zu arbeiten, was mir jedoch aufgrund einer verspäteten Lizenzfreigabe verwehrt wurde.

4.3.2.1 Einarbeitung

Aufgrund meiner bereits zuvor gesammelten Erfahrungen im Umgang mit QF-Test benötigte ich kaum Einarbeitungszeit. So konnte ich durch die zugängliche Bedienung der Aufnahmefunktion von QF-Test die ersten Testfälle dementsprechend zügig entwickeln. Lediglich das initiale Verbinden mit dem gewünschten SUT musste ich mir erarbeiten, da dies normalerweise nur einmal zu Beginn eines Projekts durchgeführt wird.

4.3.2.2 Arbeiten mit Grafiken

Im Gegensatz zu Eggplant benötigt QF-Test Grafiken nur für die „Checkpoints“, an denen das aktuelle Abbild des Bildschirms mit dem vom Testfall erwarteten Ergebnis abgeglichen wird. Selbst bei diesen Vergleichen ist aber nicht immer ein Bild notwendig, da QF-Test ein Auslesen des Inhalts von Tabellen oder Zuständen ermöglicht. Da im verwendeten SUT TESTONA jedoch nur wenige Tabellen existieren, habe ich sehr häufig mit grafischen Vergleichen gearbeitet. Abbildung 4.8 zeigt dazu die Einstellungsmöglichkeiten für einen beispielhaften Grafik-Checkpoint.

Wenn auch nicht so komfortabel bedienbar wie in Eggplant, bietet QF-Test ebenfalls die Möglichkeit Grafiken anzupassen. Allerdings ist es beispielsweise nicht möglich für eine bessere Ansicht die Farbe des Hintergrunds zu verändern. Daher sollte immer ein Abbild der ganzen Komponente aufgenommen und diese dann so genau wie möglich zugeschnitten werden. Das Zuschneiden einer Grafik erfolgt dabei durch das Klicken auf den mit einer Schere versehenen Button, nachdem der gewünschte Auswahlrahmen gezogen wurde. Direkt im Anschluss lässt sich dieser Schritt noch

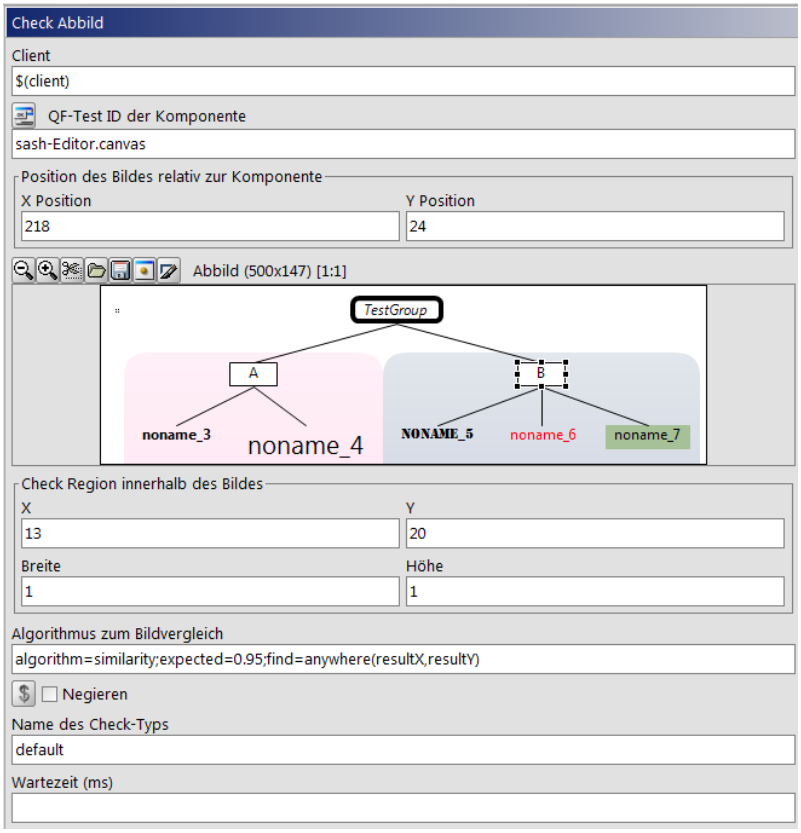

Abbildung 4.8: Grafik-Checkpoint in QF-Test

revidieren, später gibt es jedoch keine Möglichkeit mehr die Originalgrafik zu bearbeiten, außer sie wird neu aufgenommen.

QF-Test erzeugt bereits bei kleinsten Unterschieden zwischen zwei Grafiken Fehler, so dass beispielsweise eine abweichende Größe von einem Pixel zum Fehlschlag des Vergleichs führen kann. Daher ist es wichtig, das Abbild der Komponente jeweils effektiv zuzuschneiden. Zusätzlich habe ich den standardmäßig nicht verfügbaren Algorithmus `algorithm = similarity; expected=0.95;find=anywhere(resultX, resultY)` für jeden Grafik-Checkpoint verwendet, so dass sich das Abbild nicht pixelgenau an der ursprünglichen Position befinden muss, sondern auch leicht versetzt in der Originalkomponente liegen kann.

Da QF-Test auf die Elemente der GUI zugreifen kann, gibt es zum Grafikvergleich die Möglichkeit, bestimmte Zeilen, Spalten oder den Inhalt der gesamten Tabelle auszulesen (vgl. Abbildung 4.9). Neben dem Tauschen und Löschen einzelner Zeilen ist es dabei auch möglich, die einzelnen Werte der Tabelle als Text zu editieren.

4.3.2.3 Arbeiten mit Variablen

Auch für QF-Test habe ich den Pfad für die Input- und Output-Ordner in Variablen ausgelagert, um mögliche Änderungen schnell durchführen zu können. QF-Test bie-

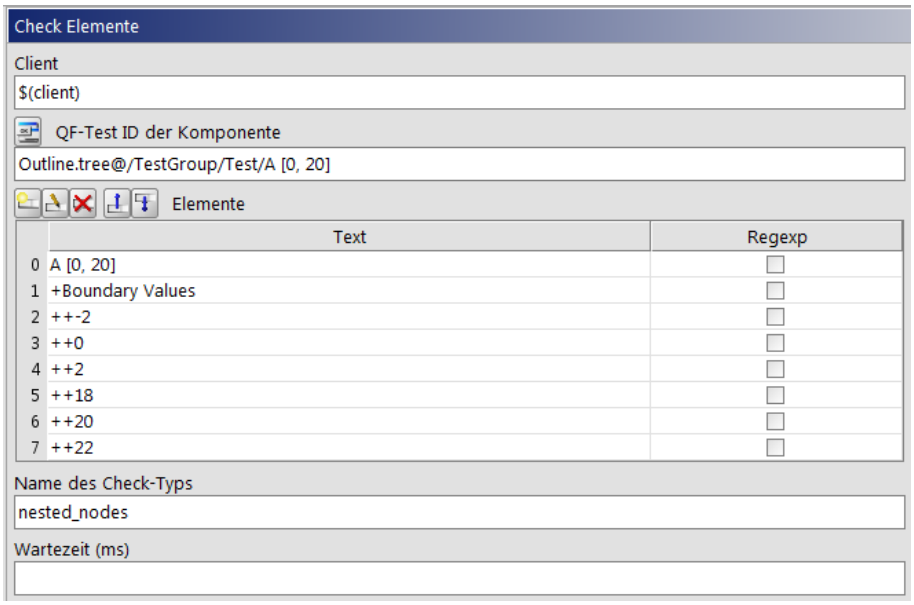

Abbildung 4.9: Tabellen-Checkpoint in QF-Test

tet dazu in den Optionen das Erstellen von Systemvariablen an, was insbesondere beim Entwickeln auf verschiedenen Computern von Vorteil ist.

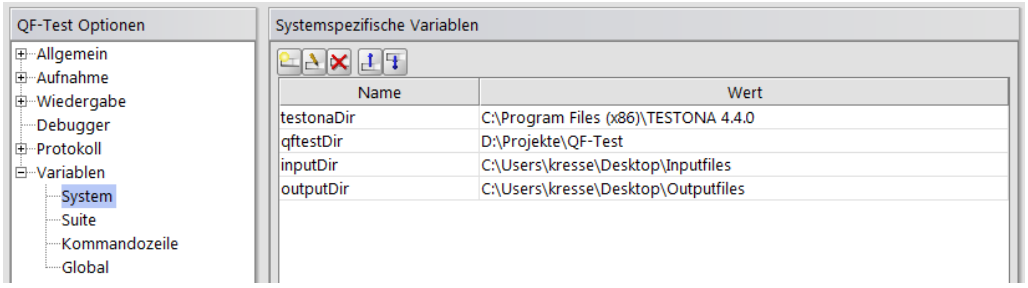

Abbildung 4.10: Variablen in QF-Test

Abbildung 4.10 zeigt das Fenster für die Erstellung und Bearbeitung von Variablen in QF-Test. Neben den Variablen für die Dateien existiert auch eine Variable für den Pfad des Projekts und für das Installationsverzeichnis des SUTs. Jeder Nutzer kann auf seinem Rechner seine eigenen lokalen Variablen anlegen, so dass diese nicht committed werden und somit keine Fehler auf anderen Systemen erzeugen.

4.3.2.4 Testskripte

Im Gegensatz zu Eggplant habe ich bei QF-Test die Aufnahmefunktion zum Erstellen der Tests verwendet, da diese eine komfortable Testentwicklung ermöglicht. Zwischen dem Beginn und dem Ende der Aufnahme hatte ich als Tester aber leider keinen Überblick darüber, was von QF-Test aufgezeichnet wurde. Besonders bei längeren Testfällen habe ich dadurch immer nur Teile eines Tests aufgenommen, um den entstandenen Code direkt zu überprüfen und ggf. Kommentare hinzuzufügen. Abbildung 4.11 zeigt den Quellcode des Tests `StartTestona`. Nachdem die ersten fünf

Zeilen das Programm starten, wird in den Zeilen sechs und sieben TESTONA in die obere linke Ecke des Bildschirms verschoben und anschließend auf eine Fenstergröße von 1000x1600 eingestellt. Anschließend wird die Prozedur `ResetPerspective` ausgeführt, die die Fenstereinstellungen von TESTONA auf den Standard zurücksetzt.

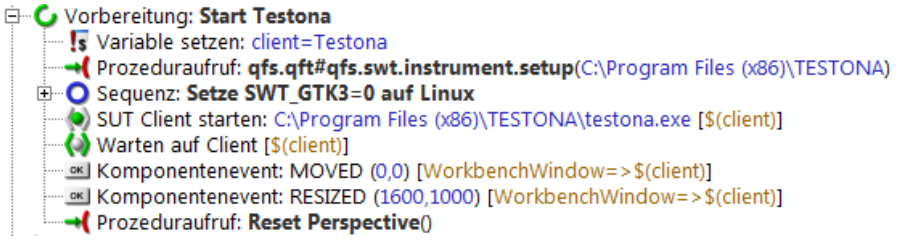

Abbildung 4.11: StartTestona in QF-Test

Skripte können in QF-Test nur durch manuelles Hinzufügen von Befehlen oder Teilaufnahmen erstellt werden. Da die Events immer auf die GUI-Schnittstelle aus dem Quellcode zugreifen, erwies sich das Einfügen von einzelnen Schritten gelegentlich als herausfordernd. Teilweise konnte ich dazu aber einfach die Werte von Koordinaten anpassen oder einen einfachen Klick zu einem Doppelklick transformieren.

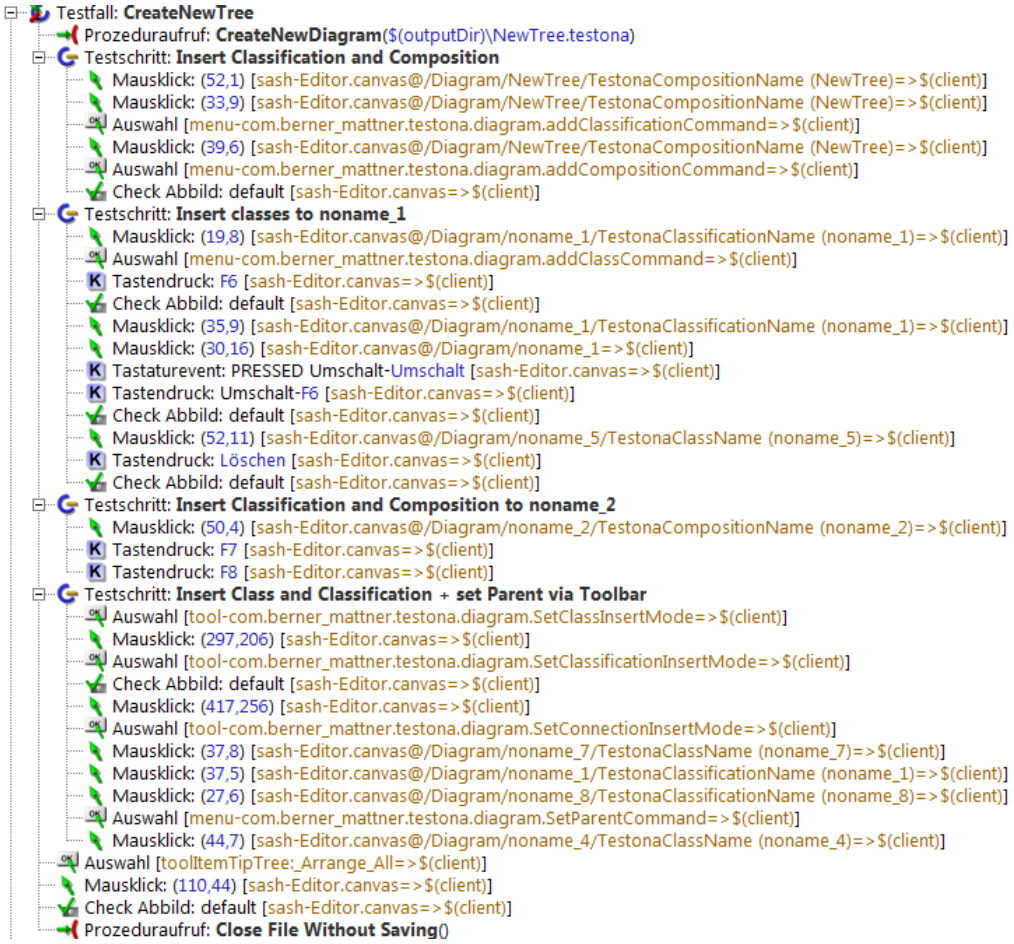

Abbildung 4.12: CreateTree in QF-Test

Abbildung 4.12 zeigt den Code des in QF-Test aufwendigeren Testfalls `CreateNewTree`. Der Test ist von mir dazu in vier Testschritte unterteilt worden, um eine höhere Übersicht zu erzeugen und die einzelnen Aktionen den Teilschritten besser zuordnen zu können. Die Aktionen lassen sich dabei grob in Mausklicks, Tastaturevents und Checks aufteilen, die jeweils nochmal spezifische Unterschiede aufweisen können. Aktionen wie ein Tastendruck auf F7 lassen sich beispielsweise schnell einordnen, während für ein Erkennen des passenden Kontexts bei komplexeren Ausdrücken wie `Mausklick:(x,y)[...]` häufig mehr Zeit benötigt wird.

4.3.2.5 MainSuite

Eine Test- bzw. Mainsuite unterteilt sich in QF-Test in die vier Hauptkomponenten „Testfallsatz“, „Prozeduren“, „Extrasequenzen“ sowie „Fenster und Komponenten“ (vgl. Abbildung 4.13). Alle Teilschritte innerhalb des obersten Testfallsatzes werden bei einem Durchlauf der Testsuite in absteigender Reihenfolge ausgeführt. Die Schritte in „Vorbereitung“ und „Aufräumen“ werden dabei jeweils vor bzw. nach jedem Testfallsatz durchgeführt.

Abbildung 4.13: MainSuite in QF-Test

Wie bereits bei Eggplant habe ich die dreigeteilte Kategorisierung der Testfälle beibehalten und sie dementsprechend in drei Testfallsätze eingeordnet. Im Ordner „Prozeduren“ sind sämtliche ausgelagerten Schritte, wie `OpenFile`, `CloseFile` oder `ResetPerspective` zu finden. Neu aufgezeichnete Sequenzen werden zunächst im Ordner „Extrasequenzen“ gespeichert, in dem sie lediglich mit dem Aufnahmedatum und der Uhrzeit gekennzeichnet sind. In diesem Ordner können die Sequenzen bereits probeweise abgespielt, getestet und nach Bedarf angepasst werden. Anschließend habe ich die Sequenzen entweder als Prozeduren ausgelagert oder in meinem Testfall an der entsprechenden Stelle eingefügt. Im Ordner „Fenster und Komponenten“ befindet sich zudem eine Liste mit allen GUI-Elementen, die in der Testsuite vorkommen.

4.3.2.6 Entwicklungszeit und −probleme

Bedingt durch die einfache Anwendung der Capture & Replay-Funktion von QF-Test konnte ich die Tests im Vergleich zu Eggplant deutlich schneller entwickeln. Tabelle 4.3 verdeutlicht meine im Verlauf immer schneller werdende Testentwicklung. Dabei musste ich die Tests `CreateEditTags` und `SearchingElements` doppelt aufzeichnen, da diese von QF-Test nicht korrekt gespeichert wurden.

QF-Test	TESTONA 4.1		TESTONA 4.3		TESTONA 4.5	
CreateNewTree	60 min					20 min
EditTree	90 min					
FoldTree	60 min					5 min
CopyPasteOfSubtrees	45 min					
WriteProtectedFiles	30 min					
PropertiesView	30 min					
Autolayout	30 min					
Outline1	30 min					
Outline2	60 min					
CreateRenameTestcases	20 min	20 min				
ChangeMarktypes	30 min				↯	10 min
GenerateTestcases	30 min				↯	45 min
ManualTestcases	105 min					
RenameGroupsAndSequences	60 min				↯	25 min
TestSequences	15 min					
BoundaryValueAnalyses	20 min					
CreateEditTags	40 min			20 min		
VariantManagementView	45 min					5 min
Help	320 min	15 min				5 min
SearchingElements	2x20 min	10 min	↯	20 min	↯	20 min
DependencyRulesLogical	20 min				↯	10 min
DependencyRulesNumeric	50 min				↯	20 min
ExportExcel	15 min	20 min [1]			↯	10 min
ExportHTML	20 min		↯	10 min	↯	35 min
Import	15 min	20 min	↯			15 min
Testcoverage	15 min					
StartTestona	60 min					
OpenFile	20 min					
CreateNewDiagram	30 min					
SaveFile	5 min					
CloseFile	5 min					
CloseAll	5 min					
CloseFileWithoutSaving	20 min					
BrowserSearch	20 min					
Summe	1460 min	85 min	3	50min	13	225 min

Tabelle 4.3: Entwicklung- & Reparaturzeit für QF-Test in Minuten

Ein Nachteil von QF-Test ist das gelegentlich auftretende, falsche Aufnehmen von Aktionen. Abbildung 4.14 zeigt die Aufnahme für die Aktion `Rename` im Rechtsklick-Menü von TESTONA, die den Namen eines Knotens umbenennt. Die auskommen-

tierten, grau geschriebenen Zeilen 4-6 wurden dabei während der Aufnahme ungewollt von QF-Test hinzugefügt, sind in diesem Fall aber überflüssig und führten sogar zum Fehlschlag der Aktion, da das Tastenkürzel F2 für `Rename` die Aktion frühzeitig beendete und in den Zeilen 4 bzw. 6 auf nicht vorhandene Menüeinträge geklickt wurde. Das unerwünschte Hinzufügen von Zeilen bei Aktionen, für die TESTONA ein Tastenkürzel besitzt, trat dabei flächendeckend auf und musste jeweils durch das manuelle Löschen der entsprechenden Zeilen von mir repariert werden.

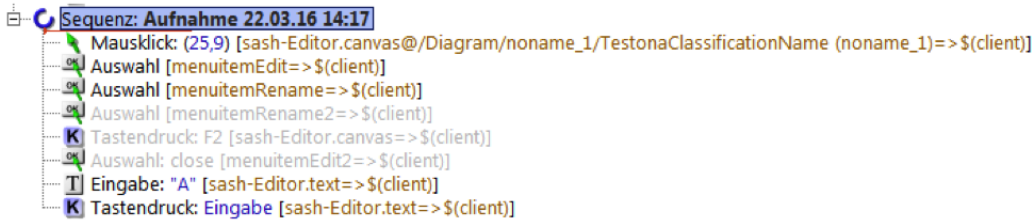

Abbildung 4.14: „Rename“ in QF-Test

Ein weiteres Beispiel ist der im Testfall `EditTree` verwendete Schieberegler, dessen Aufnahme nicht aus zwei Quelltextzeilen mit Start- und Endpunkt besteht, sondern für den QF-Test automatisch viele Zwischenschritte anlegt. Abbildung 4.15 zeigt den Beginn einer Aufnahme, in der der Regler im Test von 180 auf 75 geschoben wird. Die dabei ungewollt hinzugefügten Zeilen führten zwar nicht zu einem Fehlschlagen des Tests, sorgten jedoch für einen unnötig langen und unübersichtlichen Quelltext, so dass ich sie in den entsprechenden Testfällen jeweils entfernt habe.

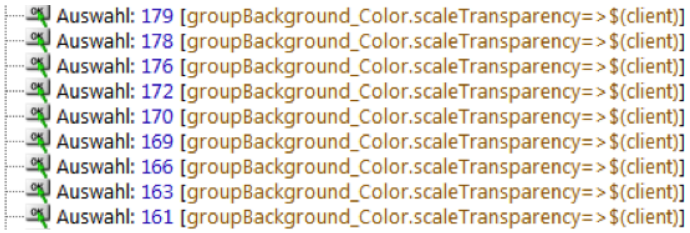

Abbildung 4.15: Verwendung eines Schiebereglers in QF-Test

Obwohl der Test `Help` nicht umfangreicher als die anderen Testfälle war, benötigte ich für seine Entwicklung mit 320 Minuten am meisten Zeit (vgl. Tabelle 4.3). In dem Test wird u. a. über einen Button die Hilfe gestartet, die sich in einem zusätzlichen Fenster öffnet. In diesem Fenster sollte dann nach dem Wort „Shortcuts“ gesucht werden. Während bei der Texteingabe zwar etwas aufgenommen wurde, funktionierte ein erneutes Abspielen anschließend jedoch nicht. Erst nach einer Beratung durch den Support von QF-Test konnte die Ursache geklärt werden: Die als Browser arbeitende Hilfe-Shell wurde nicht von QF-Test unterstützt. Nach mehrfacher Rücksprache mit dem Support konnte der Fehler durch das in Listing 4.12 dargestellte Skript gelöst werden, indem zunächst der Text im Suchtextfeld gelöscht und anschließend der gewünschte String zeichenweise eingegeben wird.

```
Delete All characters

from java.awt import Robot
from java.awt.event import KeyEvent
r = Robot()
r.setAutoDelay(100)
r.keyPress(KeyEvent.VK_CONTROL)
r.keyPress(KeyEvent.VK_A)
r.keyRelease(KeyEvent.VK_A)
r.keyRelease(KeyEvent.VK_CONTROL)
r.keyPress(KeyEvent.VK_DELETE)
r.keyRelease(KeyEvent.VK_DELETE)

input text and return

from java.awt import Robot
from java.awt.event import KeyEvent
r = Robot()
r.setAutoDelay(100)

chars = list(rc.lookup("text").lower())

for c in chars:
    r.keyPress(ord(c)-32)
    r.keyRelease(ord(c)-32)

r.keyPress(KeyEvent.VK_ENTER)
r.keyRelease(KeyEvent.VK_ENTER)
```

Listing 4.12: Skript für Help-Test

Bei einem Aufruf der Prozedur innerhalb des Tests müssen dabei einige Werte als Variablen der Suche übergeben werden (vgl. Abbildung 4.16). Dazu gehören die ID des Textfelds, der im Suchfeld einzutragende Text und die Koordinaten „x, y“ zur Bestimmung des Interaktionspunktes im Textfeld.

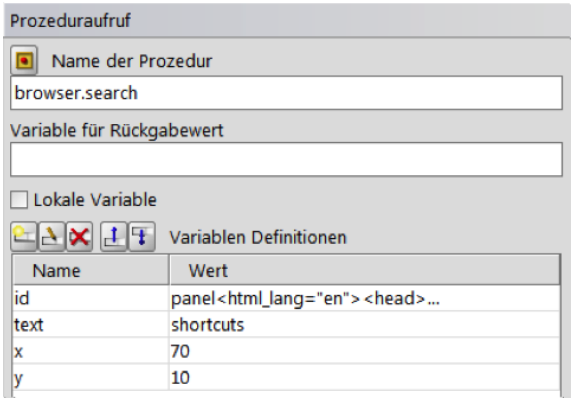

Abbildung 4.16: Aufruf der Search Prozedur

Auffallend ist der im Vergleich zu seinem Vorgänger `DependencyRulesLogical` 30 Minuten mehr Entwicklungszeit benötigende Test `DependenyRuleNumeric` (vgl. Tabelle 4.3). Grund für die höhere Entwicklungszeit war ein Button mit Dropdown-Option, der von QF-Test lediglich als Button verwendet wurde, ohne die passende Einstellung im Dropdown-Menü auszuwählen. Abbildung 4.17 zeigt den Dialog der „DependencyRules“, der unten links den beschriebenen Button mit Dropdown-Option zum Erstellen einer neuen Regel enthält. Da zu Beginn des Tests der Button auf „New Logical Rule“ eingestellt war und QF-Test dies nicht über das Dropdown-Menü umstellte, wurde der ungeplante Dialog für eine logische Regel geöffnet, was dann zum Abbruch des Testfalls führte. Um einen korrekt eingestellten Klick auf den Button zu ermöglichen, habe ich die Koordinaten für das Dropdown-Menü manuell ermittelt und statt eines direkten Interagierens mit der Komponente verwendet. Ein Arbeiten mit Koordinaten war dabei möglich, da diese nur bezüglich des Buttons und nicht in Bezug auf den gesamten Bildschirm berechnet wurden, so dass keine Gefährdung der Robustheit der Aktion vorlag.

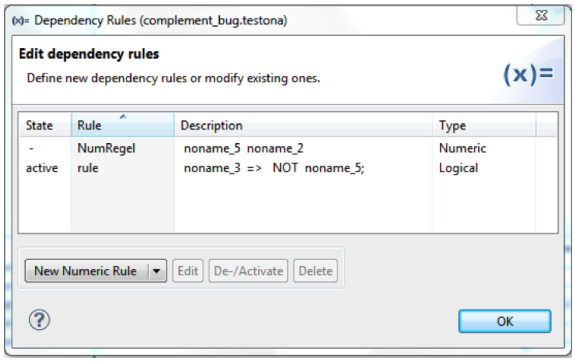

Abbildung 4.17: Dialog „Dependency Rules“

Im Gegensatz zum fehlerfreien Implementieren unter Eggplant musste ich für QF-Test nach der fertigen Entwicklung bereits fünf Tests reparieren (vgl. Tabelle 4.3). Im Testfall `CreateRenameTestcases` wurden zwei Testfälle nicht hinzugefügt, die ich durch eine effektive Verzögerung innerhalb des Tests erneut einbinden konnte. Beim Test `Help` war das Hilfefenster nicht rechtzeitig geladen worden und sorgte dadurch bei einem anschließenden Vergleich für einen Fehlschlag. Auch hier habe ich eine Verzögerung von 2000 Millisekunden vor dem Vergleich eingebaut, so dass das Fenster ausreichend Zeit zum Laden besitzt. Im Test `SearchingElements` konnte ein Knoten im Baum nicht umbenannt werden, da das Rechtsklick-Menü nicht gefunden wurde. Der Fehler konnte von mir durch die Einstellung „Popup Trigger“ beim entsprechenden Ausführungsschritt behoben werden. Die Tests `Import` und `ExportExcel` schlugen aufgrund eines noch offenen „SearchTab“ in den „Properties“ fehl, was von mir durch ein präventives Schließen des Tabs mit anschließendem Zurücksetzen der Fenstereinstellungen über `ResetPerspective` repariert wurde.

4.3.2.7 Fehlschläge von TESTONA 4.1 zu 4.3

Beim Versionswechsel von TESTONA 4.1 auf 4.3 sind lediglich zwei der drei erwarteten Fehlschläge eingetreten. Ein Test hat trotz erwartetem Fehlschlag funktioniert, während der Test `CreateEditTags` zusätzlich fehlgeschlagen ist.

CreateEditTags: Im Test konnte QF-Test die Bildlaufleiste nicht mehr verwenden, um ein bestimmtes Icon auszuwählen, so dass ich den Test neu aufnehmen musste.

SearchingElements: Der Testfall konnte nicht umbenannt werden, da sich das Rechtsklick-Menü nicht öffnete. Der Grund für diesen Fehler konnte von mir nicht ermittelt werden, eine erneute Aufnahme löste jedoch das Problem. Darüber hinaus ist aufgrund von veränderten Suchergebnissen ein Check fehlgeschlagen, der ebenfalls von mir neu aufgenommen werden musste.

HTMLExport: Der Check bezüglich der Ergebnisse bei der Suche nach dem Buchstaben „c“ ist fehlgeschlagen, da eine neue Exportvariante in TESTONA 4.3 ebenfalls diesen Buchstaben enthielt. Auch hier musste lediglich der Check von mir erneuert werden.

Insgesamt benötigte ich für die Reparatur der drei fehlgeschlagenen Tests 50 Minuten. Beide von mir erwarteten Fehlschläge sind aufgrund der angenommenen Ursache beim Versionswechsel eingetreten und waren demnach unvermeidbar. Der Fehler im Test `CreateEditTags` beruht allerdings auf einem Problem innerhalb des Quellcodes von QF-Test und hätte durch eine bessere Funktionsweise vermieden werden können.

4.3.2.8 Fehlschläge von TESTONA 4.3 zu 4.5

Der Versionswechsel des SUTs auf TESTONA 4.5 umfasste mit 13 Fehlschlägen die Hälfte aller erstellten Testfälle.

CreateNewTree: In der Preview wurde im unteren Bereich eine schmale Trennlinie eingefügt. Diese Änderung führte dazu, dass QF-Test innerhalb des Editors nicht mehr interagieren konnte. Um diesen Test zu reparieren, habe ich einen Klick innerhalb des Editors aufgenommen und dabei festgestellt, dass sich die Indexstruktur der Canvas änderte. Somit musste ich lediglich die Werte der verwendeten Canvas anpassen und die Interaktionen innerhalb des Editors funktionierten wieder in allen Tests.

FoldTree: Aufgrund einer von mir um vier Pixel zu knapp zurechtgeschnitten Grafik schlug ein Check fehl, der mit Hilfe einer entsprechenden Anpassung der Grafik von mir zügig repariert werden konnte.

ChangeMarktypes: Wie von mir erwartet führte ein fehlender Dialog beim Ändern des Modus von „Explizit“ zu „Implizit“ zum Fehlschlag des Tests. In diesem

Fall musste ich lediglich die drei Zeilen für den entsprechenden Dialog auskommentieren.

GenerateTestcases: Der neue Algorithmus für die „Default Minimal Combination“ benötigte eine mehrfache Anpassung des Tests. Zudem erschien ein neuer Warnhinweis für nicht erstellte Testfälle, der ebenfalls von mir neu aufgenommen und eingepflegt werden musste.

RenameGroupsAndSequences: Die neue Position von „Advanced Rename“ unter „Edit“ führte auch unter QF-Test zu Problemen, die ich an vier Stellen innerhalb des Testfalls reparieren musste.

VariantenManagementView: Beim Erstellen einer neuen Variante wurde der Button „New“ nicht gefunden. Da ich keinen spezifischen Grund für den Fehlschlag feststellen konnte, habe ich den Klick neu aufgenommen und ersetzt.

Help: Die Suche innerhalb der Hilfe lieferte mehr Ergebnisse als erwartet, weshalb der anschließende Vergleich durch eine Neuaufnahme angepasst werden musste.

SearchingElements: Die erweiterte Suche nach „Attributes Tags“ lieferte unerwartet mehr Ergebnisse, die innerhalb der „PropertiesView“ in Tabellenform angezeigt wurden. Bei jedem Durchlauf änderte sich jedoch die Reihenfolge der Elemente, so dass der Inhalt der Tabelle nicht konsistent war. Da somit ein normaler Bildabgleich nicht möglich war, entschloss ich mich dazu lediglich die Anzahl der Elemente innerhalb einer Spalte zu vergleichen.

DependencyRulesLogical: Der Dialog zur Eingabe des Namens einer neuen Regel ist entfallen, weshalb die Interaktionen mit dem Dialog aus dem Skript entfernt werden mussten. Zusätzlich musste ich die Texteingabe im neuen Fenster aufnehmen und im Testskript einfügen.

DependencyRulesNumeric: In diesem Testfall ist ebenfalls der Dialog zur Benennung einer neuen Regel entfallen und musste dementsprechend von mir angepasst werden. Zudem wurde ein Warnhinweis für nicht erstellte Testfälle hinzugefügt, für den ich lediglich einen zusätzlichen Klick auf den OK-Button aufnehmen musste.

ExportExcel: Obwohl sich der Dialog im Export komplett änderte, musste ich mit Ausnahme eines blockierten Klicks keine weiteren Anpassungen vornehmen.

ExportHTML: Wie beim vorherigen Testfall wurde der Dialog für den Export verändert. Ein Element konnte nicht mehr selektiert werden, da es von einem Listenelement in eine Checkbox umgewandelt wurde. Nach einer zweimaligen Neuaufnahme konnte ich den Fehler beheben. Darüber hinaus wurde eine neue Export-Variante hinzugefügt, so dass die Suche nach dem Buchstaben "c" einen zusätzlichen Treffer ergab und entsprechend der Checkpoint von mir angepasst werden musste.

Import: Im Import-Dialog änderte sich an einer Stelle die Dialogstruktur, weshalb ein Klick nicht mehr möglich war. Dieser Klick wurde von mir durch eine Neuaufnahme angepasst. Zudem wurde im fertig importierten Baum der Knoten

"Description" entfernt, so dass ich für beide Vorkommen des Knotens die Checks anpassen musste.

4.3.2.9 Jenkins Ergebnisse

Die mit QF-Test entwickelten Tests wurden zur Überprüfung ihrer Funktionalität ebenfalls mit Hilfe von Jenkins automatisiert ausgeführt. Dabei erfolgte für die Tests der TESTONA Version 4.1 eine siebenfache Ausführung, bei der nur in einem Durchlauf Probleme auftraten. In diesem Durchlauf ist ein modaler Dialog innerhalb eines Testfalls aufgetreten und hat die folgenden neun Testfälle in ihrer Ausführung behindert, so dass von einem zufälligen Fehler ausgegangen werden kann. Da QF-Test direkt auf die GUI-Elemente zugreift und keine Bildvergleiche zur Interaktion benötigt, lag die Durchschnittszeit für einen Durchlauf bei ca. 8 Minuten. Die reparierten Tests für TESTONA 4.3 besaßen zwar dieselbe Durchschnittszeit, mussten aber von mir insgesamt 21-mal ausführt werden, um fünf fehlerfreie Durchläufe zu erreichen. Die Ursache für die häufigen Fehlschläge lag dabei in der schnellen Interaktion mit dem SUT. So ist der Test `SearchingElements` in sechs Durchläufen fehlgeschlagen. Wurde der Jenkins Job von mir erneut angestoßen, lief der Test allerdings wieder ohne Probleme durch. Da die Ausführungsaktionen jeweils syntaktisch korrekt sind, könnte eine höhere Stabilität nur durch das Einfügen zusätzlicher Wartezeiten erreicht werden. Während die Durchläufe für TESTONA 4.5 mit 8,5 Minuten eine höhere durchschnittliche Ausführungsdauer besaßen, waren sie wieder deutlich stabiler. Lediglich in drei von zwölf Durchläufen traten Fehler auf, wobei jeweils ein anderer Test ohne erkennbaren Grund fehlschlug.

4.3.3 TestComplete

Das zuletzt von mir verwendete Capture & Replay-Tool zum Erstellen der Testfälle war TestComplete, das wie QF-Test aus der zweiten Generation stammt und nicht nur im Hinblick auf die Aufnahmefunktion Ähnlichkeiten aufwies.

4.3.3.1 Einarbeitung

Sowohl die Installation von TestComplete, als auch das Erstellen einer TestSuite konnten von mir problemlos und zügig durchgeführt werden. TestComplete stellt eine einfache Capture & Replay-Funktion zur Verfügung, die mir eine schnelle Aufnahme der Tests ermöglichte. Trotz der komfortablen Einarbeitung in TestComplete entstanden im Verlauf der Testerstellung viele zeitaufwendige Probleme mit Tests und Prozeduren, die ab Kapitel 4.3.3.6 ausführlicher beschrieben sind.

4.3.3.2 Arbeiten mit Grafiken

TestComplete interagiert ebenso wie QF-Test direkt über die GUI-Elemente mit dem SUT, so dass Grafiken dementsprechend nur bei „Region Checkpoints“ von mir verwendet wurden. Die Aufnahme eines Checkpoints bzw. Checks kann dabei entweder innerhalb der Aufnahme oder an deren Ende erfolgen. Bei der Erstellung einer Grafik für einen Checkpoint kann zusätzlich eine Maske für die jeweilige Grafik erstellt werden. Die Maske legt dabei die Bereiche der Grafik fest, die bei einem Vergleich vernachlässigt werden, um somit ggf. einen Fehlschlag des Tests zu verhindern.

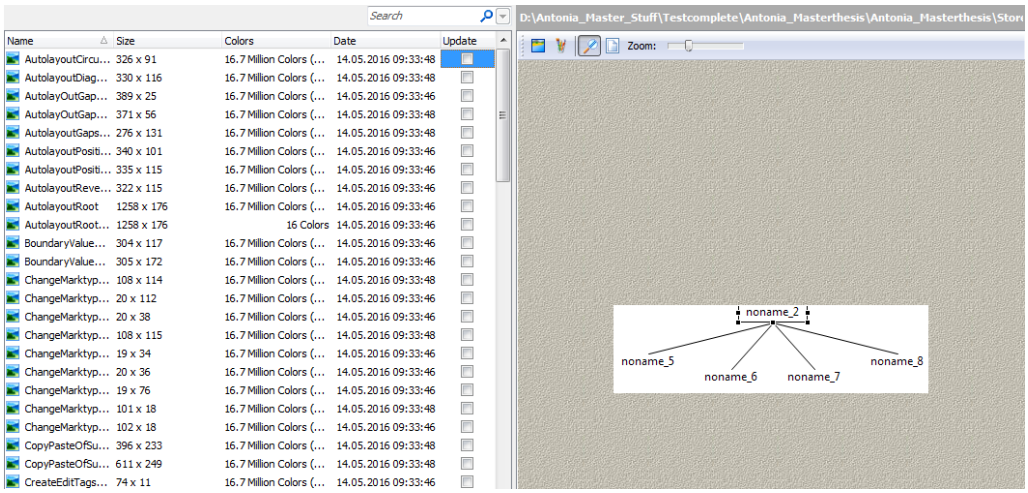

Abbildung 4.18: Region-Checkpoints in TestComplete

Abbildung 4.18 zeigt die Übersicht der Grafiken innerhalb von TestComplete. Für das an sich bereits komplexe Verwalten und Bearbeiten von Grafiken in TestComplete ist dieser Überblick notwendig, da beispielsweise beim Verkleinern oder Vergrößern einer Grafik ebenfalls die dazugehörende Maske angepasst werden muss, um eventuelle Fehler im Testablauf zu vermeiden. Die Namensgebung der Grafiken ist dabei auch in TestComplete aufgrund deren einmaliger Verwendung nicht festgelegt, sondern nur im Hinblick auf eine bessere Einordnung in den jeweiligen Kontext relevant.

4.3.3.3 Arbeiten mit Variablen

Auch bei TestComplete habe ich den Pfad der Input- und Output-Ordner in Variablen abgespeichert. Die Erstellung eines Skripts oder einer Option im Menü existierte in meiner Version von TestComplete nicht, allerdings konnte ich in der MainSuite über einen Reiter die Variablen erzeugen.

Abbildung 4.19 zeigt eine Übersicht der von mir erstellten temporären und persistenten Variablen. Während die Verwendung der Variablen in TestComplete für mich sehr intuitiv war, hatte ich anfangs Schwierigkeiten die Definition der Variablen an den passenden Stellen durchzuführen. Dabei verwendete ich zunächst in

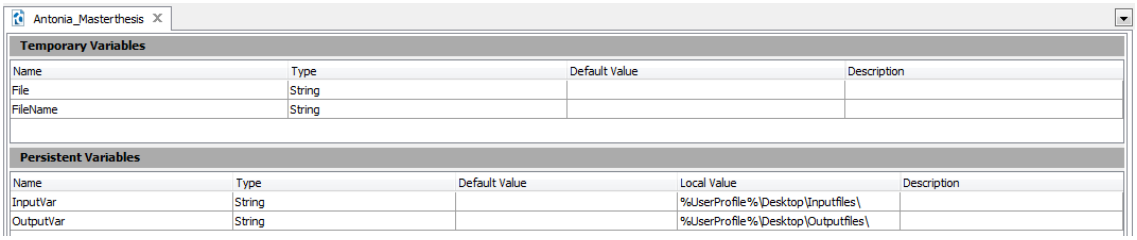

Abbildung 4.19: Variablen in TestComplete

den persistenten Variablen auch den vollen String, entschied mich später aber dazu den Anfang durch „%UserProfile% “ zu ersetzen. Durch diese Änderung musste ich dann allerdings einige Testfälle anpassen, da die Eingabe von „UserProfile“ nur an bestimmten Stellen im Dialog funktioniert.

4.3.3.4 Testskripte

Das Starten von TESTONA gelang mir in TestComplete zunächst problemlos, da lediglich einmal das SUT von mir definiert werden musste. Leider startete TestComplete dann aber bei jeder erstmaligen oder fortgesetzten Aufnahme das SUT neu, obwohl bereits eine Instanz existierte, so dass eine Warnmeldung ausgegeben wurde. Aufgenommene Skripte wurden von mir direkt im Testfall aufgenommen und nicht wie in QF-Test erst in einer Extrasequenz gespeichert.

Abbildung 4.20 verdeutlicht am abgebildeten Testfall `CreateNewTree`, dass der Quellcode ohne Kommentare nur schwer lesbar ist, weshalb ich mindestens alle vier Zeilen einen Kommentar eingefügt habe. TestComplete arbeitet auf der Objekt-Ebene, so dass die Aktionen innerhalb von mehreren sog. „Shells“ ausgeführt werden. In der ersten Spalte steht dabei das Objekt mit dem interagiert wird, in der zweiten Spalte die Art der Interaktion, in der dritten Spalte befinden sich zusätzliche Informationen und in der vierten Spalte werden automatisch bei der Aufnahme generierte Kommentare notiert. Innerhalb des Testfalls wird in den ersten Zeilen eine neue Datei angelegt, wobei der String mit dem Pfad in Zeile acht mit „SetText“ gesetzt wurde und sich aus der Variablen „OutputVar“ sowie dem Namen der Datei zusammensetzt. In den darauffolgenden Zeilen werden verschiedene Interaktionen im Baum ausgeführt, während zwischenzeitlich die Überprüfung der jeweiligen Zustände der GUI mit Checks erfolgt.

4.3.3.5 MainSuite

Eine Suite in TestComplete unterteilt sich grob in die „Project Suite“ und die „Project Suite Logs“ (vgl. Abbildung 4.21). In der „Project Suite“ befinden sich dabei unter „Script“ alle in einer Programmiersprache geschriebenen Skripte, im Ordner „KeywordTests“ alle aufgenommenen Tests inklusive der ausgelagerten Funktionen,

Abbildung 4.20: Testfall `CreateNewTree` in TestComplete

unter „NameMapping“ alle gemappten Objekte und unter „Stores“ sämtliche Grafiken und Tabellen, die für Checkpoints benötigt werden.

In TestComplete ist die MainSuite eine zusätzliche Datei, in der fertig entwickelte Testfälle gelagert und in der gewünschten Reihenfolge geordnet werden können. Im unteren Bereich ist zudem der Reiter für die Variablen zu finden.

Abbildung 4.22 zeigt die von mir verwendete MainSuite, wobei der dritte Ordner in dieser Darstellung eingeklappt ist. Wie bei den anderen Capture & Replay-Tools habe ich die Tests in drei Gruppen aufgeteilt und sie innerhalb der MainSuite mit den passenden Testnamen benannt. Obwohl in der dritten Spalte für jeden Test die Häufigkeit seiner Ausführung und in der vierten Spalte das „TimeOut“ eingestellt werden konnte, habe ich an den Basiseinstellungen keine Veränderungen vor-

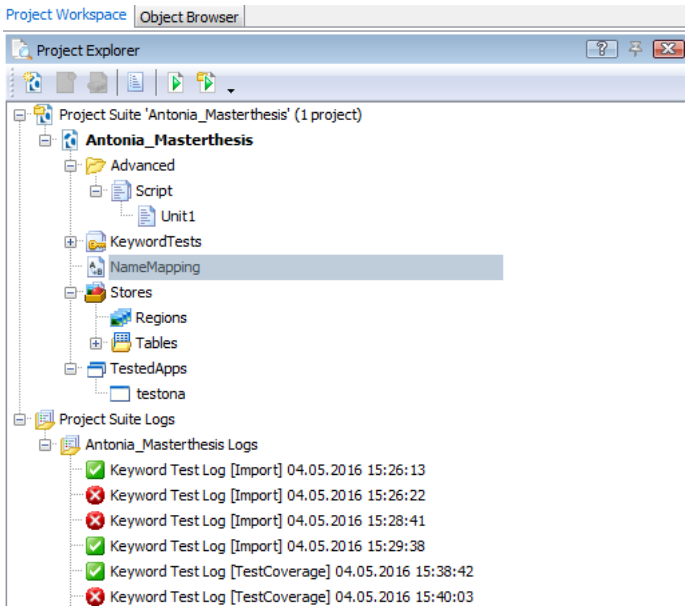

Abbildung 4.21: Suite in TestComplete

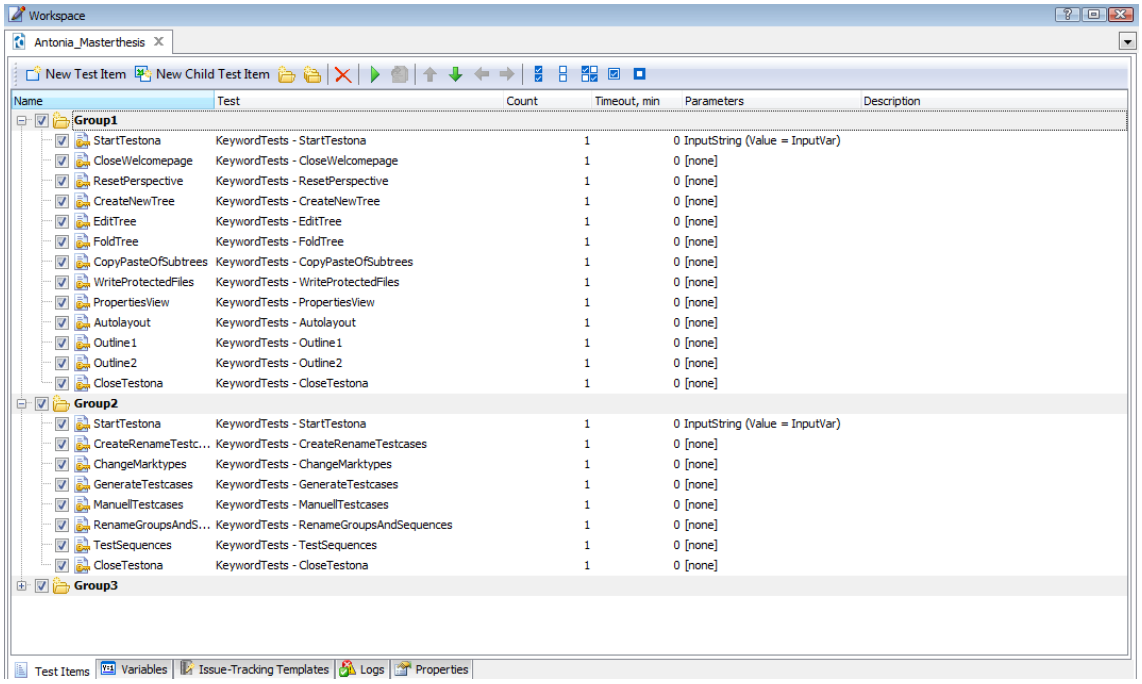

Abbildung 4.22: MainSuite in TestComplete

genommen, da bei den anderen Tools jeder Test ebenfalls nur einmal pro TestSuite ausgeführt wurde.

4.3.3.6 Entwicklungszeit und −probleme

Die erste Spalte von Tabelle 4.4 verdeutlicht den Aufwand beim Implementieren der 26 Testfälle in TestComplete für TESTONA 4.1 Auffallend ist dabei nicht nur der deutlich höhere Entwicklungsaufwand für die Tests `CreateNewTree`, `EditTree`, `FoldTree` und `PropertiesView`, sondern auch die insgesamt zahlreichen Fehlschlä-

ge von fertig entwickelten Tests, obwohl keine Änderungen am SUT vorgenommen wurden.

Testcomplete	TESTONA 4.1		TESTONA 4.3		TESTONA 4.5	
CreateNewTree	150 min	80 min				
EditTree	130 min	30 min				
FoldTree	240 min					
CopyPasteOfSubtrees	20 min	195 min				
WriteProtectedFiles	30 min					
PropertiesView	240 min					
Autolayout	70 min					
Outline1	70 min	20 min				
Outline2	40 min					
CreateRenameTestcases	45 min					
ChangeMarktypes	40 min				↯	25 min
GenerateTestcases	40 min				↯	40 min
ManualTestcases	45 min					
RenameGroupsAndSequences	40 min				↯	15 min
TestSequences	40 min					
BoundaryValueAnalyses	30 min	40 min				
CreateEditTags	50 min					
VariantManagementView	25 min					80 min
Help	30 min	100 min				
SearchingElements	30 min	130 min	↯	10 min	↯	40 min
DependencyRulesLogical	15 min				↯	60 min
DependencyRulesNumeric	35 min				↯	30 min
ExportExcel	20 min	90 min		15 min	↯	10 min
ExportHTML	20 min		↯	20 min	↯	25 min
Import	30 min	130 min	↯	20 min		45 min
Testcoverage	15 min	30 min				
StartTestona	10 min					
CloseFileWithoutSaving	15 min	30min				
Save	15 min	10min				
OpenFile	120 min	40 min				
ClickPropertyItem	120 min	30 min				
ReadNameFromCore	120 min					
Summe	1940 min	955 min	4	65 min	10	370 min

Tabelle 4.4: Entwicklung- & Reparaturzeit für TestComplete in Minuten

Die meisten Probleme machte mir bei der Entwicklung der Tests das von TestComplete verwendete Namemapping. Alle verwendeten Fenster werden in TestComplete anhand ihrer Shells gemappt, wobei diesen zur Identifizierung eindeutige Merkmale zugewiesen werden.

Abbildung 4.23 zeigt einen Ausschnitt aus den Namemappings in TestComplete für das von mir verwendete SUT TESTONA. Dabei ist direkt zu erkennen, dass die GUI von TESTONA stark verschachtelt ist und das Namemapping daher aus einer hohen Anzahl an Shells mit Unterelementen besteht. Dies führte beispielsweise dazu, dass TestComplete die Shells jeweils anhand der Titel in der obersten Leiste identifi-

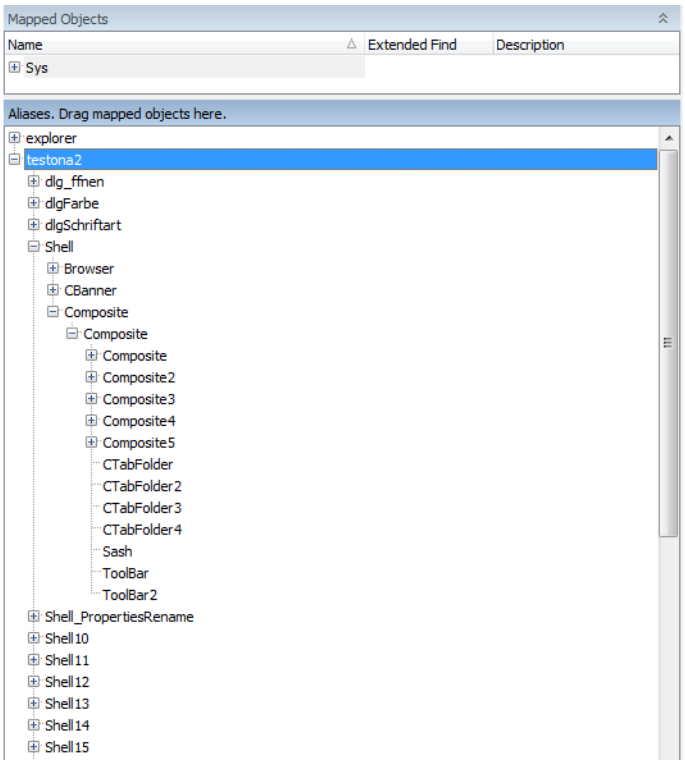

Abbildung 4.23: Namemapping in TestComplete

zierte. So stand beim Start von TESTONA im Titel lediglich „TESTONA ∗“, später beim Öffnen bzw. Erstellen einer Datei wurde der Name aber zusätzlich im Titel angezeigt. TestComplete erkannte dann, aufgrund des unterschiedlich geschrieben Titels, das Hauptfenster von TESTONA nicht mehr. Ähnliche Probleme verursachte der von mir ausgelagerte Test `FileOpen`, der ebenfalls immer einen anderen Titel erhielt und dementsprechend Folgefehler in anderen Tests auslöste. So entstanden bereits nach drei Testfällen viele unnötige Probleme mit dem Namemapping von TestComplete durch eine Anzahl an eigentlich identischen Shells.

Das Namemapping kann innerhalb von TestComplete über den in Abbildung 4.24 dargestellten Dialog bearbeitet werden. Auf der linken Seite befinden sich die von TestComplete zur Identifizierung des Dialogs verwendeten Werte, auf der rechten Seite stehen alle Werte des Dialogs. Viele der Werte auf der rechten Seite werden immer wieder neu generiert und sind demnach nicht zur Identifizierung geeignet. Das Hauptfenster von TESTONA konnte ich in eine Shell abbilden, indem ich in der Eigenschaft „Title“ alles nach TESTONA durch das Symbol „∗“ ersetzte, so dass die Angaben nach TESTONA ignoriert wurden. Das gilt auch für andere Dialoge, die beispielsweise im Titel den Namen einer Datei enthalten. Durch diese Änderung mussten bereits fertige Tests von mir erneut angepasst werden, da in ihnen andere Shells verwendet wurden. Die überflüssigen Elemente im Namemapping habe ich anschließend gelöscht, um den bestmöglichen Überblick zu behalten. Erschwert wurde dies durch die beim Namemapping erzeugte starke Verzweigung und Tiefe der

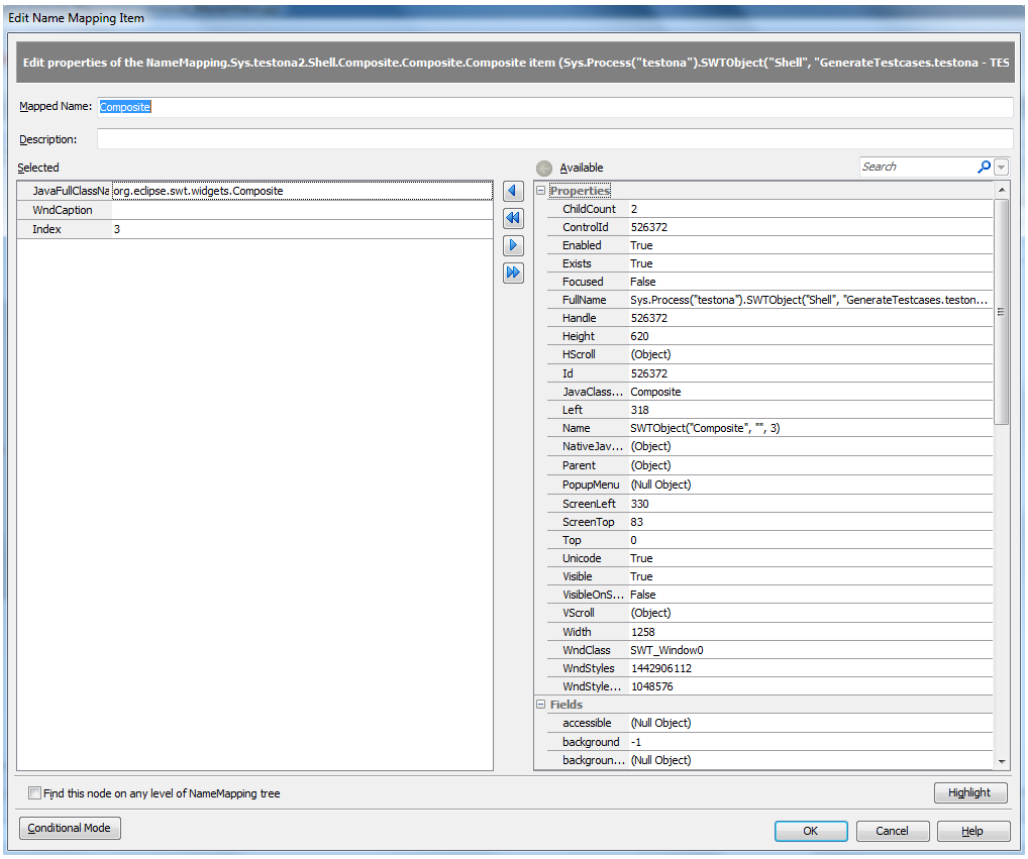

Abbildung 4.24: Bearbeiten des Namemappings in TestComplete

gemappten Objekte. So besaß beispielsweise ein Button aus dem Testfall `EditTree` zunächst den in Listing 4.13 dargestellten Pfad.

```
Aliases.testona2.Shell6.Composite.Composite.Composite2.Composite
    .PageBook.TabbedPropertyComposite.LayoutComposite.
    ScrolledComposite.LayoutComposite.LayoutComposite.
    LayoutComposite.LayoutComposite.Group.Composite.ColorButton
```

Listing 4.13: Ursprünglicher Button aus dem Testfall `EditTree`

Beim nächsten Durchlauf änderte sich der in Listing 4.14 erneut dargestellte Pfad dabei ungewünscht in der Zahl nach dem dritten „Composite“.

```
Aliases.testona2.Shell6.Composite.Composite.Composite3.Composite
    .PageBook.TabbedPropertyComposite.LayoutComposite.
    ScrolledComposite.LayoutComposite.LayoutComposite.
    LayoutComposite.LayoutComposite.Group.Composite.ColorButton
```

Listing 4.14: Angepasster Button aus dem Testfall `EditTree`

Während der Testentwicklung von `EditTree` stellte ich zudem fest, dass TestComplete die zwei Hauptfenster „Editor“ und „Properties“ nicht eindeutig identifizieren konnte und ihnen bei jedem Neustart von TESTONA einen neuen Index zuwies, so dass die Fenster zufällig als richtig oder falsch identifiziert wurden. Aufgrund fehlender, eindeutig identifizierbarer Elemente in den Fenstern, verwendete ich stattdessen

die konstante Fensterhöhe, der über dem Editor verankerten Toolbar zur Identifizierung. Neben den beiden Hauptfenstern konnte TestComplete auch die einzelnen Abschnitte innerhalb der Toolbar nicht eindeutig identifizieren, so dass ich die eindeutige Breite der jeweiligen Elemente verwenden musste.

Neben den durch das Namemapping entstandenen Problemen erstellte TestComplete bei Aufnahmen, ähnlich wie QF-Test, gelegentlich zu viele unerwünschte Teilschritte. So wurde für jedes Umbenennen zusätzlich ein Tastendruck auf F2 aufgenommen, so dass das eigentliche Umbenennen blockiert wurde. Teilweise wurden die zwei Schritte „HScroll“ und „VScroll“ zusätzlich aufgenommen, die ebenfalls zu Problemen führten und jeweils von mir manuell entfernt werden mussten. Ein weiteres Problem stellte die Ausführung von Aktionen auf falschen Objekten dar. So wurde beispielsweise der Tastendruck F6 für das Element „Text“ anstatt „Tree“ aufgenommen und bewirkte dadurch keine geplanten Veränderungen im Baum. Diese Fehler musste ich ebenfalls durch wiederholte Aufnahmen oder manuellen Anpassungen beheben. Da solche Aufnahmefehler in nahezu jedem mit TestComplete erstellten Test vorkamen, entstand zunehmend ein signifikanter Zeitaufwand für die Fehlerbehebung, der jedoch im Vergleich zu den geschilderten Problemen beim Namemapping einfach zu beheben war.

Eine weitere Herausforderung in der Testerstellung mit TestComplete stellten sämtliche Interaktionen innerhalb des Bereichs „Properties“ dar. Das erste Hindernis bestand im Auswählen eines bestimmten Menüpunkts, da dieser immer wieder neue IDs zugewiesen bekam und TestComplete ihn nicht anhand seines Namens identifizieren konnte. Nach diversen erfolglosen Lösungsansätzen mit Neuaufnahmen und Interaktionen auf einer höheren Ebene über Koordinaten, blieb nur die Möglichkeit zur manuellen Notation der in Listing 4.15 dargestellten Funktion `clickPropertyItem`. Dieser Funktion wird beim Aufruf ein Wert übergeben, der der Position des gewünschten Menüeintrags entspricht, so dass anschließend auf diesen geklickt werden kann.

```
function clickPropertyItem(index ){
  var testona;
  testona = Aliases.testona2;
  var  list = testona.Shell.Composite.Composite.Composite3.
      Composite.PageBook.TabbedPropertyComposite.LayoutComposite.
      LayoutComposite.TabbedPropertyList
  var item = list.getElementAt(index);
  var x = item.getBounds().x+10;
  var y = item.getBounds().y+10;
  list.Click(x,y);
}
```

Listing 4.15: Funktion zur Navigation innerhalb der „Properties“

Zudem sorgten auch Vergleiche von Werten innerhalb der „Properties“ für Probleme, da der gewünschte Bereich des Abbilds bei der Ausführung stets schwarz angezeigt wurde. Dementsprechend musste auch für dieses Problem von mir eine zusätzliche Funktion geschrieben werden, die den Namen an einer bestimmten Position ausliest (vgl. Listing 4.16). Da sich die Positionen der zu prüfenden Items gelegentlich änderte, existiert auch noch eine zweite äquivalente Funktion mit leicht anderen Werten.

```
function checkName(name){
  var testona;
  testona = Aliases.testona2;
  var  propTree = testona.Shell.Composite.Composite.Composite3.
      Composite.PageBook.TabbedPropertyComposite.LayoutComposite.
      ScrolledComposite.LayoutComposite.LayoutComposite.
      LayoutComposite.LayoutComposite.LayoutComposite.Tree.Tree
  var item = propTree.getItem(2);
  var data = item.getData();
  var dataName = "";
  data= propTree.getItem(2).getData();
  return data.getValueAsString().equals(name);
}
```

Listing 4.16: Funktion zum Vergleich eines Wertes innerhalb der „Properties“

Die beschriebenen Probleme während der Testerstellung mit TestComplete erzeugten nicht nur einen höheren Entwicklungsaufwand, sondern bewirkten auch die in Tabelle 4.4 dargestellten, häufigen Reparaturen nach der Implementierung. Teilweise mussten dazu von mir für funktionierende Tests sogar neue Funktionen entwickelt werden, um die geplante Funktionalität zu garantieren.

4.3.3.7 Fehlschläge von TESTONA 4.1 zu 4.3

Beim Versionswechsel von TESTONA 4.1 auf TESTONA 4.3 sind drei von vier Tests erwartet fehlgeschlagen.

SeachingElements: Die Suche nach Tags lieferte unter der neuen Version des SUTs mehr Ergebnisse, so dass der Check fehlschlug. Den von mir erwarteten Fehlschlag konnte ich durch ein Update des Checks zügig beheben.

ExportExcel: Durch eine neue Möglichkeit zum Export ist der Check für zwei bestimmte Export-Varianten unerwartet fehlgeschlagen, da sich diese nicht mehr an ihrer ursprünglichen Position befanden. Um das Problem zu beheben, musste ich lediglich den entsprechenden Vergleich aktualisieren.

ExportHTML: Der von mir erwartete Fehlschlag des Tests aufgrund der neuen Varianten für den Export entstand, da die Suche nach dem Buchstaben „c“ nun mehr Ergebnisse lieferte. Um die Funktionalität wiederherzustellen, musste ich lediglich zwei Checks anpassen.

Import: Die für diesen Test erwartete Fehlerursache ist nicht eingetreten. Allerdings wurde die Datei nicht schnell genug importiert, so dass die folgenden Vergleiche dennoch fehlschlugen. Zur Reparatur des Tests genügte eine Erhöhung der Wartezeit vor den Vergleichen von 500 auf 2500 Millisekunden.

4.3.3.8 Fehlschläge von TESTONA 4.3 zu 4.5

Beim Update von TESTONA 4.3 auf Version 4.5 sind neben acht erwarteten auch zwei zusätzliche Tests fehlgeschlagen.

ChangeMarktypes: Der Test ist aufgrund eines nicht mehr existierenden Dialogs fehlgeschlagen und konnte von mir durch das Entfernen der dazugehörenden Interaktionen zügig repariert werden. Problematisch war zudem ein von TestComplete fälschlich akzeptierter Check, obwohl sich die Symbole in der zu überprüfenden Testmatrix von TESTONA vollständig verändert hatten. Dementsprechend habe ich den davon betroffenen Check nachträglich angepasst.

GenerateTestcases: Eine neu eingeblendete Warnung für nicht generierte Testfälle konnte von mir mit einer zusätzlichen Interaktion behoben werden. Aufgrund des verbesserten Algorithmus zur Erstellung der „Default Minimal Combination“ werden zudem weniger Testfälle erstellt, so dass zwei Checks von mir angepasst werden mussten. Außerdem schlug ein Vergleich der voreingestellten Werte fälschlicherweise nicht fehl, obwohl sich zwei unterschiedliche Werte in einem Textfeld befanden, so dass ich die neue Ansicht aktualisieren musste.

RenameGroupsAndSequences: Der Menüpunkt `AdvancedRename` befand sich nicht mehr direkt im Rechtsklick-Menü, sondern im Untermenü `Edit`, so dass die Interaktion von mir neu aufgenommen und an allen betroffenen Stellen angepasst werden musste.

VariantenManagementView: Wie bei QF-Test wurde der Button zur Erstellung einer neuen Variante nicht gefunden und musste von mir erneut aufgenommen werden. Der zweite Fehlschlag entstand durch einen unpassenden Check nach der Erzeugung einer neuen Variante, so dass ich diesen neu aufgenommen und in einen Table-Checkpoint umgewandelt habe. Anschließende Probleme mit dem Namemapping der dazugehörenden GUI-Komponente erforderten eine weitere Anpassung meinerseits. Zudem schlug im weiteren Verlauf des Tests ein anderer Check fehl, da TestComplete nicht die notwendigen Informationen einer für den Check relevanten Komponente erhielt, so dass ich mit einem Klick in die Komponente vor der Durchführung des Checks den Fokus neu setzen musste. Der letzte Fehlschlag in diesem Testfall bestand aus einem nicht durchführbaren Klick auf eine „ComboBox“ innerhalb der Toolbar, der durch eine nicht eindeutige Zuordnung ausgelöst wurde und nach einigen Anpassungen im Namemapping von mir repariert werden konnte.

SearchingElements: Der erste Vergleich innerhalb des Tests schlug fehl, da TestComplete keinen Zugriff mehr auf eine benötigte Komponente besaß. Zur Behebung des Fehlers habe ich vor dem Vergleich eine passende Interaktion in den Baum eingefügt. Zudem ist ein anderer Vergleich, obwohl vier Elemente nicht mehr gefunden wurden, fälschlicherweise nicht fehlgeschlagen, so dass ich für den Vergleich eine geringere Toleranz einstellen und eine zusätzliche Maske erstellen musste.

DependencyRulesLogical: Der zusätzliche Dialog zur Benennung einer neuen Regel ist entfernt worden, so dass der Name nun direkt im Dialog „DependencyRules“ eingegeben werden musste. Mein Versuch das Problem über eine Aufnahme vom Eintippen des neuen Namens zu beheben schlug fehl, da statt der Shell des bereits vorhandenen Dialogs von TestComplete jeweils eine neue Shell erstellt wurde. Daraufhin habe ich über den Objektbrowser die neue Shell ermittelt und das Mapping des Textfelds in die andere Shell manuell durchgeführt. Der aufgenommene Befehl „SetText“ genügte in diesem Fall nicht, so dass zuvor noch der voreingestellte Inhalt mit Hilfe der Tastenkombination CTRL+A markiert und anschließend gelöscht werden musste. Da zwei Checks die Komponente zum Vergleich anschließend nicht identifizieren konnten, mussten diese von mir nachträglich angepasst werden.

DependencyRulesNumeric: Die neu eingeblendete Warnung für nicht erstellte Tests konnte von mir mit einem zusätzlichen Klick behoben werden. Die Reparatur des ebenfalls fehlenden Dialogs zur Eingabe des Namens bei der Erstellung einer neuen Regel erfolgte in Anlehnung an den vorhergehenden Testfall.

ExportExcel: Durch den geänderten Dialog zum Export war der Klick auf das Element „TestcaseSpecification“ nicht mehr möglich und musste von mir neu aufgenommen werden.

ExportHTML: Der neue Dialog zum Export beinhaltete statt Listenelementen nun Checkboxen, so dass die Einstellung „Description“ nicht mehr ausgewählt werden konnte und der Klick auf die Checkbox von mir aktualisiert werden musste. Zudem ist der Vergleich des Ergebnisses bei der Suche nach dem Buchstaben „c“ innerhalb des Exportdialogs fehlgeschlagen, da die Suche nun mehr Ergebnisse lieferte. Der Check musste von mir lediglich neu aufgenommen und aktualisiert werden.

Import: Änderungen in der Struktur des Dialogs „Import“ verhinderten die Auswahl von Elementen, die sich zuvor in einer Liste befanden, da sie nun in einer Baumstruktur angeordnet wurden. Zur Problembehebung mussten von mir die Elemente neu aufgenommen, das Namemapping angepasst und die korrekte Interaktion in den Baum eingefügt werden. Die beiden Checks der fertig importierten Datei habe ich danach erneuert, da sich die Namen der Baumelemente verändert hatten. Zudem habe ich den Check zur Überprüfung der Importvarianten aktualisiert, da für diesen eine neue Möglichkeit hinzugefügt wurde.

4.3.3.9 Jenkins Ergebnisse

Die fertig implementierten Tests wurden auch für TestComplete mit Hilfe von Jenkins automatisiert ausgeführt. Von sieben Durchläufen bei einer durchschnittlichen Dauer von 18 Minuten schlug dabei unter TESTONA 4.1 nur der Test `CreateNewTree` einmalig, ohne erkennbaren Grund fehl. Die reparierten Tests für TESTONA 4.3 benötigten dann für 19 Durchläufe durchschnittlich 19 Minuten und lieferten jeweils einen Fehlschlag bei den Tests `StartTestona` und `Help` sowie drei Fehlschläge für den Test `CreateNewTree`. Nach dem zehnten Durchgang habe ich daher im Test `CreateNewTree` zusätzliche Wartezeiten eingebaut, so dass dieser in den darauffolgenden neun Durchläufen nur noch einmal fehlschlug. Die Ausführungen der Tests unter TESTONA 4.5 benötigten ebenfalls durchschnittlich 19 Minuten und führten erneut zum Fehlschlagen des Tests `CreateNewTree`. Dieser lieferte unter Version 4.5 zwei direkt aufeinanderfolgende Fehlschläge, lief in den folgenden Durchgängen aber problemlos durch, so dass bei der Ursache wieder von zu kurzen Wartezeiten ausgegangen werden kann.

5 Auswertung

5.1 Gegenüberstellung der Tools

Im Folgenden werden die aufgezeichneten Daten aus Kapitel 4.3 „Durchführung mit den einzelnen Tools“ ausgewertet. In Tabelle 5.1 ist dazu jeweils die benötigte Zeit zum Implementieren und Warten der Tests zusammenfassend dargestellt. Die erste Spalte enthält die Namen aller 26 Tests einschließlich der unter „Zusatztests“ gebündelten sonstige Testfälle, wie beispielsweise „FileOpen“ und „StartTestona“. In der zweiten Spalte „4.1“ ist der für die Erstellung der Tests aufgebrachte Arbeitsaufwand in Minuten zusammengefasst. Die beiden Capture & Replay-Tools QF-Test und TestComplete besitzen darin jeweils zwei Spalten, da ihre Tests teilweise mit zusätzlichem Zeitaufwand repariert werden mussten. Die dritte und vierte Hauptspalte geben den Arbeitsaufwand zur Reparatur fehlgeschlagener Tests mit den jeweiligen Tools beim Versionswechsel des SUTs an. Das in einigen Spalten vorkommende Zeichen „↯“ symbolisiert einen für diesen Test erwarteten Fehlschlag (vgl. Kapitel 4.1 „Erwartungen“).

5.1.1 Implementierung

Für jedes der drei untersuchten Capture & Replay-Tools spiegelt die Reihenfolge in der Tabelle den Zeitpunkt der Entwicklung wider (vgl. Tabelle 5.1, erste Spalte). Die jeweils obenstehenden Tests der verschiedenen Tools besitzen daher oft eine höhere Entwicklungszeit, da sie von mir zu Beginn der Arbeit mit einem Tool entwickelt wurden und ich zu diesem Zeitpunkt noch nicht vollständig in das jeweilige Tool eingearbeitet war (vgl. Tabelle 5.1, zweite Spalte). Abbildung 5.1 visualisiert darüber hinaus die jeweils von mir benötigte Zeit für die Entwicklung der Tests jedes einzelnen Capture & Replay-Tools in Form eines Boxplots.

Für die Entwicklung der 26 Tests mit dem ersten Capture & Replay-Tool Eggplant benötigte ich 2750 Minuten (45h 50min). Da keiner der entwickelten Tests repariert werden musste, stellt dies zudem auch dessen Gesamtentwicklungszeit dar. Die durchschnittliche Zeit zur Implementierung eines Tests liegt demnach bei 105 Minuten und 20 Sekunden (1h 45min 20s). Im Verlauf der Testimplementierung beschleunigte sich die Entwicklung der Tests zunehmend, da ich eine bessere Einarbeitung erreichte und eine stetig wachsende, selbst erstellte Menge an Grafiken

	4.1					4.3			4.5		
	Egg	QF		TC		Egg	QF	TC	Egg	QF	TC
CreateNewTree	180	60		150	+80					20	
EditTree	180	90		130	+30				10		
FoldTree	120	60		240						5	
CopyPasteOfSubtrees	120	45		20	+195						
WriteProtectedFiles	60	30		30							
PropertiesView	120	30		240							
Autolayout	150	30		70							
Outline1	60	30		70	+20						
Outline2	90	60		40							
CreateRenameTests	120	20	+20	45							
ChangeMarktypes	240	30		40					↯ 60	10	25
GenerateTestcases	120	30		40					↯ 25	45	40
ManualTestcases	120	105		45							
RenameGroups...	120	60		40					↯ 5	25	15
TestSequences	90	15		40					10		
BoundaryValues	60	20		30	+40						
CreateEditTags	90	40		50			20				
VariantsView	90	45		25						5	80
Help	30	320	+15	30	+100				5	5	
SearchingElements	120	40	+10	30	+130	↯ 5	20	10	↯ 20	20	40
Logical Dep. Rules	60	20		15					↯ 20	10	60
Numeric Dep. Rules	60	50		35					↯ 10	20	30
ExportExcel	60	15	+20	20	+90	10		15	↯ 5	10	10
ExportHTML	60	20		20		↯ 15	10	20	↯	35	25
Import	60	15	+20	30	+130	↯		20	10	15	45
Testcoverage	60	15		15	+30						
Zusatztests	110	165		400	+70						
TOTAL	2750	1460	+85	1740	+955	30	50	65	180	225	370

Tabelle 5.1: Implementierungs- und Wartungsaufwand (Zeit in Minuten)

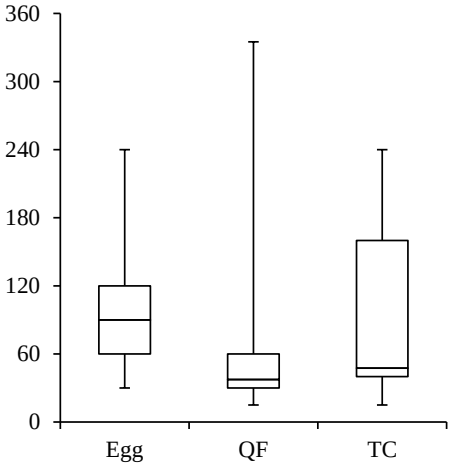

Abbildung 5.1: Implementierungsaufwand (Zeit in Minuten [min])

wiederverwenden konnte. Während die Tests zu Beginn teilweise mehr als 120 Minuten benötigten, konnte ich sie zum Ende jeweils in ungefähr einer Stunde entwickeln. Dies spiegelt sich auch in der exakt gleichmäßig aufgeteilten Box des unter Abbildung 5.1 dargestellten Boxplots für Eggplant wieder und wird durch den bei 90 Minuten in der Mitte der Box liegenden Median bestätigt. Die Entwicklung des Tests `Help`

verlief dabei mit 30 Minuten am schnellsten, während der Test `ChangeMarkTypes` mit 240 Minuten einen Ausreißer darstellt und die Spannweite des Boxplots enorm vergrößert.

Neben einer Entwicklungszeit von 1460 Minuten (24h 20min) für alle 26 Tests benötigte ich mit dem zweiten Capture & Replay-Tool QF-Test noch zusätzliche 95 Minuten (1h 15min) zur Reparatur der entwickelten Tests. Die durchschnittliche Entwicklungszeit beträgt damit inklusive Reparatur 59 Minuten und 25 Sekunden. Mit nur 15 Minuten Entwicklungszeit stehen die Testfälle `TestSequences` und `Testcoverage` in deutlichem Gegensatz zum Test `Help`, für dessen Entwicklung ich ganze 335 Minuten (5h 35min) benötigte. Diese enorme Spannweite ist auch im Boxplot von QF-Test deutlich sichtbar, steht aber nicht repräsentativ für die anderen Tests, was sowohl am sehr niedrigen Median von 37,5 Minuten, als auch an der allgemein geringen Höhe der Box zu erkennen ist (vgl. Abbildung 5.1).

Für die Entwicklung der 26 Tests mit dem dritten Capture & Replay-Tool TestComplete benötigte ich zunächst lediglich 1740 Minuten (29h). Dem folgten jedoch zusätzliche 955 Minuten (15h 55min) zur Reparatur der fehlgeschlagenen Tests. Daraus ergibt sich eine durchschnittliche Entwicklungszeit (inkl. Reparatur) von 103 Minuten und 39 Sekunden (1h 43min 39s). Mit lediglich 15 Minuten Entwicklungszeit konnte ich dabei den Test `DependencyRulesLogical` implementieren, während ich für die im Anschluss fehlerfreien Tests `FoldTree` und `PropertiesView` mit 240 Minuten (4h) am meisten Entwicklungszeit benötigte. Unter den Tests die eine Reparatur erforderten, stellte der Test `CopyPasteOfSubtrees` mit nur 20 Minuten Entwicklungszeit und anschließenden 195 Minuten (3h 15m) Reparatur den größten Anpassungsaufwand dar. Aufgrund der hohen Reparaturzeiten befindet sich der Boxplot von TestComplete im Bereich zwischen 40 und 160 Minuten, allerdings mit einem vergleichsweise niedrigen Median von 47,5 Minuten (vgl. Abbildung 5.1). Dies ist auf das schnelle und problemlose Aufzeichnen von rund der Hälfte aller Testfälle zurückzuführen.

Ein Vergleich der einzelnen Capture & Replay-Tools im Hinblick auf die Implementierung zeigt deutliche Unterschiede (vgl. Abbildung 5.1). QF-Test benötigte mit einer Gesamtentwicklungszeit von 1555 Minuten (25h 55m) und einer durchschnittlichen Entwicklungszeit von 59 Minuten und zwei Sekunden pro Test den geringsten Zeitaufwand. Obwohl einige Tests von mir repariert werden mussten, erforderte QF-Test damit nur ungefähr die Hälfte der Entwicklungszeit im Vergleich zu den anderen Tools. TestComplete befindet sich mit 2695 Minuten (44h 55m) für die gesamte Entwicklungszeit sowie einem Durchschnitt von 103 Minuten und 39 Sekunden (1h 43m 39s) zur Erstellung eines Testfalls deutlich hinter QF-Test und nahe am Drittplatzierten Eggplant. Durch das unter TestComplete verwendete Namemapping in Kombination mit der verschachtelten GUI des SUTs waren die Tests sehr fehleranfällig und benötigten fast die Hälfte der Entwicklungszeit zur Repa-

ratur. Die höchste Gesamtentwicklungszeit erforderte das Capture & Replay-Tool Eggplant mit 2750 Minuten (45h 50min) und einer durchschnittlichen Zeit von 105 Minuten und 20 Sekunden (1h 45min 20s) pro Test. Im Gegensatz zu den beiden anderen Tools musste ich dabei allerdings keinen der implementierten Tests reparieren.

5.1.2 Robustheit

Im folgenden Kapitel werden die 26 entwickelten Tests für die drei untersuchten Capture & Replay-Tools hinsichtlich ihrer Robustheit, d.h. wie viele von ihnen beim Versionswechsel des SUTs TESTONA intakt blieben, ausgewertet. Bei der Änderung des SUTs von Version 4.1 zu 4.3 wurden dabei von mir drei und beim Wechsel von Version 4.3 zu 4.5 acht Fehlschläge erwartet, die in Tabelle 5.1 mit dem Symbol „↯“ gekennzeichnet und in Kapitel 4.1 „Erwartungen“ näher beschrieben sind.

Beim Versionswechsel von TESTONA 4.1 zu 4.3 ist in Eggplant neben den beiden erwarteten Tests `SearchingElements` und `ExportHTML` zudem auch der Test `ExportExcel` fehlgeschlagen. Die Ursache ist dabei äquivalent zum fehlgeschlagenen Test `ExportHTML`: Ein Textfeld hatte sich optisch etwas verschoben. Dieser im Nachhinein für mich einfach zu reparierende Fehlschlag hätte bei beiden Tests durch eine andere Platzierung des Fokuspunkts verhindert werden können. Mit den Änderungen aus dem Test `Import` konnte Eggplant hingegen gut umgehen, so dass dieser Test intakt blieb. Beim Wechsel von Version 4.3 zu TESTONA Version 4.5 sind sieben von elf fehlgeschlagenen Tests von mir erwartet worden. Neben den zusätzlich fehlgeschlagenen Tests `Help`, `TestSequences` und `Import` wurde auch der Test `EditTree` von mir repariert, da er mehrfach nicht korrekt durchlief.

Die Ergebnisse für QF-Test beim Wechsel von Version 4.1 zu Version 4.3 ähneln denen von Eggplant. Die beiden Tests `SearchingElements` und `ExportHTML` sind durch Änderungen an der GUI fehlgeschlagen, der Test `Import` lief entgegen den Erwartungen fehlerfrei durch. Dagegen ist der Test `CreateEditTags` unerwartet fehlgeschlagen, obwohl bei seiner Ausführung keine optischen Änderungen an der GUI vorgenommen wurden. Beim zweiten Versionswechsel von TESTONA 4.3 auf 4.5 wurden von den insgesamt 13 Fehlschlägen nur acht zuvor von mir erwartet. Neben den unerwartet fehlgeschlagenen Tests `FoldTree`, `VariantenManagementView`, `Help` und `Import` habe ich den Test `CreateNewTree` stellvertretend als fehlgeschlagen gewertet, da durch eine kleine Änderung in der GUI des SUTs nicht mehr mit dem Editor von TESTONA interagiert werden konnte und somit eine Anpassung der Komponenten außerhalb der Testfälle notwendig wurde.

Neben den drei erwarteten Tests `SearchingElements`, `ExportHTML` und `Import` ist unter TestComplete beim Versionswechsel von TESTONA 4.1 zu TESTONA 4.3 der Test `ExportExcel` unerwartet fehlgeschlagen. Beim darauffolgenden Wechsel der

Version von 4.3 zu 4.5 schlugen mit zehn Testfällen im Vergleich zu den anderen Tools am wenigsten Tests fehl. Mit Ausnahme der beiden Testfälle `VariantenManagement-View` und `Import` wurden alle Fehlschläge von mir im Voraus erwartet. Der Test `Help` hätte, insbesondere im Vergleich mit den Ergebnissen der anderen Capture & Replay-Tools, ebenfalls fehlschlagen müssen, da Änderungen in der Anzeige eines Suchergebnisses verschiedene Darstellungen der GUI erzeugten. TestComplete ignorierte diesen Fehler jedoch einfach, so dass der Test von mir nicht als fehlgeschlagen gewertet werden konnte.

In Eggplant sind elf Tests fehlgeschlagen von denen 7 erwartet wurden, zwei sind zusätzlich fehlgeschlagen (`Help`, `Import`) und ein Test wurde auch ohne Fehlschlag verbessert (`EditTree`), da er häufiger nicht korrekt durchlief.

Ein Vergleich zwischen den drei untersuchten Capture & Replay-Tools im Hinblick auf die Robustheit zeigt keine signifikanten Unterschiede: Eggplant und TestComplete besitzen jeweils 14 Fehlschläge, während QF-Test mit 16 Fehlschlägen nicht wesentlich abweicht.

5.1.3 Wartungsaufwand

In diesem Kapitel erfolgt der Vergleich des Wartungsaufwands der beim Versionswechsel des SUTs fehlgeschlagenen Tests für die untersuchten Capture & Replay-Tools. Die am Anfang von Kapitel 5.1 dargestellte Tabelle 5.1 zeigt dazu in ihrer dritten und vierten Spalte die jeweils konkreten Reparaturzeiten für alle fehlgeschlagenen Testfälle. Abbildung 5.2 verdeutlicht zudem den jeweiligen Wartungsaufwand der untersuchten Capture & Replay-Tools in Form eines Boxplots.

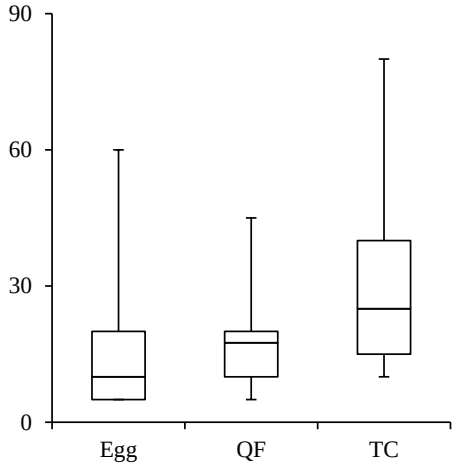

Abbildung 5.2: Wartungsaufwand (Zeit in Minuten [min])

Der gesamte Wartungsaufwand der 14 fehlgeschlagenen Tests unter Eggplant beträgt 210 Minuten, was einem durchschnittlichen Wartungsaufwand von 15 Minuten pro Testfall entspricht. Für die Anpassung der drei fehlgeschlagenen Testfälle

beim ersten Versionswechsel von TESTONA 4.1 auf Version 4.3 benötigte ich dabei 30 Minuten, während der Wartungsaufwand beim zweiten Versionswechsel von TESTONA 4.3 auf 4.5 für elf fehlgeschlagene Testfälle 180 Minuten betrug (vgl. Tabelle 5.1). Dabei bilden gleich vier fehlgeschlagene Testfälle (`SearchingElements`, `RenameGroupsAndSequences`, `Help` und `ExportExcel`) mit jeweils fünf Minuten das Minimum, so dass das Minimum dem unteren Quartil des Boxplots entspricht und einen niedrigen Median von lediglich 10 Minuten zur Folge hat.

Der Wartungsaufwand zur Reparatur der Tests beim Versionswechsel von TESTONA 4.1 zu TESTONA 4.3 betrug für QF-Test 50 Minuten bei drei fehlgeschlagenen Tests und 225 Minuten (3h 45min) für 13 fehlgeschlagene Tests beim zweiten Versionswechsel. Insgesamt ist damit bei 16 fehlgeschlagenen Tests ein Wartungsaufwand von 275 Minuten (4h 35min) entstanden, was einem Durchschnitt von 17m 11s pro Test entspricht. Dabei benötigte der fehlgeschlagene Test `GenerateTestcases` mit 45 Minuten zwar am meisten Reparaturzeit, erzeugt aber dennoch mit lediglich 40 Minuten eine niedrige Spannweite für den Boxplot von QF-Test (vgl. Abbildung 5.2).

Mit TestComplete dauerte die Reparatur der Tests beim Versionswechsel von TESTONA 4.1 auf 4.3 für vier fehlgeschlagene Tests 65 Minuten. Bei der Änderung des SUTs von TESTONA 4.3 auf TESTONA 4.5 schlugen dann zehn Testfälle fehl, für deren Reparatur ich 370 Minuten (6h 10m) benötigte. Damit ergibt sich für die 14 fehlgeschlagenen Tests ein Wartungsaufwand von insgesamt 435 Minuten (7h 15min). Der daraus resultierende Durchschnittswert von 31 Minuten und 4 Sekunden pro Test spiegelt sich auch im bei 25 Minuten liegenden Median des Boxplots wider (vgl. Abbildung 5.2). Besonders auffällig ist hier auch die Größe der sich von 15 bis 40 Minuten erstreckenden Box, die die insgesamt vergleichsweise hohen Reparaturzeiten beim Arbeiten mit TestComplete verdeutlicht. Den mit Abstand höchsten Reparaturaufwand von 80 Minuten besitzt dabei der fehlgeschlagene Test `VariantenManagementView`, der gleich mehrere Anpassungen meinerseits benötigte.

Ein Vergleich der einzelnen Capture & Replay-Tools im Hinblick auf den Wartungsaufwand zeigt deutliche Unterschiede: Mit Eggplant benötigte ich für die Reparatur der fehlgeschlagenen Tests mit durchschnittlich 15 Minuten pro Test am wenigsten Zeit, dicht gefolgt von QF-Test mit 17 Minuten und 11 Sekunden. Im Hinblick auf den Gesamtwartungsaufwand unterscheiden sich die beiden Tools allerdings um eine Stunde zugunsten von Eggplant. Mit insgesamt fast drei Stunden mehr Reparaturzeit und einem Durchschnittswert von 31 Minuten und 4 Sekunden pro Test benötigte TestComplete damit beinahe den doppelten Wartungsaufwand im Vergleich zu den anderen Capture & Replay-Tools. Die Gründe dafür sind nicht eindeutig identifizierbar, ein ausschlaggebender Faktor stellt allerdings das neben der eigentlichen Reparatur vorzunehmende Namemapping der GUI-Komponenten dar.

5.1.4 Zusammenfassung

Aus den vorherigen Kapiteln 5.1.1 „Implementierung“, 5.1.2 „Robustheit“ und 5.1.3 „Wartungsaufwand“ lassen sich die folgenden drei Erkenntnisse zusammenfassen:

- Die Implementierung der Tests verlief mit QF-Test am schnellsten, Eggplant und TestComplete benötigten deutlich mehr Zeit.
- Bezüglich der Robustheit der Tests gibt es keine signifikanten Unterschiede zwischen den verschiedenen Capture & Replay-Tools.
- Der Wartungsaufwand ist bei Eggplant eindeutig am niedrigsten. Während QF-Test noch einen akzeptablen Aufwand besitzt, benötigt TestComplete deutlich mehr Zeit für die Wartung.

Mit Rückblick auf die Abbildungen 5.1 und 5.2 kann errechnet werden, dass das in der Implementierung exzellente QF-Test nach ca. 580 reparierten Tests mit dem in der Wartung hervorragenden Eggplant gleichziehen würde. Diese eher langfristige Ergiebigkeit von Eggplant deckt sich mit den Erkenntnissen von Alégroth et al. [AFK16]. Aufgrund der im Vergleich mit den anderen beiden Capture & Replay-Tools eher schwachen Leistung von TestComplete würde ich dessen Verwendung nicht empfehlen.

5.2 Ticketanalyse

In diesem Kapitel erfolgt der Vergleich der drei untersuchten Capture & Replay-Tools im Hinblick auf eine Anfälligkeit für bestimmte Kategorien von Veränderungen am SUT. Die Veränderungen am SUT TESTONA erfolgen zwischen dessen Versionen in Form von 22 bzw. 53 Tickets (vgl. Tabelle 3.1). Diese habe ich für eine Kategorisierung in die Gruppen „Back-End“, „Statisch“ und „Dynamisch“ unterteilt: Tickets aus der Kategorie „Back-End“ weisen lediglich Änderungen im Quellcode auf und haben keine Auswirkungen auf die GUI, wozu beispielsweise Verbesserungen an der Performance von TESTONA zählen. Zur Kategorie „Statisch“ gehören sämtliche Änderungen an der grafischen Oberfläche, wie beispielsweise die Überarbeitung eines Dialogs oder eines neuen Eintrags im Menü. Die dritte Kategorie „Dynamisch“ umfasst alle Änderungen an Algorithmen, die bei der Nutzung zu unterschiedlichen Ergebnissen führen können. Dazu gehört beispielsweise die Suche nach Elementen oder die Generierung von Testfällen.

Tabelle 5.2 verdeutlicht meine Einordnung der 22 bearbeiteten Tickets zwischen den TESTONA-Versionen 4.1 und 4.3 im Hinblick auf die zuvor definierten Kategorien. Mit Hilfe der für jeden fehlgeschlagenen Test geführten Protokolle konnte ich jeweils eine oder mehrere Ursachen für den Fehlschlag analysieren und auf die

	Mit Tests abgedeckt	Tests existieren nicht
Back-End	2	4
Statisch	5	6
Dynamisch	2	3

Tabelle 5.2: Bearbeitete Tickets zwischen TESTONA 4.1 und 4.3

verschiedenen Tickets zurückführen. Teilweise existierten in den Tests mehrere Ursachen, die nicht auf Tickets zurückgeführt werden konnten, so dass diese von mir in die Kategorie „Fehlschlag“ oder „Toolfehler“ eingeteilt wurden. Da die Tickets ohne Test zu keinem Fehlschlag führen konnten, sind in den Tabellen 5.3, 5.4 und 5.5 jeweils nur mit einem Test abgedeckte Tickets berücksichtigt.

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket
Back-End		2	
Statisch		5	2
Dynamisch	1	1	

Tabelle 5.3: Analyse der Tickets von 4.1 zu 4.3 in Eggplant

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket
Back-End		2	
Statisch	1	4	1
Dynamisch	1	1	1

Tabelle 5.4: Analyse der Tickets von 4.1 zu 4.3 in QF-Test

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket
Back-End		2	
Statisch	2	4	
Dynamisch	1	1	1

Tabelle 5.5: Analyse der Tickets von 4.1 zu 4.3 in TestComplete

Aufgrund der niedrigen Anzahl an Tickets mit abgedeckten Testfällen ist eine ausführliche Analyse der Tabellen kaum möglich. Auffällig ist jedoch, dass alle Tests für das Back-End bei keinem Capture & Replay-Tool Fehlschläge verzeichnen. Im Hinblick auf die Tests aus der Kategorie„Statisch“ schneiden alle Tools mit insgesamt zwei Fehlschlägen gleichwertig ab: Während bei Eggplant alle Tests durchliefen, traten zwei Fehlschläge ohne Ticket auf. Bei QF-Test reduziert sich diese Anzahl auf nur einen Fehlschlag ohne Ticket, dafür existiert ein regulärer Fehlschlag. TestComplete weist zwar keine Fehlschläge ohne Ticket auf, dafür besitzt es aber zwei reguläre Fehlschläge. In der Kategorie „Dynamisch“ liegen die drei Capture & Replay-Tools mit jeweils einem fehlgeschlagenen und einem durchgelaufenen Test noch näher bei-

einander, lediglich QF-Test und TestComplete besitzen jeweils noch einen Fehlschlag ohne Ticket.

	Mit Tests abgedeckt	Tests existieren nicht
Back-End	3	17
Statisch	15	12
Dynamisch	2	4

Tabelle 5.6: Bearbeitete Tickets zwischen TESTONA 4.3 und 4.5

Tabelle 5.6 verdeutlicht meine Einordung der 53 bearbeiteten Tickets zwischen den TESTONA-Versionen 4.3 und 4.5 im Hinblick auf die zuvor definierten Kategorien. Auch in diesem Abschnitt habe ich für jeden fehlgeschlagenen Test ein Protokoll geführt und die Ursache auf ein Ticket bzw. einen Toolfehler zurückgeführt. Zu den Toolfehlern zählt beispielsweise der Fehlschlag eines Vergleichs, obwohl keine Änderungen vorgenommen wurden. In den Tabellen 5.7, 5.8 und 5.9 sind die mit einem Test abgedeckten Tickets für jedes der verschiedenen Capture & Replay-Tools kategorisiert. Wenn ein Ticket in zwei Testfällen zu einem Fehlschlag führte, habe ich es nur einmal gezählt und ein weiteres Auftreten dahinter in Klammern angegeben.

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket	Toolfehler
Back-End		3		
Statisch	6(2)	9	2	
Dynamisch	1	1	1	

Tabelle 5.7: Analyse der Tickets von 4.3 zu 4.5 in Eggplant

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket	Toolfehler
Back-End		3		
Statisch	9(2)	6	1	2
Dynamisch	1	1	1	

Tabelle 5.8: Analyse der Tickets von 4.3 zu 4.5 in QF-Test

	Fehlschlag	Durchgelaufen	Fehlschlag o. Ticket	Toolfehler
Back-End		3		
Statisch	9(2)	6	2	2
Dynamisch	1	1		

Tabelle 5.9: Analyse der Tickets von 4.3 zu 4.5 in TestComplete

Die Tabellen beinhalten erneut zu wenig Daten um deutliche Ergebnisse zu liefern. Auffällig ist jedoch, dass die beiden Capture & Replay-Tools der zweiten Generation im Hinblick auf fehlgeschlagene und durchgelaufene Tickets das gleiche Ergebnis liefern. Eggplant konnte auch in diesem Versionswechsel des SUTs hinsichtlich der

statischen Änderungen besser reagieren und wies keine Toolfehler auf, allerdings ist dessen Anzahl an Fehlschlägen ohne Test bei höher.

Durch die geringe Anzahl an mit Tests abgedeckten Tickets kann keine spezifische Anfälligkeit für bestimmte Kategorien von Veränderungen am SUT für eines der drei untersuchten Capture & Replay-Tools festgestellt werden. Somit ist auch ein eventuell besserer Umgang mit statischen Änderungen durch Eggplant im Vergleich zu den beiden anderen Tools nicht eindeutig belegbar. Lediglich im Hinblick auf Veränderungen im Back-End des SUTs sind, wie zuvor von mir erwartet, bei keinem Capture & Replay-Tool Fehlschläge aufgetreten.

5.3 Grenzen der Arbeit

In diesem Kapitel erfolgt die kritische Betrachtung der Gültigkeit der von mir ermittelten Erkenntnisse im Hinblick auf den durchgeführten Vergleich der drei ausgewählten Capture & Replay-Tools unter Verwendung der von Runeson und Höst [RH09] beschriebenen Validitätskriterien: Konstruktvalidität, interne Validität und externe Validität.

Dabei fragt die **Konstruktvalidität** danach, ob im Rahmen des durchgeführten Vergleichs der Capture & Replay-Tools bezüglich der aufgestellten Untersuchungskriterien (Implementierung, Robustheit und Wartungsaufwand) eine objektive Untersuchung stattgefunden hat. Aufgrund der von außen festgelegten Rahmenbedingungen dieser Arbeit konnte die Aufzeichnung, der die Untersuchungskriterien stützenden Messwerte, nur durch mich als einzelnen Tester erfolgen. Dabei könnten beispielsweise psychologische Faktoren die Testreparatur und damit die Messungen unbemerkt beeinflusst haben. Um die Messungen selbst so exakt wie möglich zu halten, habe ich neben einer Dokumentation in einem Arbeitstagebuch eine Überprüfung der SVN-Commits vorgenommen.

Die **interne Validität** prüft, in welchem Maß berücksichtigte wie unberücksichtigte Einflussfaktoren die durchgeführten Schlussfolgerungen beeinflusst haben. Im Rahmen meiner Arbeit ist der einzig variable Einflussfaktor für die aufgestellten Untersuchungskriterien der Entwurf der Testfälle, da die verwendeten Capture & Replay-Tools nicht durch mich verändert werden konnten. Dabei hängt die Qualität der implementierten Testfälle auch direkt von meiner Testerfahrung ab. So habe ich zwar in den letzten Jahren Erfahrungen im Testen von GUIs sammeln können, jedoch nicht die Möglichkeit bekommen Zertifikate zu erlangen. Trainingsprogramme und der Austausch mit anderen Testern führten zwar zur Vertiefung meiner Kenntnisse, was jedoch insbesondere im Hinblick auf eine Einarbeitung in die mir unbekannten Capture & Replay-Tools nur zu einer Reduzierung möglicher Einflüsse, nicht zu deren Beseitigung, beitragen konnte.

Abschließend hinterfragt die **externe Validität** die Gültigkeit der ermittelten Ergebnisse beim Vergleich der untersuchten Capture & Replay-Tools für eine weiterführende Forschung oder Anwendung in anderen Forschungsgebieten, d.h. ob die von mir ermittelten Ergebnisse generalisiert werden können. Dies ist gerade im Hinblick auf die Limitierung meiner Arbeit durch externe Rahmenbedingungen fraglich: Trotz des großen Spektrums an GUI-Elementen im verwendeten SUT TESTONA sollten beispielsweise die Capture & Replay-Tools unter Verwendung verschiedener SUTs, mit jeweils einer höheren Anzahl an Testfällen getestet und evaluiert werden. Zudem wäre eine Erhöhung der Anzahl zu untersuchender Capture & Replay-Tools für das Erreichen einer besseren Vergleichbarkeit sinnvoll. Im Hinblick auf die nur auszugsweise beschriebenen Limitierungen kann bestenfalls von einer wegbereitenden Arbeit für die weitere Forschung gesprochen werden.

6 Ausblick

Möglichkeiten einer Erweiterung bzw. Fortführung, der in dieser Arbeit durchgeführten Erhebungen, bietet eine Veränderung der Rahmenbedingungen. Dazu gehört zunächst der Einsatz von mehr Testern, die alle vergleichbare Tests mit den Tools entwickeln und im besten Fall unterschiedliche Erfahrungen im GUI-Testen besitzen. Eine größere Anzahl an Testern sowie zusätzliche Interviews würden einen besseren Einblick in den Umgang mit den verschiedenen Tools geben. Dadurch könnte beispielsweise besser nachvollzogen werden, ob bestimmte Entwicklungsprobleme bei mehreren Testern gleichermaßen auftreten. Darüber hinaus bietet eine Anpassung des SUTs zusätzliche Forschungsmöglichkeiten: Dabei kann zunächst TESTONA selbst, als das in dieser Arbeit verwendete SUT, in höherer Versionsanzahl betrachtet werden. Dies ist ggf. sogar mehrfach möglich, da die Ergebnisse der Studie in die kommende TESTONA-Generationen einfließen werden und somit die Effektivität von durchgeführten Veränderungen widerspiegeln könnten. Neben der Erweiterung der Versionenanzahl ist auch eine Ausweitung der Studie auf andere SUTs denkbar. Jedes SUT besitzt einen anderen Aufbau der GUI und ist unterschiedlich tief verschachtelt. In diesem Zusammenhang kann z.B. untersucht werden, ob weniger verschachtelte GUIs leichter zu testen sind. Auch haben andere Tools unterschiedliche Entwicklungssprünge zwischen ihren Release-Versionen, die zu mehr oder weniger Änderungen in der GUI führen und damit die Ergebnisse bezüglich der untersuchten Capture & Replay-Tools beeinflussen könnten. Im Hinblick auf die Capture & Replay-Tools bietet sich ebenfalls eine Erhöhung auf eine größere Untersuchungsmenge an, da jedes Capture & Replay-Tool eine andere Funktionsweise bzw. einem anderen Algorithmus für den Bildvergleich besitzt. Die so durchgeführte Erweiterung könnte zu einem klareren Ergebnis bezüglich der Einordnung eines Tools im Vergleich zu den anderen Testkandidaten führen.

Abgesehen von der Veränderung der Rahmenbedingungen bietet eine Erweiterung der entwickelten und durchgeführten Tests weitere Forschungsmöglichkeiten. So könnte beispielsweise die Tabelle 3.2 aus Kapitel 3.4 mit mehr Testfällen gefüllt werden. Durch eine größere Sammlung an Daten in der Tabelle könnten sich beispielsweise bestimmte Elemente der GUI als besonders anfällig für GUI-Tests herausstellen. Neben der Untersuchung auf eine spezielle Anfälligkeit, bleibt auch die Frage nach der Vollständigkeit der durchgeführten Tests im Hinblick auf die Elemente der GUI offen. Selbstverständlich konnte trotz sorgfältiger Planung der

durchgeführten Tests keine vollständige Abdeckung erzielt werden, da viele Elemente der GUI selbst nochmals Teilelemente besitzen, so dass eine enorme Menge an Testfällen entstehen würde. Dies könnte ggf. durch den in Kapitel 2.3.3 vorgestellten Ansatz von Memon et al. bezüglich der Abdeckungskriterien einer GUI [MSP01] gelöst werden. Eine auf diesem Ansatz basierende Arbeit müsste dann zunächst prüfen, wieviel Prozent der GUI mit den derzeitigen Tests abgedeckt wurden und diese dann erweitern bzw. vervollständigen. Die durch Abdeckungskriterien oder mittels erhöhter Tester-, SUT-, Tool- und Testanzahl ermittelten Forschungsergebnisse könnten dann konkretere Verbesserungsmöglichkeiten für Capture & Replay-Tools ermitteln und diese an Krietrien orientiert systematisieren. Dies würde im besten Fall nicht nur einen objektiven Vergleich zwischen verschiedenen Capture & Replay-Tools, sondern deren zugrundeliegenden Generationen (vgl. Kapitel 2.3.2) ermöglichen. Die dadurch gewonnenen Erkenntnisse hätten nicht nur signifikante Auswirkungen auf Weiterentwicklungen existierender Capture & Replay-Tools, sondern könnten eine neue, vierte Generation hervorbringen.

7 Fazit

Ziel meiner Masterarbeit war die Untersuchung der drei Capture & Replay-Tools Eggplant, QF-Test und TestComplete hinsichtlich ihrer Entwicklungs- und Wartungskosten. Dazu habe ich für das SUT TESTONA 26 Testfälle spezifiziert, die anschließend mit den drei Tools für Version 4.1 von mir implementiert wurden. Im Anschluss habe ich die Tests erst auf Version 4.3 und dann auf Version 4.5 von TESTONA ausgeführt, wobei jeweils zuvor die fehlgeschlagenen Tests auf die Ursache des Fehlschlags hin untersucht und anschließend die notwendigen Reparaturen vorgenommen wurden. Dabei führte ich eine detaillierte Dokumentation der jeweils aufgetretenen Herausforderungen, Testfehlschläge und des von mir benötigten Aufwands zur Implementierung und Wartung der Testfälle durch.

Ein Vergleich, hinsichtlich der für die Entwicklung der Testfälle benötigten Zeit, in den jeweiligen Capture & Replay-Tools weist deutliche Unterschiede auf. Meine Messungen ergaben für QF-Test mit 1460 Minuten die geringste Entwicklungszeit, gefolgt von TestComplete (2695 Minuten) und Eggplant (2750 Minuten), die beide deutlich mehr Zeit benötigten. Während bei Eggplant die vergleichsweise hohe Implementierungszeit von der neuen Aufnahmetechnik der dritten Generation von Capture & Replay-Tools verursacht wurde, stellte unter TestComplete das komplizierte und fehleranfällige Namemapping den ausschlaggebenden Faktor dar.

Für die 26 Testfälle habe ich dabei bereits vor der Implementierung Erwartungen hinsichtlich ihrer Robustheit beim Versionswechsel des SUTs TESTONA formuliert, die auf den bearbeiteten Tickets des Entwicklungsteams von TESTONA beruhen. Im Anschluss an das jeweilige Ausführen der Testfälle wurde deutlich, dass meine Erwartungen größtenteils zutreffend waren. Von elf erwarteten Fehlschlägen sind bei Eggplant über die gesamte Entwicklungszeit acht erwartete und vier unerwartete Tests fehlgeschlagen. Bei QF-Test schlugen zehn von elf erwarteten sowie sechs zusätzliche Testfälle fehl. Bei TestComplete wurden, neben drei zusätzlichen Fehlschlägen, alle elf fehlgeschlagenen Tests von mir korrekt vorhergesehen. Im Hinblick auf die Robustheit der Testfälle konnten daher keine signifikanten Unterschiede zwischen den verwendeten Capture & Replay-Tools ermittelt werden.

Beim Vergleich des Wartungsaufwands für beide Versionswechsel des SUTs liegt Eggplant mit 14 fehlgeschlagenen Testfällen und 210 Minuten benötigter Reparaturzeit deutlich vorn. Noch in einem akzeptablen Abstand zu diesem Ergebnis befindet sich QF-Test mit insgesamt 16 fehlgeschlagenen Testfällen und 275 Minuten War-

tungsaufwand. TestComplete schneidet mit insgesamt 14 fehlgeschlagenen Testfällen und einer Reparaturzeit von 435 Minuten im Vergleich eindeutig am schlechtesten ab. Erneut ist dabei das unter TestComplete verwendete Namemapping der ausschlaggebende Faktor für den hohen Zeitaufwand.

Beim Vergleich der Kosten für die Implementierung und den Wartungsaufwand unterliegt TestComplete damit eindeutig gegenüber QF-Test und Eggplant. Die niedrigen Entwicklungskosten und unbedeutend höheren Wartungskosten von QF-Test gleichen sich rechnerisch nach ca. 580 reparierten Tests mit dem im Hinblick auf die Implementierung zeitaufwendigen, aber hinsichtlich des Wartungsaufwands hervorragenden Eggplant aus. Aufgrund der signifikant höheren jährlichen Kosten für Support und Updates von Eggplant gegenüber QF-Test, ist zusammenfassend betrachtet damit nur bei einem langfristig geplanten Projekt mit mehreren verwendeten GUI-Technologien das Capture & Replay-Tools Eggplant zu empfehlen. Für alle mittelfristigen Projekte mit kleinerem Budget ist QF-Test die passendere Wahl.

Selbstverständlich sind die von mir ermittelten Ergebnisse dabei nicht frei von Einflussfaktoren, da beispielsweise die Aufzeichnung, der die Untersuchungskriterien stützenden Messwerte, ausschließlich durch mich als einzelnen Tester erfolgt ist. Dementsprechend könnten psychologische Faktoren oder meine bisherigen Testerfahrungen die Messungen unmerklich beeinflusst haben, so dass von mir durchgeführte Gegenmaßnahmen, wie eine vorher geplante Testreihenfolge und ausführliche Dokumentation, eventuell wirkende Einflussfaktoren lediglich reduzieren konnten. Dies gilt insbesondere im Hinblick auf die Beschränkung meiner Arbeit durch externe Rahmenbedingungen. So war es mir aufgrund der begrenzten Zeit nicht möglich, mehr als drei Capture & Replay-Tools auf einem einzigen SUT mit insgesamt 26 Testfällen zu vergleichen. Dabei verfügte das verwendete SUT auch lediglich über zwei relevante Versionssprünge, so dass beispielsweise eine langfristige Untersuchung des SUTs zur Einschätzung der Effektivität der durchgeführten Änderungen nicht möglich war.

Darüber hinaus existieren, insbesondere im Hinblick auf den Vergleich der drei Capture & Replay-Tools, noch viele offene Fragen: Wären die ermittelten Ergebnisse unter einem anderen SUT oder bei einer längeren Untersuchung des verwendeten SUTs ähnlich ausgefallen? Hätte eine Verwendung von verschiedenen SUTs mit jeweils anderem Verschachtelungsgrad der GUI zu anderen Erkenntnissen beim Vergleich der Capture & Replay-Tools geführt? Auch eine Untersuchung mit mehreren, unterschiedlich erfahrenen Testern, die alle vergleichbare Tests mit demselben Tool entwickeln, bietet im Hinblick auf die Konsistenz von Entwicklungs- und Wartungsproblemen zusätzliche Forschungsmöglichkeiten.

Meiner Meinung nach stellt dabei eine Erweiterung der verwendeten Capture & Replay-Tools bei konstantem SUT den sinnvollsten weiterführenden Forschungsansatz dar, da sowohl direkt für die bereits getesteten Werkzeuge eine bessere Einord-

nung im Hinblick auf einen gegenseitigen Vergleich bzw. spezifischen Einsatz ermöglicht wird, als auch langfristig eine Aussage über die Rolle der den Tools zugrundeliegenden Generation ermittelt werden könnte. Dies würde wiederum Rückschlüsse für einen gezielten Einsatz oder eine weiterführende Forschung an den zugrundeliegenden Generationen ermöglichen. Dabei unterstützt bereits der in dieser Arbeit durchgeführte Vergleich den Prozess der Systematisierung existierender Generationen von Capture & Replay-Tools im Hinblick auf ökonomische Bedürfnisse. Neue Forschungsansätze, wie die Analyse einer GUI mit vorher definierten Abdeckungskriterien, könnten darüber hinaus eine Weiterentwicklung aktueller Werkzeuge oder sogar ihrer zugrundeliegenden Generationen bewirken. Eine Fortführung der wissenschaftlichen Arbeit zur Systematisierung und Weiterentwicklung von Capture & Replay-Tools ist dabei nicht nur wünschenswert, sondern im Hinblick auf die zunehmende Bedeutung von GUI-Tests in modernen Softwareprojekten unabdingbar.

Literaturverzeichnis

[AFK16] Alégroth, Emil ; Feldt, Robert ; Kolström, Pirjo: Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. In: *Information and Software Technology* 73 (2016), S. 66–80

[AFO13] Alegroth, Emil ; Feldt, Robert ; Olsson, Helena H.: Transitioning manual system test suites to automated testing: An industrial case study. In: *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on* IEEE, 2013, S. 56–65

[AFR15] Alégroth, Emil ; Feldt, Robert ; Ryrholm, Lisa: Visual gui testing in practice: challenges, problemsand limitations. In: *Empirical Software Engineering* 20 (2015), Nr. 3, S. 694–744

[AGOM15] Alégroth, Emil ; Gao, Zebao ; Oliveira, Rafael ; Memon, Atif: Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In: *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on* IEEE, 2015, S. 1–10

[Alé15] Alégroth, Emil: *Visual GUI Testing: Automating High-level Software Testing in Industrial Practice*, Chalmers University of Technology, Diss., 2015

[BEF11] Barmi, Zeinab A. ; Ebrahimi, Amir H. ; Feldt, Robert: Alignment of requirements specification and testing: A systematic mapping study. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* IEEE, 2011, S. 476–485

[BF12] Börjesson, Emil ; Feldt, Robert: Automated system testing using visual GUI testing tools: A comparative study in industry. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* IEEE, 2012, S. 350–359

[BWK05] Berner, Stefan ; Weber, Roland ; Keller, Rudolf K.: Observations and lessons learned from automated testing. In: *Proceedings of the 27th international conference on Software engineering* ACM, 2005, S. 571–579

[CL02] Cheon, Yoonsik ; Leavens, Gary T.: A simple and practical approach to unit testing: The JML and JUnit way. In: *ECOOP 2002—Object-Oriented Programming*. Springer, 2002, S. 231–255

[EBI06] Ellims, Michael ; Bridges, James ; Ince, Darrel C.: The economics of unit testing. In: *Empirical Software Engineering* 11 (2006), Nr. 1, S. 5–31

[Egg16] `http://www.testplant.com/`

[Fin01] Finsterwalder, Malte: Automating acceptance tests for GUI applications in an extreme programming environment. In: *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, 2001, S. 114–117

[GG93] Grochtmann, Matthias ; Grimm, Klaus: Classification trees for partition testing. In: *Software Testing, Verification and Reliability* 3 (1993), Nr. 2, S. 63–82

[GXF09] Grechanik, Mark ; Xie, Qing ; Fu, Chen: Creating GUI testing tools using accessibility technologies. In: *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on* IEEE, 2009, S. 243–250

[HS93] Horowitz, Ellis ; Singhera, Zafar: Graphical user interface testing. In: *Technical eport Us C-C S-93-5* 4 (1993), Nr. 8

[HS02] Hailpern, Brent ; Santhanam, Padmanabhan: Software debugging, testing, and verification. In: *IBM Systems Journal* 41 (2002), Nr. 1, S. 4–12

[Jen16] `https://jenkins.io/`

[KK16] Kresse, Antonia ; Kruse, Peter M.: Development and Maintenance Efforts Testing Graphical User Interfaces: A Comparison. In: *Proceedings of Workshop on Automated Software Testing (A-TEST), 2016*, ACM Press, 11 2016

[KL10] Kruse, Peter M. ; Luniak, Magdalena: Automated test case generation using classification trees. In: *Software Quality Professional* 13(1) (2010), S. 4–12

[KS15] Khazal, Anmar ; Sigurdsson, Armann D.: *Component-based Capture & Replay vs. Visual GUI Testing: an Empirical Comparison in Industry*, Göteborg: Chalmers tekniska högskola, Diplomarbeit, 2015

[LW00] LEHMANN, Eckard ; WEGENER, Joachim: est Case Design by Means of the CTE XL. In: *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000),* (2000)

[Mem02] MEMO, Atif M.: GUI Testing: Pitfalls and Process. In: *IEEE Computer Society* 35 (2002), Nr. 08, S. 87–88

[MSP01] MEMON, Atif M. ; SOFFA, Mary L. ; POLLACK, Martha E.: Coverage criteria for GUI testing. In: *ACM SIGSOFT Software Engineering Notes* Bd. 26 ACM, 2001, S. 256–267

[NRBM14] NGUYEN, Bao N. ; ROBBINS, Bryan ; BANERJEE, Ishan ; MEMON, Atif: GUITAR: an innovative tool for automated testing of GUI-driven software. In: *Automated Software Engineering* 21 (2014), Nr. 1, S. 65–105

[OAB12] OLSSON, Helena H. ; ALAHYARI, Hiva ; BOSCH, Jan: Climbing the" Stairway to Heaven"–A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In: *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on* IEEE, 2012, S. 392–399

[OAFG98] OSTRAND, Thomas ; ANODIDE, Aaron ; FOSTER, Herbert ; GORADIA, Tarak: A visual test development environment for GUI systems. In: *ACM SIGSOFT Software Engineering Notes* Bd. 23 ACM, 1998, S. 82–92

[Ola03] OLAN, Michael: Unit testing: test early, test often. In: *Journal of Computing Sciences in Colleges* 19 (2003), Nr. 2, S. 319–328

[PSH+16] PARENTEAU, Josh ; SALLAM, Rita L. ; HOWSON, Cindi ; TAPADINHAS, Joao ; SCHLEGEL, Kurt ; OESTREICH, Thomas: *Gartner Reprint.* `https://www.gartner.com/doc/reprints?id=1-2XXET8P&ct=160204`. Version: 2016

[QFT16] *QF-Test.* `https://www.qfs.de/de/qftest/index.html`. Version: 2016

[QN13] QURESHI, Imran A. ; NADEEM, Aamer: GUI Testing Techniques: A Survey. In: *International Journal of Future Computer and Communication* 2 (2013), Nr. 2, S. 142

[RH09] Runeson, Per ; Höst, Martin: Guidelines for conducting and reporting case study research in software engineering. In: *Empirical software engineering* 14 (2009), Nr. 2, S. 131–164

[RMPM12] Rafi, Dudekula M. ; Moses, Katam Reddy K. ; Petersen, Kai ; Mäntylä, Mika V.: Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: *Proceedings of the 7th International Workshop on Automation of Software Test* IEEE Press, 2012, S. 36–42

[RUCH01] Rothermel, Gregg ; Untch, Roland H. ; Chu, Chengyun ; Harrold, Mary J.: Prioritizing test cases for regression testing. In: *Software Engineering, IEEE Transactions on* 27 (2001), Nr. 10, S. 929–948

[SAP06] Sjösten-Andersson, Erik ; Pareto, Lars: Costs and Benefits of Structure-aware Capture/Replay tools. In: *SERPS'06* (2006), S. 3

[SL10] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest.* dpunkt, 2010

[XGFC09] Xie, Qing ; Grechanik, Mark ; Fu, Chen ; Cumby, Chad: Guide: A gui differentiator. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* IEEE, 2009, S. 395–396

[YCM09] Yeh, Tom ; Chang, Tsung-Hsiang ; Miller, Robert C.: Sikuli: using GUI screenshots for search and automation. In: *Proceedings of the 22nd annual ACM symposium on User interface software and technology* ACM, 2009, S. 183–192

Abbildungsverzeichnis

2.1 Theoretisches Modell eines Systems hinsichtlich der Unterscheidung von Front-End und Back-End, sowie deren Möglichkeiten für manuelles und automatisiertes Testen [Alé15] 11
2.2 Unterschiede zwischen Blackbox und Whitebox 12
2.3 Funktionsweise von Capture & Replay-Tools aus Ostrand u. a. (1998) 14
2.4 Aufbau von GUI-Komponenten in Tools der zweiten Generation ([Alé15]) 16
2.5 Aufbau von GUI-Komponenten in Tools der dritten Genetation ([Alé15]) 17

3.1 Testablauf 21
3.2 Vorgehensweise zur Datenerfassung 22
3.3 GUI-Komponenten von TESTONA 23
3.4 Exemplarisches Testdesign in der GUI von TESTONA 24
3.5 Grafische Oberfläche von QF-Test 26
3.6 Grafische Oberfläche von TestComplete 27
3.7 Zwei-System Model von Eggplant Functional [Egg16] 28
3.8 Grafische Oberfläche F_1 von Eggplant Functional 28
3.9 Grafische Oberfläche F_2 von Eggplant Functional 29

4.1 Aufbau der Testumgebung 37
4.2 Automatisierte Testausführung für QF-Test & TestComplete 38
4.3 Automatisierte Testausführung für Eggplant 39
4.4 Bildbearbeitung in Eggplant 41
4.5 Grafik mit Fokuspunk 41
4.6 Grafiken in Eggplant 42
4.7 Grafische Elemente der Testmatrix 51
4.8 Grafik-Checkpoint in QF-Test 55
4.9 Tabellen-Checkpoint in QF-Test 56
4.10 Variablen in QF-Test 56
4.11 StartTestona in QF-Test 57
4.12 CreateTree in QF-Test 57
4.13 MainSuite in QF-Test 58
4.14 „Rename“ in QF-Test 60
4.15 Verwendung eines Schiebereglers in QF-Test 60
4.16 Aufruf der Search Prozedur 61

4.17 Dialog „Dependency Rules“ 62
4.18 Region-Checkpoints in TestComplete 66
4.19 Variablen in TestComplete 67
4.20 Testfall `CreateNewTree` in TestComplete 68
4.21 Suite in TestComplete 69
4.22 MainSuite in TestComplete 69
4.23 Namemapping in TestComplete 71
4.24 Bearbeiten des Namemappings in TestComplete 72

5.1 Implementierungsaufwand (Zeit in Minuten [min]) 80
5.2 Wartungsaufwand (Zeit in Minuten [min]) 83

Tabellenverzeichnis

3.1 Entwicklung von TESTONA 23
3.2 Verwendete GUI-Komponenten innerhalb der Testszenarien 31

4.1 Erwartete Fehlschläge (↯) der Testfälle 35
4.2 Entwicklungs- & Reparaturzeit für Eggplant in Minuten 50
4.3 Entwicklung- & Reparaturzeit für QF-Test in Minuten 59
4.4 Entwicklung- & Reparaturzeit für TestComplete in Minuten 70

5.1 Implementierungs- und Wartungsaufwand (Zeit in Minuten) 80
5.2 Bearbeitete Tickets zwischen TESTONA 4.1 und 4.3 86
5.3 Analyse der Tickets von 4.1 zu 4.3 in Eggplant 86
5.4 Analyse der Tickets von 4.1 zu 4.3 in QF-Test 86
5.5 Analyse der Tickets von 4.1 zu 4.3 in TestComplete 86
5.6 Bearbeitete Tickets zwischen TESTONA 4.3 und 4.5 87
5.7 Analyse der Tickets von 4.3 zu 4.5 in Eggplant 87
5.8 Analyse der Tickets von 4.3 zu 4.5 in QF-Test 87
5.9 Analyse der Tickets von 4.3 zu 4.5 in TestComplete 87

Anhang

EditTree

1. TestGroup.testona öffnen
2. `noname_1` im Baum anklicken
3. `noname_1` über das Rechtsklick-Menü → Edit → Rename in `A` umbenennen
4. `noname_2` über den Shortcut F2 in `A` umbenennen
5. `noname_3` über die Menüleiste fett machen
6. `root` über die Menüleiste kursiv machen
7. ⇒ Prüfen ob die Schrift von `root` kursiv ist
8. Das Kursiv von `root` über die Menüleiste deaktivieren
9. ⇒ Prüfen ob die Schrift normal ist
10. `root` über die Menüleiste kursiv machen
11. Die Schrift von `noname_4` über die Menüleiste von 9 auf 16 erhöhen
12. Die Schriftart von `noname_5` über die Menüleiste auf „Stencil“ ändern
13. Die Schriftfarbe von `noname_6` über das Rechtsklick-Menü → Format → Font in Rot ändern
14. Die Hintergrundfarbe von `noname_7` über die Properties → Appearance → Knoten Hintergrundfarbe in Grün ändern
15. Arrange All
16. ⇒ aktuellen Zwischenstand überprüfen
17. `A` anklicken
18. In Properties → Appearance → Backgroundcolor Pink auswählen (1. Zeile, 2te Farbe von rechts)
19. Transparenz auf den sechsten Anzeigestrich von links ziehen
20. Rechtsklick auf `B` → Format → Subtree Color → Blue auswählen
21. ⇒ Endzustand überprüfen
22. Erst Close File und dann Don‘t Save klicken

FoldTree

1. TestGroup.testona öffnen
2. Über das Rechsklickmenü `noname_1` falten
3. ⇒ der Teilbaum von `noname_1` ist nicht mehr sichtbar und neben `noname_1` sind zwei Pfeile sichtbar
4. Im Rechtsklick-Menü „Show Subtree“ auswählen
5. ⇒ Ansicht ist in den Tab mit dem Teilbaum gewechselt
6. Rechtsklick-Menü von `noname_1` „Unfold“
7. ⇒ Das Tab vom Teilbaum ist verschwunden und der ganze Baum ist sichtbar
8. Doppelklick auf `noname_2` um den Teilbaum zu falten
9. ⇒ Der Teilbaum von `noname_2` ist nicht mehr sichtbar
10. Doppelklick auf die Pfeile neben `noname_2` um zur Ansicht des Subtrees zu gelangen
11. Mit F6 eine weitere Klasse zu `noname_2` hinzufügen
12. Das Tab des Subtrees schließen
13. Doppelklick auf `noname_2` um den Teilbaum auszuklappen
14. Arrange All
15. ⇒ Die neue Klasse (`noname_8`) ist sichtbar
16. Rechtsklick-Menü auf Root
17. ⇒ Prüfen ob die Option „Unfold“ deaktiviert ist
18. Erst Close File und dann Don‘t Save klicken

CopyPasteOfSubtrees

1. TestGroup.testona öffnen
2. Über das Rechtsklick-Menü von `noname_1` Select → Subtree auswählen
3. CTRL+C
4. CRTL+V
5. Den eingefügten Teilbaum nach links ziehen
6. Rechtsklick auf `noname_7` Edit → Paste
7. Arrange All in Toolbar klicken
8. ⇒ der eingefügte Baum wurde unter `noname_7` eingefügt
9. Den einzelnen Baum mittels „Drag and Drop“ markieren und CTRL+X
10. `noname_3` auswählen und den Teilbaum aus der Zwischenablage mit CTRL+V einfügen
11. ⇒ Der Teilbaum ist nun ein Kind von `noname_3`
12. Rechtsklick auf `noname_2` Select → Subtree
13. CTRL+C
14. Neue Datei erstellen
15. Root auswählen
16. CTRL+V
17. ⇒ Der Teilbaum wurde in die neue Datei eingefügt
18. Erst Close File und dann Don‘t Save klicken
19. Erst Close File und dann Don‘t Save klicken

WriteProtectedFiles

1. WriteProtected.testona öffnen
2. `noname_5` und `noname_6` eine Klassifikation mit „F6“ hinzufügen
3. Datei Speichern
4. ⇒ Es öffnet sich eine Warnung, dass die Datei „Read Only“ ist.
5. „No“ klicken
6. `noname_1` anklicken
7. Über die Toolbar → Select → Select Subtree, den Teilbaum markieren
8. Mit CTRL+C den Teilbaum markieren
9. In das Tab der neuen Datei wechseln
10. Den Wurzelknoten auswählen
11. Den Teilbaum aus der Zwischenablage mit CTRL+V einfügen
12. Datei Speichern
13. File → Close
14. ⇒ Es öffnet sich eine Fehlermeldung „WriteProtected.testona“ ist schreibgeschützt
15. „no“ klicken

PropertiesView

1. TestTree.testona öffnen
2. `noname_10` auswählen
3. Unten in den Properties in das Menü „Core“ gehen
4. Im Propertiesfenster auf `noname_10` klicken um den Knoten umzubenennen
5. Nach dem Klick erscheint rechts ein Button mit drei Punkten, auf diesen klicken
6. Den Namen löschen und „NewName“ eingeben
7. Ok Klicken
8. ⇒ Der Name vom Knoten hat sich auch im Baum auf `NewName` geändert
9. Rechtsklick auf `noname_5`, Edit → Rename
10. Namen auf `TestName` ändern
11. Erneut in den Properties → „Core“ auswählen
12. ⇒ Der neue Name wird dort richtig angezeigt
13. TestGroup.testona öffnen
14. ⇒ In den Properties → „Core“ steht bei Name „TestGroup.testona“
15. TestGroup.testona schließen
16. Properties → „Core“ zeigt bei Name „TestTree.testona“ an
17. `noname_2` im Baum auswählen
18. „Allowed Arcs“ in den Properties auswählen und
19. ⇒ Prüfen ob die drei Knoten richtig angezeigt werden
20. Erst Close File und dann Don‘t Save klicken

Autolayout

1. TestGroup.testona öffnen
2. `noname_2` auswählen
3. mit „F6“ eine Klasse hinzufügen
4. In den Properties „Autolayout“ auswählen
5. „Diagonal Allignment“ auswählen
6. ⇒ Die Blätter von `noname_2` sind diagonal nach rechts unten angeordnet
7. „Reverse Allignment“ auswählen
8. ⇒ Die Blätter von `noname_2` sind diagonal nach links unten angeordnet
9. „Circular Allignment“ auswählen
10. ⇒ Die Blätter von `noname_2` sind diagonal auf zwei Ebenen angeordnet
11. Den Skrollbalken bei „Position“ nach ganz links schieben
12. ⇒ `noname_5` ist genau unter `noname_2`, alle anderen Kinder befinden sich rechts davon
13. Den Skrollbalken bei Position nach ganz rechts schieben
14. Den Wurzelknoten auswählen
15. „Arrange All“ in der Toolbar auswählen
16. ⇒ `noname_8` ist genau unter `noname_2`, alle anderen Kinder befinden sich links davon
17. ⇒ Das Autolayout vom Wurzelknoten ist noch „Hierachial“ und die Position mittig eingestellt
18. Bei Gaps „Horizontal between children“ den Wert „-16“ eingeben
19. ⇒ `noname_1` und `noname_2` rücken näher an einander
20. Bei Gaps „Vertical“ mittels den Pfeiltasten nach unten auf „10“ stellen
21. ⇒ `noname_1` und `noname_2` rücken näher an die Wurzel
22. `noname_2` auswählen
23. Bei Gaps „Vertical between children“ 100 eingeben
24. ⇒ `noname_6` und `noname_7` rücken deutlich weiter nach unten als `noname_5` und `noname_8`
25. Erst Close File und dann Don‘t Save klicken

Outline1

1. TestGroup.testona öffnen
2. In der Outline `noname_1` auswählen
3. Mit F6 eine neue Klasse hinzufügen
4. ⇒ `noname_1` hat im Baum eine Klasse mehr (`noname_8`)
5. In der Outline `noname_8` auswählen
6. Mit F7 eine Klassifikation hinzufügen
7. ⇒ `noname_8` hat im Baum ein Kind
8. F6 drücken
9. ⇒ Es wurde keine Klasse hinzugefügt
10. Mit F8 eine Komposition hinzufügen
11. ⇒ `noname_8` hat ein weiteres Kind
12. In der Outline `noname_9` auswählen
13. Mit SHIFT+F6 eine Klassifikation hinzufügen
14. ⇒ Im Baum und in der Outline ist `noname_11` fokussiert
15. In der Outline auf den Pfeil neben `noname_1` klicken
16. ⇒ `noname_1` wird in der Outline und im Baum gefaltet
17. Anzeige in der Outline auf „flach“ stellen
18. ⇒ Die Kinder von `noname_1` werden weiterhin nicht angezeigt, alle Elemente befinden sich auf der gleichen Ebene
19. Im Baum `noname_1` über das Rechtsklick-Menü entfalten
20. ⇒ In der Outline werden auch die Kinder von `noname_1` angezeigt
21. In der Outline-Toolbar „Alphabetisch Sortieren“ auswählen
22. ⇒ Die Elemente werden von `noname_1` bis `TestGruppe` alphabetisch angezeigt
23. „Alphabetisch Sortieren“ wieder deaktivieren
24. Die Anzeige in der Outline wieder auf „Hierachical“ stellen
25. Erst Close File und dann Don‘t Save klicken

Outline2

1. TestGroup.testona öffnen
2. In der Outline Rechtsklick auf `noname_5` → Edit → Rename
3. `TestName` eingeben und mit Enter bestätigen
4. ⇒ Der Knoten hat auch im Baum den neuen Namen
5. `noname_6` in der Outline auswählen und mit F2 in `NewName` umbenennen und mit Enter bestätigen
6. ⇒ Knoten hat auch im Baum den neuen Namen
7. Rechtsklick auf `noname_7` Edit → Cut
8. `noname_1` auswählen
9. CTRL+V
10. „Arrange All“ in Toolbar klicken
11. ⇒ `noname_7` ist jetzt ein Kind von `noname_1`, die Kante im Baum wurde geändert
12. Rechtsklick auf `noname_2` Select → Subtree
13. CTRL+X
14. `noname_7` auswählen und Rechtsklick → Edit → Paste
15. Arrange All
16. ⇒ Der Baum wurde unter `noname_7` eingefügt
17. „Link with diagram“ deaktivieren
18. `noname_3` in der Outline auswählen
19. ⇒ Der Fokus im Baum wurde nicht auf `noname_3` geändert
20. Rechtsklick auf `noname_2` Select → Subtree
21. ⇒ Der Subtree wurde im Baum nicht selektiert
22. Rechtsklick auf `noname_2` Select → All
23. ⇒ Alle Elemente wurden im Baum selektiert
24. „Link with diagram“ aktivieren
25. Erst Close File und dann Don‘t Save klicken

CreateRenameTestcases

1. TestTree.testona öffnen
2. In den Testcasetree gehen und mit EINFG einen Testfall hinzufügen
3. Rechtsklick → Add Testcase
4. Im ersten Testfall `noname_9` und `noname_4` markieren
5. Im zweiten Testfall `noname_11` und `noname_5`
6. Dem Wurzelknoten über das Rechtsklick-Menü eine Klassifikation hinzufügen
7. ⇒ In beiden Testfällen kommt in der Testmatrix rechts ein neues Fragezeichen hinzu
8. Ein Klick auf das Fragezeichen verändert nichts
9. `noname_14` über das Menü Rechtsklick eine Klasse hinzufügen
10. Beide Fragezeichen markieren
11. ⇒ Es sind keine Fragezeichen mehr sichtbar
12. Rechtsklick auf `Testcase2` → Edit → Rename
13. `Test1` eingeben, Enter drücken
14. ⇒ Die Properties zeigt bei Core den richtigen Namen an
15. `Testcase1` auswählen
16. In Properties → Core auf den Namen `Testcase1` klicken
17. Nach dem Klick erscheint rechts ein Button mit drei Punkten, darauf klicken
18. Den Namen löschen und `Test3` eingeben
19. Ok klicken
20. ⇒ Der Name wird im Testfallbaum und in den Properties richtig angezeigt
21. Erst Close File und dann Don‘t Save klicken

ChangeMarktypes

1. TestGroup.testona öffnen
2. `Testgroup 1` ausklappen
3. In der obersten Zeile den Mark bei `noname_3` setzen
4. ⇒ alle Markierungen darunter passen sich an und sind ganz links
5. Erneut auf die oberste Markierung bei `noname_3` klicken
6. ⇒ Die ersten zwei Spalten bestehen aus Fragezeichen
7. Die oberste Markierung bei `noname_4` setzen
8. Rechtsklick auf die Markierung von `noname_4` im ersten Testfall bei Mark Type → Circle auswählen
9. ⇒ Der Kreis verändert sich und wird weiß
10. Rechtsklick auf die Markierung von `noname_4` im zweiten Testfall bei Mark Type → Solid Square auswählen
11. ⇒ Der Kreis wird zum schwarzen Viereck
12. Rechtsklick auf die Markierung von `noname_4` im dritten Testfall bei Mark Type → Square auswählen
13. ⇒ Der Kreis wird zum weißen Viereck
14. Window → Preferences
15. TESTONA → Edit
16. Mode for Marks auf Implicit ändern
17. Apply
18. Es öffnet sich ein neues Fenster → „Convert marks with non-standard mark types to explicit marks“ auswählen und mit OK bestätigen
19. Erneuten Hinweis mit Ok bestätigen
20. Preferences Fenster mit Ok Schließen
21. ⇒ Die Markierungen haben sich geändert
22. Markierung von `noname_3` im ersten Testfall setzen
23. Klick auf die Markierung von `noname_4` im dritten Testfall
24. ⇒ Es wird ein impliziter Kreis
25. Shift + Klick auf die Markierung von `noname_4` im vierten Testfall
26. ⇒ Bei `noname_3` und `noname_4` sind die Markierungen Dreiecke
27. Klick auf die Markierung von `noname_4` im fünften Testfall
28. ⇒ Bei `noname_3` und `noname_4` sind die Markierungen implizite Dreiecke
29. Implizit wieder auf Explizit stellen
30. Close File und Dont't Save

GenerateTestcases

1. GenerateTestcases.testona öffnen
2. Tools → Generate Test Cases ...
3. Generate → OK
4. Ordner „Default Minimal Combination“ ausklappen
5. ⇒ Der Ordner enthält 4 Testfälle
6. Gruppe auswählen
7. Tools → Generate Test Cases ...
8. New
9. „Test“ als Name eingeben → OK
10. Als Regel „`noname_1 * noname_4;`“ eingeben
11. Delete Button klicken
12. Im neuen Fenster mit „Yes“ bestätigen
13. ⇒ In der Regel steht wieder „`noname_1 + noname_2 ;`“
14. Bei Rules and Naming „Default Maximal Combination“ auswählen
15. Advanced Options klicken
16. „Complement Existing“ auswählen
17. Generate → OK
18. ⇒ Die Gruppe wurde durch vier weitere Testfälle ergänzt und enthält nun 8 Testfälle
19. Gruppe Löschen (Rechtsklick → Edit Delete)
20. Tools → Generate Test Cases ...
21. Bei Rules and Naming „Default Pairwise Combination“ auswählen
22. Short auf ID umändern
23. Generate → OK
24. Ordner „Default Pairwise Combination“ ausklappen
25. ⇒ Der Ordner enthält 8 Testfälle
26. Tools → Generate Test Cases ...
27. Bei Rules and Naming „Default Pairwise Combination“ auswählen
28. „Complement Existing“ auswählen
29. Generate → OK
30. ⇒ Es sind keine neuen Tests dazu gekommen (weiterhin 8)
31. Erst Close File und dann Don‘t Save klicken

ManualTestCases

1. Generate.testona öffnen
2. Tools → Generate Test Cases ...
3. New
4. „Test2“ als Name eingeben → OK
5. `noname_17` in die Regel ziehen
6. Bei Operators den Button ∗ an klicken
7. `noname_16` in die Regel ziehen
8. Klick auf Apply
9. Erneut in die Regel klicken
10. Semikolon löschen
11. Bei Operators den Button + an klicken
12. `noname_5` in die Regel schreiben
13. Klick auf Apply
14. Generate → OK
15. `Test2` ausklappen
16. ⇒ Es sind 2 Testfälle enthalten
17. `Test2` Löschen
18. Tools → Generate Test Cases ...
19. In das Prioritized Tab wechseln
20. Generate → OK
21. „PMC occurence“ ausklappen
22. Der Ordner enthält 5 Testfälle
23. Erst Close File und dann Don‘t Save klicken

RenameGroupsAndSequences

1. TestSequence.testona öffnen
2. Rechtsklick → Add Testcase
3. Testgruppe ausklappen
4. Testfall in Testgruppe ziehen (oben auf die Testgruppe)
5. ⇒ `Testcase1` steht ganz unten in der Testgruppe
6. Rechtsklick auf `Testgruppe` → Advanced Rename auswählen
7. `Start With` auf 2 ändern und mit Ok bestätigen
8. ⇒ Die Testfälle wurden nicht umbenannt
9. `TestGroup1 Testcase1` bis zum einzelnen `Testcase 1` markieren
10. Rechtsklick → Advanced Rename mit Ok bestätigen
11. ⇒ Testfälle werden in `TC2` bis `TC5` umbenannt
12. Testsequenz ausklappen
13. `Teststep1` bis `3` markieren
14. Rechtsklick → Advanced Rename
15. Replace with „TS“, Start With 0 → Ok
16. ⇒ Namen sind `TS0` bis `TS2`, ohne Leerzeichen
17. Testsequenz zwischen `TC 3` und `TC 4` ziehen
18. `TC 2` bis `TC 5` markieren
19. Rechtsklick → Advanced Rename
20. Replace with „Test“, Start With „0“ → Ok
21. Testsequenz ausklappen
22. ⇒ Die Tests wurden umbenannt, die innerhalb der Sequenz nicht
23. Erst Close File und dann Don‘t Save klicken

TestSequences

1. TestSequenz.testona öffnen
2. Testsequenz ausklappen
3. `Teststep 1` auswählen
4. ⇒ in den Properties steht „Teststep . . . “
5. Rechtsklick auf Testsequenz → Add Teststep
6. ⇒ `Teststep4` ensteht
7. Rechtsklick auf Testsequenz → Change Type to „Testgroup“
8. ⇒ in den Properties steht „Testgroup“
9. Testsequenz ausklappen
10. `Teststep 1` auswählen
11. ⇒ in den Properties steht „Testcase . . . “
12. Rechtsklick auf Testgruppe → Change Type to „Testsequenz“
13. Testgruppe ausklappen
14. `Testcase 1` auswählen
15. ⇒ in den Properties steht „Teststep . . . “
16. Rechtsklick auf Testgruppe → Change Type to „Testcase“
17. ⇒ Es entsteht ein Testfall mit dem Namen „Testgroup 1“
18. Rechtsklick auf Testgruppe → Change Type to „Testsequenz“
19. ⇒ in den Properties steht „Testsequence . . . “
20. Erst Close File und dann Don‘t Save klicken

BoundaryValueAnalysis

1. TestGruppe.testona öffnen
2. Tools → Boundary Value Analysis klicken
3. Add Button klicken
4. Klick in Parameter Name
5. Mit CTL+A alles markieren, dann mit ENTF löschen
6. Neuen Namen „Test“ eingeben
7. Data Type auf „Signet Short“ ändern
8. Step Width auf „2“ ändern
9. Add
10. Intervall Name: „A“, Upper Border: „20“
11. Ok
12. Generate → Vergleich
13. Tools → Boundary Value Analysis
14. Edit
15. Add
16. Intervall Name: „B“, Lower Border: „-5“, Upper Border: „10“, Endpoint Exclude
17. Ok
18. Generate
19. ⇒ Prüfen ob der Baum richtig generiert wurde
20. Erst Close File und dann Don‘t Save klicken

CreateEditTags

1. TestGruppe.testona öffnen
2. Tools → Tags... öffnen
3. Create
4. OK
5. `NewTag1` selektieren
6. Name in „TestTag“ ändern (alles mit CTL+A auswählen)
7. Apply
8. Close
9. Über das Rechtsklick-Menü von `noname_4` → Tags → TestTag aktivieren
10. ⇒ Prüfen ob TestTag im Properties Menü erschienen ist
11. Über das Rechtsklick-Menü von `noname_14` → Tags → TestTag deaktivieren
12. ⇒ Prüfen ob TestTag im Properties Menü verschwunden ist
13. Tools → Tags... öffnen
14. Attributes auswählen
15. Icon auf „Add custom icon“ ändern
16. New
17. „Icon.jpg“ auswählen
18. Öffnen
19. Icon auf „Icon.jpg“ ändern
20. Apply
21. Close
22. Über das Rechtsklick-Menü von `noname_3` → Tags → Attributes aktivieren → Prüfen ob Icon angezeigt wird
23. Erst Close File und dann Don‘t Save klicken

VariantenManagementView

1. Varianten.testona öffnen
2. Menü → Window → Open Perspective → Variant Management
3. Im Dropdown Menü in der Toolbar „Variant2“ auswählen
4. ⇒ `noname_4` ist nun grau gefärbt
5. Window → Preferences
6. Testona → Varianten Management
7. „Color of excludes variant elements“ in die erste Farbe bei Farbschema ändern
8. Apply klicken und Preference Dialog schließen
9. ⇒ `noname_4` hat die Farbe verändert
10. Rechtsklick auf `noname_5` → Variant → Variant 2 deaktivieren
11. ⇒ `noname_5` ist ebenfalls rot gefärbt
12. Rechtsklick auf `noname_3` → Variant → Variant 2 deaktivieren
13. ⇒ Teilbaum von `noname_3` ist rot gefärbt
14. Mit Pfeil nach links in Toolbar zu Variante 1 wechseln
15. ⇒ Alle Markierungen im Baum sind weg
16. Tools → Variants
17. ⇒ in Preferences öffnet sich ein neues Tab
18. Tab Schließen
19. Tools → Varianten Management
20. New Variant
21. Name: „TestVariant“, OK → TestVariant erscheint in der Varianten Übersicht → OK Button
22. Rechtsklick auf `noname_2` → Variant → TestVariant deaktivieren
23. Mit Pfeil nach links in Toolbar zu TestVariant wechseln
24. ⇒ Teilbaum von `noname_2` ist rot gefärbt
25. Über die Pfeile oben Rechts in die Standard-View wechseln
26. Erst Close File und dann Don‘t Save klicken

Help

1. TestGruppe.testona öffnen
2. Tools → Dependency Rules... öffnen
3. Den Button mit dem Fragezeichen drücken
4. ⇒ es öffnet sich die Hilfe
5. ⇒ Prüfen ob Dependency Rules als Überschrift da steht
6. Hilfe Fenster schließen
7. Dependency Rules Dialog mit OK schließen
8. Tools → Statistics öffnen
9. Den Button mit dem Fragezeichen drücken
10. ⇒ es öffnet sich die Hilfe
11. ⇒ Prüfen ob Statistics als Überschrift da steht
12. Oben in das Search Fenster klicken und „Shortcuts“ eingeben
13. ⇒ Prüfen ob es zwei Ergebnisse gibt
14. Hilfe Fenster schließen
15. Statistics mit OK schließen
16. Close File

SearchingElements

1. TestTree.testona öffnen
2. Klick auf das Search-Symbol in der Toolbar
3. Den String `noname_5` eingeben → Search klicken
4. ⇒ `noname_5` ist im Baum markiert
5. ⇒ in den Propterties hat sich ein neuer Reiter „Search“ geöffnet, in dem unten der Name `noname_5` und das geöffnete File steht
6. „Add Testcase“
7. Rechtsklick → Edit → Rename
8. `noname_5` eingeben
9. Klick auf den Pfeil rechts neben dem Search-Symbol in der Toolbar
10. Den Pfeil bei „Search String“ anklicken und `noname_5` auswählen
11. ⇒ Es gibt zwei Suchergebnisse: Der Knoten und der Testfall
12. Menü → Search
13. Als String „`*_5`“ eingeben
14. Beide Haken bei TreeItems wegnehmen → Search
15. ⇒ Es gibt ein Ergebnis: Der Testfall
16. Bei `noname_1` und `noname_6` das Attributes Tag aktivieren
17. CTRL+H
18. Searchstring = „*“, nur einen Hacken bei TreeItems Tags
19. ⇒ Es werden 2 Elemente gefunden: „Attributes [noname1/6]“
20. Erst Close File und dann Don‘t Save klicken

DependencyRulesLogical

1. DependenyRules.testona öffnen
2. Tool → Dependency Rules
3. Klick „New Logical Rule“
4. „TestRule“ als Namen eingeben → OK
5. Als Regel `noname_7` eingeben
6. Apply, OK
7. ⇒ Die eingegebene Regel, sowie der Inhalt der Beschreibung wird in der Übersicht angezeigt
8. Regel auswählen
9. Edit klicken
10. Den „NOT“ Button auswählen
11. Apply, OK
12. Den Rule Checker aktivieren
13. ⇒ Testfälle werden aktualisiert
14. ⇒ `Testcase 8` wird rot
15. Tool → Dependency Rules
16. Testregel löschen
17. „delet rule“ deaktivieren
18. ⇒ der Status hat nur noch einen Strich
19. OK
20. Die Testfälle wurden geupdated
21. Rulechecker deaktivieren
22. Erst Close File und dann Don‘t Save klicken

DependencyRulesNumeric

1. Complement_bug.testona öffnen
2. Rulechecker aktivieren
3. `Testgroup 1` selektieren
4. Tools → Generate Test Cases ...
5. „Default Maximal Combination“ auswählen
6. Advanced Options → Complement Existing auswählen
7. Generate, OK
8. Testgruppe aufklappen
9. ⇒ Es sind weiterhin 5 grüne Testfälle in der Testgruppe
10. Im dritten Test `noname_3` in der Testmatrix markieren
11. ⇒ Der dritte Testfall hat einen roten Kreis und die Gruppe wird rot angezeigt
12. Tool → Dependency Rules
13. Auf den Pfeil bei „New Logical Rule“ klicken und in dem neuen Menü „New Numeric Rule“ auswählen
14. Auf „New Numeric Rule“ klicken
15. Als Name „NumRule“ eingeben → OK
16. Convention auf „ID“ ändern
17. `noname_2` per Drag and Drop in die Regel ziehen
18. `noname_5` per Drag and Drop in die Regel ziehen
19. ⇒ es stehen die IDs „25“ und „39“ in der Regel
20. ⇒ Oben wird eine Fehlermeldung angezeigt
21. Apply
22. OK
23. ⇒ Die Regel wird in der Übersicht angezeigt, State ist „−“
24. Regel auswählen und „Delete“ klicken
25. ⇒ Die Regel wurde gelöscht
26. OK
27. Erst Close File und dann Don‘t Save klicken

ExportExcel

1. TestTree.testona öffnen
2. File → Export öffnen
3. „Image“ ausklappen
4. „Image“ auswählen
5. Next
6. Bei Page Selection `Testtree` auswählen
7. Den Datei-Pfad eingeben
8. Format „JPG“ auswählen
9. Back klicken
10. Next klicken
11. ⇒ Die Einstellungen für den Pfad und JPG wurden gespeichert
12. Cancel
13. File → Export öffnen
14. Report ausklappen
15. Excel auswählen
16. Next
17. Datei -Pfad anpassen
18. Format auf „Excel 2007/2010 (.xlsx)“ wechseln
19. „Export Tree Items“ selektieren
20. Bei Tag Selection „TestcaseSpecification“ auswählen
21. Finish
22. File → Export öffnen
23. „Tool“ ausklappen
24. ⇒ Prüfen ob „Messina“ und „Matlab“ angezeigt werden
25. Cancel
26. Close File

ExportHTML

1. TestTree.testona öffnen
2. File → Export öffnen
3. („Report“ ausklappen) „HTML“ auswählen
4. Next
5. Bei Page Selection „TestTree“ auswählen
6. Next
7. Datei-Pfad anpassen
8. File in „TestTreeHTML“ umbenennen
9. „Export Test Cases“ auswählen
10. Naming Convention in „unique“ umändern
11. Tag Selection „Description“ auswählen
12. UP-Button auswählen
13. Finish
14. File → Export öffnen
15. ⇒ Prüfen ob „PDF/RTF“ angezeigt wird
16. ⇒ Prüfen ob „Text“ angezeigt wird
17. Bei „Select an export destination“ „i“ eingeben
18. ⇒ es wird nur noch „Image“ angezeigt
19. Bei „Select an export destination“ „c“ eingeben
20. ⇒ es werden nur noch „Excel / HTML Requirements Coverage“, sowie „Excel“ angezeigt
21. Cancel
22. Close File

Import

1. Menu File → Import
2. ⇒ Prüfen ob „CTA“ und „Mindmap“ sichtbar sind
3. Excel ausklappen und „Excel“ auswählen
4. Next
5. Klick Button „Select File“
6. ImportExcel mit Pfad eingeben
7. Öffnen
8. „Test Cases“ auswählen
9. Next
10. „Columns represent testcases“ auswählen
11. ⇒ Prüfen ob rechts „Parameter“ statt „Test_Case“ steht
12. Click „ImportRowsRepresentTestcases“
13. Next
14. Bei „Create New Testonafile“ den Pfad anpassen das es in die Outputfiles kommt
15. Finish
16. ⇒ Prüfen ob `Description` als Knoten im Baum sichtbar ist
17. ⇒ Prüfen ob die anderen Knoten sichtbar sind
18. „Imported Testcases“ ausklappen
19. ⇒ Prüfen ob es Testgroup und Testsequence als normalen Test gibt
20. Erst Close File und dann Don‘t Save klicken

TestCoverage

1. TestSequence.testona öffnen
2. Tools → Test Coverage
3. Tab von „Code Coverage“ größer ziehen
4. Coverage Rule „Pairwise Combination“ auswählen
5. Refresh klicken
6. ⇒ 81% und 13 von 16 Tupeln
7. `Testgruppe 1` auswählen
8. Unten „Selected“ wählen
9. Refresh klicken
10. ⇒ 56% und 9 von 16 Tupeln
11. Coverage Rule „Maximal Combination“ auswählen
12. Refresh klicken
13. ⇒ 25% und 3 von 12 Tupeln
14. All auswählen
15. Refresh klicken
16. ⇒ 5 von 12 Tupeln
17. „RuleChecker only “ aktivieren
18. ⇒ Es erscheint ein Hinweis
19. „Rule Checker“ in der Toolbar aktivieren
20. ⇒ Der Hinweis verschwindet
21. Den „Rule Checker“ ausschalten
22. Reset Perspective
23. Close File

Protokoll für fehlgeschlagene Tests

Capture & Replay-Tool:

TESTONA Version:

Name des Tests:

Warum ist der Test fehlgeschlagen?

- Grund 1:
- Grund 2:
- ...

Stelle im Code:

- Stelle im Code
- Stelle im Code
- ...

Wie wurde der Test repariert?

- Neuer Code/Befehl
- Neuer Code/Befehl
- ...

Anzahl geänderter LOC/Befehle:

-
-
- ...

Benötigte Zeit:

-
-
- ...

Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit verwendet habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: