Bachelorthesis

Testautomatisierung mit QF-Test

zur Erlangung des akademischen Grades

Bachelor of Science

eingereicht im Fachbereich Mathematik, Naturwissenschaften und Informatik an der Technische Hochschule Mittelhessen

von

Oliver Thomas

20. September 2017

Referent: Prof. Dr. Peter Kneisel

Korreferent: Dennis Priefer

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Gießen, den 20. September 2017 Oliver Thomas

Diese Arbeit beschäftigt sich mit der Erstellung einer Testautomatisierung für eine Java-basierte Anwendung. Die Sicherstellung der Softwarequalität wird oft vernachlässigt, spielt aber eine immer größer werdende Rolle. Testprozesse sind in der Zukunft ein unausweichliches Thema, da immer mehr kritische Prozesse vom Computer übernommen werden. Ein Beispiel hierfür sind unter anderem selbstfahrende Fahrzeuge.

Mit dieser Arbeit wird das nötige Hintergrundwissen vermittelt, um Testprozesse und deren Ziele zu verstehen. Dabei werden auch verschiedene Methoden (beispielsweise Black-Box Testing) vorgestellt, mit denen man Testfälle spezifizieren und ausführen kann. Auch die Testebenen, welche den Testbereich definieren, werden beschrieben.

Des Weiteren wird erläutert, welche Vorteile eine Testautomatisierung bringen kann und wann sich der Aufwand zur Erstellung einer Testautomatisierung lohnt. Dafür werden unter anderem verschiedene Testautomatisierungswerkzeuge evaluiert. Das für diese Arbeit am besten geeignete Werkzeug (QF-Test) wird dann anhand von Beispielen und der Realisierung selber vorgestellt.

Die Arbeit stützt sich dabei auf wissenschaftliche Artikel, Bücher, Arbeiten und den Entwicklern (Dokumentationen/Handbücher). Diese werden in den einzelnen Kapiteln kenntlich gemacht und sind im Literaturverzeichnis aufgelistet.

Das Ergebnis der Arbeit ist eine Testsuite, welche ohne Programmierkenntnisse verwaltet, gewartet und erweitert werden kann. Mit QF-Test ist auch die Erstellung einer Testsuite ohne Programmierkenntnis möglich, aber nur wenn die zu testende Anwendung ohne Probleme erkannt und ausgeführt wird. Bei komplexeren Problemen sind Programmierkenntnisse von Vorteil.

Die Testsuite wird auch evaluiert und eventuelle Probleme werden erörtert. Zukünftige Softwareprojekte können davon profitieren, da in dieser Arbeit auftretende Schwierigkeiten mit potentiellen Lösungen präsentiert werden.

Inhaltsverzeichnis

1 Einführung

Bei der Softwareentwicklung kommt es immer zu Fehlern, jedoch sind nicht alle Fehler schwerwiegend. Als Ursache für Softwarefehler kommen verschiedene Aspekte in Frage wie beispielsweise [Wü15]:

- **Kommunikation:** Wenn der Entwickler die Spezifikation falsch versteht oder die Spezifikation fehlerhaft/unvollständig/nicht vorhanden ist.
- **Zeit:** Projekte haben oft eine Deadline und um diese einzuhalten, wird unter Umständen schnell und unsauber entwickelt. Die Zeit, um die Softwaretests durchzuführen, kann zudem fehlen.
- **Mangelnde Softwaretests:** Softwaretests werden oft vernachlässigt. Mögliche Gründe dafür sind: Zeitmangel, fehlendes Budget und dass den Softwaretests eine geringe Bedeutung beigemessen wird.
- **Komplexität:** Die Anforderungen für eine Software steigen stets, weshalb Softwareprojekte immer komplexer werden. Entwickler können den Überblick verlieren und Zusammenhänge nur mit viel Zeitaufwand nachvollziehen. Außerdem steigt die Anzahl der Softwarefehler (mehr Code führt zu weiteren potentiellen Fehlerquellen).
- **Umgebung:** Die Entwicklungs-/Testumgebung kann unterschiedlich zur Einsatzumgebung sein, sodass sich die Software beim Kunden anders verhält. Auch die Datenmenge, Datenart und Ausführung kann variieren, sodass nicht getestete Kombinationen zu Fehlern führen.
- **Wiederverwendung von Code:** Der kopierte Code kann unter Umständen anders laufen als erwartet. Beispielsweise wenn der kopierte Code mangelhaft dokumentiert wird. Zudem wird dieser Code vom Entwickler oft nicht explizit getestet.
- **Menschlicher Faktor:** Projekte werden immer komplexer und Menschen haben eine Fähigkeitsgrenze. Des Weiteren kann Unwissenheit oder Flüchtigkeit zu Fehlern führen.

Es gibt viele Beispielfälle, die zeigen, dass die von uns genutzte Software oft nicht fehlerfrei funktioniert und darüber hinaus Sicherheitslücken aufweisen kann. [Kin15] Beispiele dafür sind [Huc15]:

- **Ariane 5, Explosion:** Im Juni 1996 ist eine unbemannte Rakete (Ariane 5) nur 40 Sekunden nach dem Start explodiert. Die Rakete hatte ihren ersten Flug nach einem Jahrzehnt Entwicklung bei Kosten von 7 Milliarden Dollar. Die Rakete und ihre Fracht werden selbst auf 500 Millionen Dollar geschätzt. Grund für die Explosion war ein Softwarefehler und zwar wurde eine 64-Bit-Gleitkommazahl, welche für die horizontale Geschwindigkeit der Rakete verantwortlich war, in eine 16-Bit (signed) Zahl umgewandelt. Die 64-Bit Zahl war in diesem Fall größer als 32,767 (die größte Zahl, die sich in einem 16-Bit signed Integer abspeichern lässt), weshalb die Umwandlung scheiterte. Das führte zu ungeplanten Kursabweichungen und Korrekturen, die zur Folge hatten, dass die Rakete letztendlich durch das Neutralisationsprogramm gesprengt wurde. Der Fehler wurde nicht gefunden, weil dieser Codeabschnitt von der Ariane 4 für Ariane 5 wiederverwendet wurde. Bei der Ariane 4 wurden so hohe Zahlen/Geschwindigkeiten nicht erreicht, weshalb vorher kein Fehler aufgetreten ist. Fehlerursache in diesem Beispiel ist dementsprechend: Wiederverwendung von Code.

- **Mars Climate Orbiter, Verlust:** Am 23. September 1999 hat die NASA den Kontakt mit dem Mars Climate Orbiter auf dem Weg zum Mars verloren. Die Sonde ist wegen einem Navigationsfehler zerstört worden. Ursache dafür war, dass der Impuls von der NASA in metrischen Einheiten (Newton) berechnet wurde, während das Navigationssystem der Sonde den Impuls in imperiale Einheiten (Pound-force/Kraftpfund) berechnet hat. Dadurch war der Mars Climate Orbiter in einer dichten Atmosphäre des Mars, sodass die Sonde durch die Hitze zerstört wurde. Fehlerursache in diesem Beispiel: Kommunikation.

- **Therac-25, Überdosierung:** Von 1985 bis 1987 wurde ein Bestrahlungsgerät namens Therac-25 eingeführt und angewandt. In diesem Zeitraum wurden 6 Fälle mit einer massiven Überdosis bekannt, welche zu Tod oder schweren Verletzungen geführt haben. Ermittlungen haben ergeben, dass eine Race-Condition die Ursache dafür war. Das Bestrahlungsgerät konnte durch diese Race-Condition so eingestellt werden, dass die Strahlen im leistungsstarken Modus gestrahlt haben, während wichtige Teile intern nicht in Position waren. Fehlerursache in diesem Beispiel: Umgebung und menschlicher Faktor.

- **Bahnhof Hamburg Altona, Ausfall:** Am 12.03.1995 wurde ein elektronisches Stellwerk beim Bahnhof Hamburg Altona in Betrieb genommen. Es kam zu ungeplanten selbstständigen Sicherheitsabschaltungen des Systems. Der Zugverkehr wurde aus Sicherheitsgründen eingestellt und eine Fehlersuche wurde eingeleitet.

Dies führte zu erheblichen Verspätungen im gesamten Bahnverkehr. Ursache für die selbstständige Sicherheitsabschaltungen war ein zu kleiner Speicher. Der Speicher war für die Anzahl der Stellaufträge zu gering, weshalb es unter bestimmten Bedingungen regelmäßig zu einem Überlauf kam. Weiterhin war die Routine, welche den Überlauf behandelt, falsch programmiert, was zu einer Endlosschleife führte. Fehlerursache in diesem Beispiel: Umgebung und menschlicher Faktor.

- **goto fail, Sicherheitslücke:** Mit einem Sicherheitsupdate von Apple (iOS[1] Version 7.0.6) wurde eine Sicherheitslücke mitgeliefert. Bei einer Verbindung wird, anstatt zu überprüfen ob der Schlüssel für eine gesicherte Verbindung korrekt ist, jeder beliebige Schlüssel akzeptiert. Somit ist es nicht möglich für das Gerät einen Betrug zu erkennen. Es war für unberechtigte Benutzer möglich Daten mitzulesen (Online-Banking, soziale Netzwerke und vieles mehr). Der Fehler war ein doppeltes 'goto fail' nach einem if-Statement. Dies war in diesem Fall problematisch, da keine Strukturierung durch Blöcke (mit geschweiften Klammern) stattfand. Somit wurde das doppelte 'goto fail' unabhängig vom if-Statement in jedem Fall ausgeführt. Im konkreten Fall sah der Codeabschnitt folgendermaßen aus (fehlerhaft ist die Zeile 6) [Kin15]:

  Listing 1.1: goto fail sslKeyExchange.c Ausschnitt

```
1     .....
2   if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
3   goto fail;
4   if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
5   goto fail;
6   goto fail;  /* Fehlerhaft! Doppeltes goto fail wird ausgefuehrt
        unabhaengig vom if-Ergebnis */
7   if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
8   goto fail;
9   .....
```

  Fehlerursache in diesem Beispiel: Wiederverwendung von Code und menschlicher Faktor.

Diese und viele weitere Vorfälle zeigen, dass Software nicht restlos zuverlässig ist. Softwarefehler können viel Schaden anrichten, vom hohen finanziellen Verlust bis hin zur Schädigung einer Person. Daher werden Bemühungen, diese zu minimieren, stets verstärkt. Jedoch wird vor allem bei Nutzeranwendungen Features über Sicherheit gestellt. Das liegt daran, dass für viele Nutzer Features wichtiger sind als Sicherheit. Daraus resultiert, dass Entwickler mehr Features entwickeln müssen und die Deadlines immer knapper werden - die Menge zum Testen erhöht sich und die verfügbare Zeit

1 Apple Betriebssystem (Operating System)

verringert sich. [Kin15]

Je mehr Zeilen Code (Lines of Code) eine Software besitzt, umso mehr Fehlerquellen hat die Software. Werden diese nicht behandelt, kann es zu schwerwiegenden Fehlern kommen. Folgendermaßen sieht die (ungefähre) Fehlerrate aus [McC04]:

- Eine durchschnittliche Software hat 15 bis 50 Fehler pro 1 000 Zeilen Code.
- Eine ungetestete Microsoft-Anwendung hat 10 bis 20 Fehler pro 1 000 Zeilen Code.
- Eine gute (und getestete) Software hat bis zu 2 Fehler pro 1 000 Zeilen Code. Eine getestete Anwendung von Microsoft hat noch 0,5 Fehler pro 1 000 Zeilen Code.
- Eine Space Shuttle-Software hat 1 Fehler (oder weniger) pro 10 000 Zeilen Code.

Jeder 10. Fehler davon soll schwerwiegend sein. [Wü15] Die NASA schafft es 500.000 Zeilen fehlerfreien Code zu schreiben, dafür fallen jedoch sehr hohe Kosten an (mehrere tausend Dollar pro Zeile Code). [McC04]

Jede Permutation einer Software zu testen ist sehr aufwendig. Schon einfache Programme können bis zu 100 oder 1 000 mögliche Eingabe- und Ausgabekombinationen haben. Um Testfälle für jede Kombination zu erstellen, ist dementsprechend viel Aufwand nötig. Die Umsetzung, um komplett fehlerfreien Code zu schreiben, ist also (vor allem für komplexe Software) sehr teuer und für viele Unternehmen nicht rentabel. [GjM11] Um wirtschaftlich zu testen, ist es wichtig, dass der Tester die Software und ihre Grenzen kennt. So kann er mit den Grenzwerten arbeiten und eventuelle Fehler auslösen. [Pat05]

Softwaretests werden mit der Zeit immer einfacher aber auch immer schwieriger. Es werden stetig neue Betriebssysteme, Programmiersprachen und Hardwaresysteme entwickelt, welche das Softwaretesten erschweren. Software umfasst immer mehr Geräte, welche in der heutigen Zeit alltäglich eingesetzt werden. Einfacher werden die Softwaretests, weil zunehmend mehr Softwareprogramme existieren, die das Testen erleichtern oder teilweise übernehmen. Die Programmiersprachen und Betriebssysteme werden stets weiterentwickelt und entdecken auch Fehler. Es gibt viele bewährte Routinen, die eine Software nach Defekten überprüfen kann. Der Tester oder Entwickler muss somit nicht alles von Grund auf selbst entwickeln. Viele Elemente und Features lassen sich einfach von einer Bibliothek implementieren, die schon gründlich getestet wird. [GjM11]

In der Industrie werden Softwaretester immer häufiger gebraucht. Selbst in kleinen Unternehmen, die lediglich Plugins für eine Software schreiben, kann eine Testumgebung viel Zeit und Geld sparen.

1.1 Problembeschreibung

Werden bei einem Unternehmen regelmäßig neue Versionen, Updates und Plugins entwickelt, kann es zu Softwarefehlern kommen, vor allem wenn diese nicht getestet werden. Auch wenn die neuen Funktionen/Veränderungen der aktuellsten Version (oder des Updates/Plugins) explizit manuell getestet werden, können noch Fehler enthalten sein. Das liegt daran, dass ein Update/Plugin eine Wirkung außerhalb der neuen Funktionen haben kann und somit die bisherigen Funktionen beeinträchtigt werden. Zudem ist es ein zeitintensiver Prozess, verstärkend wenn die Zeitintervalle für die neuen Versionen/Updates/Plugins gering ausfallen.

Hat das Unternehmen nun ein kleines Team ohne einen Softwaretester, müssen diese manuellen Tests regelmäßig von Mitarbeitern, die einer anderen Tätigkeit nachgehen, durchgeführt werden. Das Softwaretesten liegt somit nicht in ihren Kompetenzen. Dies führt dazu, dass die Tests unstrukturiert manuell ausgeführt werden. Funktionen können in so einem Fall mehrfach (was Zeit und Geld kostet) oder auch gar nicht (kann zu unentdeckten Fehlern führen) getestet werden.

Wird ein/e fehlerhafte Version/Update/Plugin eingespielt, kann es zu Ausfällen des Betriebs kommen (Mitarbeiter können beispielsweise ihrer Tätigkeit nicht nachkommen, wenn eine wichtige Funktion der Software nicht mehr funktioniert). Ein sogenannter Rollback (Zurücksetzen der Software auf eine vorherige funktionierende Version) kann, je nach Software, Umfang und Anzahl der betroffenen Systeme, Stunden oder Tage dauern. Die Software wäre in diesem Zeitraum nicht nutzbar, was sich besonders negativ auswirkt, wenn die Arbeiten zentral über diese Software laufen und Projekte mit einer Deadline nicht weiter bearbeitet werden können. Sind Mitarbeiter gezwungenermaßen untätig, verliert der Arbeitgeber durch diesen Ausfall Zeit und Geld (Lohn für die Mitarbeiter ohne Fortschritt).

Übernehmen die Entwickler das Testen, ist eine Fehlerprävention zwar vorhanden, jedoch sind einzelne Entwickler in einem großen Team oft nicht mit jeden Teil der Software vertraut. Für viele Module sind oft verschiedene Entwickler zuständig. Die gesamte Software (und wie die einzelnen Module zusammenarbeiten) kann daher undurchsichtig sein. Außerdem sind viele Entwickler nicht als Softwaretester geschult oder zertifiziert, weshalb die Zeit beim Programmieren meist besser investiert ist.

Ähnlich ist es bei KC[1] Becker AG beziehungsweise Hexagon. Hexagon entwickelt eine Software, welche von KC Becker AG genutzt wird. Diese Anwendung heißt GeoMedia SmartClient Kommunal. Für diese Software werden regelmäßig (teilweise wöchentlich) von KC Becker AG selbst Plugins entwickelt, welche dann integriert werden. Da kein Tester involviert ist, werden hier jedoch die Plugins/neue Versionen zuerst in ein Testprojekt eingespielt, welches dann von jedem Mitarbeiter genutzt und getestet werden

1 Kommunal Consult

soll. Da auf diese Weise nicht effizient getestet wird, können Fehler unentdeckt bleiben und dementsprechend erst entdeckt werden, wenn die neue Version auf jedes Projekt eingespielt wird. Das kann dazu führen, dass Deadlines nicht (oder nur knapp) eingehalten werden können.

Da eine neue Version von GeoMedia SmartClient in Entwicklung ist, besteht die Aufgabe darin, diese zu testen. Das sollte möglichst mit einer Testautomatisierung realisiert werden, damit ein Grundstein für weitere Tests gelegt ist.

Ein anderes Problem ist, dass das Testen großer Projekte oft nicht mit jeder Testmethode funktioniert. Der Code muss oft angepasst werden, was bei großen etablierten Projekten viel Zeit in Anspruch nehmen kann.

1.2 Ziele dieser Arbeit

Ziel der Arbeit ist es, den Testvorgang eines Unternehmens maßgeblich zu verbessern. Es wird eine nachhaltige Testautomatisierung für ein Softwareprojekt eingeführt, welche mit wenig Kenntnissen ausgeführt werden kann.

Dafür werden verschiedene Testmethoden (sowie Software zur Testautomatisierung) angeschaut und die passende ausgewählt und angewandt. Die Arbeit beschäftigt sich dabei vor allem mit QF-Test[1], da die Testautomatisierung der Anwendung (GMSC-K[3]) mit QF-Test realisiert wird.

Es wird die neuste (noch nicht veröffentlichte) Version des Softwareprojekts als Testobjekt genutzt. Diese Version hat keine Plugins/Addons, sodass nur die Basisfunktionen getestet werden. So wird eine Grundlage und Testumgebung für das weitere Testen geschaffen. Diese Grundlage soll beliebig erweiterbar sein, damit automatische Tests für die betriebsinternen Plugins möglich sind. Es sollen außerdem die Fehler der neuen Version gefunden, dokumentiert und kommuniziert werden.

Um möglichst viele Funktionen zu testen, wird eine Excel-Datei, in der Testfälle dokumentiert sind, abgearbeitet. In dieser Datei sind mehr als 3 000 Zeilen Testfälle dokumentiert (auch von vorherigen Releases). Dabei wird nicht jeder Testfall automatisiert getestet. Diese Vorgänge sollen ohne Veränderungen am Code stattfinden, weshalb die Testautomatisierung hauptsächlich mit der GUI der Software arbeiten wird. Da Vorkenntnisse der Anwendung beim Erstellen einer Testautomatisierung fundamental sind, wird die Anwendung ebenfalls vorgestellt. Das Ergebnis der Arbeit sollte idealerweise folgende Punkte verbessern:

- **Fehlerrate verringern**

1 Quality First Test (GUI[2]-Testtool für Java & Web)
3 GeoMedia SmartClient Kommunal

- **Zeitaufwand verringern**
- **Wartbarkeit verbessern**
- **Mitarbeiter entlasten**

Dabei werden folgende Fragestellungen untersucht und versucht zu beantworten:

- **Wie entwickelt man eine Testautomatisierung in der Black- beziehungsweise Gray-Box für eine bereits etablierte Anwendung ohne die interne Struktur der Anwendung zu verändern?**
- **Welche Werkzeuge sind dafür nötig? Gibt es Alternativen?**
- **Sind Programmierkenntnisse bei der Erstellung einer Testautomatisierung zwingend erforderlich?**
- **Rechtfertigt die Zeitersparnis den Aufwand eine Testautomatisierung zu entwickeln und zu pflegen?**
- **Was ist von einem Testprozess zu erwarten? Was genau ist der Mehreffekt vom Testen?**
- **Reicht eine Testautomatisierung aus, um alle Testfälle abzudecken und alle potentiellen Fehler zu finden?**
- **Lässt sich die erstellte Testautomatisierung auch ohne den Entwickler/Ersteller durchführen und auswerten?**

1.3 Abgrenzung

Das in dieser Arbeit vorgestellte Konzept (sowie die Realisierung) wird vordergründig für die Bedürfnisse der betroffenen Unternehmen (KC Becker AG und Hexagon) erstellt. Daher ist die Realisierung mit QF-Test in diesem Projekt nicht auf jedes beliebige Projekt übertragbar. Das Konzept kann jedoch möglicherweise passen und somit übernommen werden.

Diese Arbeit befasst sich nicht lückenlos mit den theoretischen Aspekten des Testens. Das bedeutet, dass nicht auf jede mögliche Testmethode/Testansätze eingegangen wird und/oder genauer erläutert wird. Es wird nicht jede alternative Software zu QF-Test genannt und die genannten Alternativen werden nicht vollständig beschrieben. Auch von QF-Test wird nicht jede Funktion erwähnt bzw. komplett beschrieben.

In dieser Arbeit werden vor allem, aber nicht ausschließlich, die Funktionen behandelt, die zum Lösen des Problems nötig sind. Außerdem wird der Quellcode von QF-Test

nicht weiter erläutert und es wird kein Code für eine Testautomatisierungssoftware präsentiert. Des Weiteren wird auch kein Quellcode der Software, die getestet wird, veröffentlicht. Das liegt auch daran, dass die Tests nicht in der White-Box (sondern in der Gray-/Black-Box) erstellt und durchgeführt werden.

Da bereits eine Testspezifikation existiert, wird in dieser Arbeit keine erstellt. Die vorhandene Testspezifikation wird abgearbeitet und erläutert. Eine lückenlose Erläuterung der Testschritte ist jedoch nicht möglich, da die Testspezifikation mehr als 3 000 Zeilen beinhaltet. Auch die getestete Software wird nicht vollständig beschrieben. Es wird nicht auf jede Funktion eingegangen und nicht alle Module und Plugins werden ausführlich geschildert.

Es werden keine Unit-Tests gemacht und daher wird kein Quellcode für das Testen optimiert. Es wird jedoch erläutert was Unit-Tests sind. Die Tests laufen größtenteils über die GUI. Es werden Events erstellt, die Mausklicks, Mausbewegungen und Texteingaben ausführen. Daraufhin folgen dann Abfragen (beispielsweise Textabfragen oder Bildvergleiche) mit den zu erwartenden Werten. Abweichungen werden dabei als Fehler dokumentiert.

1.4 Vorgehensweise

Der erste Schritt ist das Kennenlernen der zu testenden Software. Hierfür werden alltägliche wirtschaftliche Arbeiten (welche mit der Anwendung erledigt werden) übernommen, um so die Funktionsweise, den Nutzen und den Hintergrund der Software zu verstehen. Ein fundamentales Wissen über die Anwendung ist beim Testen unverzichtbar.

Dann folgt die Rücksprache mit dem Entwickler. Bei dieser Rücksprache soll das Verständnis über die Software vertieft werden. Es soll außerdem die neue Version (und die Veränderungen) präsentiert und auf eine passende Testumgebung installiert werden. Die Testspezifikation wird ebenfalls in diesem Gespräch definiert beziehungsweise vorgestellt. Eventuelle Probleme und Anforderungen werden spezifiziert.

Nach der Rücksprache wird eine an die Anforderung passende Testsoftware ermittelt. Empfohlene Testanwendungen der Entwickler werden dabei besonders berücksichtigt. Sobald eine passende Testsoftware ermittelt wird, ist eine weitere Rücksprache erforderlich. Hierbei wird die ausgewählte Testsoftware kurz vorgestellt, um eine Bestätigung der Vorgesetzten zu erhalten. Mit der passenden (und bestätigten) Testsoftware wird die Testspezifikation abgearbeitet. Dabei werden Testfallsätze mit ihren Testfällen und Sequenzen erstellt.

Da die neuste Version viele interne Veränderungen hat, wird die Testautomatisierung mit der aktuell genutzten Version erstellt. Das hat den Vorteil, dass die Tests auf einer bereits bewährten Version erstellt werden und somit Fehler nicht fälschlicherweise als korrekt angesehen werden können. Die fertig erstellte Testsuite soll dann mit der neuen

Version durchgeführt werden, um Abweichungen und Fehler zu finden. Bei weiteren Problemen und Schwierigkeiten mit der neuen Version wird immer Rücksprache mit den Entwicklern gehalten. Alle Fehler werden dokumentiert und dementsprechend kommuniziert.

Da sich nicht alle Testfälle von der Testspezifikation mit QF-Test ohne sehr großen Aufwand erstellen lassen, werden die Testfälle auch manuell durchgeführt. Das liegt daran, dass manche Testfälle eine Vielzahl an Daten ändern. Dieser Prozess kann sehr zeitintensiv sein. Eine Testautomatisierung mit unzähligen Abfragen würde dann nicht nur bei der Erstellung sondern auch bei der Ausführung impraktikabel sein.

1.5 Struktur der Arbeit

Diese Arbeit ist in sechs Hauptkapitel gegliedert. Im ersten Kapitel wird das Thema eingeleitet. Es beschäftigt sich mit der Motivation für die Erstellung dieser Arbeit. Außerdem werden in diesem Kapitel die Ziele, welche erreicht werden sollen, beschrieben. Die Vorgehensweise, wie diese Ziele erreicht werden, wird auch erläutert.

Im nächsten Kapitel (Kapitel 2) wird ein Grundwissen vermittelt. Dieses Wissen ist notwendig, um die Arbeit nachzuvollziehen. Es wird beschrieben, was Testen ist und welche Rolle Softwaretests bei der Entwicklung einnehmen. Zudem werden mehrere Testmethoden vorgestellt und erläutert. Dabei wird vor allem der Testautomatisierung eine große Beachtung geschenkt.

In Kapitel 3 wird das Konzept zusammen mit der Anforderungsanalyse vorgestellt. Dafür werden die relevanten Funktionen von QF-Test (aber auch von anderen Testwerkzeugen) gezeigt und demonstriert.

Das im dritten Kapitel vorgestellte Konzept wird dann in Kapitel 4 umgesetzt und beschrieben. Es werden Testfälle als Beispiel gezeigt und beschrieben. Die Testsuite wird erläutert und eventuell benötigter Code präsentiert.

Die Realisierung wird dann in Kapitel 5 evaluiert. Es wird auf die Laufzeit einzelner Testfälle eingegangen. In diesem Kapitel wird auch versucht zu ermitteln, in welchem Umfang die Testabdeckung ausfällt. Außerdem wird auf die Abstrahierbarkeit eingegangen. Des Weiteren wird auch QF-Test einer Bewertung unterzogen.

Im 6. und letzten Kapitel wird ein Fazit getroffen. Es wird auf den aktuellen Stand der Arbeit eingegangen und daraufhin auf den Ausblick (weitere Schritte und Möglichkeiten).

2 Hintergrund

In diesem Kapitel werden vor allem die theoretischen Grundlagen erläutert. Diese sind notwendig, um den weiteren Verlauf der Arbeit nachzuvollziehen. Besonders beim Softwaretesten ist es wichtig zu verstehen, was genau erreicht werden soll. Deshalb ist eine klare Definition wichtig, die in diesem Kapitel erarbeitet wird.

Dann werden verschiedene Testmethoden beziehungsweise Testarten vorgestellt. Auch die zu testende Anwendung wird in diesem Kapitel beschrieben. Um den Rahmen der Bachelorthesis nicht zu sprengen, wird der Umfang dieses Kapitels reduziert. Ein erweitertes Hintergrund-Kapitel lässt sich bei [Tho17] finden.

2.1 Softwaretests

Auch wenn Softwaretesten eine technische Aufgabe ist, spielt die menschliche Psychologie eine Rolle. Da Menschen sehr zielorientiert arbeiten, ist eine exakte Definition besonders wichtig. Definiert der Softwaretester eine falsche Interpretation des Softwaretesten für sich, arbeitet dieser womöglich auf das falsche Ziel hin. [GjM11]

2.1.1 Definition und Psychologie

Es gibt verschiedene Definitionen vom Softwaretesten, davon führen viele aber nicht zum gewünschten Ziel wie beispielsweise folgende Definitionen [GjM11]:

- Testen ist ein Prozess, welcher demonstriert, dass keine Fehler vorhanden sind.
- Testen soll nachweisen, dass die Anwendung und deren Funktionen ordnungsgemäß ablaufen.
- Testen ist ein Prozess um sicherzustellen, dass die Anwendung so funktioniert wie sie funktionieren sollte.

Die oben genannten Definitionen sind falsch, da sie genau das Gegenteil vom gewünschten Ziel bewirken. Das Testen verfolgt das Ziel, die Anwendung zu verbessern (Zuverlässigkeit und/oder Qualität).

Um das zu realisieren, muss man Fehler finden und beseitigen. Daher wird beim Testen nicht gezeigt, dass die Anwendung funktioniert oder keine Fehler enthält. Vielmehr sollte man davon ausgehen, dass Fehler vorhanden sind. Ziel ist es dann, so viele Fehler wie möglich zu finden und zu beheben. Die richtige Definition demzufolge lautet also [GjM11]:

- **Testen ist der Prozess, bei dem die Anwendung ausgeführt wird, mit der Absicht Fehler zu finden.**

Auch wenn es zunächst den Anschein erweckt, dass die Definitionen ähnlich oder identisch sind, liegt ein gravierender Unterschied vor. Arbeitet ein Softwaretester nach einer falschen Definition, will er demonstrieren, dass die Software keine Fehler enthält. Dies führt dazu, dass der Softwaretester unbewusst genau auf dieses Ziel hinarbeitet und somit nur Testdaten auswählt, die keine (oder nur eine geringe) Wahrscheinlichkeit haben einen Fehler zu finden. Aber verfolgt der Softwaretester die Absicht zu demonstrieren, dass Fehler in einer Anwendung vorhanden sind, werden die Testdaten dementsprechend ausgewählt und die Wahrscheinlichkeit Fehler zu finden ist höher.

Ein weiterer Aspekt beschäftigt sich mit der Frage, wann ein Test erfolgreich oder erfolglos ist. In vielen Fällen wird davon ausgegangen, dass ein Testfall, mit dem ein Fehler gefunden wird, nicht erfolgreich ist und Testfälle ohne gefundene Fehler werden dagegen als erfolgreich angesehen. Doch diese Ansicht oder Definition kann auch in diesem Fall zu falschen Ergebnissen führen.

Nicht erfolgreich zu sein ist meistens nicht erstrebenswert. Daher sollten Testfälle, die Fehler finden, als erfolgreich angesehen werden. Diese Testfälle sind aber auch erfolgreich, wenn sie keine Fehler entdecken (sofern die Testfälle gründlich erstellt und ausgeführt werden). Die einzigen Testfälle, die als nicht erfolgreich angesehen werden sollten, sind Testfälle, welche die Software nicht gründlich genug überprüfen und dementsprechend keine Fehler finden.

Findet ein Testfall neue Fehler, handelt es sich um Fortschritte, da die Qualität der Anwendung (nach der Fehlerbehebung) verbessert wird. Somit zahlt sich die Investition in Softwaretests mit diesem Testfall aus und kann daher als erfolgreich eingestuft werden. [GjM11]

Ein weiteres Problem mit einer falschen Definition ist, dass es unmöglich ist, dieses Ziel zu erreichen. Zu demonstrieren (und zu garantieren), dass keine Fehler vorhanden sind, ist selbst bei einfachen Anwendungen nahezu ausgeschlossen. Scheint eine Aufgabe für einen Menschen unmöglich, sinkt die Produktivität. [GjM11] Da es nahezu aussichtslos ist, alle Fehler zu finden, ist es wichtig, sinnvolle Testfälle mit möglichst vielen Grenzfällen und Grenzwerten zu erstellen. So lassen sich viele Situationen herbeirufen, die zu eventuellen Fehler führen können. [Wü15]

Viele Entwickler sind erleichtert und stolz, wenn die entwickelte Anwendung funktioniert. Das hat zufolge, dass die Entwickler ihre eigene Anwendung unter Umständen

nicht gründlich genug testen. Unvorhersehbare Eingaben und Ausführungen werden eventuell nicht beachtet, da Entwickler konstruktiv tätig sind. Der Entwickler kennt sein eigenes Programm und die Ausführung, was dazu führen kann, dass nur der Entwickler die Anwendung benutzen kann, wenn Beschreibungen und Anweisungen fehlen. Problemsituationen können unbewusst vermieden werden, da gefundene Fehler für den Entwickler Mehraufwand bedeuten. [Wü15]

Daher sollten Tests nicht beziehungsweise nicht ausschließlich vom Entwickler durchgeführt werden. Denn der Entwickler selbst ist nicht in der Lage seine Anwendung in der Black-Box zu testen.

2.1.2 Black-Box-Test

Bei Black-Box-Tests ist die interne Struktur und der Code nicht bekannt. Bei dieser Testmethode verfügt der Tester kein Wissen darüber wie die Funktionen implementiert sind. Die Testfälle werden dementsprechend funktionsorientiert entwickelt. Dabei wird die Spezifikation mit ihren Anforderungen als Grundstein genutzt. Die Anwendung wird also als Black-Box betrachtet. [Ren12]

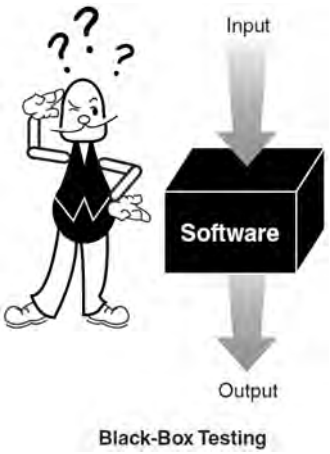

Abbildung 2.1: Black-Box Testing - Tester kennt die interne Struktur nicht. [Pat05]

Dies hat vor allem den Vorteil, dass die Anwendung auch von Softwaretestern geprüft werden kann, die nicht bei der Entwicklung mitwirken. Auch eine Einweisung in den Quellcode ist beim Black-Box-Test somit nicht nötig beziehungsweise sogar nicht erwünscht. Wird die Spezifikation vom Entwickler falsch verstanden, so wird dies schnell vom Tester entdeckt, da die Testfälle auf die Spezifikation aufbauen und diese verifizieren. [Wü15] Allerdings besteht der Nachteil darin, dass der Tester die Schwächen und Stärken des Quellcodes nicht kennt. Somit werden eventuell kritische Codeteile nicht übermäßig revidiert. [Wü15]

Da Black-Box-Tests Input- und Output-basiert sind, ist eine komplette Abdeckung der Möglichkeiten ausgeschlossen. Um potenzielle Fehler zu finden, müsste man jede

gültige Eingabe durch die Anwendung laufen lassen. Dies könnte bei einer Integer-Eingabe eine sehr große Anzahl an Testfällen sein. Die genaue Anzahl an Testfällen unterscheidet sich bei den gültigen Eingaben, da die Größe je nach Programmiersprache unterschiedlich ausfällt (Integer bei Java von -2147483648 bis 2147483647).

In der Black-Box ist es in der Theorie nötig, jede mögliche Eingabe zu testen, da es je nach Wert spezielle Abfragen und Fälle geben könnte. Um jedoch alle Fehler zu finden, müsste man auch alle ungültigen Eingaben testen. Somit wären auch Testfälle für Buchstaben, Zeichen, Wörter, Zeichenketten und alle möglichen Kombinationen (auch mit Zahlen) erforderlich. Um also eine Anwendung vollständig in der Black-Box zu testen, ist eine unendliche Menge an Testfällen nötig. Eine Anwendung vollkommen zu testen ist also unmöglich, da es auch schon in kleinen Projekten meistens mehr als nur eine Funktion gibt, die ein User-Input erwartet. [GjM11] Dementsprechend ist es auch mit Black-Box-Tests nicht möglich eine Fehlerfreiheit zu garantieren.

Da es aussichtslos ist eine Anwendung vollkommen zu testen, ist es besonders wichtig Testfälle zu entwickeln, welche viele Situationen effektiv abdecken. Das Ziel ist also mit einer endlichen Anzahl von Testfällen möglichst viele Fehler zu finden. [GjM11] Dafür gibt es verschiedene Vorgehensweisen. Ist ein Projekt relativ neu und die Anwendung noch sehr instabil, unterscheidet man erstmal in zwei Vorgehensweisen [Pat05]:

- **Testing to Pass:** Bei einer Anwendung, die vermutlich noch bei korrekten Eingaben viele potenzielle Fehler enthält, werden Testfälle geschrieben, bei denen erwartet wird, dass die Anwendung diese besteht. Bevor man die Anwendung also auf eventuelle Extremsituationen und Ausnahmefälle testet, wird hiermit sichergestellt, dass die Anwendung in ihrer Grundfunktionalität in Ordnung ist.

- **Testing to Fail:** Bewährt sich die Anwendung beim Testing to Pass, werden beim Testing to Fail Testfälle geschrieben, bei denen erwartet wird, dass die Anwendung Fehler ausgibt. Hier werden Grenzwerte getestet und überschritten. Ziel ist es, die Anwendung möglichst zu strapazieren und zum eventuellen Absturz zu bringen, um so vorhandene Fehler zu finden.

Um ökonomisch und effektiv zu testen, muss aber noch eine Methode vorgestellt werden. Als Beispiel wird eine Anwendung mit einer Benutzereingabe, welche die Zahlen 0 bis 365 (Tage im Kalenderjahr) akzeptiert, präsentiert. Testfälle nach dem Testing to Pass-Prinzip wären in diesem Fall jeder gültige Wert. Testfälle, die zu Fehlern beziehungsweise Fehlerausgaben führen sollten, wären dann Werte über 365 und negative Zahlen. Somit sind viele Testfälle mit gültigen Eingaben möglich. Bei den ungültigen Testfällen lassen sich sogar unendlich viele Testfälle erstellen.

Da aber von vielen Testfällen dasselbe Ergebnis erwartet wird, ist es nicht nötig, jede einzelne Eingabe zu testen. Diese Eingaben sind äquivalent und können somit in eine **Äquivalenzklasse** zugeteilt werden. Mit Äquivalenzklassen wird die unendliche Menge

an Testfällen in eine endliche, aber dennoch effektive Menge reduziert. [Pat05] Folgende Äquivalenzklassen lassen sich für das genannte Beispiel bilden:

- **gültige Äquivalenzklassen:** [0 ... 365]
- **ungültige Äquivalenzklassen:** [INT_MIN ... -1], [366 ... INT_MAX]

Da bei dem Kalendertagen-Beispiel keine großen Zahlen vorgesehen sind, wird von der Spezifikation vorgegeben Integer zu nutzen. INT_MIN und INT_MAX stehen für den jeweiligen minimalen (beziehungsweise maximalen) Wert, den die Programmiersprache für Integer vorsieht. Das sind bei Java und C folgende Werte: INT_MAX = +2147483647 und für INT_MIN = -2147483648. [Wü15] Folgende Testwerte lassen sich für das Beispiel ermitteln:

- **Endgültige Testfälle:** INT_MIN, -1, 0, 365, 366, INT_MAX
- **Davon ungültige Eingaben:** INT_MIN, -1, 366, INT_MAX
- **Davon gültige Eingaben:** 0, 365

Wie sich diese Testfälle beziehungsweise Testwerte systematisch ermitteln lassen, kann man [Tho17] entnehmen.

Es ist möglich, die Grenzwertanalyse mithilfe der Äquivalenzklassen durchzuführen, um so auch die Testwerte zu ermitteln. Mit dieser Methode könnte das folgendermaßen aussehen:

Abbildung 2.2: Grenzwertanalyse mit Äquivalenzklassen

Man erstellt einfach für jeden Grenzwert einer Äquivalenzklasse einen Testfall. Dadurch ergeben sich dieselben Testfälle wie vorher: INT_MIN, -1, 0, 365, 366 und INT_MAX (siehe Abbildung 2.2).

Jedoch gibt es auch Fälle, bei der eine Grenzwertanalyse nicht ausreicht. Es ist durchaus möglich, dass sich Fehler in der Mitte von Äquivalenzklassen aufhalten. Solche Defekte sind in der Black-Box nur schwer zu entdecken, da es für solche Fälle keine Methode gibt. Diese Fehler lassen sich daher (in der Black-Box) nur mit Glück finden. [Wü15]

Da in einer Anwendung nicht nur Zahlen vorkommen und man mit Äquivalenzklassenbildung und der Grenzwertanalyse nicht alles abdecken kann, gibt es noch weitere wichtige Methoden beim Testen in der Black-Box. Das sind insbesondere folgende [MEK12]:

- **Fuzzing:** Beim Fuzzing werden zufällige Eingaben an einer Anwendung getestet.
- **Cause-Effect Graph:** Ein Cause-Effect Graph (oder auch Ursache-Wirkung Graph) ist eine grafische Darstellung (siehe Abbildung 2.3), welche die logische Beziehung zwischen Ursachen und Wirkungen von einer Software-Spezifikation beschreibt. [AP97]

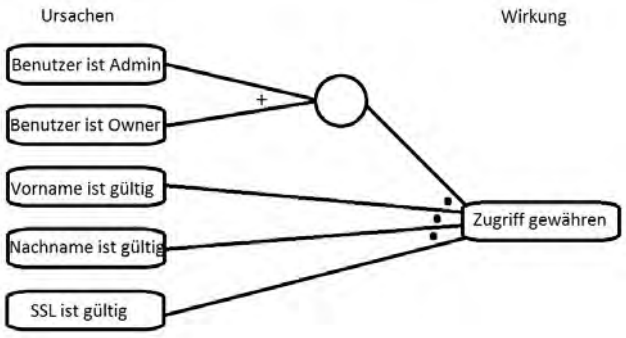

Abbildung 2.3: Cause-Effect Graph

- **Zustandsbezogene Tests**: Die zustandsbezogenen Tests kann man nutzen, wenn die Anwendung als Zustandsautomat aufgebaut ist, aber auch als Navigation für die GUI. Die Anwendung wird als Zustandsdiagramm modelliert und daraus können Testfälle abgeleitet und erstellt werden. Abbildung 2.4 präsentiert ein Zustandsdiagramm für einen Bankautomat.

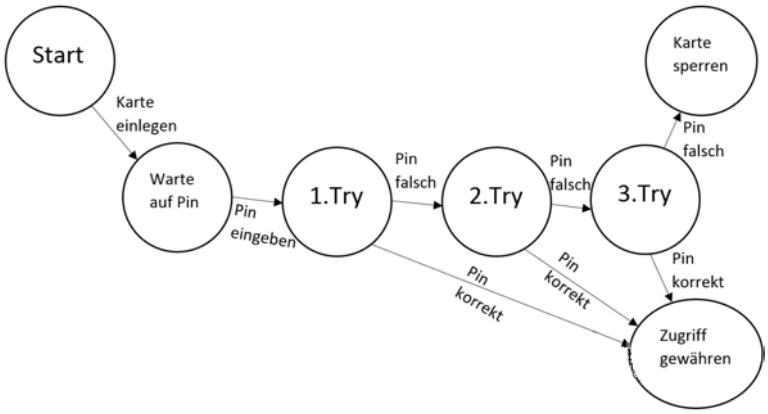

Abbildung 2.4: Beispiel für ein Zustandsdiagramm

- **Orthogonal Array-Tests:** Orthogonal Array-Tests werden genutzt, wenn die Zahl der Inputs relativ klein ist aber dennoch zu groß, um vollständig jede Kombination zu testen. Damit wird die Testabdeckung erhöht während die Zahl der Testfälle gering bleibt. Bei einer Funktion die drei Parameter mit drei verschiedenen Werten erwartet, wären für eine vollständige Abdeckung $3 \cdot 3 \cdot 3 = 27$ Testfälle nötig. Mit Orthogonal Array-Tests wird diese Zahl auf 9 reduziert, da hier jede (paarweise) Kombination nur einmal auftritt.

- **All Pair Testing:** Beim All Pair Testing wird, ähnlich wie beim Orthogonal Array-Test, (hier aber gezielt) jede paarweise Kombination genau einmal getestet. Beide Methoden reduzieren die Anzahl der Testfälle.

Eine detaillierte Beschreibung der Methoden findet man bei [Tho17]. Mit der Black-Box lassen sich so folgende Testverfahren realisieren: Lasttest, Performance-Test, GUI-Test, Usability-Test, Sicherheitstest, Integrationstest, Regressionstest, Funktionstest.

Aber auch mit allen genannten Methoden werden nicht alle Fehler aufgedeckt. Wird dabei außerdem lediglich gegen die Spezifikation getestet und wenig mit den Entwicklern kommuniziert, werden eventuelle weitere Funktionen nicht getestet. Fehler in der Spezifikation selbst können so unentdeckt bleiben. [Wü15] Ergänzend zum Black-Box-Test sollte deshalb auch mit der White-Box getestet werden.

2.1.3 White-Box-Test

Anders als bei der Black-Box kennt man in der White-Box den Quellcode. Sollte der Code dem Tester noch nicht bekannt sein, hat er zumindest Zugriff auf den Quellcode, um so die Testfälle in der White-Box zu ermitteln. Somit richten sich die Tests nach den Eigenschaften und Logiken des Codes. [Hal84]

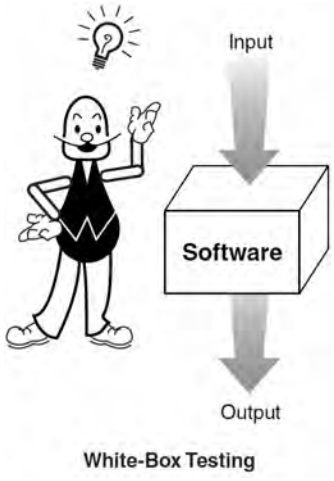

Abbildung 2.5: White-Box Testing - Tester kennt die interne Struktur. [Pat05]

Um in der White-Box effizient zu testen, gibt es bewährte Methoden, da eine komplette Abdeckung des Codes und deren Statements unmöglich ist. [GjM11] Weil in dieser Arbeit nicht mit der White-Box gearbeitet wird, werden diese nicht lückenlos erläutert. Weitere Informationen zu den Methoden findet man unter anderem bei [Tho17]. Folgende Methoden kommen in der White-Box zum Einsatz [MEK12]:

- **Kontrollfluss Testing:** Das Programm wird als Kontrollflussgraph (wie bei Abbildung 2.6) dargestellt, dafür muss die interne Struktur, das Design und der

Code bekannt sein. Mithilfe des Kontrollflussgraphen können dann Zweig-, Pfad- und Anweisungsüberdeckung besser realisiert werden.

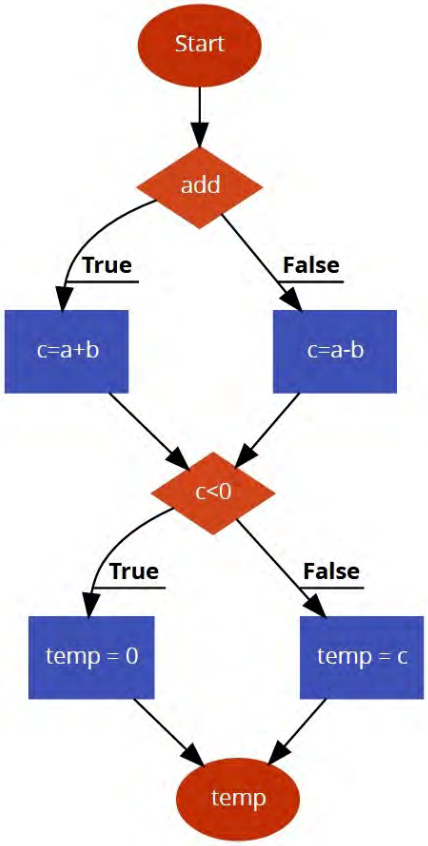

Abbildung 2.6: Kontrollflussgraph

- **Zweigüberdeckung:** Bei der Zweigüberdeckung wird das Ziel verfolgt, jeden Zweig einmal zu durchlaufen. Um das zu realisieren, müssen die Testfälle so entwickelt werden, dass die Bedingungen jeweils einmal true und false ergeben.

- **Anweisungsüberdeckung:** Hier ist es das Ziel, jede Anweisung einmal auszuführen. Das bedeutet, dass jeder Knoten einmal durchlaufen wird. Da es möglich ist, Anweisungen durchzuführen ohne alle Zweige zu durchlaufen, ist die Anweisungsüberdeckung schwächer als die Zweigüberdeckung.

- **Pfadüberdeckung:** Bei der Pfadüberdeckung soll jeder Pfad (vom Startknoten bis zum End- beziehungsweise Ausgangsknoten) ausgeführt werden. Damit deckt man bei der Pfadüberdeckung auch die Zweigüberdeckung ab (und somit auch die Anweisungsüberdeckung). Bei der Pfadabdeckung kommt es zu sehr vielen Testfällen.

- **Bedingungsüberdeckung:** Da bei den Pfad-/Zweig-/Anweisungsüberdeckungen die Bedingungen meist nur wenige Male durchlaufen werden und dabei nicht die Absicht verfolgt, die Bedingung zu testen, ist eine Bedingungsüberdeckung

wichtig. Das Ziel hierbei ist die Bedingung auf Fehler zu testen. Dabei gibt es drei verschiedene Ansätze, die sich in der Stärke unterscheiden.

- **Loop Testing:** Beim Loop Testing liegt der Fokus auf das Validieren der Schleifen. Da sich bei der Pfadüberdeckung die Anzahl der Testfälle durch Schleifen signifikant erhöht, müssen spezielle Verfahren für Schleifen definiert werden. Dabei wird der Umfang mit einer Zahl beschrieben, welche die Anzahl der Ausführung durch eine Schleife bestimmen. [Zel06]

Mit der White-Box sind beispielsweise folgende Tests möglich: Unit-Tests, Code-Metriken, Code Coverage, Boundchecker, Profiler und mehr.

2.1.4 Gray-Box-Test im Vergleich

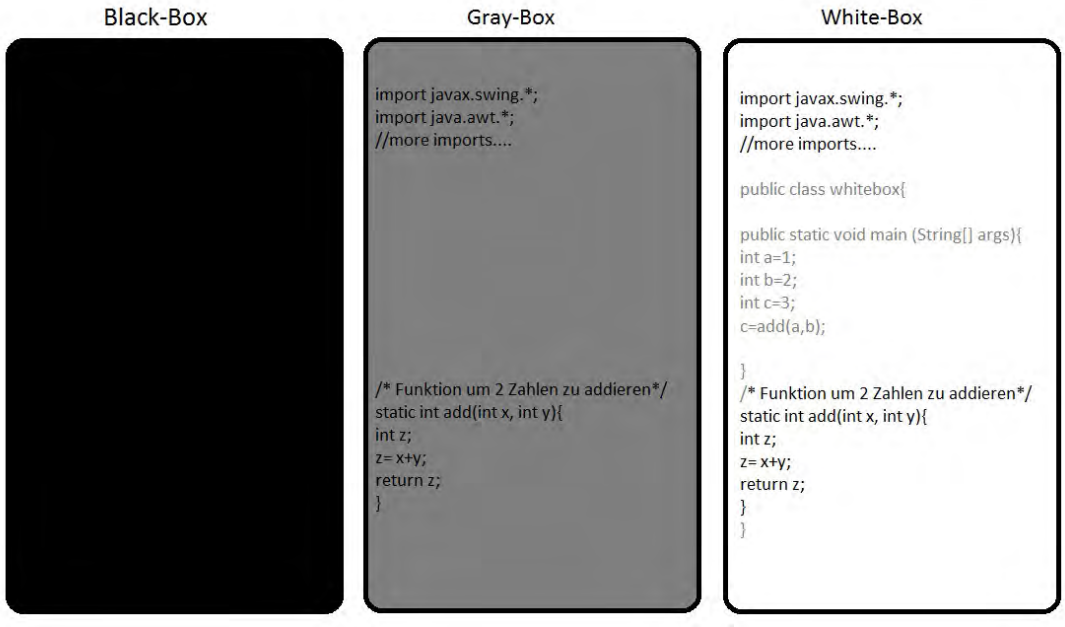

Abbildung 2.7: Kenntnisse der internen Struktur in der Black-, Gray- und White-Box

Gray-Box-Test ist eine Kombination aus der Black- und White-Box (siehe Abbildung 2.7). Die Testabdeckung soll so erhöht werden, da der Tester eventuell Einsicht über kritische Stellen im Quellcode verfügt. Die interne Struktur, die Dokumentation und die genutzten Algorithmen sind dem Tester bekannt. So soll sichergestellt werden, dass mithilfe der Methoden, die aus der Black- und White-Box bekannt sind, möglichst viele Fehler gefunden werden. [MEK12]

Da sich die Testarten unterscheiden, ist ein Vergleich bei einem Projekt essenziell. Um sich für eine Testart (oder Kombinationen von Testarten) zu entscheiden, sollten die Anforderungen klar definiert sein. Mithilfe der Tabelle 2.1 lässt sich dann die passende Methode auswählen.

Tabelle 2.1: Entscheidungstabelle für die Testarten

Black-Box	Gray-Box	White-Box
Kennt die interne Struktur nicht	Kennt die interne Struktur teilweise	Kennt die interne Struktur komplett
Niedrige Detailgenauigkeit	Mittlere Detailgenauigkeit	Hohe Detailgenauigkeit
Kann vom Tester und Benutzer durchgeführt werden	Kann vom Tester, Benutzer und Entwickler durchgeführt werden	Kann vom Tester und Entwickler durchgeführt werden
Weniger zeitintensiv dafür aber niedrigere Abdeckung	Mittlere Abdeckung	Sehr zeitintensiv dafür aber höhere Abdeckung
Testet über die GUI durch Ausprobieren	Interne Grenzen und Strukturen können (wenn bekannt) gezielt getestet werden	Interne Grenzen und Strukturen können komplett getestet werden

Da Gray-Box jedoch einen anderen Blickwinkel als die Black- und White-Box liefert, werden noch andere Techniken genutzt. Eine wichtige Technik, die mit der Gray-Box umgesetzt werden kann, sind die **Regressionstests**.

Als Regressionstest bezeichnet man Tests, die auf eine modifizierte Anwendung ausgeführt werden. Damit soll sichergestellt werden, dass der modifizierte Code funktioniert und den Rest der Anwendung nicht negativ beeinflusst. Somit testet ein Regressionstest die Anwendung vollständig, auch unveränderte Funktionen und Methoden werden geprüft. [GR00]

Da Testfälle bei einem Regressionstest also regelmäßig wiederholt werden müssen (bei Bugfixes, neuen Releases/Versionen und Ähnlichem), ist der Aufwand dementsprechend groß. Um den Aufwand zu verringern, ist eine Automatisierung der Tests von Vorteil. Hier spricht man von der **Testautomatisierung**.

2.2 Testautomatisierung

Die Testautomatisierung entlastet den Tester, indem sie monotone zeitaufwendige Arbeiten übernimmt. In vielen Fällen müssen Testfälle mehr als einmal durchlaufen werden. Wenn beispielsweise ein Testfallsatz Fehler findet und diese behoben werden, muss der Testfallsatz erneut durchgeführt werden. Dies kann je nach Umfang sehr aufwendig sein. [Pat05] Eine Testautomatisierung ist anfangs eine größere Investition, aber der ROI[1] ist meist schon nach dem zweiten bis dritten Testdurchlauf gegeben (siehe Abbildung 2.8).

1 Return on Investment

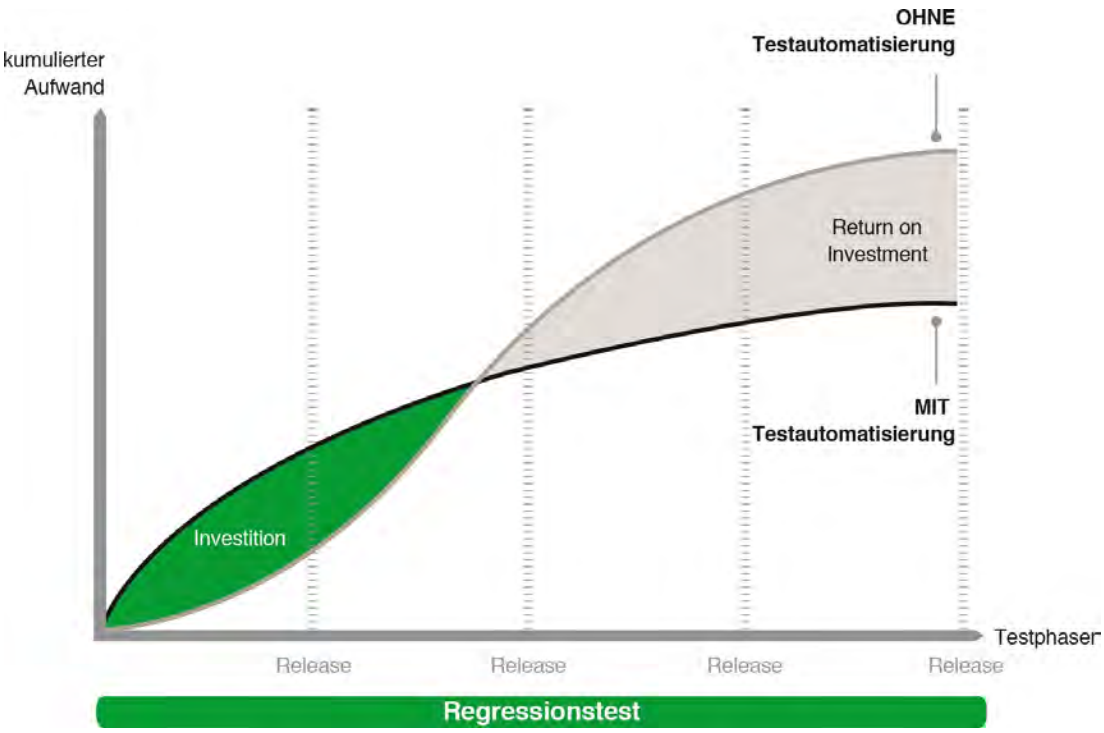

Abbildung 2.8: Return on Investment der Testautomatisierung [Con17]

Insbesondere bei größeren Projekten können Regressionstest ohne eine Testautomatisierung (also manuell) kaum vollständig abgearbeitet werden. Dabei verbessert die Testautomatisierung noch weitere Punkte wie [Pat05]:

- **Geschwindigkeit:** Manuelle Tests dauern länger als bereits erstellte automatische Tests. Während ein Mensch pro einfachen Testfall 5 Sekunden oder mehr benötigt (Testspezifikation lesen, Knöpfe drücken, Ergebnis prüfen), kann eine gute Testautomatisierung den Prozess in einem Bruchteil der Zeit durchführen.

- **Effizienz:** Wenn der Tester mit manuellen Testen beschäftigt ist, kann dieser keiner anderen Tätigkeit nachgehen. Übernimmt eine Testautomatisierung das Testen, ist es dem Tester beispielsweise möglich, neue Tests zu planen und zu entwickeln.

- **Genauigkeit und Präzision:** Führt der Tester die monotonen Tests schon zum wiederholten Male durch, sinkt die Aufmerksamkeitsspanne. Dies führt dazu, dass Fehler möglicherweise übersehen werden. Eine Testautomatisierung arbeitet stets konstant und liefert die gewünschten Ergebnisse (sofern die Testautomatisierung richtig entwickelt wird).

- **Durchhaltevermögen:** Eine Maschine erbringt kontinuierlich dieselbe Leistung ohne zu ermüden.

Eine Testautomatisierung muss jedoch erstmal entwickelt werden. Dies ist eine umfangreiche Aufgabe, welche mit verschiedenen Werkzeugen realisierbar ist. Es stehen jedoch nicht alle Tools kostenfrei zur Verfügung (was auch schon in Abbildung 2.8 miteinbezogen wird).

Je nach Testart und Fachwissen des Testers eignen sich unterschiedliche Werkzeuge. In einer Black-Box eignen sich beispielsweise Testautomatisierungen über die GUI. Diese Art von Test bezeichnet man auch als nichtinvasiv, da keine Änderungen an der Anwendung erfolgen beziehungsweise erfolgen müssen. Muss für das Werkzeug die Anwendung geändert werden (oder ändert das Tool die Anwendung automatisch), spricht man von einem invasiven Tool. Nichtinvasive Tools werden von Testern (und Entwicklern) bevorzugt, da die Testergebnisse so nicht verfälscht werden. Außerdem können Änderungen vom Entwickler nicht erwünscht sein. [Pat05] Test-Tools kann man meist unter folgenden Kategorien einteilen [Pat05]:

- **Driver:** Driver steuern und führen Anwendungen automatisch aus. Die einfachste Art eines Drivers sind Batch-Dateien, welche die Anwendungen automatisch mit verschiedenen Inputs ausführen können. Eine moderne Lösung mit Driver ist Selenium[1] mit WebDriver[2]. [Dev17]

- **Stubs:** Stubs ersetzen Codeteile oder Module, um die Softwaretests effizienter zu machen. Beispielsweise können Druckvorgänge ersetzt werden (spart Papier, Druckerpatrone und Zeit während die Genauigkeit steigt).

- **Stress-Tools:** Die Anwendung wird mithilfe von Tools großem Input ausgesetzt um sicherzustellen, dass die Software auch unter schweren Bedingungen funktioniert.

- **Analyse-Tools:** Mit den Analyse-Tools wird der Entwickler entlastet. Es gibt Analyse-Tools, die den Code nach Fehlern/Schwachstellen absuchen und diesen dazu formatieren.

- **Macro Recorder:** Macro Recorder sind ein wichtiger Schritt zur kompletten Testautomatisierung. Diese können den User-Input aufnehmen/aufzeichnen und dann nach Belieben wiedergeben. Mit den Macro Recodern ist eine Überprüfung der Ergebnisse jedoch nicht möglich.

- **Macro programmieren:** Macros lassen sich nicht nur mit einem Macro Recorder aufnehmen sondern auch programmieren. Dies hat den Vorteil, dass man die Macros genauer einstellen kann. Außerdem kann man die Aufnahme jederzeit

1 Framework für automatisierte Softwaretests von Webanwendungen
2 Driver von Selenium

pausieren und mit entsprechenden Abfragen den Tester auf die Outputs/Ergebnisse hinweisen.

Heutige Tools zur Testautomatisierung vereinen viele der zuvor aufgezählten Funktionen/Tools. Somit ist es möglich, mit nur einem Tool Macros aufzunehmen oder zu programmieren, die gewünschten Ergebnisse zu überprüfen/validieren, die Anwendung zu analysieren und den ganzen Prozess in einer Log-File zu dokumentieren.

Jedoch sind für die verschiedenen Testebenen meist unterschiedliche Tools nötig. Es ist also wichtig eine Testautomatisierung auszuwählen, welche die Anforderungen der Testebene erfüllt (falls überhaupt möglich).

2.3 Testebenen

Die Testebenen beschreiben im Prinzip den Umfang des Testbereichs. Dieser wird, je nach Definition, in drei oder vier Testebenen unterteilt. Diese werden wie folgt genannt:

- **Modultest** oder auch Unit-Test, Komponententest, Entwicklertest
- **Integrationstest**
- **Systemtest**
- **Abnahmetest**

Da der System- und Abnahmetest im Prinzip auf derselben Ebene arbeiten, werden diese oft in einer Kategorie vereinigt. Eine Darstellung mit drei Testebenen kann man Abbildung 2.9 entnehmen.

Idealerweise sollten Tests in jeder Testebene entwickelt und durchgeführt werden. Dies ist jedoch (je nach Entwicklungsstand) nicht immer möglich. Sind beispielsweise Module noch nicht fertig, können Tests in der höheren Ebene nicht ausgeführt werden. Modultests ermöglichen jedoch das Testen einzelner Module. Ist der Entwicklungsstand wiederum zu fortgeschritten und sind die Module schon fest integriert, kann unter Umständen nur in der höchsten Ebene (in der Systemebene) getestet werden. Sind die Abhängigkeiten nämlich zu groß, wären Änderungen am Code notwendig und eventuell unerwünscht. Die ersten Tests sollten in der Modultestebene erstellt und ausgeführt werden, da diese den innersten Kern der Anwendung widerspiegeln.

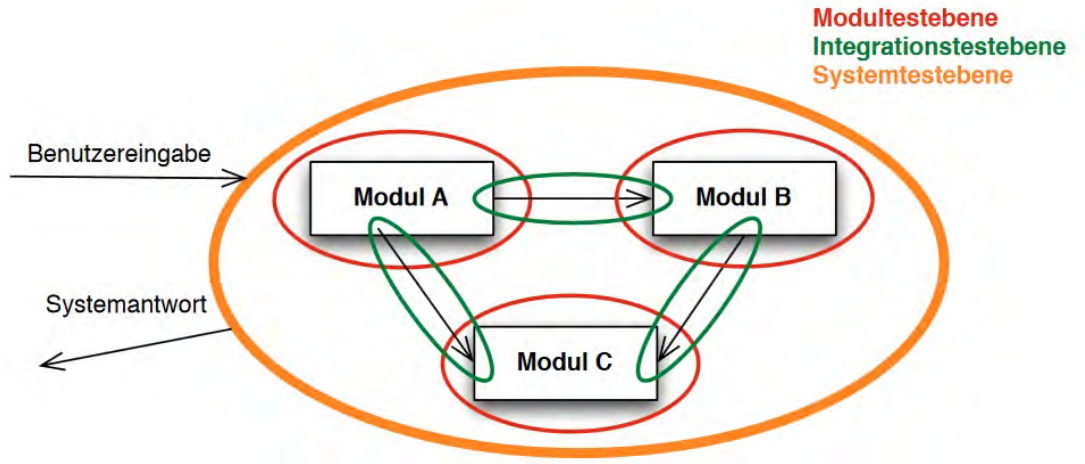

Abbildung 2.9: Darstellung der Testebenen in einem System [NK11]

2.3.1 Modultestebene

Hier ist es das Ziel, Module einzeln zu testen um sicherzustellen, dass die Systemteile (Funktionen, Methoden, Klassen, Module et cetera) korrekt funktionieren. Es ist also möglich, auf der kleinsten vorkommenden Ebene zu testen.

Unit-Tests werden üblicherweise in der White-Box erstellt, da hier die Funktionen (und deren Bezeichner) bekannt sind. Aber auch in der Gray- beziehungsweise Black-Box ist es möglich, Unit-Tests anhand der Spezifikation zu erstellen. Vor allem mit dem Test-First-Ansatz, bei dem erst die Tests entwickelt werden, ist das Erstellen der Tests anhand der Spezifikation in der Black-Box gut realisierbar. [NK11]

In der Regel wird aber nach dem Code-First-Ansatz gearbeitet. Dabei wird zuerst der Code geschrieben und anhand des Codes die Unit-Tests erstellt. Ein bekanntes Beispiel für Unit-Tests sind die JUnit[1]-Tests in Java. Diese kann man programmieren und jederzeit starten, somit realisieren die JUnit-Tests sogar eine Testautomatisierung. Die Module werden integriert und in der Integrationstestebene geprüft sobald die meisten Fehler in der Modultestebene gefunden und beseitigt sind.

2.3.2 Integrationstestebene

Die integrierten Module werden in dieser Ebene erstmalig als Komponente eines komplexen Systems getestet. Die Integrationstestebene soll das korrekte Zusammenspiel der Systemteile prüfen und sicherstellen. Dabei sind vor allem die Schnittstellentests relevant. Die Tests der einzelnen Module lassen sich hier oft übernehmen um sicherzustellen, dass die Funktionen weiterhin korrekt funktionieren.

1 Framework zum Testen von Java Programmen

Auch in dieser Ebene werden die Tests meist in der White-Box durchgeführt. Die Integrationstests sind auch mit JUnit realisierbar, sofern die Module in Java programmiert werden. [NK11] Ist die Integration (und die Testfälle) erfolgreich abgeschlossen, wird das System in der letzten Ebene getestet.

2.3.3 System- und Abnahmetestebene

Bei dem Systemtest wird das komplette Softwaresystem überprüft. Es wird anhand der Anforderungen (sowohl funktional als auch nicht-funktional) validiert, ob alle Kriterien erfüllt werden. Die Systemtests sollen realitätsnah gestaltet und simuliert werden. Um das zu gewährleisten, werden System- und Abnahmetests in der Black-Box (aber auch in der Gray-Box) durchgeführt. [NK11] Des Weiteren sind die JUnit-Tests in dieser Ebene deshalb nicht praktikabel, da ein Zugriff auf die GUI nötig ist.

Da Regressionstests in dieser Ebene angesiedelt sind, lohnt sich eine Testautomatisierung besonders. Eine Testautomatisierung lässt sich hier beispielsweise mit dem Macro Recorder/der Macro Programmierung realisieren, sofern eine geeignete Software die Ergebnisse auch überprüft.

Abnahmetests sind Systemtests sehr ähnlich, hier wird lediglich die Test- beziehungsweise Zielumgebung geändert. Statt auf der Entwicklungsumgebung, testet der Kunde (eventuell mit einem Berater/Entwickler) die Software in seiner Einsatzumgebung.

In dieser Arbeit wird in der Systemebene gearbeitet, weshalb für das Projekt eine Software zur Testautomatisierung ausgewählt wird, welche über die GUI von GeoMedia SmartClient Kommunal testet.

2.4 GeoMedia SmartClient

GeoMedia SmartClient ist ein GIS[1], welches von Hexagon (Intergraph) entwickelt wird. Es ist eine Java-basierte Anwendung, mit welcher man räumliche Daten erfassen, bearbeiten, analysieren, organisieren und präsentieren kann. Abbildung 2.10 zeigt die GUI von GeoMedia SmartClient. Dargestellt wird dort eine Karte von Las Vegas, wobei Hotels dort hervorgehoben werden. [Hex13]

GeoMedia SmartClient hebt sich von anderen GIS-Anwendungen hervor, da eine unternehmensweite Bereitstellung möglich ist. Nicht nur erfahrene GIS-Nutzer können die Anwendung nutzen.

1 Geoinformationssystem

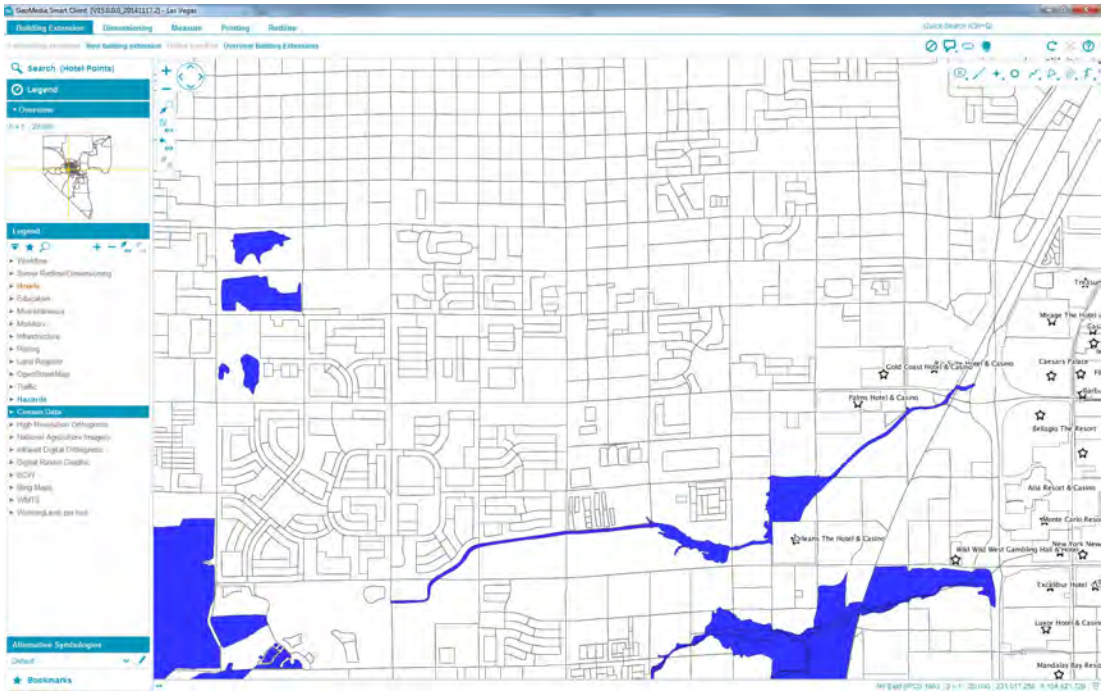

Abbildung 2.10: GeoMedia SmartClient GUI von 2014 [Hex13]

Diese Features werden auch durch den sogenannten SmartClient ermöglicht. Dieser kombiniert Webtechnologie zur Kommunikation (HTTP[1], SOAP[2], XML[3]) vollkommen separat, weshalb keine Ausführung eines Internetbrowsers nötig ist (siehe Abbildung 2.11). [Hex13] Für den Benutzer ist es eine Desktop-Anwendung, die wie bei Abbildung 2.10 dargestellt wird.

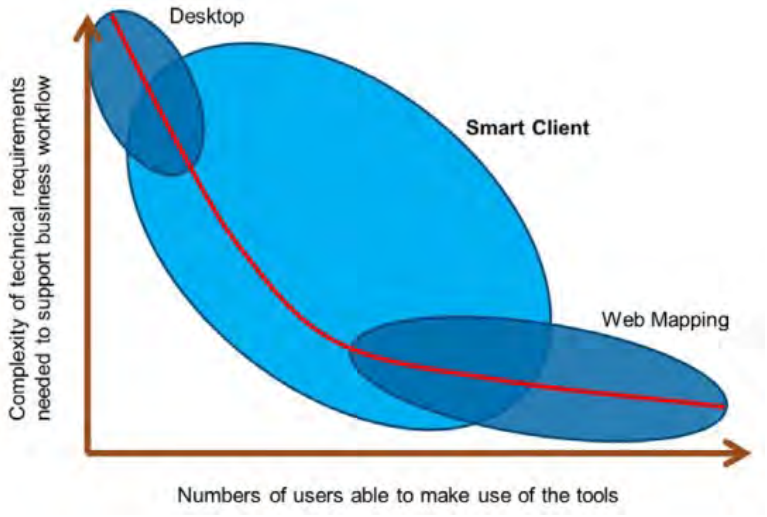

Abbildung 2.11: SmartClient verbindet Web- und Desktop-Applikation [Hex13]

1 Hypertext Transfer Protocol
2 Simple Object Access Protocol
3 Extensible Markup Language

Ursprünglich wird GeoMedia SmartClient nur für den deutschen Raum entwickelt, mittlerweile wird die Anwendung jedoch weltweit eingesetzt. KC Becker AG nutzt eine Version/Erweiterung von GeoMedia SmartClient, welches speziell für kommunale Tätigkeiten entwickelt wird. Diese nennt sich GeoMedia SmartClient Kommunal.

2.4.1 GeoMedia SmartClient Kommunal

Mithilfe von GeoMedia SmartClient Kommunal werden verschiedenste Anforderungen von Kommunen erfüllt. Die Software bewährt sich bei KC Becker AG schon jahrelang und die angebotene Kommunalberatung entlastet viele Kommunen erfolgreich. [Bec15] Dabei werden viele kommunale Aufgaben mithilfe der Anwendung umgesetzt, wie beispielsweise die Flurstückverwaltung, Doppik und viele weitere. Außerdem lässt sich die Anwendung mithilfe von Plugins erweitern, sodass es möglich ist, weitere (bisher noch nicht implementierte) kommunale Aufgabenstellungen zu lösen. [Int13]

KC Becker AG hat eigene Entwickler, welche Plugins für GMSC-K entwickeln. Somit ist es für KC Becker AG realisierbar, auch die wiederkehrenden Straßenbeiträge mithilfe der Anwendungen (und dem Plugin) für die Kommunen zu übernehmen.

Die Funktionen von GMSC-K umfassen unter anderem Bemessungen (siehe Abbildung 2.12), Ermittlung und Bearbeitung von Vektordaten (beispielsweise Flächen erstellen, löschen und Ähnliches wie bei Abbildung 2.13), Redlining und Analyse (siehe Abbildung 2.14). [Hex13]

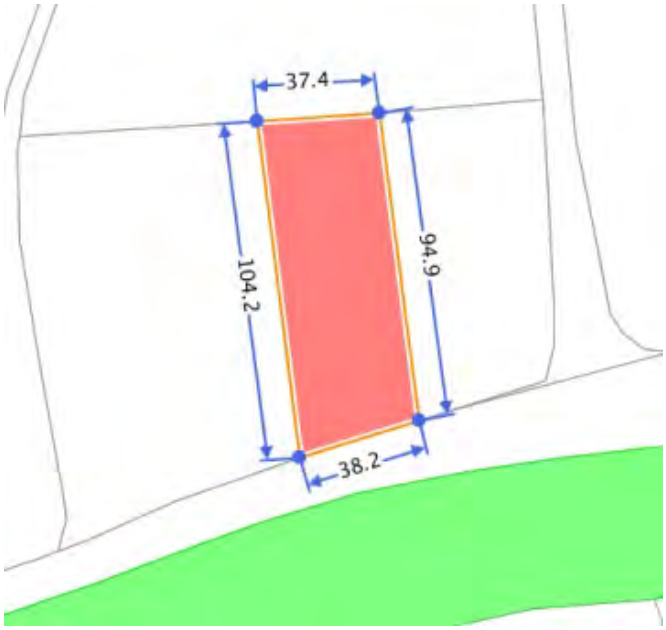

Abbildung 2.12: Bemessungen bei GeoMedia SmartClient [Hex13]

Abbildung 2.13: Flächenbearbeitung bei GeoMedia SmartClient [Hex13]

Abbildung 2.14: Analyse und Auswertung bei GeoMedia SmartClient [Hex13]

Da GMSC-K regelmäßig aktualisiert wird und ein neuer Release (Version 2017) mit PostgreSQL[1] Unterstützung entwickelt wird, ist ein Testvorgang notwendig. Dafür muss erstmal ein Konzept entwickelt werden.

1 Postgre Structured Query Language

3 Konzept

Das Konzept einer Testautomatisierung wird in diesem Kapitel thematisiert. Dabei wird erstmal auf die Anforderungen eingegangen. Danach werden Werkzeuge, welche den Anforderungen genügen könnten, vorgestellt.

Es wird vor allem auf das im Kapitel 4 genutzte Werkzeug (QF-Test) und die Funktionen eingegangen. Es werden Beispiele anhand simpler Anwendungen (beispielsweise dem Windows-Taschenrechner) dargestellt. Das Konzept der Testautomatisierung wird anschließend bei GMSC-K in Kapitel 4 angewendet.

3.1 Anforderungen

Zu Beginn müssen sowohl funktionale als auch nicht-funktionale Anforderungen definiert werden, welche bei der Realisierung eingehalten und umgesetzt werden. Dies ist bei einer Testautomatisierung wichtig, um so beispielsweise geeignete Werkzeuge zu finden. Außerdem kann auf diese Weise der Testumfang besser eingeschränkt und geplant werden. Mit geeigneten Anforderungen lassen sich also auch gewünschte Ergebnisse mit geringem Aufwand gezielt realisieren.

Die Anforderungen sind in der Regel vom Auftraggeber zu definieren, lassen sich aber mit Fachleuten noch weiter ausbauen und spezialisieren um zu garantieren, dass der Ablauf wie gewünscht stattfindet. In diesem Projekt werden die Anforderungen hauptsächlich von Hexagon mündlich festgelegt.

3.1.1 Funktionale Anforderungen

Bei den funktionalen Anforderungen werden die technischen Aspekte beschrieben. Hier wird definiert, welches Testverhalten erwünscht ist und über welche Funktionen eine Testumgebung beziehungsweise das Testwerkzeug verfügen sollte. Folgende Anforderungen werden dabei gestellt:

- **Automatisierter Testprozess:** Der Testvorgang soll automatisiert werden. Tests sollten immer wieder ausführbar sein um so möglichst sicherzustellen, dass die Funktionen bei Änderungen der Anwendung weiterhin funktionieren.

- **Testprozess findet über GUI mit Capture & Replay statt:** Das Testwerkzeug hat eine Capture & Replay-Funktion, um Testfälle schnell zu erstellen. Da alle nötigen Testfälle über die GUI erreichbar sind, sollte das Testen über die GUI möglich sein.
- **Kein extremes invasives Verhalten:** Es sollten keine Änderungen am Code erforderlich sein. Der Testprozess sollte den Code nicht verändern.
- **Tests in der Black- beziehungsweise Gray-Box:** Eine Einarbeitung in den kompletten Code würde den Umfang sprengen, weshalb die Tests in der Black- beziehungsweise Gray-Box durchgeführt werden sollen. Dadurch werden außerdem die vielen Vorteile der Black-Box umgesetzt.
- **Bildabgleich soll möglich sein:** Da bei GMSC-K mit Grafiken gearbeitet wird (Flächen, Straßen und vieles mehr), sollte es möglich sein, diese auf Richtigkeit zu überprüfen. Dies soll mit einem Bildabgleich realisiert werden. Somit wird gewährleistet, dass die Anwendung die Grafiken korrekt erstellt und anzeigt.
- **Java-kompatibel:** GMSC-K basiert auf Java und nutzt viele Java-Technologien, dementsprechend sollte das Testwerkzeug Java unterstützen.
- **Ablaufsteuerung möglich:** Der Ablauf muss steuerbar sein. Es soll beispielsweise möglich sein, eine Verzögerung einzubauen (dies ist nötig, wenn eine Funktion nicht in einer konstanten Geschwindigkeit arbeitet).
- **(automatische) Dokumentation der Testergebnisse:** Neben einer manuellen Dokumentation soll das Testwerkzeug auch automatisch die Testergebnisse (beziehungsweise Fehler) dokumentieren und präsentieren können.

3.1.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen beschreiben im Prinzip die Qualität, in der die funktionalen Anforderungen realisiert werden. Diese sind wie folgt geschildert:

- **Durchführen der Smoketests:** Mithilfe der Testautomatisierung sollen die Smoke- beziehungsweise Regressionstests abgearbeitet werden.
- **Grundbaustein für weitere Tests:** Die Testsuite soll als Grundbaustein für weitere Plugins dienen. Es sollte also erweiterbar sein.
- **Ausführbar ohne tiefe Kenntnisse:** Die Tests sollten ohne Kenntnisse der internen Struktur ausführbar sein. Auch Personen, die nicht bei der Testerstellung

beteiligt sind und keine Erfahrung mitbringen, sollten in der Lage sein, die erstellten Tests zu starten.

- **Erweiterbar ohne tiefe Programmierkenntnisse:** Es sollte möglich sein, die Testsuite ohne große Programmierkenntnisse zu erweitern beziehungsweise zu ändern. Da die Anzahl der Entwickler vor Ort bei KC Becker AG begrenzt ist, findet so eine Entlastung statt.

- **Report der Testergebnisse in klarer Struktur:** Die Testergebnisse sollen an die Entwickler kommuniziert werden. Sie sollten gut strukturiert sein, damit die Entwickler die Ergebnisse ohne Aufwand mit den betroffenen Funktionen verknüpfen können.

- **Nachvollziehbarkeit der Testergebnisse:** Die Ergebnisse müssen nachvollziehbar sein. Das wird zum Teil mit der Testautomatisierung erfüllt. Beim Dokumentieren der Testergebnisse sollte aber auch die Fehlerursache beziehungsweise die Fehlermeldung beschrieben werden.

Da in diesem Rahmen eine erstmalige Einführung der Testautomatisierung stattfindet, werden keine Anforderungen zur Laufzeit und Testabdeckung genannt. Es werden aber Werkzeuge empfohlen, welche den Anforderungen gerecht werden könnten.

3.2 Werkzeugauswahl

Bei der Auswahl des Werkzeugs sollten die Anforderungen eine zentrale Rolle spielen. Im Idealfall deckt ein Werkzeug alle Anforderungen ab. Es ist dennoch je nach Anforderung möglich, dass mehrere Werkzeuge zum Einsatz kommen müssen.

In dieser Arbeit werden Testautomatisierungstools, welche den Anforderungen genügen könnten, vom Arbeitgeber beziehungsweise Auftraggeber empfohlen. Dementsprechend müssen diese erstmal durchleuchtet werden, um so das passende Werkzeug zu ermitteln. Die genannten Werkzeuge sind QF-Test, SikuliX und Jubula.

3.2.1 SikuliX

SikuliX ist ein Open Source[1]-Projekt, welches momentan von Raimund Hocke (RaiMan) entwickelt und unterstützt wird. Angefangen hat es beim MIT[2], als die User Interface Design Group (Tsung-Hsiang Chang und Tom Yeh) sich im Jahr 2009 dazu entschloss,

1 Softwareprojekte, deren Quellcode öffentlich ist (nutz- und änderbar)
2 Massachusetts Institute of Technology

Sikuli zu entwickeln. 2012 wurde jedoch die Entwicklung eingestellt, was dazu führte, dass Raimund Hocke das Projekt übernahm und den Namen von Sikuli zu SikuliX änderte. [Hoc17]

SikuliX zählt zu den VGT[1]-Werkzeugen, weshalb es für Testautomatisierungen über die GUI geeignet ist. [FPBM16] Mit SikuliX lässt sich alles, was auf dem Bildschirm dargestellt wird, automatisieren. Dafür nutzt SikuliX eine Bilderkennungsbibliothek von OpenCV[2], um GUI-Komponenten zu identifizieren. [Hoc17]

SikuliX selbst ist eine jar[3]-Datei, weshalb eine Java-Installation vorhanden sein muss. Es ist jedoch möglich, die Testfälle in verschiedenen Programmiersprachen beziehungsweise Skriptsprachen zu erstellen [Hoc17]:

- **Python (Jython)**
- **Ruby (JRuby)**
- **Java (JavaScript)**
- **Alle Java-integrierten Programmiersprachen (Scala, Clojure und weitere)**

Eine Unterstützung auf verschiedenen Betriebssystemen wird außerdem gewährleistet (Windows, Mac, Linux/Unix). [Hoc17] Zudem ist eine Dokumentation, FAQ[4]-Sektion sowie Support bei Fragen ebenfalls vorhanden. Damit lassen sich viele Probleme und Startschwierigkeiten schnell klären.

Bei Abbildung 3.1 sieht man die SikuliX GUI, wo ein simpler Testfall mit Jython für den Windows-Taschenrechner erstellt wird. Die Anwendung wird in diesem Fall unter Windows 7 gestartet. Links in der Abbildung 3.1 sieht man die verschiedenen Aktionen, welche SikuliX anbietet.

SikuliX lässt sich neben der eigenen SikuliX-IDE[5] (wie in Abbildung 3.1) auch in anderen IDE's (Netbeans[6] , Eclipse[7], IntelliJ IDEA[8]) als Bibliothek mit den verfügbaren API[9]'s integrieren. In diesen IDE's kann man mit Java-basierten Programmiersprachen SikuliX nutzen. Jedoch wird in dieser Arbeit lediglich die SikuliX-IDE behandelt.

1 Visual GUI Testing
2 Open Source Computer Vision Library
3 Java-Archiv
4 Frequently Asked Questions
5 Integrated Development Enviroment (Integrierte Entwicklungsumgebung)
6 Ist eine freie IDE, welche mit Java geschrieben wurde
7 Open Source IDE
8 IDE für Java
9 Application Programming Interface (Programmierschnittstelle)

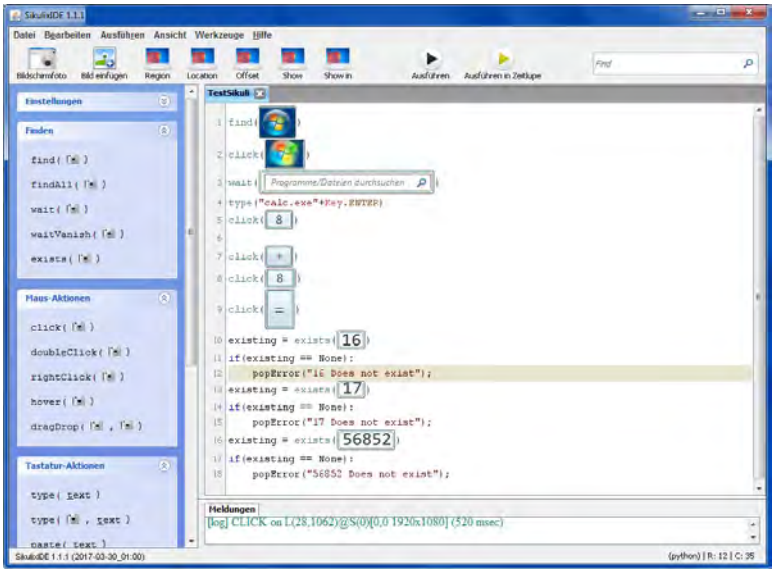

Abbildung 3.1: SikuliX GUI mit Taschenrechner-Beispiel

SikuliX unterscheidet zwischen drei Funktionen: Maus-Aktionen, Tastatur-Aktionen und eine Finden-Funktion. Diese lassen sich einfach per Mausklick in dem Skript einfügen. Ist bei einer dieser Funktionen eine Kamera abgebildet, muss ein Bild ausgewählt werden. Dafür kann man entweder ein zuvor erstelltes Bild nutzen oder man definiert ein Bildausschnitt mit SikuliX (dafür muss man einfach die Funktion anklicken und ein Rahmen um den gewünschten Ausschnitt ziehen).

Die Maus-Aktionen sind dafür da, um Komponente oder Elemente einer GUI zu betätigen. Dafür stehen folgende Maus-Aktionen zur Verfügung: Mausklick, Doppelklick, Rechtsklick, Drag & Drop und Bewegen des Mauszeigers.

Bei den Tastatur-Aktionen werden die Eingaben in Textfeldern behandelt. Folgende Tastatur-Aktionen sind dafür in SikuliX implementiert: Text eintippen und Text einfügen. Diese gibt es in zwei verschiedenen Varianten. Die erste Variante gibt den Text einfach ein ohne den Fokus zu kontrollieren. Die zweite Variante wählt den Fokus anhand einer angegebenen Abbildung und gibt dort den Text ein.

Dann gibt es noch die Funktionen, um Abbildungen zu finden/identifizieren. Mit denen lässt sich der Ablauf steuern und Kontrollstrukturen erstellen. Um das zu realisieren, gibt es in SikuliX die Funktionen: finden, alle finden, warten, warten auf Verschwinden und existiert. Alle Funktionen in dieser Kategorie arbeiten nur mit Abbildungen.

Bei SikuliX wird jede Abbildung als png[1] gespeichert. Bei größeren Projekten kann dies problematisch werden, da SikuliX viele verschiedene Bilder speichern muss, um so Zugriff auf die Komponenten zu erhalten. Der Prozess lässt sich aber optimieren (Bilder mehrfach nutzen und größere Abbildungen mit mehreren zu testenden Funktionen erstellen). Die

1 Portable Network Graphics

interne Struktur beziehungsweise die Dateistruktur für das Taschenrechner-Beispiel aus Abbildung 3.1 wird in folgender Abbildung 3.2 dargestellt:

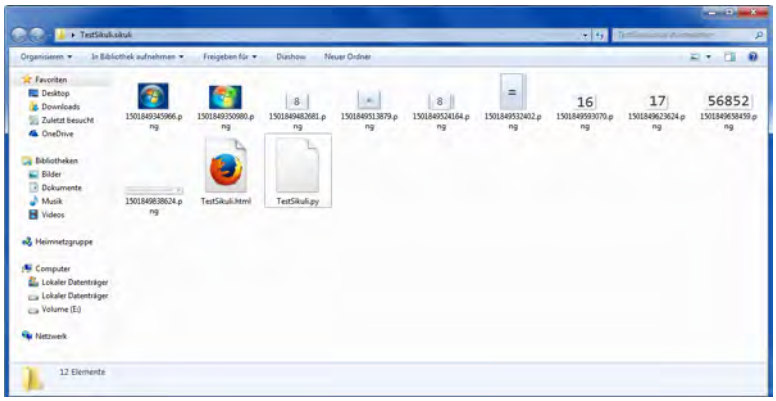

Abbildung 3.2: SikuliX-Dateistruktur vom Taschenrechner-Beispiel

Wie man der Abbildung 3.2 entnehmen kann, befinden sich neben den Bildern auch noch eine HTML[1]- und eine py[2]-Datei in dem Ordner.

Die py-Datei ist dabei das Jython-Skript und ist dementsprechend der entscheidende Faktor bei der Ausführung der Testfälle. Für das erstellte Beispiel sieht das Jython-Skript folgendermaßen aus:

Listing 3.1: Jython-Skript vom SikuliX-Taschenrechner-Beispiel

```
1 find("1501849345966.png") //Finde das Windows-Symbol
2 click("1501849350980.png") //Klicke das Windows-Symbol (Hier koennte man
      dasselbe Bild wie in Zeile 1 benutzen - 1501849345966.png)
3 wait("1501849838624.png") //Warte bis das Textfeld erscheint
4 type("calc.exe"+Key.ENTER) //Tippe calc.exe in das Textfeld und oeffne
      mit Enter dann den Taschenrechner
5 click("1501849482681.png") //Klicke auf die "8" beim Taschenrechner
6 click("1501849513879.png") //Klicke auf das "+" beim Taschenrechner
7 click("1501849524164.png") //Klicke auf die "8" beim Taschenrechner (Es
      ist moeglich dasselbe Bild wie in Zeile 5 zu benutzen -
      1501849482681.png)
8 click("1501849532402.png") //Klicke auf das "=" beim  Taschenrechner
9 existing = exists("1501849593070.png") //Schau ob die  Zahl 16 existiert
      (als Ergebnis) und speichere das Ergebnis zwischen
10 if(existing == None): //Pruefe ob die Zahl gefunden wurde
11 popError("16 Does not exist"); //Fehlermeldung falls die Zahl nicht
      gefunden wurde
12 existing = exists("1501849623624.png") //Schau ob die  Zahl 17 existiert
      (als Ergebnis) und speichere das Ergebnis zwischen
13 if(existing == None): //Pruefe ob die Zahl gefunden wurde
14 popError("17 Does not exist"); //Fehlermeldung falls die Zahl nicht
      gefunden wurde
```

1 Hypertext Markup Language
2 Python-Datei/Skript

```
15 existing = exists("1501849658459.png") //Schau ob die  Zahl 56852
       existiert (als Ergebnis) und speichere das Ergebnis zwischen
16 if(existing == None): //Pruefe ob die Zahl gefunden wurde
17 popError("56852 Does not exist"); //Fehlermeldung falls die Zahl nicht
       gefunden wurde
```

Die HTML-Datei dient lediglich für die SikuliX-IDE zur Darstellung der erstellten Schritte. Die Darstellung in einem Browser präsentiert sich wie das rechte Fenster bei Abbildung 3.1 (die Instruktionen mit den dazugehörigen Bildern).

Problematisch bei SikuliX ist jedoch die Stabilität beziehungsweise die Genauigkeit. Selbst bei simplen Testfällen wie in Abbildung 3.1 kann es schon zum Fehlverhalten aufgrund der Bildabgleiche kommen. Es kann beispielsweise vorkommen, dass anstatt des Plus-Symbols (+) das Minus-Symbol (-) angeklickt wird. Dieser Vorgang erzeugt einen Fehler, der aber nicht von der zu testenden Anwendung ausgeht, sondern ein Fehlverhalten von SikuliX widerspiegelt.

Da SikuliX jede Bilddatei abspeichern muss, ist ein wachsender Speicherverbrauch schwer vermeidbar. Besonders bei großen SUT[1] ist dies ein schwerwiegender Nachteil. Eine selbstständige Dokumentation ist ebenfalls nicht vorhanden, lässt sich aber mit den Programmiersprachen realisieren. Das ist allerdings ein Zeitaufwand, der bei anderen Testwerkzeugen eventuell entfällt.

Änderungen sind nachträglich schwer zu realisieren und die Strukturierung der Testfälle kann bei SikuliX schnell unübersichtlich werden (Ordner mit vielen Dateien für einen einzelnen simplen Testfall). Außerdem sind Programmierkenntnisse notwendig, um gute Testfälle zu erstellen.

Aus diesen Gründen ist eine Testautomatisierung mit SikuliX für GMSC-K zwar machbar, aber nicht effizient. Vor allem die fehlende Stabilität und Genauigkeit machen SikuliX in großen Projekten zu einer nicht optimalen Werkzeugwahl.

3.2.2 Jubula

Das Testautomatisierungswerkzeug namens Jubula, welches von Eclipse Foundation, Inc. kostenlos zur Verfügung gestellt wird, dient dazu, grafische Benutzeroberflächen automatisch zu testen. Dieses Tool kann Anwendungen, die mit SWT[2], Swing[3] oder RCP[4] programmiert werden, auf deren Funktionalität/Funktionsfähigkeit prüfen. Aber auch HTML-Seiten können getestet werden. Unterstützte Anwendungen im Überblick [BRE17]:

1 System Under Test
2 Standard Widget Toolkit (Java GUI-Toolkit)
3 GUI-Toolkit für Java
4 Rich Client Platform

- **Swing-Anwendungen**
- **SWT-/RCP-/GEF[1]-Anwendungen**
- **JavaFX[2]-Anwendungen**
- **HTML-Anwendungen**

Bei dem Jubula-Projekt handelt es sich um ein Open Source-Projekt. Es kann als installierbares Feature zu jeder Eclipse-Anwendung hinzugefügt werden. Eine Eclipse-Anwendung, welche Jubula bereits beinhaltet, ist das 'Eclipse for Testers'-Paket. Außerdem wird auch eine Standalone[3]-Anwendung angeboten. [BRE17]

Im Jahr 2013 wurde das kostenpflichtige GUIdancer eingestellt und als Open Source in Jubula integriert. GUIdancer erweitert/ergänzt Jubula um weitere Funktionen wie das Erstellen von Reports und Testdokumentationen, das Ausführen der Tests im Batch-Modus, Code Coverage und andere Toolkits. Ebenfalls integriert sind hilfreiche Tutorials und Dokumentationen über das Tool. [Fro11]

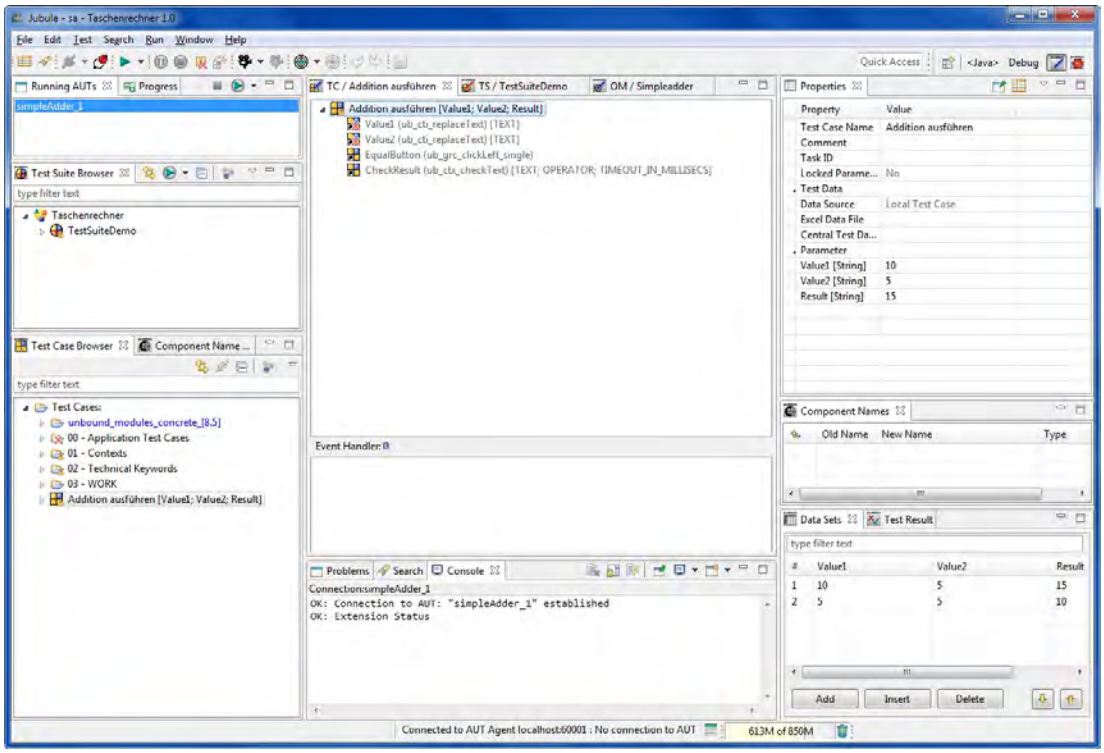

Abbildung 3.3: Jubula GUI

Da Jubula neben den Java-Technologien nur HTML unterstützt, ist es nicht möglich, den nativen Windows-Taschenrechner zu testen. Daher wird das folgende Beispiel mit

1 Graphical Editing Framework
2 Java Framework für Multimedia und GUI
3 Anwendung funktioniert eigenständig

einer Java-Anwendung ausgeführt (ein simpler Rechner mit nur einer Funktion - der Addition). In Abbildung 3.3 sehen wir die GUI von Jubula (Standalone) mit den bereits erstellten Testfällen für den Taschenrechner.

Jubula bietet viele Funktionen wie beispielsweise Anwendung starten, Keyboard Inputs, Mausklicks, Drag & Drop, Kopieren, Screenshots erstellen, Stoppuhr, Pause, Warten auf Komponenten, Auswählen und viele weitere Funktionen. Eine wichtige Funktion für die Testautomatisierung, welche Jubula bietet, sind die Checks. Diese sind in Jubula gut integriert und müssen (anders als bei SikuliX) nicht selbst mit if-Bedingungen abgefragt werden.

In Jubula ist es vergleichsweise aufwendig Testfälle zu erstellen, da die Komponenten der zu testenden Anwendung nicht selbstständig erkannt werden. Um in Jubula die Aktionen zu automatisieren, müssen erstmal die Komponenten verknüpft werden. Um das zu realisieren, ist es erforderlich, eigene Komponente (die Aktionen) in Jubula zu erstellen. Diese sollten einen eindeutigen Namen haben, um das Verknüpfen zu vereinfachen. Anschließend wird mit Jubula die sogenannte AUT[1] gestartet, um so über den Object Mapping Mode die erstellten Aktionen für den Testfall mit der Anwendung zu verknüpfen. Im Beispiel mit dem simplen Rechner sind nur vier Komponenten nötig (zwei Zahlen, ein Mausklick auf '=' und ein Check für das Ergebnis siehe Abbildung 3.4).

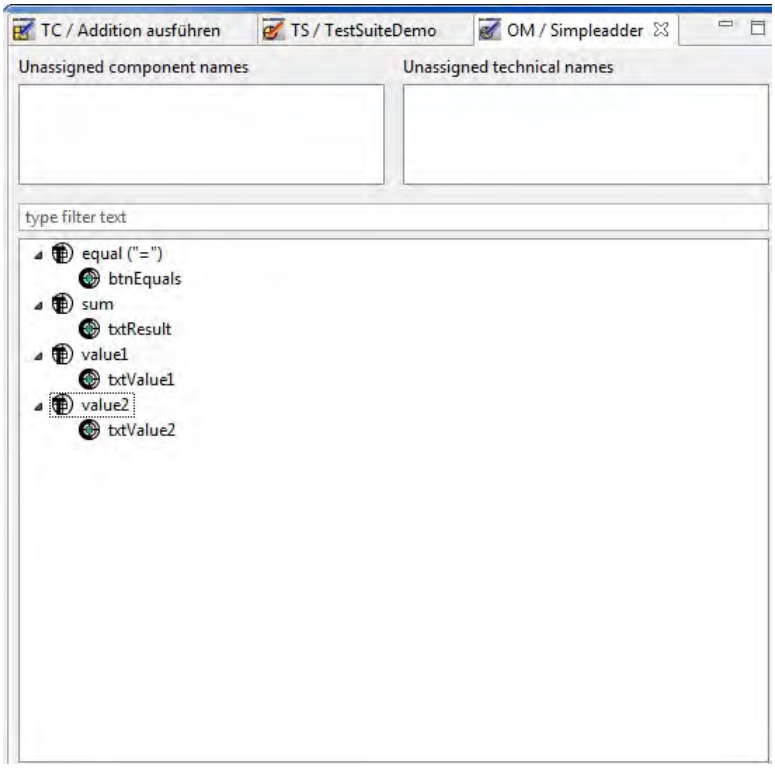

Abbildung 3.4: Mapping der Objekte in Jubula

1 Application Under Test (Anwendung unter Test)

Sind die Objekte jedoch verknüpft, ist es leicht verschiedene Testwerte für den Testfall zu nutzen. Dafür muss man lediglich Werte in den Data Sets einfügen (zu sehen in Abbildung 3.3 im unteren rechten Fenster). Diese Testwerte führt Jubula dann alle beim Testen aus.

Ein weiterer Vorteil besteht darin, dass die Testautomatisierung parallel zur Entwicklung stattfinden kann, da die Komponenten erst bei der Testausführung verknüpft werden. Sind die Anforderungen dementsprechend klar definiert, können die Testfälle in Jubula erstellt werden. [Fro11] Allerdings ist der Vorteil für GMSC-K zu vernachlässigen, da sich die Anwendung nicht in der Entwicklung befindet. Es werden lediglich weitere Releases entwickelt, die Anwendung existiert jedoch schon.

Eine Capture & Replay-Funktion bietet Jubula zudem auch nicht oder nur teilweise (für Swing- und RCP-Anwendungen) an. Außerdem werden Anwendungen, die verschiedene Technologien miteinander kombinieren, nicht vollständig unterstützt. Startet man beispielsweise eine SWT-Anwendung, welche auch Swing-Komponenten nutzt, können lediglich die SWT-Komponenten getestet werden. [BRE17]

Auch wenn Jubula eine robuste Testautomatisierung ist, scheint es aufgrund mangelnder Unterstützung der Technologien, fehlendem Capture & Replay und dem daraus folgenden zeitaufwendigen Prozess der Testerstellung ein ungeeignetes Werkzeug für das Projekt mit GMSC-K zu sein.

3.2.3 QF-Test

QF-Test ist ein Projekt von QFS[1]. Es ist ein Werkzeug zur Testautomatisierung über die GUI für Java- und Webanwendungen. QF-Test selbst besitzt eine GUI, über die man alle zur Verfügung stehenden Funktionen nutzen kann. [Fro11]

QF-Test fällt unter die Kategorie der Capture & Replay-Werkzeuge, bietet aber noch viele weitere Funktionen an. Der Fokus liegt vor allem auf Java, aber auch Webanwendungen und andere Technologien werden unterstützt wie beispielsweise [Sof17]:

- **Alle Java GUI-Technologien:** Swing, SWT, AWT[2], JavaFX, RCP, Applets und mehr
- **Web GUI's:** HTML5, AJAX[3], ExtJS[4], GWT[5] und mehr
- **Hybride Systeme:** eingebettete Browser-Komponenten mit JavaFX, Webview, JXBrowser und SWT-Browser

1 Quality First Software (Unternehmen)
2 Abstract Window Toolkit
3 Asynchronous JavaScript and XML
4 Extended JavaScript
5 Google Web Toolkit

Anders als bei Jubula und SikuliX ist QF-Test nicht Open Source. Es werden Lizenzen zur Verfügung gestellt, welche sich je nach genutzten Technologien im Preis unterscheiden. Bei Abbildung 3.5 sieht man die genaue Gestaltung der Lizenzpreise (Stand 08.08.2017). [Sof17]

Produkt	Basispreis*
Entwicklerlizenz (Testerstellung und -ausführung): **Eine** GUI-Technologie enthalten.	**1.995 €**
GUI-Technologie-Erweiterung für Entwicklerlizenz: Gleich mitbestellen oder später bei Bedarf ergänzen.	**500 €**
Runtimelizenz (Testausführung): Für Lasttests beträchtliche Mengenrabatte. **Eine** GUI-Technologie enthalten.	**995 €**
GUI-Technologie-Erweiterung für Runtimelizenz: Gleich mitbestellen oder später bei Bedarf ergänzen.	**250 €**
Pflegevertrag (verbindlich bei Neukauf und für 1 Jahr): alle Upgrades + Support	**ab 480 €**

Abbildung 3.5: Lizenzkosten für QF-Test (Stand 08.08.2017) [Sof17]

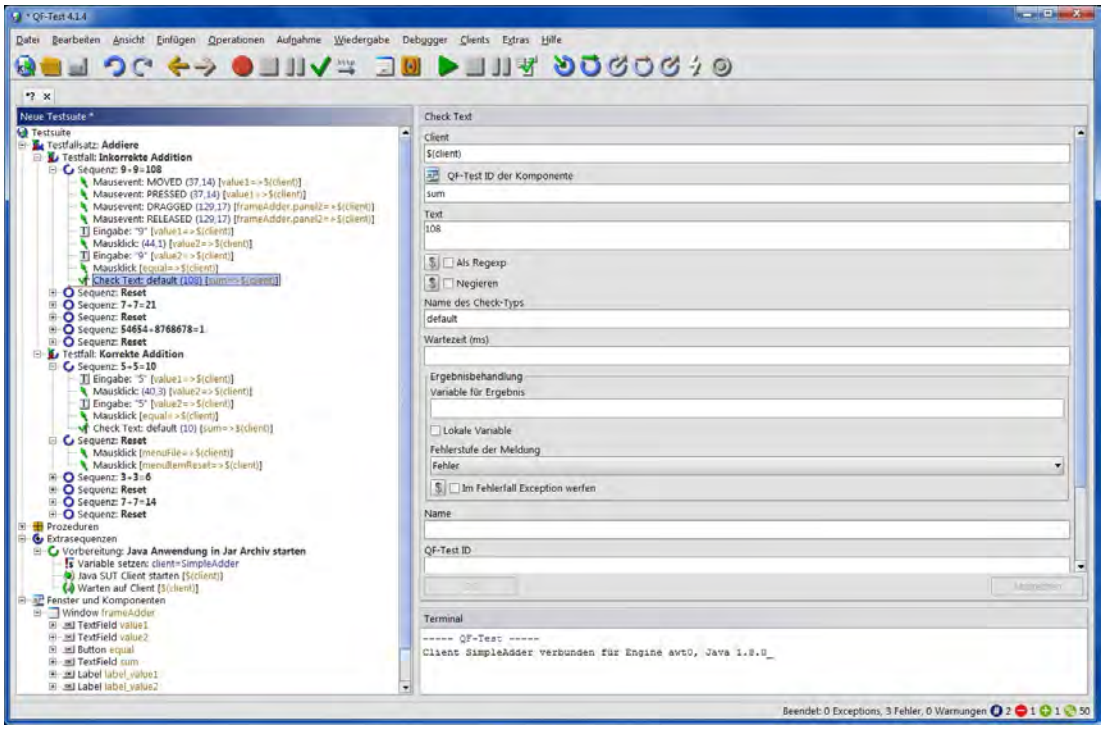

Abbildung 3.6: GUI von QF-Test

Des Weiteren hat QF-Test Zugriff auf die Komponenten, weshalb eine Verknüpfung (die bei Jubula nötig ist) wegfällt. Dies hat den Vorteil, dass sich die Testfälle auch bei Veränderung der Anwendung robust verhalten sollten. [Heg14]

QF-Test ist sehr gut dokumentiert und der Support ist schnell und hilfreich. Daher werden in dieser Arbeit nicht alle Funktionen ausführlich beschrieben. Vor allem die genutzten Funktionen werden thematisiert.

In Abbildung 3.6 sieht man QF-Test mit Testfällen, welche mit der Capture & Replay-Funktion aufgenommen wird. Die in QF-Test sogenannte SUT ist bei der Abbildung ein Java-Taschenrechner mit einer Funktion (Addition). Im linken Fenster bei der Abbildung 3.6 sieht man die Testfälle, Sequenzen und weitere Knoten in einer Baumstruktur. Die Funktionen haben verschiedene Symbole, um so die Erkennung und damit die Entwicklung zu erleichtern. In bestimmten Hierarchieebenen stehen manche Knoten nicht zur Verfügung (beispielsweise kann man kein Testfallsatz-Knoten in einer Sequenz erstellen). Folgende Bestandteile sind bei der Struktur einer Testsuite in QF-Test enthalten [Sof17]:

- **Testsuite:** Die Testsuite ist der Wurzelknoten des Baums. Er enthält eine beliebige Anzahl von Tests, Prozeduren, Extrasequenzen und dem Fenster- und Komponenten-Knoten. Die Struktur der Testsuite ist definiert und die Testsuite selbst kann nicht in einem Knoten enthalten sein.

Abbildung 3.7: Wurzelknoten

- **Test- und Sequenz-Knoten:** Mit diesen Knoten lassen sich die Tests strukturieren. Sie sind somit die wichtigsten Strukturelemente einer Testsuite. Folgende Knoten fallen unter diese Kategorie: Testfall, Testfallsatz, Testaufruf, Test, Sequenz, Testschritt, Sequenz mit Zeitlimit und die Extrasequenz. In dieser Arbeit wird vor allem mit Testfällen, Testfallsätzen und Sequenzen gearbeitet. Testfälle sind die wichtigsten Knoten, denn diese dienen dazu, die eigentlichen Testfälle zu realisieren. Die Sequenzen erfüllen dabei meist die Logik (die Ausführung mit Maus- und Tastaturaktionen) und mit den Testfallsätzen wird der Prozess noch weiter strukturiert.

Abbildung 3.8: Test- und Sequenz-Knoten

- **Abhängigkeit:** Diese Knoten sind im Prinzip zur Vorbereitung und Nachbereitung der Testfälle relevant. Das ist wichtig, um Testfälle möglichst zu isolieren. Dadurch wird ermöglicht, einzelne (eventuelle fehlerhafte) Testfälle ausführen zu können ohne den ganzen Testfallsatz durchlaufen zu müssen. Diese Kategorie umfasst folgende Knoten: Abhängigkeit, Bezug auf Abhängigkeit, Vorbereitung, Aufräumen und Fehlerbehandlung.

Abbildung 3.9: Abhängigkeiten-Knoten

- **Datentreiber:** Mit den Datentreibern können Testfälle mit verschiedenen Datensätzen ausgeführt werden. In QF-Test gibt es für die Verwendung der Daten keine Einschränkungen. Meistens werden sie jedoch für die erwarteten Ergebnisse in den Check-Knoten und für die Eingabewerte in Events genutzt. Folgende Knoten gelten als Datentreiber in QF-Test: Datentreiber, Excel-Datei, CSV-Datei (steht für Comma-separated values) und die Datenschleife.

Abbildung 3.10: Datentreiber-Knoten

- **Prozeduren:** Prozeduren erfüllen den Zweck, ähnliche Abläufe auszulagern. Somit bleibt die Struktur überschaubar und die Komplexität der Testsuite wird reduziert. Prozeduren lassen sich von jedem Punkt aus aufrufen. Sogar andere Testsuiten können die erstellten Prozeduren aufrufen. Ein weiterer Vorteil ist, dass eventuelle Veränderungen nicht an vielen verschiedenen Stellen durchgeführt werden müssen, sondern nur in der Prozedur selbst erfolgen müssen. Es ist außerdem möglich, Parameter beim Aufruf einer Prozedur zu übergeben. Unter diese Kategorie fallen die Knoten: Prozedur, Prozeduraufruf, Return, Package und Prozeduren.

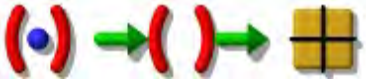

Abbildung 3.11: Prozedur-Knoten

- **Ablaufsteuerung:** Mit Kontrollstrukturen lässt sich der Ablauf des Testvorgangs steuern. Die Kontrollstrukturen von QF-Test orientieren sich an Programmiersprachen wie Java. Somit werden bedingte Ausführungen und Schleifen ermöglicht, die wie bei Java Exceptions werfen können. Die Bezeichnungen der Knoten sind den Entwicklern daher in der Regel bekannt. Die Knoten heißen wie folgt: Schleife, While, Break, If, Elseif, Else, Try, Catch, Finally, Throw, Rethrow, Server-Skript und SUT-Skript. Mit den Server- beziehungsweise SUT-Skripts lässt sich der Ablauf mit den Programmiersprachen Jython oder Groovy steuern.

Abbildung 3.12: Ablaufsteuerung-Knoten

- **Prozesse:** Knoten, die in diese Kategorie fallen, dienen der Kommunikation zwischen der SUT und QF-Test. Diese Knoten sind für eine Testsuite fundamental, da die SUT über QF-Test gestartet werden muss. Sonst besteht keine Verbindung zur Anwendung und die Testfälle können nicht erstellt beziehungsweise ausgeführt werden. Unter Prozesse fallen die Knoten: Java SUT Client starten, SUT Client starten, Browser starten, Programm starten, Shellkommando ausführen, Warten auf Client, Programm beenden und Warten auf Programmende.

Abbildung 3.13: Prozess-Knoten

- **Events:** Bei den Event-Knoten handelt es sich um Knoten, welche Aktionen in der SUT auslösen. Diese sind bei Testfällen beziehungsweise Sequenzen elementar, da mit den Events die Ausführung realisiert wird. Folgende Events stehen zur Verfügung: Mausevent (Mausbewegungen, Mausklicks, Drag & Drop-Operationen), Tastaturevent, Texteingabe, Fensterevent, Komponentenevent, Auswahl und die Dateiauswahl. Nutzt man die Capture & Replay-Funktion von QF-Test, werden diese Events automatisch genutzt und in einer Sequenz abgespeichert.

Abbildung 3.14: Event-Knoten

- **Checks:** Die Checks ermöglichen die Abfrage von Werten. Mit den Checks wird die Testautomatisierung vervollständigt, denn diese übernehmen das manuelle Vergleichen der Werte beziehungsweise Daten. Stimmen die Werte, welche die SUT liefert, nicht mit den Checks überein, wird ein Fehler verzeichnet. Es ist jedoch auch möglich, Exceptions zu werfen und die Ergebnisse einer Variable zuzuweisen. Folgende Checks werden von QF-Test geliefert: Check Text, Check Boolean, Check Elemente, Check selektierbare Elemente, Check Abbild und Check Geometrie.

Abbildung 3.15: Check-Knoten

- **Abfragen:** Verhält sich die SUT dynamisch, sind die Abfragen von QF-Test wichtig. Mit diesen Knoten können Daten zur Laufzeit aus der GUI der SUT ausgelesen und in der Testsuite verwendet werden. Dafür nutzt QF-Test folgende Knoten: Text auslesen, Index auslesen und Geometrie auslesen.

Abbildung 3.16: Abfrage-Knoten

- **Weitere Knoten:** Die folgenden Knoten von QF-Test fallen unter keine bestimmte Kategorie, sind jedoch auch wichtig: Variable setzen, Warten auf Komponente, Warten auf Laden des Dokuments, Warten auf Ende des Downloads, Ressourcen laden und Properties laden.

Abbildung 3.17: Weitere Knoten

- **Fenster, Komponenten und Elemente:** Diese Knoten sind das Fundament einer Testsuite. Sie spiegeln die SUT wider. Da Events und Checks auf die SUT zugreifen, wird mit diesen Knoten die Verknüpfung realisiert. So soll sichergestellt werden, dass die richtigen Fenster, Komponenten und Elemente angesprochen werden. Dafür sieht QF-Test diese Knoten vor: Fenster, Webseite, Komponente, Element, Fenstergruppe, Komponentengruppe sowie Fenster und Komponenten.

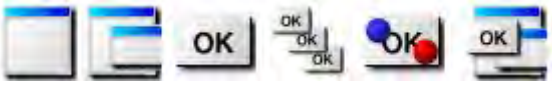

Abbildung 3.18: Fenster-, Komponenten- und Element-Knoten

Alle Knoten lassen sich in der Regel anpassen, um so ideal den Bedürfnissen der Testsuite zu entsprechen. Beispielsweise lassen sich Checks negieren[1], wenn nur ein bestimmter Wert unzulässig sein soll.

In Abbildung 3.6 sieht man den ‘Check Text: default (108)’. Dieser überprüft ob 9+9=108 ergibt und wirft in diesem Zustand einen Fehler, falls das Ergebnis nicht 108 beträgt. Die Fehlermeldung von QF-Test kann man in Abbildung 3.19 sehen. Negiert man diesen Check, indem man ein Häkchen bei ‘Negieren’ setzt, wird kein Fehler geworfen, da so alle Werte zulässig sind außer 108.

Aus dem Protokoll (Abbildung 3.19) heraus kann man ein Report erstellen. Dafür hat man verschiedene Optionen zur Auswahl. Es kann beispielsweise festgelegt werden, welche Elemente (Testschritte, Checks, Exceptions und Weitere) aufgelistet werden sollen (siehe Abbildung 3.20). Es ist außerdem möglich, verschiedene Reports zu erstellen (HTML, XML und JUnit). Der Report wird lokal gespeichert und kann dann weitergeleitet oder online gestellt werden. Der Ordner, welcher beim HTML-Report erstellt wird,

1 Wahrheitswert wird in sein Gegenteil gewandelt

umfasst CSS[1]-Dateien, HTML-Dateien, Icons und Images.

Den HTML-Report kann man im Browser aufrufen. Der Report für dieses Beispiel wird in Abbildung 3.21 dargestellt. Dort werden die Fehler strukturiert präsentiert. So kann der Tester nochmals überprüfen, ob der Testfall richtig eingerichtet ist. Wird tatsächlich ein Fehler in der SUT gefunden, kann man den/die Fehler dokumentieren oder den Report weiterleiten.

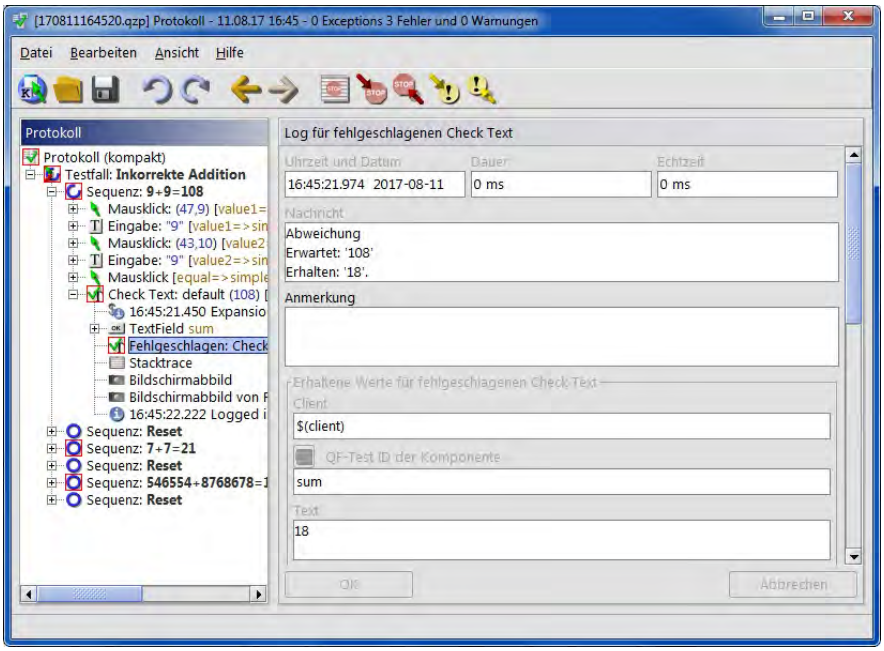

Abbildung 3.19: Protokoll von QF-Test für gefundene Fehler

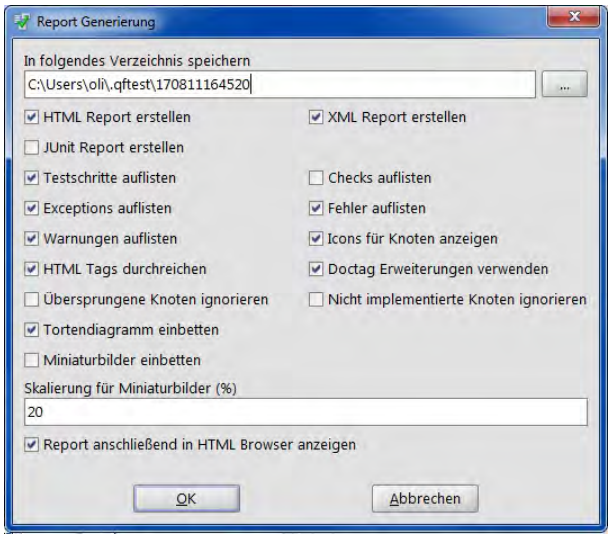

Abbildung 3.20: Generierung eines Reports in QF-Test

1 Cascading Style Sheets

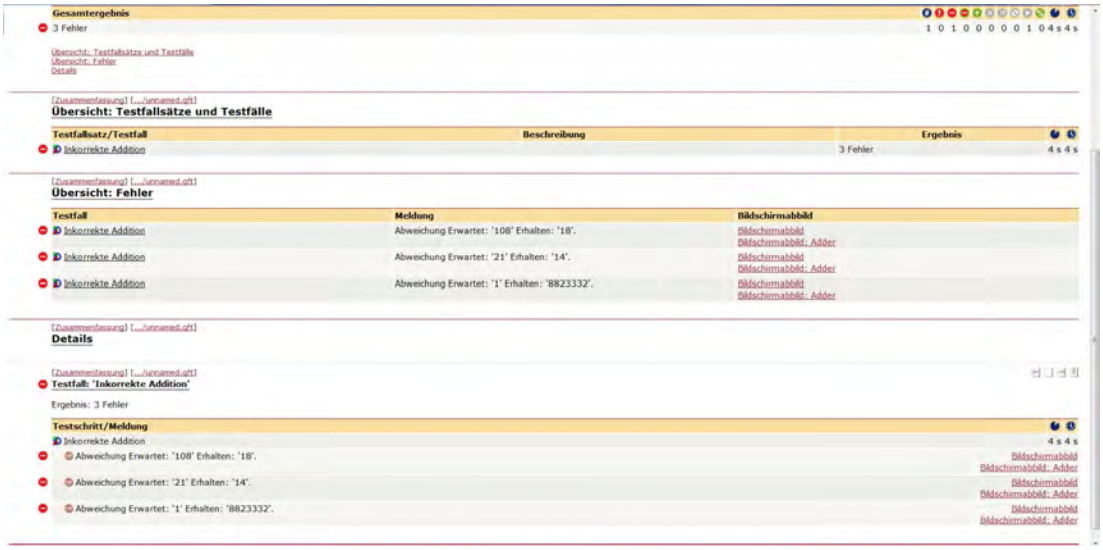

Abbildung 3.21: HTML-Report von QF-Test

Für weitere Details der Bestandteile und Features eignet sich die Dokumentation von QF-Test. Diese umfasst mehr als 1 100 Seiten, weshalb in dieser Arbeit QF-Test nicht detailliert beschrieben werden kann. [Sof17]

3.2.4 Weitere Werkzeuge zur Testautomatisierung

Neben den genannten Werkzeugen vom Auftraggeber gibt es noch weitere Alternativen. Diese werden im Rahmen der Arbeit jedoch nicht intensiv behandelt und daher nur namentlich erwähnt:

- **Selenium:** Testautomatisierungstool für Webanwendungen (Open Source). [Sel17]
- **Automated GUI Recorder:** Automatische GUI-Steuerung von TPTP[1] in der Entwicklungsumgebung Eclipse (Eclipse-Plugin). [Fo17]
- **Silktest:** Testautomatisierungstool von Micro Focus für Web-, Mobil- und Desktop-Anwendungen. [Mic17]
- **TestComplete:** Werkzeug zur Testautomatisierung, welches von SmartBear Software entwickelt wird. Eignet sich für Web-, Mobil- und Desktop-Anwendungen. [Sma17]
- **HP[2] Unified Functional Testing:** Dieses Werkzeug von HP ermöglicht die Testautomatisierung von Web-, Desktop- und Mobilanwendungen. [Pac17]

1 Test and Performance Tools Platform
2 Hewlett-Packard (Unternehmen)

- **Squish:** Squish ist ein Testautomatisierungswerkzeug über die GUI. Es werden viele Plattformen (sowie Cross-Plattform) unterstützt. Es eignet sich für Desktop-, Mobil- und Webanwendungen. Auch eingebettete Systeme lassen sich mit Squish testen. [Fro17]

Die Liste ist nur ein Auszug von Werkzeugen zur Testautomatisierung und ist dementsprechend nicht vollständig.

3.3 Testprozess nach ISTQB

Dieser weltweit anerkannte Testprozess (entwickelt vom ISTQB[1]) findet beim systematischen Testen oft Anwendung. Es gibt auch Weiterbildungen nach ISTQB, die das Erwerben eines Zertifikats ermöglichen. Diese sind im Arbeitsmarkt sehr angesehen. Abbildung 3.22 bildet den Testprozess ab.

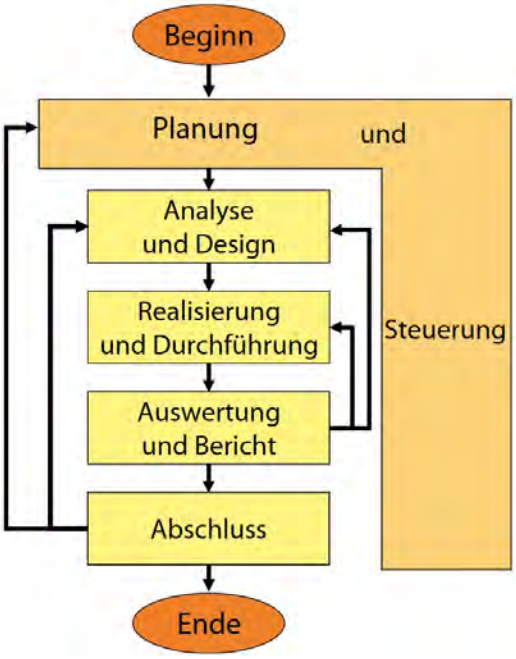

Abbildung 3.22: Testprozess nach ISTQB [Spi08]

Die einzelnen Prozesse lassen sich wie folgt beschreiben [Spi08]:

- **Planung und Steuerung:** In diesem Prozess wird die Teststrategie festgelegt. Es wird geplant, welche Teile des Systems mit welcher Intensität getestet werden sollen. Auch das Testverfahren wird in diesem Prozess bestimmt. Die Einhaltung der Planung soll dann über den gesamten Testprozess gewährleistet werden. Dafür

1 International Software Testing Qualifications Board

kann auch steuernd eingegriffen werden, um Korrekturen durchzuführen. Hier wird also der ganze Testprozess definiert.

- **Analyse und Design:** In diesem Bereich werden die Anforderungen anhand der Spezifikationen und ähnlicher Dokumente analysiert. Mit diesen Informationen werden Testziele und Testkriterien bestimmt. Die Testumgebung wird eingerichtet, um so erste Testfälle zu entwerfen und zu realisieren. Auf diese Weise wird sichergestellt, dass das Testen in der Testumgebung realisierbar ist.

- **Realisierung und Durchführung:** Hier werden die konkreten Testfälle realisiert. Die Testfälle können dann entweder manuell oder automatisch ausgeführt werden. Wichtig ist, dass die Testdaten bereitstehen und die Ergebnisse ausgewertet werden können (Soll- und Ist-Ergebnis). Außerdem werden die Testfälle in diesem Prozess strukturiert (mit Testsuiten, Testfallsätze, Testfälle und Ähnlichem).

- **Auswertung und Bericht:** In diesem Prozess werden die Ergebnisse analysiert und dokumentiert. Insbesondere gefundene Fehler werden hervorgehoben und in der ausgewählten Dokumentation beschrieben. Auch weitere Besonderheiten und Informationen, wie beispielsweise welche Testfälle sich nicht automatisieren lassen, gefundene Unstimmigkeiten zur Spezifikation oder Ähnliches, können erläutert werden. Eventuell müssen auch noch weitere Testfälle erstellt und ausgeführt werden, um die gewünschten Testergebnisse zu erreichen.

- **Abschluss:** Zum Abschluss werden die Tests zusammen mit den Testergebnissen bewertet. Es wird ermittelt, ob ausreichend getestet wird. Falls dies nicht der Fall ist, müssen weitere Tests spezifiziert und erstellt werden. Die Dokumentationen und Testspezifikationen werden außerdem gespeichert, um zukünftige Regressionstests zu beschleunigen. Auch Fehler und Schwachstellen sollten dokumentiert werden, um diese beim nächsten Testprozess zu verbessern oder zu umgehen.

Die Prozesse sind nicht zwingend nacheinander auszuführen. Das lässt sich auch zeitlich oft nicht vereinbaren, weshalb sich eine Überlappung nicht vermeiden lässt. [Spi08]

Beim Projekt mit GMSC-K wird größtenteils mit dem Testprozess nach ISTQB gearbeitet, jedoch wird die Reihenfolge nicht strikt eingehalten, da manche Prozesse auch nicht mehr nötig sind.

4 Realisierung

In diesem Kapitel wird das Konzept an einem konkreten Projekt (Testautomatisierung mit QF-Test mit der SUT GMSC-K) umgesetzt. Die Wahl des Werkzeugs ist dementsprechend QF-Test, da es den Anforderungen genügt und sogar weitere positive Aspekte (wie Bedienbarkeit, intuitive Oberfläche und Funktionen) mit einbringt.

Die Evaluation in der Master-Thesis „Untersuchungen zu automatisierbaren Testszenarien für User Interfaces von kaufmännischen Softwaresystemen“ von Christian Froh zeigt außerdem, wie sich QF-Test gegen andere Werkzeuge zur Testautomatisierung durchsetzen kann. [Fro11]

Um die Realisierung zu strukturieren, wird der im Konzept vorgestellte Testprozess nach ISTQB genutzt. Der Abschluss wird nicht in diesem Kapitel behandelt, da dieser Prozess der Evaluierung (Kapitel 5) gleicht.

4.1 Planung und Steuerung

Die Teststrategien werden von den Anforderungen vorgegeben, dementsprechend wird in der Black-Box und in der Gray-Box getestet. Der Fokus ist aber auf die Black-Box gerichtet, in der Gray-Box soll nur das Fundament geschaffen werden, um das Testen in der Black-Box zu ermöglichen und effektiv zu gestalten.

Es soll das ganze System mit den vollen Grundfunktionen (ohne Plugins und Ähnlichem) getestet werden. Somit lässt sich dieser Vorgang als Regressionstest beziehungsweise Smoketest in der Systemebene einstufen.

Smoketests unterscheiden sich zu den Regressionstests ein wenig, denn hier werden die Funktionen nicht so intensiv getestet wie bei den Regressionstests. Der Fokus bei den Smoketests liegt in der Funktionalität. Es soll den Erhalt der Funktionen prüfen um so sicherzustellen, dass die getestete Version ohne Ausfall in einem Betrieb eingesetzt werden kann. Falsche Eingaben werden nur sporadisch getestet. Dadurch wird die Testsuite nicht zu sehr aufgebläht und kann so schneller Ergebnisse liefern, während ein intensiver Regressionstest mit dementsprechend vielen Testfällen Stunden oder sogar Tage benötigen kann.

4.2 Analyse und Design

Da ältere Versionen von GMSC-K schon manuell getestet werden, gibt es Dokumente mit Testfällen, welche die zu testenden Funktionen abdecken. Daher müssen in dieser Arbeit nicht viele Testfälle selbst spezifiziert werden.

Zusammengefasst sind über 3 000 Testfälle vorhanden, aufgrund von Schwierigkeiten beim Export aus einem Testprogramm (wo die Testfälle von Hexagon dokumentiert werden) sind aber manche Testfälle doppelt aufgelistet. Das Dokument ist eine .xls[1]-Datei mit 3 365 Zeilen. Ein Ausschnitt des Dokumentes kann man Abbildung 4.1 entnehmen.

	A	B	C	D	E	F	G	H
1	Test ID	Testname	Testbeschreibung	Ergebnis	Anzahl Schritte	Teststschritt Name	Testschritt Beschreibung	Testschritt Vorbedingung
2	30	Datenquelle anlegen		Adminer startseite wird angezeigt	6	Step 1	PolyGis Adminer starten	
3	30	Datenquelle anlegen		Das Kontextmenü erscheint	6	Step 2	Im Adminer mit rechter Maustaste auf Datenbanken Klicken	
4	30	Datenquelle anlegen		Die Maske zum Anlegen einer neue Datenquelle erscheint	6	Step 3	Im Kontextmenü zu Neu Navigieren, dann kommt ein weiteres Kontextmenü und jetzt auf neu Datensatz Klicken	
5	30	Datenquelle anlegen			6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen. Bezeichnung eingeben Datenbankserver: Name oder IP-Adresse des Servers eingeben Port auf den der Server hört eingeben Nutzername: Name des Datenbankbenutzer angeben Kenntwort:Kenntwort des Datenbankbenutzer eingeben	
6	30	Datenquelle anlegen			6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen. Bezeichnung eingeben Datenbankserver: Name oder IP-Adresse des Servers eingeben Port auf den der Server hört eingeben Nutzername: Name des Datenbankbenutzer angeben Kenntwort:Kenntwort des Datenbankbenutzer eingeben	
7	30	Datenquelle anlegen		Daten sind eingegeben	6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen.Bezeichnung eingebenDatenbankserver: Name oder IP-Adresse des Servers eingeben DB NamePort auf den der Server hört eingebenNutzername: Name des Datenbankbenutzer angebenKenntwort:Kenntwort des Datenbankbenutzer eingeben	
8	30	Datenquelle anlegen		Daten sind eingegeben	6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen. Bezeichnung eingeben Datenbankserver: Name oder IP-Adresse des Servers eingeben Port auf den der Server hört eingeben Nutzername: Name des Datenbankbenutzer angeben Kenntwort:Kenntwort des Datenbankbenutzer eingeben	
9	30	Datenquelle anlegen		Pop-up-Fenster mit der Meldung "Verbindung erfolgreich aufgebaut" erscheint	6	Step 5	Oben in der Eingabemaske auf Datenquelle prüfen klicken und im Pop-up-Fenster "OK" klicken	
	30	Datenquelle anlegen		unten links in der Statusleiste erscheint die	6	Step 6	Oben in der Eingabemaske auf Speichern klicken	

Query1

Abbildung 4.1: Dokument mit den Testfällen

Anhand der Abbildung 4.1 kann man die Struktur der Testspezifikation auslesen. Diese sieht wie folgt aus:

- **Test ID** ist in der Spalte A und ist intern (bei Hexagon) für die Identifikation der Testfälle verantwortlich. Die Test ID ist für die Realisierung der Testfälle nicht relevant.
- **Testname** gibt den Testfall einen Namen. Damit lassen sich Testfälle leichter zuordnen. Diese befindet sich in Spalte B im Dokument und ist für die Realisierung der Testfälle relevant, da die Namen der Testfälle in QF-Test identisch sein sollten.

1 Microsoft Excel-Datei

- **Testbeschreibung** befindet sich in Spalte C und wird nur genutzt, wenn der Testfall weiter spezifiziert werden muss.

- **Ergebnis** spiegelt das Soll-Ergebnis wider. Mit dieser Spalte (D) werden die Checks realisiert. Wird ein anderes Ergebnis (Ist-Ergebnis) als das in Spalte D dokumentierte Ergebnis (Soll-Ergebnis) ausgegeben, ist eventuell eine Klärung nötig. Da mit diesen Testfällen auch ältere Releases getestet werden, könnte sich das Soll-Ergebnis geändert haben. Daher muss eine Abweichung nicht zwingend ein Fehler sein.

- **Anzahl Schritte** in Spalte E sagt aus, wie viele Testschritte der Testfall besitzt. Jeder Testschritt ist mindestens eine Sequenz in QF-Test.

- **Testschritt Name** beschreibt im Prinzip, in welchen Testschritt man sich befindet. Führt man beispielsweise zwei Testschritte aus, würde der nächste Testschritt den Namen 'Step[1] 3' besitzen. Diese befinden sich in Spalte F und sind wichtig um beispielsweise zu überprüfen, ob keine Testschritte durch den Export verloren gegangen sind.

- **Testschritt Beschreibung** ist für die Realisierung fundamental, da mit dieser Testbeschreibung die Ausführung des Testschrittes erläutert wird. So weiß der Tester was zu tun ist und kann die beschriebene Aktion genau ausführen. Die Beschreibung befindet sich in Spalte G.

- **Testschritt Vorbedingung** ist in Spalte H und gibt Auskunft darüber, welche Vorbedingungen getroffen werden müssen, um den Testschritt ordnungsgemäß ausführen zu können. Das ist wichtig, um so das gewünschte Ergebnis liefern zu können. Manche Testschritte lassen sich außerdem ohne die Vorbedingung nicht ausführen. Die vorherigen Testschritte in einem Testfall gelten als Vorbedingung. Diese werden in dieser Spalte jedoch nicht speziell beschrieben, da sie sich aus Spalte E und F ableiten lassen.

Da die Testfälle somit spezifiziert sind, muss die Entwicklungsumgebung eingerichtet werden. Dafür muss auf dem Computer im Wesentlichen nur Microsoft Office mit Excel, Java und QF-Test installiert sein. Außerdem muss der Rechner Zugriff auf die Dienste von GMSC-K und eine Datenbank haben. Die entsprechenden Lizenzen müssen auch zur Verfügung stehen.

Um nun die ersten Testfälle auf der Entwicklungsumgebung zu realisieren, muss die SUT erstmal in QF-Test eingebunden werden. Dafür kann man den Schnellstart-Assistenten von QF-Test nutzen. Hier wählt man aus um welchen Typ es sich bei der

1 Schritt

Anwendung handelt. Bei GMSC-K handelt es sich um eine Java WebStart-Anwendung. Beim Schnellstart-Assistenten stehen noch mehr Typen zur Auswahl, wie beispielsweise ein Java-Archiv oder eine Java-Anwendung. Die verfügbaren Typen kann man der Abbildung 4.2 entnehmen.

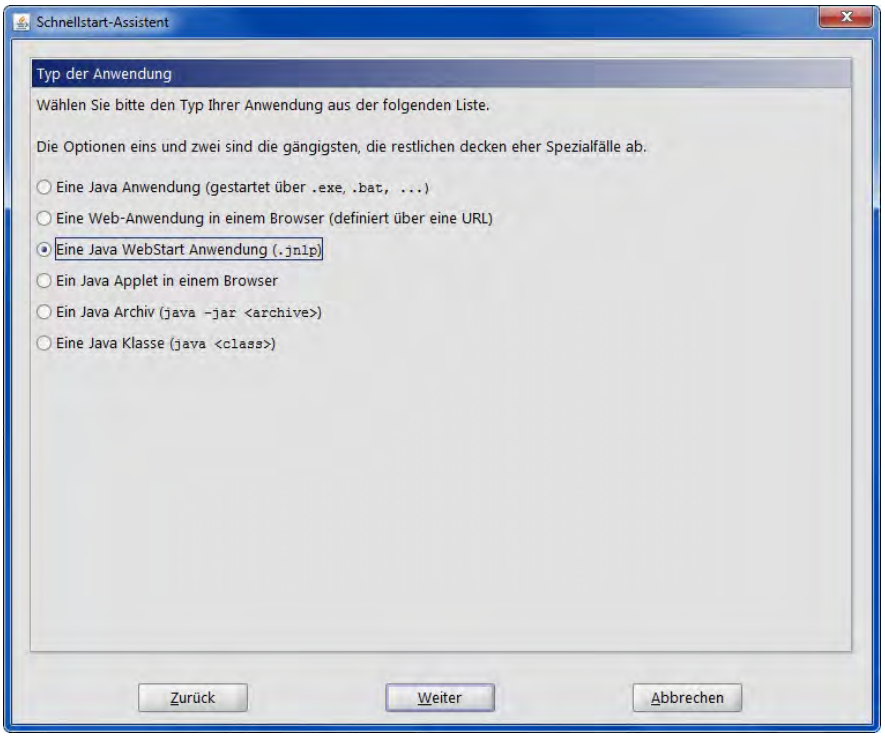

Abbildung 4.2: Auswahl des Typs der Anwendung beim Einbinden in QF-Test

Als Nächstes wird man aufgefordert den Programmpfad zum Java WebStart anzugeben (siehe Abbildung 4.3). Der Standardwert 'javaws' funktioniert bei einer typischen Installation meistens.

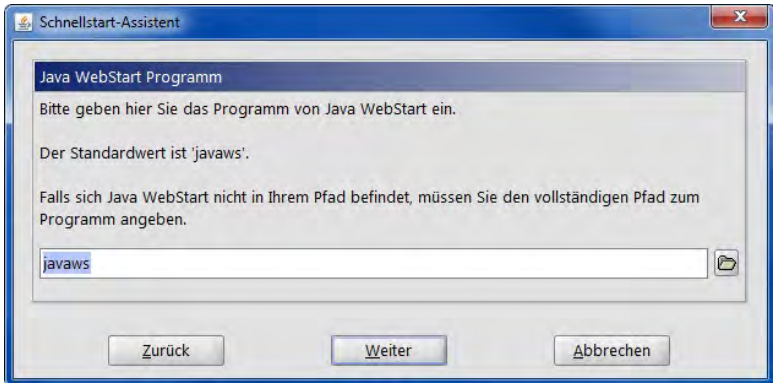

Abbildung 4.3: Angabe zum Java WebStart-Pfad in QF-Test

Dann muss die JNLP[1] angegeben werden. Je nach Projekt/Stadtteil in GMSC-K unterscheidet sich hier der Wert, auch weil diese teilweise auf verschiedenen Servern liegen. Angegeben wird diese dann im HTTP-Format (siehe Abbildung 4.4).

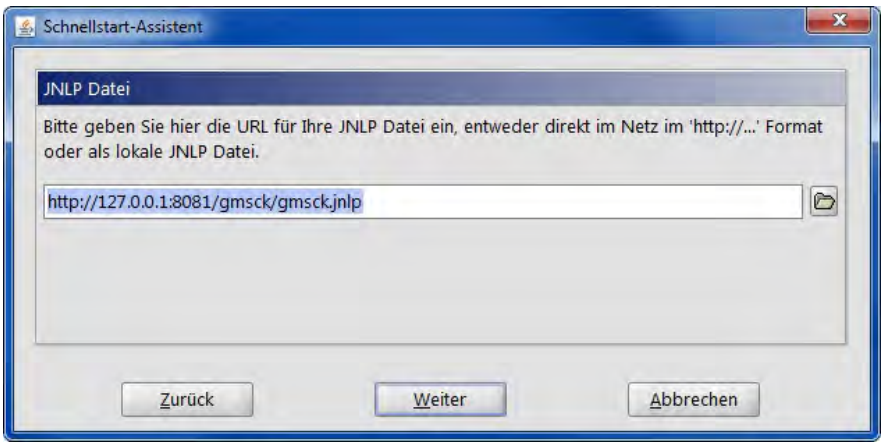

Abbildung 4.4: Eingabe der URL[2] zur JNLP in QF-Test

Als Letztes muss ein Name definiert werden, dieser dient als zentrale Referenz, um den Client in QF-Test zu identifizieren. Diesen kann man frei auswählen. Sind diese Schritte erfolgreich, kann man die Anwendung über QF-Test starten und mit der Testerstellung beginnen.

4.3 Realisierung und Durchführung

Nun werden mithilfe des Dokuments und den Vorbereitungen die Testfälle realisiert. Dafür muss erstmal die SUT über QF-Test gestartet werden. Daraufhin sollte sich die SUT öffnen und QF-Test gibt wie in Abbildung 4.5 an, dass der Client mit den genutzten Java-Techniken verbunden ist. Wie man außerdem der Abbildung 4.5 entnehmen kann, erkennt QF-Test für GMSC-K drei Techniken: AWT, JavaFX und Web. Die Abbildung 4.5 zeigt, dass der Schnellstart-Assistent eine Sequenz mit drei Prozeduraufrufen erstellt. Außerdem wird eine Variable (Name des Clients) gesetzt.

Bevor nun die ersten Testfälle erstellt werden, sollte man sicherstellen, dass die genutzten Techniken erkannt werden. Zudem sind erste Versuche nötig um sicherzustellen, dass die Verbindung zum Client korrekt abläuft und die Tests beziehungsweise Checks funktionieren.

1 Java Network Launching Protocol

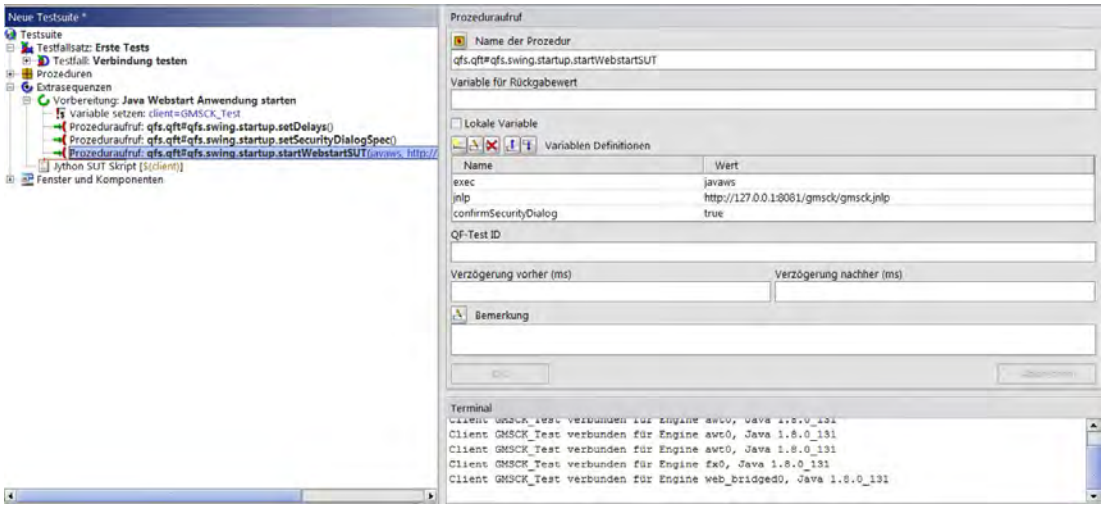

Abbildung 4.5: QF-Test ist mit der SUT verbunden

Bei GMSC-K funktionieren die Abbild-Checks mit der Karte nicht auf Anhieb. Die Grafiken für Flurstücke, Linien, Bemaßungen und Weiteres werden nicht wie gewünscht (siehe Abbildung 4.6) dargestellt. Stattdessen wird eine komplett leere Karte angezeigt (siehe Abbildung 4.7).

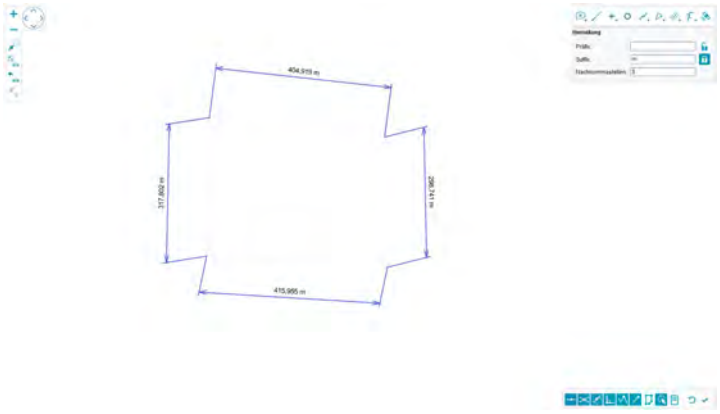

Abbildung 4.6: Gewünschte Abbildung vom Check-Abbild

Abbildung 4.7: Gelieferte Abbildung vom Check-Abbild

Da die wichtigsten Elemente im Abbild 4.7 fehlen, ist eine Lösung dieses Problems unausweichlich. So ist eine Testautomatisierung für GMSC-K nicht realisierbar, da viele Funktionen mit der Karte verknüpft sind.

Mithilfe des Support Teams von QF-Test ist es möglich ein Skript zu entwickeln, mit dem das Problem umgangen wird. Dieses Skript erlaubt es, Abbilder von verschiedenen GUI-Techniken zu realisieren. Das Ergebnis ist ein Jython-Skript (zu sehen beim Listing 4.1), wo ein eigener Checker implementiert wird.

Listing 4.1: Jython-Skript zum Erstellen eines eigenen Checker für Abbildungen

```
1 # {{{ Register checker
2 .....
3 glImageCheckType = DefaultCheckType(
4 "Rendered_image",               # Identifier
5 CheckDataType.IMAGE,            # Data type
6 "Rendered image"       # Text for menu
7 )
8 .....
9 def unregister():
10 try:
11 CheckerRegistry.instance().unregisterChecker("javax.swing.JComponent",
12 glImageChecker)
13 except:
14 pass
15
16 def register():
17 global glImageChecker
18 unregister()
19 glImageChecker = GLImageChecker()
20 CheckerRegistry.instance().registerChecker("javax.swing.JComponent",
21 glImageChecker)
22
23 register()
```

Wichtig in dem Listing 4.1 sind Zeile 11 und 20. Hier kann man die gewünschte Technik festlegen, welche erfasst werden soll. Da es sich in GMSC-K bei den Kartenelementen beziehungsweise Kartengrafiken um swing.JComponent handelt, wird dies in den jeweiligen Zeilen im Skript eingefügt. Dieser benutzerdefinierte Check mit dem Namen 'Rendered Image' liefert das Abbild wie in Abbildung 4.6.

Da die Funktionen sonst nativ funktionieren, sind keine weiteren Skripte zur Erweiterung des Funktionsumfang nötig. Somit ist das Erstellen der Testfälle möglich. Dafür nutzt man die Aufnahme-Funktion von QF-Test und führt die Testschritte wie in der Testspezifikation beschrieben aus. Neben den Checks des Endresultats sollten außerdem weitere Checks erstellt werden um beispielsweise sicherzustellen, dass die gewünschten Fenster der SUT mit ihren Bezeichnern korrekt sind.

Wie ein Testfall in der Testsuite für GMSC-K aussieht, kann man Abbildung A.1 entnehmen. Bei dem Testfall in Abbildung A.1 handelt es sich um die Testfälle 'Datensatz aus Suchergebnis öffnen' und 'Datensatz aus Suchergebnisse in der Karte selektieren'. Diese werden in QF-Test als eine Sequenz zusammengefasst, da sie unmittelbar zusammenhängen. Zudem ist der Umfang der einzelnen Testfälle überschaubar. Wie die spezifizierten Testfälle im Dokument beschrieben werden, sieht man in Abbildung A.3.

Step 1 vom Testfall 'Datensatz aus Suchergebnis öffnen' beschreibt im Prinzip einen Mausklick auf eine Lupe, um das Suchcenter zu starten. Außerdem handelt es sich in Abbildung A.1 um die erste Aktion der Sequenz 'Datensatz aus Suchergebnisse öffnen und in der Karte anzeigen'. Nach diesem Muster werden die Mausklicks, Checks und Eingaben der Reihe nach abgearbeitet, um so die Testsuite zu vervollständigen.

Bei GMSC-K ist zu beachten, dass es verschiedene Mappen gibt. Einmal die Mappe mit den Grafiken, die das Verarbeiten von Flächen und Daten umfasst und dann gibt es noch den 'Adminer'. Im 'Adminer' werden die Benutzer verwaltet und Datenquellen/Datenbanken verknüpft. Hier kann der Administrator also Benutzerrechte einstellen und die Mappen verwalten. Da ein Wechsel von der Mappe mit Grafik zum Adminer zeitaufwendig ist, werden verschiedene Testfallsätze angelegt. So gibt es einen Testfallsatz, in dem speziell die Testfälle vom Adminer abgearbeitet werden (siehe Abbildung 4.8). Durch diese Trennung wird die Laufzeit der Testfälle verringert.

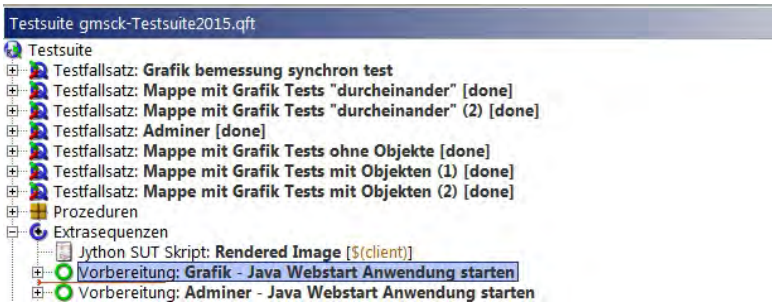

Abbildung 4.8: Testfallsätze für GMSC-K in QF-Test

Innerhalb der Testfallsätze befinden sich Testfälle. Diese wiederum beinhalten die konkreten Sequenzen mit den Aktionen und Checks. In Abbildung A.2 wird diese Struktur dargestellt. Wie eine einzelne Sequenz aussieht, kann man (teilweise) Abbildung A.1 entnehmen.

Sobald die Testfälle erfolgreich erstellt werden, müssen diese noch ausgeführt werden. Da die Testsuite mit einer älteren (stabilen) Version von GMSC-K verbunden ist, muss zunächst die neue Version mit der Testsuite in QF-Test verknüpft werden. Auf diese Weise können die Testfallsätze an der neuen Version durchgeführt werden.

4.4 Auswertung und Bericht

Zum Dokumentieren der Ergebnisse wird eine Kopie der Excel-Datei erstellt und bearbeitet. Da die Ergebnisse übersichtlich in einem Dokument aufgelistet werden sollen, müssen die Reports von QF-Test manuell auf dieses Dokument übertragen werden. Ein Auszug des Dokuments kann man in Abbildung 4.9 sehen.

	A	B	C	D	E	F	G	H
1	Test ID	Testname	Testergebnis: Grün - Kein Fehler gefunden Rot - Fehler gefunden Gelb - Test nicht ausgeführt (Doppelte Tests / nicht unterstützt in meiner Testumgebung / schwer nachzuvollziehen...)	Ergebnis	Anzahl Schritte	Testschritt Name	Testschritt Beschreibung	Testschritt Vorbedingung
2	30	Datenquelle anlegen		Adminer startseite wird angezeigt	6	Step 1	PolyGis Adminer starten	
3	30	Datenquelle anlegen		Das Kontextmenü erscheint	6	Step 2	Im Adminer mit rechter Maustaste auf Datenbanken Klicken	
4	30	Datenquelle anlegen		Die Maske zum Anlegen einer neue Datenquelle erscheint	6	Step 3	Im Kontextmenü zu Neu Navigieren, dann kommt ein weiteres Kontextmenü und jetzt auf neu Datensatz Klicken	
5	30	Datenquelle anlegen			6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen. Bezeichnung eingeben Datenbankserver: Name oder IP-Adresse des Servers eingeben Port auf den der Server hört eingeben Nutzername: Name des Datenbankbenutzer angeben Kenntwort:Kenntwort des Datenbankbenutzer eingeben	
7	30	Datenquelle anlegen		Daten sind eingegeben	6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen.Bezeichnung eingebenDatenbankserver: Name oder IP-Adresse des Servers eingeben DB NamePort auf den der Server hört eingebenNutzername: Name des Datenbankbenutzer angebenKenntwort:Kenntwort des Datenbankbenutzer eingeben	
8	30	Datenquelle anlegen		Daten sind eingegeben	6	Step 4	Datenbanktyp wird angeben: Dropdownlist benutzen und Datenbankmanagementsystem auswählen. Bezeichnung eingeben Datenbankserver: Name oder IP-Adresse des Servers eingeben Port auf den der Server hört eingeben Nutzername: Name des Datenbankbenutzer angeben Kenntwort:Kenntwort des Datenbankbenutzer eingeben	
9	30	Datenquelle anlegen		Pop-up-Fenster mit der Meldung "Verbindung erfolgreich aufgebaut" erscheint	6	Step 5	Oben in der Eingabemaske auf Datenquelle prüfen klicken und im Pop-up-Fenster "OK" klicken	
10	30	Datenquelle anlegen	Meldung "Datenbank <Bezeichnung> wurde gespeichert" wird so nicht angezeigt sondern nur "Datenbank <Bezeichnung>	unten links in der Statusleiste erscheint die Meldung "Datenbank <Bezeichnung> wurde gespeichert	6	Step 6	Oben in der Eingabemaske auf Speichern klicken	

Abbildung 4.9: Auszug der Testergebnisse

Wie man der Abbildung 4.9 entnehmen kann, wird Spalte ‘C’ abgeändert, um die Testergebnisse zu dokumentieren. Dabei werden Farben verwendet, um das Testergebnis zu visualisieren. Grün steht für ‘keine Fehler gefunden’ und rot für ‘Fehler beziehungsweise Abweichung gefunden’. Sind Felder gelb markiert, werden diese Testfälle nicht ausgeführt. Dies kann verschiedene Ursachen haben (wie beispielsweise doppelte Tests, Testfall lässt sich nicht automatisieren, Testfall wird in der Testumgebung nicht unterstützt, Testfall ist veraltet oder nicht erwünscht).

Manche Testfälle werden zwar nicht automatisiert aber trotzdem manuell durchgeführt. Dies wird im Dokument verdeutlicht, indem das Testergebnis (Spalte C) die Farbe rot oder grün besitzt, der Testfall (Spalte A/B) aber die Farbe gelb.

Das fertige Dokument mit den Ergebnissen wird dann an den Entwickler geschickt. Dieser kann mithilfe des Dokuments die Fehler eingrenzen und beheben. Bei Unklarheiten stehen die Entwickler und der/die Tester weiterhin in Kontakt.

5 Evaluation

In diesem Kapitel wird die Testsuite evaluiert. Dabei wird die Testsuite auf verschiedene Eigenschaften geprüft, wie beispielsweise die Laufzeit, Testabdeckung und Ähnlichem.

Es werden, wie beim Abschlussprozess vom ISTQB, die Schwächen der Testsuite dokumentiert. So wird sichergestellt, dass eventuelle Schwachstellen in Zukunft verbessert oder vermieden werden können.

5.1 Laufzeit

Die Laufzeit lässt sich mit QF-Test gut bestimmen, da dieser Wert nach der Ausführung der Testfälle im Protokoll angezeigt wird. Der Wert ist jedoch volatil, da sich die Reaktionszeiten von GMSC-K ändern und QF-Test dementsprechend länger auf das Laden von Komponenten warten muss. Laufen die von GMSC-K benötigten Dienste länger ohne neu zu starten, dauern die Aktionen und Prozesse länger. Somit lässt sich keine eindeutige Aussage zur Laufzeit treffen. Es wird aber ein Durchschnittswert ermittelt und angegeben. Da die Testfallsätze ungefähr den gleichen Umfang haben, wird die Durchschnittslaufzeit eines Testfallsatzes mit der Anzahl der Testfallsätze multipliziert.

In Abbildung 5.1 kann man die Laufzeit für einen Testfallsatz entnehmen, bei denen die Dienste nicht lange (maximal ein Tag) laufen. Dabei ist die Dauer in Echtzeit relevant, da diese die exakt benötigte Zeit widerspiegelt (mit dem Warten auf Komponenten und Ähnlichem). Es werden fünf Durchläufe ausgeführt, was zu einer Durchschnittslaufzeit (ohne die Millisekunden eingerechnet) von 8,04 Minuten (oder 8:02,400 min) führt. Optimiert man den Prozess, sollte sich die benötigte Echtzeit an der angegebenen Dauer von QF-Test nähern. Mit dem Durchschnittswert ergibt sich eine ungefähre Laufzeit der ganzen Testsuite von $8{,}04 Minuten \cdot 6$ (relevante Testfallsätze) = 48,24 Minuten.

Da in dieser Testsuite viele sogenannte ‘harte Events’ genutzt werden, steigert sich die Laufzeit zunehmend. Diese werden jedoch für GMSC-K benötigt, da viele Events (Mausklicks) nicht als ‘weiche Events’ erkannt beziehungsweise ausgeführt werden. Vor allem bei der Karte (beispielsweise bei der Erstellung von Flurstücken als Grafik über die Karte) werden die weichen Events nicht vollständig erkannt.

Abbildung 5.1: Laufzeiten für einen Testfallsatz

5.2 Testabdeckung

Die Testabdeckung (englisch: Code coverage) ist in der Black-Box nur schwer nachzuvollziehen, da der Zugriff auf den Quellcode fehlt. Üblicherweise ermittelt man die Testabdeckung in der White-Box.

Weil der Funktionserhalt im Fokus steht und jede getestete Funktion mindestens einmal ausgeführt wird, sind die Anforderungen für den Black-Box-Test erfüllt. Trotzdem lässt sich eine Aussage über die Abdeckung treffen und zwar die Tatsache, dass eine Funktionsabdeckung stattfindet.

Da nicht jeder Testfall automatisiert wird, kann man nicht von einer hundertprozentigen Funktionsabdeckung ausgehen. Um die Testabdeckung also in diesem Fall zu ermitteln, wird die Anzahl der automatisierten Testfälle durch die Gesamtanzahl der Testfälle (abzüglich der veralteten Testfälle sowie der Testschritte zur Installation) dividiert. Doppelte Testschritte werden zudem von Excel (teilweise) gefiltert. Das Ergebnis liefert dann die Testabdeckung in Prozent. Damit ergeben sich aus 1 957 automatisierten Testfällen beziehungsweise Testschritten aus insgesamt 3 365 (abzüglich 931) Testschritten folgende Testabdeckung: 1957/(3365-931) = 0,8040 -> 80.40% Testabdeckung

Die fehlende Testabdeckung von circa 20% lässt sich vor allem auf schwer zu testende Imports/Exports zurückführen. Um diese vollständig/korrekt zu automatisieren, müsste die Testautomatisierung viele Datensätze überprüfen, was die Laufzeit und den Umfang der Testsuite signifikant steigern würde. Zudem nehmen die Imports/Exports an sich viel Zeit in Anspruch. Die Abzüge (931) lassen sich unter anderem aus folgenden Ursachen ermitteln: Installationsschritte, veraltete Testschritte und fehlende Funktionen in der Testumgebung.

5.3 Stabilität

Um eine stabile beziehungsweise robuste Testsuite zu entwerfen, bietet QF-Test mehrere Ansätze an. Dabei spielt die Komponentenerkennung eine äußerst wichtige Rolle. Mit der Standardeinstellung von QF-Test werden die Komponenten anhand der Namen, der Klasse (beziehungsweise des Typs) und der Hierarchie erkannt. [Sof17] Mit diesen Einstellungen wird GMSC-K getestet.

Für eine robuste Testsuite mit QF-Test sollten die Komponenten im Idealfall über deren Namen erkannt werden. Dies kann jedoch nur funktionieren, wenn die Entwicklung den Komponenten eindeutige Namen/ID vergibt. [Sof17] Wird den Komponenten (besonders Fenstern und Dialogen) keine Namen/ID vergeben, funktioniert die Erkennung in QF-Test per Wahrscheinlichkeitsrechnung über die Beschriftung/Titel, Struktur und Geometriedaten. Es ist möglich, die Gewichtung der einzelnen Elemente zu modifizieren, um den Bedürfnissen der Anwendung gerecht zu werden. Starke Veränderungen der Namen oder der Komponentenhierarchie können mit diesen Einstellungen zu Problemen führen. [Sof17]

In GMSC-K wird den Komponenten keine Namen vergeben, weshalb die Erkennung von QF-Test mit der Wahrscheinlichkeitsrechnung durchgeführt wird. Dies kann jedoch Probleme verursachen, insbesondere wenn neue Versionen von GMSC-K getestet werden. Ändert sich die Auflösung oder die Beschriftung/Titel von Fenstern, werden diese wahrscheinlich nicht mehr erkannt. [Sof17]

Mit QF-Test lässt sich die Stabilität allerdings auch in so einem Szenario weiter verbessern, man kann beispielsweise sogenannte 'NameResolver' einsetzen. Diese ermöglichen Namen aus der Sicht von QF-Test zu ändern (der Quellcode der SUT bleibt unverändert), um die Erkennung zu perfektionieren. Dieser Prozess kann allerdings sehr aufwendig sein, weshalb es praktikabler ist, wenn die Entwickler Namen beziehungsweise ID's vergeben. Mehr zum Thema 'NameResolver' findet man im Handbuch über QF-Test in Abschnitt 28.4 und 41.1.6. [Sof17]

Es lassen sich auch dynamische Fenstertitel verwenden (zu sehen in Abbildung 5.2), diese kann man einfach im Merkmal-Attribut in QF-Test setzen. Sollte sich der Fenstertitel ändern (Versionsnummer, eingeloggte Nutzer oder Ähnliches), ist es möglich, reguläre Ausdrücke zu benutzen. Dies hat den Vorteil, dass die Fenster auch mit dynamischen Titeln erkannt werden.

Des Weiteren spielt die Fenstergröße eine wichtige Rolle in der Testsuite für GMSC-K, da diese viele Bildabgleiche beinhaltet. Ändert sich die Fenstergröße, wird auch die Größe des Fensters für die Karte angepasst, was dazu führt, dass die Bildabgleiche fehlerhaft sind. Es lassen sich zwar Vergleichsgrößen im Algorithmus-Attribut beim Abbild-Knoten erzwingen, diese führen aber zu Unschärfen. Außerdem könnten wichtige Elemente bei einem kleineren Fenster fehlen, sodass die Checks weiterhin fehlerhaft sind.

Um dieses Problem zu lösen, sollte ein Vorbereitungsknoten erstellt werden, welcher das Hauptfenster der SUT auf eine Ausgangsgröße setzt. Damit wird gewährleistet, dass die Bildabgleiche auch auf anderen Systemen (und Bildschirmen) korrekt ausgeführt werden. Es sollte generell darauf geachtet werden, dass die Ausgangssituation immer möglichst unverändert bleibt.

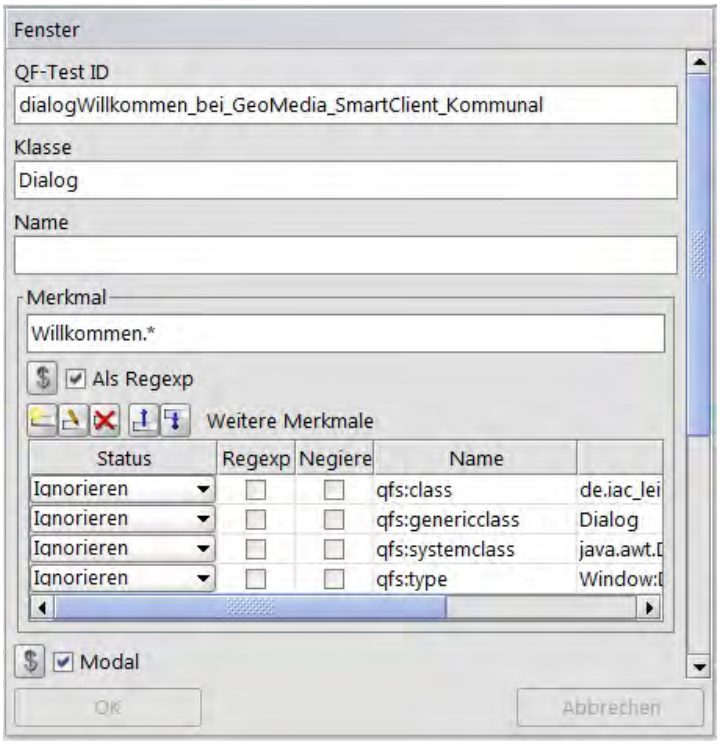

Abbildung 5.2: Dynamische Fenstertitel in QF-Test

5.4 Abstraktion

Um eine Testsuite in QF-Test möglichst abstrakt zu halten, müssen verschiedene Bereiche beachtet werden. Ein eigenständiges Kapitel ist für ein solches Konzept im QF-Test-Handbuch nicht zu finden. Es ist jedoch mit der Funktionsvielfalt von QF-Test möglich, eine größtenteils abstrakte Testsuite zu erstellen. Für dieses Konzept sind folgende Bereiche relevant:

- **Testfälle und Testfallsätze:** Diese Knoten lassen sich gut abstrahieren, da man diese per Include-Dateien (siehe Abbildung 5.3) oder expliziten Referenzen in unterschiedlichen Testsuiten aufrufen kann. Nutzt man Datenquellen (beispielsweise Excel-Datentreiber), müssen nur die Dateinamen und die Verzeichnisstruktur gleich bleiben. Die konkreten Daten werden dann mit den Prozeduren verarbeitet.

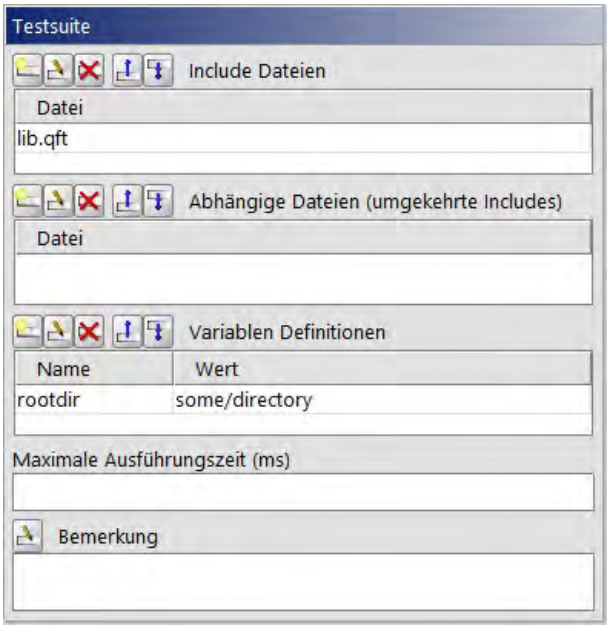

Abbildung 5.3: Attribute einer Testsuite mit Include-Dateien [Sof17]

- **Prozeduren:** Prozeduren sind schwerer zu abstrahieren, da diese den Ablauf der GUI abbilden. Um diese ansatzweise abstrahierbar zu machen, muss ein Rahmen geschaffen werden, in dem jeder Dialog in ein Package gepackt wird. Dieser wird dann mit den absolut nötigsten Prozeduren gefüllt (nur Pflichtfelder ausfüllen und Ähnliches). So lässt sich der Aufruf in einer anderen Testsuite variabel zusammenstecken.

- **Komponenten:** Komponenten sind sehr anwendungsabhängig, da mit ihnen die GUI der Anwendung widergespiegelt wird. Sind sich die Anwendungen jedoch ähnlich und mit demselben Framework gebaut, ist es möglich, die Komponenten zu abstrahieren. QF-Test bietet dafür generische Komponenten an, mit denen die Attribute der Komponenten manuell entfernt werden, um die Erkennung unschärfer beziehungsweise variabler zu gestalten. Mit den generischen Komponenten ist es außerdem möglich, viele ähnliche Dialoge einer SUT in eine Komponente zu packen. Diese Komponente könnte man dann mit einer Prozedur wie ‘klickeJa’ in jedem Dialog ansprechen. Werden die Testfälle aufgezeichnet, ist dieses Konzept ungeeignet. Stattdessen sollten Prozeduren genutzt werden. Mehr zu den generischen Komponenten findet man im QF-Test-Handbuch, Kapitel 28.6.

Das Kapitel 21 im QF-Test-Handbuch präsentiert Schlüsselwort-getriebene Testsuiten mit variablen Prozeduraufrufen und generischen Komponenten. Dort wird nicht das Konzept einer Abstraktion an sich behandelt, jedoch werden die dafür benötigten Funktionen genutzt.

Um eine Testsuite zu abstrahieren, müssten die Tests von Beginn an darauf konzipiert werden. Mit den genutzten Einstellungen und der momentanen Realisierung der Testfälle lässt sich die in dieser Arbeit erstellte Testsuite nicht abstrahieren, um andere (ähnliche) Anwendungen zu testen. Da dies auch keine Anforderung ist und ähnliche Projekte/Anwendungen nicht geplant sind, wird diese Eigenschaft vernachlässigt.

6 Zusammenfassung

In diesem Kapitel werden die in der vorliegenden Thesis erarbeiteten Erkenntnisse rekapituliert.

6.1 Fazit

Das Ziel der Arbeit ist es, eine Testautomatisierung mit QF-Test zu entwickeln. Anhand der Fragestellungen (Kapitel 1.2) und Anforderungen (Kapitel 3.1) wird diese Arbeit im Fazit bewertet. Dabei wird erst auf die Fragestellungen eingegangen.

Das Ergebnis zeigt, dass es möglich ist, mit QF-Test alleine eine Testautomatisierung in der Black-Box für eine etablierte Anwendung zu erstellen, welche auch ohne Programmierkenntnisse weiter gewartet und genutzt werden kann. Mit den passenden Werkzeugen lässt sich eine Testautomatisierung für die meisten Anwendungen erstellen, ohne die interne Struktur zu verändern. Wie die Arbeit zeigt, ist es jedoch von Vorteil, wenn man die interne Struktur anpassen kann, um die Testprozesse zu verbessern.

Um eine effektive Testautomatisierung in der Black-Box zu erstellen, gibt es Capture & Replay-Testwerkzeuge wie QF-Test. Solche Werkzeuge reichen meist aus, um eine Testautomatisierung zu erstellen. Es gibt jedoch auch Alternativen wie Jubula (Kapitel 3.2.2), mit denen es möglich ist, eine Testautomatisierung ohne Capture & Replay in der Black-Box zu erstellen.

Auch wenn es Alternativen zu QF-Test gibt, bietet QF-Test einen sehr guten und schnellen Support an, welcher sich von anderen (vor allem kostenfreien) Werkzeugen abhebt. QF-Test ist lückenlos dokumentiert und es werden Schulungen (auch Web-Seminare) angeboten, um den Einstieg zu erleichtern. Dies hat den Vorteil, dass (Anfänger-)Fehler vermieden werden und von Anfang an eine effiziente und robuste Testsuite erstellt werden kann. Des Weiteren werden viele verschiedene Ansätze und Funktionen von QF-Test geliefert, weshalb dieses Werkzeug besonders hervorsticht.

Wie die Arbeit in Kapitel 4 zeigt, sind bei der Erstellung einer Testsuite mit Testwerkzeugen wie QF-Test in den meisten Fällen keine Programmierkenntnisse erforderlich, aber von Vorteil. Außerdem lässt sich die Testsuite auch von nicht involvierten Personen mit einer kurzen Einarbeitung erweitern. Die Grundfunktionen (und auch die GUI an sich) von QF-Test lassen sich intuitiv nutzen. Trotzdem bietet es viele weitere komplexe

Möglichkeiten (bei denen dann Programmierkenntnisse notwendig sind), um eine Testsuite robuster und effektiver zu gestalten (beispielsweise mit den NameResolvern) ohne den Quellcode zu verändern.

Der Testprozess wird mit der Testautomatisierung verbessert und vereinfacht. Vor allem wenn vorher keine strukturierten Testprozesse stattfinden, sinkt durch die Testautomatisierung die Fehlerwahrscheinlichkeit rapide. Wird die Testsuite robust gestaltet und weiter gewartet, ist der Aufwand der Entwicklung einer Testsuite gerechtfertigt, da das zeitaufwendige (regelmäßige) manuelle Testen von hunderten oder tausenden Funktionen wegfällt. Der Mehreffekt und die Definition vom Testen wird im Kapitel 2 ausführlich beantwortet.

Eine Testautomatisierung mit QF-Test alleine reicht allerdings in den meisten Fällen nicht aus, um alle Testfälle abzudecken wie die Evaluation (Kapitel 5) deutlich macht. Manche Funktionen lassen sich nicht (oder nicht besonders gut) automatisieren. Außerdem sollten Tests in allen Testebenen stattfinden, weshalb auch Unit-Tests erstellt werden sollten.

Die erstellte Testautomatisierung in dieser Arbeit lässt sich auch ohne den Entwickler durchführen und auswerten. Mit QF-Test ist es möglich, die Testsuite mit einem Mausklick zu starten und das gelieferte Protokoll anschaulich auszuwerten.

Die funktionalen Anforderungen werden größtenteils eingehalten. Die Testsuite liefert automatisierte Testprozesse in der Black-Box, welche über die GUI mit Capture & Replay stattfinden. Auch wenn QF-Test kein extremes invasives Verhalten besitzt, ist es von Vorteil, wenn Änderungen am Code durchgeführt werden, was in Kapitel 5.3 verdeutlicht wird.

Mithilfe des Listings 4.1 sind Bildabgleiche mit QF-Test möglich. Zudem ist die Kompatibilität mit Java gegeben, da QF-Test speziell für Java entwickelt wird. Mit QF-Test lässt sich der Ablauf sehr genau steuern, da viele Ablaufsteuerung-Knoten dafür vorgesehen sind (siehe Kapitel 3.2.3). Nach jedem Durchlauf der Testsuite präsentiert QF-Test die Testergebnisse automatisch. Damit lassen sich über QF-Test Reports erstellen.

Auch die nicht-funktionalen Anforderungen werden eingehalten, da die Smoketests abgearbeitet werden. Jedoch sind nicht alle Testschritte in der Testautomatisierung integriert, weshalb diese Anforderung nicht vollständig erfüllt wird. Die Testsuite bietet trotzdem ein solides Grundbaustein für weitere Tests. Da die relevanten Listings schon eingefügt sind, lässt sich die Testsuite ohne (Programmier-) Kenntnisse ausführen und erweitern. Die Reports können klar strukturiert in QF-Test erstellt werden und an die Entwickler kommuniziert werden, um so die Testergebnisse möglichst effizient nachvollziehen zu können.

Die Ergebnisse der Arbeit zeigen, dass die Testsuite den Anforderungen zwar genügt, jedoch noch Verbesserungspotenzial besteht. Vor allem die Stabilität ist noch einer der größeren Schwachstellen. Die Fragestellungen werden in dieser Arbeit vollständig beantwortet, auch wenn die Antworten vom Anwendungsfall abhängen.

6.2 Weitere Ansätze

Zu dem Thema Testautomatisierung gibt es zahlreiche Ansätze, jedoch gibt es keine, die sich explizit mit der Erstellung einer Testsuite mit QF-Test für eine GIS-Anwendung beschäftigt.

Es gibt aber Arbeiten, die sich mit QF-Test auseinandersetzen. Eine davon ist von Moritz Hegel - „Design und Implementierung einer Testautomatisierung der Schnittstellentests zwischen Einsatzleitsystemen und einem Notrufabfragesystem“. [Heg14] Dieser befasst sich mit der Testautomatisierung von Schnittstellentests zwischen dem Einsatzleitsystem und einem Notrufabfragesystem. Die Liste mit Arbeiten, welche sich mit QF-Test auseinandersetzen, kann man [Sof17] entnehmen. Dort sind unter den Kundenmeinungen Evaluationsberichte aufgelistet, welche sich mit QF-Test beschäftigen.

6.3 Nächste Schritte

Da mit der aktuellen Testsuite das Fundament geschaffen wird, ist der nächste sinnvolle Schritt die Implementierung von Testfällen für Plugins. Dafür müssen erstmal Testfälle spezifiziert werden, da (im Gegensatz zu den Grundfunktionen) für die Plugins noch keine Tests dokumentiert sind. Aufgrund vieler verschiedener Plugins in GMSC-K (gesplittete Abwassergebühren, wiederkehrende Straßenbeiträge und viele mehr) ist dies nochmal eine umfangreiche Aufgabe.

Ein weiterer Schritt ist die Verbesserung der Testsuite. Um diese robuster zu gestalten, sollte erstmal ein Vorbereitungsknoten das Hauptfenster auf eine Ausgangsgröße setzen, damit die Bildabgleiche auch in anderen Testumgebungen funktionieren. Die Testsuite muss zudem regelmäßig gepflegt und gewartet werden, um sich so stets den Veränderungen der SUT anzupassen.

6.4 Ausblick

Ein Testprozess hat selbst in kleineren Unternehmen eine immer größere Bedeutung. Um wertvolle Ressourcen sinnvoll einzusetzen, sollte dieser Prozess weitestgehend automatisiert werden.

Die Testsuite ist in der Testumgebung voll funktionsfähig, ist aber stets erweiterbar. Mit den genannten Anpassungen lässt sich die Testsuite allerdings auch in weiteren Umgebungen ausführen. Außerdem sollte die Komponentenerkennung verbessert werden. Dafür sollten NameResolver oder dynamische Fenstertitel für die SUT in QF-Test eingeführt werden. Da dies jedoch ein zeitaufwendiger Prozess sein kann, wird als Alternative auch über Namen/ID im Sourcecode nachgedacht.

Es wäre möglich und sinnvoll, einen Standard zu entwickeln, mit welchen man die Testfälle spezifiziert. Auch für die Umsetzung/Erstellung der Testfälle sollte ein Standard entwickelt werden um zu garantieren, dass die Testsuite eine ordentliche Struktur beibehält. Die Testfälle kann man so spezifizieren, dass es möglich ist, Schlüsselwort-getriebenes (Keyword-Driven) Testing mit QF-Test zu nutzen (mehr dazu im Kapitel 21 vom QF-Test-Handbuch). [Sof17] Damit wird auch eine gewisse Abstraktion geschaffen.

Literaturverzeichnis

[AP97] Amit Paradkar, M.A. Vouk, K.C. Tai: Specification-based testing using cause-effect graphs (1997)

[Bec15] Becker, KC: Kommunal-Consult Becker AG (2015), URL http://www.kc-systemhaus.de/

[BRE17] BREDEX: Professionell entwickeltes Testautomatisierungswerkzeug ohne Lizenzkosten (2017), URL https://www.bredex.de/leistungen/qualitaetssicherung/jubula/

[Con17] Consultants, Financial Software: Test-Automatisierung (2017), URL http://www.fsc-consultants.de/test-automatisierung/

[Dev17] Developer, Mozilla: Setting up your own test automation environment (2017), URL https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Your_own_automation_environment

[Fo17] FH-osnabrueck: TPTP Automated GUI Recorder (2017), URL http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/TPTP/TPTP_AGR.html

[FPBM16] Fachrul Pralienka Bani Muhamad, et al.: Visual GUI testing in continuous integration environment (2016)

[Fro11] Froh, Christian: Untersuchungen zu automatisierbaren Testszenarien für User Interfaces von kaufmännischen Softwaresystemen (2011)

[Fro17] Froglogic: Squish GUI Tester (2017), URL https://www.froglogic.com/squish/

[GjM11] Glennford j. Meyers, Corey Sandler, Tom Badgett: *The Art of Software Testing* (2011)

[GR00] Gregg Rothermel, Jeinay Dedhia, Mary Jean Harrold: Regression Test Selection for C++ Software (2000)

[Hal84] Haley, Allen: Development and application of a white box approach to integration testing (1984)

[Heg14] Hegel, Moritz: Design und Implementierung einer Testautomatisierung der Schnittstellentests zwischen Einsatzleitsystemen und einem Notrufabfragesystem (2014)

[Hex13] Hexagon: GeoMedia SmartClient Dokumentation (2013), URL http://smartclient.intergraph.at/documentation

[Hoc17] Hocke, Raimund: RaiMan's SikuliX (2017), URL http://sikulix.com/

[Huc15] Huckle, Thomas: Collection of Software Bugs (2015), URL https://www5.in.tum.de/persons/huckle/bugse.html

[Int13] Intergraph: GeoMedia SmartClient Kommunal Datenblatt (2013), URL http://www.intergraph.com/assets/pdf/GeoMedia-Smart-Client-Kommunal_PRINT.pdf

[Kin15] Kinder, Johannes: Hypertesting: The Case for Automated Testing of Hyperproperties (2015), URL https://pure.royalholloway.ac.uk/portal/files/24051361/hotspot15.pdf

[McC04] McConnell, Steve: *Code Complete: A Practical Handbook of Software Construction, Second Edition* (2004)

[MEK12] Mohammed Ehmer Khan, Farmeena Khan: A Comparative Study of White Box, Black Box and Grey Box Testing Techniques (2012)

[Mic17] MicroFocus: Silk Test Überblick (2017), URL https://www.microfocus.com/de-de/products/silk-portfolio/silk-test/

[NK11] Nadja Krümmel, Bodo Igler: Kurzanleitung Testen (2011), URL https://homepages.thm.de/~hg11260/mat/testentwurf.pdf

[Pac17] Packard, Hewlett: Unified Functional Testing (2017), URL https://saas.hpe.com/de-de/software/uft

[Pat05] Patton, Ron: *Software Testing (2nd Edition)* (2005)

[Ren12] Renz, Burkhardt: Softwaretechnik - Systematisches Testen (2012)

[Sel17] SeleniumHQ: SeleniumHQ Browser Automation (2017), URL http://www.seleniumhq.org/

[Sma17] SmartBear: TestComplete Übersicht (2017), URL https://smartbear.com/product/testcomplete/overview/

[Sof17] Software, Quality First: QF-Test - Das GUI Testtool für Java und Web (2017), URL https://www.qfs.de/qf-test/testautomatisierung-mit-qf-test.html

[Spi08] Spillner, Andreas: *Systematisches Testen von Software* (2008)

[Tho17] Thomas, Oliver: Erweiterung der Bachelorthesis Testautomatisierung mit QF-Test (2017), URL https://www.mni.thm.de/forschung/institute-a-gruppen/ii/ii-ueberblick

[Wü15] Wüst, Klaus: Einführung in den systematischen Softwaretest (2015)

[Zel06] Zeller, Andreas: Software-Test: Strukturtest (2006)

Abkürzungsverzeichnis

AJAX Asynchronous JavaScript and XML

API Application Programming Interface (Programmierschnittstelle)

AUT Application Under Test (Anwendung unter Test)

AWT Abstract Window Toolkit

CSS Cascading Style Sheets

Eclipse Open Source IDE

ExtJS Extended JavaScript

FAQ Frequently Asked Questions

GEF Graphical Editing Framework

GIS Geoinformationssystem

GMSC-K GeoMedia SmartClient Kommunal

GUI Graphical User Interface

GWT Google Web Toolkit

HP Hewlett-Packard (Unternehmen)

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Enviroment (Integrierte Entwicklungsumgebung)

IntelliJ IDEA IDE für Java

ISTQB International Software Testing Qualifications Board

iOS Apple Betriebssystem (Operating System)

JavaFX Java Framework für Multimedia und GUI

JNLP Java Network Launching Protocol

JUnit Framework zum Testen von Java Programmen

jar Java-Archiv

KC Kommunal Consult

MIT Massachusetts Institute of Technology

Netbeans Ist eine freie IDE, welche mit Java geschrieben wurde

negieren Wahrheitswert wird in sein Gegenteil gewandelt

OpenCV Open Source Computer Vision Library

Open Source Softwareprojekte, deren Quellcode öffentlich ist (nutz- und änderbar)

PostgreSQL Postgre Structured Query Language

png Portable Network Graphics

py Python-Datei/Skript

QFS Quality First Software (Unternehmen)

QF-Test Quality First Test (GUI-Testtool für Java & Web)

RCP Rich Client Platform

ROI Return on Investment

Selenium Framework für automatisierte Softwaretests von Webanwendungen

SOAP Simple Object Access Protocol

Standalone Anwendung funktioniert eigenständig

Step Schritt

SUT System Under Test

Swing GUI-Toolkit für Java

SWT Standard Widget Toolkit (Java GUI-Toolkit)

TPTP Test and Performance Tools Platform

URL Uniform Resource Locator

VGT Visual GUI Testing

WebDriver Driver von Selenium

XML Extensible Markup Language

.xls Microsoft Excel-Datei

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

A Anhang 1

Abbildung A.1: Ein Testfall für GMSC-K in QF-Test

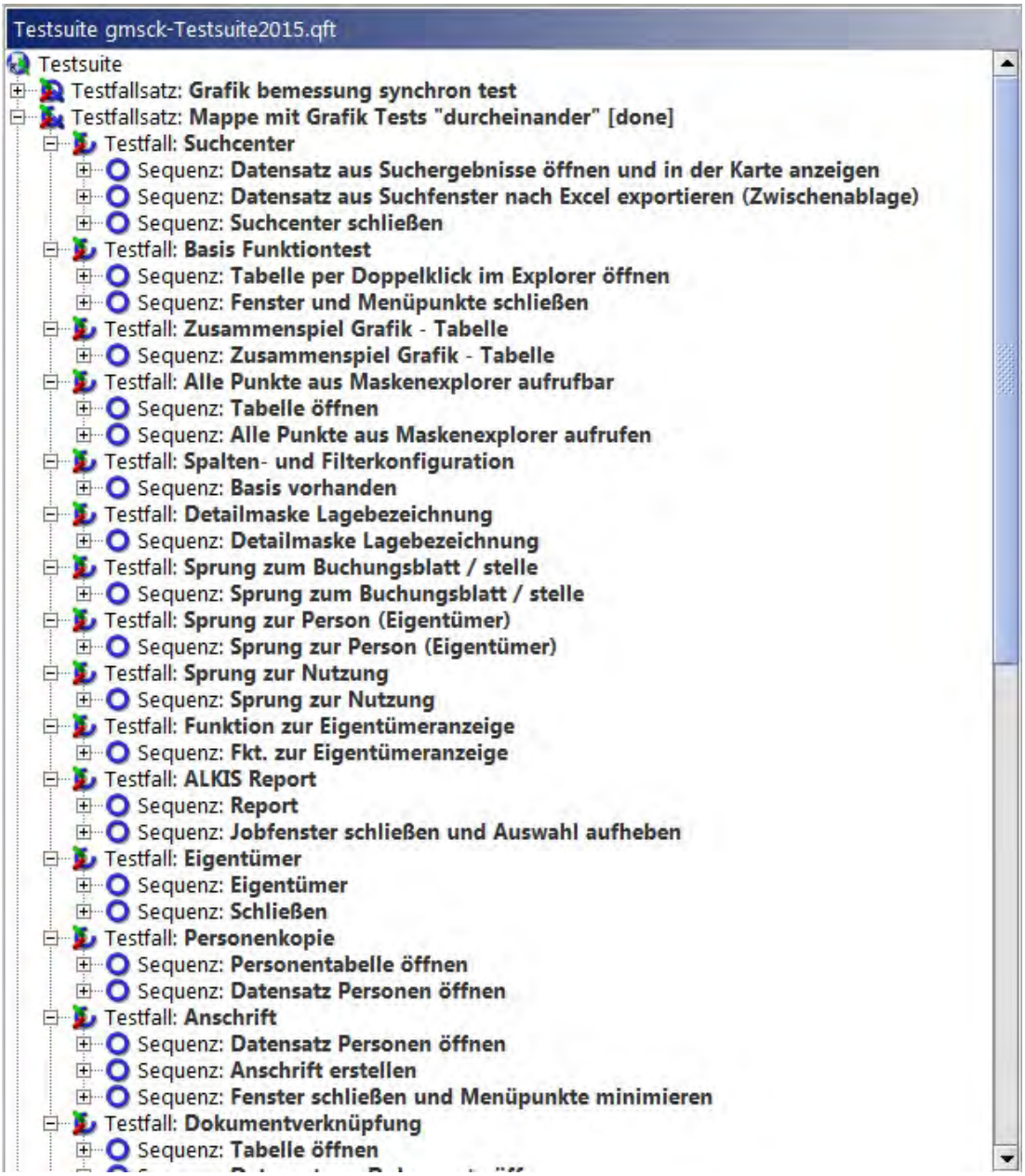

Abbildung A.2: Struktur der Testfälle in QF-Test

	A	B	C	D	E	F	G	H
926	359	Datensatz aus Suchergebnissen öffnen		Das Suchfenster startet und enthält alle Filterkonfigurationen die dem Suchfenster zugewiesen wurden	3	Step 1	Das GMK Fenster öffnen und unten rechts in Statusleiste das Suchcenter starten(Lupe)	Suchdefinition vorhanden
927	359	Datensatz aus Suchergebnissen öffnen		Das Suchfenster startet und enthält alle Filterkonfigurationen die dem Suchfenster zugewiesen wurden	3	Step 1	Das GMSCK Fenster öffnen und unten rechts in Statusleiste das Suchcenter starten(Lupe)	Suchdefinition vorhanden
928	359	Datensatz aus Suchergebnissen öffnen		Das Suchfenster startet und enthält alle Filterkonfigurationen die dem Suchfenster zugewiesen wurden	3	Step 1	Das PolyGis Fenster öffnen und unten rechts in Statusleiste das Suchcenter starten(Lupe)	
929	359	Datensatz aus Suchergebnissen öffnen		Das Suchfenster startet und enthält alle Filterkonfigurationen die dem Suchfenster zugewiesen wurden	3	Step 1	Das PolyGis Fenster öffnen und unten rechts in Statusleiste das Suchcenter starten(Lupe)	Suchdefinition vorhanden
930	359	Datensatz aus Suchergebnissen öffnen		Die Suchergebnisse erscheinen in der Tabelle	3	Step 2	im Suchcenter auf eine Suche klicken und unten die Suchparameter eingeben und auf " Suche starten" klicken	
931	359	Datensatz aus Suchergebnissen öffnen		Datensatz wird geöffnet	3	Step 3	aus der Ergebnismenge den Datensatz selektieren den man öffnen will und auf "öffnen" klicken	
932	359	Datensatz aus Suchergebnissen öffnen		Datensatz wird geöffnet	3	Step 3	aus der Ergebnissmenge der die Datensätze selektieren die man öffnen will und auf "öffnen" klicken	
933	360	Datensatz aus Suchergebnisse in der Karte selektieren		Das Suchfenster startet und enthält alle Filterkonfigurationen die dem Suchfenster zugewiesen wurden	3	Step 1	Das GMSCK Fenster öffnen und unten rechts in Statusleiste das Suchcenter starten(Lupe)	Suchdefinition vorhanden
934	360	Datensatz aus Suchergebnisse in der Karte selektieren		Die Suchergebnisse erscheinen	3	Step 2	im Suchcenter auf eine Suche klicken und unten die Suchparameter eingeben und auf " Suche starten" klicken	
935	360	Datensatz aus Suchergebnisse in der Karte selektieren		die Objekte werden in der Karte selektiert dargestellt	3	Step 3	Aus der Ergebnismenge die Datensätze selektieren die man in der Karte sehen will und auf "Objekte in der Karte anzeigen" klicken	
936	360	Datensatz aus Suchergebnisse in der Karte selektieren		die Objekte werden in der Karte selektiert dargestellt	3	Step 3	aus der Ergebnissmenge der die Datensätze selektieren die man in der Karte sehen will und auf "Objekte in der Karte anzeigen Klicken"	

Abbildung A.3: Auszug der Testfälle vom Dokument