Hochschule Neubrandenburg
University of Applied Sciences


**Bachelor-Studiengang Geoinformatik**
**Benjamin Naedler, 6. Semester**

# Bachelorarbeit

Zum Erlangen des akademischen Grades
„Bachelor of Engineering“ (B.Eng.)

Thema:

## Analyse der fachlichen und technischen Anforderungen für automatische Tests von GIS-Komponenten in profil c/s und die prototypische Umsetzung mit Sikuli

**Betreuer:** **Prof. Dr.-Ing. Andreas Wehrenpfennig**
**Dipl. Informatiker Robert Theel**

**URN:** **urn : nbn : de : gbv : 591 - thesis 2012 - 0198 - 8**

**Abgabedatum:** **17.8.2012**

„Software products are never released

- they escape“

[Cem Kaner, LV: 01]

# Zusammenfassung und Abstract

Die Arbeit befasst sich mit dem Problem des automatisierten Testens von GIS-Komponenten, also dem Test des Features, das im Kartenausschnitt dargestellt wird. Beispielhaft für den GIS-Bereich dient das Tool RFK-GIS-Editor der Software **profil c/s**. Das bisher eingesetzte Testwerkzeug zur Testautomatisierung, QF-Test, kann diese Kontrolle nicht durchführen. Daher wird nach einem Testwerkzeug zur Erweiterung der Testautomatisierung mit QF-Test gesucht. Dafür werden die fachlichen und technischen Anforderungen untersucht. Dies beinhaltet die Analyse des RFK-GIS-Editors, also die Auflistung, was zu testen ist, und die daraus folgenden Anforderungen an die Testsoftware für die Erweiterung der Testautomatisierung. Nach Recherchen standen vier Softwares zur Auswahl. Das einzige Werkzeug der Auswahl, welches für die Testumgebung geeignet ist und die entsprechenden Tests durchführen kann, ist Sikuli. Dieses Testwerkzeug realisiert automatisierte Abfolgen basierend auf Bildvergleichen, deren Ähnlichkeit stufenlos definierbar ist, und eignet sich somit für die Erweiterung der Testautomatisierung. Die prototypische Umsetzung von beispielhaften Testfällen verdeutlicht dies ebenfalls und stellt das Ergebnis der Arbeit dar.

The subject of this study is the automated testing of GIS components, so the feature text, which is represented in a map section. The tool commonly used in the field of GIS as far as now is RFK-GIS-Editor from the software **profil c/s**. QF Test, the automated testing tool, is unable to carry out this checking operation thoroughly. Therefore a further testing instrument is being looked for in order to enhance the automated testing with QF-test.
With this aim in view, specific and technical requirements have been investigated. This includes the analyse of RFK-GIS Editor, so the list of what is to be tested and further and consequently what will be required from the test software for the enhancement of the automated testing operation. After this research four software could be selected. The only testing tool in this selection, which is perfectly adjusted to the test environment and able to perform the automated testing, is Sikuli. This testing tool performs automated sequences by comparing images, which similarity can be defined continuous and this makes it suitable for an enhancement of the automated testing. Prototypical implementation of automated testing cases illustrate this ascertainment and bring the results of this study.

# Inhaltsverzeichnis

# 1 Einleitung

## Einführung

Ein wichtiger Bestandteil der Softwareentwicklung ist das Testen und somit die Qualitätssicherung der Software über den gesamten Entwicklungsprozess. Die Qualitätssicherung umfasst als erstes die Planung, also die Festlegung von Merkmalen, nach denen geprüft wird. Anschließend erfolgt die Prüfung, ob die Voraussetzungen erfüllt werden. Zum Schluss wird die Umsetzung durchgeführt.
Auch die Projektgruppe Qualitätssicherung der data experts gmbh (deg) hat diese Aufgaben und setzt sie manuell und automatisch für die Software **profil c/s**, die im Kapitel 2.1 genauer beschrieben wird, um. So wie jede Software muss **profil c/s** auch getestet werden, wobei zum einen die neuen Anpassungen, als auch die bisherigen Funktionalitäten, das heißt, die fachlichen und technischen Umsetzungen, auf Fehler untersucht werden müssen.
Aus dem vorher absolvierten Praktikum kristallisierte sich die Aufgabenstellung heraus. In der Bachelorarbeit wird untersucht, ob Sikuli zum Testen von GIS-Komponenten der Software **profil c/s** in der deg arbeitsunterstützend ist.

# Aufgabenstellung

Die Aufgabenstellung der Bachelorarbeit ist die Analyse der fachlichen und technischen Anforderungen für automatische Tests von GIS-Komponenten in **profil c/s** und die prototypische Umsetzung mit Sikuli.

In dieser Arbeit sollen als erstes die fachlichen und technischen Anforderungen für automatische Tests im GIS-Bereich analysiert werden. Zu Beginn wird die Software **profil c/s** der deg und die GIS-Komponente RFK-GIS-Editor beschrieben. Danach wird zum einen auf die fachlichen Anforderungen, wie zum Beispiel die Testinhalte und Testaufgaben, eingegangen. Anschließend werden die technischen Anforderungen für automatische Tests untersucht. Diesbezüglich werden die Anforderungen, die eine Software für das automatische Testen erfüllen sollte, spezifischer erläutert und definiert. Dabei darf nicht die Unterscheidung zwischen den fachlichen und geometrischen Daten vergessen werden und wie diese zu testen sind.
Als Zweites soll die prototypische Umsetzung mit dem Programm Sikuli erläutert und ausgeführt werden, das heißt, es wird zum einen beschrieben, wie Sikuli im Allgemeinen einsetzbar ist und zum anderen, wie Sikuli in der deg zum Einsatz kommen kann. Des Weiteren wird die Umsetzung von Testfällen des RFK-GIS-Editors mit Sikuli erläutert.
Als Ergebnis der Arbeit soll die Frage, inwiefern Sikuli für den Einsatz zum automatischen Testen von GIS-Komponenten in der deg geeignet ist, ausführlich beantwortet werden, was gleichzeitig auch das Ziel der Arbeit ist.
Ein Bestandteil der Arbeit ist die Vorstellung verschiedener Programme für das automatisierte Testen. Die Bewertungen und Kriterien einiger Softwares werden auf Quellen basierend analysiert, bewertet und verglichen, da nur bestimmte Softwares zum Einsatz kommen.

# 2 Testobjekt und -klassifikation

## 2.1 profil c/s

**profil c/s** steht für „**PRO**grammsystem zur rechnergestützten Verwaltung von **F**ördermitteln **I**n der **L**andwirtschaft“ [IV: 01] und wurde von der deg entwickelt. Es dient der Umsetzung von InVeKoS und angrenzenden Fördermaßnahmen in den Landwirtschaftsämtern, Landwirtschaftskammern oder zuständigen Senaten. **profil c/s** hilft bei der Verarbeitung der Anträge von landwirtschaftlichen Unternehmen auf Agrarförderungen und unterstützt dadurch die Zahlstellen und Behörden bei der Zusammenführung, Auswertung und zentralen beziehungsweise dezentralen Zahlungen der jeweiligen Fördermittel.

Dabei ist **profil c/s** die Zusammenfassung folgender Bestandteile. Die EU stellt Verordnungen auf und liefert somit die Vorgaben, vielmehr die fachlichen Richtlinien für die Fördermittelverwaltung. Die deg setzt diese Vorgaben in der Software **profil c/s** um und stellt diese den Landwirtschaftsministerien bereit. Die Ministerien nutzen die Software und stellen Wünsche bezüglich Änderungen und Anpassungen an die deg, die diese wieder einarbeitet. Somit ergeben sich folgende Funktionalitäten von **profil c/s**: Durchführung von Abgleichen, Auswertungen, Datenerfassung, Zahlungsverfahren, Kontrollen und die Anbindung von **profil c/s** an externe Schnittstellen.

Zu der Durchführung von Abgleichen, Auswertungen oder Kontrollen gehört auch die geografische Verwaltung der Flächen. **profil c/s** hat dafür das Referenzflächenkataster (RFK), indem die einzelnen Referenzflächen (RFL) mit dem RFK-GIS-Editor bearbeitet werden können. Eine detaillierte Beschreibung erfolgt in den Kapiteln 2.2 und 2.3.

Die nachfolgende Grafik veranschaulicht die Zusammenfügung der einzelnen Komponenten und die Einbindung des RFK mit dem RFK-GIS-Editor.

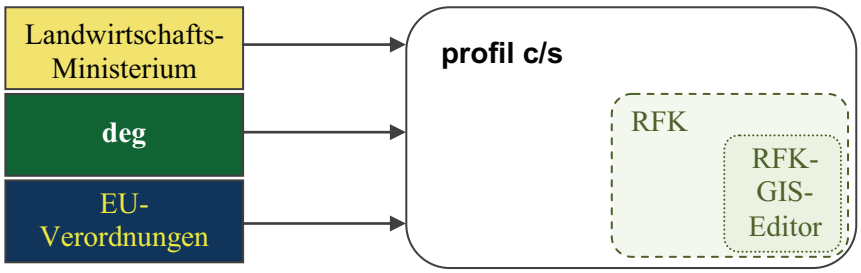



**Abb. 1: Zusammensetzung und Aufgaben von profil c/s**

## 2.2 Referenzflächen

Ein Teilbereich von **profil c/s** ist das RFK, welches die Aufgaben der Verwaltung und Pflege von RFL übernimmt. Dazu werden alle Informationen einer RFL digital in einer Mappe gespeichert und können über diese auch aufgerufen werden. Die nachfolgende Abbildung zeigt beispielhaft die Mappe einer RFL.

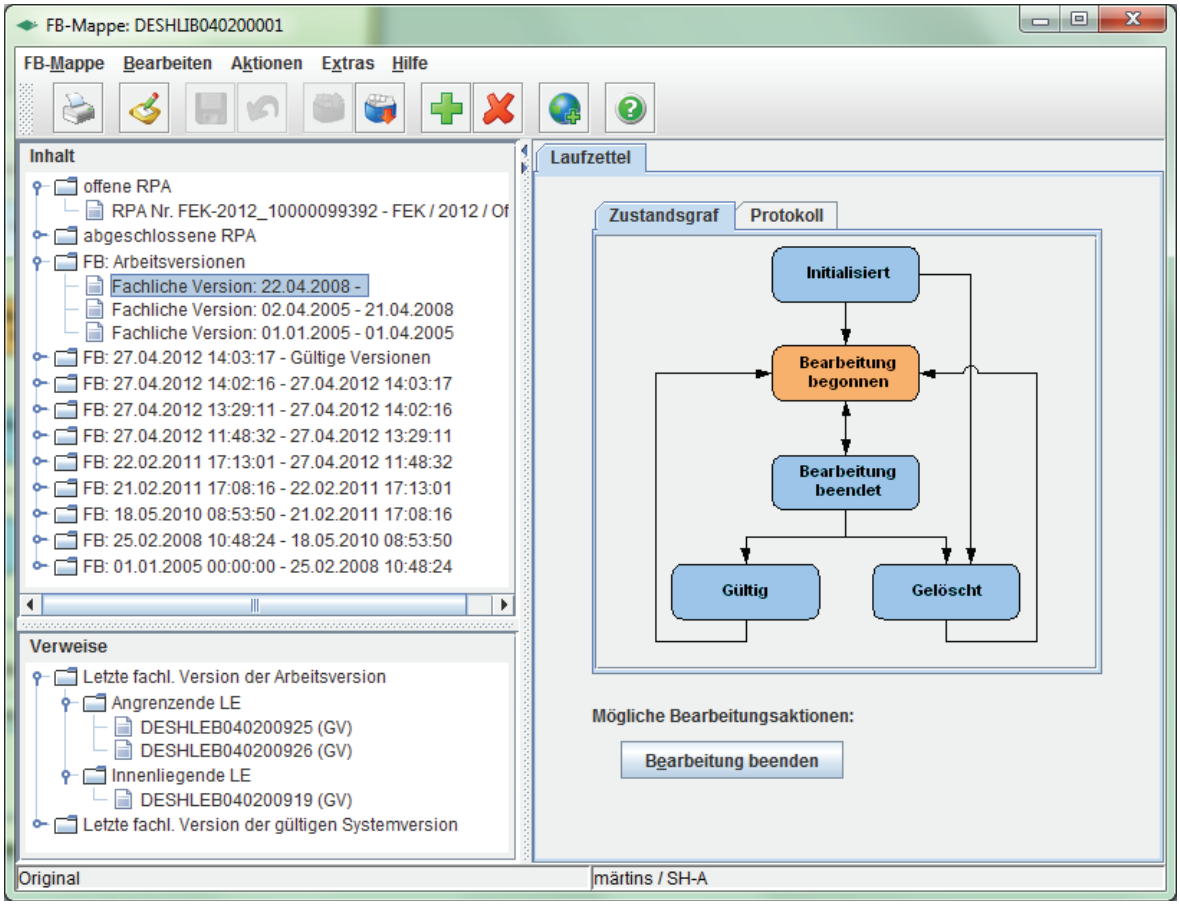



**Abb. 2: RFL-Mappe für die RFL DESHLIB040200001**

In der in Abbildung 2 dargestellten RFL-Mappe sind einzelne Versionen bezüglich der fachlichen Gültigkeit enthalten. Die Version mit dem aktuellsten Stand ist die, die noch kein Enddatum der Gültigkeit im Namen enthält (blau markierte Zeile). Des Weiteren werden Referenzflächenpflegeaufträge (RPA) aufgelistet, sofern welche vorhanden sind. Die Mappe kann verschiedene Zustände haben, die im Zustandsgraf angezeigt werden. Nur im Zustand „Bearbeitung begonnen“ können die Daten und somit auch die Geometrie der RFL bearbeitet werden. Wechselt man in diesen Zustand, wird eine Arbeitsversion erstellt, in

der die Änderungen vorgenommen werden können. Dazu öffnet man diese Version und es erscheint der RFL-Bearbeiter.

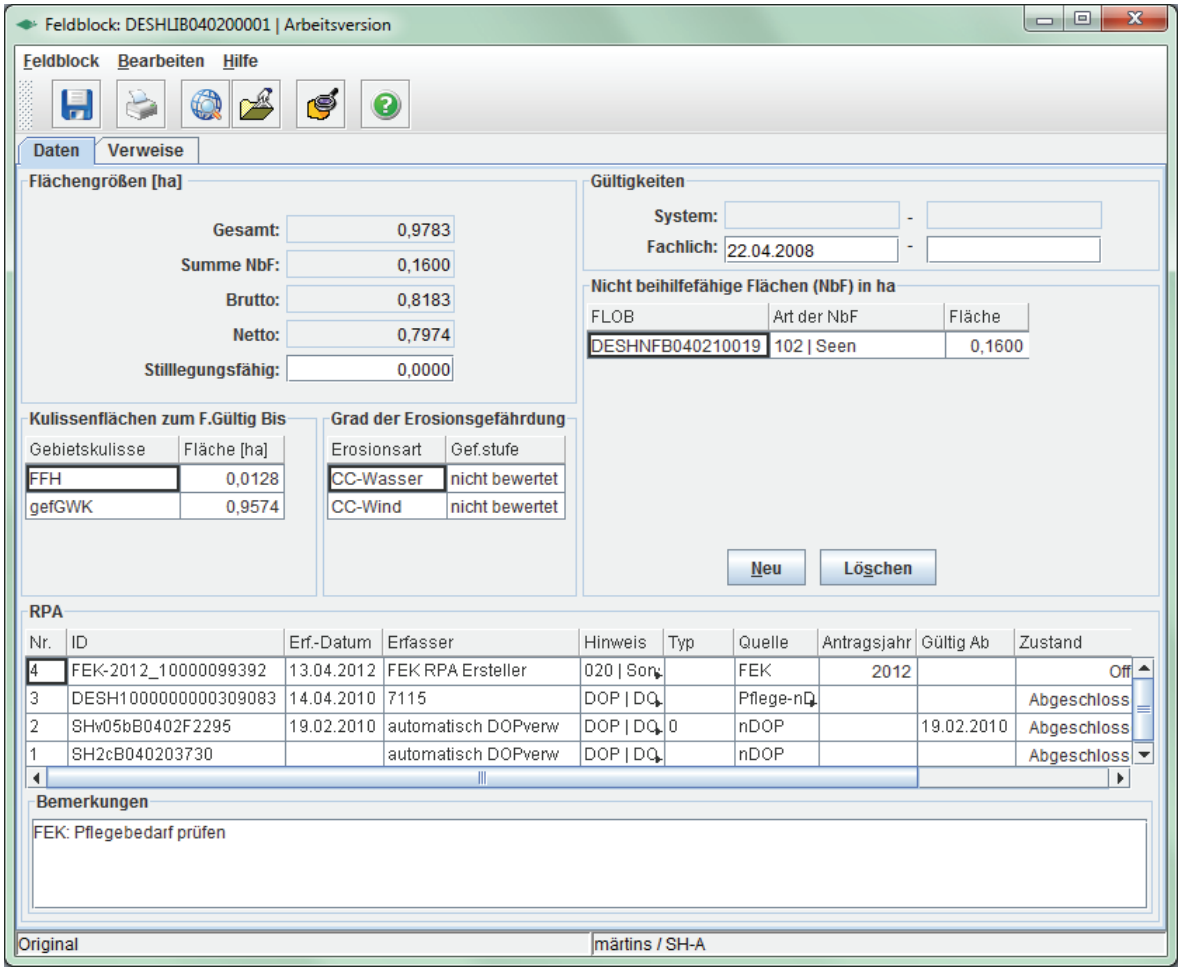



**Abb. 3: RFL-Bearbeiter für die RFL DESHLIB040200001**

In diesem Fenster, welches die vorherige Abbildung darstellt, werden alle Informationen bezüglich der Version der RFL angezeigt, wie zum Beispiel die Flächengröße, Informationen über einen RPA, über nicht beihilfefähige Flächen (NbF) oder den Grad der Erosionsgefährdung. Vom RFL-Bearbeiter kommt man über einen Button in der Toolbar zum RFK-GIS-Editor, der im Kapitel 2.3 beschrieben wird.

## 2.3 RFK-GIS-Editor

Der RFK-GIS-Editor ist ein Tool von **profil c/s** im Teilbereich RFK und dient dazu, RFL durch Orthofotos und Vektorgeometrien, für die Agrar-Fördermittelverwaltung, zu bearbeiten, das heißt überprüfen, pflegen, messen, beschneiden, verschmelzen, teilen. Um sich die RFL im RFK-GIS-Editor anzeigen zu lassen, klickt man im RFL-Bearbeiter auf den entsprechenden Button in der Toolbar. Anschließend öffnet sich das Fenster des RFK-GIS-Editors, der in Abbildung 4 dargestellt ist.

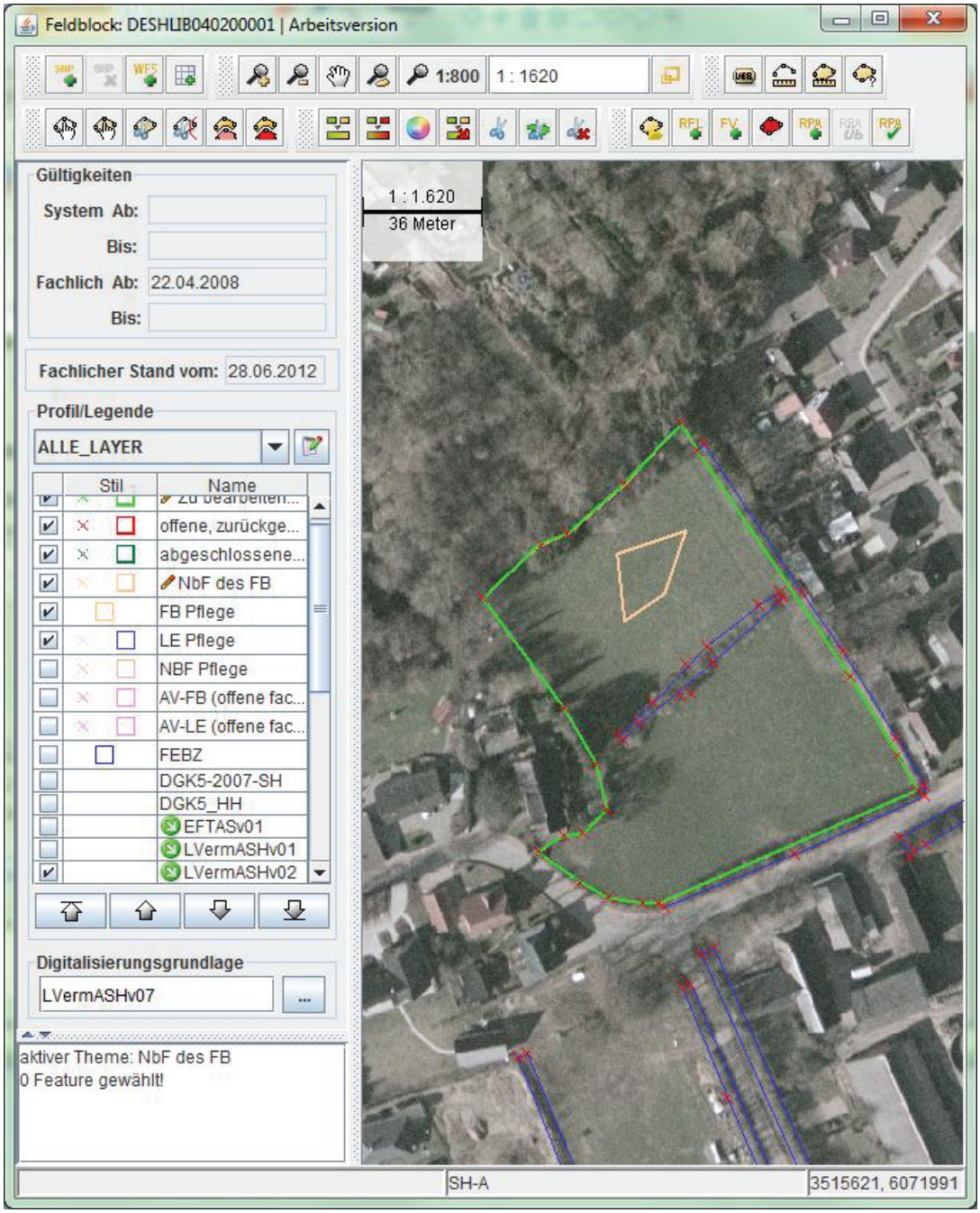



**Abb. 4: RFK-GIS-Editor mit der RFL DESHLIB040200001**

In der Legende des RFK-GIS-Editors (siehe Abb. 4 (S. 6), Profil/Legende) werden die Layer aufgelistet und lassen sich darüber ein- und ausblenden. In der Menüleiste sind verschiedene Aktionen auswählbar, diesbezüglich ist das Menü in sechs Symbolleisten unterteilt.

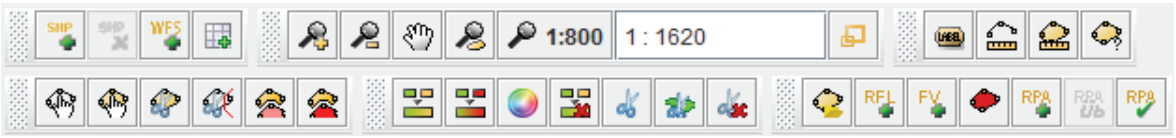
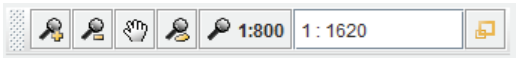
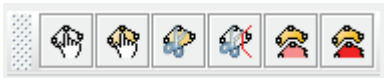
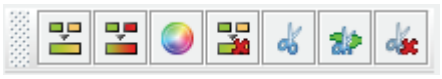



**Abb. 5: Toolbar mit sechs Symbolleisten des RFK-GIS-Editors**

Die nachfolgende Tabelle listet die Funktionalität gegliedert in die einzelnen Symbolleisten auf (bezüglich Abb. 5, von links beginnend).

**Tab. 1: Funktionalität der Symbolleisten im RFK-GIS-Editor**

| **Symbolleiste** | **Funktionalität** |
|---|---|
| Layer  | Shapefile-Layer hinzufügen und löschen, WFS und WMS Layer hinzufügen |
| Navigation  | Zoomen, Kartenansicht verschieben, Maßstab definieren |
| Information  | Ident bzw. Label anzeigen, Strecken und Flächen messen, Geometrie auswählen |
| Editierung  | Bearbeitung der Geometrie: Punkt oder Geometrie verschieben, Geometrie beschneiden (automatisch, manuell oder mit einem Polygon) |
| Funktionen  | RFL verschmelzen oder aufteilen, Aufteilen abbrechen, Geometrie vereinigen |
| Aktionen mit anderen RFL  | RFL oder RPA erstellen, RPA übernehmen oder abschließen, Mappe einer anderen RFL bzw. Kulisse öffnen oder löschen |

## 2.4 Testklassifikation

Die Aufgabe eines Tests in der Informatik ist die „Überprüfung eines Programms oder eines Softwaresystems auf Funktionsfähigkeit“ [IV: 05]. Der Zweck des Testens ist es, Fehler im System aufzuspüren und nicht die Korrektheit nachzuweisen, was man Programmverifikation nennt.

Zu Beginn des Testens sollten auch die Ziele des Tests klar strukturiert sein, um die Testbeschreibung, -methoden und den -durchlauf daraufhin anzupassen. Dabei kann man sich an die Anforderungen an eine Software orientieren. So ist zum Beispiel ein Ziel die Stabilität der Software, das bedeutet, dass „kein bekannter Testfall eine unvorhergesehene Fehlermeldung oder gar einen Absturz provozieren darf“ [IV: 07]. Ein weiteres Ziel ist die Korrektheit, also die Einhaltung der Anforderungen an die Software [Vergleiche IV: 07].

Nachdem das Ziel des Testes definiert ist, wird eine Entscheidung für eine von drei möglichen Testmethoden getroffen. Zum einen gibt es die Black-Box-Testmethode, bei der der Tester nur das Ergebnis seiner Aktionen, aber nicht die internen Strukturen der Software kennt. Diese funktionalen Tests „erfordern eine genaue Spezifikation des zu testenden Objektes“ [IV: 22], weil mit ihnen nur auf der Kenntnis des Verhaltens und nicht des Aufbaus der Software getestet wird. Zum anderen gibt es die White-Box-Testmethode, die das Gegenteil der ersten Testmethode ist. Das heißt, der Tester kennt die Struktur der Software und kann davon die Testfälle ableiten, um gezielt Schwachpunkte zu testen. Eine Kombination aus beiden Methoden ist der Gray-Box-Test. Dafür werden Testfälle für die Anforderungen der Software auf Grundlage des Quellcodes erstellt. [Vergleiche: IV: 22 und LV: 03]

Des Weiteren gibt es zwei Testvarianten: die manuelle und die automatisierte Durchführung eines Tests. Manuelle Durchführung bedeutet, der Entwickler beschreibt einen Testablauf und der Testingenieur führt diesen händisch durch und untersucht ihn auf Fehler. Bei der automatischen Durchführung erstellt der Testingenieur bestimmte Abläufe eines Testfalles in einer Testsoftware, startet diese und die Testsoftware führt die Aktionen des Ablaufes automatisch aus.

Um einen Testfall, unabhängig vom manuellen oder automatisierten Testen, auszuführen, sollte für diesen eine Testumgebung vor dem Durchlauf vorbereitet und nach einem Testdurchlauf bereinigt werden. Der Grund für die Bereinigung der Testumgebung liegt einerseits darin, dass ein Fehler in einem Testfall nicht den Durchlauf von weiteren Testfällen

beeinträchtigen soll und andererseits nach dem Testfall die veränderten Daten wieder hergestellt werden und somit das System und einen erneuten Durchlauf des Testfalles nicht beeinflussen, oder gar verhindern.

Tests können, wie bereits beschrieben, manuell und automatisch durchgeführt werden. Bezüglich des Themas der Arbeit wird für die weitere Betrachtung nur auf die Testautomatisierung eingegangen. Dabei wird in der Arbeit zwischen automatischen Tests und Testwerkzeugen unterschieden. Zu den automatischen Tests gehören Unit-Tests. Diese werden nur bezüglich abgeschlossenen Programmcodes ausgeführt. Um diese im weiteren Verlauf der Arbeit von automatischen Tests, die keine Unit-Tests sind, zu unterscheiden, werden sie mit „UT“ abgekürzt. Testwerkzeuge sind Programme, mit denen automatisierte Testfälle erstellt und ausgeführt werden. Der Ablauf eines Testfalles wird vom Testwerkzeug in Skripten gespeichert. Da mit Testwerkzeugen ebenfalls automatische Tests, aber nicht zwangsläufig Unit-Tests, erstellt werden können, erfolgt die Unterscheidung anhand der Abkürzung „AT“.

Die nun folgende Abbildung fasst die genannten Anforderungen an einen Test zusammen.

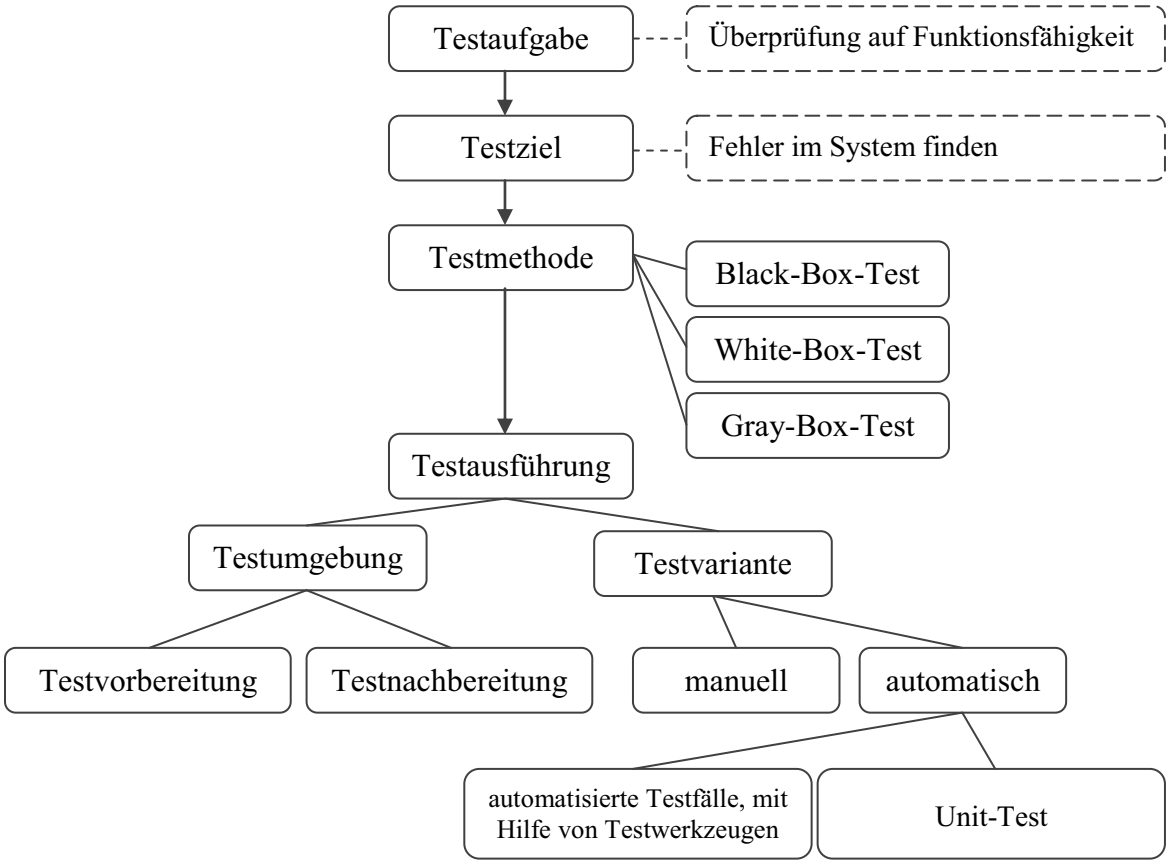



**Abb. 6: Anforderungen an einen Test**

In der deg werden Tests manuell und automatisch vollzogen. Bei der Testautomatisierung handelt es sich um sogenannte Regressionstests, das heißt, es werden bestimmte Testfälle wiederholt ausgeführt, um die Fehlerfreiheit grundlegender Funktionen nach Änderungen zu gewährleisten. Dabei handelt es sich immer um Black-Box-Tests, da der Tester der Projektgruppe Qualitätssicherung die Software nur in Bezug auf die Anforderungen überprüft und die interne Struktur nicht kennt. Bei der Testvorbereitung für diese Tests wird auch die Testumgebung erstellt. Diesbezüglich müssen unter anderem die Rechte des testenden Nutzers und die Testdaten dem Testfall entsprechend angepasst werden. In der deg müssen manchmal auch Skripte als Testvorbereitung auf der Kommandoebene ausgeführt werden. Diese Skripte sind als Shell-Dateien gespeichert und dienen dazu, spezielle Java-Programme auszuführen, beziehungsweise Migrationen durchzuführen. Die Bereinigung der Testumgebung erfolgt in der deg bei AT mit QF-Test und UT mit JUnit. Bei manuellen Tests entfällt dies in der Regel aufgrund von Zeitersparnis oder weil es fachlich nicht möglich ist.
Auf die genaue Umsetzung der Testautomatisierung wird im nächsten Kapitel eingegangen.

# 3 Anforderungsbetrachtung

## 3.1 Analyse der fachlichen Anforderungen

Bei der Testautomatisierung für **profil c/s** und somit auch für den RFK-GIS-Editor sind alle Funktionalitäten und Daten zu testen. Dabei sind im GIS-Bereich nicht nur die Mappen und die darin angezeigten Informationen und Berechnungen zu überprüfen oder im RFL-Bearbeiter, die Berechnung der Flächengröße, sondern vor allem die Umsetzung des Testfalles im Kartenausschnitt muss getestet werden. Konkret für den RFK-GIS-Editor müssen Funktionalitäten, Sachdaten und besonders die Geometriedaten überprüft werden. Zu den Funktionalitäten gehört zum Beispiel das Zerschneiden, Teilen, Verschmelzen und Verschieben. Die Sachdaten der RFL sind beispielsweise die Gültigkeit, Zustandswechsel der RFL-Mappe oder die RFL-Art. Geometriedaten sind unter anderem die Anzahl und Position der Stützpunkte, geometrische Form und vor allem die Darstellung des Kartenausschnittes. Diese Daten sind bei der Testautomatisierung zu kontrollieren.
UT erfolgen in der deg via JUnit-Tests. Für das Schnitt-Tool des RFK-GIS-Editors wurde so ein Test bereits erstellt. Der Test prüft die Zerschneidung der Geometrie anhand des Polygons. Bei der Zerschneidung müssen nur die entscheidenden Geometrien beschnitten werden und die nicht betroffenen unverändert bleiben. Dies wird mittels der Flächengröße der Geometrien, die aus Koordinaten berechnet werden, kontrolliert. Ob die Fläche nach dem Zerschneiden auch entsprechend dargestellt wird, lässt sich darüber aber nicht prüfen. Auf eine andere Art der Unit-Tests wird unter dem Kapitel 3.2.2 gesondert Bezug genommen.
Die AT werden in der deg mit Hilfe des Testwerkzeuges QF-Test erstellt und ausgeführt, allerdings noch nicht für den GIS-Bereich. Das Testwerkzeug wird unter dem Kapitel 4.1.1 näher erläutert. Mit QF-Test können die Funktionalitäten und die Daten getestet werden. Ob QF-Test auch für die Überprüfung des Kartenausschnittes geeignet ist, wird in Kapitel 3.2.2 analysiert.
Die folgenden Tabellen (S. 12) sollen zeigen, welche Testfälle in Bezug auf die genannte Problematik relevant sind, welche davon bereits automatisch und welche bisher nur manuell getestet werden. Kategorisiert wurden die Testfälle nach ihrem Testinhalt, wobei dies bei allen nicht explizit möglich ist, da sie mehrere Testinhalte aufweisen.

Es gilt folgende Symbolik:

**Legende**
√ wird getestet X wird nicht getestet

**Tab. 2: relevante Testfälle mit geometrischem Testinhalt**

| Testfall | Testobjekt | manuell | JUnit-Test | QF-Test |
|---|---|---|---|---|
| Anzeigen von Labels | RFK-GIS-Editor | √ | X | X |
| Grafische Kontrolle der bearbeiteten Geometrie | RFK-GIS-Editor | √ | X | X |
| Grafische Kontrolle des Kartenausschnittes | RFK-GIS-Editor | √ | X | X |
| Feature[1] selektieren | RFK-GIS-Editor | √ | X | X |
| Stil eines Layers ändern | RFK-GIS-Editor | √ | X | X |
| Rasterlayer anzeigen | RFK-GIS-Editor | √ | X | X |
| Feature verschieben | RFK-GIS-Editor | √ | X | X |

**Tab. 3: relevante Testfälle mit fachlichem Testinhalt**

| Testfall | Testobjekt | manuell | JUnit-Test | QF-Test |
|---|---|---|---|---|
| Speichern der RFL-Mappe | RFL-Mappe, RFL-Bearbeiter | √ | X | √ |
| Zustandswechsel der RFL-Mappe | RFL-Mappe | √ | X | √ |
| Verweise in der RFL-Mappe | RFL-Mappe | √ | X | X |
| Datum der Gültigkeit | RFL-Bearbeiter | √ | X | X |
| Erstellung eines NbF | RFL-Bearbeiter | √ | X | √ |
| Berechnung der Flächengröße | RFL-Bearbeiter | √ | X | X |
| Öffnen einer RFL-Mappe aus dem GIS-Editor | RFK-GIS-Editor | √ | X | √ |
| Löschen einer RFL-Mappe aus dem GIS-Editor | RFK-GIS-Editor | √ | X | √ |
| RFL erstellen | RFK-GIS-Editor | √ | X | X |

**Tab. 4: relevante Testfälle mit funktionalem u. geometrischem Testinhalt**

| Testfall | Testobjekt | manuell | JUnit-Test | QF-Test |
|---|---|---|---|---|
| RFL mittels Polygon zerschneiden | RFK-GIS-Editor | √ | √ | X |
| RFL teilen | RFK-GIS-Editor | √ | X | X |
| RFL verschmelzen | RFK-GIS-Editor | √ | X | X |
| Polygon verschieben | RFK-GIS-Editor | √ | X | X |

Allgemein lässt sich sagen, dass die Testautomatisierungen mit geometrischem und funktionalem Testinhalt bisher noch nicht umgesetzt worden sind. Der Grund dafür liegt in der Prüfung der grafischen Oberfläche (GUI), konkret der Kartenanzeige im RFK-GIS-Editor.

[1] Geometrie einer Fläche, die in einer Karte angezeigt wird

Diese Komponente kann bisher nur manuell getestet werden, da die eingesetzte Testsoftware QF-Test in der deg die Features in der Karte nicht als solche erkennt, sondern nur die Karte als Ganzes sieht. Zwar kann die Testsoftware ein Abbildvergleich durchführen, allerdings lässt sich die Fehlertoleranz nicht definieren. Speziell für AT mit geometrischem Testinhalt sind grafische Checks (GUI-Check[2]) wichtig. Damit wird automatisiert überprüft, dass zum Beispiel bei der Erstellung oder der Bearbeitung einer RFL keine Fehler auftreten und die Fläche letztendlich so bearbeitet und angezeigt, wie es erwartet wird.
Den Durchlauf eines Testfalles mit QF-Test kann man sich schematisch, wie in der nächsten Abbildung dargestellt, vorstellen. Zu Beginn erfolgt die Testvorbereitung, die bereits einen Kartentest, also einen GUI-Check bezüglich des Kartenausschnittes, beinhaltet. Wurde die Testumgebung durch die Testvorbereitung erstellt, kann nun der Testfall ausgeführt werden, der ebenfalls einen Kartentest enthält. Sind beim Durchlauf des Testfalles keine Fehler aufgetreten, erfolgt anschließend die Testnachbereitung. Wie bereits erwähnt, stellt der Test des Kartenausschnittes das Problem dar.

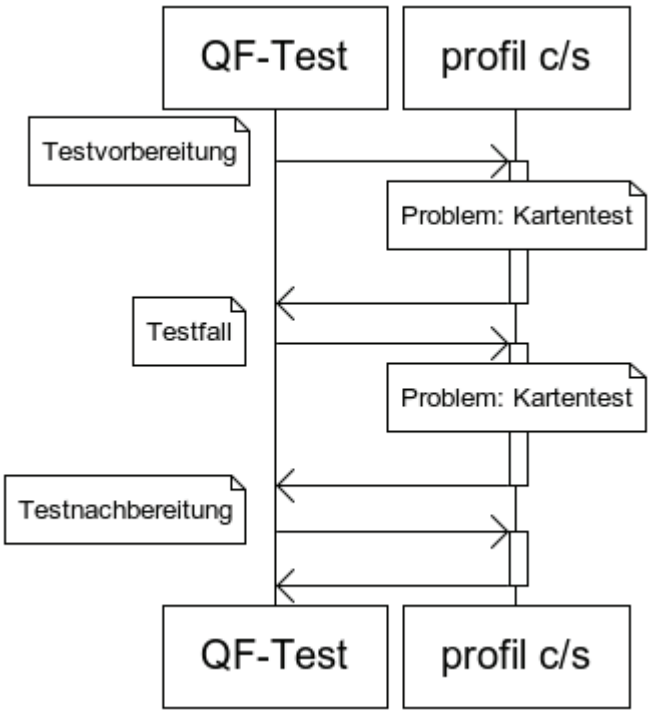



**Abb. 7: Testdurchlauf des RFK-GIS-Editors mit QF-Test**

[2] Bildvergleich zwischen dem, was angezeigt und dem was erwartet wird

Ein Testfall für den RFK-GIS-Editor könnte zum Beispiel so aussehen: zu Beginn wird eine Testumgebung erzeugt. Diese Testumgebung enthält eine neue, erstellte RFL, die mit Hilfe eines Kartentests überprüft wird. Die genaue Umsetzung der Testvorbereitung wird im Kapitel 4.2 beschrieben.
Anschließend beginnt der eigentliche Testfall, indem die RFL verändert, zum Beispiel mit einer anderen RFL verschmolzen wird. Ist keine RFL für den Testfall vorhanden, muss, wie Kapitel 4.2 beschrieben, eine erstellt werden. Bei der Verschmelzung gibt es mehrere Aspekte zu testen. Die zu verschmelzenden RFL müssen mindestens einen gemeinsamen Punkt, eine gemeinsame Polygonlinie oder Teilfläche haben. Eine RFL muss dabei im Zustand „In Bearbeitung“ und die andere im Zustand „Gültig“ sein. Das Verschmelzen geht nur in der Ansicht des RFK-GIS-Editors der Mappe, die im Bearbeitungs-Zustand ist. Dadurch ist der Button aktiv und die Verschmelzung kann erfolgen. Nach dem Verschmelzen muss die vorher gültige Mappe im Zustand „Gelöscht“ sein und beide Geometrien als ein Feature anzeigt werden. Dies könnte man über einen GUI-Check testen. Des Weiteren darf sich nach der Verschmelzung der Zustand der bearbeiteten Mappe nicht verändert haben und das Protokoll beider Mappen muss mit einem entsprechenden Eintrag aktualisiert worden sein. Auch die Beziehungen der RFL untereinander, das heißt, die Angrenzungen oder Umschließungen von RFL, müssen neu überprüft werden.
Zum Schluss, als Testnachbereitung, wird die Mappe gespeichert und in den Zustand „Gelöscht“ gesetzt, da in diesem Zustand die Mappe im RFK-GIS-Editor standardisiert nicht angezeigt wird. Auch dieser Vorgang muss im Einzelnen getestet werden. Konkret bedeutet das, dass der Speicher-Button inaktiv werden muss und die Daten für den nächsten Aufruf verfügbar sein müssen.

## 3.2. Analyse der technischen Anforderungen

### 3.2.1 Hardwaretechnische Anforderungen

Zu den hardwaretechnischen Anforderungen gehört, bezüglich der Problemstellung, die Auflösung und somit die Leistungsfähigkeit der Grafikkarte und des Bildschirmes. Da aber UT davon unabhängig funktionieren und AT in der deg über eine virtuelle Maschine ausgeführt werden, ist diese Anforderung ohne Bedeutung.
Allgemein spielt in der heutigen Zeit der Softwareentwicklung die Hardware nur eine sehr geringe Rolle, weil die Hardwareentwicklung schneller voranschreitet als die der Software. Da bei der Verwendung von Testsoftwares der Nutzer durch die Hardware nicht eingeschränkt wird, können diese Anforderungen für die weitere Arbeit vernachlässigt werden.

### 3.2.2 Softwaretechnische Anforderungen

Ein Bereich der softwaretechnischen Anforderungen ist der Umgang mit Ereignissen. Da bei AT ein Ablauf erstellt und dieser durchlaufen wird, muss auch darauf geachtet werden, dass man unverhoffte Ereignisse berücksichtigt. Zudem können auch Probleme bei der Synchronisation von Ereignissen auftreten, da die Antwortzeiten, zum Beispiel von Datenbanken und Applikationsservern, nicht immer konstant sind. [Vergleiche IV: 08]
Prinzipiell gibt es zwei Testkonzepte um GUI zu testen: Capture/Replay[3]-Verfahren und GUI-Unit-Tests.
Mit dem Capture/Replay-Verfahren können spezielle Funktionen einer GUI getestet werden, indem die gewünschten Aktionen aufgenommen und hinterher wieder abgespielt und damit wiederholt werden können. Bei diesem Testkonzept sind ausschließlich die auszuführenden Aktionen relevant. Dafür ist ein Werkzeug notwendig, welches diese Benutzeraktionen, wie zum Beispiel Tastatureingaben oder Mausklicks, erfasst und die Testabläufe speichert. Dies geschieht in Form von Skripten [Vergleiche IV: 07, 08]. Meistens ist es aber nicht ausreichend, nur den gewünschten Ablauf aufzunehmen. Man muss die Testknoten, die jeweils eine Benutzeraktion beinhalten, anpassen, um einen Testfall allgemeingültig zu halten.

[3] dt. = Aufnehmen/Abspielen

Die GUI-Unit-Tests basieren auf den bekannten UT, unterteilen die GUI aber in kleine Einheiten und testen diese systematisch. Dabei werden nur auf der Objektebene die Tests durchgeführt, was zur Folge hat, dass bestimmte Korrektheitskriterien, wie zum Beispiel die Überlappung zweier Fenster, nicht geprüft werden können. Dadurch, dass GUI-Tests nur die Komponenten identifizieren, aber nicht die Sicht des Nutzers erkennen können, sind GUI-Unit-Tests für das Testen der Karten-Komponente im RFK-GIS-Editor nicht geeignet, weil der Kartenausschnitt, inklusive dargestellter Feature, nur als eine Komponente erkannt wird. Deswegen wird dieses Testkonzept nicht weiter erläutert.
Bei der Definierung der Kriterien der Testsoftware wird sich in der Arbeit auf die wesentlichen Punkte beschränkt, die für den Test des Kartenausschnittes des RFK-GIS-Editors relevant sind.
Mit der Auswahl des Testkonzeptes auf das Capture/Replay-Verfahren steht ein Merkmal fest. Die erstellten Testfälle sollten durch Skripte, das Tool durch Plug-ins oder Ähnlichem erweiterbar sein. Zusätzlich ist empfehlenswert, wenn die Testsoftware bei der Testvorbereitung eine unterstützende Funktion übernimmt, indem die Software die Vorbereitung zum Beispiel in separaten Prozeduren speichert, die von beliebigen Testfällen zu Beginn aufgerufen werden können.
Ein wichtiges Kriterium ist zum einen die Integration des Testwerkzeuges zu der zu testenden Software. Dabei muss das Werkzeug mit der zu testenden Software kooperieren und sollte unter anderem auch über den Batch-Modus ausgeführt werden können. Das hat den Vorteil, dass keine GUI des Testwerkzeuges gestartet wird und somit in der Regel die Performanz steigt, weil keine Bildschirmbedienung erforderlich ist und Hardwareressourcen zur Verfügung stehen. Das entscheidende Kriterium ist aber der bereits in Kapitel 3.1 erwähnte GUI-Check. Die Testsoftware muss einen Bildvergleich durchführen oder anderweitig geometrische Flächen und Strukturen erkennen und verarbeiten können. Zum Bildvergleich gehört auch eine Bildaufnahmefunktion, um das erwartete Bild zu definieren. Ebenso sollte das Testwerkzeug transparente Bilder, beziehungsweise Bilder mit transparenten Bereichen, verarbeiten und die durchsichtigen Bereiche als solche interpretieren.
Zusätzlich optionale Anforderungen an die Testsoftware wären zum Beispiel, dass man die erstellten Tests gliedern und somit strukturieren oder wiederholende Abfolgen in Prozeduren oder Methoden auslagern kann. Ein weiteres Beispiel ist die Plattformunabhängigkeit, um den Anwender nicht auf ein spezielles Betriebssystem einzuschränken. Da aller

dings die Durchführung von AT in der deg nur auf einem Betriebssystem ausgeführt werden, ist dieses Kriterium für die weitere Betrachtung von geringer Bedeutung. Weiterhin sollte das Werkzeug eine Protokollierung für den Durchlauf des Testfalles anbieten, in denen auch Fehler beim Durchlauf ersichtlich und nachvollziehbar sind. Da QF-Test weiterhin genutzt werden soll und eine sehr gute Protokollierung anbietet, die auch in der deg genutzt wird, ist dieses Kriterium ebenfalls irrelevant.
Zusammengefasst ergibt sich mit der nachfolgenden Tabelle eine Anforderungsliste für die Bewertung einer Testsoftware für automatische Tests im GIS-Bereich.

**Tab. 5: Anforderungen an eine Testsoftware für die Testautomatisierung im GIS-Bereich**

| Anforderung | Gewichtung |
|---|---|
| Capture/Replay Verfahren | 1 |
| Integration zu der zu testenden Software | 2 |
| Bildvergleich | 4 |
| Erkennung geometrischer Formen | 3 |
| Aufnahmemöglichkeit der Bilder | 3 |
| Transparenz beim Bildvergleich | 3 |
| Strukturierung der Skripte | 1 |

| Legende |
|---|
| 1 = grundlegend |
| 4 = unverzichtbar |

# 4 Softwarebetrachtung für Testautomatisierung im GIS-Bereich

## 4.1 Analyse der Testsoftwares

In diesem Abschnitt sollen fünf verschiedene Testsoftwares auf die in Tabelle 5 (S. 17) genannten Kriterien untersucht und bewertet werden. Durch eine Rücksprache mit Testingenieuren anderer Softwarefirmen, die Programme mit GIS-Anwendungen anbieten, fiel die Entscheidung auf folgende Testwerkzeuge: JMeter, Selenium, Canoo WebTest, QF-Test und Sikuli.

JMeter dient dazu, Anfragen an webbasierte Systeme zu stellen und diese auszuwerten. Eine kurze Beschreibung des Werkzeuges befindet sich im Anhang[1]. Die Testsoftware ist in Java geschrieben und somit vom Betriebssystem unabhängig. Auf JMeter wird allerdings nicht weiter eingegangen, da die Testsoftware die entscheidenden Kriterien, wie beispielsweise die Erkennung geometrischer Formen oder Integration zu der zu testenden Software, nicht erfüllt.

Selenium testet Webanwendungen automatisiert. Dafür wird es als Add-on in den Browser Firefox installiert und ausgeführt. Es gibt vier Projekte von Selenium die im Anhang[2] konkreter erläutert werden. Die erstellten Testabläufe werden in einer programmeigenen Sprache geschrieben, die sich Selenese nennt und auf HTML aufbaut. Das Open-Source-Tool kann aber nicht auf den Swing-Client zugreifen und keinen Bildvergleich durchführen und wird daher nicht weiter erläutert.

Canoo WebTest ist ebenfalls ein Werkzeug zur Testautomatisierung von Webanwendungen, ähnlich wie Selenium, und ist ausführlich im Anhang[3] beschrieben. Es ist auch Open Source und plattformunabhängig. Ein Unterschied zu Selenium ist, dass die Skripte der Testabläufe in XML gespeichert werden, jedoch kann auch Canoo WebTest keinen Bildvergleich realisieren oder geometrische Formen erkennen und entfällt für die weitere Betrachtung.

[1] Anhang B.1, Seite XV
[2] Anhang B.2, Seite XVI
[3] Anhang B.3, Seite XIII

Somit wird QF-Test, als Testsoftware, die aktuell in der deg eingesetzt wird, und Sikuli, als mögliche Erweiterung zur bisherigen Testautomatisierung in der deg im nächsten Kapitel vorgestellt und erläutert.

### 4.1.1 QF-Test 3.2.2

QF-Test steht für Quality First-Test und wird aktuell in der Version 3.2.2 in der deg für die Erstellung und Durchführung von AT eingesetzt. Das Testwerkzeug ist kostenpflichtig. Dafür kann man ein Support mit Dokumentation, Tutorial und Forum in Anspruch nehmen. Regelmäßig werden aktuelle Versionen zur Verfügung gestellt.
Mittels Capture/Replay-Verfahren können Testfälle zügig aufgenommen und abgespielt werden. Dabei erkennt QF-Test die Komponenten und speichert diese in der Testsuite, die die Gesamtheit der zu einer Thematik gehörenden Testfälle darstellt und zusammenfasst.
Durch die Benutzeroberfläche des Testwerkzeuges ist die Bedienung und Funktionsweise verständlich. QF-Test startet die zu testende Software (SUT)[4] als Client, erkennt die Komponenten der SUT und die dazugehörigen Methoden. Dadurch lassen sich Events, wie zum Beispiel Mausklicks, für entsprechende Komponenten präzise definieren. Ändert sich der Name der Komponente in der SUT, so kann dieser einfach im Testfall angepasst werden. Durch Skripte, die in den Sprachen Jython oder Groovy geschrieben werden können, lassen sich spezielle Abfragen realisieren, die über angebotene Aktionen des Werkzeuges nicht möglich sind. Ein Beispiel dafür ist die Anwendung von Methoden bezüglich einer Komponente, um bestimmte Parameter auszulesen.
Infolge der Strukturierung einzelner Aktionen in Testknoten und Zusammenfassen dieser in Testschritten, Testfällen und Testfallsätzen lässt sich die Abfolge sehr gut nachvollziehen. Für die gute Strukturierung und Erweiterbarkeit eines Testfalles stehen die Auslagerung von Prozeduren und Komponenten sowie die Definierung von Testabhängigkeiten. Durch die Abhängigkeiten können vor und nach jedem Testfall bestimmte Aktionen ausgeführt werden. Diese dienen der Testvor- und -nachbereitung.
Beim Durchlauf eines Testes erstellt QF-Test automatisch ein Protokoll, welches die Abfolge der einzelnen Testknoten wie beim Durchlauf auflistet, was für eine gute Lesbarkeit sorgt. Infolgedessen ist die Fehlersuche in einem erstellten Testfall umso leichter. Für

[4] In QF-Test wird die zu testende Software SUT (System under Test, dt. = System unter Test) genannt

die Fehleranalyse können Breakpoints[5] gesetzt und der Durchlauf von beliebigen Testknoten aus gestartet werden.

QF-Test ist plattform- und versionsunabhängig und somit flexibel im Einsatz.

In der nachfolgenden Grafik ist die Benutzeroberfläche von QF-Test, mit einer Testsuite als Beispiel, zu sehen.

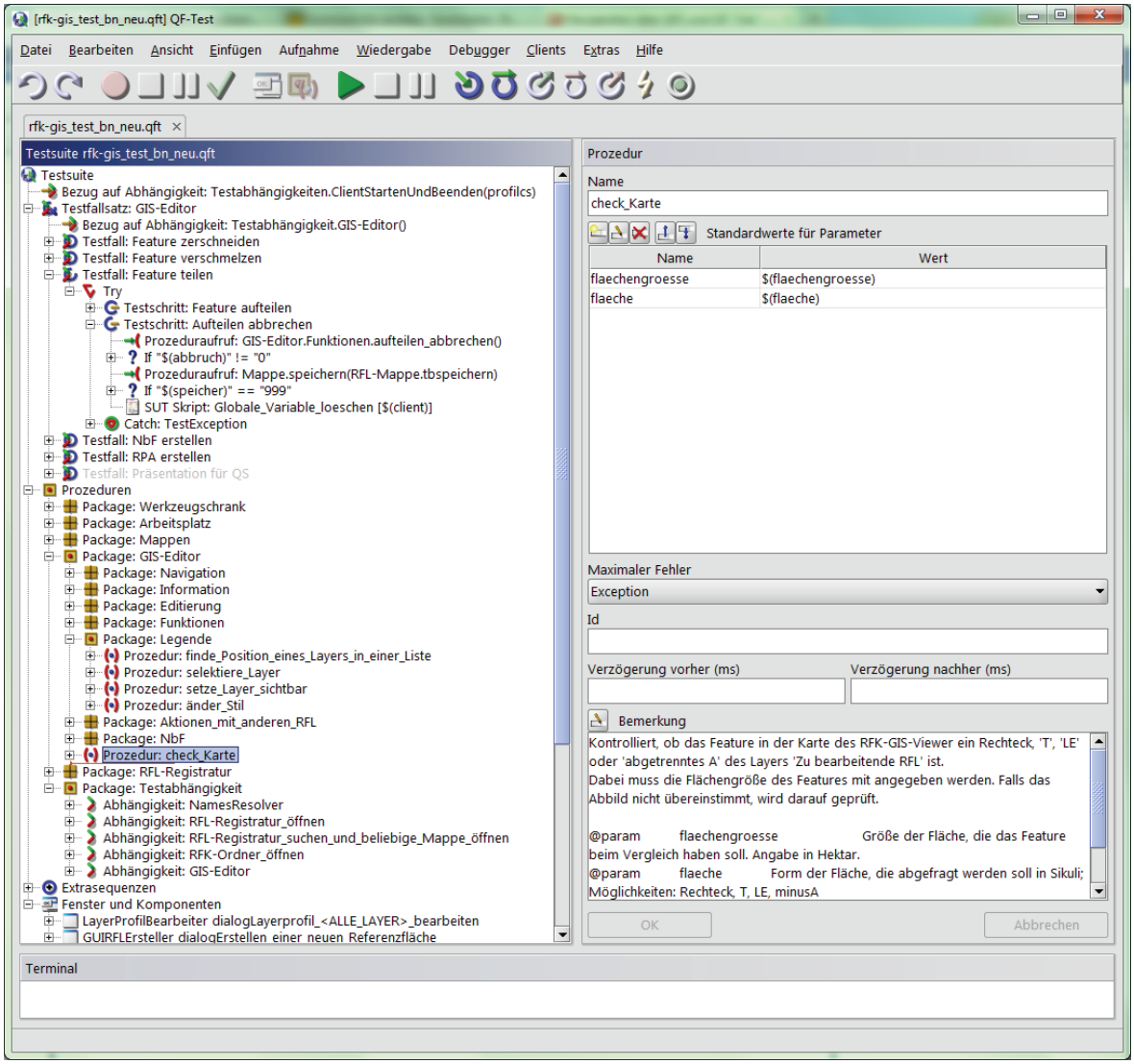



**Abb. 8: GUI von QF-Test 3.2.2**

### 4.1.2 Sikuli X r930

Das zweite Testwerkzeug, welches analysiert wird, ist das Open-Source-Tool Sikuli[6]. Der Name Sikuli kommt aus Mexiko, wo die Huicholen leben, die ein religiöses und kulturelles

[5] Markierungspunkt im Skript, an dem der Durchlauf pausiert und auf eine Aktion des Testers wartet
[6] Benutzeranleitung auf der CD, Anhang C, Seite XIX

Objekt namens „God's Eye“[7] haben. Die Huicholen nennen es selbst Sikuli, was für die Kraft zum Sehen und Verstehen von unbekannten Dingen steht. Sikuli setzt diese Thematik durch das Automatisieren von Abläufen, basierend auf Screenshots, also das, was man sieht, um.

Auch für dieses Testwerkzeug gibt es eine ausführliche Dokumentation, Tutorials und ein sehr lebendiges Forum. Auch wenn Sikuli nicht ganz so oft aktualisiert wird wie QF-Test, so gibt es doch sehr stabile Versionen für die Betriebssysteme Windows, Linux und Mac OS.

Die Handhabung ist nachvollziehbar. Man erstellt ein Skript, indem man die Abfolge von Aktionen beschreibt. Dazu gibt es von Sikuli vorgegebene Aktionen, die eingesetzt werden können. Das folgende Beispiel soll die Vorgehensweise beim Erstellen von Aktionen beschreiben: Damit eine Verknüpfung auf dem Desktop ausgeführt wird, muss ein Doppelmausklick auf diese Verknüpfung gemacht werden. Um dies zu automatisieren, wird die Aktion *doubleclick()* in Sikuli eingefügt. Da diese Aktion einen Bildvergleich enthält, muss eine Grafik als erwartetes Bild angegeben werden. Sikuli geht laut Standardeinstellung davon aus, dass der Nutzer gleich nach dem Einfügen einer Aktion, die einen Bildvergleich enthält, einen Screenshot als erwartetes Bild einfügen möchte. Dazu wechselt Sikuli in einen Aufnahmemodus, indem ein Bereich ausgewählt werden kann, der dann als erwartetes Bild, bezüglich der Aktion, in das Skript eingefügt und dort als kleine Miniatur dargestellt wird.

Das erwartete Bild muss nicht über einen Screenshot, sondern kann auch über eine Bilddatei im *.png*-Format eingefügt werden. Jedes erwartete Bild kann noch in Bezug auf die Ähnlichkeit, mit der es gefunden werden soll, dem Namen und der Position, an der im Falle eines Mausklicks geklickt werden soll, angepasst werden. Mit fundierten Programmierkenntnissen lassen sich somit auch komplexere Abläufe und damit auch AT erstellen und anpassen.

Um komplexere Tests mit Sikuli zu erstellen ist die Auslagerung und der Import von Methoden von Bedeutung, da dadurch eine hohe Wiederverwendbarkeit gewährleistet werden kann. Auch wenn dies lt. Dokumentation einfach zu programmieren ist, so treten doch unerwartet Fehler auf, weil Sikuli Zeilen falsch interpretiert, was dem Anwender viel Zeit kostet.

[7] dt. = Gottes Auge

Die anschließende Abbildung zeigt die GUI von Sikuli, auf der der Nutzer Abläufe erstellt. Sikuli stellt dabei die aufgenommenen Screenshots oder eingefügten Bilder in Miniaturansicht dar.

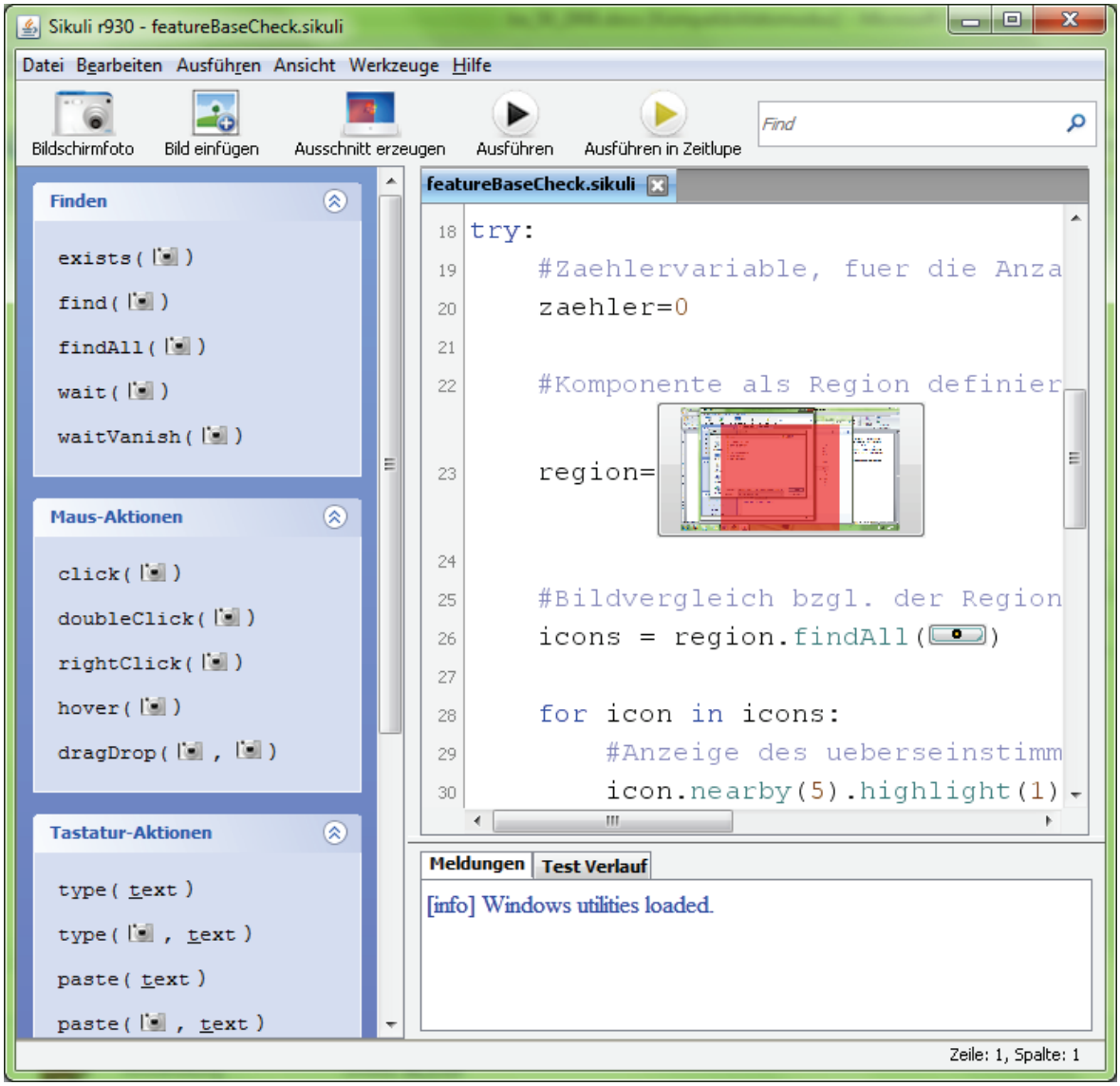



**Abb. 9: GUI von Sikuli X r930**

### 4.1.3 Vergleich der Testsoftwares

In Zahlen zusammengefasst ergibt sich eine Bewertung in Bezug auf die in Tabelle 5 (S. 17) genannten Anforderungen und deren Gewichtung. Diese wird in der nachfolgenden Tabelle (S. 23) dargestellt und anschließend erläutert.

**Tab. 6: Ergebnis der Analyse der Testsoftwares**

| Anforderung | Gewichtung | QF-Test 3.2.2 | | Sikuli X r930 | |
|---|---|---|---|---|---|
| | | Einzel-bewertung | bzgl. Gewichtung | Einzel-bewertung | bzgl. Gewichtung |
| Capture/Replay Verfahren | 1 | **3** | 3 | 1 | 1 |
| Integration zu der zu testenden Software | 2 | **4** | 8 | 3 | 6 |
| Bildvergleich | 4 | 1 | 4 | **4** | 16 |
| Erkennung geometrischer Formen | 3 | 0 | 0 | 0 | 0 |
| Aufnahmemöglichkeit der Bilder | 3 | 2 | 6 | **3** | 9 |
| Transparenz beim Bildvergleich | 3 | 0 | 0 | **1** | 3 |
| Strukturierung der Skripte | 1 | **4** | 4 | 2 | 2 |
| Gesamtbewertung: | | | 25 | | **37** |

| Legende bzgl. Gewichtung |
|---|
| 1 = grundlegend |
| 4 = unverzichtbar |

| Legende bzgl. Wertung |
|---|
| 0 = mangelhaft |
| 4 = sehr gut |

Das Ergebnis der Softwarebetrachtung zeigt, dass QF-Test bezüglich der Anforderungen schlechter als Sikuli ist.

Auch wenn QF-Test eine gute Capture/Replay-Funktionalität aufweist und Komponenten erkennt, so sieht das Testwerkzeug nur die Karte im RFK-GIS-Editor als eine Komponente, aber nicht die angezeigten Features. Daher erreicht QF-Test nicht die maximale Anzahl an Punkten bei diesem Kriterium. Sikuli verfügt nicht direkt über diese Funktion. Über eine Aktionenleiste[8] kann aber sehr schnell eine einfache Abfolge von Events erstellt werden. Daher bekommt Sikuli für diese Eigenschaft noch eine mangelhafte Bewertung.

QF-Test kann durch die Erkennung der Komponenten diese im vollen Umfang ansteuern und ist damit vollkommen integrationsfähig zur SUT. Sikuli erkennt zwar keine Komponenten aber bestimmte Bereiche durch den Vergleich mit dem Bildschirm und kann in Bezug auf diese Aktionen ausführen. Demzufolge ist Sikuli, der Integration betreffend, nur leicht nachteilig gegenüber QF-Test.

Beim Bildvergleich erreicht Sikuli mit Abstand die maximale Punktzahl, da Sikuli, unabhängig von einer Komponente, einen Bildvergleich mit stufenlos regelbarer Ähnlichkeit ausführen kann. QF-Test 3.2.2 hingegen kann nur einzelne Pixel vergleichen und bietet keine Alternative.

[8] Menüleiste in Sikuli, die Standard-Aktionen wie Mausklicks zum Einfügen zur Verfügung stellt

Leider können beide Testwerkzeuge keine geometrischen Formen erkennen, was eine Alternative zum Bildvergleich wäre.

Bei der Aufnahme von Bildern, die als erwartete Grafik beim Bildvergleich eingesetzt werden, bieten beide Testwerkzeuge eine einfache Bedienung und ermöglichen das Einfügen von externen Bilddateien. Da QF-Test sich immer auf die dazugehörige Komponente bezieht, kann kein direkter, Komponenten überschneidender Bildvergleich durchgeführt werden. Da Sikuli unabhängig von den Komponenten arbeitet, ist das ein entscheidender Vorteil gegenüber QF-Test. Somit können auch zum Beispiel Überschneidungen von Fenstern überprüft werden.

Die Einbindung von Bildern mit transparenten Bereichen (BmtB) ist in beiden Programmen möglich. QF-Test erkennt den transparenten Bereich aber als Fehler und kann dadurch mit solchen Bildern nicht arbeiten. Sikuli wandelt beim Bildvergleich, mit einer Ähnlichkeit von unter 99%, das Abbild des Bildschirmes und das erwartete Bild in Graustufen um und vergleicht diese miteinander. Bei der Erstellung von BmtB kann anstelle der transparenten Bereiche weiß oder schwarz eingefügt werden. Damit ist der Vergleich von BmtB in Sikuli möglich, aber abhängig von der Umgebung des Screenshots auf dem Bildschirm, da sonst bei einer hellen Umgebung und schwarzen Bereich des BmtB, beziehungsweise umgekehrt, der Test fehlschlägt. Folglich ist Sikuli in der Hinsicht mangelhaft, aber besser als QF-Test.

Das letzte Kriterium ist die Strukturierung des Skriptes eines Testfalles. Dies bezüglich ist QF-Test sehr gut, weil die Software dem Anwender viele Möglichkeiten dazu bietet, zum Beispiel durch die Erstellung von Methoden und der separaten Trennung von Komponenten und Testfällen. Um dies auch in Sikuli umzusetzen, braucht man spezielle Kenntnisse der Testsoftware und der Programmiersprache Python. So können zum Beispiel die Screenshots in einem separaten Ordner gespeichert werden. Dafür ist es erforderlich, das Verzeichnis im Skript anzugeben.

QF-Test hat seine Stärken in der Integration zur SUT und der Strukturierung der Skripte, kann aber bei den grafischen Kriterien, wie zum Beispiel beim Bildvergleich und der Bildaufnahme, nicht mit Sikuli standhalten. Die Gesamtbewertung zeigt, dass Sikuli in Bezug auf die Anforderungen eine sehr gute Lösung darstellt.

Da die Testfälle bereits mit QF-Test erstellt wurden und das Testwerkzeug weiterhin genutzt werden soll, wird im weiteren Verlauf der Arbeit analysiert, ob und wie Sikuli als

Erweiterung der Testautomatisierung eingesetzt werden kann, was die nachfolgende Grafik zeigt.

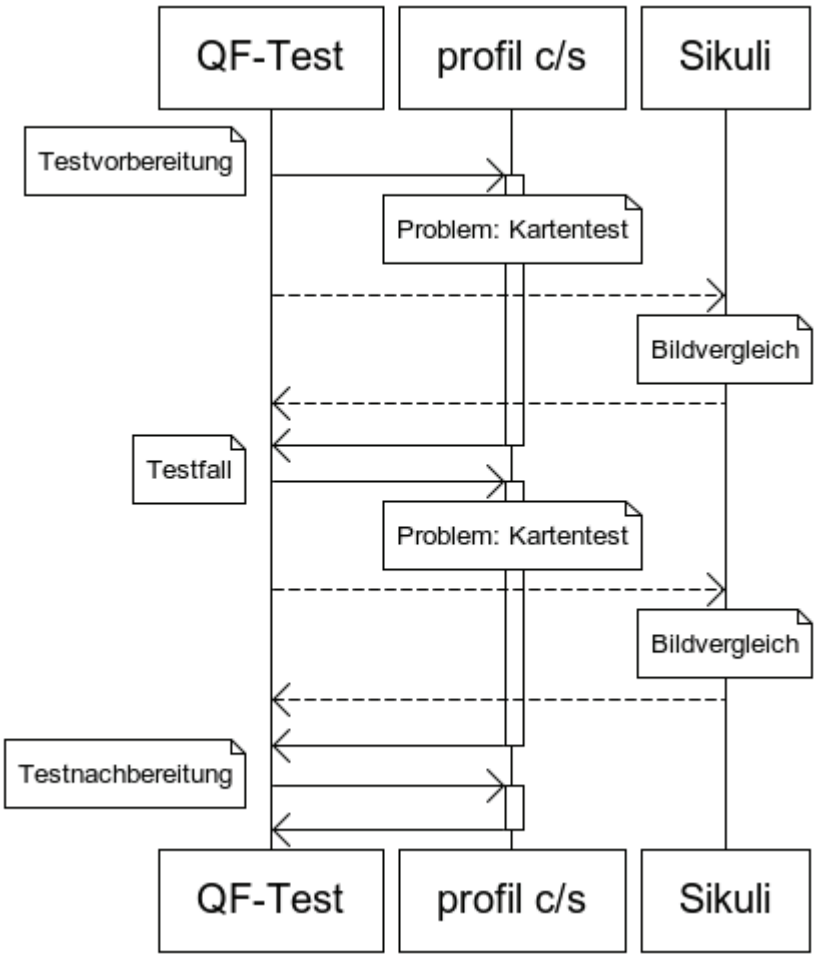



**Abb. 10: Mögliche Erweiterung von QF-Test mit Sikuli**

## 4.2 Erstellung der Testumgebung

Bevor ein Testfall im RFK-GIS-Editor durchgeführt werden kann, muss eine Testumgebung geschaffen werden. Zu Beginn der Arbeit war ein Service der Projektgruppe GIS der deg für die Erstellung der Testumgebung geplant. Dieser sollte beim Start eines Testfalles eine definierbare RFL erstellen und nach dem Durchlauf des Testfalles die Testnachbereitung vollziehen. Da der Wartungsaufwand dieses Services allerdings aus Sicht der Projektgruppe zu groß wäre, wurde die Testumgebung durch QF-Test erstellt.

Dazu gehört das Starten eines **profil c/s** Clients über Java-Webstart. Nachdem der Client gestartet wurde, muss der Nutzer sich anmelden. Nach einer erfolgreichen Anmeldung wird der Arbeitsplatz von **profil c/s** angezeigt. Nun muss eine neue RFL erstellt werden, die als Testobjekt dient. Da man eine RFL nur aus dem RFK-GIS-Editor heraus erstellen kann, muss dieser vorher für eine ausgesuchte Mappe geöffnet werden. Die Eingrenzung auf eine bestimmte, ausgesuchte Mappe ist notwendig, um die Testumgebung für diese Mappe durch vorhandene Daten nicht zu beeinflussen. Ein Beispiel für solche Daten sind RFL, die in einer beliebigen Mappe geografisch naheliegend auftreten können. Anschließend wird die RFL erzeugt, wofür der entsprechende Button nicht inaktiv sein darf. Danach muss überprüft werden, dass der Ident[9] fortlaufend weitergeführt und die RFL auch geografisch dort erstellt wird, wo die Position angegeben wurde. Die neue RFL muss die vorher gewählte Art und das fachlich gültige Datum haben und die Digitalisierungsgrundlage der vorherigen RFL übernehmen. Des Weiteren muss sie im Zustand „In Bearbeitung“ sein, dem Nutzer, der die RFL angelegt hat, als Original vorliegen und die Form eines Quadrates aufweisen. An dieser Stelle könnte schon ein GUI-Check testen, ob die neue RFL den Anforderungen entsprechend in der Karte angezeigt wird. Dieser GUI-Check ist identisch mit dem in Kapitel 4.3.2 beschriebenen Testfall.

Diese Funktionalitäten, die bei der Erstellung der Testumgebung genutzt werden, müssen bereits durch einen Testfall zuvor überprüft werden, wobei es keine Einschränkung beim Öffnen des Editors oder sonstigen Funktionalitäten geben sollte.

Ein Problem war die Bezeichnung des RFK-GIS-Editors, denn diese war identisch mit der Bezeichnung des RFL-Bearbeiters. Dadurch konnte QF-Test die Fenster nicht unterscheiden und die Events konnten der Komponente nicht eindeutig zugewiesen werden. Mit Hilfe eines Resolvers[10], der nun Bestandteil der Testvorbereitung ist, erfolgt eine Anpas-

[9] fortlaufende Nummer zur eindeutigen Identifizierung einer RFL
[10] Namensauflösung, lokale Änderung einer Bezeichnung [Vergleiche: IV: 25]

sung der Bezeichnung, wodurch der RFK-GIS-Editor vom RFL-Bearbeiter unterschieden werden kann.

Die nachfolgende Abbildung zeigt den sequenziellen Ablauf der Testvorbereitung.

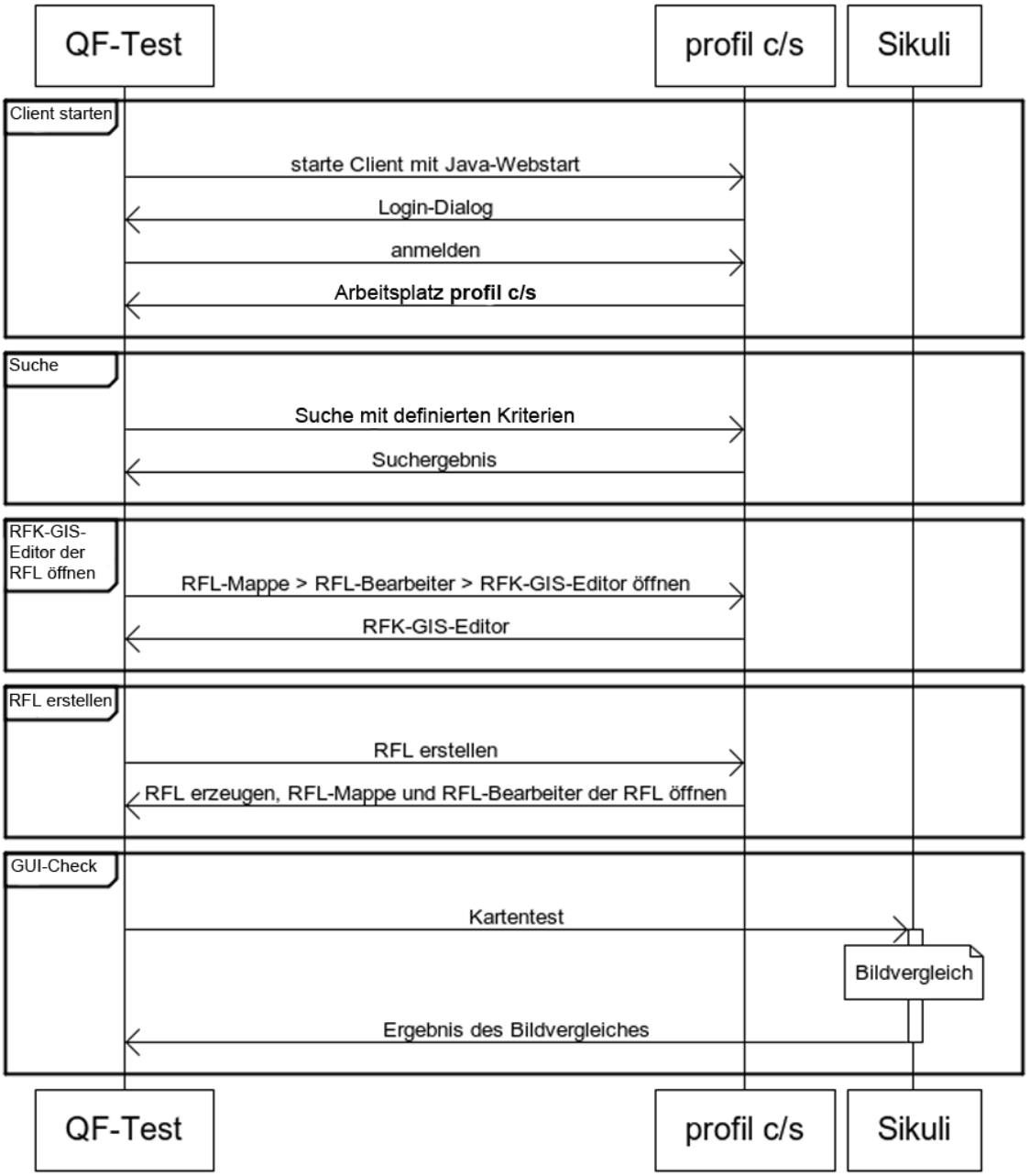



**Abb. 11: Sequenzdiagramm der Testvorbereitung**

Als allgemeine Beispiele sollen eine bearbeitete Geometrie grafisch kontrolliert und ein Feature selektiert werden. Die Kontrolle erfolgt über einen Aufruf von Sikuli aus QF-Test heraus. Die Ermittlung der Anzahl der angezeigten Features, anhand der dargestellten Labels, ist das extreme Beispiel, da die Labels sich überlappen und Sikuli dadurch eventuell nicht alle findet.

## 4.3 Prototypische Umsetzung von Testfällen mit Sikuli

Im Folgenden wird die Umsetzung der Testfälle beschrieben. Dabei soll der Begriff „Grafik“ für den Bereich des Bildschirmes stehen, der mit dem im Sikuli-Skript erwarteten Bild verglichen wird.
Außerdem beschränkt sich die Arbeit auf drei Testfälle aus der Tabelle 2 (S. 12): grafische Kontrolle der bearbeiteten Geometrie, Feature selektieren und Anzahl der Features ermitteln. Diese Testfälle sollen die allgemeine Umsetzung und die auftretenden, kritischen Aspekte von Sikuli zum Inhalt haben. Die mögliche Umsetzung der aus der Tabelle 2 nicht beschriebenen Testfälle werden im Kapitel 6 kurz beschrieben.

### 4.3.1 Start des Sikuli-Skriptes aus QF-Test heraus

QF-Test kann Programme über verschiedene Testknoten starten. So kann die SUT über den Testknoten „Java SUT Client starten“ ausgeführt werden, bei dem QF-Test ein Programm aufruft, welches wiederum Java ausführt. Eine weitere Möglichkeit ist der „SUT Client starten“ Testknoten, der den Start einer Applikation, die sonst via Shellskript oder Programm gestartet werden, realisiert. Auch für das Starten von Browsern gibt es einen speziellen Testknoten. Der Testknoten „Programm starten“ führt das Programm auf Shellebene, mit übergebenen Parametern, aus. [Vergleiche LV: 05]
Nachdem eine RFL erstellt wurde, wird das erste Mal mit Sikuli geprüft, ob der RFK-GIS-Editor die RFL richtig darstellt. Dies erfolgt in einer Prozedur in QF-Test über einen Aufruf heraus. Die Prozedur ist in zwei Sequenzen geteilt, wobei die, mit dem Aufruf von Sikuli, im folgenden Bild (S. 29) dargestellt und anschließend erläutert wird.

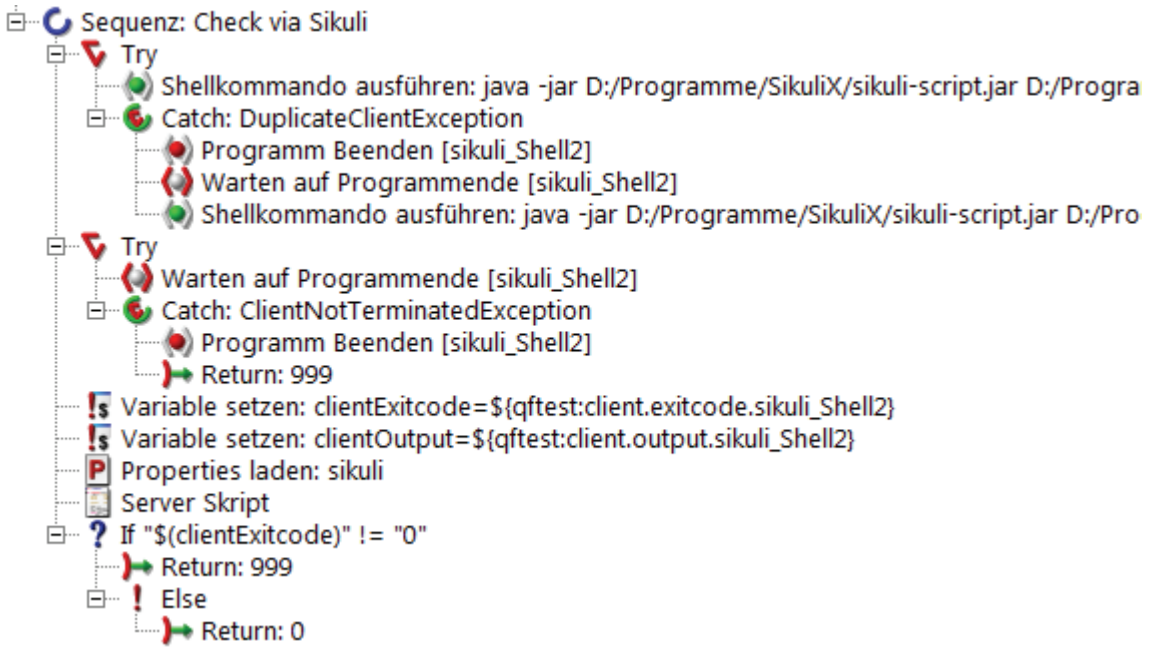



**Abb. 12: Sequenz in QF-Test mit Sikuli-Aufruf**

Zu Beginn versucht QF-Test ein Kommando auf Shellebene auszuführen, wofür die Testsoftware einen separaten Client startet. Als Erstes kommt das *jar*-File, dann kommen das Skript mit Verzeichnisangabe und die Übergabe eines Parameters, der im Sikuli-Skript abgefragt wird. Nachfolgend der ausgeschriebene Programmaufruf:

```
java -jar D:/Programme/SikuliX/sikuli-script.jar
  D:/Programme/SikuliX/gisMapCheck.sikuli $(flaeche)
```

**Abb. 13: Programmaufruf des Sikuli-Skriptes aus QF-Test**

Mit diesem Shellkommando wird das entsprechende Sikuli-Skript durchlaufen. Nach dem Durchlauf des Sikuli-Skriptes wartet QF-Test darauf, dass der Sikuli-Testlauf beendet wurde und setzt das Ergebnis des Durchlaufes und die Ausschrift von Sikuli auf jeweils eine Variable. Das Ergebnis des Vergleiches, den Sikuli ausführt, schreibt das Skript in eine *properties*-Datei, die nun von QF-Test gelesen und ausgewertet wird. Ist das Ergebnis des Testlaufes positiv, wird in der Prozedur „0“ als Rückgabewert ausgegeben, andernfalls „999“. Mit diesem Wert wird in QF-Test weitergearbeitet.

### 4.3.2 Testfall: grafische Kontrolle der bearbeiteten Geometrie

Bei dem Testfall „grafische Kontrolle der bearbeiteten Geometrie“ wird zuerst via QF-Test die RFL bearbeitet und anschließend das Sikuli-Skript „gisMapCheck.sikuli“[11], wie in Kapitel 4.3.1 beschrieben, aufgerufen.

Dieses Skript importiert zu Beginn zwei Skripte die Methoden enthalten: „pictureTest.sikuli“[12] und „featureTopLeft.sikuli“[13]. Anschließend überprüft Sikuli, ob ein Rasterlayer angezeigt wird oder nicht. Sofern eins angezeigt wird, bricht der Durchlauf ab, da Sikuli andernfalls nicht das Feature in der Karte erkennt. Dieses Problem lässt sich auch nicht mit dem Vergleich zwischen einem BmtB und dem Bildschirm beheben. Das Problem wurde bereits in Kapitel 4.1.3 erläutert. Die Grafiken für den Bildvergleich, die als erwartetes Bild dienen, sind in einem Verzeichnis gespeichert. Dieses Verzeichnis wird zu den Bildverzeichnissen von Sikuli hinzugefügt. Für die Protokollierung der Ergebnisse wird eine *properties*-Datei mit Schreibrechten geöffnet. Ist diese noch nicht vorhanden, wird sie angelegt. Danach wird der Parameter, der von QF-Test übergeben wird, ermittelt und der Variablen *argument* zugewiesen. Nachfolgend wird die Methode *runTest()* in der „pictureTest.sikuli“ mit der Variablen *argument* als Parameter aufgerufen.

Diese Methode versucht in einer Schleife eine Grafik, dessen Name den Wert der Variablen *argument* beinhaltet, in Bezug auf eine vorher definierte Region mit einer sinkenden Ähnlichkeit zu finden, beginnend bei 100%. Wird die Grafik gefunden, zeigt das eine Funktion an. Diese heißt *highlight()* und umrahmt den gefundenen Bereich für eine definierte Zeit. Der Wert der Ähnlichkeit, bei der die Grafik gefunden wurde, wird als Return-Wert zurückgegeben. Im Falle, dass die Grafik nicht gefunden wird, also der Bildvergleich negativ ausfällt, ist der Return-Wert „999“.

Das Ergebnis der Methode wird auf eine Variable gespeichert, die anschließend auf den Wert „999“ überprüft wird. Diesbezüglich wird eine Fehlermeldung in die Protokoll-Datei geschrieben und der Durchlauf abgebrochen. Im positiven Fall wird nun die Methode *getTopLeft()* in der „featureTopLeft.sikuli“ mit der Variablen *argument* und der vorher ermittelten Ähnlichkeit als Parameter aufgerufen.

Diese Methode versucht die Grafik, in Bezug auf eine Region, mit der global gespeicherten Ähnlichkeit zu finden, und die linke obere Ecke dieser Grafik in Bildschirmkoordinaten zu

[11] Anhang A.1, Seite IX
[12] Anhang A.2, Seite X
[13] Anhang A.3, Seite XI

ermitteln. Wird die Grafik nicht gefunden, wird „999“ als Return-Wert ausgegeben, andernfalls werden die Bildschirmkoordinaten in Bezug zu der Komponente berechnet, indem die linke obere Ecke der Komponente ermittelt und die Differenz gebildet wird. Der Grund dafür liegt darin, dass QF-Test die Position von Mausevents bezüglich der Komponente ausführt. Als Return-Wert werden die X- und Y-Koordinaten der Grafik bezüglich der Komponente zurückgegeben.

Auch das Ergebnis der Methode wird auf eine Variable gespeichert, die ebenfalls auf den Wert „999“ überprüft und im positiven Fall der Durchlauf vorzeitig beendet wird. Sollte das Ergebnis aber die Koordinaten enthalten, werden diese in die Protokoll-Datei geschrieben und der Durchlauf mit „0“, als erfolgreiches Ergebnis, beendet.

Liefert der Sikuli-Test „0“ als Ergebnis, wird in QF-Test die Testumgebung aufgeräumt und der Testfall beendet.

Die nachfolgenden Abbildungen (S. 32 - 34) zeigen den Programmablaufplan dieses Testfalles.

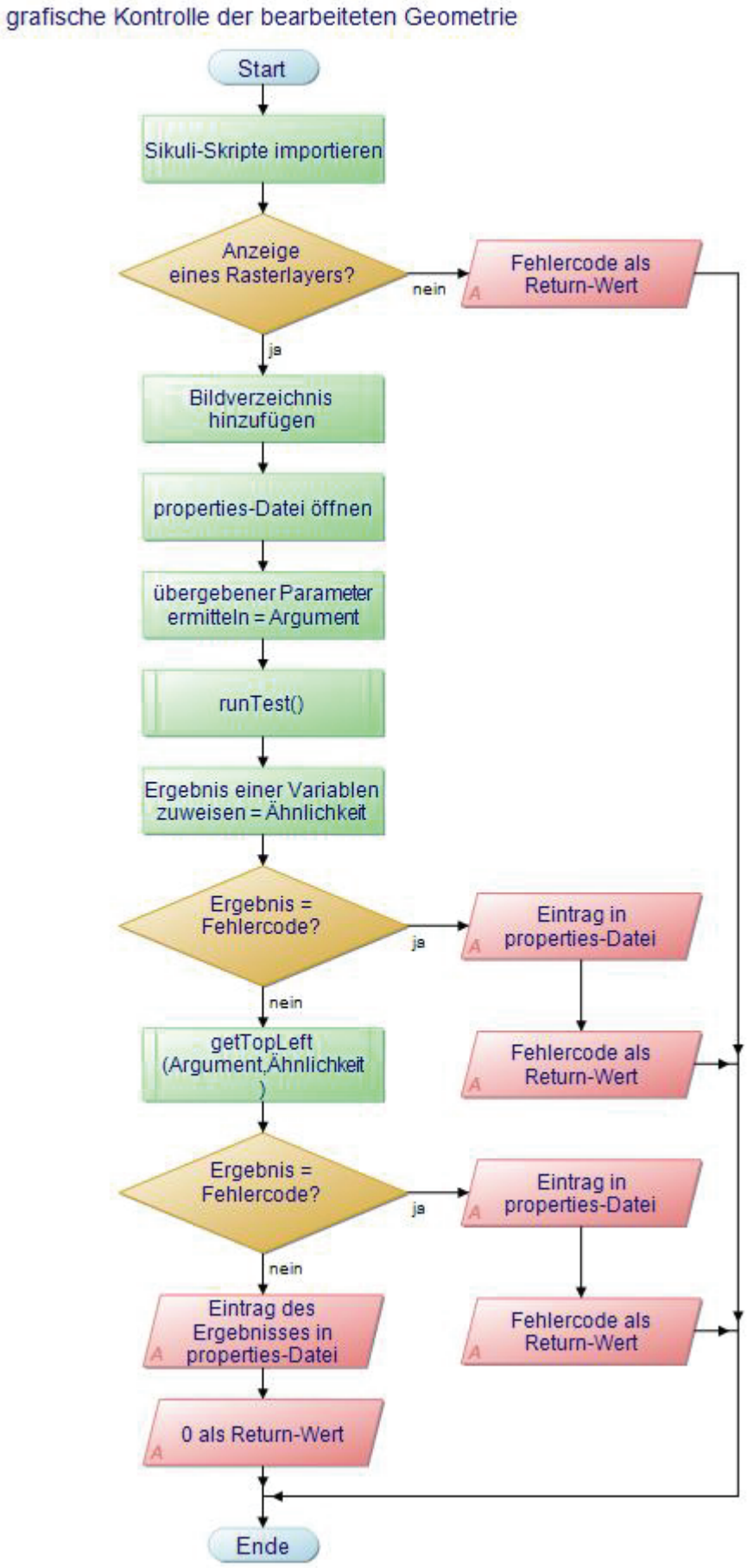



**Abb. 14: Programmablaufplan Testfall "grafische Kontrolle der bearbeiteten Geometrie"**

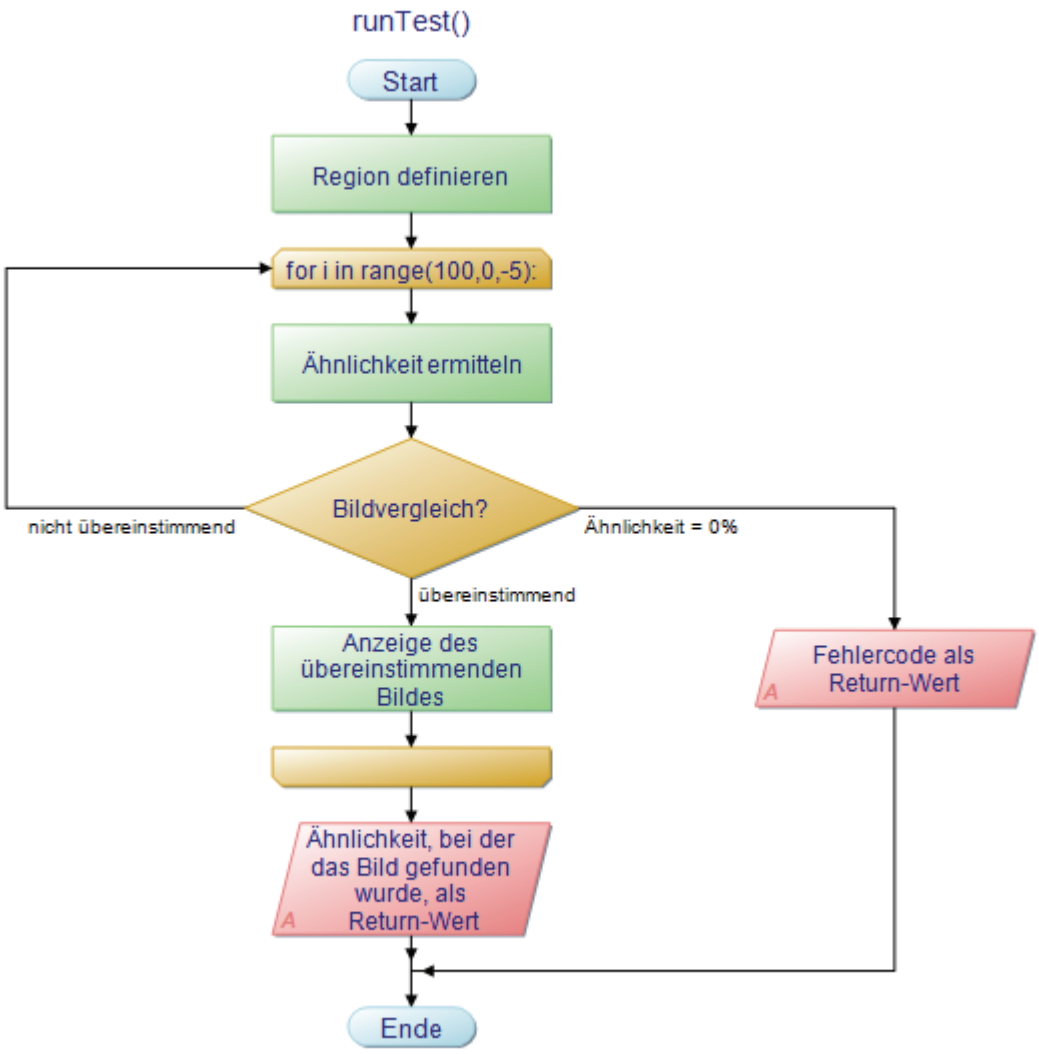



**Abb. 15: Programmablaufplan Methode „runTest()“**

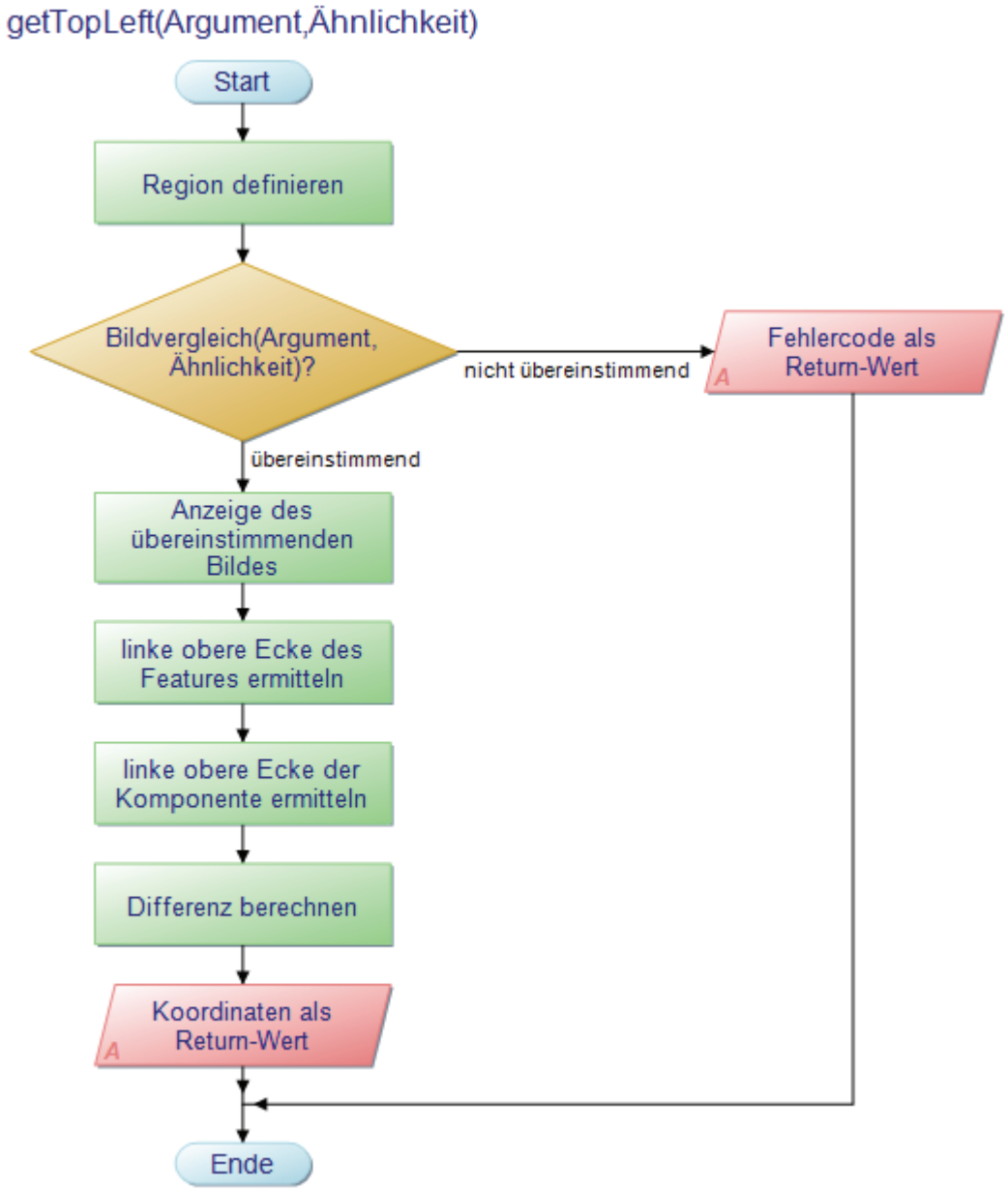



**Abb. 16: Programmablaufplan Methode „getTopLeft()“**

### 4.3.3 Testfall: Feature selektieren

Wie in Kapitel 4.3.1 beschrieben, wird zu Beginn des Testfalles als Testvorbereitung eine Testumgebung von QF-Test erzeugt, in der letztendlich getestet und das Sikuli-Skript „featureBaseCheck.sikuli“[14] dementsprechend aufgerufen wird.

[14] Anhang A.4, Seite XII

Zu Beginn des Testfalles wird das Feature im RFK-GIS-Editor ausgewählt und nachfolgend das Sikuli-Skript aufgerufen.

Das Sikuli-Skript importiert am Anfang das Sikuli-Skript „featureTopLeft.sikuli“[15], fügt das Verzeichnis, mit den beinhaltenden Grafiken, hinzu und öffnet eine *properties*-Datei, in der wichtige Daten geschrieben werden, die im Nachhinein von QF-Test ausgelesen werden kann. Anschließend werden eine Zählervariable und eine Region definiert. Danach versucht das Skript, in Bezug auf die Region, alle Grafiken, die einem Stützpunkt gleichen, zu finden und die Position der Grafik zu ermitteln. Die Koordinaten beziehen sich auf den Bildschirm und müssen noch transformiert werden, damit sie sich auf die Komponente beziehen, was in einer Methode im Skript „featureTopLeft.sikuli“ erfolgt. Die transformierten Koordinaten schreibt das Skript in die *properties*-Datei. Nach jeder gefundenen Grafik wird die Zählervariable hochgesetzt, sodass man am Ende die Anzahl der gefundenen Treffer hat, die das Sikuli-Skript ebenfalls in die Protokoll-Datei schreibt. Sofern Grafiken gefunden wurden, wird das Skript anschließend mit „0“ beendet, andernfalls mit „999“.

Die folgenden Abbildungen (S. 36 u. 37) stellen den Programmablaufplan dieses Testfalles dar.

[15] Anhang A.3, Seite XI

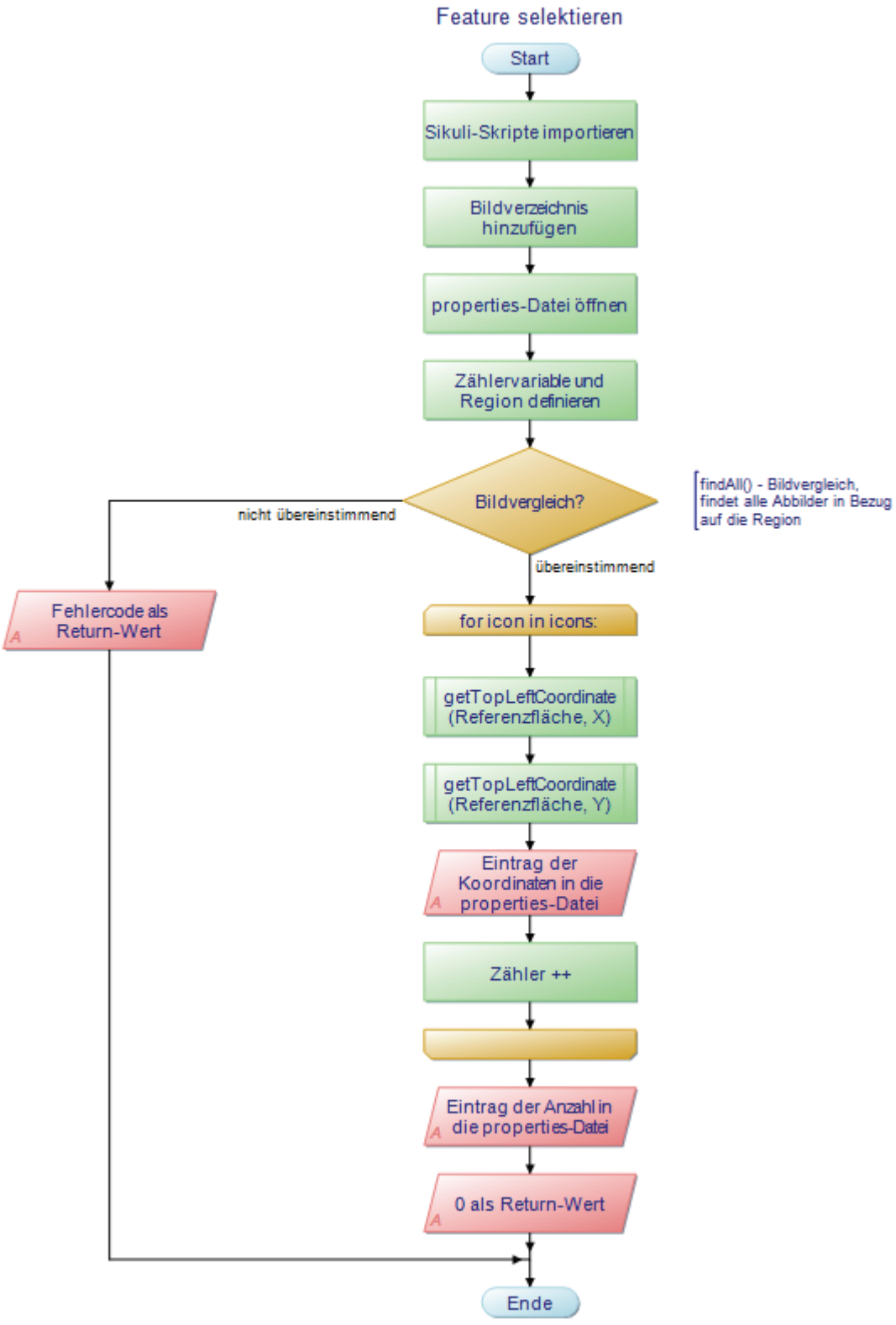



**Abb. 17: Programmablaufplan Testfall "Feature selektieren"**

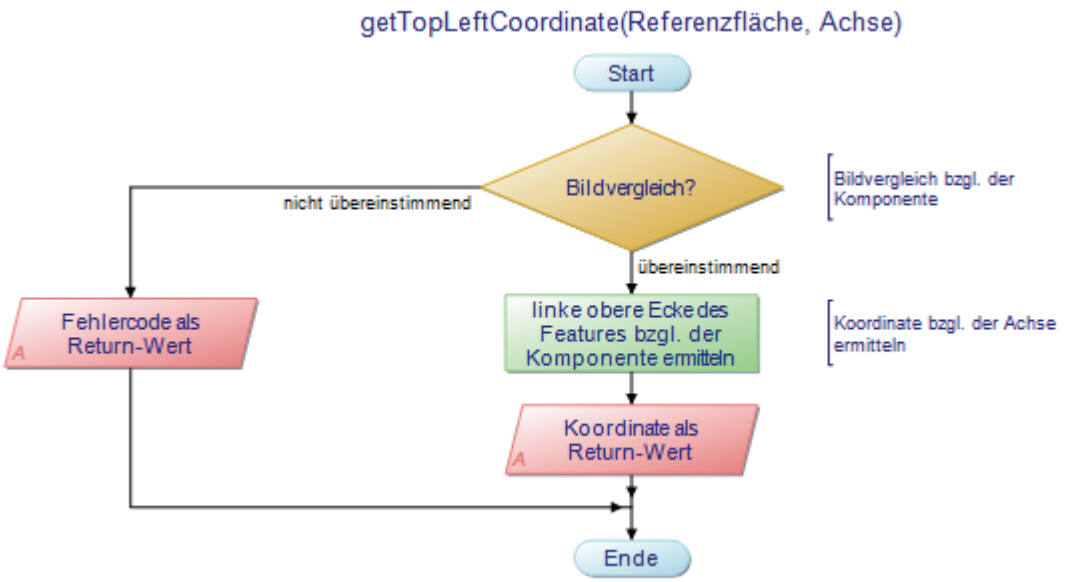



**Abb. 18: Programmablaufplan Methode „getTopLeftCoordinate()“**

### 4.3.4 Testfall: Anzahl der Features ermitteln

Der Test soll die Anzahl der angezeigten Features anhand der angezeigten Labels ermitteln. Dafür gibt es in Sikuli die Aktion *findAll()*[16], die alle Abbilder, in diesem Fall die Labels, auf dem Bildschirm sucht und als Liste speichert. Leider kann nicht die Standardmethode von Python zur Ermittlung der Länge der Liste genutzt werden, da das Ergebnis der Aktion *findAll()* ein Listen-Objekt ist, das keine Methode enthält, um die Anzahl der Elemente zu bestimmen. Alternativ dazu wird in einer Schleife die Liste durchlaufen und bei jedem Durchlauf eine Variable um eins addiert, was einem Zähler entspricht. Für den Fall, dass kein Bereich mit dem erwarteten Bild übereinstimmt, kommt ein Warnungsfenster mit dem Hinweis „keine Übereinstimmung gefunden“. Andernfalls wird am Ende des Durchlaufes die Anzahl der gefundenen Flächen in einem Fenster ausgegeben.
Dieser Testfall dient als extremes Beispiel, da sich die Labels bei der Darstellung im Kartenausschnitt überlagern, somit verdecken und folglich für Sikuli schwerer zu finden sind.
Die nächste Grafik (S. 38) illustriert den Programmablaufplan des Testfalles.

[16] CD > Sikuli > howToSikuli.pdf > Kapitel 2.3.1

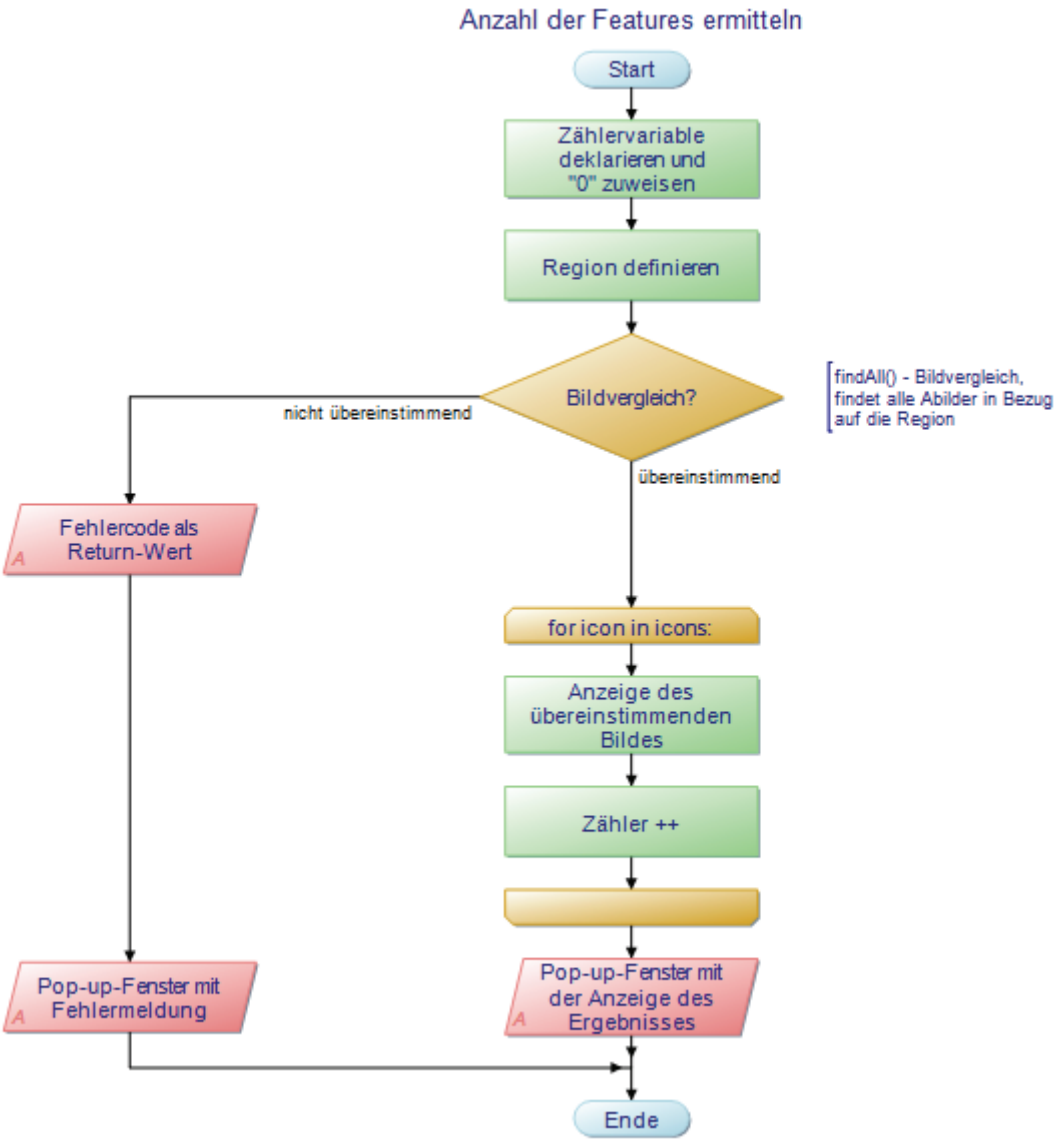



**Abb. 19: Programmablaufplan Testfall "Anzahl der Features ermitteln"**

### 4.3.5 Ergebnisse der Umsetzung

Der Testfall „grafische Kontrolle der bearbeiteten Geometrie“ dient als Schablone für andere Testfälle. Dabei erfolgt der Kartentest, also der Test des Kartenausschnittes mit der bearbeiteten Geometrie im RFK-GIS-Editor, mit Sikuli in der Testvorbereitung, nach der Erstellung einer RFL, und im Testfall, nachdem die RFL bearbeitet wurde. Die folgende Abbildung (S. 39) stellt diesen Zusammenhang symbolisch dar.

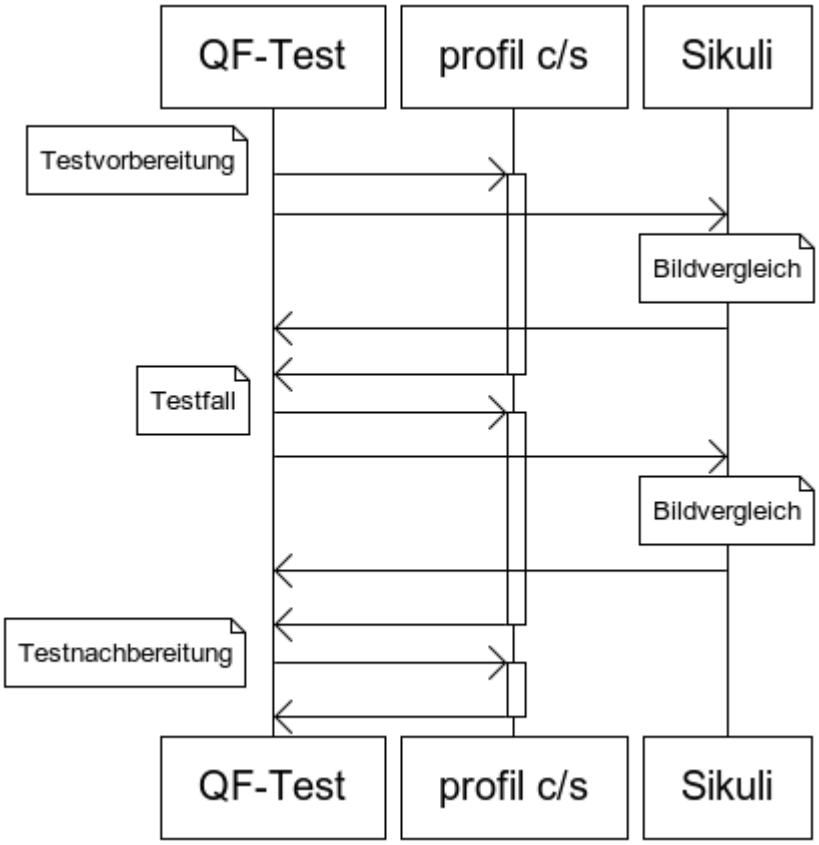



**Abb. 20: Verknüpfung von Sikuli und QF-Test**

In diesem Testfall wird noch die Koordinate der linken oberen Ecke des Features bezüglich der Kartenkomponente des RFK-GIS-Editors ermittelt. Dafür ist eine kleine Berechnung nötig, da Sikuli die Koordinaten nur auf den Bildschirm bezieht.

Auch der Testfall „Feature selektieren“ kann als Vorlage für weitere Testfälle genutzt werden, allerdings übernimmt Sikuli hier eine Aufgabe, und zwar die optische Erkennung, ob das entsprechende Feature selektiert wurde und wie viele Stützpunkte das Feature hat. QF-Test kann nur mit Hilfe eines SUT-Skriptes und der Angabe einer Koordinate, die sich innerhalb des zu testenden Features befindet, ermitteln, ob das Feature selektiert wurde, allerdings nicht, ob dies auch in der Karte angezeigt wird. Zusätzlich schreibt Sikuli die Bildschirmkoordinaten und die Anzahl der Stützpunkte in eine Protokoll-Datei, die dann von QF-Test ausgewertet werden kann.

Als extremes Beispiel soll der letzte Testfall „Anzahl der Features ermitteln“ die Grenzen von Sikuli aufweisen. Es wird dabei die Anzahl der angezeigten Features, mittels der Suche nach den Labels, ermittelt. Das Problem dabei ist, dass im Kartenausschnitt des RFK-GIS-Editors die Labels bei einer bestimmten Zoomstufe überlappend dargestellt werden und somit für Sikuli schlechter auffindbar sind.

# 5 Nutzung des Prototyps

## 5.1 Verwendung der erstellten Testfälle

Alle erstellten Skripte in Sikuli sind funktionstüchtig und die allgemeinen Beispiele, der umgesetzten Testfälle, können von QF-Test aus aufgerufen werden. Sollte es doch Probleme bei der Ausführung geben, empfiehlt sich das Kapitel „Problembehandlung“ [1] im How-to-Sikuli, welches sich auf der CD befindet.
Durch die Auslagerung von Bildern und wichtigen Programmabläufen in separate Skripte ist die Wiederverwendbarkeit und Erweiterbarkeit der erstellten Testfälle sehr hoch. Dadurch können andere Testfälle auf diese Methoden zugreifen und Änderungen werden nur an einer zentralen Stelle vollzogen.
Der Testfall „grafische Kontrolle der bearbeiteten Geometrie“ lässt sich gut wiederverwenden, da dieser für verschiedene Bildvergleiche zum Einsatz kommen kann, indem man in QF-Test den entsprechenden Parameter übergibt und das dazugehörige Bild, welches im Ordner „images“ ist, dann für Sikuli zum Bildvergleich zur Verfügung steht und genutzt werden kann. Durch das Einfügen neuer Bilder und den dazugehörigen Parameter, der von QF-Test genutzt wird, ist der Testfall schnell und einfach erweiterbar und kann auch für weitere Testfälle eingesetzt werden.
Auch der Testfall zur Ermittlung der Anzahl der Stützpunkte eines selektierten Features, „Feature selektieren“, hat eine gute Wiederverwendbarkeit, da der Bereich des Bildvergleiches eingeschränkt und die Transformierung der Bildschirmkoordinaten zum Bezug zur Komponente in einer separaten Methode durchgeführt wird, die durch die Auslagerung von anderen Testfällen ebenfalls aufgerufen werden kann.
Der als extremes Beispiel vorgestellte Testfall „Anzahl der Features ermitteln“ ist zwar von der Aufgabe sehr simpel, aber komplex in der Umsetzung, um ein exaktes und vor allem richtiges Ergebnis zu erhalten. Wie bereits mehrfach erwähnt, liegt das Problem darin, dass der RFK-GIS-Editor die Labels bei einer bestimmten Zoomstufe überschneidet, wodurch diese von Sikuli nicht mehr als solche erkannt werden können. Ein weiteres Problem ist, dass man die Vorschau bei den Einstellungen eines erwarteten Bildes[2] nicht als voraussichtliches Ergebnis beim Bildvergleich interpretieren kann. Um dies aber

[1] CD > Sikuli > howToSikuli.pdf > Kapitel 7
[2] CD > Sikuli > howToSikuli.pdf > Kapitel 2.1.2

trotzdem zu gewährleisten, muss das erwartete Bild auf den entscheidenden Bereich zugeschnitten und der Suchbereich verkleinert, also auf den Kartenausschnitt des RFK-GIS-Editors angepasst werden. Somit lässt sich auch diese Umsetzung des Testfalles realisieren.

## 5.2 Ergebnis der Verwendung

Das Resultat des Prototyps ist, dass mit Sikuli, und dem damit verbundenen Bildvergleich, der Kartenausschnitt des RFK-GIS-Editors und auch der Kartenkomponenten anderer GIS-Tools in **profil c/s** automatisiert getestet werden können. Die in der Arbeit vorgestellten Beispiele zeigen die Möglichkeiten, die Sikuli bietet, wie beispielsweise die Kontrolle, ob ein selektiertes Feature auch als solches dargestellt wird und wie viele Stützpunkte es hat. Allerdings werden auch die Grenzen, die das Open-Source-Tool hat, deutlich, und zwar bei der Erstellung und Ausführung von Testfällen. Wie bereits beschrieben, ist das Suchergebnis der Vorschau bei den Einstellungen eines erwarteten Bildes[3] nicht immer identisch mit dem Ergebnis der letztendlichen Suche beim Durchlauf. Ebenso ist der Wechsel zwischen der Ansicht mit Minitaturvorschauen der erwarteten Bilder (siehe Abb. 8 (S. 22)) und dem harten Quellcode über Sikuli nicht möglich. Auch wenn man das Python-Skript in einem Editor bearbeiten kann, so zieht das Probleme nach sich, die ohne detailliertes Wissen nicht schnell behoben werden können. Ein Beispiel dafür ist das für Python erforderliche Einrücken von Zeilen bezüglich Operationen. Sowohl in einem Editor als auch in Sikuli sieht der Quellcode für den Programmierer identisch aus, wird aber von Sikuli anders interpretiert, als von einem Editor. Ein weiteres Problem von Sikuli ist auch der softwareeigene Cache, der erst dann geleert wird, wenn in Sikuli keine Skripte offen sind und die Testsoftware geschlossen wurde. Dadurch entsteht ein zusätzlicher Zeitaufwand bei der Erstellung der Skripte. Ein Merkmal, das auch bei Sikuli vermisst wurde, ist das Setzen von Breakpoints und der Start eines Durchlaufes von einer beliebigen Stelle. Beides würde die Fehlersuche und Fehlerkorrektur im Skript bei der Erstellung und Ausführung erleichtern.

Trotz der genannten Probleme bleibt die Einsatzmöglichkeit der Testfälle uneingeschränkt, da die Probleme nur Verbesserungsvorschläge für das Testwerkzeug sind. Demzufolge lässt sich die Frage, inwiefern Sikuli für den Einsatz zum automatischen Testen von GIS-Komponenten in der deg geeignet ist, wie folgt beantworten: Sikuli schließt die offene Lücke bei der automatisierten Prüfung der GIS-Komponenten, speziell des Kartenausschnittes, und ist somit sehr gut für die Erweiterung der Testautomatisierung in der deg geeignet, auch wenn das Testwerkzeug kleinere Bedienungsprobleme aufweist. Allgemein

[3] CD > Sikuli > howToSikuli.pdf > Kapitel 2.1.2

lässt sich sagen, dass Sikuli genau dann eingesetzt werden kann, wenn die Umsetzung eines Testfalles mit QF-Test nicht realisierbar ist. Ein Beispiel dafür ist der in der Arbeit beschriebene Kartentest der GIS-Komponenten. Folglich kann Sikuli nicht nur für andere Testfälle mit geometrischem Testinhalt, wie die der in Tabelle 2. (S. 12) genannten GIS-Komponenten, sondern auch zum Testen von weiteren Komponenten, zum Beispiel mit grafischem oder funktionalem Testinhalt, eingesetzt werden. Damit bekommt Sikuli eine weitere Bedeutung für die deg.

Für den Einsatz sprechen die Möglichkeiten der Einstellungen beim Bildvergleich, wie zum Beispiel die stufenlose, prozentuale Angabe der Ähnlichkeit, die mit QF-Test nicht möglich ist. Ein weiterer Vorteil von Sikuli ist die Unabhängigkeit von Komponenten, denn dadurch können Komponenten übergreifende Aktionen getestet und ausgeführt werden. Für den Einsatz von Sikuli sind auch keine Änderungen im bisherigen System vorzunehmen. Es müssen lediglich die Testfälle mit QF-Test und die Skripte in Sikuli für den Bildvergleich programmiert werden. Hinzu kommt, dass Sikuli ein Open-Source-Tool ist und damit keine zusätzlichen Anschaffungs- oder Lizenzkosten verursacht.

# 6 Zusammenfassung und Ausblick

Zusammenfassend lässt sich zum Ende der Arbeit sagen, dass durch Sikuli eine effiziente und funktionsfähige Möglichkeit für die Erweiterung der Testautomatisierung in der deg vorgestellt wird.
Die Arbeit beschreibt zu Beginn das Testobjekt, den RFK-GIS-Editor, der ein Tool der Software **profil c/s** ist. Dies bezüglich wurden fachliche Anforderungen analysiert, um zu ermitteln, was zu testen ist. Das Resümee war, dass der Kartenausschnitt des Testobjektes mit dem bisherigen Testwerkzeug QF-Test nicht automatisiert getestet werden kann. Um diese Lücke in der Testautomatisierung in der deg zu schließen, ist der Einsatz einer weiteren Testsoftware erforderlich, wobei diese nur als Erweiterung von QF-Test eingesetzt werden sollte. Dafür wurden die technischen Anforderungen aufgestellt, welche die Suche nach einem nützlichen Testwerkzeug eingrenzten. Das Ergebnis der Recherche waren vier Werkzeuge. Durch eine kurze Softwarebetrachtung erwies sich Sikuli als zutreffende Erweiterung der Testautomatisierung. Daraufhin wurde QF-Test und Sikuli vorgestellt, auf die technischen Anforderungen analysiert und miteinander verglichen. Das Ergebnis des Vergleiches ist, dass Sikuli in Bezug auf die Anforderungen besser geeignet ist. Demnach erfolgte die prototypische Umsetzung von ausgewählten Testfällen die bereits in den fachlichen Anforderungen vorgestellt wurden.

Als Überblick gebend beschreibt die Arbeit nicht nur die Funktionsweise des Testwerkzeuges, sondern weist auch anhand von Beispielen die Umsetzung und Funktionsfähigkeit von Testfällen und Integration von Sikuli zum bisherigen Testwerkzeug QF-Test auf. Auch wenn letztendlich nur ein Testwerkzeug zur Erweiterung der Testautomatisierung untersucht und angewendet wurde, was auch durch die Aufgabenstellung so vorgegeben war, so erfolgte dies ausführlich, mit Beispielen belegend und ausblickend für weitere Testfälle, die nachfolgend beschrieben sind. So vermittelt die Arbeit nicht nur neue Kenntnisse von Testwerkzeugen, sondern auch Erweiterungen der Testautomatisierung.

Bei der prototypischen Umsetzung traten auch Fehler auf, die vorher bei der Softwarebetrachtung nicht ersichtlich waren. Ein Problem dabei war, dass in der Anzeige der Eigenschaften des erwarteten Bildes bei der „Vorschau Übereinstimmungen (Matches)“[1] mehr

[1] CD > Sikuli > howToSikuli.pdf > Kapitel 2.1.2

Treffer angezeigt, als letztendlich gefunden werden. Weitere Probleme können dadurch entstehen, dass das erwartete Bild zu viel Rand beinhaltet. Somit ist es manchmal erforderlich, die aufgenommenen Screenshots anschließend mit einem separaten Bildbearbeitungsprogramm zu zuschneiden. Sikuli bietet dafür keine Funktion oder Verlinkung an. Weitere Probleme sind im How-to Sikuli[2] nachlesbar. Dadurch, dass in dieser Arbeit die Probleme gefunden und genannt wurden, erleichtert das die weitere Arbeit mit Sikuli, denn bei der Erstellung weiterer Testfälle können diese berücksichtigt werden.

Als Ausblick wird die Umsetzung der Testfälle, mit geometrischem Testinhalt aus der Tabelle 2 (S. 12), betrachtet, die nicht im Kapitel 4.3 erläutert werden. Der in Kapitel 4.3.2 beschriebene Testfall dient als Grundriss für den Testfall „Grafische Kontrolle des Kartenausschnittes“, denn dabei soll der Bildvergleich bezüglich der ganzen Kartendarstellung ausgeführt werden. Somit kann sich die Umsetzung an den Testfall von Kapitel 4.3.2 orientieren oder sogar diesen anpassen und nutzen.
Die Änderung des Stils eines Features kann über QF-Test erfolgen, aber ob diese auch richtig im Kartenausschnitt dargestellt wird, lässt sich mit Sikuli überprüfen. Auch dieser Testfall gleicht dem in Kapitel 4.3.2 beschriebenen Testfall und kann wie der vorherige realisiert werden.
Der Testfall, ob ein Rasterlayer angezeigt wird, lässt sich bereits jetzt testen, da der in Kapitel 4.3.2 erläuterter Testfall bei der Anzeige eines Rasterlayers den Durchlauf abbricht und „990“ als Return-Wert zurückliefert. Ist es aber das Ziel, des noch nicht erstellten Testfalles, die Kontrolle der Anzeige des Rasterlayers an bestimmten geografischen Bereichen, so ist ein neuer Testfall mit einem Bildvergleich zu erstellen. Eine hilfreiche Methode dafür wäre im Sikuli-Skript „pictureTest.sikuli“[3] bereits vorhanden.
Komplexer wird der Testfall „Feature verschieben“, bei diesem muss zu Beginn und zum Ende die Position des Features ermittelt werden. Die Differenz zwischen beiden Positionen muss dabei mit dem Wert der Mausbewegung identisch sein. Die Mausbewegung selbst kann sowohl von QF-Test als auch von Sikuli durchgeführt werden. Unterstützend kann Sikuli dabei zur Ermittlung der Koordinaten des jeweiligen Stützpunktes eingesetzt werden.

[2] CD > Sikuli > howToSikuli.pdf > Kapitel 7
[3] Anhang A.2, Seite X

Die nächste Tabelle zeigt das Ergebnis der Erweiterung, wenn alle Testfälle mit Hilfe von Sikuli umgesetzt werden. Es gilt folgende Symbolik:

**Tab. 7: Umsetzung der Testfälle mit geometrischem Testinhalt**

| Testfall | manuell | JUnit-Test | QF-Test | Sikuli |
|---|---|---|---|---|
| Anzeigen von Labels | √ | X | X → | √ |
| Grafische Kontrolle der bearbeiteten Geometrie | √ | X | X → | √ |
| Grafische Kontrolle des Kartenausschnittes | √ | X | X → | √* |
| Feature selektieren | √ | X | X → | √ |
| Stil eines Layers ändern | √ | X | X → | √* |
| Rasterlayer anzeigen | √ | X | X → | √* |
| Feature verschieben | √ | X | X → | √* |

* Testfälle, die noch nicht umgesetzt wurden

| Legende | | | |
|---|---|---|---|
| √ wird getestet | X wird nicht getestet | → wird umgesetzt von | |

Als Ausblick wäre es möglich und unter Abschätzung des Kosten-Zeit-Aufwandes eventuell sogar nützlich, Testwerkzeuge, die ähnlich wie Sikuli mit Bildvergleichen fungieren, mit diesem zu untersuchen und zu vergleichen. Bevor dies aber als kritischer Aspekt der Arbeit betrachtet wird, sollte eine Analyse erfolgen, die den Grund des Einsatzes von Sikuli, beim Testen von GIS-Komponenten und keinem anderen Testwerkzeug, mit gleicher oder ähnlicher Testmethode, beinhaltet.

Ebenfalls könnte eine Analyse weitere Einsatzmöglichkeiten von Sikuli in der deg, wie zum Beispiel neue Testobjekte, aufweisen und damit die Testautomatisierung noch weiter unterstützen.

Die Arbeit belegt, dass mit dem Einsatz von Sikuli ein neues Potenzial für die Erstellung beziehungsweise Umsetzung von AT besteht, das über den analysierten Bereich des Kartenausschnittes des RFK-GIS-Editors hinaus genutzt werden kann.

Allgemein ist das Testen ein Bestandteil der Softwareentwicklung. Fehler einer Software lassen sich meistens auch durch langfristiges Testen, wie zum Beispiel durch Testautomatisierung, nicht beheben. Das bestärkt nur den von mir zu Beginn erwähnten und von Cem Kaner gesagten Satz: „Software products are never released - they escape“ [Cem Kaner, LV: 01]

# Quellen

## Literaturverzeichnis (LV)


01: Testing Computer Software; C. Kaner, J. Falk, H. Q. Nguyen; John Wiley & Sons; 1999

02: Angewandte Geoinformatik 2011 – Beiträge zum 23. AGIT-Symposium; J. Strobl, T. Blaschke und G. Griesebner; Herbert Wichmann; 2011

03: Abenteuer Softwarequalität; Kurt Schneider; dpunkt.verlag GmbH; 2007

04: Lexikon der Informatik, 14. Überarbeitete Auflage; P. Fischer, P. Hofer; Springer-Verlag: 2008

05: QF-Test – Das Handbuch, Version 3.2.2; Gregor Schmid, Quality First Software GmbH; 2010

# Internetverzeichnis (IV)


01: Profil; www.profilglossar.data-experts.de/index.php/Profil_(Programm); Stand 25.06.2012

02: RPA; www.profilglossar.data-experts.de/index.php/RPA; Stand: 25.06.2012

03: Referenzfläche; www.profilglossar.data-experts.de/index.php/Referenzfläche; Stand: 25.06.2012

04: NbF; www.profilglossar.data-experts.de/index.php/NBF; Stand: 25.06.2012

05: Testen; www.wirtschaftslexikon.gabler.de/Definition/testen.html; Stand 25.06.2012

06: Programmverifikation; www.wirtschaftslexikon.gabler.de/Definition/programmverifikation.html; Stand 25.06.2012

07: Automatisches Testen von Software; www.hompages.thm.de/~hg6458/ BlockseminarvortraegeWS05/ Automatisches_testen.pdf; Stand: 26.06.12

08: Automatisiertes Testen grafischer Benutzeroberflächen; BlueNote Qualitätssicherung; www.bea-projects.de/files/BlueNotes/BlueNote - GUI-Tests.pdf; Stand: 26.06.12

09: Grafische Benutzeroberfläche; www.geoinformatik.uni-rostock.de/einzel.asp?ID=826; Stand: 26.06.2012

10: InVeKoS; www.profilglossar.data-experts.de/index.php/InVeKoS; Stand: 26.06.2012

11: Feldblock; www.profilglossar.data-experts.de/index.php/Feldblock; Stand: 29.06.2012

12: Apache JMeter; http://jmeter.apache.org/; Stand: 06.07.2012

13: JMeter 2.6; http://www.heise.de/download/jmeter-3614889.html; Stand: 30.06.2012

14: Apache JMeter, Arbeit von Bundi Beat; http://www.gruntz.ch/courses/sem/ss03/JMeter.pdf; Stand: 30.06.2012

15: JMeter; https://wiki.thm.de/index.php/JMeter; Stand: 07.07.2012

16: Add-on; http://wirtschaftslexikon.gabler.de/Definition/add-on.html; Stand: 10.07.2012

17: SeleniumHQ; http://seleniumhq.org; Stand: 10.07.2012

18: Selenium; http://de.wikipedia.org/wiki/Selenium; Stand: 12.07.2012

19:Selenium für automatisierte Akzeptanztests einer Drupal-Installation; http://www.comm-press.de/blog/seleium-f-r-automatisierte-akzeptanztests-einer-drupal-installation; Stand: 13.07.2012

20:Analyse und Bewertung des Tools Selenium; http://winfwiki.wi-fom.de/index.php/Analyse_und_Bewertung_des_Tools_Selenium; Stand: 13.07.2012

21: Canoo WebTest; http://webtest.canoo.com/webtest/manual/WeTestHome.html; Stand: 13.07.2012

22: Qualitätssicherung im Softwareentwicklungsprozess der Software Life GmbH; http://winfwiki.wi-fom.de/index.php/Qualit%C3%A4tssicherung_im_Softwareentwicklungsprozess_der_Software_Life_GmbH#Softwaretests; Stand: 13.07.2012

23: Canoo WebTest, Testautomatisierung von Webanwendungen; http://people.apache.org/~sgoeschl/presentations/canoo-20030521.pdf; Stand: 16.07.2012

24:Automatisierte Testverfahren für webbasierte Anwendungen; http://www-dssz.informatik.tu-cottbus.de/publications/master-theses/oFischer_master_2009.pdf; Stand: 16.07.2012

25: Resolver; einstein.informatik.uni-oldenburg.de/rechnernetze/resolver.htm; Stand: 24.07.2012

26: JMeter Präsentation; https://wiki.thm.de/images/6/6d/JMeter.zip; Stand: 10.07.20127

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungs- und Begriffsverzeichnis

| | |
|---|---|
| Add-on | „Funktionserweiterung bestehender Hard- oder Software“ [IV: 16] |
| AT | automatisierte Testfälle;<br>Bereich der Testautomatisierung, indem eine Testsoftware einen Testfall als Ablauf speichern und ausführen kann |
| BmtB | Bild mit transparentem Bereich |
| deg | data experts gmbh |
| FAQ | Frequently Asked Questions;<br>dt. = häufig gestellte Fragen |
| FB | Feldblock;<br>„ist eine zusammenhängende landwirtschaftlich nutzbare Fläche, die von erkennbaren Außengrenzen umgeben ist“ [IV: 11] |
| FB-Zone | Feldblock-Zone;<br>Zusammenfassung mehrerer FB zu einer Zone |
| Feature | Geometrie einer Fläche, die in einer Karte angezeigt wird |
| GIS | Geo-Informations-Systeme;<br>rechnergestützte Verarbeitung komplexer, raumbezogener Informationen |
| GUI | Grafische Benutzeroberfläche;<br>„~ bieten eine Methode zur Bedienung des Computers unter Verwendung von bildlichen Schaltflächen und Befehlsleisten, die mit der Maus gesteuert werden.“ [IV: 09] |
| GUI-Check | Bildvergleich zwischen dem was angezeigt und dem was erwartet wird |
| InVeKoS | Integriertes Verwaltungs- und Kontrollsystem;<br>„~ ist ein Hilfsmittel für die Beihilfenverwaltung zur Umsetzung der EU-Agrarreform, inzwischen sogar ein Hilfsmittel für den Landwirt bei der richtigen Antragstellung.“ [IV: 10] |
| Java-Swing | Java-Bibliothek um GUI zu programmieren |
| JUnit-Tests | „Java-Modul-Test; Java-Klassen werden direkt und parallel zur Erstellung automatisiert getestet.“ [LV: 03] |

| | |
|---|---|
| Komponente | „In der objektorientierten oder verteilten Programmierung große, mit universellen Schnittstellen versehene und deshalb in mehreren Anwendungen einsetzbare oder durch solche verwendbare Objekte.“ [LV: 04] |
| NbF | Nicht beihilfefähige Fläche;<br>„~ sind Bereiche auf Feldblöcken, auf denen der Antragsteller keine Beihilfe beantragen darf. Dabei handelt es sich zum Beispiel um Strommasten, Silos oder Güllegruben.“ [IV: 04] |
| Orthophotos | entzerrtes Luftbild |
| Plug-In | Zusatzprogramm bzw. Code-Module, das vom Browser bei Bedarf geladen wird und die Funktionalität erweitert |
| **profil c/s** | „**PRO**grammsystem zur rechnergestützten Verwaltung von **F**ördermitteln **I**n der **L**andwirtschaft“ [IV: 01] |
| Programmverifikation | „Formale Vorgehensweise mit dem Ziel, die Korrektheit eines Programms bzw. Moduls zu beweisen.“ [IV: 06] |
| Regressionstest | „~ sollen die Erhaltung von bekannten funktionalen Vorgängen sicherstellen, d.h. dass keine Änderung eine funktionale Beeinträchtigung zur Folge hat.“ [IV: 07] |
| RFK | Referenzflächenkataster;<br>Verwaltung und Pflege der RFL |
| RFL | Referenzfläche;<br>„Die RFL bezeichnet zwei Dinge: 1. Die Fläche in Hektar einer Referenzparzelle im Referenzsystem (RFK oder ALB) 2. Die Referenzparzelle (Feldblock (FB) / Flurstück) bzw. das Landschaftselement (LE) selber.“ [IV: 03] |
| RPA | Referenzflächenpflegeauftrag;<br>„Die Pflege einer RFL wird durch einen Pflegeauftrag ausgelöst. Der Pflegeauftrag beschreibt die durchzuführenden Änderungen. Über einen Status ist ersichtlich, ob der Auftrag abgearbeitet ist, oder nicht.“ [IV: 02] |
| Shapefile | Dateiformat für Geodaten; entwickelt von ESRI |
| SUT | System Under Test (dt. = Testsystem)<br>System, das getestet und unter einem automatisierten Test ausgeführt wird, Begriff wird nur in QF-Test verwendet |

| | |
|---|---|
| UT | Unit-Test;<br>Alle Arten von Tests lauffähiger Teile der Software [LV: 03] |
| WFS | Web Feature Service;<br>Dienst der OGC; Schnittstelle zur Beschreibung von Operationen zur Datenmanipulation von Geometrieobjekten |
| WMS | Web Map Service;<br>Schnittstelle zur Erzeugung und Darstellung von georeferenzierten und überlagerten Karten |
| XML | e**X**tensible **M**arkup **L**anguage;<br>Beschreibungssprache zur Darstellung hierarchisch, strukturierter Datensätze |

# Anhang

## A Quellcode

### A.1 gisMapCheck.sikuli

```
# -*- coding: utf-8 -*-
from sikuli import *

#Ueberpruefen, ob die Geometrie entsprechend der Vorgabe angezeigt wird und
#Ermittlung der Bildschirmkoordinaten der linken oberen Ecke des Features

#Importieren von Sikuli-Skripten
import pictureTest
import featureTopLeft

#Test, ob Rasterlayer ausgeblendet ist
try:
  find(Pattern("grayBackground.png").exact())
except FindFailed:
  #Rasterlayer ist eingeblendet: Abbruch des Durchlaufes, da sonst der Bildvergleich misslingt
  #Rueckgabewert an QF-Test
  sys.exit(990)

#Pfad zu Sikuli
pfad = "D:\\Programme\\SikuliX\\"

#Pfad zum Bildverzeichnis
pfadImages = pfad+"images"
addImagePath(pfadImages)

#Protokoll-Datei anlegen
pfadProperties=pfad+"log.properties"
log = open(pfadProperties,"w")

#uebergebener Parameter ermitteln
argument = sys.argv[1]

#Methodenaufruf zum Bildvergleich
check = pictureTest.runTest(argument)

#Kontrolle, ob die Methode erfolgreich durchlief,
#andernfalls Protokollierung und Abbruch
if check == 999:
```

```
39    #Eintrag in die Protokoll-Datei
40    log.write("X=Fehler \n")
41    log.write("Y=Bildvergleich fehlerhaft")
42    #Schliessen der Protokoll-Datei
43    log.close()
44
45    #Rueckgabewert an QF-Test
46    sys.exit(991)
47
48  #Methodenaufruf Koordinate des Features bzgl. der Kompo-
    nente ermitteln
49  coordinate= featureTopLeft.getTopLeft(argument, check)
50
51  if cood == 999:
52    #Eintrag in die Protokoll-Datei
53    log.write("X=Fehler \n")
54    log.write("Y=Berechnung von TopLeft fehlerhaft")
55
56    #Schliessen der Protokoll-Datei
57    log.close()
58
59    #Rueckgabewert an QF-Test
60    sys.exit(992)
61  else:
62    #X und Y Werte ermitteln
63    index = cood.index("/")
64    x = cood[:index]
65    y = cood[index+1:]
66
67    #Eintrag in die Protokoll-Datei
68    log.write("XKoordinate="+x+"\n")
69    log.write("YKoordinate="+y)
70
71    #schliessen der Protokoll-Datei
72    log.close()
73
74    #Rueckgabewert an QF-Test
75    sys.exit(0)
```

## A.2 pictureTest.sikuli

```
1  # -*- coding: utf-8 -*-
2  from sikuli import *
3  #Schleife, die die Aehnlichkeit des Bildes prueft
4  #und dabei von 100 Prozent ausgehend die Aehnlichkeit
5  #reduziert, bis der find() positiv ist
6
7  def runTest(picture):
8      #Komponente als Region definieren
9      region = Region(250,109,740,760)
```

```
10
11    for i in range(100,0,-5):
12      #prozentuale Aehnlichkeit berechnen
13      simi = float(i)/100
14
15      grafik = region.exists(Pattern("feature"+
          picture+".png").similar(simi))
16      if grafik:
17        #Anzeige des ueberseinstimmenden Bildes
18        grafik.nearby(5).highlight(1)
19
          #Rueckgabewert entspricht der Aehnlichkeit, bei der
20        das Bild gefunden wurde
21        return(simi)
22
23    #Rueckgabewert ist 999, was einem Fehler entspricht
      return(999)
```

## A.3 featureTopLeft.sikuli

```
1  # -*- coding: utf-8 -*-
2  from sikuli import *
3  #Berechnung der Koordinate der linken oberen Ecke eines
   Features
4  #bzgl. der Kartenkomponente im RFK-GIS-Editor
5
6  def getTopLeft(argument, similarity):
7    #Berechnung der Bildschirmkoordinate mit Bildvergleich
8    try:
9      #Komponente als Region definieren
10     region = Region(250,109,740,760)
11
12     #Bildvergleich des Features
13     referenzlaeche = region.find(Pattern("feature"+
          argument+".png").similar(similarity))
14
15     #Anzeige des ueberseinstimmenden Bildes
16     referenzlaeche.nearby(5).highlight(1)
17
18     #linke obere Ecke des Features ermitteln
19     referenzlaecheKoordinateX = rfl.getX
20     referenzlaecheKoordinateY = rfl.getY
21
22     #Bildvergleich der Komponente
23     komponente=find(Pattern("editor_kartenKomponente.png")
         .similar(0.75))
24     komponente.nearby(5).highlight(1)
25
26     #linke obere Ecke der Komponente ermitteln
27     komponenteKoordinateX = komponente.getX
```

```
28      komponenteKoordinateY = komponente.getY
29
30      #linke obere Ecke des Features bzgl. der Komponente
        ermitteln
31      koordinateX = referenzlaecheKoordinateX-
          komponenteKoordinateX
32      koordinateY = referenzlaecheKoordinateY-
          komponenteKoordinateY
33
34      #Rueckgabewert ist die X- und Y-Koordinate, getrennt
          mit einem Slash
35      return(str(koordinateX)+"/"+str(koordinateY))
36
37    except FindFailed:
38      #Rueckgabewert ist 999, was einem Fehler entspricht
39      return(999)
40
41  def getTopLeftCoordinate(referenzflaeche, achse):
42    #Berechnung der X-Koordinate bzgl. der Komponente
43    try:
44      #Bildvergleich der Komponente
45      komponente=find(Pattern("editor_kartenKomponente.png")
          .similar(0.75))
46
47      #X- bzw. Y-Wert der linken oberen Ecke des Features
            bzgl. der Komponente ermitteln
48      if achse == "x":
49          komponenteKoordinate = komponente.getX()
50      if achse == "y":
            komponenteKoordinate = komponente.getY()
51
52      #linke obere Ecke des Features bzgl. der Komponente
53        ermitteln
54      return(int(referenzflaeche)-komponenteKoordinate+3)
55
56    except FindFailed:
57      #Rueckgabewert ist 999, was einem Fehler entspricht
58      return(999)
```

## A.4 featureBaseCheck.sikuli

```
1  # -*- coding: utf-8 -*-
2  from sikuli import *
3
4  #ermittelt die Anzahl der Stuetzpunkte eines selektierten
   Features
5
6  #Importieren von Sikuli-Skripten
7  import featureTopLeft
8
```

```
 9 #Pfad zu Sikuli
10 pfad = "D:\\Programme\\SikuliX\\"
11
12 #Bildverzeichnis hinzufügen
13 addImagePath(pfad+"images")
14
15 #Protokoll-Datei anlegen
16 log = open(pfad+"log.properties","w")
17
18 try:
19   #Zaehlervariable, fuer die Anzahl der Stuetzpunkte
20   zaehler=0
21
22   #Komponente als Region definieren
23   region = Region(250,109,740,760)
24
25   #Bildvergleich
26   Icons = region.findAll(Pattern("stuetzpunkt.png")
       .similar(0.85))
27
28   for icon in icons:
29     #Anzeige des uebereinstimmendes Bildes
30     icon.nearby(5).highlight(1)
31
32     #Umwandeln der Koordinaten zum Bezug zur Komponente
33     iconScreenX = featureTopLeft.getTopLeftX(
         str(icon.getX()))
34     iconScreenY = featureTopLeft.getTopLeftY(
         str(icon.getY()))
35
36     #Koordinaten der gefundenen Icons ermitteln
37     log.write("X"+str(zaehler)+"="+str(iconScreenX)+"\n")
38     log.write("Y"+str(zaehler)+"="+str(iconScreenY)+"\n")
39
40     #Zaehlervariable hochsetzen
41     zaehler = zaehler + 1
42
43 except FindFailed:
44   #Protokoll-Datei schliessen
45   log.close()
46
47   #Rueckgabewert an QF-Test
48   sys.exit(999)
49
50 #Anzahl der Stuetzpunkte in die Protokoll-Datei schreiben
51 log.write("Anzahl= "+str(zaehler))
52
53 #Schliessen der Protokoll-Datei
54 log.close()
55
56 #Rueckgabewert an QF-Test
57 sys.exit(0)
```

## A.5 labelCountCheck.sikuli

```
# -*- coding: utf-8 -*-
from sikuli import *

#Zaehlen der angezeigten Labels

try:
  #Zaehlervariable definieren
  n=0

  #Komponente als Region definieren
  region = Region(250,109,740,760)

  #Bildvergleich
  Icons=region.findAll(Pattern("DESHL.png").similar(0.70))

  #alle gefundenen Bereiche anzeigen
  for icon in icons:
    icon.nearby(3).highlight(1)

    #Zaehler hochsetzen
    n=n+1
except FindFailed:
  #Warnung, wenn kein Bereich gefunden wurde
  popup("keine Uebereinstimmung","Warnung")

#Ausgabe der Anzahl, wieviele Bereiche gefunden wurden
popup("Es wurden "+str(n)+" FLIKs gefunden","Information")
```

# B ausgeschlossene Testwerkzeuge

## B.1 JMeter 2.7

Mit JMeter können Lasttests bezüglich Server/Clientanwendungen ausgeführt werden, wobei die Anwendung in Java geschrieben sein muss. Die Tests beziehen sich auf webbasierte Systeme und erzeugen Anfragen an diese. Dabei ist die Testsoftware durch die Java-Architektur unabhängig vom Betriebssystem. Eine ausführliche Dokumentation[1], mehrere Tutorials[2], ein Wiki[3] unter anderem mit FAQs, bieten dem Anwender ausreichend Hilfe für Probleme.

Eine Installation ist nicht erforderlich, da die aktuellste Version (letzte Änderung im Mai 2012) portabel ist und direkt ausgeführt werden kann. Durch die GUI der Software ist die Bedienung für den Nutzer nachvollziehbar, wobei es auch einen None-GUI-Modus und Server-Modus gibt. Ein Testfall heißt in JMeter Testplan. Das Skript eines Testplanes wird in *.xml*-Dateien gespeichert. Die beinhaltenden Punkte eines Testplanes werden strukturiert dargestellt. Ein Bestandteil eines Testplanes ist zum Beispiel ein *Logic Controller*, mit dem sich Wiederholungen von Anfragen an das System realisieren lassen. Sampler[4] stellen die eigentliche Anfrage an den Server. Über *Listeners* erfolgen Protokollierungen, die nach dem Durchlauf eines Testfalles ausgewertet werden können. Ebenso lassen sich auch Wartezeiten über *Timer* zwischen einzelnen Abfragen definieren. Weitere Bestandteile sind *Assertions*, *Configuration* – und *Processor* Elemente. Durch eine integrierte Capture/ Replay-Funktion des Programmes lassen sich neue Testfälle schnell erstellen. Dadurch können Änderungen gut gewartet werden.

Die Testsoftware lässt sich durch den offenen Programmcode an persönliche Anforderungen anpassen. Dazu gehört auch die Erweiterung auf dem Prinzip von Add-ons. Das Einfügen einzelner Skripte ist nur darüber möglich.

Die Erkennung von geometrischen Elementen auf der GUI ist allerdings nicht möglich. JMeter testet nur die Anfragen und Antworten zwischen dem Client und dem Server, aber nicht, ob die Darstellung auch so erfolgt. [Vergleiche IV: 12 – 15, 25]

[1] http://jmeter.apache.org/usermanual/index.html
[2] u.a. http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.pdf
[3] http://wiki.apache.org/jmeter/
[4] spezielle Controller

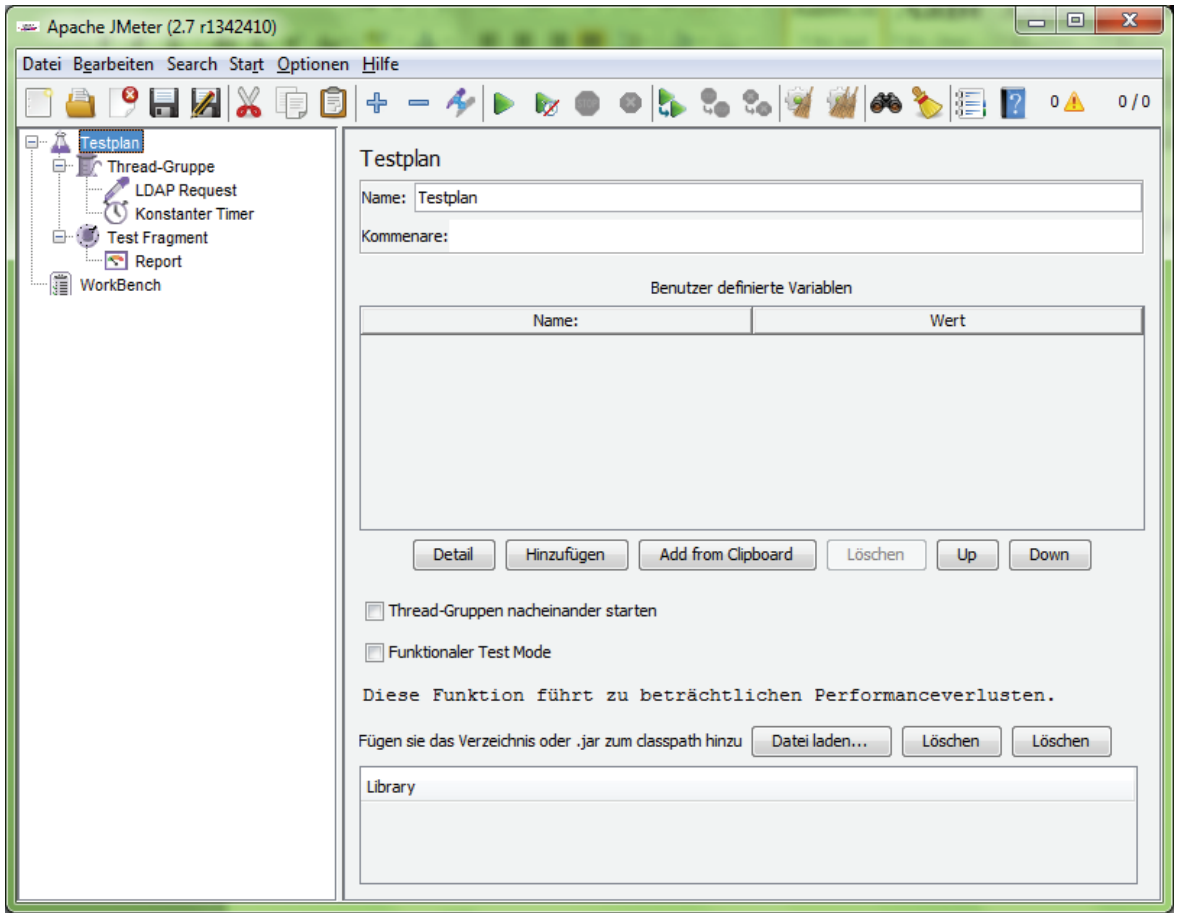



**Abb. 21: GUI von JMeter**

## B.2 Selenium 1.8.1

Selenium ist ein Open-Source-Testwerkzeug, um für Webanwendungen automatisierte Tests zu erstellen. Dafür werden vier Projekte angeboten. Das erste Projekt ist Selenium IDE und dient dazu, einfache Testfälle via Capture/Replay-Verfahren oder programmierten Skripten zu erstellen (siehe Abb. 22 (S. XVII)). Dabei wird das Tool als Add-on in den Browser Firefox installiert und dort ausgeführt. Bearbeitete Testfälle können über das zweite Projekt, Selenium Remote Control (Selenium RC), durchgeführt und bearbeitet werden. Dieses Projekt ist ein Client/Server System, mit dem über verschiedenste Programmier-sprachen Web-Browser gesteuert werden können. Das dritte Projekt, Selenium WebDriver, kontrolliert einen Browser auf lokalen oder entfernten Computern. Das letzte Projekt ist Selenium Grid, welches Selenium RC um Funktionalitäten, wie zum Beispiel Lauftests auf parallelen Servern, erweitert.

Die Erstellung von Testfällen ist bezüglich Selenium IDE durch die Aufnahmemöglichkeit sehr einfach und verständlich. Der aufgenommene Testablauf wird in der Selenium eige-

nen Syntax Selenese gespeichert. Da Selenese keine logischen Operationen hinzufügen kann, ist es möglich, die Syntax in eine standardisierte Programmiersprache, wie zum Beispiel Java, C#, PHP oder Python, zu übersetzen und anschließend zu bearbeiten, wodurch Testfälle erweiterbar sind. Weil aber Selenium nur seine eigene Syntax Selenese interpretiert, braucht man für die Ausführung Selenium RC. Das Tool ist ein Plug-In und damit unabhängig vom Browser.

Der Anwender kann beim Einsatz des Testwerkzeuges auf eine Dokumentation (nicht vollständig [IV: 19]), einem Support und Wiki zurückgreifen, die über die Homepage oder das Werkzeug aufrufbar sind. Selenium ist ein Open-Source-Produkt und wird regelmäßig aktualisiert. Der Nutzer wird allerdings auf die Funktionalitäten von JavaScript eingeschränkt, wodurch zum Beispiel keine alert-Fenster geschlossen werden können. [Vergleiche LV: 02, IV: 17 – 20]

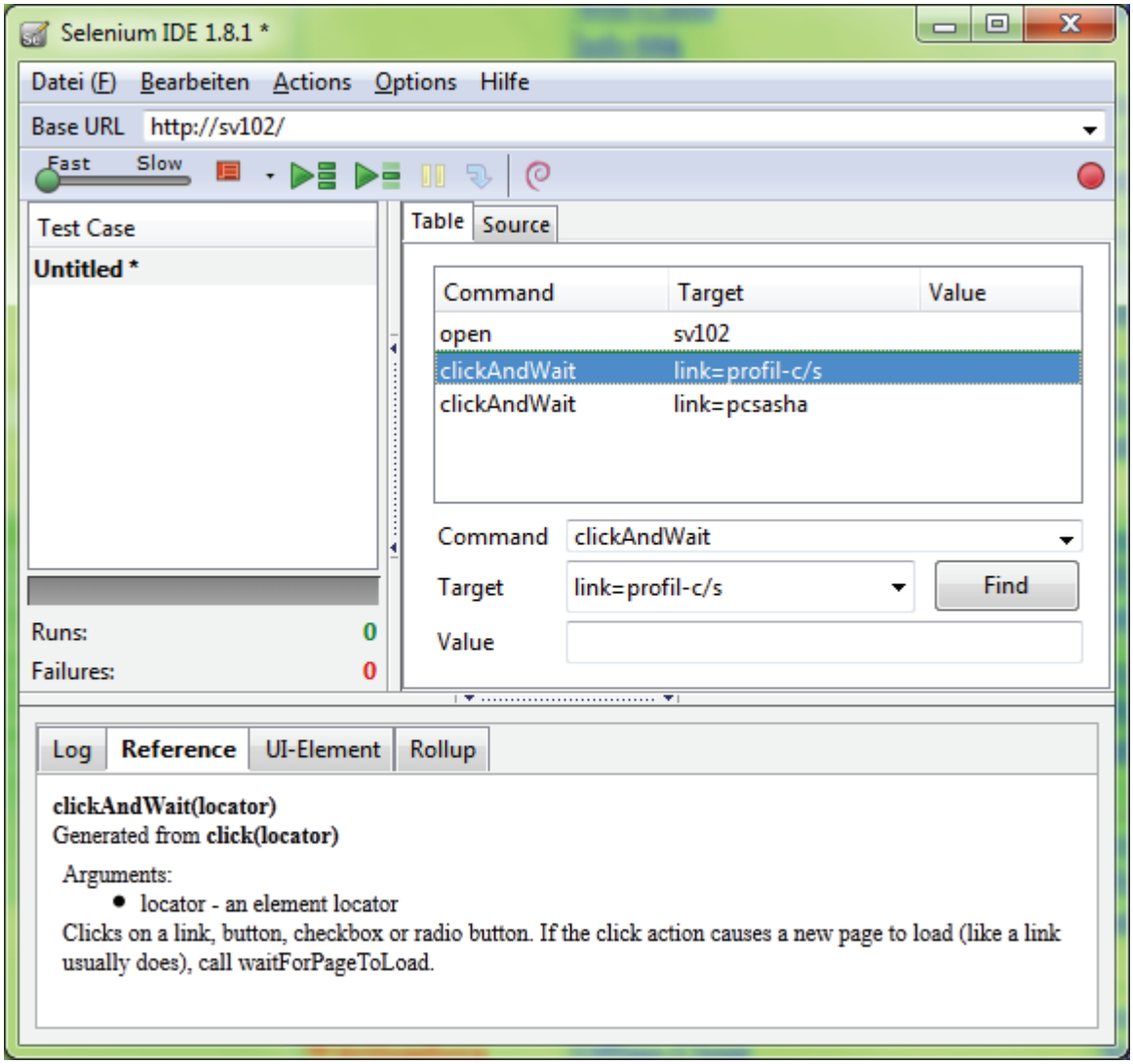



**Abb. 22: GUI von Selenium IDE**

## B.3 Canoo WebTest 3.0

Canoo WebTest ist ein Open-Source-Testwerkzeug, um für Webanwendungen automatisierte Tests zu erstellen. Es verfügt über einen ausführlichen Support, also Dokumentation, Wiki, Mail-Liste, Nachrichtenarchiv. Auch eine Report- und Capture/Replay-Funktion ist vorhanden. Die aufgezeichneten Skripte werden in XML oder Groovy gespeichert.
Durch die Java-Bibliothek Apache ANT ist Canoo WebTest plattformunabhängig, braucht aber für die Ausführung das Java Development Kid. Das Testwerkzeug basiert auf HTMLUnit und HTMLUnit.
Durch die Trennung von Daten und Skripten steigt die Erweiterbarkeit des Tools.
Die Software wird stets erweitert, wobei die letzte Version im April 2012 erschien. [Vergleiche LV: 02, IV: 21 – 24 ]

webtest
WebTest Test Report  Tests started at Thu Sep 06 15:48:04 CEST 2007

Result Summary

| WebTests | # | % | Graph |
|---|---|---|---|
| ✓ | 1 | 100 | |
| ✗ | 0 | 0 | |
| Sum | 1 | 100 | |

| Steps | # | % | Graph |
|---|---|---|---|
| ✓ | 5 | 100 | |
| ✗ | 0 | 0 | |
| ○ | 0 | 0 | |
| Sum | 5 | 100 | |

Server Roundtrip Timing Profile

| Secs | # | % | Histogram |
|---|---|---|---|
| 0 - 1 | 1 | 50 | |
| 1 - 3 | 1 | 50 | |
| 3 - 5 | 0 | 0 | |
| 5 - 10 | 0 | 0 | |
| 10 - 30 | 0 | 0 | |
| > 30 | 0 | 0 | |
| Sum | 2 | 100 | |
| Avg | | 711 ms | |

Test Scenario Overview (1431)

| # | Result | Name | # Steps | Timing profile Duration | % | Graph |
|---|---|---|---|---|---|---|
| 1 | ✓ | check that WebTest is Google's top 'WebTest' result | 5 / 5 | 1431 | 100 | |

✓ check that WebTest is Google's top 'WebTest' result

Test started at Thu Sep 06 15:48:04 2007, lasting 1431 ms.

| # | Result | Name | Parameter | Duration |
|---|---|---|---|---|
| 1 | ✓ | invoke<br>Go to Google (in English)<br>Resulting page | url http://www.google.com/ncr | 1130 |
| 2 | ✓ | verifyTitle | text Google | 3 |
| 3 | ✓ | setInputField | name q<br>value WebTest | 6 |
| 4 | ✓ | clickButton<br>Resulting page | label I'm Feeling Lucky | 292 |
| 5 | ✓ | verifyTitle | text Canoo WebTest Homepage | 0 |

Back to Test Report Overview

Created using Canoo Webtest (R_1603). Report created at 06.09.2007 15:48

webtest is an Open Source project founded and hosted by canoo

**Abb. 23: Protokoll eines Testdurchlaufes mit Canoo WebTest**

# C CD-Inhalt

Auf der CD, die sich auf der letzten Seite im Cover befindet, sind Dateien in folgender Gliederung zu finden:

- Bachelorarbeit
  - bachelorarbeit.pdf
  - abstract.pdf
  - Website
    - bachelorarbeit_de.htm
    - bachelorarbeit_en.htm
    - ba.pdf *Bachelorarbeit, identisch mit der Datei im übergeordnetem Verzeichnis*
    - images
- Sikuli
  - howToSikuli.pdf
  - read_me.txt
  - Sikuli-r930-win32 *Programm, Version X r930 für Windows*
  - Testumgebung *Screenshots, mit denen die Tests nachgeahmt werden können*
  - Skripte[5]
    - gisMapCheck.sikuli
    - pictureTest.sikuli
    - featureTopLeft.sikuli
    - featureBaseCheck.sikuli
    - labelCountCheck.sikuli
    - images

[5] siehe Anhang F, Seite XII ff.

# Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Neubrandenburg, den 17.08.2012

_____

Benjamin Naedler

# Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei der Erstellung und Bearbeitung unterstützt und motiviert haben.

Ich bedanke mich bei Herrn Prof. Dr. Andreas Wehrenpfennig, der mich in der Zeit der Entstehung der Arbeit betreut und beraten hat.

Ein besonderer Dank gilt auch der Betreuung durch die deg, insbesondere Frau Dr. Malve Hauer und Herrn Dipl. Inf. Robert Theel. Ihr Einsatz und ihre kritische Meinung halfen mir, die Arbeit in diesem Rahmen, Umfang und Ausfertigung fertigzustellen.

Auch bei meiner Familie und meiner Freundin möchte ich mich für die Ausdauer, Geduld und den Rückhalt bedanken.