Projektarbeit 1

Evaluation Swing-kompatibler Capture-and-Replay-Tools

DANIEL KRAUS

Hochschule Karlsruhe – Technik und Wirtschaft

Fakultät für Informatik und Wirtschaftsinformatik

Matrikelnummer:	54536
Referent:	Prof. Dr.-Ing. Holger Vogelsang

Zusammenfassung

Capture and Replay stellt eine effiziente Möglichkeit dar, automatisiert ablaufende Regressionstests zu erstellen. Die so generierten Tests gelten jedoch als sehr fragil und erfordern meist einen hohen Pflegeaufwand. Ein großes Problem hierbei ist die Wiedererkennung von Komponenten der grafischen Benutzungsoberfläche. Häufig wird auf manuell erstellte Testskripte und aufwendige Patterns wie Page Objects zurückgegriffen, um den Einsatz von Assets zu reduzieren. Skripte erfordern allerdings Programmierkenntnisse, was das Testen durch Domänenexperten erschwert. So entstehen auch hier Mehrkosten, u. a. durch den erhöhten Kommunikationsbedarf. Mit dem Ziel, den Capture-and-Replay-Ansatz auf mögliche Stärken und Schwächen zu untersuchen, werden vier kommerzielle sowie quelloffene Tools für die Java-Oberflächentechnologie Swing evaluiert. Zu diesem Zweck wird ein entsprechendes Testszenario konzipiert, das typische Defizite solcher Tools adressiert und der Simulation alltäglicher Entwicklungsprozesse dient. Diese Arbeit zeigt, dass die Ergebnisse zum Teil stark variieren und welche Vor- sowie Nachteile Capture and Replay gegenüber manuell erstellten Testskripten bietet.

Stichwörter: Java, GUI, Swing, Capture and Replay

1 Einleitung

Die Entwicklung von Software ist sehr dynamisch, d. h. die zugrunde liegende Codebasis wird kontinuierlich verändert. Diese Änderungen dürfen die Korrektheit der Anwendung in keinster Weise negativ beeinflussen. Konsequentes Testen stellt daher eine unerlässliche Tätigkeit dar, die maßgeblich über den Erfolg oder Misserfolg von Software entscheidet. Effektive Tests sowie eine hohe Testabdeckung verringern das Risiko von Fehlern und unerwünschten Seiteneffekten, die bei Anwendern rasch auf Frustration stoßen oder gar zum Boykott einer Anwendung führen können. Bei modernen Entwicklungsmethoden wie etwa Test-Driven Development (TDD), werden die Tests noch vor der Implementierung der zu testenden Komponenten angefertigt. Martin[1] zufolge sind Tests sogar wichtiger als die eigentliche Anwendung, da sie eine exakte Spezifikation dieser darstellen: „You can (and do) create the system from the tests, but you can't create the tests from the system" [Mar13].

Doch was sind gute Tests? Neben implementierungstechnischen Details stellt sich auch die Frage, welche Bausteine in welchem Umfang adressiert werden sollen. Eine mögliche Antwort hierauf liefert die von Cohn[2] eingeführte *Test-Pyramide* (siehe Abbildung 1). Demnach sind automatisierte und isolierte Low-Level-Tests zu bevorzugen: „This way tests can run fast, be independent (from each other and shared components) and thereby stable" [and13, S. 42]. Hierfür stehen im Java-Umfeld zahlreiche Frameworks zur Verfügung, wie beispielsweise JUnit[3] (Automatisierung) und Mockito[4] (Isolation). Nichtsdestotrotz müssen auch die oberen Schichten verifiziert werden, besonders kritisch ist dabei das Testen des *Graphical User Interface (GUI)*. Die GUI stellt oftmals die einzigste Schnittstelle zwischen dem Anwender und der Software dar, wodurch ihre korrekte Funktionsweise die subjektiv empfundene Qualität stark beeinflusst. Das Testen der GUI gilt jedoch als sehr mühsam, Fowler bezeichnet dies im Allgemeinen als „brittle, expensive to write, and time consuming to run" [Fow12]. Zum einen führen bereits kleine Eingabemasken zu zahlreichen Kombinationsmöglichkeiten, was das Erstellen adäquater Tests erschwert, zum anderen entscheidet in der Regel die Sequenz dieser Eingaben darüber, welche Funktionen in der GUI überhaupt getestet werden können.

[1] Robert C. Martin, auch bekannt als „Uncle Bob", ist ein US-amerikanischer Softwareentwickler und Initiator des agilen Manifests – dem Fundament der heutigen agilen Vorgehensmodelle. [Hig01]

[2] Mike Cohn ist ebenfalls US-amerikanischer Softwareentwickler und gilt, u. a. mit Jeff Sutherland und Ken Schwaber, als einer der Erfinder des agilen Vorgehensmodells Scrum. [Den12]

[3] `http://junit.org`.

[4] `http://mockito.org`.

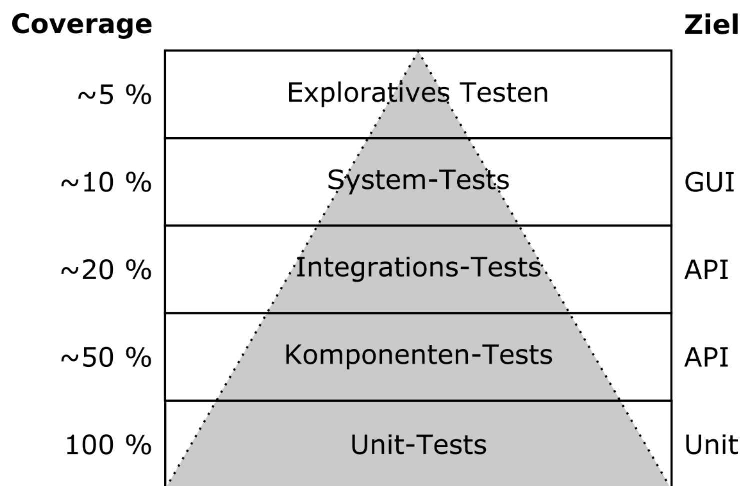

Abbildung 1: Test-Pyramide nach Cohn [and13, S. 60]

Insbesondere Regressionstests führen hierbei zu Problemen. Beim Regressionstesten werden Testfälle fortlaufend wiederholt, sodass Änderungen keine neuen Fehler – die Regressionen – in bereits getesteten Komponenten hervorrufen bzw. um diese frühzeitig zu entdecken. Ändert sich die GUI, kann sich jedoch der Ausführungspfad (ergo die Sequenz der Eingaben) eines solchen Tests ebenso ändern, wenn z. B. die Position einer GUI-Komponente modifiziert wurde. Tests müssen daher immer wieder erstellt bzw. angepasst, durchgeführt und ausgewertet werden.

Ein weit verbreiteter Ansatz um dies zu automatisieren ist *Capture and Replay*. Die Tests werden hierfür einmalig manuell durchgeführt und dabei aufgezeichnet (Capture), anschließend können diese beliebig oft und vollautomatisch wiederholt werden (Replay). Für gewöhnlich „lauscht" das Tool zu diesem Zweck auf verschiedene *Events*, wie etwa Mausklicks oder Tastatureingaben, und erstellt auf dieser Basis ein *Testskript*. Dies hat den Vorteil, dass Tests rasch erstellt werden können und keine Programmierkenntnisse voraussetzen, wodurch sich solche Tools besonders für Domänenexperten eignen. Dennoch entstehen auch hier Probleme bei der Durchführung der Regressionstest:

> [...] these test cases often cause difficulty during software maintenance and regression testing, because relatively minor changes to the GUI can cause a test case to break, or, cease to be executable against an updated version of the software. When such a situation occurs, a large manual effort is often required to repair some subset of the cases in a test suite, or worse yet; [sic] [MM09, S. 1]

In dieser Arbeit werden vier kommerzielle sowie quelloffene Capture-and-Replay-Tools für die Oberflächentechnologie *Swing* evaluiert. Swing dient der Implementierung von Rich Clients[5] mit Java, zwar gilt die Technologie als veraltet, nichtsdestotrotz wird sie noch in zahlreichen Anwendungen eingesetzt und ist in puncto Popularität nahezu gleichauf mit dem Nachfolger JavaFX [Col15].

Zunächst wird in Abschnitt 2 der Capture-and-Replay-Ansatz im Detail erläutert, in Abschnitt 3 findet sich eine Einführung in Java Swing. Abschnitt 4 beinhaltet die eigentliche Evaluation und beginnt mit der Beschreibung des Testszenarios (4.1). Anschließend werden die getesteten Tools aufgeführt und kurz beschrieben (4.2), woraufhin die Präsentation der Evaluationsergebnisse erfolgt (4.3). Im letzten Teil der Arbeit, Abschnitt 5, werden diese Ergebnisse interpretiert und kritisch bewertet.

2 Capture and Replay

Wie im vorherigen Abschnitt bereits erwähnt wurde, werden in der ersten Phase von Capture and Replay die gewünschten Testfälle manuell ausgeführt und hierbei aufgezeichnet. Die zweite Phase dient der vollautomatischen Wiederholung der so erstellen Testskripte, mithilfe derer die zu testende Applikation auf mögliche Regressionen untersucht wird. Die durch das Tool erstellten Skripte basieren in der Regel auf Events, welche Nutzeraktionen repräsentieren. Solche Events werden in Java üblicherweise als Spezialisierung von `EventObject` dargestellt, wie z. B. Objektinstanzen der Klasse `MouseEvent`. Die *Event Queue* des Oberflächen-Toolkits verwaltet die jeweiligen Event-Objekte und reicht diese an den GUI-Thread weiter. Diese sequenzielle Verarbeitung führt letztendlich zur Interaktion zwischen dem Benutzer und der Anwendung.

Unglücklicherweise genügt es nicht, ausschließlich die Event Queue zu überwachen, um konsistente und robuste Testskripte zu erhalten. [Ada+11, S. 803 f.] Swing bzw. das Abstract Window Toolkit (AWT) unterteilen Events grundsätzlich in zwei Kategorien: *Low-Level Events* sowie *semantische Events*. Erstere werden durch Maus- oder Tastatureingaben ausgelöst, hingegen können semantische Events auch durch die Applikation selbst erstellt werden. Beispielsweise versendet ein `JTextField` ein sogenanntes `CaretEvent`, wenn der darüberliegende Mauszeiger bewegt wird, um etwaige Listener zu benachrichtigen. Solche Events werden für gewöhnlich nicht über die Event Queue transportiert, ergo erfordert die Aufzeichnung vollständiger Tests die Überwachung weiterer Mechanismen. Aber auch einfache Low-Level Events bergen

[5] Bezeichnet eine Anwendung, bei der Geschäftslogik sowie GUI clientseitig implementiert sind.

Risiken: So beinhaltet etwa ein `MouseEvent` u. a. die Koordinaten des Mauszeigers. Werden diese Informationen beim Abspielen dogmatisch übernommen, führen bereits kleine Änderung am GUI-Layout gegebenenfalls dazu, dass die Zielkomponente nicht adressiert wird und die Ausführung dadurch fehlschlägt.

Hinzu kommt, dass Event-getriebene Anwendungen – dem vorherrschenden Paradigma bei der Programmierung von GUIs – nicht nur auf Nutzeraktionen reagieren, sondern immer häufiger auch durch Aktionen des Betriebssystem-Timers beeinflusst werden. Im Kontext von Capture and Replay wird dies als *Temporal Synchronization Problem* bezeichnet und betrifft in erster Linie animierte sowie zeitabhängige GUI-Komponenten. [Ada+11, S. 821 f.] Typisch ist hier etwa die Auswahl eines Datums in einem Kalender, da dessen Zustand (z. B. der vorausgewählte Monat) meist vom Zeitpunkt der Ausführung abhängig ist. Wird ein entsprechender Test aufgezeichnet und zu einem anderem Zeitpunkt erneut abgespielt, können der Zustand des Kalenders während der Capture- und der Replay-Phase variieren, wodurch auch hier der Test unter Umständen zu einem falschen Ergebnis führt.

Daneben bereitet oftmals die Wiedererkennung von Komponenten große Schwierigkeiten. Die Formalisierung dieser Problematik wurde erstmals in [MM09] durchgeführt. Hierbei wird ein separates Fenster innerhalb einer Anwendung als Menge W bezeichnet, welche diverse ausführbare GUI-Komponenten $\{e_1, \ldots, e_n\}$ beinhaltet. $W' = \{e'_1, \ldots, e'_m\}$ stellt eine Folgeversion von W dar, wobei möglicherweise $n \neq m$ gilt. Das *GUI Element Identification Problem* ist nun wie folgt definiert:

> For each actionable GUI element e_i in W, find a corresponding (possibly modified) element e'_j in W' whose actions implement the same functionality. [MM09, S. 3]

Hierzu muss jedes $e \in W \cup W'$ einer der folgenden drei Mengen zugeordnet werden:

Deleted Elemente aus W mit keinen korrespondierenden Elementen in W', ergo wurden diese in der Folgeversion entfernt.

$$D = \{d \mid d \in W \land \neg\exists e'_j \in W' : e'_j \equiv d\}$$

Created Elemente aus W' ohne entsprechende Zuordnung in W, folglich wurden diese in der neuen Version hinzugefügt.

$$C = \{c \mid c \in W' \land \neg\exists e_i \in W : e_i \equiv c\}$$

Maintained Elemente aus W die nach wie vor in W' enthalten sind, jedoch unter Umständen modifiziert wurden.

$$M = \{m \mid m \in W \land \exists e'_j \in W' : e'_j \equiv m\}$$

Nutzt ein Test Elemente aus D, bedarf es der Anpassung des Skriptes bei neuen Versionen der GUI.[6] Bei der Verwendung von Elementen aus C muss entschieden werden, inwieweit diese bestehende Skripte beeinflussen können (beispielsweise durch starke Veränderung des Layouts) und ob weitere Tests notwendig sind, welche die entsprechenden Komponenten adressieren. Demnach ist für das GUI Element Identification Problem insbesondere die Berechnung der Menge M relevant, um für jedes vorhandene m eine korrekte Zuordnung zu einem Element e'_j zu finden. Hierzu sind spezielle Heuristiken notwendig, die durch eine geringe Fehlerwahrscheinlichkeit möglichst genaue Ergebnisse erzielen, ohne die Ausführungsgeschwindigkeit der Tests spürbar zu beeinträchtigen.[7]

Capture and Replay erfordert daher zum einen eine effektive Strategie zur Selektion relevanter Events, um die Konsistenz von Testskripten sicherzustellen. Zum anderen bedarf es einer zuverlässigen Komponentenerkennung, damit aufgezeichnete Tests dauerhaft ein zufriedenstellendes Maß an Robustheit bieten. Beherrscht ein Tool diese Fähigkeiten, erlaubt es dem Tester den Fokus auf seine eigentliche Aufgabe zu richten, statt fortlaufend obsolete Testskripte anzupassen, und ermöglicht die GUI-Testautomatisierung.

3 Java Swing

Die erste Java-API zur Implementierung grafischer Benutzungsoberflächen war AWT. Eine Besonderheit von AWT ist die Verwendung sogenannter Peer-Klassen, welche als Schnittstelle zwischen Java- und nativem Code fungieren. Alle dargestellten Komponenten werden mithilfe dieser Peers vom zugrunde liegenden Betriebssystem bezogen, wodurch sich die GUI wie andere native Programme verhält und sich auch dem Aussehen der Plattform fügt. Allerdings führt dies zu einem signifikantem Nachteil: Da jede Komponente auf allen Betriebssystemen zur Verfügung stehen muss, ist deren Aus-

[6] Dies erfolgt in der Regel manuell, es existieren jedoch auch wenige automatisierte Ansätze wie z. B. [Mem08] oder [HCM10]

[7] An dieser Stelle sei auf [Ngu+14] sowie [Ste+00] verwiesen, welche am Beispiel der Tools GUITAR bzw. jRapture eine entsprechende Einführung bieten.

wahl stark beschränkt, weshalb sich mit AWT lediglich rudimentäre GUIs realisieren lassen.[8] Durch die unmittelbare Anforderung von Betriebssystemressourcen, deren Speicherverwaltung nicht unter Obhut der Java Garbage Collection erfolgt, werden Oberflächen-Toolkits wie AWT auch als *Heavyweight Toolkits* bezeichnet.

Angesichts dieser Defizite kündigte der damalige Java-Eigentümer Sun im April 1997 die *Java Foundation Classes (JFC)* an. Obwohl Swing lediglich die enthaltenen GUI-Komponenten bezeichnet, wird der Name häufig synonym für JFC verwendet. Seit der Release von Java 1.2, Ende 1998, ist die Klassenbibliothek fester Bestandteil des Java Development Kit (JDK) und bietet neben Swing folgende weitere Funktionen:

- Pluggable Look and Feel: Das gesamte Look and Feel[9] der Anwendung kann – zur Laufzeit und unabhängig vom Betriebssystem – ausgetauscht werden. Mithilfe des `synth`-Pakets können zusätzlich eigene Look and Feels kreiert werden.

- Accessibility API: Ermöglicht die Anbindung weiterer Interaktionsmöglichkeiten zur Unterstützung von Menschen mit Behinderungen, wie z. B. Screenreader, die Verwendung einer Bildschirmlupe oder Sprachsteuerung.

- Java 2D API: Dient der Erstellung und Transformation hochauflösender, vektorbasierter 2D-Objekte sowie von Text und Rastergrafiken. Ähnlich PostScript, erlaubt eine spezielle (interne) Seitenbeschreibungssprache zudem die Ansteuerung von Druckausgabegeräten.

- Internationalization: Handhabung unterschiedlichster Text- und Zeichenformate wie etwa Arabisch oder Chinesisch. Des Weiteren ist die komplexe Lokalisierung von Anwendungen möglich, d. h. beispielsweise die Berücksichtigung besonderer Datums- sowie Zeitformate und den Einsatz von Währungen.

Trotz all dieser weiteren Features gilt Swing als wichtigster und populärster Bestandteil von JFC. Im Gegensatz zu AWT wird Swing als *Lightweight Toolkit* bezeichnet, da es keine Peers verwendet und alle Komponenten durch primitive Zeichenoperationen erzeugt. Dies erlaubt weitaus flexiblere GUIs ohne weitreichende Restriktionen durch die unterstützten Betriebssysteme, transparente Komponenten sowie ein konsistentes Look and Feel über Plattformgrenzen hinweg. Zwangsläufig führt dies jedoch auch

[8] Aufgrund dessen wurde das „Abstract“ in AWT innerhalb der Community oftmals durch Adjektive wie „Annoying“, „Awful“ oder „Awkward“ ersetzt.

[9] Als Look and Feel werden standardisierte Designaspekte bei der Entwicklung von GUIs bezeichnet. Dies betrifft beispielsweise die Auswahl von Farben, Schriftfonts sowie Animationen.

dazu, dass nicht alle vorhandenen Möglichkeiten ausgeschöpft werden, hierzu zählen u. a. hardwarebeschleunigte Grafikoperationen und Antialiasing.

Als Fenster und zentraler Container dient die Klasse `JFrame`, die `java.awt.Frame` erweitert. Hier und an vielen anderen Stellen bedient sich Swing zahlreicher AWT-Klassen und erweitert diese. So nutzt beispielsweise jede GUI-Komponente die Basisklasse `JComponent`, die ihrerseits eine Spezialisierung von `java.awt.Component` ist. Ein `JFrame`, ergo ein Fenster, besteht aus unterschiedlichen Ebenen, deren Fundament die *Root Pane* darstellt. Sie ist die Wurzel der Container-Hierarchie, über die auf alle weiteren Schichten zugegriffen wird. Die nächste Ebene bildet die *Layered Pane*, die dazu dient, den Fensterinhalten eine z-Koordinate zuzuweisen, um deren Zeichenreihenfolge zu bestimmen. Darauf folgen die *Content Pane*, welche die eigentlichen GUI-Komponenten beinhaltet, sowie eine optionale Menüleiste (`JMenuBar`). Bedarfsweise kann als oberste Schicht eine *Glass Pane* angebracht werden, mit deren Hilfe sich z. B. Events filtern lassen oder auf diese gesondert reagiert werden kann.

Bereits in Abschnitt 2 wurden die unterschiedlichen Event-Kategorien erläutert, auf eine Vertiefung dieser Thematik im Sinne von Event Handling wird an dieser Stelle verzichtet. Zum Einstieg in Swing und zur Behandlung weiterführender Themen eignet sich hervorragend das offizielle Tutorial seitens Oracle[10].

4 Evaluation

Im folgenden Abschnitt wird zunächst das Testszenario erläutert, auf dessen Basis die Evaluation durchgeführt wird. Danach erfolgt eine Kurzbeschreibung der ausgewählten Tools sowie abschließend die Präsentation der Evaluationsergebnisse.

4.1 Testszenario

Wie bereits in Abschnitt 2 erwähnt, führen das GUI Element Identification Problem sowie das Temporal Synchronization Problem häufig zu Komplikationen bei der Anwendung von Capture and Replay und nicht zuletzt zu einer kontroversen Reputation: „Record-playback tools are almost always a bad idea for any kind of automation, since they resist changeability and obstruct useful abstractions“ [Fow12]. Die Fähigkeit eines Tools Testskripte auszuführen, die auf einem obsoleten Stand der Software basieren (d. h. bestmöglich beide Probleme lösen), ist somit ein entscheidendes Qualitätskrite-

[10] `http://docs.oracle.com/javase/tutorial/uiswing`.

rium. Der Fokus dieser Arbeit liegt daher auf der Komponentenerkennung und der daraus resultierenden Robustheit eines Tools. Zu diesem Zweck wird das Testszenario in folgende vier Schwierigkeitsstufen unterteilt:

Stufe 0 Initial wird überprüft, welche Swing-Komponenten grundsätzlich aufgezeichnet und abgespielt werden können. Hierzu werden verschiedene ausgewählte Komponenten und Funktionen klassifiziert sowie getestet:

- Data: `JTable`, `JTree`, `JList`.
- Controls: `JButton`, `JCheckBox`, `JRadioButton`, `JComboBox`, `JSlider`, `JSpinner`.
- Text: `JFormattedTextField`, `JHistoryTextField`, `JPasswordField`, `JEditorPane`.
- Choosers: `JFileChooser`, `JColorChooser`, `JOptionPane`.
- Misc: Drittanbieter-Komponenten, Drag and Drop, asynchrone Prozesse.

Stufe 1 Die erste Stufe befasst sich mit den *externen* Eigenschaften von Komponenten, d. h. Eigenschaften, die das GUI-Element zwar beeinflussen, jedoch von außen gesteuert werden können (z. B. das Look and Feel oder das Layout).

Stufe 2 In der zweiten Stufe der Evaluation werden die *internen* Eigenschaften von Komponenten betrachtet. Ergo Merkmale, die grundsätzlich über die Komponente selbst verändert werden, wie etwa die Größe, die Farbe oder ein Label.

Stufe 3 Die dritte und letzte Stufe kombiniert die zuvor eingeführten Anpassungen. Beispiel: Innerhalb einer Änderung wird eine Komponente in einem zusätzlichen Layout-Container verschachtelt, umbenannt und dessen Größe modifiziert.

Diese unterschiedlichen Stufen erlauben bei Fehlschlägen nicht nur Rückschlüsse auf die Strategie bei der Wiedererkennung von Komponenten, sondern dienen auch der Simulation alltäglicher Entwicklungsprozesse und der damit verbundenen Probleme.

Als *System Under Test (SUT)* dient die von Oracle bereitgestellte *SwingSet3*-Demo[11]. Hierbei handelt es sich um eine offizielle Beispielanwendung, die einen Großteil der verfügbaren Swing-Funktionen und -Komponenten beinhaltet, ähnlich dem heutigen Ensemble[12] für JavaFX. Wie Abbildung 2 zeigt, finden sich dort bereits die Klassen mit den zugehörigen Komponenten aus Stufe 0. Für `JFileChooser` sowie Drag

[11] `https://java.net/projects/swingset3`.
[12] `http://oracle.com/technetwork/java/javase/overview/javafx-samples-2158687.html`.

Abbildung 2: Screenshot der SwingSet3-Oberfläche

and Drop existieren separate Minianwendungen, als Drittanbieter-Komponente wird die Datumsauswahl `JDatePicker`[13] in SwingSet3 integriert. Zum Test asynchroner Prozesse eignet sich die vorhandene `JProgressBarDemo`, welche zusätzlich mit einer variierenden Ladezeit versehen wird.

Auf dieser Basis werden nun die Stufen 1 – 3 inkrementell hinzugefügt, um so die Komplexität der Tests schrittweise zu erhöhen. Um dies für jedes zu testende Tool möglichst einfach und wiederholbar durchführen zu können, wird das SwingSet3-Projekt zunächst nach Git[14] portiert und via GitHub[15] veröffentlicht. Dort kann auch die Umsetzung der einzelnen Schwierigkeitsstufen auf Quellcodeebene begutachtet werden.

[13] `http://jdatepicker.org`.
[14] Im Folgenden werden Grundkenntnisse in verteilten Versionsverwaltungssystemen bzw. Git vorausgesetzt. Unter `https://git-scm.com` findet sich eine entsprechende Einführung sowie Dokumentation zur Vertiefung.
[15] `https://github.com/beatngu13/swingset3`.

Wenn Änderungen in einer lokalen Git-Repository eingecheckt werden sollen, so geschieht dies mithilfe von Commits. Jeder Commit erzeugt dabei eine neue *Revision*, die den Zustand der zugrunde liegenden Codebasis mittels SHA-1-Hashes versioniert. Das zugehörige Protokoll lässt sich mit dem `log`-Kommando ausgeben, wie Listing 1 exemplarisch zeigt. Die Parameter `--abbrev-commit` sowie `-3` sorgen dafür, dass lediglich ein eindeutiges Präfix des 20-Byte-Hashwerts angezeigt und die Ausgabe auf die letzten drei Revisions reduziert wird. Mittels `checkout` kann nun die lokale Repository auf einen beliebigen Zustand zurückversetzt werden. So führt etwa `git checkout 408a37f` dazu, dass im Beispiel die letzten beiden Änderungen (temporär) verworfen werden. Anschließend ist es möglich, den Endzustand Schritt für Schritt wiederherzustellen. Dadurch lassen sich die Änderungen am Quellcode bzw. die korrespondierenden Schwierigkeitsstufen mühelos und unter gleichen Bedingungen für jedes Tool realisieren.

```
$ git log --abbrev-commit -3
commit 406c93d
Author: Daniel Kraus <daniel.kraus@mailbox.org>
Date:   Mon Dec 14 23:40:20 2015 +0100

    Clear history button.

commit 2019c22
Author: Daniel Kraus <daniel.kraus@mailbox.org>
Date:   Thu Nov 12 12:38:33 2015 +0100

    History text field for login.

commit 408a37f
Author: Daniel Kraus <daniel.kraus@mailbox.org>
Date:   Tue Nov 10 15:13:26 2015 +0100

    Added login functionality.
```

Listing 1: Beispielausgabe von `git log`

Die Durchführung der Evaluation erfolgt auf einem Apple MacBook Pro (Retina 13″, Anfang 2015) mit 2,7 GHz Intel Core i5 und 8 GB Arbeitsspeicher sowie OS X El Capitan (10.11.2). Für Tools die ausschließlich Windows unterstützen, wird Windows 10 Education (Version 1511) als Boot-Camp-Installation eingesetzt. Zur Ausführung von Java-Anwendungen wie SwingSet3, wird das JDK in Version 8u66 verwendet.

4.2 Tools

In diesem Abschnitt erfolgt eine kurze Beschreibung der ausgewählten Capture-and-Replay-Tools in alphabetischer Reihenfolge.

Marathon
Bei Marathon (`http://marathontesting.com`) handelt es sich um ein Java-basiertes Tool, das eine freie Version des kommerziellen MarathonITE darstellt. Während das Open-Source-Produkt unter GNU Lesser General Public License (LGPL)[16] vornehmlich kleine Projekte adressiert, bietet MarathonITE verschiedene Features zur Arbeit mit komplexeren Anwendungen, wie etwa verteiltes Testen in der Cloud und eine verbesserte Wiedererkennung von Komponenten. Beide Varianten unterstützen ausschließlich Swing-GUIs und erstellen Testskripte in Ruby[17], die nach belieben manipuliert werden können. Evaluiert wird Marathon in Version 4.0.0.0.

Pounder
Pounder (`http://sourceforge.net/projects/pounder`) ist ein reines Open-Source-Tool, ebenfalls in Java implementiert und unter LGPL lizenziert. Ursprünglich entwickelt von Matthew Pekar, wird das Projekt nicht mehr aktiv gepflegt. Nichtsdestotrotz eilt Pounder ein sehr guter Ruf voraus und es wurden diverse Versuche unternommen, das Projekt zu erweitern (z. B. [Ada+11]). Pounder bietet alleinig Capture and Replay für Swing und erstellt Testskripte im XML-Format, deren manuelle Bearbeitung nicht vorgesehen ist. Leider verursacht die aktuelle Version 0.96beta eine `NullPointerException` beim Start der Anwendung, weshalb auf die letzte stabile Version 0.95 zurückgegriffen wird.

QF-Test
Das in Java implementiert Tool QF-Test (`https://qfs.de`) steht ausschließlich als kommerzielle Variante zur Verfügung. Neben GUIs auf Basis von JavaFX, Swing, AWT u. v. m., ist auch das Testen von Web-Anwendungen möglich. Zum Skripting kann wahlweise Jython[18] oder Groovy[19] verwendet werden. Darüber hinaus stellt QF-Test beispielsweise Reporting-Mechanismen und eine Integration von Build-Tools wie Jenkins bereit. Offiziell wird Mac OS X nicht unterstützt, das Testen von Swing-GUIs ist jedoch möglich. Zur Evaluation wird QF-Test in Version 4.0.9 verwendet.

---

[16] `http://gnu.org/copyleft/lgpl.html`.
[17] `https://ruby-lang.org`.
[18] `http://jython.org`.
[19] `https://groovy-lang.org`.

Ranorex
Mit Ranorex (http://ranorex.de) wird ein weiteres proprietäres Produkt evaluiert, welches das Testen von Desktop-, Web- sowie mobilen Applikation ermöglicht. Zur Ausführung von Ranorex wird Windows als Betriebssystem vorausgesetzt, allerdings können in der SUT zahlreiche Oberflächentechnologien zum Einsatz kommen. Die aufgezeichneten Tests nutzen keine Skriptsprache, sondern werden wahlweise in C#[20] oder Visual Basic .NET[21] erstellt. Nebst Capture and Replay erlaubt Ranorex beispielsweise die Einbettung in Continuous-Integration-Prozesse sowie die Anbindung an Testmanagement-Tools oder Visual Studio. Die Evaluation von Ranorex erfolgt mit Version 5.4.5.19886.

4.3 Ergebnisse

Zunächst finden sich in Tabelle 1 die Resultate aus Stufe 0. Wie zu sehen ist, unterstützen nahezu alle Tools eine Vielzahl der typischen Swing-Komponenten. Einige Besonderheiten gab es jedoch bereits hier: Beispielsweise erlaubt Pounder grundsätzlich die Benutzung von `JTable`, der Test schlug jedoch fehl, da die Pounder-Startklasse von `Component` oder `Window` erben muss. In SwingSet3 werden die Daten der Tabelle allerdings an anderer Stelle geladen, weshalb keine Tabelleninhalte zu sehen waren bzw. zusätzliche Anpassungen notwendig gewesen wären. Pounder stellt hierfür die Schnittstelle `ComponentConduit` zur Verfügung. Da dies bei großen Enterprise-Anwendungen mit vielen internen und externen Abhängigkeiten einen entsprechend hohen Anpassungsaufwand erfordert, scheint Pounder eher ungeeignet. Nicht zuletzt hatte Pounder als einziges Tool große Schwierigkeiten mit Animationen (Temporal Synchronization Problem), wie etwa beim Test des `JColorChooser`. Aber auch Marathon erforderte an diversen Stellen die Aktivierung eines sogenannten „Row Recording Mode“, um alle durchgeführten Aktionen tatsächlich aufzuzeichnen. Dieser Modus arbeitet allerdings mit absoluten Koordinaten, wodurch die so generierten Tests entsprechend fragil sind.

Beim Testen von Texteingabefeldern traten ebenfalls diverse Probleme auf. Zwar unterstützen die getesteten Tools die selektierten Komponenten, jedoch ist die Art und Weise, wie der vorhandene Text manipuliert wird, entscheidend. So kann z. B. Text per Maus markiert und anschließend verändert werden, aber auch ein einfacher Klick in das entsprechende Textfeld ist denkbar, um daraufhin per Tastatur die glei-

[20] https://wikipedia.org/wiki/C-Sharp_(programming_language).
[21] https://wikipedia.org/wiki/Visual_Basic_.NET.

chen Änderungen vorzunehmen. Hier müssen die Eigenheiten des jeweiligen Tools berücksichtigt werden, um sicherzustellen, dass alle Nutzeraktionen korrekt erfasst werden und die Änderungen am Text zum gewünschten Ergebnis führen.

Die Kompatibilität zu Drittanbieter-Komponenten ist aufgrund des `JDatePicker`-Tests keineswegs per se repräsentativ, zumal dies auch stark von der Implementierung der Komponente abhängig ist. Er zeigt jedoch, wie gut oder schlecht ein Tool mit unbekannten Inhalten zurecht kommt. Aktionen via Drag and Drop wurden lediglich bei Marathon ignoriert, auch zur Unterstützung asynchroner Prozesse bieten alle Tools entsprechende Mechanismen, um auf Operationen zu warten – bei Pounder waren sogar keinerlei Anpassungen notwendig. Lediglich Ranorex markierte den Test als erfolgreich, obwohl die Aktion nicht beendet worden war. Darüber hinaus verlangsamte Ranorex deutlich die Ausführungsgeschwindigkeit von SwingSet3, was das Aufzeichnen der Tests sehr mühsam gestaltete.

		Marathon	**Pounder**	**QF-Test**	**Ranorex**
Data	`JTable`	●		●	●
	`JTree`	●	●	●	●
	`JList`	●	●	●	●
Controls	`JButton`	●	●	●	●
	`JCheckbox`	●	●	●	●
	`JRadioButton`	●	●	●	●
	`JComboBox`	●	●	●	●
	`JSlider`	●	●	●	●
	`JSpinner`	●		●	
Text	`JFormattedTextField`	●	●		
	`JHistoryTextField`		●	●	●
	`JPasswordField`	●	●	●	●
	`JEditorPane`	●	●		
Choosers	`JFileChooser`	●	●	●	●
	`JColorChooser`	●		●	●
	`JOptionPane`	●		●	●
Misc	Drittanbieter		●	●	
	Drag and Drop		●	●	●
	Asynchronität	●	●	●	

Tabelle 1: Ergebnisübersicht aus Stufe 0

Tabelle 2 zeigt die Ergebnisse der Stufen 1 – 3, wobei die Zeilen mit den Hashwerten der Git-Commits korrespondieren. Jeder dieser Commits stellt dabei einen separaten

Test dar, der einer bestimmten Stufe des Testszenarios zugeordnet ist.[22] Überraschend ist der Vergleich zwischen Pounder und Ranorex: Hier schneidet das kostenfreie und quelloffene Tool gar besser ab. Nichtsdestotrotz sind die beiden Resultate mangelhaft, da bereits geringfügige Änderungen der GUI, Anpassungen der Testskripte erfordern. Auch die Tests `302063e` sowie `9cad753` konnten in beiden Fällen nicht durchgeführt werden: Unter Mac OS X befindet sich die Menüleiste typischerweise fest am oberen Bildschirmrand und ist nicht in das Fenster der Applikation integriert. In den zuvor genannten Tests wurde das Look and Feel der Anwendung angepasst, um ein solches Verhalten zu erzwingen. Im Falle von Pounder war dieser Test nicht möglich, da die SwingSet3-Starklasse die Anforderungen von Pounder nicht erfüllte und somit stets nur einzelne Fenster getestet werden konnten. Hingegen unterstützt Ranorex lediglich Windows als Betriebssystem, wodurch der Test an dieser Stelle keinen Sinn ergibt.

		Marathon	**Pounder**	**QF-Test**	**Ranorex**
Stufe 1	`302063e`	●	n. a.	●	n. a.
	`83b61c5`	●	●	●	●
	`34a273f`	●		●	●
	`8a8165f`	●		●	
Stufe 2	`ee19457`				
	`978d746`		●		
	`bd26216`		●	●	
	`ebfaa99`	●		●	
	`53a2622`	●		●	
Stufe 3	`9cad753`	●	n. a.	●	n. a.
	`09eb2c3`			●	●
	`d343e1e`		●		
	`23e447d`				
	`9f10afe`	●		●	

Tabelle 2: Ergebnisübersicht der Stufen 1 – 3

Marathon und QF-Test erzielten demgegenüber weitaus bessere Ergebnisse. Änderungen an externen Eigenschaften von Komponenten verursachten keinerlei Probleme und die entsprechenden Tests liefen weiterhin stabil. Wurden interne Eigenschaften verändert, führte das Modifizieren der `name`-Property[23] bei allen Tools zum nicht Erken-

[22] Auch hier sei auf GitHub verwiesen, die zugehörigen Kommentare finden sich unter `https://github.com/beatngu13/swingset3/commits/master`.

[23] Die AWT-Basisklasse nahezu aller Swing-Komponenten – `Component` – besitzt ein Attribut `name`, welches bedarfsweise mit einem Wert belegt werden kann. Zu berücksichtigen ist hierbei, dass das Toolkit die Eindeutigkeit dieser Namen nicht voraussetzt.

nen der Komponente. Aber auch das Anpassen von Labels oder Icons bereitete Schwierigkeiten. Dies lässt sich darauf zurückführen, dass viele Tools eine Kombination aus Name, Typ und Hierarchie (Aufbau der GUI) zur Wiedererkennung nutzen. Einerseits garantiert dies ein deterministisches Verhalten, andererseits treten Probleme auf, wenn beispielsweise Namen dynamisch vergeben werden oder die Komponentenhierarchie stark verändert wird. Hier bietet QF-Test gute Konfigurationsmöglichkeiten, um z. B. auch anhand von Wahrscheinlichkeiten Komponenten zu erraten. Zu erwähnen sei, dass bei allen Tools aus Zeitgründen lediglich die Standardeinstellungen verwendet wurden. Dies zeigt jedoch, ob die Einhaltung von Konventionen den Konfigurationsaufwand reduziert (Convention over Configuration).

Kombinatorische Änderungen aus Stufe 3 führten zu ähnlichen Ergebnissen wie die Tests der 2. Stufe: Waren Änderungen an Namen, Icons oder Labels vorhanden, so wurden die entsprechenden Komponenten in den seltensten Fällen wiedererkannt. Auch hier war QF-Test geringfügig besser als Marathon.

5 Fazit und Ausblick

Die Ergebnisse dieser Arbeit haben gezeigt, dass die Effektivität von Capture-and-Replay-Tools zum Teil stark variiert. Im Gegensatz zu einschlägigen Publikationen wie [NB13] wurde nicht nur die Kompatibilität gegenüber GUI-Komponenten bzw. den entsprechenden Events überprüft, sondern auch das Verhalten der Tools in ihrer eigentlich Disziplin evaluiert – dem Regressionstesten. Mit Pounder, Marathon, QF-Test sowie Ranorex wurden vier Tools ausgewählt, die unterschiedliche Hintergründe besitzen: Pounder als quelloffenes und freies Tool, das zwar nicht mehr aktiv gepflegt wird, dennoch eine gute Reputation besitzt. Marathon hingegen ist nicht nur Open Source und kostenfrei verfügbar, sondern wird auch kontinuierlich durch fest abgestellte Entwickler gewartet. Mit QF-Test wurde ein proprietäres Tool evaluiert, welches einen hohen Reifegrad besitzt und zahlreiche zusätzliche Features anbietet. Ranorex besitzt ein ähnlich ausgeprägtes Profil, ist jedoch im Rahmen der Evaluation das einzigste Tool, das nicht in Java implementiert ist.

Diese Unterschiede sind allerdings auch bei der Bewertung der Ergebnisse zu berücksichtigen. Zwar geht QF-Test als das Tool hervor, welches die meisten Tests erfolgreich abschließen konnte, ist jedoch gegenüber dem Zweiplatzierten Marathon sowohl geschlossen als auch kostenpflichtig. Zumal mit MarathonITE eine erweiterte, vielversprechende Variante von Marathon zur Verfügung steht. Als Kompromiss muss

hierbei auf Open Source und kostenlose Verfügbarkeit verzichtet werden. Pounder und Ranorex, auf Platz drei bzw. vier, scheinen gänzlich ungeeignet: Pounder erfordert aufgrund seiner Einschränkungen zu starke Anpassungen zum Start und zur Verwaltung komplexer Anwendungen, Ranorex verlangsamt die Ausführungsgeschwindigkeit enorm und schneidet am schlechtesten bei den eigentlichen Tests ab. So scheint es von Vorteil, wenn SUT und Tool die identische Plattform verwenden, wie im vorliegenden Fall Java bzw. die Java Virtual Machine (JVM).

Somit liegen mit QF-Test und Marathon zwei Tools vor, die eine effizient Möglichkeit darstellen, automatisiert ablaufende Regressionstests zu generieren. Domänenexperten ohne Programmierkenntnisse können dabei eigenhändig fachliche Tests erstellen, ohne entsprechende schriftliche Spezifikationen aufzusetzen, die mühsam von Entwicklern umgesetzt werden müssen. Beide Tools bieten dennoch die Anbindung von Skriptsprachen, wodurch Capture and Replay auch in frühen Stadien der Entwicklung effizient von Programmierern eingesetzt werden kann. Darüber hinaus scheinen Anpassungen, um die zu testende Applikation für Capture-and-Replay-Tools zu optimieren (beispielsweise die Vergabe eindeutiger Namen) weniger aufwendig, wie etwa spezielle Patterns (z. B. Page Objects [Fow13]), die häufig beim Testen mit manuell erstellten Testskripten zum Einsatz kommen.

Diese These erfordert jedoch weitere empirische Studien, um die Effektivität von Capture and Replay unmittelbar mit alternativen Lösungsansätzen vergleichen zu können. Die vorliegenden Ergebnisse können bei der Auswahl eines geeigneten Tools in Betracht gezogen werden. Auch bietet das konzipierte Testszenario einen guten Ausgangspunkt, weitere Vergleiche durchzuführen. An dieser Stelle sind allerdings Verbesserungen notwendig: Die Evaluation hat gezeigt, dass diverse Tests zu stark voneinander abhängig sind. Weitere, unabhängigere Tests ermöglichen eine einfachere Analyse der eigentlichen Fehlerursache. Darüber hinaus können das Testszenario bzw. SwingSet3 dahingehend erweitert werden, typische Eigenschaften von Softwareprojekten zu übernehmen, um die Alltagstauglichkeit eines Tools oder einer alternativen Teststrategie genauer evaluieren zu können. Hier sind beispielsweise die Anbindung externer Datenquellen oder die Integration in Build-Prozesse denkbar.

Literaturverzeichnis

[Ada+11] Andrea Adamoli et al. „Automated GUI Performance Testing“. In: *Software Quality Journal* 19.4 (Dez. 2011), S. 801–839.

[and13] andrena objects AG. *Agile Software Engineer. Student Edition*. Karlsruhe: Selbstverlag, 2013.

[Col15] Stephen Colbert. *Should Oracle Spring Clean JavaFX?* Nov. 2015. URL: `https://codenameone.com/blog/should-oracle-spring-clean-javafx.html` (besucht am 11. Dez. 2015).

[Den12] Steve Denning. *The Power of Scrum*. März 2012. URL: `http://forbes.com/sites/stevedenning/2012/03/01/the-power-of-scrum` (besucht am 4. Dez. 2015).

[Fow12] Martin Fowler. *TestPyramid*. Mai 2012. URL: `http://martinfowler.com/bliki/TestPyramid.html` (besucht am 16. Nov. 2015).

[Fow13] Martin Fowler. *PageObject*. Sep. 2013. URL: `http://martinfowler.com/bliki/PageObject.html` (besucht am 12. Nov. 2015).

[HCM10] Si Huang, Myra Cohen und Atif M. Memon. „Repairing GUI Test Suites Using a Genetic Algorithm". In: *Proceedings of the 3rd IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Washington, DC (USA), 2010.

[Hig01] Jim Highsmith. *History: The Agile Manifesto*. Feb. 2001. URL: `http://agilemanifesto.org/history.html` (besucht am 8. Dez. 2015).

[Mar13] Robert C. Martin. *Test First*. Sep. 2013. URL: `https://blog.8thlight.com/uncle-bob/2013/09/23/Test-first.html` (besucht am 8. Dez. 2015).

[Mem08] Atif M. Memon. „Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing". In: *ACM Transactions on Software Engineering and Methodology* 18.2 (Nov. 2008), S. 1–36.

[MM09] Scott McMaster und Atif M. Memon. „An Extensible Heuristic-Based Framework for GUI Test Case Maintenance". In: *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Edmonton, Alberta (Canada), 2009.

[NB13] Stanislava Nedyalkova und Jorge Bernardino. „Open Source Capture and Replay Tools Comparison". In: *Proceedings of the 6th International C* Conference on Computer Science and Software Engineering*. Porto (Portugal), 2013.

[Ngu+14] Bao N. Nguyen et al. „GUITAR: An Innovative Tool for Automated Testing of GUI-driven Software". In: *Automated Software Engineering* 21.1 (März 2014), S. 65–105.

[Ste+00] John Steven et al. „jRapture: A Capture/Replay Tool for Observation-Based Testing“. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. Portland, Oregon (USA), 2000.