Aufzeichnen, Abspielen, Protokollieren

Automatisierung des Tests von Java-Swing-GUIs

Christian Glatschke

Softwaresysteme werden durch den Anwender heute meist über grafische Benutzerschnittstellen (GUI, Graphical User Interface) gesteuert. Aufgrund der vielfältigen Anwenderinteraktionen und der grafischen Kontrollelemente in der GUI sind wohldefinierte Testszenarien komplex und daher schwer umzusetzen. Solche Tests sind zeitraubend und kostenaufwändig. Kein Entwickler ist zudem erfreut, wenn er alle Interaktionen manuell austesten und dokumentieren muss, da solche Tests nicht nur monoton, sondern auch entsprechend arbeitsintensiv sind. Abhilfe schaffen hier Werkzeuge, die solche Szenarien automatisieren, so genannte GUI Capture & Replay-Werkzeuge.

„Ein weitverbreiteter Fehler, den Menschen begehen, wenn sie etwas vollkommen Narrensicheres entwickeln, ist den Einfallsreichtum vollkommener Narren zu unterschätzen."

Douglas Adams

Der Aufwand zur Automatisierung macht sich dann bezahlt, wenn bestimmte Testaufgaben ständig wiederholt und die Ergebnisse neu verifiziert werden müssen. Die Regressions- und Akzeptanztests fallen in der Produktentstehung spätestens am Ende der Entwicklung, also unmittelbar vor der Systemintegration, an. Die Gründe zur Automatisierung lassen sich wie folgt zusammenfassen:

- Reduktion des manuellen Aufwands,
- Reduktion der Testkosten,
- Verbreiterung der Testüberdeckung,
- Reproduzierbarkeit und
- Qualitätsverbesserung.

CR-Werkzeuge

GUI Capture & Replay(CR)-Werkzeuge sind für den Einsatz in der Entwicklung von Anwendersoftware konzipiert. Zur Beurteilung der Softwarequalität können diese Werkzeuge eingesetzt werden, um Testabläufe aus Benutzerschnittstellen heraus zu automatisieren, d. h. erwartete Benutzereingaben können hiermit simuliert werden. Diese Testart ist daher realitätsgetreu, sie testet die Anwendung in ähnlicher Art und Weise, wie sie auch später verwendet werden soll.

Als grundlegende Anforderung an CR-Werkzeuge können folgende vier Charakteristika spezifiziert werden:

- Capture Mode,
- Programming Mode,
- Checkpoints und
- Replay Mode.

Capture Mode (Aufzeichnung): Während der initialen Testphase zeichnet das Werkzeug alle Anwenderinteraktionen auf. CR-Werkzeuge zeichnen sich dadurch aus, dass sie objektorientiert alle Events der GUI auffangen, und eben nicht nur die X/Y-Koordinaten der aktuellen Mausposition speichern. Das Werkzeug erkennt, welches GUI-Element, z. B. Button, Menü oder Icon, angewählt wurde, und zeichnet alle Objektcharakteristika, wie den Namen des Objektes, auf.

Programming Mode: Die zuvor generierten und aufgezeichneten Testschritte werden in Skripten, in der Regel XML, gespeichert. Die Werkzeugfunktionalitäten, in Verbindung mit den Skripten, sollen es ermöglichen, auch komplexere Testszenarien zu implementieren. Einzelne Testszenarien können somit später zu einer Testsuite zusammengefasst werden.

Checkpoints: Für das Testen notwendige Messungen verifizieren, ob zuvor definierte Ergebnisse erreicht werden oder ob eventuelle Fehler auftreten. Die tatsächlichen Messergebnisse werden an den im Skript gesetzten Checkpoints realisiert. Daraus folgt, dass die fenster- und layoutspezifischen Charakteristika im direkten Zusammenhang mit den funktionalen Charakteristika getestet werden können. Checkpunkte können gesetzt werden, um:

- Zustände verschiedener GUI-Elemente,
- Datenbankinhalte oder
- Grafiken und Texte

abzufragen. Der Vorteil der Checkpunkte liegt darin, dass mit ihrer Hilfe selbst verifizierende Tests generiert und abgespielt werden können, die keiner direkten Überwachung des Testers bedürfen. Nach Ablauf der Tests werden Reports erstellt, die alle Ergebnisse auflisten.

Replay Mode (Wiederholung): Durch das wiederholte Abspielen der zuvor aufgezeichneten Skripte sind die Testszenarien zu jeder Zeit reproduzierbar. Sollte sich nun das Layout, und damit verbunden einzelne Elemente der GUI, geändert haben, so können die GUI-Elemente dennoch wiedererkannt werden, wenn der Test wiederholt wird. Java stellt ab der Version JDK 1.3 für die Simulation von AWT- und Swing-Events die Klasse **java.awt.Robot** zur Verfügung. Diese Klasse wird verwendet, um Ereignisse nativer Systemeingaben zu generieren. Diese Ereignisse können dann zur Testautomatisierung oder für selbstlaufende Demos verwendet werden, für die Maus- oder Tastaturkontrolle benötigt wird. Die Vor- und Nachteile dieser Klasse werden hier aber nicht diskutiert.

Testautomatisierung

Bevor es zu einem Test kommen kann, sollten im Vorfeld exakte Spezifikationen zu Design und Funktion der Schnittstellen erstellt werden. Ob die Anwenderbedürfnisse tatsächlich erfüllt werden, muss in einem frühen Präsentieren und Testen der grafischen Schnittstellen verifiziert werden. Das hieraus gewonnene Feedback kann somit frühzeitig, vor der Implementierung der Funktionen, im Design um-

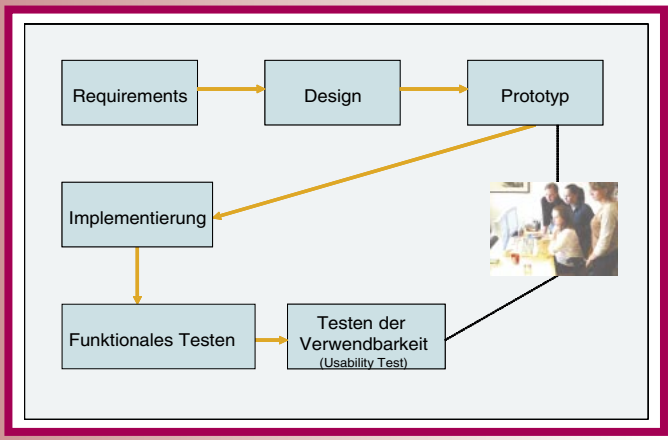

Abb. 1: Entwicklungszyklus grafischer Anwenderschnittstellen

gesetzt werden. Die Testplanung an den Anfang der Entwicklung zu stellen, ist eminent wichtig, da zum Ende der Entwicklung häufig die Zeit nicht mehr ausreichend ist, und somit die Anzahl der Tests dann reduziert wird oder das Testen eventuell sogar ganz gestrichen wird. Sollten von Kundenseite keine klaren Anforderungen und Spezifikationen vorhanden sein, so müssen die Ergebnisse der Prototypingphase in die Spezifikation einfließen.

Abbildung 1 beschreibt den Entwicklungszyklus grafischer Anwenderschnittstellen. Die funktionalen Tests sind essentiell für das GUI-Testen. Im Vordergrund steht in dieser Phase nicht mehr die Anwenderfreundlichkeit, diese sollte im Prototyping verifiziert werden, sondern vielmehr, ob alle erwarteten Funktionalitäten der Applikation erfüllt werden.

Spätestens an diesem Punkt muss entschieden werden, ob manuell oder automatisiert getestet werden soll. Während beim manuellen Testen auf dynamische Aspekte sozusagen „on the Fly" eingegangen werden kann, muss beim automatisierten Testen die Festlegung der einzelnen Schritte detailliert im Vorfeld geschehen. Da die Alternative für automatische Tests mit einem höheren Aufwand der Vorbereitung verbunden ist, soll vor der Entscheidung hierzu die Frage beantwortet werden, welche Tests es wert sind automatisiert zu werden? Bei der Beantwortung können folgende Kriterien hilfreich sein:

- Verursacht die Automatisierung höhere Kosten als der manuelle Test? Wie viel mehr?
- Automatisierte Tests haben eine begrenzte Lebensdauer. Wie lange wird das Testszenario gültig sein?
- Während der Gültigkeit des Testszenarios können weitere Fehler entdeckt werden, die in vorangegangenen Durchläufen nicht auftauchten. Wie groß ist dieser Nutzen gegenüber den Mehrkosten der Automatisierung?

Wenn zur Automatisierung ein neues Werkzeug eingesetzt wird, oder die genaue Testinfrastruktur noch gar nicht festgelegt wurde, so lässt sich der Kostenaufwand nur schwer und ungenau im Vorfeld bestimmen. Hier muss eventuell auf „Erfahrungswerte" zurückgegriffen werden.

Testmodus

Während der Aufzeichnung der Testskripte können zwei Modi unterschieden werden:

- analog und
- kontextsensitiv.

In der *analogen* Form werden alle Anwenderinteraktionen, wie Mausbewegungen und Tastatureingaben, in einem Skript protokolliert. Diese Events werden bei einem erneuten Abspielen, ohne weitere Rücksicht auf Änderungen der Systemzustände und Umgebung, akkurat nachgebildet. Obschon man auf diese Art jede Eingabe genauestens reproduzieren kann, ist diese Möglichkeit jedoch mit erheblichen Nachteilen behaftet, da schon minimalste Änderungen an Applikation oder Umgebung zu fehlerhaftem Verhalten der Skripte führen. Da in den analogen Skripten die Positionen der einzelnen GUI-Elemente festgehalten werden, würde eine Positionsänderung in der GUI dazu führen, dass das Skript das Element unweigerlich an der ursprünglichen Lage sucht, und somit in einen Fehlerzustand läuft. Auf der anderen Seite sind Nachbearbeitungen der Skripte, bezüglich der Änderungen, nahezu unmöglich bzw. nur unter erheblichem Aufwand durchzuführen.

Im Gegensatz dazu werden im *kontextsensitiven* Modus alle Interaktionen an dem betreffenden GUI-Element erfasst und festgemacht. CR-Werkzeuge stellen hierfür Befehle zur Verfügung, die in den Skripten eingearbeitet werden. Für die grafischen Oberflächenelemente werden außerdem einheitliche Identifikatoren vergeben, dies können werkzeug-interne Namen oder System-IDs sein. Hierdurch können alle Elemente, auch nach Änderungen von Form und Lage, eindeutig und zu jederzeit reproduzierbar wiedererkannt werden. Die hierbei generierten Skripte sind in ihrer Wartbarkeit und Erweiterbarkeit viel gefälliger zu bearbeiten als die analogen Pendants.

Testwerkzeuge

Neben **qftestJUI**, das hier verwendet wird, möchte ich noch folgende Testwerkzeuge nennen:

- *Marathon 0.84* [Mar] erfasst semantische Aktionen, die auf Komponenten ausgeführt werden, anstatt sich auf Tastatur- und Mausevents zu konzentrieren. Marathon besteht aus einem Recorder, einer Ablauflogik und einem Editor. Die Testskripts sind in Python aufgebaut.
- *Pounder 0.95* [Pou] ermöglicht dynamisches Laden von GUIs, die Aufzeichnung von Skripten und deren Wiederverwendung in Testframeworks wie JUnit. Pounder unterliegt der GNU Library General Public License und ist somit frei verfügbar.
- *jfcUnit* [JFC] ist eine Erweiterung des JUnit Frameworks, die es ermöglicht Unit-Tests mit Swing-Komponenten zu verbinden. jfcUnit verwendet XML zur Aufzeichnung seiner Tests.

Testszenario

Zur Demonstration automatisierter GUI-Tests dient folgendes Szenario. Für die initiale Abfragelogik, einer Applikation für ein Reisebüro, soll der Mitarbeiter in einer Maske Ab- und Rückflugdatum sowie Abflug- und Ankunftsziel definieren können. Des Weiteren kann die gewünschte Personenzahl einer Reise angegeben werden. In dieser Maske wird vorausgesetzt, dass der Kunde ausschließlich mit dem Flugzeug reisen will. Die hierfür verwendete Eingabemaske finden Sie in Abbildung 2 (siehe nächste Seite).

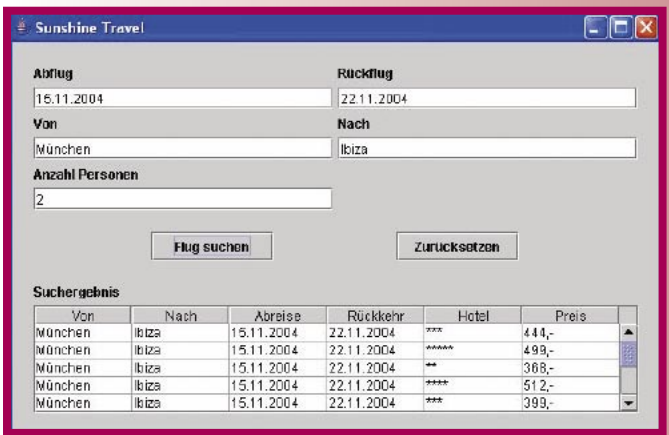

Abb. 2: Die zu testende GUI

Abb. 3: Die einzelnen Nutzereingaben im Projektbaum

Abb. 4: Eigenschaftsdialog der einzelnen Nutzereingaben

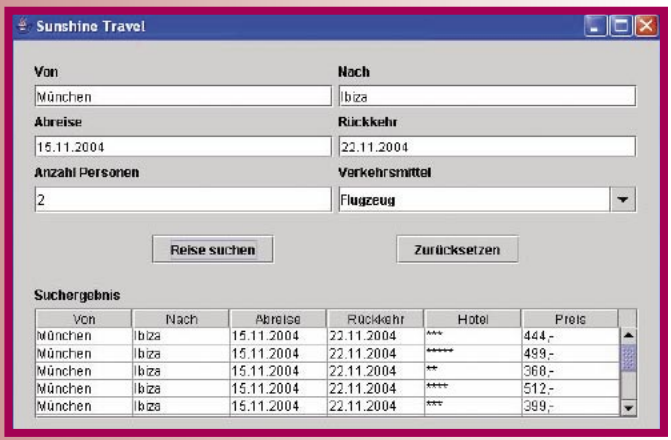

Abb. 5: Die GUI nach der Ergänzung

Als Testwerkzeug wurde **qftestJUI** aus dem Haus Quality First Software [QFS] verwendet. **qftestJUI** registriert sämtliche Reaktionen der Anwendung auf die simulierten Aktionen. Dieses CR-Werkzeug kann Werte wie z. B. den Inhalt einer Tabelle, welche die Anwendung an ihrer Oberfläche anzeigt, auslesen und mit Vorgaben vergleichen. Dabei passt es sich den Änderungen der Anwendung im Entwicklungszyklus an, etwa einer veränderten Position eines GUI-Elements oder einer geänderten Tabellenstruktur.

Nach dem Starten der GUI wurden verschiedene Nutzereingaben simuliert und aufgezeichnet. Die einzelnen Schritte werden in **qftestJUI** in einer Baumstruktur visualisiert (s. Abb. 3). Dieser Baum spiegelt die hierarchische Struktur des GUI der Anwendung wieder und ist das zentrale Element von **qftestJUI**s grafischer Oberfläche. Er ermöglicht den schnellen Zugriff auf jedes Detail der aufgezeichneten Information und gibt eine gute Übersicht über die Daten und deren Zusammenhänge. Die grundlegenden Funktionen zum schnellen Erstellen von einfachen Tests sind direkt über **qftestJUI**s grafische Oberfläche verfügbar. Mit einem Mausklick lässt sich die Aufnahme eines manuellen Tests starten. Ein einmal aufgezeichnetes Testskript kann nun beliebig oft wiederholt und auf geänderten GUIs angewendet werden.

Jede einzelne Sequenz kann hierbei in einem Eigenschaftsfeld modifiziert werden. Testsuites und Protokolle werden als XML-Dateien gespeichert, d. h. als normaler Text in wohldefinierter Syntax. Zum Suchen und Ersetzen und bei der Überprüfung von Daten, die an der Oberfläche angezeigt werden, können reguläre Ausdrücke verwendet werden. Ein Eigenschaftsdialog in **qftestJUI** ist in Abbildung 4 dargestellt.

Anstelle der fest verdrahteten Texteingaben können auch variable Werte verwendet und die aufgenommene Sequenz in eine Prozedur umgewandelt werden. Auf diesem Weg lassen sich schnell datengetriebene Tests erstellen, die immer wieder dieselbe Sequenz durchlaufen, aber jedes mal mit anderen Daten. Diese können z. B. aus einer Datei oder einer Datenbank gelesen werden.

Nach der Aufzeichnung des Testszenarios, im kontextsensitiven Modus, wird der Anforderung Rechnung getragen, dass in der Maske nicht nur Flugreisen, sondern auch Bus und Bahn berücksichtigt werden sollen. Hierfür war es in der GUI nötig, Labels umzubenennen, Elemente zu verschieben und ein neues Pull Down-Menü hinzuzufügen. Das zuvor aufgezeichnete Testskript soll auf den durchgeführten Änderungen Anwendung finden. Die Änderungen an der GUI sind in Abbildung 5 dargestellt.

Aktionen, die der Anwender ausführt, werden von der Java VM in Events umgewandelt. Jeder Event hat eine Zielkomponente. Für einen Mausklick ist das die Komponente unter dem Mauszeiger, für einen Tastendruck die Komponente, die den Tastaturfokus (*keyboard focus*) besitzt. Wenn **qftestJUI** einen Event aufzeichnet, nimmt es zusätzlich Informationen über die Zielkomponente mit auf, sodass die Komponente später beim Abspielen des Events wieder lokalisiert werden kann.

Das mag trivial und offensichtlich klingen. Tatsächlich ist die Wiedererkennung der Komponenten aber einer der komplexesten Teile von **qftestJUI**. Der Grund dafür liegt in der Notwendigkeit, mit einem gewissen Grad an Veränderungen zurechtzukommen. **qftestJUI** ist als Werkzeug für die Durchführung von Regressionstests ausgelegt. Wird eine neue Version des *System Under Test (SUT)* getestet, sollten bestehende Tests idealerweise unverändert durchlaufen. Folglich muss sich **qftestJUI** an ein möglicherweise geändertes GUI anpassen können.

Hierfür wird in **qftestJUI** die neue Oberfläche gestartet und das zuvor generierte Testskript dazu aufgerufen. Da **qftestJUI** jedes einzelne GUI-Element zuvor anhand seines Elementnamens im Quellcode erkannt und diesen als ID im Skript übernommen hat, kann dasselbe Skript nun zur Anwendung kommen, da sich die Namensgebung der einzelnen Elemente durch die Repositionierung nicht geändert hat. In diesem Fall wird es nach dem Neuabspielen des Testskripts zu keiner Fehlermeldung kommen, da in der Logik zum vorhergehenden GUI nur Ergänzungen gemacht wurden, d. h. alle zuvor angesprochenen GUI-Elemente sind noch ansprechbar.

In typischen Java GUIs gibt es Komponenten wie Bäume und Tabellen, die ziemlich komplex sind und eine beliebige Zahl von Unterelementen wie Baumknoten oder Tabellenfelder enthalten können. Diese Unterelemente sind selbst keine echten GUI-Komponenten, sondern nur eine grafische Darstellung der zugrunde liegenden Daten. Diese Unterscheidung ist rein technischer Natur und da es aus Sicht eines Testers durchaus Sinn macht, die Elemente als eigenständig und als mögliche Ziele für Events anzusehen, bietet **qftestJUI** dafür spezielle Möglichkeiten, indem es ein Unterelement durch einen Element-Knoten repräsentiert.

Auch wenn es ganz unterhaltsam sein kann, Sequenzen aufzuzeichnen und dem Spiel der Fenster des SUT beim Wiederabspielen zuzusehen, geht es doch eigentlich darum, herauszufinden, ob sich das SUT dabei auch korrekt verhält. Diese Aufgabe übernehmen in **qftestJUI** so genannte Checks. Der Check-Text-Knoten liest den Text aus einer Komponente, oder einem Unterelement, wie einem Baumknoten oder Tabellenfeld, aus und vergleicht ihn mit einem vorgegebenen Wert. Stimmen diese nicht überein, wird ein Fehler signalisiert.

Auswertung der Testergebnisse

Nachdem ein Testlauf beendet ist, erscheint in der Statuszeile des Hauptfensters von **qftestJUI** eine Meldung mit dem Ergebnis. Im Idealfall lautet diese „Keine Fehler". Sind Probleme aufgetreten, wird die Zahl der Warnungen, Fehler und Exceptions angezeigt und gegebenenfalls zusätzlich ein Dialogfenster geöffnet.

Beim Abspielen eines Tests notiert **qftestJUI** jede einzelne Aktion in einem Protokoll. Die Struktur dieses Protokolls ist der einer Testsuite sehr ähnlich, mit einem Unterschied: Knoten werden bei ihrer Ausführung in das Protokoll aufgenommen.

Das Protokoll ist das entscheidende Hilfsmittel, wenn es herauszufinden gilt, was bei einem Testlauf fehlgeschlagen ist, wo es passiert ist und im besten Fall auch warum es passiert ist. Daher liegt das Gewicht bei einem Protokoll bei der Vollständigkeit der Information.

Neben den Knoten, die aus der Testsuite übernommen wurden, enthält ein Protokoll Zeitstempel, optionale Anmerkungen, Informationen über Variablenexpansion, verschiedene Arten von Meldungen und insbesondere Fehlerinformationen.

Da die gesammelten Informationen über einen längeren Testlauf gewaltige Mengen an Arbeitsspeicher verbrauchen, erstellt **qftestJUI** normalerweise kompakte Protokolle, bei denen nur die für den Report oder die Fehlerdiagnose relevanten Informationen aufgehoben werden.

Wie bei jeder komplexen Entwicklung wird es ab einem gewissen Punkt nötig sein, Probleme in einer Testsuite zu debuggen, die nicht mehr einfach durch Analysieren der Elemente und der Struktur einer Testsuite zu lösen sind. Zu diesem Zweck verfügt **qftestJUI** für den Applikationscode über einen intuitiven Debugger.

Fazit

Der Artikel konzentriert sich ausschließlich auf das Testen von Swing GUIs, doch gelten viele der Testanforderungen auch für Thin Clients (JSF, JSP, HTML). Für das Testen in komplexen Applikationsentwicklungen ist es unabdingbar, die grafischen Nutzerschnittstellen zu erproben. Dies sollte sowohl in einem Akzeptanztest der zukünftigen Anwender als auch in einem funktionalen Test innerhalb der Systemlogik geschehen. Für diesen Zweck werden diverse CR-Werkzeuge zur Verfügung gestellt, und im Hinblick auf Kostenreduktion durch Testautomatisierung sind diese Toolinvestitionen durchaus gerechtfertigt.

Literatur und Links

[JFC] jfcUnit, **http://jfcunit.sourceforge.net/**
[Mar] Marathon 0.84, **http://marathonman.sourceforge.net/**
[Pou] Pounder 0.95, **http://pounder.sourceforge.net**
[QFS] Testwerkzeug **qftestJUI** von Quality First Software, **www.qfs.de**
[Tha02] G. E. Thaller, Interface Design,
Software und Support Verlag, 2002

Christian Glatschke ist seit Jahren als Berater und Trainer im Bereich Softwareentwicklung tätig. Er ist dabei u.a. im Bereich Qualitätssicherung in Softwareprojekten engagiert und setzt sich hier mit unterschiedlichen Anforderungen an QA auseinander. Außerdem betreut er die Kolumne zum JCP. E-Mail: c.glatschke@simuvid-software.de.