The Black Art of GUI Testing[1]

Christoph Thelen, Technische Hochschule Mittelhessen, christoph.thelen@mni.thm.de

Viel zu oft zeichnen sich Oberflächentests durch die folgenden drei Dinge aus: lange Laufzeiten, Fragilität und einen hohen Aufwand in Konzeption und Wartung. Im Kontext von Web-Apps soll dieser Vortrag zeigen, wie die drei Probleme mit unterschiedlichen Mitteln bekämpft werden können. Den Zuhörerinnen und Zuhörern sollen neben dem großen Ganzen zahlreiche praktische Tipps vermittelt werden, die sie noch am selben Tag in ihren Projekten umsetzen können.

Die Probleme im Kontext von GUI-Testing treten nicht über Nacht auf; vielmehr ist es ein schleichender Prozess, der durch eine Vielzahl kleinerer Probleme entsteht. Um sie zu vermeiden, sind drei Dinge hilfreich:

1. die Grundstruktur einer idealen Testhierarchie zu kennen,
2. die einzelnen Probleme bereits im Ansatz zu identifizieren und
3. Strategien parat zu haben, wie die Probleme im Großen und im Kleinen gelöst werden können.

Die Test-Pyramide

In einer idealen Welt wird bei einem Softwareprojekt viel Wert auf das separate Testen von drei Schichten gelegt: der GUI, den Services und den einzelnen Units. Veranschaulicht als Pyramide[2] (Abb. 1) bilden Unit-Tests das Fundament, während GUI-Tests die Spitze symbolisieren. Die Größe der Abschnitte gibt die relative Anzahl der jeweiligen Tests am Gesamtprojekt an. Überhalb der Pyramidenspitze wird oftmals noch eine Wolke platziert, welche die manuellen Tests symbolisiert.

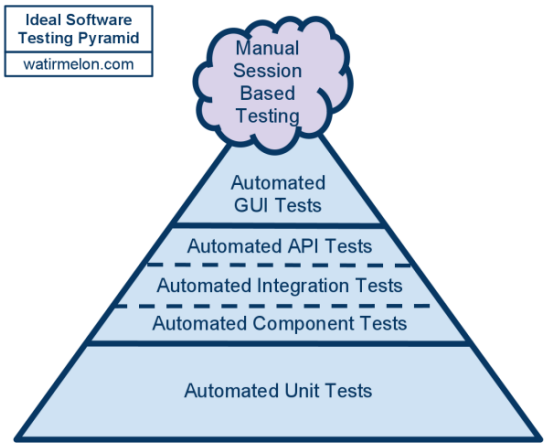

Abbildung 1: Die Teststrategie folgt idealerweise einer Pyramide, mit möglichst wenig GUI-Tests, aber zahlreichen Unit-Tests.

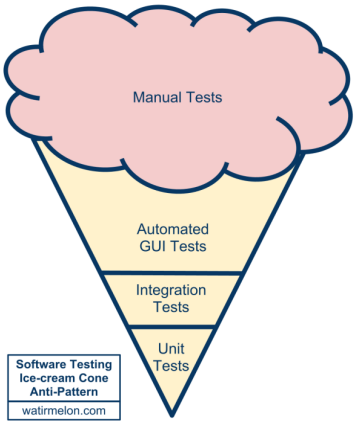

Abbildung 2: Typischerweise wird jedoch die genau gegensätzliche Struktur zu Abb. 1 umgesetzt -- wenige Unit-Tests und umso mehr GUI-Tests.

Bilder: Alister Scott, Lizenz: CC BY 2.5 AU

[1] Nach dem gleichnamigen Artikel von Laurence R. Kepple in Dr. Dobb's Journal. Veröffentlicht im Februar 1994 besitzt er überraschenderweise auch nach 20 Jahren noch Relevanz.

[2] Die Grundidee der Test-Pyramide stammt von Mike Cohn.

In der Realität herrscht dagegen oftmals “verkehrte Welt”: Die Pyramide ist auf den Kopf gestellt, die Wolke ist aufgebläht. Es entsteht eine Eistüte (Abb. 2), eine umgedrehte Testpyramide: Die Wolke macht nun den größten Teil aus; es folgen die GUI-Tests und in immer kleinerer Anzahl die Service-Tests sowie schließlich in geringster Anzahl die Unit-Tests.

Die Abkehr von der idealen Vorstellung geschieht über einen längeren Zeitraum in vielen kleinen Schritten. Wenn diese Schritte identifiziert werden, kann rechtzeitig aktiv gegengesteuert werden. Dazu ist es wichtig zu wissen, wie die Probleme überhaupt entstehen.

Wie die Probleme entstehen

Blinder Einsatz von Capture-Replay-Tools

Die Tendenz hin zu einer Vielzahl an GUI-Tests liegt zum Beispiel darin begründet, dass mit relativ wenig Aufwand ein (vermeintlich) großer Teil der Anwendung getestet werden kann. Wurden in dem Projekt bislang keine Tests geschrieben, ist das Einführen von Unit-Tests umso schwieriger. Sie bleiben deswegen in der Minderheit. Neue GUI-Tests lassen sich dagegen zügig erstellen, etwa mit Capture-Replay-Tools wie Selenium[3].

So praktisch diese Werkzeuge für das Erstellen von Tests auch sein mögen, die Aufzeichnungen sind in der Regel nicht ohne weitere Anpassungen einsetzbar. Bei der Aufzeichnung eines Login-Vorgangs müssen etwa Benutzername und Passwort auf allen Testsystemen funktionieren. Des Weiteren sind oftmals die CSS-Selektoren so spezifisch, dass die angesprochenen Elemente sehr schnell nicht mehr auffindbar sind. Ein weiteres Problem ist das richtige Timing etwa von Ajax-Events. All diese Dinge führen zu fragilen Tests.

Schlechtes Timing

Gerade bei Apps mit viel Asynchronität ist das richtige Timing wichtig. Stimmt dieses nicht, existieren zum Beispiel wichtige Elemente noch nicht auf der Webseite. Die Tests schlagen dadurch fehl, obwohl kein echter Fehler vorhanden ist -- ein weiterer Punkt, der zu fragilen Tests führt.

Wird versucht, das Problem durch längere Wartezeiten zu lösen, erhöht sich automatisch die Laufzeit der Tests. Schwierig wird es außerdem, wenn die Testsysteme zeitweise unter hoher Last stehen. Dann können selbst längere Wartezeiten zu sporadischen Ausfällen führen. Insgesamt führt das zu gleichzeitig fragilen wie langsamen Tests.

Abhängigkeiten zwischen Tests

Bei langsamen Tests kann Parallelisierung Abhilfe schaffen. Dies erfordert jedoch, dass es keine Abhängigkeiten zwischen den Tests gibt. Diese ergeben sich jedoch sehr schnell: Der erste Test legt etwa einen Blogpost an, der in einem zweiten Test editiert und in einem dritten schließlich gelöscht wird. Zuvor wurden diese Tests der Reihe nach ausgeführt; durch die Parallelisierung geht diese Ordnung aber verloren. Die Tests sind plötzlich fehlerhaft. Existieren viele chronologische Abhängigkeiten, ist eine großflächige Parallelisierung ohne Änderungen der Tests nicht möglich.

Undurchdachte Tests

Je mehr Tests entstehen, desto höher fällt auch die Gesamtlaufzeit aus. Die parallele Ausführung schafft Abhilfe, doch das Hauptproblem besteht oftmals in der Struktur. Für viele Tests muss ein Kontext aufgebaut werden, ein Login zum Beispiel, der den eigentlichen Test in die Länge zieht. Je länger der Aufbau dauert,

[3] S. http://docs.seleniumhq.org/

desto länger dauert folglich der gesamte Testlauf. Fehler beim Kontextaufbau können eine Vielzahl an Tests scheitern lassen, was die Fehlersuche zusätzlich erschwert.

Stetig sinkendes Vertrauen

Mit der Anzahl an Tests steigt auch der Wartungsaufwand. Je fragiler die Tests dabei sind, desto größer ist letztendlich auch der Arbeitsaufwand. Wenn immer wieder Tests eigentlich grundlos fehlschlagen, geht irgendwann das Vertrauen in sie verloren, sodass sie bald keine echte Stütze mehr sind. Fehler werden nicht mehr ernst genommen, durch die langen Laufzeiten werden sie eventuell sogar übersprungen und nur noch sporadisch ausgeführt -- um deswegen noch aufwändiger in der Wartung zu werden, weil sich die Fehler anhäufen.

Ohne Vertrauen werden die Tests zum Ballast. Jedoch gibt es Möglichkeiten, um die angesprochenen Probleme zu lösen oder gar nicht erst aufkommen zu lassen.

Lösungen

Die ersten drei Lösungen verstehen sich als High-Level-Ansätze für die grundsätzliche Gestaltung von GUI-Tests und der dafür benötigten Infrastruktur. Die weiteren drei Lösungen sind praktische Tipps, wie auch bereits bestehende Tests robuster und schneller gemacht werden können.

High-Level

1. Test-Strategie entwickeln

Zunächst stellt sich die Frage, was überhaupt getestet werden soll. Besonders wenn ein Projekt nicht im Test-First-Ansatz entwickelt wurde, muss an irgendeiner Stelle begonnen werden. Eine gute Möglichkeit ist es, die eingangs erwähnte Test-Pyramide zunächst abwärts zu bearbeiten. Zuerst wird mit einem GUI-Test begonnen, da dieser initial am aufwändigsten in der Umsetzung ist. Dabei sollte ein Kernablauf der App getestet werden -- typischerweise der Ablauf, der vorher am häufigsten manuell getestet wurde. Die Wahrscheinlichkeit ist hoch, dass es sich dabei um das wichtigste Feature handelt. Der erste Test sorgt dafür, dass die Infrastruktur für weitere Tests in ihren Grundzügen bereits vorhanden ist. Erste Erfolgserlebnisse sind da, ab hier wird es nun einfacher!

Von weiteren Tests sollte aber zunächst abgesehen werden. Zwar können sie jetzt leicht ergänzt werden, jedoch werden oftmals zu schnell eher kleinere Aspekte getestet, etwa ob Fehlermeldungen erscheinen. Vielmehr sollte die Pyramide eine Stufe herab gestiegen und die API getestet werden. Web-Apps bringen diese API meistens automatisch mit; es ist typischerweise die HTTP-Schnittstelle des Servers. Und schließlich können im letzten Schritt Unit-Tests entwickelt werden.

Mit diesem Test-Rahmenwerk kann nun über eine Automatisierung mittels Continuous Integration nachgedacht werden. Als nächstes stellt sich die Frage, welche Dinge in welcher Stufe der Pyramide überprüft werden sollen.

2. Geschäftslogik nicht mittels GUI testen

Die goldene Regel lautet, dass die Geschäftslogik nicht durch die GUI getestet werden sollte. Als einfaches Beispiel soll eine Taschenrechner-App dienen: Es wäre sehr einfach, über einen GUI-Test sämtliche Funktionen des Taschenrechners zu überprüfen -- zwei Zahlen und eine Operation werden über die Buttons eingegeben, dabei erscheinen jeweils die Zahlen und am Ende das Ergebnis im “Display”. Die Geschäftslogik sei hier also die Operation des Taschenrechners.

Es stellt sich die Frage, ob es nicht ausreicht zu testen, wie die GUI auf die verschiedenen Ereignisse reagiert. Zum Beispiel, dass im Display Zahlen angezeigt werden -- oder eigentlich überhaupt "etwas". Ist es für die GUI von Belang, ob im Display das korrekte Ergebnis erscheint? Solche, von der Geschäftslogik losgelösten GUI-Tests werden automatisch robuster, da sie keine konkreten Kenntnisse mehr über die tatsächliche Implementierung haben müssen.

Dennoch sollten in geeigneter Zahl auch so genannte End-to-End-Tests vorliegen, die also die Anwendung von vorne bis hinten und zurück überprüfen. Sie sollen die wichtigsten Use-Cases absichern. Es ist wichtig, hierbei die Balance zu finden zwischen ausreichender Absicherung und einer ebenso akzeptablen Laufzeit und Stabilität.

3. Testbus-Ansatz

Alle grundlegenden Teilbereiche der Software durch eine Schnittstelle vom Rest zu trennen -- das ist die Kernidee des Testbusses. Der Ansatz wurde von Robert C. Martin publiziert [1] und soll vor allem dazu dienen, für jeden Teilbereich eine eigene Testschnittstelle zur Verfügung zu haben. Dadurch kann die Implementierung der für die Tests irrelevanten Komponenten ausgetauscht werden.

Konkret kann die GUI mit einem eigenen Test-Backend betrieben werden, um die Tests zu vereinfachen. Genauso kann die Datenbank durch eine Testdatenbank ersetzt werden, ohne dass jeweils der zu testende Systemteil davon etwas mitbekommt. Eine solche Trennung erlaubt es, die Geschäftslogik direkt über die API zu testen, ohne den Umweg über die GUI gehen zu müssen.[4]

Low-Level

1. Selektoren vereinfachen und Testdaten verwenden

Der schnellste Weg zu mehr Stabilität liegt darin, die Tests möglichst einfach zu halten. Dies fängt bei der Auswahl der Selektoren an: Wie für die Interaktion benötigte Elemente angesprochen werden, kann einen großen Einfluss auf die Robustheit haben. Werden sie über CSS-Selektoren oder XPath-Ausdrücke ausgewählt, sollten diese von der Umgebung so wenig wie möglich berücksichtigen. Es ist offensichtlich, dass ein XPath-Ausdruck, der die gesamte HTML-Struktur berücksichtigt, bei der nächsten Änderung der Struktur keine Ergebnisse mehr liefern wird.

Des Weiteren sollten immer Testdaten verwendet werden, die für jeden Test aufgebaut und danach wieder abgebaut werden. Damit entstehen erst gar nicht etwaige Abhängigkeiten zwischen Tests, die einen Blogpost erstellen, editieren und löschen. In den Tests kann sich außerdem gefahrlos auf die Testdaten bezogen werden. Sind die Daten aussagekräftig, helfen sie, die Lesbarkeit zu erhöhen.

Die nötigen Daten können beispielsweise direkt in die Datenbank geschrieben werden. Allerdings müssen diese Daten bei Schema-Änderungen ebenfalls angepasst werden. Die Daten können auch direkt über die reguläre API erstellt werden. Dies würde die Tests aber wiederum anfällig für Fehler innerhalb der API machen. Die möglichen Nachteile gilt es abzuwägen.

2. View-Navigation vereinfachen

Bei Oberflächentests kann viel Zeit dafür aufgewendet werden, dass überhaupt erst zum "Ort des Geschehens" navigiert werden muss. Diese Zeit kann eingespart werden, wenn die entsprechenden Views direkt erreichbar sind. Am einfachsten geht das -- sofern möglich -- über das Öffnen der konkreten URLs. Wenn ein Login notwendig ist, kann dies aber noch nicht ausreichend sein. Eine weitere Möglichkeit ist das

[4] Martin Fowler nennt dies übersetzt "unter der Haut gelegene Tests" (s. http://martinfowler.com/bliki/SubcutaneousTest.html, Abruf: 06.01.2014)

direkte Absenden eines Login-Requests im Hintergrund per HTTP, um das Ausfüllen eines Webformulars abzukürzen.

Ein anderes Beispiel sind Abläufe, die über mehrere Ansichten und Schritte reichen. Wenn in der fünften Ansicht etwas überprüft werden soll, können die notwendigen Eingaben der vorangegangen Ansichten womöglich durch Setzen der richtigen Einträge etwa im LocalStorage bereits “erledigt” worden sein, sodass automatisch die richtige Ansicht erscheint.

Bereits kleine Zeitersparnisse können sich bei einer Vielzahl von Tests merklich auswirken. Je weniger Arbeit ein Test hat, desto weniger anfällig ist er auch für Fehler. Ein Fehler im Login-Formular kann zum Beispiel 50% aller Tests fehlschlagen lassen, selbst wenn diese gar nichts mit dem Login an sich zu tun haben.

3. “Wackelige” Tests mehrfach wiederholen

Bei der Implementierung von GUI-Tests werden in aller Regel Frameworks und Bibliotheken eingesetzt, die das Testen erleichtern. Es lohnt sich, einen genaueren Blick hinter die Kulissen zu werfen. Mit Kenntnis über die inneren Abläufe können die Frameworks etwa um eigene Konstrukte erweitert werden. Warum nicht zum Beispiel die Ausführung von Tests erst dann mit einem Fehler abbrechen lassen, wenn sich der Fehler in mehreren Versuchen bestätigt hat? Dies hilft, sporadisch ausfallende Tests zu beheben. Erst wenn ein Problem dauerhaft besteht, handelt es sich wahrscheinlich um einen tatsächlichen Fehler.

Natürlich kann es passieren, dass dadurch vielleicht doch ein echter Fehler übersehen wird, der eben wirklich nur sporadisch auftritt. Es ist daher immer eine gute Idee, die Fehlerfälle -- auch wenn sie sich nicht mehrfach reproduzieren lassen -- zumindest in die Logs aufzunehmen und mit Screenshots und sonstigen Informationen zu versehen.

Zusammenfassung

Der Vortrag zeigt die typischen Probleme bei der Umsetzung von GUI-Tests im Kontext von Web-Apps und bietet den Zuhörerinnen und Zuhörern mögliche Lösungsansätze,

- wie GUI-Tests robust gestaltet werden können,
- wie eine Architektur konstruiert werden kann, die GUI-Tests erleichtert und
- wie lange Laufzeiten von GUI-Tests verringert werden können.

Die beschriebenen Lösungen werden mit Beispielen aus der Open-Source-Welt veranschaulicht. Den Zuhörerinnen und Zuhörern werden somit neben der reinen Wissenvermittlung konkrete Implementierungen an die Hand gegeben, die sie für ihre eigenen Projekte adaptieren können.

Literatur

[1] Martin, R.C., "The test bus imperative: architectures that support automated acceptance testing," *Software, IEEE* , vol.22, no.4, pp.65,67, July-Aug. 2005