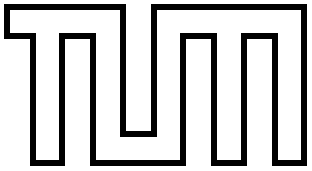


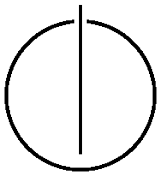
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Seminararbeit in Informatik

Warum zerbrechen meine automatisierten GUI-Tests?

Leonard Husmann



Ich versichere, dass ich diese Seminararbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den July 28, 2020 Leonard Husmann

Abstract

In heutigen Softwareprojekten sind GUI-Tests häufig vertreten. Diese dienen, wie alle Tests, dazu die Softwarequaliät aufrecht zu erhalten und somit die Geschwindigkeit in der Entwicklung möglichst linear zu halten. Doch diese Software befindet sich in der Entwicklung und ist somit Gegenstand konstanter Veränderung. Insbesondere verändert sich dabei auch die GUI mit ihren Komponenten. Mit diesen Änderungen an den Komponenten müssen die GUI-Tests umgehen können. Andernfalls schlagen sie fehl, sie zerbrechen. Ein häufiger Grund dafür ist, dass geänderte Komponenten von dem eingesetzten Framework für GUI-Tests nicht mehr gefunden und wiedererkannt werden. In dieser Seminararbeit sollen anhand einer Fallstudie einzelne Klassen an Fehlern in GUI-Tests gefunden werden. Ziel ist es diese zu vermeiden und somit den Wartungsaufwand für die GUI-Tests möglichst gering zu halten. Dabei werden eine Menge an Testfällen für die Version v2.00 von KeePass erstellt. Die nachfolgende Entwicklung wird simuliert, indem die Tests auf die nächste Version, also v2.01, angewendet werden. So lässt sich feststellen, ab wann die Tests fehlschlagen. Diese werden dann in Klassen gruppiert. Zusammengefasst lässt sich sagen, dass sich durch die Berücksichtigung von Persistenz in der Testumgebung, Trivialen Umbenennungen und Eigenschaften des GUI-Test Frameworks die überwiegende Mehrheit der Fehler vermeiden lässt.

Contents

List of Figures

List of Tables

1 Theorie

Im folgenden sollen wenige theoretische Grundlagen für die anschließende praktische Arbeit gelegt werden.

1.1 Gründe für GUI-Tests

Zu Beginn soll dabei die Relevanz von GUI-Tests in Softwareprojekten hervorgehoben werden. Neben den allgemeinen Gründen für Tests sind hier spezifische Gründe für GUI-Tests erläutert.

1.1.1 End-to-End Tests

Ein Grund für GUI-Tests ist die Möglichkeit sogenannte End-to-End Tests zu erstellen. Dies bedeutet, dass die Tests aus der Sicht eines Nutzers durchgeführt werden. Es werden also Eingaben in der GUI getätigt und deren Reaktion überprüft. Die Tests setzen also am "Ende" des Systems, der GUI, an und testen den Kontrollfluss innerhalb des Systems über verschiedene Komponenten oder Subsysteme hinweg. Die Checks der Tests werden ebenfalls auf Ebene der GUI, also dem "Ende", durchgeführt. Somit laufen die Tests von Ende zu Ende. Es wird also das System als Ganzes getestet und somit werden mögliche Fehler und Bugs durch die Integration (Schnittstellen von Komponenten etc.) gefunden.

1.1.2 Akzeptanztests während der Entwicklung

Ein weiterer Grund für GUI-Tests sind die damit möglichen Akzeptanztests. Die Besonderheit liegt darin, dass mit GUI-Tests diese schon während der Entwicklung zur Verfügung stehen können. Es können also Akzeptanztests anhand der Requirements definiert werden und diese dann mit Hilfe von automatisierten GUI-Tests abgebildet werden. Dem Entwicklerteam stehen somit während der Entwicklung diese Akzeptanztests schon zur Verfügung und werden laufend überprüft. Somit kann vermieden werden, dass erst bei Durchführung der Aktzeptanztests durch den Kunden oder Nutzer am Ende des Projekts Mängel aufgedeckt werden. [MS04]

1.2 Unterscheidung von GUI-Tests

Möchte man die Gesamtheit an GUI-Tests in einzelne Klassen oder Gruppen unterteilen, so kann man dies anhand verschiedener Dimensionen machen. Anhand dieser Dimensionen als abstrakte Kriterien lassen sich konkrete Klassen als Typen von GUI-Tests definieren [ISO13]. Im Folgenden sind einige Beispiele für Dimensionen aufgelistet, die jedoch keinen Anspruch auf Vollständigkeit erhebt.

1.2.1 Automtisierte vs. Manuelle Tests

Vielleicht die offensichtlichste Dimension zur Unterscheidung ist, ob ein GUI-Test automatisiert oder manuell ausgeführt wird [BH16]. Natürlich spielen in diesem Zusammenhang die automatisierten Tests die zentrale Rolle. Jedoch sollten auch die manuellen Tests, allein aus Gründen der Vollständigkeit, genannt werden.

1.2.2 Funktionale vs. Nicht-Funktionale Tests

Hierbei ist die Unterscheidung ähnlich zu der bei Requirements. So sollten z.B. auch die Requirements bei der Definition der GUI-Tests oder Tests im Allgemeinen dienen. Zwei konkrete Beispiele für je eine Testklasse sind im Folgenden näher erläutert.

Lasttests Bei Lasttests, als Beispiel für Nicht-Funktionale Tests, wird das System unter Last oder auch Stress gesetzt und getestet. Erreicht wird dies beispielsweise, dadurch dass die Zeitabstände zwischen simulierten Nutzereingaben drastisch verkürzt werden. Somit ist die Menge an Eingaben pro Zeit und damit auch die zu verarbeitenden Ereignisse für das System größer. Eine weitere Möglichkeit Stress für ein System zu simulieren sind mehrere parallele Nutzer. Dies ist allerdings davon abhängig, ob das getestete System überhaupt mehrere Benutzer zulässt. Dadurch lassen sich Aussagen darüber treffen, wie das System unter diesen Bedingungen weiterhin funktioniert oder nicht. Wichtig zu erwähnen ist, dass es sich bei Lasttests meist um modifizierte funktionale Tests handelt. Die Modifikation liegt wie bereits erwähnt an den Zeitabständen zwischen Nutzereingaben.

Regressiontest Bei Regressionstest handelt es sich um eine Untergruppe der Funktionalen Tests. Diese überprüfen nicht neue Funktionen auf Korrektheit, sondern überprüfen, ob nach einer Änderung eine Regression vorliegt [MS03]. Eine Regression liegt vor, wenn durch eine Änderung eine korrekte Funktionalität nun nicht mehr funktioniert ist. Also wenn durch eine Änderung an der Software ein bereits bestehendes Feature kaputt geht.

2 Studie

Übersicht

2.1 Zielsetzung

Ähnlich zu Dimensionen an denen sich verschiedene Testarten ableiten lassen (vgl. Kapitel 1.2) lassen sich auch verschiedene Dimensionen definieren durch die sich das SUT verändert. Beispiele dafür sind die Version, die Sprache oder die Platform. In dieser Seminararbeit sollen nur entstandene Änderungen durch die Entwicklung, also Version, betrachtet werden.

Ziel dieser Studie ist es Fehler in GUI-Tests und Möglichkeiten diese zu vermeiden zu finden. Somit steigt die Qualität der GUI-Tests bei der initialen Erstellung und diese zerbrechen später weniger häufig. Es soll also mit einem zusätzlichen Zeiteinsatz bei der Erstellung die Qualität gesteigert werden, um im späteren Verlauf häufiges Anpassen der GUI-Tests zu vermeiden. Ist dabei die zusätzlich eingesetzte Zeit zu Beginn geringer als die Summe im späteren Verlauf potentiell aufgewendeten Zeit zur Anpassung ist der Wartungsaufwand gesunken. Dies ist in allen Fällen wünschenswert.

2.2 Vorgehen und Umsetzung

Als konkrete zu testende Software, also SUT, dient KeePass für Windows in den Versionen v2.00 bis v2.18. Als Test-Framework dient QF-Test v5.0.3 mit der Test Engine für Windows Applikationen.

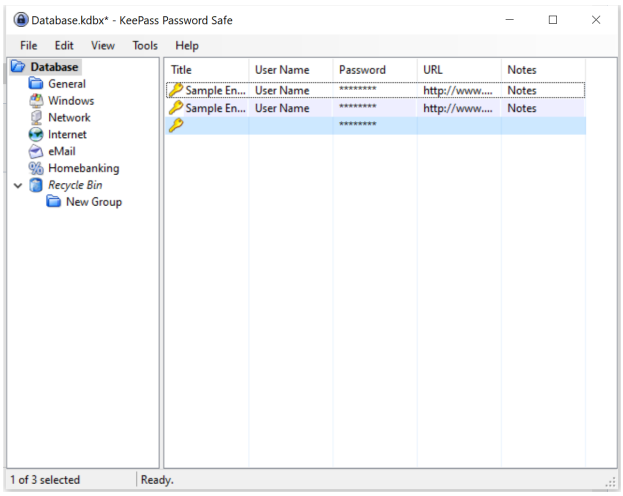


Figure 2.1: KeePass in der Version 2.00

Zu Beginn werden GUI-Tests auf v2.00 von KeePass definiert. Es geht dabei nicht darum Funktionen von KeePass auf Korrektheit zu testen, sondern darum die Komponenten zu erkennen. Bei den meisten Nutzereingaben handelt es sich also um einfache Mausklicks auf Komponenten. Erstellt werden die GUI-Tests mit Hilfe von Capture and Replay. Dabei speichert QF-Test die Attribute der Komponenten für die spätere Wiedererkennung ab. Anhand dieser versucht QF-Test die Komponenten im ausgeführten Test wieder zu erkennen. Ändern sich die Attribute zu stark, erkennt QF-Test die Komponente nicht wieder und wirft eine ComponentNotFoundException. Zu diesem Zeitpunkt ist der GUI-Test also zerbrochen.

Sind die GUI-Tests definiert besteht der nächste Schritt darin die Version von KeePass schrittweise zu erhöhen. Somit soll die Entwicklung und Änderung der Softare und insbesondere der Graphischen Benutzeroberfläche simuliert werden. Danach werden die Tests auf die späteren Versionen (v2.01, v2.02, ...) ausgeführt. Verändern sich dabei Komponenten der GUI zu stark zerbrechen die Tests erwartungsgemäß. Dann werden die Tests so angepasst, dass sie auch mit dieser Änderung nicht zerbrechen, der Test wird also gewartet, und der Vorgang wird wiederholt.

2.3 Ergebnisse

Das unmittelbaren Ergebnis der Testausführung ist in Abbildung 2.2 zu sehen. Die erfolgreichen Tests sind in grün dargestellt und die fehlgeschlagenen in roter Farbe.

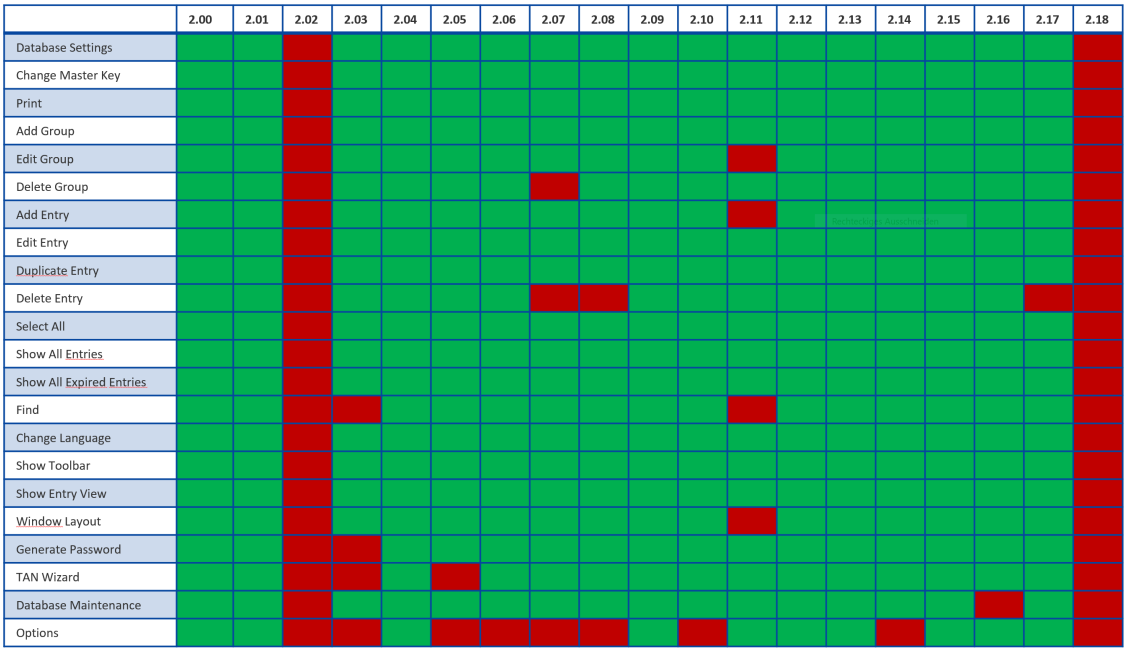


Figure 2.2: Die unmittelbaren Ergebnisse

Deutlich erkennbar stechen v2.02 und v2.18 heraus. Bei diesen beiden Versionen sind jeweils alle Tests fehlgeschlagen. Grund dafür waren Popup-Fenster beim Start (Information über Änderung des Datenbankformats von KeePass und Suche nach Updates). Diese Fehler sind in GUI-Tests kaum vermeidbar und sollen deswegen in 2.4 nicht betrachtet werden. Ebenso Fachliche Änderungen, wenn z.B. eine Komponente ersatzlos verschwindet. Die Ergebnisse ohne Fehler dieser Art, also Popup oder fachliche Änderungen sind in Abbildung 2.3 dargestellt.

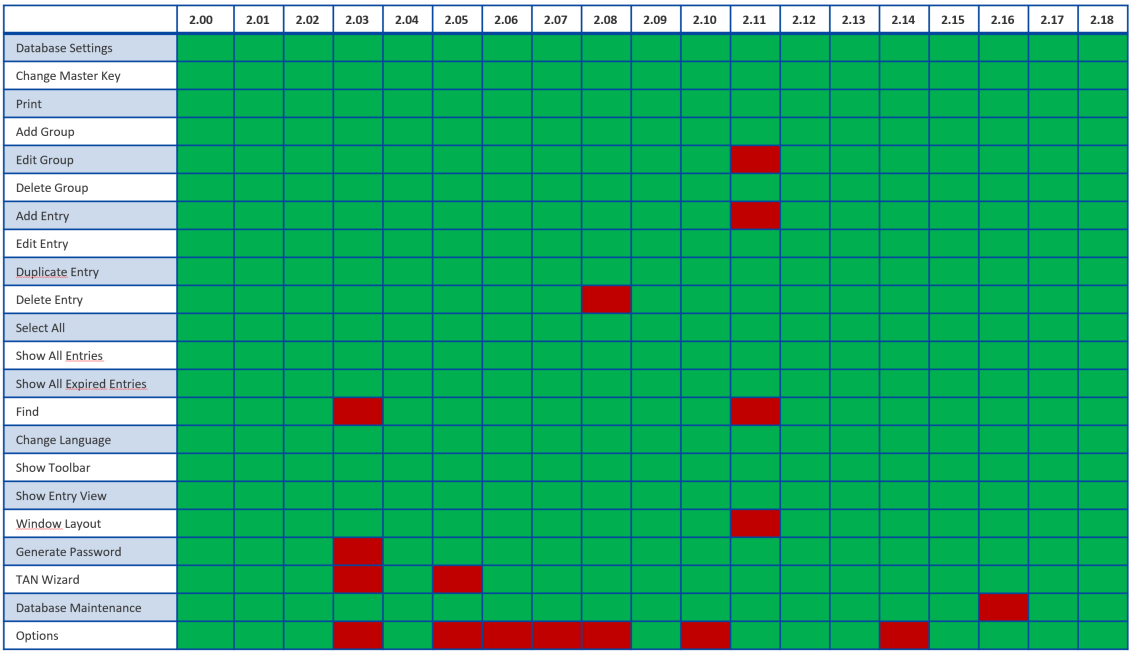


Figure 2.3: Die Ergebnisse ohne fachliche Änderungen

2.4 Interpretation

Folgend sollen die Fehler aus den vorangegangenen Ergebnissen in Abbildung 2.3 gruppiert werden. Für die jeweilge Gruppe werden dann Möglichkeiten aufgezeigt diesen Fehler zu vermeiden.

2.4.1 Persistenz in der Testumgebung

Die erste Gruppe an Fehlern wurde durch Persistenz in der Testumgebung verursacht. Dies bedeutet, dass über die Ausführung von Tests hinweg vom SUT Daten persistent geschrieben und genutzt werden.

Im Fall von KeePass war dies im beispielsweise Dialog "Einstellungen" vorhanden. Ab Version 2.07 speichert KeePass den geöffneten Tab beim Schließen des Dialoges. Beim nächsten Öffnen des Dialoges ist der gespeicherte Tab vorausgewählt und nicht der erste Tab wie in den Version 2.00 bis 2.06. Der Tests waren unter der Annahme, dass immer der erste Tab ausgewählt ist geschrieben. Sie schlugen somit ab v2.07 fehl.

Insgesamt sind zwei der 17 fehlgeschlagenen Tests durch diesen oder einen anderen Grund fehlgeschlagen. Dieser Fehler ist sehr leicht zu vermeiden, indem die persistenten Daten vor der Ausführung der Tests gelöscht werden. Bei einem selber entwickelten SUT ist auch eine Funktion denkbar bei der das SUT durch ein Argument einen sogenannten *clean start* ausführt. Also ohne Berücksichtigung der persistenten Daten.

2.4.2 Triviale Umbenennungen

Die in dieser Fallstudie ermittelte größte Gruppe an Fehler bilden triviale Umbenennungen. Dies sind Umbenennungen, die einem normalen Nutzer möglicherweise nicht auffallen jedoch bei automatisierten GUI-Tests aufgedeckt werden. Beispiele aus KeePass dafür sind Umbenenungen von *Find* zu *Find...*, von *Side-By-Side* zu *Side by Side* oder von *Lock Windows* zu *Lock Window or Logout*.

Mit zehn von 17 fehlgeschlagenen Tests aus diesem oder einem anderen Grund ist diese Gruppe die Größte der hier aufgeführten Gruppen. Es ist also von großen Interesse diese Fehler zu vermeiden.

Möglichkeiten diese Fehler zu vermeiden sind breiter gefächert als bei der ersten Gruppe von 2.4.1 Beispielsweise können durch Tests ohne *Case Sensitivity* diese Fehler vermieden werden. Falls eine Komponente bereits durch einen Substring ausreichend charakterisiert ist, kann lediglich auf diesen Substring gematcht werden. Ein anderer Fall, in dem eine gewisse Fehlertolernz zugelassen werden soll, könnten diese Toleranz durch eine Levenshtein Distanz realisiert werden. So können die Komponenten im Rahmen dieser Toleranz umbenannt werden ohne dass die GUI-Tests zerbrechen. Ermöglicht das GUI-Test Framework den Einsatz von Regulären Ausdrücken können auch diese mit dem damit verbundenen Grad an Flexibilität eingesetzt werden.

2.4.3 Eigenschaften des GUI-Test Frameworks kennen

Die letzte Gruppe an Fehlern lässt sich vermeiden, indem man die Eigenschaften des GUI-Test Frameworks kennt. In dieser Fallstudie also QF-Test und die Eigenschaften bei der Wiedererkennung von Komponenten. Bei QF-Test handelt es sich um einen wahrscheinlichkeitsbasierten Ansatz. Anhand von bestimmten Eigenschaften wird für jede Komponente eine Wahrscheinlichkeit berechnet, dass es sich um die gesuchte Komponente handelt. Die Komponente mit dem höchsten Ergebnis wird schließlich ausgewählt. Dabei wird der eindeutige Name der Komponente, der beispielsweise mittels *setName()* gesetzt werden kann besonders stark gewichtet.

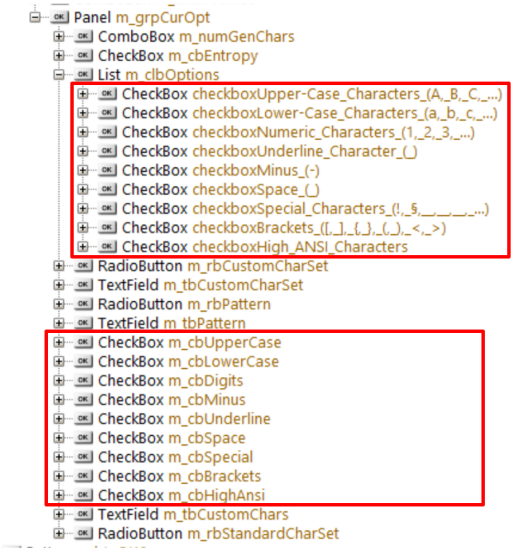


Figure 2.4: Komponenten mit geänderten Namen ohne fachliche Änderungen im SUT

Wie in Abbildung 2.4 zu erkennen wurden in KeePass während der Entwicklung die Namen von Komponenten geändert. Sonstige Eigenschaften dieser Komponenten änderten sich hingegen nicht, wodurch diese Änderung optisch nicht zu erkennen ist. Jedoch sind die Tests mit QF-Test und der verwendeten Gewichtung fehlgeschlagen.

Insgesamt sind sechs der 17 fehlgeschlagenen Tests durch geänderte Namen oder einen anderen Grund fehlgeschlagen. Auch diese Fehler lassen sich leicht vermeiden, indem die Namen der Komponenten zu Beginn konsistent gesetzt werden.

3 Schluss

Das Ergebnis der Fallstudie ist insgesamt deutlich. So sind 14 der 17 fehlgeschlagenen Tests durch mindestens einen der drei Fehler gescheitert. Lediglich die restlichen drei fehlgeschlagenen Tests liegen außerhalb dieser Fehlerklassen. Natürlich können diese Ergebnisse nicht ohne Einschränkungen auf jedes beliebige Softwareprojekt übertragen werden. Die hier gezeigten Klassen an Fehler sind sehr spezifisch und gelten im Gesamten nur für die Bedingungen der Fallstudie. Inwiefern die Ergebnisse auf andere Softwareprojekte übertragen lassen ist möglicher Gegenstand anderer Arbeiten. Mögliche Änderungen an den Bedingungen umfassen beispielsweise das verwendete GUI-Test Framework, das SUT, die Versionen, die Tests und Testcases, das verwendete Betriebssystem und viele weitere.

Jedoch hat sich deutlich gezeigt, dass durch wenige Fehler in den GUI-Tests ein großer Anteil dieser zerbricht. Es lässt sich also die Robustheit von automatisierten GUI-Tests erhöhen, indem Fehler innerhalb der ermittelten Fehlerklassen vermieden werden. Als direkte Folge davon sinkt der Wartungsaufwand für die GUI-Tests, was in jedem Fall wünschenswert ist.

Bibliography

[BH16] Dr. Benedikt Hauptmann. *Reducing System Testing Effort by Focusing on Commonalities in Test Procedures*. PhD thesis, Technische Universität München, 2016. page: 17.

[ISO13] ISO/IEC/IEEE. Standard 29119-1: Software and systems engineering - software testing - part 1: Concepts and definitions, 2013.

[MS03] Atif M. Memon and Mary Lou Stoffa. Regression Testing of GUIs, 2003.

[MS04] G. J. Myers and C. Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. page: 104.