FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automatisierung von Heuristikbasierte Usability-Inspektion mittels GUI-Event Sequenzierung

Diplomarbeit

in die teilweise Erfüllung der Anforderungen zur Erlangung des Grades

Diplom Ingenieur

in

Software Engineering und Internet Computing

von

Amir Banaouas
Matrikelnummer 0927741

an die Fakultät für Informatik
at TU Wien

Betreuer: Thomas Grechenig
Assistent: Stefan Taber

Wien, Oktober 10, 2019

(Unterschrift der Verfasser) (Unterschrift der Betreuer)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automation of Heuristic-based Usability Inspection using GUI Event Sequencing

Master's Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Amir Banaouas
Registration Number 0927741

to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig
Assistance: Stefan Taber

Vienna, October 10, 2019 ____________________ (Signature of Author) ____________________ (Signature of Advisor)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub

Automation of Heuristic-based Usability Inspection using GUI Event Sequencing

Master's Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Amir Banaouas
Registration Number 0927741

elaborated at the
Institute of Information Systems Engineering
Research Group for Industrial Software
to the Faculty of Informatics
at TU Wien

Advisor: Thomas Grechenig
Assistance: Stefan Taber

Vienna, October 10, 2019

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Statement by Author

Amir Banaouas
Lerchengasse 28-30/1, 1080 Wien

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

____________________________ ____________________________

(Place, Date) (Signature of Author)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I would first like to thank my thesis advisor Professor Thomas Grechenig and his Assistant Stefan Taber of the Institute of Computer Aided Automation Research Group for Industrial Software to the Faculty of Informatics at the Vienna University of Technology.

A very special gratitude goes out to Assistant Taber for his valuable guidance whenever I faced a challenge, for finding time whenever I asked for a meeting, and for his highly appreciated patience whenever I was behind schedule. He consistently pushed me in the right direction and taught me how to improve my work.

I am very grateful to all who reviewed the thesis, and in particular Christoph Wimmer for his important feedback on the sections dealing with usability engineering. A special thanks goes as well to Dr. Brigitte Brem whose advice helped shape the final version of the work.

Also, I would like to thank the team behind Micro Focus, and especially Tal Halperin, for valuing the study and extending my trial license for their popular tool Unified Functional Testing (UFT).

Finally, my utmost gratitude and respect is expressed to the Vienna University of Technology for teaching me so much about the joy and sorrow that comes with pursuing a dream, about discipline and hard work, and for changing my way of thinking to see new chances and opportunities.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

The usability degree of a Graphical User Interface (GUI) might deteriorate after a design change, and detecting this quality loss quickly can be challenging. Automating heuristic-based usability inspection even partially could reduce the efforts required for sustaining an appropriate level of usability. On the other hand, GUI testing is a field highly conversant with automation. The purpose of this study is to show that even though GUI testing and usability testing are two distinct fields, they might influence each other. It is theorized that the automation of heuristic evaluation can be greatly improved thanks to GUI event sequencing, a test automation technique.

To that end, an existing heuristic set is selected and tailored to fit the context of Windows desktop applications. The derived set is then analysed in accordance with the feasibility of its automation. The examined guidelines are classified into three categories: Those suitable for automation with GUI event sequencing, those that can be verified automatically without sequencing events, and those appropriate for manual testing. Then, a sample of heuristics from the first category is evaluated in a GUI testing tool to validate the feasibility of their automation in practice.

The results showed that 55% of the heuristics could be tested automatically with event sequencing, and that 75% of them could be automated in general. This indicates a potential to greatly decrease the manual work of the usability evaluator. Moreover, a pattern was noted among guidelines from the first category. Their evaluation requires at least one user-system interaction and does not focus on attributes uncatchable by GUI testing tools such as the aesthetic value. These results imply that a heuristic set can be arranged in advance so it consists only of guidelines suitable for automation.

Keywords

Automated usability testing, automated heuristic evaluation, GUI testing, event sequencing, event-driven, Windows desktop application.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Der Nutzungsgrad einer grafischen Benutzeroberfläche (GUI) kann sich nach einer Designänderung verschlechtern, und das Erkennen dieses Qualitätsverlustes kann schwer sein. Die Automatisierung der auf heuristikbasierten Usability-Inspektion könnte sogar den Aufwand für die Aufrechterhaltung eines angemessenen Grad der Benutzerfreundlichkeit verringern. Andererseits ist das GUI Testen Bereich mit der Automatisierung vertraut. Das Ziel dieser Arbeit ist es zu zeigen, dass GUI und Usability Testen sich jedoch gegenseitig beeinflussen können. Es wird theoretisiert, dass die Automatisierung der heuristischen Evaluierung, dank der GUI Automatisierungstechnik GUI-Event Sequenzierung, erheblich verbessert werden kann.

Zu diesem Zweck wird ein vorhandener Heuristik-Set ausgewählt und an den Kontext von Windows Desktopanwendungen angepasst. Der resultierende Set wird dann entsprechend der Durchführbarkeit seiner Automatisierung analysiert. Die untersuchten Richtlinien werden in drei Kategorien unterteilt: Die für die Automatisierung mit GUI-Event Sequenzierung geeignet sind, diejenigen, die ohne GUI-Event Sequenzierung automatisch überprüft werden können, und die, die für manuelle Tests geeignet sind. Anschließend werden manche Heuristiken der ersten Kategorie in einem Testwerkzeug bewertet, um die Machbarkeit ihrer Automatisierung in der Praxis zu überprüfen.

Die Ergebnisse zeigten, dass 55% der Heuristiken automatisch mit der GUI-Event-Sequenzierung getestet werden konnten und allgemein 75% davon automatisierbar sind. Dies deutet auf ein Potenzial hin, das die manuelle Arbeit des Usability-Evaluators erheblich reduziert werden kann. Darüber hinaus wurde ein Muster unter den Richtlinien der ersten Kategorie festgestellt. Ihre Bewertung erfordert mindestens eine Benutzer-System Interaktion und konzentriert sich nicht auf Attribute, die von Testwerkzeugen nicht erfasst werden können. Diese Ergebnisse implizieren, dass ein Heuristik-Set vorab angeordnet werden kann, so dass es nur aus Richtlinien besteht, die für die Automatisierung geeignet sind.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

1 Introduction **1**
1.1 Problem Statement 1
1.2 Motivation 2
1.3 Aim of the work 4
1.4 Structure of the work 4

2 Basics of Graphical User Interfaces and Usability Engineering **6**
2.1 Graphical User Interfaces 6
2.1.1 User Interface and Interaction Design 6
2.1.2 Components of Graphical User Interfaces 8
2.2 Usability Engineering 10
2.2.1 Definition of Usability 10
2.2.2 Definition of Usability Engineering 10
2.2.3 Usability Metrics 11
2.2.4 Usability Criteria 13

3 Fundamentals of Software Testing **16**
3.1 Definitions and Basics in Software Testing 16
3.1.1 Errors, Faults and Failures 16
3.1.2 Testing Process 17
3.1.3 Testing Techniques 18
3.1.4 Testing Methods 21
3.1.5 Testing Types 24
3.1.6 Testing Levels 26
3.1.7 Test Automation 28
3.2 GUI Testing 31
3.2.1 Definition of GUI Testing 32
3.2.2 Aim of GUI Testing 32
3.2.3 Benefits of Automating GUI Testing 32
3.2.4 Methods of Automating Testing GUIs 34
3.2.5 Limitations of GUI Test Automation 36
3.3 Usability Evaluation 37
3.3.1 Defining Usability Evaluation 37
3.3.2 Benefits of Usability Tests 39
3.3.3 Methods of Usability Evaluation 41
3.3.4 The Need for Automating Usability Evaluation 49
3.3.5 Toward Automating Usability Evaluation 50
3.3.6 Limits of Usability Evaluation 57

4 Automating Heuristic-based Usability Inspection with GUI Event Sequencing **58**
4.1 Comparing GUI and Usability Evaluation 58
4.1.1 Similarities between GUI Testing and Usability Evaluation 58
4.1.2 Differences between GUI Testing and Usability Evaluation 59
4.1.3 Review of GUI Test Automation Tools 61

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1.4 Review of Usability Evaluation Tools 64
4.1.5 GUI Testing Tools Versus Usability Tools 66
4.2 Evolution of Heuristics 66
4.2.1 Ergonomic Roots of Usability Heuristics 67
4.2.2 Traditional Versus Modern Heuristics 68
4.3 GUI Event Sequencing Potential 70

5 Proof of Concept for Automating Heuristic-based Usability Inspection with GUI Event Sequencing 71
5.1 Summary of the Problem Addressed 71
5.2 Deriving and Structuring adequate Heuristics 71
5.2.1 Filtering Usability Heuristics for Windows Applications 72
5.2.2 Listing Heuristics Compatible with Automation by Event Sequencing . . 73
5.2.3 Listing Heuristics Fitting for Automation without Sequencing 83
5.2.4 Listing Heuristics Suitable for Manual Testing 86
5.3 Development of Heuristic Evaluation in a Testing Tool 90
5.3.1 Preparing Needed Development Environment 90
5.3.2 Instances of Usability Heuristics Evaluation in a GUI Test Automation Tool 91
5.4 Evaluation of the Results 104
5.4.1 Assessment of the outcome of the Heuristics Filtering Process 105
5.4.2 Analysis of Automation Feasibility in Heuristic Evaluation 107

6 Conclusion 111

Bibliography 113
References 113
Tool-Related Web References 121

A Appendix: Additional UFT Flow Diagrams of automated heuristic evaluations 125

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

1.1 Comparative snapshots of the tabbed pane in the configuration settings window of the Dolphin emulator under the English (left) and German (right) localised versions . . . 3

2.1 The stages of the usability engineering lifecycle (see [34]) 12

3.1 Curve showing percentage of found usability issues in relation with the number of participants (see [85]) 43
3.2 Reperesentation of usability problems discovered by evaluators in a heuristic evaluation case study (see [97]) 46
3.3 Curve showing the return on investment in relation to the number of evaluators in a heuristic evaluation (see [97]) 47
3.4 Partial snapshot of a user sample test from "UserTesting" delivered with attached relevent notes and observations (see [109]) 52

4.1 Screenshot of a pagination area under search results retrieved from the Ebay website 70

5.1 UFT flow diagram for a general test case in Eclipse Photon with an expanded view of its nested actions 92
5.2 UFT flow diagram for a general test case in MyFlight application 93
5.3 UFT flow diagram for guideline 1.0/4 "Fast Response" 94
5.4 UFT flow diagram for guideline 1.3/10 "Upper and Lower Case Equivalent in Search" 95
5.5 Side by side view of the tabbing numeric order in the login windows of Steam (left), and MyFlight application (right) 97
5.6 UFT flow diagram for guideline 2.7.5/3 "User-Specified Windows" with an expanded view of its nested actions 98
5.7 An instance of an Eclipse custom Java perspective 98
5.8 The help window of the Eclipse Integrated Development Environment (IDE), prompted while testing guideline 3.0/1 Flexible Sequence Control 99
5.9 The three states of the "Order" button ("disabled" on the left, "enabled" in the middle, and "invisible" on the right) 99
5.10 UFT flow diagram for guideline 3.0/20 "Indicating Control Lockout" 100
5.11 UFT flow diagram for guideline 3.2/10 "Only Available Options Offered" 101
5.12 The availability of the "Search" button in the "Search" window of Eclipse Photon . . 101
5.13 Screenshot of the Netflix desktop application when started up without Internet connection 102
5.14 UFT flow diagram for guideline 4.3/13 "Cursor Placement Following Error" 103
5.15 UFT flow diagram for guideline 6.0/5 "Protection from Interrupts" 105
5.16 Confirmation message of a to be deleted booking order in the MyFlight application . 105
5.17 Overall classification results after analysis of test automation feasibility on a derived set of adequate heuristics 108

A.1 UFT flow diagram for guideline 1.4/15 "Explicit Tabbing to Data Fields" 125
A.2 UFT flow diagram for guideline 1.7/3 "Non-Disruptive Error Messages" 126
A.3 UFT flow diagram for guideline 3.0/1 "Flexible Sequence Control" 126
A.4 UFT flow diagram for guideline 3.1.3/7 "Menu Selection by Keyed Entry" 127

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

A.5 UFT flow diagram for guideline 3.3/3 "Cancel Option" 127
A.6 UFT flow diagram for guideline 3.5/10 "UNDO to Reverse Control Actions" 128
A.7 UFT flow diagram for guideline 6.0/18 "User Confirmation of Destructive Actions" . 128

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

2.1 Ergonomic dialogue principles from ISO 9241-110 and their corresponding description (see [23]) . . . 7
2.2 Attributes of information presentation as recommended by ISO 9241-12 (see [24]) . 8

4.1 Comparative list of active tools for automated GUI testing . . . 62
4.2 Comparative list of active tools for usability evaluation . . . 65

5.1 Usability guidelines selected for practical experiments . . . 91
5.2 Guidelines selection data grouped by functional interaction area . . . 106
5.3 Feasibility of heuristic evaluation automation grouped by functional area . . . 108

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Listings

5.1 Code Snippet with the GUI event sequence verifying login responsiveness [1] . . 94
5.2 Code Snippet with the GUI event sequence checking tabbing navigation in the login window [1] 96
5.3 Source code with the GUI event sequence verifying data protection from an interrupting action [1] 104

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Abbreviations

ACTA Applied Cognitive Task Analysis

AI Artificial Intelligence

AMME Automatic Mental Model Evaluator

ASQ After Scenario Questionnaire

CAMPUS Cognitive-Affective Model of Perceived User Satisfaction

CDM Critical Decision Method

CFM Cognitive Function Model

CLI Command Line Interface

CSS Cascading Style Sheet

EPL Eclipse Public License

ESD Electronic System Division

ETIT External Internal Task Mapping

GOMS Goals, Operators, Methods, and Selections

GUI Graphical User Interface

HPE Hewlett Packard Enterprise

IBAN International Bank Account Number

IDE Integrated Development Environment

ISO International Organization for Standardization

ISTQB International Software Testing Qualifications Board

IxD Interaction Design

NNG Nielsen Norman Group

QUIS Questionnaire For User Interaction Satisfaction

SUMI Software Usability Measurement Inventory

SUPR-Q Standardized User Experience Percentile Rank Questionnaire

SUS System Usability Scale

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TKS Task-Knowledge Structures

UFT Unified Functional Testing

UI User Interface

UME Usability Magnitude Estimation

UX User Experience

WIMP Windows, Icons, Menus, and Pointers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1 Introduction

1.1 Problem Statement

Software testing is a task, necessary for ensuring the high quality of an application, especially, in a world of changing requirements. Many forms of testing exist with diverse goals, and various techniques, suitable for different stages of development. One of these forms is Graphical User Interface (GUI) testing (and it is covered in details by section 3.2). Since applications today have a significant proportion of GUIs which are becoming larger and getting more complex, the need for automating said process is always growing. In contrast, manual testing of big applications is very slow and resource-expensive. For instance, even a simple application such as Microsoft WordPad has more than 324 possible GUI operations [2]. For a more complex program, a manual GUI testing approach could lead to serious issues especially when testing the application's functionalities after major software updates [3]. Considering how frequently software goes through changes during development, the demand for GUI automation increased on web, mobile and desktop platforms in response to this problem. This led to the creation of many GUI automation testing tools, saving testers a lot of time and effort (such as Selenium in case of web applications [4], Ranorex for Windows applications [5], and the Android Testing Support Library for mobile Android applications [6]).

On the other hand, one quality attribute that can greatly enhance the GUI of an application is usability. The International Organization for Standardization (ISO) defines usability as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [7]. Usability evaluation is the practice of assessing the usability of a software application (and is detailed more in section 3.3). Generally, testing the usability revolves around observing and evaluating the interaction between users and interfaces, then it's followed by suggestions for enhancing the quality of the GUI through certain improvements that boost the efficiency and elegance of the interaction. Moreover, the usability evaluation process is dominated by manual aspects. Acquiring accurate usability test results require a cooperating user sample (of a sufficient size), and a suitable lab environment (which can be quite expensive) [8].

In the case of mobile and web applications, some tools can assist testers by providing them data that may compensate for a lack of a lab and a test sample such as, for instance, Google Analytics reports [9] [10]. On the other hand, desktop programs benefit less from such analytical tools as user feedback is less frequent than in web and mobile applications, and is generally limited to error reporting. Therefore, observing user behaviour on windows applications using an analytical tool is more challenging, as well as its collected data being less relevant.

A more resource-friendly usability evaluation approach is heuristic evaluation, defined by Jakob Nielsen, as a discount usability engineering method where a set of evaluators inspects a user interface with respect to a small set of fairly broad usability principles, which are referred to as the heuristics in order to identify its usability problems [11]. Contrary to a usability testing method where problems are identified through the observation of user and system interaction, heuristic evaluation is considered an easy to perform and cheap inspection method that can find many major and minor usability problems [12]. However, heuristic inspections are still mainly performed manually despite not requiring end users in the evaluation.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Therefore, automating the inspection process would reduce the evaluation time and the needed resources, which would by extension, lower designing and testing costs, and contribute to shortening iterative design cycles (which are discussed more in section 2.2.2) [13]. Furthermore, the automation would help reduce the risk of usability loss after each major software update. Additionally, manual heuristic-based inspection performed by different evaluators might lead to inconsistencies in the usability issues detected, which can be caused by misinterpretation of usability guidelines or due to applying them in the wrong context of use [14]. Automation would overcome such inconsistencies introduced by limitations in the human element, and bring more stability to the testing process [15] [16]. Furthermore, once enough guidelines are checked automatically, the dependence on the presence of usability experts for conducting every single part of the inspection would be greatly reduced.

Unlike the circumstances of other usability engineering methods, usability tools currently on the market do not seem to focus on further automating inspection methods, but rather mainly focus on capturing usability-relevant data from the observed user sample (as is explained in section 4.1.4). In the academic field, previous research attempts at automating the inspection method focused on identifying aesthetic or navigational usability issues more so than function-oriented problems [17] [18] [15] [13] [16]. For instance, some usability tools might tell if text fields are not aligned properly, if the font used is too small or if two buttons are too far apart, but they cannot, for instance, test if a functionality can be undone after its completion, or if the sequence of possible user actions can be different for beginners and expert users.

In fact, any task requiring the accomplishment of a series of GUI events is not supported in the usability tools currently on the market or those created for academic research [17] [18] [15] [13] [16]. Sequencing GUI events is traditionally considered an area for GUI automation testing tools, and a lot of these have event-driven sequencing capabilities as long as the tool in question supports script-based GUI testing.

All points considered, it appears reasonable to assume that GUI test automation techniques such as event sequencing may positively impact the way heuristic evaluation is performed. By enclosing the context of use to cover only Windows applications, it becomes simpler to gather a list of stable and general usability heuristics, whose automation feasibility would be analysed with and without the use of GUI event sequencing. Automating this traditionally manual aspect may provide a notable improvement in the usability engineering field. The problem, however, is that not all usability heuristics are suitable for automation, and even if they were, not all of them require sequencing GUI events. Therefore, it's needed to select a minimalistic, relevant, and clear set of heuristics for Windows applications, examine them, and re-categorize them into three groups: Those that should only be checked manually, those that can be automated without requiring GUI event sequencing, and finally those that can benefit from sequencing some GUI events in a similar manner to the way a GUI test automation tool does. Afterwards, a GUI test automation tool, and some software applications (suited for a Windows desktop environment) would be selected and then used to detect some guideline violations with event sequencing in practice, in order to affirm the correctness of the analysis.

1.2 Motivation

In general, automating a testing process greatly reduces the time, cost, and resources needed for testing, as long the cost of maintaining the automated tests does not exceed the cost of manual tests. Naturally, the same goes for GUI testing and usability evaluation. GUI test automation benefits are discussed in details in section 3.2.3, while the advantages of automating usability evaluation are covered in section 3.3.4. However, one additional benefit is gained, in particular,

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

when a GUI testing technique is successfully used for assisting in the automation of a usability inspection. That is to say that the two fields would be able to share GUI event sequencing as a common testing technique, despite their many differences (which are detailed in section 4.1.2). This might lead to researching more aspects that can be brought to usability engineering from other software practices to push automation efforts further.

Currently, with GUI testing tools, it's possible to detect functionality loss. However, there's no tool in existence that checks if an application had lost some degree of usability after a major software update. Moreover, there's also no tool or testing technique that ensures a software has the same degree of usability across all its different localised versions. For instance, figure 1.1 shows the tabbed pane component present on the configuration window of the same software "Dolphin Emulator", but under two localised versions (English on the left and German on the right). The figure includes a red circle bringing attention to a usability issue on the German version that does not exist on the English version (which has the same respective area circled in green). Because German words are often longer than their English counterparts, a usability problem emerged. The "path" tab is not visible on the tapped pane. It requires horizontal scrolling to be seen, and the scroll bar is so short that it almost looks like two small buttons. Furthermore, the shown configuration window cannot be resized by users. If the problem has been detected, it could have been solved by increasing the width of the German window, or simpler yet, by allowing users to resize the configuration window a little more. However, performing separate usability evaluations on each localised versions greatly increases the testing cost, and would require the presence of usability experts fluent in each language used. Therefore, implementing automated usability inspections would help detect usability issues emerging from such situations. In this particular case, automation would solve this, for instance, by checking if all the tabs in the tabbed pane are visible (no matter the localised version).

Figure 1.1: Comparative snapshots of the tabbed pane in the configuration settings window of the Dolphin emulator under the English (left) and German (right) localised versions

On top of helping assess the usability of a completed GUI. The automation of heuristic-based inspection can assist in earlier development stages, and in particular, during GUI prototyping. The same applies for iterative design, which is an important stage of usability engineering (described in section 2.2.2). If it were possible to run an automated usability inspection after creating each prototype. The design cost would be greatly reduced, and the time needed for assessing the same heuristics later on would be shortened (if not eliminated). The overall development cost would go down as well, especially when considering that according to Jakob Nielsen it's a best practice to spend around 10% of a project's budget on usability [19].

Relying on a user sample for usability evaluation is the basis of most evaluation methods in the field. However, it's not the only way for testing usability. Despite that, the market is dominated by tools for assisting in user-oriented evaluations, while heuristic-based inspections are being marginalised. Moreover, some usability engineers are generally not enthusiastic about the potential of automation in usability evaluation because they consider end users to be irreplaceable. Analysing and providing new usability automation options, is an attempt to increase interest in cheaper and faster usability assessments.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

1.3 Aim of the work

The aim of this thesis is the analysis and research of the effect of GUI event sequencing on the automation of inspection-type usability methods, and more specifically, heuristic evaluation. The goal is to lessens the burden on usability evaluators by reducing the sum of needed manual tests. For instance, usability heuristics cover checking the visibility of system status, recovery from errors, as well as flexibility and efficiency of use. These sorts of guidelines are broad rules of thumb and their automation may vary in difficulty depending on the context of use.

To that end, the theoretical part of the thesis would examine a complete set of heuristics for designing user interfaces on Windows desktop applications, and then argue the feasibility of the automation of their inspection with and without GUI event sequencing. Similar heuristics would be gathered into different categories according to their compatibility with automation. Afterwards, a GUI test automation tool capable of event sequencing under a Windows environment would be selected, alongside some software applications to be partially evaluated, then the automation of some previously gathered usability heuristics would be tested using said tool. By documenting, implementing, and evaluating some heuristics as part of an experiment, the practicality of event sequencing is challenged.

However, all efforts related to automating heuristic-based inspection is limited to assisting the evaluator. The aim of this study is not to totally replace the human expert, because even an automation tool can make a mistake, or can fail to detect an obvious issue due to an irregularity in the application. At the end, the usability evaluator's own judgement is solely responsible over the relevance of the results of any kind of automated inspection. Therefore, the purpose of the study is not to find a way for replacing all manual work, but rather it's comprehending how to decrease it as much as possible.

1.4 Structure of the work

This thesis is compromised of six chapters (including this one), which are briefly described in the following:

- Chapter 1: The first section is the introduction, it describes the general topic of this thesis and provides some background on the current situation of usability evaluation including the insufficient efforts toward automating heuristic evaluation. Then, a solution is proposed and its benefits are discussed.
- Chapter 2: The second chapter is called "Basics of Graphical User Interfaces and Usability Engineering", and it covers the essentials needed to comprehend how GUI is designed, as well as what the basic concepts in Usability are.
- Chapter 3: The third part is titled "Fundamentals of Software Testing". It provides the groundwork for understanding key concepts in testing software in general, and testing GUI and usability in particular.
- Chapter 4: The fourth section is referred to as "Automating Heuristic-based Usability Inspection with GUI Event Sequencing". This chapter examines in details the relation between GUI testing and usability evaluation, and reviews some of their respective tools. It is followed by a discussion of the origins of heuristics, how they are deduced, and how they get updated. Finally, the hypothesis questioned in this thesis is presented.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Chapter 5: The fifth chapter is titled "Proof of Concept for Automating Heuristic-based Usability Inspection with GUI Event Sequencing". This section derives a relevant group of guidelines from a sizeable heuristic set, and categorises them into different classes according to their focus, and the feasibility of their automation. Then, their classification is validated by automating a representative heuristic sample, in the tool best suited for this task according to the information gathered in the fourth chapter.
- Chapter 6: The final section is the conclusion, it summarises the results of the study, and the lessons learned. Also, it proposes suggestions on how to further research the subject of automating usability inspections.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2 Basics of Graphical User Interfaces and Usability Engineering

With software demand increasing and with technology evolving, different ways of interacting with computers have been created. It is important to comprehend the basics of this interaction and why it is preferred to design it so it is easy for users to work with.

2.1 Graphical User Interfaces

Nowadays, machines can not only be used, but can also be interacted with. This type of interaction or communication between human and machine can be broad, and encompass multiple fields such as design, multimedia, computer science, and behavioural studies. In order to get a narrow view on this interaction, it's important to examine the technical vocabulary associated with the subject.

2.1.1 User Interface and Interaction Design

A user interface is the portion of software where interaction between human and machine happens, giving effective control to the human side, and displaying for him simultaneous feedback for his actions from the machine side, thus, assisting him in the cognitive process involved with solving problems or conducting activities [20]. A GUI is a class of user interfaces that facilitates human-computer interaction through the use of graphics and visual indicators such as windows, icons, and menus. These are usually manipulated through a pointing device such as a mouse or stylus, or through a touchscreen [20]. Unlike Command Line Interface (CLI), which only relies on text and is accessed by a keyboard, GUI uses a consistent set of graphical elements, which improves software learnability by not requiring the user to learn additional commands for each program he's using.

User Interface (UI) design is the process of visually guiding the user through a GUI via interactive elements. UI design focuses on users, their needs and their requirements, in order to make software most efficiently usable and useful. This process aims at improving software effectiveness and efficiency, as well as maximizing user satisfaction by properly taking account of humans factors and ergonomics. A UI designer is not only responsible for implementing the UI with developers, but also for UI prototyping, ensuring correct interactivity, and adding proper responsiveness to different device screen sizes. From this background, the process of improving the quality of interaction between a user and all facets of a company is called User Experience (UX) design. A more broad definition would be the creation and synchronisation of the elements that affect user experience with a particular company, with the intention of influencing user perspective and behavior [21]. These elements do not stop at the GUI but also include what the user can physically touch (such as product packaging), or hear (such as commercials). This is a cognitive science defined predominantly by digital companies despite theoretically being a non-digital practice. UX design incorporates essential characteristics of UI design, Interaction Design (IxD), user research, information architecture, and other disciplines. From this angle, IxD is a subset of UX design examining interactive software systems. Even though IxD is based in theory, practice and methodology on UI design, its focus is on defining the complex dialogues that happen between users and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Design Principle	Description
Suitability for the task	A dialogue is suitable for a task to the degree that it assist users effectively and efficiently in completing tasks.
Suitability for learning	A dialogue is fit for learning to the degree that it supports and guides the user in performing his tasks during the learning phase.
Suitability for individualisation	A dialogue is fit for individualisation to the extent that it allows its own modification for a certain task, in order to best suit the user's needs or skills.
Conformity with user expectations	A dialogue matches user expectation to the degree that it conforms to common conventions and behaves according to the user's experience with the task.
Self descriptiveness	A dialogue is self-describing to the point that everyone of its contained steps is completely and immediately comprehensible through feedback from the information system, or in response to user queries.
Controllability	A dialogue is sufficiently controllable when the user can maintain direction across the whole interaction until the point where the he achieves his goal.
Error tolerance	A dialogue is tolerant of errors to the point that the results expected from performing a task might be achieved, with minimal to no corrective action, despite entering evident erroneous input.

Table 2.1: Ergonomic dialogue principles from ISO 9241-110 and their corresponding description (see [23])

interactive systems. IxD is by its nature contextual, it solves a particular issue under a specific context using the available resources [22].

Given the particular background, the ISO 9241-110 sets forth the ergonomic requirements which apply to the design of dialogues between humans and information systems [23]. This set of principles, listed and described in table 2.1, is presented in general terms without considering the situation of use, application, environment or technology, and does not consider aspects such as marketing, aesthetics and corporate design.

The information organized and coded in a GUI should display the presentation attributes recommended by ISO 9241-12 [24]. The way information is arranged, aligned, grouped, its labels, and its location, are all forms of information organization. The shape, size, and the color of data are part of information coding. Information presentation contain both information oranization and coding. The presentation attributes from ISO 9241-12 contribute to the application of the dialogue principles from ISO 9241-110, and especially to the principle of conformity with user expectations [20]. These attributes are listed and described in table 2.2, and represent the static aspect of a GUI (also called the "look" of the interface).

Good UI design have to contribute directly and indirectly in guiding user interaction. ISO 9241-13 provides recommendations, and describes how to best use prompts, feedback, status information, error management, and on-line help in the context of user guidance [25]. The primary principle governing this aspect of UI design is maintaining consistency. Design consistency implies pre-

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Presentation Attributes	Description
Clarity	The information content is relayed to the user quickly and accurately.
Discriminability	The displayed information can be accurately distinguished.
Conciseness	Users are not overburdened with irrelevant or inapplicable information.
Detectability	The attention of the user is directed towards the information he seeks.
Legibility	The content can be read with ease.
Comprehensibility	The meaning of the presented information is clearly understandable, recognizable, and unambiguous.

Table 2.2: Attributes of information presentation as recommended by ISO 9241-12 (see [24])

dictability of system response to user inputs [26]. Every action made by the user should result in a noticeable feedback from the computer.

2.1.2 Components of Graphical User Interfaces

GUI components are visual elements offering consistent information representation. One method of classifying said elements is to focus on what they represent. Therefore, components representing generic information by applying conventions are called structural elements, while components representing the state of an ongoing interaction operation are described as interaction elements [27].

Structural GUI elements define the entire appearance or look of the interface, and contains windows, menus, icons, controls and tabs [27]:

- Windows: A window is a display area that could present some content or encompass other GUI elements on the screen. The screen can be divided into different areas, each area representing information in a window, whose content can be displayed independently from other windows or the rest of the screen. Each window can be moved around, change in shape and size, and run a different program. Only the system memory can limit the number of windows that can be opened simultaneously.
- Menus: A menu gives the user the choice between a list of selectable executable commands. Menus are convenient and practical because they present a simplified representation of the software's features and capabilities. The offered options are mainly accessed through a pointing device and keyboard shortcuts.
- Icons: An icon is a small graphical representation that stands for digital objects such as commands, files, or windows. They are a quick way to run programs, they can enhance software learnability, and take advantage of user familiarity with similar tasks to improve user guidance.
- Controls (or widgets): These allow direct manipulation of information when interacting with an application. Each widget can make a specific user-computer interaction easier by restructuring a GUI while maintaining consistency throughout the entire information system.

TU Bibliothek Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar. The approved original version of this thesis is available in print at TU Wien Bibliothek.

- Tabs: These elements allow multiple panels or documents to be represented within a single window. Tabs facilitate the navigation between sets of documents, and add consistency to the information presentation.

Interaction elements can either show the parts of the GUI that the user can interact with, or be some form of visual reflection of the user's intent. This category mainly includes pointers, cursors, and selections [27]:

- Pointer: This element is a symbol appearing on the display screen that moves in a manner reflecting the movements of the pointing device. The pointer locates and initiates actions through direct manipulation such as with a click, a double-click, a drag-and-drop movement, or through touching.
- Cursor: This is a position indicator used to show the current position for user interaction on the display screen, or where the GUI would respond to input. While the pointer component is controlled by a pointing device such as a mouse, the cursor, on the other hand, is controlled by a text input device such as the keyboard. A text cursor is also called an insertion point or caret.
- Selection: A selection is a list of one or more items on which user operations can take place. The selection may be created automatically by the information system, or may be created manually by the user. Elements can be selected usually through a pointing device, by hand (on a touchsreen device), or through keyboard shortcuts. Selecting more than one element at a time is called multiple selection.

Another way of categorizing components focuses on their function in the GUI. In other words, an element might belong to more than one group if it has multiple functions. This method adds more clarity to the goal of the element, which can be in one of the following four classes [28]:

- Input control components: These enable doing various functions based on giving control to the user throughout the interaction. This group contains elements such as checkboxes, radio buttons, dropdown lists, list boxes, buttons, toggles, text fields, file fields, and date fields.
- Navigational components: These components define the possible ways of navigation in a GUI. This group contains elements such as search fields, sliders, pagination, and tags.
- Informational components: These elements provide feedback or useful information to the user, that assist him in the cognitive aspect of the interaction. This group contains components such as progress bars, icons, and message boxes.
- Container components: A container component gathers other elements and presents them in an organized fashion. An instance of a container component is the accordion. An accordion is a vertically stacked list of items that either show or hide functionalities. When an item is clicked on, the GUI section under it expands to show the content within. No element, one element, or more can be showing their content at a time, depending on their configuration.

Additionally, key components in a GUI might be classified into four groups on the basis of how ISO divided their recommendations: First, are menu dialogue components, which are described in ISO 9241-14 alongside their best practices [29]. Subsequently some command dialogue components. Recommendations on their usage are described in ISO 9241-15 [30]. The third group have, direct manipulation dialogue components, and they are presented in ISO 9241-16 [31]. Finally, form filling dialogue components are the forth group, and their recommendations are described in 9241-17 [32].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Usability Engineering

GUI designers and GUI developers might allow themselves the freedom to create the interfaces that they like the most, but that does not mean that the users will also like their experiences with these interfaces. Many guidelines exist for properly balancing a GUI's form with its function, and following these guidelines might make a big difference in the success of a GUI. Therefore, in order to learn how to achieve the optimal design for users, it's important to understand what usability means, and the basic concepts in usability engineering.

2.2.1 Definition of Usability

Usability is defined in ISO 9241-11 [7], as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use". Examining the definition, effectiveness answers the question of whether the product or system enables users to perform their work as anticipated during development, efficiency measures the required effort for users to complete their intended tasks, and lastly, satisfaction refers to the system or application's ease of use from the perspective of its intended users. Usability does not assess the functionality of a system, if an application increases its features and functionalities, it would not systematically signify an improvement in its usability. Finally, the term user-friendly is used to describe systems with a high usability degree.

Usability is, according to Jakob Nielsen, a quality attribute assessing UI ease-of-use, also referring to enhancement methods for that ease-of-use during the design process, and traditionally associated with multiple key quality components [8]:

- Learnability: It should be easy to learn system functionalities, to the degree that users can rapidly perform basic tasks in their first time encountering the UI.
- Efficiency of use: The software has to be efficient, in a manner allowing the user to be more productive after learning to use the system.
- Memorability: Remembering how to use the system should easy, so that a user, who became inactive for a while, can do the tasks again without first wasting time relearning them.
- Scarce and non-severe errors: The error rate of the software should be low, so that no catastrophic errors occur, and so that when less severe errors are encountered, it is easy for the system to recover from them.
- Subjective satisfaction: Using the software should subjectively be a pleasant experience to the degree that users like it (or at least don't get frustrated from using it).

2.2.2 Definition of Usability Engineering

Usability engineering is the field in which user friendliness of UIs is studied and analysed in order to produce structured methods for designing efficient and elegant interfaces. In other words, usability engineering focuses more on assessing usability and suggesting ways to improve it, rather than directly engaging the design process. This practice includes a set of activities that happen ideally throughout product or system development, and is not a one-time event. It was recognized early on in this field, that usability engineering should have multiple supplementary stages. However, this was not always followed in practice [33]. This lifecylce offers the possibility to include usability in the development process to ensure that the resulting system is user-friendly. Even if not

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

all the steps in the lifecycle are performed, usability engineering efforts can still wield successful results [8]. These stages are represented in figure 2.1, and are briefly described below [34]:

- Analysis: The analysis stage may include user analysis, task analysis, functional analysis, and competitive analysis. It should always result in concrete goal setting.
- Design: One approach for doing this is parallel design, which aims at exploring different design alternative before choosing the best one that would be further developed and fine-tuned iteratively. Another approach is participatory design which involves the users in the design process to avoid potential mismatch between actual user tasks and the model representing the tasks, which is used by the developers. No matter the approach, it's important to focus on consistency when designing.
- Prototyping: A prototype is an incomplete software version that allows the early evaluation of some considered solutions. Creating prototypes instead of implementing actual designs, saves up on time, cost and effort. Decreasing the feature quantity is called vertical prototyping, while decreasing functionality depth is called horizontal prototyping. Finally, reducing both the quantity and the depth to match a certain use case scenario is called scenario-based prototyping.
- Expert evaluation: After building a prototype, it is used to explore and evaluate one or more aspects of the usability of the system. The evaluation is done by experts, who examine how real users interact with the designed interface.
- Empirical testing: In this phase, usability is examined and tested by conducting experiments on real users, or by carrying one or any combinations of usability evaluation methods. This stage aims to identify and analyse usability issues, their severities, and their causes.
- Iterative design: Based on the usability problems, identified by the empirical testing, a new interface is designed. That design is then discussed with usability experts, and thoughtfully analysed by checking its conformity with usability heuristics. If the issue persists, the interface is redesigned. Major iterations can benefit from additional tests relying on real users. This iterative process continues until the usability problem is deemed as resolved.
- Feedback: After the system is released and exposed to a high number of users, more information about the system's usability can be gathered. This data can come from logging files, secondary data (such as client complaints), or special market studies. The collected feedback can form the basis for a new analysis (and thus start the cycle anew).

2.2.3 Usability Metrics

A usability metric is a way of measuring or evaluating usability in a consistent and reliable manner. By using the same method of measurement, it's possible for a company to consistently assess its competitive position compared to others. Companies can also keep track of their usability improvement progress between releases, or decide whether a system is, or is not user-friendly enough for release. Additionally, measuring usability may assist higher-level company executives in making bonus plans, such as giving bonuses for lead-developers, based on how low customer-support cost was during that year. Usability metrics can also be useful for proving that a certain interface design has a higher usability value compared to another.

The actual methods for assessing usability might differ depending on the evaluator. One way of measuring it is by focusing on the key factors defining usability which are effectiveness, efficiency and user satisfaction.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

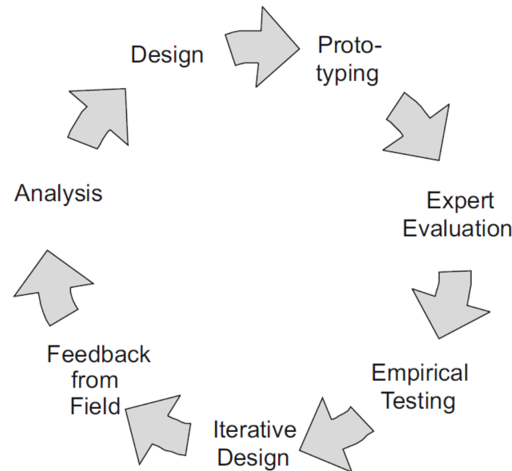

Figure 2.1: The stages of the usability engineering lifecycle (see [34])

Effectiveness: It can be measured by calculating the success rate (also called completion rate), this is one of the fundamental usability metrics. It's simple and relatively easy to perform. The success rate is represented by a percentage, and uses the following equation [35] [36]:

$$\text{Success Rate: } \frac{S}{N} \times 100$$

where:

S = Number of successfully completed tasks by the user sample

N = Total number of tasks

Effectiveness can also be measured by calculating the error rate. This can be achieved by counting the number of mistakes made by users participating in usability testing experiments. The evaluator observing the user would usually assign a severity rating to each encountered error, and classify it under a corresponding category. The average number of errors per task provides important diagnostic data about the system.

Efficiency: This attribute takes more time and effort to measured than effectiveness. The evaluator would observe users performing tasks and note how much time was needed for each activity. Time-based efficiency is one way of measurement calculated by the equation shown below [35] [36]:

$$\text{Time-based Efficiency: } \frac{\sum_{j=1}^{U} \sum_{i=1}^{N} \frac{n_{i,j}}{t_{i,j}}}{N \times U}$$

where:

U = Total number of users

N = Total number of tasks

n_{ij} = The outcome of task "i" by user "j", if he finishes the task successfully, then $n_{ij} = 1$, if he does not, then $n_{ij} = 0$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

t_{ij} = The time needed to finish task "i" by user "j". If the user gives up on completing the task, then the moment the user quits is considered for calculating the total time value.

Time-based efficiency has a unit of goals per second, where a goal represents a task performed successfully.

Another way of assessing this attribute is through calculating the overall relative efficiency. This process gives the percentage ratio of the time required to successfully perform the tasks in relation to the total time taken for all the tasks and by all users. The variables used in the equation for determining the overall relative efficiency, match the same concepts as their counterparts in the equation for time-based efficiency [35] [36]:

$$\text{Overall Relative Efficiency: } \frac{\sum_{j=1}^{U}\sum_{i=1}^{N} n_{i,j} t_{i,j}}{\sum_{j=1}^{U}\sum_{i=1}^{N} t_{i,j}} \times 100$$

Satisfaction: One way of measuring user satisfaction is to have users fill in a questionnaire immediately after they finish their attempt at completing a task. These post-task questionnaires usually consist of less than five questions, asking the user about the task's difficulty [36]. This is also called assessment of task level satisfaction. Instances of this type of questionnaires include the After Scenario Questionnaire (ASQ) with three questions [37], and the Usability Magnitude Estimation (UME) with just one question [38].

Another method of measurement focuses on test level satisfaction. This occurs after a test session concludes, and aims at assessing the participant's overall impression of the system's usability. Examples of questionnaires in this category include the Standardized User Experience Percentile Rank Questionnaire (SUPR-Q) with thirteen questions, the Software Usability Measurement Inventory (SUMI) with fifty questions, the Questionnaire For User Interaction Satisfaction (QUIS) with twenty-four questions, and System Usability Scale (SUS) with ten questions. Choosing the appropriate questionnaire depends mainly on the company's budged for user satisfaction and the importance of the user's perspective on satisfaction for the overall project.

The main downside of relying on usability metrics is the cost in time and resources. According to the Nielsen Norman Group (NNG), usability quantitative measures might cost up to four times as much as performing a qualitative study [39]. Using the presented metrics is expensive, and usability resources are typically scarce. Qualitative evaluation methods provide useful insight for improving design while quantitative metrics provide numbers that assess a design and keep track of the improvement progress (These qualitative usability evaluation methods are covered in details in section 3.3.3).

2.2.4 Usability Criteria

Usability criteria (or principles) refer to the set of standards describing how a system can achieve a high level of usability. Jakob Nielsen presented in his book Usability Engineering ten broad guidelines (also called rules of thumb or heuristics) that can be applied to any type of user interfaces [8]:

1. Visibility of system status: The system should always provide users with the information concerning what is happening on the system side and the state of their interaction. This

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

information should be relevant to the context of use in its entirety, because any part of that information that is not needed reduces the visibility of the useful part. Additionally, all feedback should be sent within a reasonable time.

2. Match between system and the real world: The information system should communicate with the user by familiar means. System-oriented words, phrases, concepts, or even symbols that are unfamiliar to the user would cause confusion and should therefore be avoided. Real-world conventions make information appear natural and logical. For example, a developer might appreciate knowing if a system status error has the 404 code or the 500 code. However, the average end-user does not comprehend the meaning of such status error codes, and should be spared the confusion.

3. User control and freedom: Users who make mistakes should, whenever possible, have the option to easily undo their activity if possible, as well as the option to redo it. Users should also always be able to terminate an operation.

4. Consistency with standards: Conventions should be consistently followed, to prevent users from making mistakes or getting frustrated, by not comprehending whether various words or actions have the same meaning. For instance, users are accustomed to treating animated text as an advertisement, or seeing hyperlinks as underlined blue text. Breaking such conventions, by building an animated flashing button that works like a hyperlink would only confuse users. Another example would be the common practice of displaying an asterisk near required text fields when a user is filling a form. Giving the asterisk symbol another meaning will only drop the usability value of an interface.

5. Error prevention: This can be done by eliminating error-prone conditions or checking their validity before letting the user confirm he's committing an action. For instance, using a date picker for a certain field will assist users in knowing what date convention to choose.

6. Recognition rather than recall: Memorizing prior shown information should not be expected of users while they are progressively working on a task, or when performing tasks between long time intervals. Minimizing what the user have to remember can be achieved by making actions, objects, and options visible. Additionally, the user should easily find instructions for use of the system whenever appropriate.

7. Flexibility and efficiency of use: An instance of this would be shortcuts (or accelerators). These elements may assist the expert user in performing frequent tasks faster or easier, while shielding unneeded complexity from novice users. This way, the system can attend the needs of both beginner and expert users.

8. Aesthetic and minimalist design: The screen layout and colors should be simple and visually appealing. Also, information of low or no relevance to users should not be displayed on the dialogues.

9. Assist users in recognizing, diagnosing, and recovering from errors: The system should express error messages in plain language, precisely describe the problem, and suggest a constructive solution.

10. Help and documentation: User documentation (or user guide) and help for the system should be complete, and it should also be easy to search for information in them. The help should be context-sensitive and user-oriented. The most common tasks must be explained in concrete steps, and not be too long.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Since these criteria are presented with enough abstraction, their application can be generalised on all types of UIs. Additionally, each one of them can still be composed of an additional group of guidelines, that differ depending on the context of use. For instance, the consistency with standards principle applies to GUIs for desktop, mobile and web applications. However, the standards for desktop, mobile and web applications are different. Therefore, the subset of possible proposed guidelines, which are grouped under the same principle, would differ depending on the nature of the GUI.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3 Fundamentals of Software Testing

There are different phases and different models in software development with a varying impact on the final product. However, no matter the development model used or the size of the team involved, software testing is always needed. Therefore, it's important to comprehend the fundamentals of software testing, its techniques, its types, and their significance to the users.

3.1 Definitions and Basics in Software Testing

In software development, testing is an integral section, examining the quality, the correctness, and the completeness of software implementations. It can also be stated as a subset of activities related to software quality assurance, including the validation and verification that a software program or service is behaving correctly with respect to some specified requirements [40]. Verification is the process of evaluating whether a program behaves as expected, and whether it complies with a specified regulation, requirement, or imposed condition. It is an internal process mostly to be applied on system components, rather than the end-product or service [40]. In contrast, validation evaluates whether the end-product or service meets the requirements specified by the stakeholders, in order to demonstrate that the built product fulfills its intended purpose when placed in its intended environment [40].

One of the main focuses of software testing includes running the program to be tested on selected input, and checking whether the output is correct with respect to a given specification. Consequently, analysing the correctness of a program's behaviour also requires an understanding of what makes a program incorrect. To that end, it is needed to define and explain terms such as errors, faults and failures.

3.1.1 Errors, Faults and Failures

When developing software, a human mistake might introduce an incorrect computation result called an error. The result of the computation is deemed incorrect, if the observed, measured, or computed state of the output is different from the true, specified state, or the theoretically correct condition [41].

Many types of errors exist [42]: A dynamic error (also called run-time error) is a mistake whose occurrence depends on time variations, applied to an aspect of the input. On the other hand, a static error (also called compilation error) occurs independently of time-based variations of the input. A fatal error is one that results in the total inability of a system or one of its components to function. An inherited error happens in a sequential process when an error propagates forward from a previous step to the next. A semantic error results from a misunderstanding of the meaning behind a symbol, a word, or behind the relationship between a group of symbols or words.

Errors can occur in any stage of development, and might affect the code, the models, or even the documentation. The effort put into checking for errors in the source code might depend on the programming languages used. In object oriented languages, developers might introduce errors when referencing data such as using an uninitialized referenced variable, or when declaring data, such as assigning a variable a wrong datatype or giving it a problematic length [43]. Errors also

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

happen when computing or comparing inconsistent datatypes. They are also caused by having a control flow issue such as an infinite loop, or a loop with too many or too few iterations. Furthermore, developers can make interface errors, like for instance a mismatch between the number of parameters received by a module and the number of arguments sent by other modules. Moreover, input and output errors can occur. They are produced, for example, from not opening files before use, not closing files after use, or from having insufficient available memory for treating a certain file [43].

Subsequently, a fault is the manifested result of an error within the software. It's possible for one error to cause multiple faults, or for several errors to lead to identical faults. Faults can be of two types [44]: Faults of commission, which contain incorrect information in their representation, and faults of omission which indicate a missing part in the representation that needs to be present. An example of a fault of commission would be source code representing an object containing an incorrect value, while a fault of omission might be, for instance, a representation of a transaction that produces states and values that are partially or completely not persisted. The detection of faults of omission is more challenging and requires more effort.

When executing the code corresponding to a fault, a software failure might occur. In other words, failures are mainly triggered by executing a representation of a fault. Generally speaking, a failure is an event referring to the inability of a component or system to perform a required function or deliver an expected result, in the manner specified by the requirements [41].

There are different kinds of failures [42]: A hard failure is one that leads to the complete shut-down of a system, while a soft failure allows a system to continue operating with partial capabilities. A random failure occurs in an unpredictable manner, while a systematic failure have a deterministic cause. Finally, a catastrophic failure is one that happens in safety critical software. In general, both faults and failures can be referred to by the term defect.

For instance, in the context of an E-commerce website, one of the requirements might precise that a user who logs in on his birthday, should receive a digital gift coupon as a monetary incentive for shopping on the site. A Further requirement is that, this incentive would be offered to a user once and only once in a year, even if this user edits the date of his birthday. However, if the developer forgets to properly check some conditions when developing the incentive giving function, the user might be able to login on his birthday, receive a gift coupon, spend it immediately, then login again on the same day, and receive an additional gift coupon one more time. This mistake made by the developer is an error. The written function in the source code responsible for insufficiently checking the user's eligibility for receiving one and only one gift, has a fault of omission. If the scenario occurs in runtime, where the user repeatedly performs the actions that lead to receiving the monetary incentive multiple times on the same day, then the system's behaviour clearly deviates from the specified requirements, and therefore becomes incorrect, which is by definition a failure.

3.1.2 Testing Process

A process is a series of activities carried out to fulfill a certain purpose. All processes accept input and produce tangible output. A detailed description of a process might include defining the entry and exit criteria, stating the purpose of the process, describing the role of each activity, explaining the metrics enforced, showing the templates used, and examining the verification points [45].

The testing process presented in the International Software Testing Qualifications Board (ISTQB) syllabus could have as input, a test strategy which is basically an outline describing the testing approach, and a project plan formally describing the project excution and control [45]. The input could also include a master test plan which incorporates general test planning and offers documents for test management, and finally some status information on the testing progress. On the other

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

hand, the output of this process might consist of a test plan detailing the workflow, and the test specification in the form of test suites and test cases. The output might have the test environment which includes a specification of the software and hardware setup needed for test excecution, and the actual test data. The output could also include test logs, test summary reports, progress reports, and test experience reports [45].

The activities in the testing process are performed in the following order: Test planning and control, test analysis and design, test implementation and execution, evaluating exit criteria and reporting, and finally test closure activities.

- Test planning and control: Test planning includes tasks such as the definition of the mission and objectives of testing, as well as resource planning of the necessary time and manpower for the test process. Test control refers to checking the test activities and verifying that the project's practical progress in testing conforms to the plan. This task also involves reporting any inconsistencies between the project's actual status and what has been planned, then taking corrective decisions based on those reports. All made plans are to be regularly verified and adjusted, throughout the development of the project. The main activity in planning revolves around determining and documenting the general test strategy. Moreover, due to the impracticable nature of exhaustive testing, the prioritization of the tests becomes necessary, and is determined based on the assessed risks and on the severity effect of possible failures [40].

- Test analysis and design: The goal of this task is to specify the test environment needed, as well as to design high-level test cases and test conditions. A test case is composed of input and output values, preconditions and postconditions for execution, and expected results. It has the purpose of verifying a test condition [46]. This task starts by reviewing and analysing the project's basis documentation, then it is followed by designing logical test cases for both expected and unexpected inputs [40].

- Test implementation and execution: The goal of this task is to transform the logical test cases into concrete physical test cases, and executing them in a verified test environment. This phase aims at examining the completeness and correctness of the features by implementing and running these test cases. Additionally, recording test results should also be taken into account because there's no value to tests with no logs [40]. Moreover, the test cases could be grouped inside test suites to increase test execution efficiency, improve understandability, and simplify reproducibility.

- Evaluating exit criteria and reporting: Testing is continued until the exit criteria are met. Test evaluation requires comparing actual test results with the group of exit criteria defined in the test plan. If the exit criteria has been achieved, then testing is terminated, if not, then either more test cases are executed, or the criteria is deemed too difficult [40]. Reporting test results enables the measurement and tracking of the test progress. This also makes it possible for stakeholders to be presented accurate and useful data. [45].

- Test closure activities: The knowlegde and experiences gained throughout testing are analysed, recorded and archived for guidance in future projects. Despite its usefulness, this task is quite often skipped in practice [40].

3.1.3 Testing Techniques

The standard ISO/IEC/IEEE 29119-4 covers a variety of test design techniques to be relied upon throughout the testing process [47]. The test techniques themselves can be changed depending

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

on the specific project needs. These techniques defined in this standard, are also grouped into different categories which are: Specification-based techniques, structure-based techniques, and experience-based techniques.

Specification-Based Testing

The specification-based test design techniques are also called black-box tests and functional tests. These techniques can help design and produce test cases according to the specified requirements of the product. They do not examine the inner set-up of the software tested. Since testing all the possible input data and different input combinations is too exhausting (or even impossible if the potential input is infinite), it is more appropriate to use techniques that simplify the testing process by choosing the input data in a reasonable manner [40]. Some of these techniques are the following [47]:

- Equivalence Partitioning: This technique divides the input into different equivalence classes. An equivalence class is a data subset under the assumption that all its members are handled the same way by the testing object. Therefore, it's enough to test one representative from that class, if that value is valid, then all other members of the class are valid. If it detects a defect, then the other members would have detected that same problem. It is also important to choose another value from outside the equivalence class, and check if it really is invalid [48]. Before determining the valid and invalid ranges, all test conditions and requirements should be considered.
- Boundary Value Analysis: This technique adds significantly to the test cases produced by equivalence partitioning. Since defects are often detected when checking the borders of an equivalence class, it is reasonable to choose values on or near the boundaries of the said class. If the data has a type that makes choosing the exact boundary not possible, then it is sufficient to choose one valid value from inside the equivalence class, and an invalid value from the outside [48].
- State Transition Testing: Since the behaviour of a system might differ depending on its state or the history of events and inputs that triggered its current state, it is important to design tests covering the system in every possible state, in every valid and invalid transition, as well as in specific significant sequences of states.
- Scenario testing: A scenario, in this context, is a hypothetical story about how the application is used. By extension, a scenario test is a test that relies on hypothetical but realistic system scenarios or use cases. These tests are usually more complex than typical test cases, and their completions require passing through multiple steps [49].

Structure-Based Testing

Structure-based testing is also named white-box testing. It bases itself on the test object's source code, thus, it is necessary for the code and structure of the system to be accessible. These techniques require that all existing code would be executed at least once. However, the expected output of the tests should not be based on the code, but rather on the specified requirements in order to detect defects. The parts of the code, whose execution has been tested is coined as "covered". The following methods assist in achieving high test coverage [47] [40]:

- Statement testing: This technique have the test cases execute, at least once, either every statement in the code, or at least a predefined percentage of all statements. Before testing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

starts, a control flow graph might be created to make the process easier. A downside to this technique would be that even if all statements in the source code are covered, multiple faults might not be detected. This is due to statement testing not needing using test cases where the execution flow skips some lines of source code. For instance, some faults can be found only when an if-block is not entered, but on the other hand, when the if-block is entered and all the statements in that source code are executed, then no fault is detected.

- Branch and decision testing: Each time the code execution flow can go in two different paths, a decision has to be made. Consequently, a branch is the result of completing a decision. From the basis of a control flow graph, every edge in the graph represents a branch, while each node stands for a decision. In other words, this testing method concentrates on the execution of all decisions instead of all the statements themselves. However, by achieving a high decision coverage, a high statement coverage is implied.

- Condition testing: Since branch and decision testing does not check the complexity of partial conditions that can potentially exist inside a decision (and instead only assign the decision a general "true" or "false" value), it becomes important to evaluate those internal conditions that are usually separated by logical operators. One way of doing this would be by assigning different boolean values to every part of a combination of conditions, and then check how they affect the outcome of the decision.

- Data flow testing: Some defects can only be produced by making a series of decisions and having the flow of code execution follow a specific path. In order to detect such defect, data flow testing is used. In other words, every possible path in the code is checked. Every time a variable is defined or used, a new path for the data flow emerges. A very challenging aspect to this technique would be covering the exponentially high number of possible paths.

Experience-Based Testing

Experience-based techniques rely on the personal skills and gathered knowledge of the team in order to uncover defects that would otherwise fall under the radar of other formal testing techniques, such as when a fault is only triggered by repeating the same activity a certain number of times. These techniques complement well the specification-based and structure-based methods, and do not have a strong dependency on the requirements specification. Some techniques of experience-based test design are the following [50]:

- Error guessing: This is a test technique that makes the most out the experience of a skilled tester. In other words, the tester is supposed to be creative, and check for situations that may lead to defect detection. Examples of such test cases typically include checking the reaction to a division by zero, and observing system behavior after inserting a blanc input or one containing some problematic symbol characters.

- Exploratory testing: This technique is only appropriate for experienced testers, because effectively exploring the system requires a set of harnessed skills for predicting defects, as well as a good knowledge of testing techniques. This method involves less test planning and more test execution.

- Checklist-based testing: A checklist can present possible faults that usually go undetected by regular testing. These can take the form of rules of thumb or can be gathered from previous experiences in other projects. They can form the basis for a quick start in experience-based testing, and assist the lesser-experienced testers in keeping track of their test progress and picking up their pace.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1.4 Testing Methods

Software can be inspected using different testing methods, such as white-box testing, which is also called structure-based testing, and has already been described in section 3.1.3. Another method is black-box testing, which is also called specification-based testing, and was discussed in section 3.1.3. These two methods has their own strengths and weaknesses when compared with each other, and these points need to be examined. Consequently, it's also possible to somehow combine these two methods in order to attempt gray-box testing, whose details also need to be discussed.

White-Box Versus Black-Box Testing

In order to shed additional insight on white-box and black-box testing, their advantages and disadvantages must be detailed. Among the strengths of the former are the following [51]:

- Efficiency in exhaustive defect identification: By having access to the code, it becomes easier, for instance, to choose the input data and expected output that can challenge the correctness and completeness of the system. White-box tests can identify hidden defects that would not be exposed without investigating the structure of the code.
- Code optimization and improvement of the overall code quality: By detecting defects and fixing made mistakes, the code is constantly updated and improved. As developers are more concious of the importance of good code quality and early testing, more best practices are endorsed in development to fix, prevent or limit defects. Moreover, white-box tests can also help unravel dead code (which is the code executed but never used).
- Usefulness for achieving maximum test coverage: The testing effort should be measurable, the progress should be monitored, and when needed, presented in an understandable manner to stakeholders. Having access to the source code is the only way for efficiently testing and tracking the execution of all its parts.

White-box testing also has its limitations. Among the disadvantages of white-box testing are the following points [51]:

- Inability to detect missing functionalities: If some requirements were not implemented, this type of tests will not reveal the problem. White-box tests focus entirely on the structure of the written code. Defects originating from absence of a feature, or missing some part of a functionality, are not identified.
- Lack of focus on issues related to the runtime environment: Dependencies on external libraries, third party tools, or even different running operating system, may lead a product's functionality to partially or completely deviate from its intended purpose. Vulnerabilities originating from the running environment might also be exploited to misuse the product. However, white-box tests do not detect such problems.
- Expensive costs in time and human resources: Performing good white-box tests needs a great deal of time and effort. The complexity brought by these tests obligates the tester to be familiar with the system being tested, and to have great skills. This in itself translates into higher cost.

The second method to examine is black-box testing, it holds different strong points from its counterpart and these are the following [51]:

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

- Efficiency for rapid testing of programs with a substantial amount of source code: Testing becomes more expensive, the bigger the program gets. Therefore, focusing on the functionalities of such a system becomes easier, especially since a program having more lines of code, does not mean that it offers more or better functionalities.
- Simplifying the process of understanding legacy code [52]: As legacy code is often outdated or requires extensions, simply reading the code (or its documentation) does not necessarily provide developers with the needed understanding of its functionalities. However, reading or performing black-box tests, provides a faster and simpler way of learning about a system's behaviour, how to use it, and also how not to use it.
- Usefulness when testing from the user's point of view: A separation between the perspectives of the user and developer is required, in order to show that a system truly meets customer requirements.

After listing the advantages of black-box testing, it's imperative to look at its weaknesses and limitations. Some of these points are discussed below [51]:

- Inability to detect unneeded extra functionalities: Providing functionalities beyond the product's specifications does not necessarely signify exceeding customer expectations. On the contrary, extra features that are not demanded by the requirements, are not covered by black-box tests. These untested supplementary features often lead to defects that undermine the security of the whole application [40].
- Strong reliance on concise requirements specification: If system specification is not clear, it becomes very challenging to design effective test cases. It's also hard to decide on the appropriate input values without clearly understanding the functional specifications.
- Partial coverage: Since black-box tests focus entirely on the functionalities of the system without examining its internal structure, test cases do not cover all data execution paths. In addition to having those paths untested, some other paths might be repeatedly checked by different test cases.

Based on the presented strengths and weaknesses of each method, it can be summarized that white-box testing examines the system in a deeper level of detail than black-box testing, but the first is generally more expensive and time consuming than the latter in both the time needed and the required skill set.

Gray-Box Testing

Gray-box testing is a
method where the tester has partial knowledge about the inner workings of the system [53]. Compared to white-box and black-box testing where the inner workings of the program are either fully known or unknown at all, gray-box testers have access to more information than needed for black-box tests, and less data than provided for the white-box tests. However, in all cases, the amount of details known by testers go beyond the list of specified requirements. Therefore, gray-box tests can be generated from data such as an architecture diagram [51], or the list of protocols used. Among the advantages of gray-box testing are the following [53]:

- Non intrusiveness: Since gray-box tests can be based on interface definition, there's no need for the tester to intrude on the source code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Objectivity in testing: Because of their knowledge about the system, it's challenging for designers and developers not to be biased when they create and perform the tests themselves. Giving the tester limited knowledge about the system, ensures that the resulting test suites are as unbiased as possible.
- Smart test design: Due to their partial knowledge of the system, gray-box testers focus their efforts on checking the test scenarios that are most likely to uncover problems, such as, for instance, testing how different data types are handled by a function.

On the other hand, the gray-box testing method has its own inconveniences as well. These might include the following points shortly described below [53]:

- Partial code coverage: Due to the limited knowledge about the inner program structure, many code execution paths would not be run by the tests. This means that the achieved coverage depends on the amount of additional information provided to the tester.
- Difficulty in defect identification: When relying on gray-box tests, defects (which are explained in section 3.1.1) might become hard to uncover especially in a distributed environment, because the defect identification might depend on the system's capacity to throw helpful exceptions, as well as the manner they propagate across the system. This means that the knowledge provided to the tester might be insufficient, if the source code has a poor quality, or if the testing environment is unstable.

In conclusion, gray-box testing is some sort of middle ground between white-box and black-box testing, especially useful for integration testing (described in section 3.1.6). Choosing the appropriate method depends on factors such as the time, cost, goals, and access degree to the internal program structure [53].

Static Versus Dynamic Testing

Static testing revolves around testing software manually or with help of tools, but without executing any part of the actual source code. The main methods by which this is performed are reviewing the code, or doing a static program analysis.

A code review refers to the manual examination of the source code in order to uncover errors. There are many types of reviews such as inspections, peer-reviews, and walkthroughs. Every type have a different level of formality, but all follow a similar procedure, consisting of the following main operations: First a plan is devised, then, a team reads or visually inspects the source code of a program separately. Finally, they meet and share the errors they discovered.

On the other hand, static program analysis relies on tools to examine the source code without running it, it can find defects such as uninitialized or unused variables, inconsistencies between the interfaces of a module and a component using it, syntax violations, or code that is never executed [54].

Dynamic testing is based on evaluating software by running the test object, and observing its behavior during execution. White-box testing, black-box testing, gray-box testing and experience-based testing are all considered dynamic testing techniques. Even the previously described activities of the ISTQB testing process, are meant for effectively performing dynamic tests. It's especially important to plan these tests considerably well.

To summarize, static testing provides general information about a program's logical aspect and coding style, while dynamic testing can validate system functionalities and also uncover defects

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

originating from the code's logic and structure. Both types aim to find as much defects as possible, but only dynamic tests can truly validate software quality. Static tests, however, can add to the verification efforts. Additionally, tools can be used to support both types of testing: Static analysis tools provide similar functions to optimization algorithms used by compilers, while tools for dynamic testing can uncover defects such as memory leaks or issues with runtime dependencies [55]. Running the source code together with some tools to check its quality is called dynamic analysis.

3.1.5 Testing Types

There are different testing types and subtypes, all of which can generally be divided into functional or non-functional tests. Functional testing is a type of black-box testing. It starts by identifying expected software features, then input and output data is chosen based on the feature's documented specifications. Then, following the execution of each designed test case, the actual test result is compared with the specified expected output. The emphasis of these tests is on verifying that those comparison conditions are always true. In simpler terms, their purpose is to check if the system does what is expected from it to do.

Non-functional testing focuses on checking the implementation of non-functional requirements. These non-functional requirements are also referred to as quality attributes. On one hand, there are execution qualities observable at runtime such as usability, security, or even privacy. On the other hand, there are evolution qualities that can be observed only in the inner code structure such as scalability, extensibility, or maintainability [56]. Generally speaking, non-functional testing aims to check on how well a system does its functionalities.

In practice, due to time and budget constraints (or developer's inexperience), functional tests are generally prioritized over non-functional ones. Moreover, testing non-functional requirements often needs the presence of testers with a highly specialized skill set, and testing is sometimes not even possible due to technical difficulties. The specifications for these qualities are also not always clearly stated during the planning phase, which may lead to ambiguity and confusion when trying to test them [57].

Regression Testing

Regression testing is a group of functional tests which refers to the process of re-testing software after committing some changes to it. Even small code modifications might potentially cause the system's behaviour to deviate from its intended purpose. Risking adding new defects does not stop at the changed parts, the effects of these modification or extension in one section of code, might cause a different distant section to malfunction. It's therefore important to cover the entirety of the system in regression tests. The depth and intensity of those tests should depend on the current stage of development and on the risk level of the new functionaries.

When working on big projects, if every possible test case is included in regression tests, then these tests might become unmanageably large. This in turn, may lead to long test execution time and exhaustion of computational resources. On the other hand, small regression tests can miss some of the newly introduced defects, especially if the system's implemented functionalities and its endorsed software qualities are not covered well enough. Therefore, it's important to choose well the minimum set of test cases that efficiently covers the system [58].

TU Wien Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Continuous Testing

Continuous testing aims to deliver fast and continuous feedback on tests associated with system requirements. It is a form of functional testing, and is achieved by incorporating testing as part of the software build process. If testing is planned early and is executed continuously, then newly introduced defects get discovered faster, and are prevented from migrating to the next software build or release. According to studies by David Saff and Michael Ernst [59], this testing practice helps in decreasing the overall development time by about 8% to 15%, and may significantly reduce regression related wasted time by about 92% to 98% [59]. Additionally, continuous testing enables frequent measurement of some software qualities as well as tracking the state of the system. Finally, after examining the tracked information, it becomes possible to optimize the process and continuously improve it.

Performance Testing

Performance testing is a subgroup of non-functional testing that assesses software qualities such as speed, stability, response time, and scalability. It usually relies on observing and measuring system behavior under an increased load. In other words, the software has to maintain its correct behavior when performing tasks within the planned specifications of the acceptable response and execution times. For instance, these tests would check how fast the system performs a task under some specified conditions, or how long the internal processing time is, for finishing some specified tasks under some fixed circumstances. The main subtypes of performance testing are the following [45]

- Load testing: These tests check how well the system performs its functionalities under the average expected workload, as well as under the maximum anticipated load. The aim of such tests is to identify information, such as, when possible reductions in the quality of a functionality occur, and the degree of that quality drop.
- Stress testing: It tests the system by putting it under an extreme workload, in order to observe how it reacts to, for instance, high user traffic or increased data processing. The purpose of these tests is to determine the circumstances that would crash the system (or cause severe failures).
- Scalability testing: The objective of these tests is to assess the software's ability to meet future requirements efficiently. In other words, it observes how well the system reacts to growth, in order to pre-emptively make decisions and changes before acquiring losses in functionality.

This type of testing can be very costly in both time and effort. However, tools for performance testing can collect useful information and produce helpful reports, presented generally in text and graphical forms.

Security Testing

Security testing controls if a software application protects its data from being accessed or modified by unauthorized parties, all the while maintaining the system's intended functionalities for legitimate users. Consequently, this type of testing is non-functional. Furthermore, security testing holds the purpose of verifying that the main principles listed below, are sufficiently followed [57]:

- Confidentiality: The system should protect its data from being disclosed to third parties. This also includes protecting the privacy of the users (to an agreed upon extent).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Integrity: The information stored on the system should be modified or deleted only by the authorized users. Moreover, no new additional data should be created by an unauthorized party.
- Authentication: The system should, for example, be able to confirm the identity of a user, or the legitimacy of another program accessing one of its functionalities.
- Authorization: The software system should allow functions to be accessed or performed only by their intended users. This can be achieved by enforcing some form of access control.
- Availability: Authorized parties should always have access to their intended data. In other words, the system should ensure the availability of needed data to all its legitimate users.
- Non-repudiation: The system should always guaranty that the sender and receiver of a transferred message cannot deny their role in this data transfer process.

Other Testing Types

A large variety of testing types exist, some are similar to the types previously presented, and some are only appropriate in a specific context. For instance, installation testing ensures the correct installation of the system, and that it works well in its intended environment [48]. Then there's conformance testing which verifies that the system is following a specific set of standards [48]. Another example is accessibility testing which tests the system from the perspective of users with disabilities, and verifies how well the system conforms with accessibility standards [48]. Finally, this chapter also focuses in details on two other testing types which are GUI testing (discussed in section 3.2) and usability evaluation (discussed in section 3.3).

3.1.6 Testing Levels

A test level is a group of testing activities that can be organized and managed collectively [46]. Generally, each level is associated with a distinct software development responsibility, which leads to a categorization where the scope of the tests increases with the level, resulting in a classification that consists of the following: Unit testing, component testing, integration testing, system testing, and finally acceptance testing.

Unit Testing

The purpose of unit testing is isolating the smallest testable part of the software from the rest of the source code. Consequently, the behaviour of that isolated code is examined, and its output is compared with the expected result. A program unit can be an interface, a class, or a set of classes when working under object-oriented programming languages (or sometimes it might be a function or a procedure in the case of other programming paradigms) [60]. A single software unit can be associated with multiple unit tests that check its behaviour and states.

A good set of unit tests have many advantages: It assists the most with help in finding defects, and preventing their propagation. Detecting a problem early on, saves up the time and effort that would have been wasted otherwise. It's also important to note how well these tests complement code refactoring efforts. In other words, if the series of unit tests are weak or absent, then restructuring the code would most likely introduce more problems than improvements. Another advantage would be the guiding role they can play in assisting a reader who wants to learn how to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

properly work with the code (with examples of valid and invalid input). Unit tests are often more informative, and more up-to-date than the documentation.

It's important to keep unit tests short and concise, so that running them would not take long, and can be done frequently. It's strongly advised to separate from unit tests any code that directly accesses the database, communicates over a network, or that edits system configuration files.

Component Testing

This is also called module testing. Components (or modules) are the different independently developed parts of the software. These components are generally linked to each other, and can be comprised of a one unit or a combination of several smaller components [40]. Component testing is similar to unit testing but occurs on the level of components. This method of testing focuses on the effectiveness of each component of the application separately. This leads to the identification of defects within a module itself, rather than defects that lurk in the interactions between modules. Generally, it's preferred to perform these tests before starting with integration testing [40].

Integration Testing

The purpose of integration testing level is to test all the different modules and their interactions. Integration testing can be considered a logical extension to unit testing or component testing. Given the basis that the individual code units have already been tested, the integrated combinations of units from separate modules are tested next [44]. If an integration test fails while the unit tests pass, then it indicates that an issue likely exist in the interaction between said individual units, or in the actual data transfer. All modules that form a process should eventually be tested together. Many methods of integration testing exist, some of them are briefly described below [44]:

- Big bang testing: In the big bang approach, most of the modules are grouped together, and tested all at once. This method considerably shortens the time spent on integration testing but it becomes difficult for the team to trace the cause of a bug. The big bang test can only be performed after all modules have been developed, and once it runs, it usually requires long execution time.
- Bottom-up integration testing: Since modules can instantiate other modules, their interactions produce a module hierarchy having a tree structure. Bottom-up integration testing is an incremental approach where the first to be tested are the lowest level modules of the hierarchy, then these tests are used to simplify the testing process of higher level modules. This way of integration continues until reaching the module situated at the summit of the hierarchy.
- Top-down integration testing: This is also an incremental approach where testing begins with the highest level integrated module, then the branches of that modules are tested incrementally until the last one of those related modules is reached.
- Sandwich integration testing: This is a combination of the bottom-up and top-down approaches. Modules are simply tested as they become available.

If an integration test has to verify the interaction between two modules, but one of them have not been implemented yet, integration tests might simulate the behavior of the missing component. This is particularly useful for incremental approaches. Choosing an appropriate integration technique depends on the system architecture, and the risks associated with the modules [44].

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

System Testing

Within this testing level, both functional and non-functional requirements are investigated. The purpose is to check how well the product is meeting the requirements, and how well system components communicate with each others and with the system in general. It is usually considered a black-box testing technique, and is only applied on a complete and fully integrated software system.

In addition to functional tests, many supplementary types of testing can also be performed. Examples would be regression testing, performane testing, and security testing (all described in section 3.1.5). Choosing which types of system testing to adopt, depends on the available time, the budget, the risk factor, and the skill set of the testers [44].

Acceptance Testing

The acceptance testing level formally evaluates the software system from the customer's perspective based on the user requirements. Afterwards, a decision is made regarding whether the software satisfies the acceptance criteria [46]. It is performed after system testing and before delivery.

Unlike other types of testing, acceptance testing does not aim to identify defects, but is rather expected to demonstrate that the software solution is working correctly. The presence of the customer or end-user is necessary for doing this test (sometimes both are required). They can evaluate the fulfillment of the specified requirements themselves, or witness someone else perform the acceptance test. The test object constitutes the whole system. Also, the testing techniques relied upon are usually experience-based [45].

3.1.7 Test Automation

The goal of test automation is to decrease the number of manually performed test cases, and not to replace manual testing entirely. Therefore, one cannot fully understand test automation without first defining manual testing. As the name implies, manual testing is done by a human who manually follows the test steps that constitute a test case, thus, the tester simulates some detailed actions of the end-user [58]. On the other hand automated testing relies on an automation software tool to plan, generate, or execute test case suites, then compares the actual outcome with the expected result. The automation tool used is a separate software from the object under test, but the properties of the developed software can greatly influence the selection of the tool [45].

Automating tests brings in numerous advantages, such as the fast speed at which the tests are executed, which far exceeds that of a manual effort. Furthermore, automated tests are easily repeatable, and can be reused on different software versions [58]. Moreover, testers can design and create test suites in a comprehensive manner, that makes it easier for newer team members to understand both expected and unwanted system behaviour. Thus, it indirectly improves the productivity in the long run. This is particularly obvious when the documentation is absent or not up-to-date [61]. Section 3.2.3 discusses these advantages in more details, with a particular focus on automated GUI testing, which is, as the name implies, a subset of automated testing. Additional automation benefits are also presented in section 3.3.4 from a usability evaluator perspective.

On the other hand, test automation also has its own disadvantages. These include, for instance, a high initial development cost per feature. In other words, more development time is needed for creating the initial tests. Additionally, test automation requires team members with a larger skill set, as well as more tooling needs (such as testing frameworks and test runners). Furthermore, when an automated test fails, it might prove difficult to comprehend whether the test failed because

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of a bug in the source code, a bug in the test code, or due to a change in the requirements [45]. In section 3.2.5, the limitations of test automation are more expended upon, with a focus on the GUI testing subgroup.

Test Automation Key Factors

The extent of the success of test automation depends on some key factors that need to be considered among which are the following [58]:

- The number of regression cycles: If the number of product builds or releases is high, then the number of times that regression tests need to be executed is also high. Therefore, relying on test automation helps in saving on cost in time and effort. Additionally, this is necessary for effectively adopting continuous testing.
- The regression test suite's size: The smaller the test suite's size, the easier it becomes for the testers to manage it, but in order to reach a higher coverage, testers keep expending their test suites. Furthermore, getting closer to the maximum coverage requires an exhaustive and extensive testing effort, which might end up being counter-productive in some cases. Besides, bigger-sized test suites, does not necessarily reflect a better test coverage. Therefore, the better approach is to aim for an optimal coverage using minimal-sized test suites.

  Automated tests can run without human intervention. If the regression test suite is large in size, it can be executed overnight. This frees the computational resources for development during the day and significantly saves up on time.
- Automation Tool capabilities: The chosen tool has to be compatible with the software tested, as well as being appropriate for the tasks to be performed. Some tools are more appropriate for functional tests while others are better for regression tests. The selection of the tool should not be influenced by its popularity but should rather be based on the automation requirements.
- Automation cost: Automating tests needs time and a specialised skill set. In the long run, it reduces the testing cost, the time spent on testing, and improves the software quality. However, the initial cost investment is higher than manual testing. For instance, purchasing licences for an automation tool and training the employees to use it, can cost a company a considerable sum without producing a single automated test. Besides, even after the tests have been fully automated, a maintenance cost still applies.

Test Automation Steps

All stages in software development can benefit from testing and by extension from test automation. Tools can assist in early identification of conflicting requirements during the requirement analysis phase, they can help in validating the system's architecture in the design phase, and they can also do a static or dynamic analysis during the implementation, as well as perform functional tests and performance tests during the testing phase. However, no matter the development stage, introducing test automation should pass through the following steps [45]:

- Test tool selection: It starts by analysing the specified requirements and identifying aspects where tools can improve the testing process, then clear selection criteria is defined. Afterwards, tools fitting these criteria are searched for, investigated, and evaluated. Choosing a tool not only depends on its supported technologies and environments, but also on the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

supported types of testing. Other objective criteria that might influence the selection are the tool's scripting language, its training cost, as well as the organisation's readiness for change. It's also possible to conduct a proof of concept to prove that a tool meets the specified selection requirements.

- Defining the scope of automation: The scope of the automation refers to the part of the system that will be automated. Determining the scope is based on the functionalities with the highest priority, the scenarios that consume the most time and effort, task repeatability, and technical feasibility.
- Test planning and design: In this step, an appropriate test automation strategy is defined, containing the test types to be performed, on which test level, and which tests are automated by the tool [62].
- Test execution: During this step, the automation scripts of the tests are executed. These scripts usually require some test data as input, and produce detailed test reports as output.
- Test maintenance: Since the software under test would continuously acquire new functionalities until its final release, new automation scripts are frequently added, and the existing ones are maintained to improve the quality of testing, and especially the effectiveness of regression tests.

Test Measurement and Evaluation

Most software testing metrics concentrate on evaluating either the coverage, the progress, or the quality. First of all, automated test coverage is a percentage referring to the extent of source code that gets executed by running the automated test suites. Many types of coverage exist such as statement coverage and function coverage (see section 3.1.3). This percentage evaluates the completeness of the test coverage and can measure the extent of automation relative to the total coverage of tests (including the manual ones). The following equation represents this metric [63]:

$$\text{Automated Test Coverage: } \frac{AC}{TC} \times 100$$

where:

AC = Automation coverage

TC = Total test coverage (comprising the sum of manual and automated coverages)

An additional metric is the automation index, which indicates the percentage of test cases that are suitable for automation among all test cases. The decision of whether to automate a test or not is based on how much time and effort the automation would save compared to the alternative. The automation index is measured using the following equation [63]:

$$\text{Automation Index: } \frac{AT}{TT} \times 100$$

where:

AT = Number of automatable test cases

TT = Total number of test cases

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Subsequently, an additional metric is the automation progress, which is a percentage referring to the number of tests that have been automated compared to the overall number of automatable test cases. The following equation represents this relation [63]:

$$\text{Automated Progress: } \frac{AA}{AT} \times 100$$

where:

AA = Current number of automated test cases

AT = Number of automatable test cases

Furthermore, the quality of the tests is an important aspect, and in particular, their ability to uncover defects. Defect removal efficiency is a metric used to evaluates tests regardless of their mode of execution. However detecting variations in the defect removal efficiency under various automation efforts gives a company a good perspective on the quality of its automated tests and the effect of automation on defect detection. Defect removal efficiency follows the equation stated below [63]:

$$\text{Defect Removal Efficiency: } \frac{DT}{DT + DD} \times 100$$

where:

DT = Number of defects identified during testing

DD = Number of defects found after delivery (which is different from the total number of existing defects, and rather refers only to the quantity of defects found)

All these presented metrics help in tracking and evaluating the automation efforts for a project. However, measuring the added value of automated testing does not correspond to the percentage of automated tests compared to the manual ones or similar percentages, but is rather based on the return on investment (which is the ratio of profit divided by expenses). In order to measure the savings in time and resources, the cost to execute each test case manually needs to be captured first, then mapped to the cost of automation of that same test case. Consequently, the execution frequency of these tests is considered to compare the cost of manual and automated testing, and determine the extent of savings in time and effort (or potential loss). Generally, the unit of measurement for this is in man-hours.

To summarize, in order for test automation to be successful, the automation goals need to be clearly defined at the start and tracked across all phases of the software life cycle. Evaluation metrics such as the automation index, the automation progress, and the automated testing coverage, as well as the automation's return on investment can all help immensely in assessing the current testing status and how to adjust accordingly.

3.2 Graphical User Interface Testing

As discussed in section 2.1, a GUI encompasses the interaction between the system and the user. Therefore, even if the non-graphical part of the software is developed in a way that supports all its functional requirements, it's still possible for a mistake in the GUI to obstruct or partially hinder the users from performing their tasks. Consequently, testing the GUI is needed to confirm that the software truly meets the specified requirements [40]. Therefore, it's needed to comprehend GUI testing, its benefits, its limitations, and its methods.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.2.1 Definition of GUI Testing

GUI testing refers to the practice of testing a software's GUI to ensure it meets the specified requirements. Usually, this implies performing a set of test cases that entirely covers the functionalities offered by the system and fully exercises the GUI in order to validate the properties of major GUI elements [27].

GUI testing can be performed manually or be automated with the help of a tool. An automated test can give a predefined input to a GUI object and compare the result with the expected outcome. Automation can also simulate GUI events such as a click or pressing a button. The set of values on the properties of each GUI element, during execution, constitute the GUI state.

3.2.2 Aim of GUI Testing

One of the goals of GUI testing is to uncover defects. A defect manifests itself, sometimes, only by triggering certain GUI events in a specified order. Such problematic GUI event sequences are generally difficult to locate, due to the high number of elements and events in a GUI, which result in an exponentially high number of possible combinations [64].

In addition to finding defects, GUI testing focuses on performing functional tests from the user's perspective. Every requirement would have test cases in charge of checking if the requirement is sufficiently met. Therefore, this process is critically important at the system testing level (but not exclusively for that level) [65].

Furthermore, higher quality GUI tests might check for the following [65]:

- After the GUI window is resized, the different GUI elements should not overlap or become unrecognisable. Besides, their new positions should be acceptable. This is especially important for the length and width of text input fields.
- Text fields should have some form of validation. For instance, if only numbers are accepted in a field, then the system should prevent the user from submitting text. Furthermore, date input fields should only accept data presented in the correct date format.
- Error messages should be displayed at the correct moment and should have text relevant to the error that occurred.
- The distance between GUI elements should be consistent, and should be properly aligned vertically and horizontally. Moreover, images used should have good quality and also be properly aligned.
- The position and size of GUI elements should be responsive to different screen sizes (and screen resolutions).
- The overall design of the GUI and its colors should be aesthetically pleasant. The font used should be readable, and the text should be correct and concise.

3.2.3 Benefits of Automating GUI Testing

As previously stated, the process of GUI testing can be performed manually or be automated. However, even though manual GUI testing should not be completely replaced, automation still offers many benefits, which are discussed in the following [45] [58]:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Saving up on execution time: Test automation tools can execute test cases in significantly less time than manually performing them. It should be noted that preparing the test scripts takes up extra time, but once they have been written and are ready for execution, automated GUI tests run faster than their manual counterparts. Also, maintaining these tests might become highly time consuming to the extent of being counter-productive, it's therefore important to choose automating sufficiently stable tasks, and wisely design the test cases, to avoid high maintenance cost.
- Increasing the test coverage: An automation test tool can increase the depth and scope of the performed tests in a way that would be considered challenging manually. An automated test suite can help in checking the functionalities of a software product, or also examine the inner program structure and behaviour, such as the effect of the performed task on the system's memory, or if some internal object states are always holding the expected values as the test is executing. Besides, automation can assist in verifying the effect of combining and running different sequences of GUI events, as well as generating a variety of test data, and analysing system responses to such data. All this leads to an increase in test coverage.
- Decreasing the number of errors: When performing a test manually, the human actor is prone to errors. This is particularly true when the task is complicated and has to be repeated many times. Automated GUI tests can perform the same steps correctly every time without forgetting to log and report the results in details.
- Simplifying tasks that are challenging if done manually: Some tasks such as manually testing different localised versions of a GUI can be extremely difficult for a tester. However, if these GUIs only differ in the language used and not in their graphical elements, then the same automated GUI test case that checked one GUI, can easily check the other localised versions. Besides, even GUI performance testing of applications requiring complex interactive sessions that rely on timer-driven activities, can be performed correctly by an automation tool [66]. This saves up the hassle of repeatedly performing complex GUI performance tests manually.
- Running tests unattended: Since the execution of automated tests does not require human intervention, an automation tool can run the tests overnight when the computational power is not needed. These tests can also be run on multiple computers to make the process faster.
- Being repeatable: After every change in the source code, some defect might lurk into the system and the old tests need to be repeated. GUI test cases should be present in regression testing, to ensure high software quality after each release. Once these test cases have been automated, they can run continuously without additional cost in time and effort (except for test maintenance). On the other hand, the cost of manual tests increases the more these tests are repeated. This is especially true when in need of testing the same scenario in multiple environments, as automated GUI tests would usually only require making changes in the configuration file, then executing the same test suite. In comparison, repeating the same manual test in various environment is much more exhausting.
- Test case reusability: Following some testing best practices and respecting some coding principles can make a test suite highly reusable which saves up significantly in testing time and cost. Some of the factors improving test reusability are understandability, changeability and independence [67]. First, understandability refers to how easy to read and comprehend a test case is. On the other hand, changeability refers to the extent to which a test can be changed. It highly depends on the representation and structure of data. For instance, using variables instead of constants in a test case increases its changeability. Finally, independence measures the extent to which the execution of a test case depends on other ones. For

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

example, if a test case can only be run after another test case finishes, then its independence aspect is poor, and reusing the test case is difficult.

- Long term decrease in testing cost: Acquiring new tools, training testers, and scripting test cases can cost a considerable sum that makes manual GUI testing cheaper in the short term. However, as development progresses, the cost for manual GUI testing does not decrease. In comparison, once implemented, automated GUI tests can run repeatedly without an additional programming effort. This saves up on the cost in human effort. Additionally, by saving time, the cost also decreases, and by increasing the test coverage, more defects are found, which leads to the conservation of the extra cost that would have been wasted on debugging, or even potential losses caused by a decreased trust in the delivered products (since some software failures might ruin a company's reputation).

- Freeing tester and developers for other tasks: As more tasks are automated, testers and developers have more time to focus on other tasks instead of repeating the same monotonous activities. It's better to free the human resources and direct them toward solving more challenging problems or improving their skill set.

To summarise, the general rules that determine if a testing task is more suitable for automation also apply when deciding whether to automate or not a GUI test. In other words, a GUI test case is better automated, if there's a need to execute it repeatedly, as well as if running the test case manually is more time consuming or more difficult (which results in the manual tester being more likely to make mistakes). Moreover, GUI test automation is also preferred if the activity being tested is not subject to frequent changes, and is considered business critical or is associated with high risks.

3.2.4 Methods of Automating Testing GUIs

There are various ways of automating GUI test cases, such as the capture and replay method, the model-based method, and the script-based method.

Capture and Replay Method

The capture and replay method is performed with a test automation tool and it has of two stages. The first is called the capture process. In this phase, the tester executes the GUI test case manually all the while recording all his steps with help of the tool. Next, during the replay process, the computer executes the pre-recorded steps on the application under test without any direct human intervention. All user actions can be recorded and played back including mouse movements and key presses [65].

To achieve this, the automation tool registers itself as an event listener during the recording phase, and adds to its event notification method all the details of the occurring events. Afterwards, during the playback phase, the tool registers itself as an event source, and replays all the events that have been recorded in the exact same manner and with the same timing.

This method supports automated GUI regression testing. The execution of the first phase is the only costly part, as it requires manually performing the test case. Afterwards, the second phase can be repeated endlessly without any additional cost. Even though this method is easy to perform and does not demand having some specialised skills like programming, it still holds a major disadvantage. Any rearrangement in the positions of the GUI elements can break the tests, and would require recording the test cases anew.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Model-based Method

The second automation method is model-based GUI testing. This helps in examining and predicting the behavior of the application. Since a model can provide a graphical representation of a system's behavior, it can also assist in the generation of efficient test cases based on the specified requirements. In a first step, using a tool with support for model-based testing, the model is built [67]. This model can be written as a textual program or be represented visually (depending on the tool used). Then, based on the specifications, a test suite is generated from the model. Following this step, the inputs and expected outputs are determined for this model, and after running the tests, the actual result is compared with the expected outcome. If the test fails, then the behavior of the tested application does not match the model's expectations, which is generally caused by an erroneous implementation. However, it is also possible that the error occurred while building the model, or that the gathered requirements have an inconsistency.

There are various ways of performing a model based GUI test, however, all of them require a significant initial effort in building the model, before gaining any advantage over alternative methods. Generally, such benefits include the following points [68]:

- Specifying the rules derived from the requirements is done only once.
- The cost for test maintenance decreases because the test cases are easier to generate and re-generate without needing to write new tests for every new feature added.
- Higher test coverage is achieved due to continuous efforts in defect identification.
- Generating tests for the application would benefit from fluid design, since adding a new feature means adding a new action to the model, and by running this action, it would be automatically combined with other existing running actions. Therefore, by generating test cases, the new feature gets covered automatically.

Combining the capture and replay method with the model based approach is possible and when done correctly can result in significantly reducing test maintenance efforts [69]. This effect is linked to the fact that changing one single GUI element leads to recapturing one single test step instead of recording the complete test case anew.

Script-based Method

The last method is the script-based test automation approach, it generally involves writing programs to be executed by the automation tool. The scripted tests would run in a similar manner to unit tests, except that the values checked belong to the properties of GUI elements [64]. This approach yields test cases that can run repeatedly and efficiently as regression tests. Additionally, if these tests have a high abstraction level and both the tester and the developer collaborate and communicate properly, then, the written tests would be more robust and more accepting of change.

These tests usually involve running a sequence of GUI events, tracking the main properties of key GUI components, and observing the result, thus, they do not usually get affected by changes in the position or other trivial properties of a GUI component. Triggering an ordered sequence of GUI events in a test case generally aims at mimicking a certain usage scenario. This technique is called GUI event sequencing [64].

The script-based approach is also called event capture. The main idea behind this method is to concentrate on the underlying aspects of the GUI rather than its external appearance. If the software's architecture used supports separation of concern, then this approach becomes easier to

TU Bibliothek, Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar. The approved original version of this thesis is available in print at TU Wien Bibliothek.

follow because the GUI would be isolated from the business logic, and would therefore be simpler to read and easier to test [64].

3.2.5 Limitations of GUI Test Automation

As previously stated, GUI test automation should not aim to replace manual testing entirely. A test case is not suitable for automation, if the functional requirements of the application frequently change. This is still true even if the modification in a requirement occurs only partially. What's more important is the frequency of the change. Even a small rearrangement in a GUI might cause some previously written GUI test cases to cease to function (depending on the method of automation) [70].

In other words, frequent changes in requirements cause frequent source code updates. The automated test suite must be reliable in producing accurate test results. However, due to the nature and properties of GUIs, a change in their source code might result in the need for adjustments in the previously scripted test suite, or redoing the tests anew [70]. Neglecting these adjustments might result in false positives or false negatives, which endanger the correctness and reliability of the test suite. In some cases, failing to adjust a scripted GUI test case would cause it to become non-executable, such as when a GUI object needed in the test case has been replaced in the source code. Instead of leaving the code with possible vulnerabilities or an unusable test suite, it's better to continuously adapt the test cases. Consequently, the test maintenance cost would increase the more frequent the changes in GUI occur. In this particular case, it's more beneficial to test the graphical interface with the capture and replay method (or manually), and only consider script-based GUI automation when the requirements reach an acceptable level of stability [45].

Even after the release of a certain software version, performing regression tests on the GUI of the next version can be very challenging. When a test case fails to execute, it's sometimes difficult to identify whether an issue has been encountered, or whether it's a matter of a new improvement to the application. This problem manifests itself further if the changes in GUI are not documented. Then, the tester finds himself compelled to test the GUI manually to comprehend the situation. Afterwards, he either adjusts the automated tests or reports a defect [70].

Moreover, even if a GUI test case is completely suitable for automation, it is considered a best practice to execute the test case manually at least once before automating. This is done in order to first be sure that the task can be successfully performed. Otherwise, if a defect already exist, the automated test case that would have been written would never come to pass, and it won't be possible to know if the test case would also still fail even if the defect has been handled. In other words, it won't be possible to identify an erroneous GUI test case without performing the task manually at least once [58].

Some problems occur only under special circumstances. Experienced testers who are proficient in error guessing have quick ways to check the robustness of a GUI. Depending on the application's nature or its context, the most suited testing methods vary. Trying to automate all the different GUI test cases based on error guessing is time consuming and unnecessary. It is sufficient to manually perform these tests once by a qualified tester, since repeating the execution of such tests on regular basis is unlikely to identify more defects [70].

Furthermore, if the repeated execution of GUI test cases is not required or not possible due to time constraints, then manual GUI testing becomes more cost effective than test automation. As previously stated, automation requires a high initial investment to acquire tools and prepare the testing infrastructure before even writing the first test case. This goes without mentioning that producing a script for a GUI test case requires more time than manually performing it. To summarize, if

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the repeated execution of these tests is not needed, then test automation for these tasks is also not needed [58].

Finally, it's important to note that the reusability of a written test suite relates directly to how profitable the investment in automation can become. In other words, the more reusable a test suite is, the less the cost of testing gets. However, the properties of the tested application might influence the applicability of test automation. Organizations that develop generic and independent software solutions have better chances in reusing their automated tests than their counterparts that focuses on producing customized software [70]. This means that the more specialised and customised a software becomes, the more dependent it gets on various third party interfaces, and by consequence, it results in less reusable automated test cases.

3.3 Usability Evaluation

Usability is an important non-functional software requirement (as previously explained in section 2.2.1), and just like most non-functional requirements, evaluating the usability of an application is performed in a special manner and requires special skills and knowledge, which makes the process of usability evaluation very different from functional tests.

3.3.1 Defining Usability Evaluation

Usability evaluation serves the purpose of usability assessment of a system's GUI, through the identification of usability issues and through the measurement of the usability quality (by relying, for instance, on the metrics discussed in section 2.2.3).

The focus of usability evaluation is on how well the end-users would interact with a GUI, how easy it is for them to learn about using the software, and also how effectively the system would support end-users in performing their tasks and recovering from errors. In order to accurately collect such data, it's important to comprehend when and how to rely on the various evaluation methods that were developed through rigorous usability engineering research.

Generally, there are three main approaches for evaluating usability, based on the type of data used for checking the usability quality of the GUI. The first approach focuses on end-users, the second approach relies on the knowledge of usability experts, and finally the third approach is centered around models of human-computer interactions. Therefore, usability evaluations can be described as user-based, expert-based or model-based [71].

User-Based Evaluation

There are two types of user-based usability evaluation. The first is called formative evaluation and the second is summative evaluation [72]:

- Formative usability evaluation: As the name implies, formative evaluation (which is also called formative testing) helps in forming the design of a GUI. This process evaluates the usability of the software starting from the design stage until the end of development, and often occurs in an iterative manner, which proves significantly useful when working with prototypes. It's important to note that formative testing techniques involve cooperating with a representative user sample that performs specific tasks through a GUI, in order to gather information for improving the design of future prototypes iteratively until producing the final version. Evaluation techniques that exclude the aspect of working with representative

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

end-users, are not considered part of formative testing [73]. The same applies for other evaluation methods in which the users do not perform tasks or directly engage the design of a GUI. This means that techniques such as surveys are also not part of formative usability evaluation. Furthermore, the goal of formative evaluation is to collect data for eleminating usability problems in future design iterations, and it's not about measuring usability. Therefore, it's better for these tests to be conducted in a fast pace. Consequently, for the evaluation to happen quickly and more frequently, the user sample needs to be small, and the tasks have to be directed at a specific part of the GUI. This also typically results in more informal test reporting mechanisms [71].

- Summative usability evaluation (also called summative testing): It focuses on assessing complete or near-complete GUI designs. Summative testing techniques evaluate and document the effectiveness, the efficiency, and the user satisfaction degree of a GUI design. This is done towards the end of development, and is performed in a more formal manner than formative evaluation. If the software developed is targeted at different user groups (or user classes), then summative tests has to include representatives for each one of these groups. Typically, each class of end-users is represented by about 5 to 7 participants [71]. The tasks to be performed depict the main system functionalities. Their purpose is to find out if the GUI design meets some specified goals. Therefore, summative evaluation techniques might be employed to measure usability in a laboratory environment that tries to duplicate realistic conditions of use, and eliminate distractions.

In short, formative evaluation plays the role of usability verification, while the summative evaluation approach delivers a usability validation for the final design (with verification and validation being concepts described in section 3.1). Both testing methods are centered around users and no specific approach can be declared superior to the alternative.

Expert-Based Evaluation

Besides user-based evaluation, there are also techniques for expert-based evaluation. These can rely on guidelines and standards, or rely on the knowledge and skills of usability experts. This makes them less expensive, less time consuming, and easier to perform. However, these evaluation techniques cannot uncover all the usability issues that real end-users might encounter.

Model-Based Evaluation

Model-based usability evaluation relies on models to predict user behavior. Once the models have been prepared and validated, they can be used for repeatedly testing the usability of any number of GUI designs. This makes the process less expensive than its counterpart, however, the first steps in creating accurate models and their validation are very time consuming [71].

Furthermore, even though the usability evaluation process might differ depending on the selected approach, the process preferably include some common steps that are stated in the following order [74]:

1. Specify the goals of the usability evaluation.
2. Determine which aspects of the GUI to evaluate.
3. Define the target user group.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4. Choose the usability metrics
5. Choose the evaluation method or the combination of methods.
6. Choose the tasks to be performed.
7. Design the experiments.
8. Collect and record usability-relevant data.
9. Examine and interpret the gathered data.
10. Suggest usability solutions or improvements.
11. If needed, partially or completely iterate the process by, for instance, repeating the experiments to see if the suggested solutions work, or by going further back in the process to determine new aspects of the GUI to evaluate [75].
12. Present the evaluation results in a straightforward and an easy to understand format.

Choosing how and when to evaluate the usability quality of a project is a decision to be made at the start of a project depending on various factors such as the budget, the time constraints, and the risk associated with the tasks to be performed. More concrete usability evaluation instances are described in section 3.3.3, alongside a more in depth method classification.

3.3.2 Benefits of Usability Tests

Even though there are many different types and methods of usability evaluations with various strong points and weaknesses when compared with each other, they still all hold some common benefits that make usability tests worth the effort. Some of these merits are the following [8] [34]:

- Increasing user satisfaction: It's very important for the products or services of a company to meet or exceed the requirements of their clients, because that would translate to a higher rate of customer satisfaction, and by extension, to higher customer purchase intentions and lower client turnover rates. Even if a software product or service offers excellent features and much needed functionalities, it can still have low customer satisfaction, if using the software frustrates the end-users due to some usability issues to the point that they can't use some software features effectively. In other words, if a big percentage of users cannot figure out how to properly use a feature, that particular feature would become a liability. Usability evaluation efforts help in identifying and solving design issues, which greatly improves user satisfaction.
- Decreasing customer support costs: Customer support refers to services by which companies provide assistance to its customers in making cost effective and correct use of their products and services. It can be delivered through e-mail, a website, or some live support software. Generally, the more difficulties clients are facing when using a software, the more likely they are to contact technical support. This means that usability issues would translate directly into higher customer support cost. By contrast, detecting and solving these issues preemptively would save up on post-release costs immensely.
- Increasing employee productivity: By increasing the usability value of a software, its end-users would be able to perform their daily tasks faster. If the software built is targeted at business users, then it's important that new employees would quickly learn how to use the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

software and perform their tasks. Also, error prevention is important, and if users do make mistakes, the software should assist them in the recovery. Consequently, usability efforts help employees in doing more work in less time, thus, increasing their productivity.

- Reducing development and redesign efforts and costs: The early detection of usability problems can reduce the cost of development and redesign. The same goes for documentation cost, by ensuring the software has a high usability quality, the need for detailed documentation is minimized. Moreover, the knowledge gained through usability evaluations can be applied in future designs of other similar products, and is also useful for avoiding wasting time and effort on features that users do not need [76].

- Increasing the revenue: Naturally, user-friendly products are better accepted by customers and resonate well with their needs. This is especially accentuated if the product is in the field of e-commerce. For instance, usability evaluations would check how easily a client can find the product he's searching for, and how effectively the design guides him in his purchase decision process. Consequently, improving usability increases sales and user trust in the offered product or service [76].

- Resolving team disagreements regarding design: Usability tests are a great way to stop design arguments in a team because metrics can be relied upon to assess different designs, compare them, and prove with numbers which one is more user-friendly (such methods of measurement are discussed in section 2.2.3).

- Gaining a competitive advantage: By increasing the usability of a software product or service, the overall user satisfaction also improves (as previously described). This constitutes an advantage for a company when clients compare it with the competition. However, for usability efforts to yield the highest advantage value, it's advised to perform competitive usability evaluations. This type of evaluations can be applied to any usability test, but expert-based methods are the most appropriate. The purpose of competitive usability evaluations is to assess a certain GUI design used by a company compared to the designs of its competitors. It allows for an understanding of what the strengths and weaknesses of other companies are, and how well their designs accentuate their offered features. The lessons learned would help in avoiding the mistakes made by the competition, and thus they also reduce risks of losses and misuse of resources. Furthermore, these evaluations can also determine functionalities or design approaches that significantly aid users, and as a result, they assist in guiding the development and design efforts towards implementing these features.

- Improving accessibility: Accessibility is a subset of usability and it addresses how usable a software product or service is for users with disabilities. Companies offering software solutions with good accessibility tend to retain the loyalty of their disabled customers [77]. However, several accessibility requirements also apply to users and situations that are the focus of inclusive design. In other words, improving the usability and accessibility would also benefit users without disabilities such as those with low literacy, those who are not fluent in the language, or those working on legacy equipment [78].

- Maintaining or improving the company's reputation: Sometimes, just one failed product ruins a company's reputation even if all its previous products were well received by its customers. In the software industry, failing to meet customer expectations might cause clients to lose their trust in the company, leading to huge financial losses since development is very costly. Generally, one of these expectations is for the GUI to be sufficiently easy to use. Therefore, in order to protect or improve a company's reputation, it's indispensable for every one of its products to be of high quality including user-friendliness. Usability

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

evaluations provide the means for ensuring the delivery of this important quality attribute to end-users.

To further comprehend usability evaluation and what every assessment method offers, it's important to examine concrete evaluation methods, how and when they're applied, as well as their similarities and differences compared to other usability evaluation techniques.

3.3.3 Methods of Usability Evaluation

Usability engineers developed over decades of research various evaluation methods for collecting useful data about users and their tasks, examining their needs, solving design issues, and measuring usability improvements. As described in section 3.3.1, it's possible to classify these evaluation methods into user-based, expert-based, and model-based approaches. However, another more detailed and in-depth categorisation exist. Researchers Ivory Melody and Hearst Marti of the university of California classified all usability engineering methods into five different categories which are: testing, inquiry, inspection, analytical modeling, and simulation [79].

The Testing Method

Usability testing refers to evaluating a product or service by having one or more evaluators observe how actual users interact with an interface in order to determine usability problems. There are many concrete methods that belong to this class, some of them are the following [79]:

- Thinking aloud: As the name implies, in a thinking aloud test, the usability evaluators ask the test participants to perform representative tasks using the system while continuously thinking out loud. It enables knowing the true opinions of users about a design. Participants are encouraged to say whatever comes to their mind as they are performing tasks, including what they look at, think, do, and feel. This provides valuable insight about the users cognitive processes, and can be done during any phase of development rather cheaply. Therefore, this technique is among the most popular evaluation methods out there [80].
- Question asking protocol: This may be considered an extension to the thinking aloud protocol, as the evaluator would frequently ask the participants some questions about the product during the time they are performing their tasks. The purpose is to gather data about how the design is being interpreted by users, and in particular about aspects of the GUI that might be neglected by the user during a simple thinking aloud test.
- Job shadowing: The idea behind shadowing is to have the observer accompany the participant in order to examine how the product or service is being used within its natural environment. This means that the evaluator would follow the participant throughout his workday in order to comprehend user behavior in details, and later adapt the design to better suit that behavior. The main rule for shadowing is to not interfere with the participant, because any interference might cause the user's behavior to change or deviate from normal, which compromises the validity of test results. Therefore, if the evaluator needs to ask a question, then he has to direct it at an expert user who is not participating in the experiment. The expert would explain the behavior of the participants to the evaluator, without influencing the users who are performing their tasks.
- Teaching method: In a first step, the participant has to interact with the system until he becomes familiar with it. Afterwards, he needs to describe how to perform the tasks to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

another novice user. Naturally, the novice user has to limit his participation and not become an active problem solver, because the true purpose behind this technique is to encourage the teaching participant to express himself verbally in order to uncover his cognitive thought processes or search strategies. With the teaching method, the participants usually speak far more than during the thinking aloud tests, which leads to capturing more data about the design [81].

- Performance measurement: This technique provides valuable quantitative data based on assessments of predefined usability metrics (as discussed in section 2.2.3). The tests are to be conducted formally in a laboratory environment to accurately collect measurements. The evaluator is strongly advised to minimize or completely avoid interacting with the participants during the experiments in order to not influence the user, and also by extension, the test results. Performance measurements are very useful for comparative testing of different designs, for validating that certain requirements have been met, and also for identifying usability issues that might not be discovered through less formal methods. However, the cost and effort required for performing these evaluations is relatively high compared to qualitative studies such as the thinking aloud method.

- Log file analysis: This technique involves analysing user behavior through the examination of interaction logs. For instance, information such as clicks, navigation paths, and errors are all recorded as log files. This data can be analysed to uncover problems such as which part of the GUI get repeatedly ignored by the user, or if the user's navigation path differs from expectations. Gathering this sort of data is easier for web applications than desktop programs, and naturally, the purpose of this method is to evaluate usability after product release and help visualize superior future designs. Log file analysis can be combined with other evaluation methods to produce a better understanding of user behavior and cross validate evaluation results [82].

- Remote testing: This method allows the observation of user behavior in a natural environment by using a screen sharing software or a tool with support for remote usability testing. This technique can be moderated or unmoderated. To explain in further details, in a moderated remote testing, the evaluator and the participant are present online simultaneously and it is possible for them to communicate or ask questions in a manner similar to that of a laboratory environment. However, in an unmoderated remote test, the participant complete tasks independently without having to ask or answer real-time questions. Consequently, unmoderated tests contain less verbalisations than their moderated counterparts. Remote usability testing is an appropriate solution for companies with limited time or budget, as well as when willing participants are hard to locate [83].

It's important to note that, in a study by Tom Landauer and Jakob Nielsen [84], they derived a formula for calculating the percentage of usability issues that can be identified in a usability test, depending on the number of users participating. The mathematical function representing that relation is the following [84]:

$$f(x) = N(1 - (1 - \lambda)^x)$$

where:

x is the users number.

N refers The total usability issues number.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

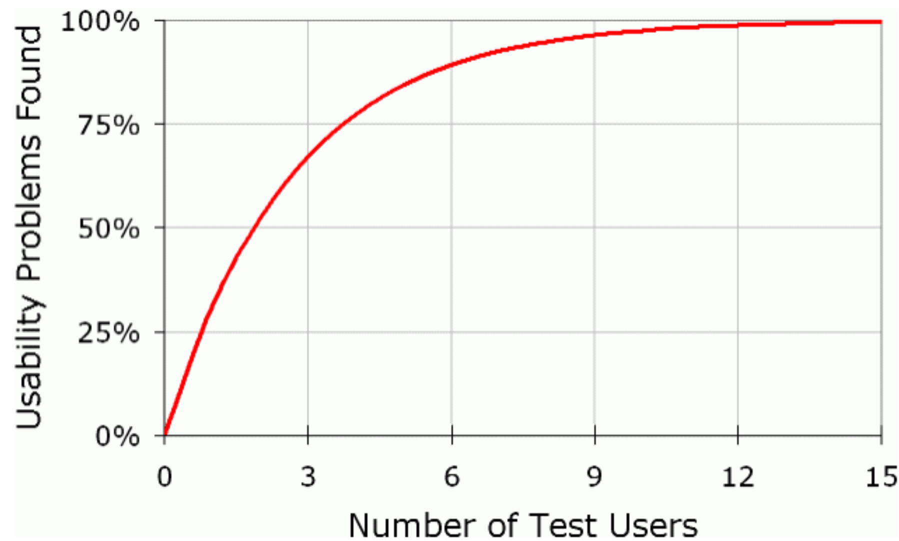

Figure 3.1: Curve showing percentage of found usability issues in relation with the number of participants (see [85])

λ is the predicted probability of finding a usability issue while testing with an average single user. In this particular study, λ is estimated to be ".31", and was deduced by calculating the mean value of previous predictions from 11 past studies, where the number of participating evaluators range between 11 and 77.

$(1-\lambda)$ is the predicted probability of a usability issue remaining undiscovered during a test, given that it has not yet been found in previous tests.

Naturally, the number of users cannot be negative, therefore it's enough to graphically represent the positive input of the function. This results in the curve shown in figure 3.1. The x-axis of the graph represents the percentage of found usability issues, while the y-axis shows the number of users. Examining that graph, it can be derived that relying on 15 users would uncover 100% of the usability problems. However, with just 5 users, 85% of the problems can be identified. Consequently, testing with 5 participants can yield the biggest return on investment. Even if a company has the budget to perform the experiment with 15 users, it would be preferable to perform 3 usability tests with 5 participants in a context of iterative design [85]. An exception is when the target users are divided into highly distinct groups, then, each group needs to be represented by at least 3 users. It should also be noted that discovering 100% of the problems outside of a lab environment is challenging to prove. However, in this experiment, the usability engineers had total control over the number of usability problems that they wanted in the system. It's therefore possible to claim with certainty that 100% of the issues have been uncovered [85].

The Inquiry Method

This group of usability methods collects subjective usability-related information by asking users about their views and opinions about various aspects of the GUI in order to learn from their feedback. These methods can be combined with usability testing or they can be performed after the product release to gather supplementary data that guides design decisions of future releases. Moreover, the evaluators can rely on inquiry methods in the early stages of design to investigate and assess user needs. Some of these methods are the following:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Individual interviews: The evaluator asks a user a series of questions about the system and its issues. The discussion with the user can be either structured or unstructured. In a structured interview, the questions are prepared in advance and each user is asked the same questions in the same order. This guarantees that the collected information can be correctly aggregated, and also enables the comparison of interviews results within various user groups or between different time periods. However, an unstructured interview is not composed of standardized questions. As this method is less formal, the relationship between the evaluator and the user is more balanced which leads to a more natural conversation, and by extension, the collected data suffers less from social desirability bias [86]. Finally, a semi-structured interview is a method that combines prepared questions with some open questions, so the conversation can be guided toward a particular issue to be explored further if needed.

- Contextual inquiry: This is a semi-structured interviewing method aiming to obtain data about the context of use. First, the interviewee is presented some standardized questions, then he's observed in his regular environment and asked questions that arise from his behavior in that environment. Therefore, the data collected is more accurate and more realistic than that of a usability laboratory. The use of contextual inquiry is appropriate when defining usability requirements for improving work practices in unusual technical, physical, or social environments. Moreover, it's conducted early on in development and can last for up to a year or more, since it's usually intended as a long term research study [87].

- Field observation: This method is less structured and less formal than contextual inquiry. The evaluators would observe the participating users, as they're working on a system in their natural environment, and may also ask them questions. However, contrary to a contextual inquiry, field observation is usually orchestrated after product release. In other words, the information collected would, most of the time, help improve a future release of the product instead of creating one from scratch. The observation can occur directly by having the evaluators present on site, or indirectly, by video recording the users as they are performing their tasks.

- Focus groups: This is an informal method for understanding people's feelings and opinions about a certain design. Generally, the evaluator would moderate a two-hour meeting between about six to nine users to discuss issues concerning some aspects of a GUI. The proper purpose of focus groups is not the assessment of the usability of a design, but it's rather the identification of user needs and preferences. Therefore, focus groups should not be the only data source for researching user behavior [88]. Compared to other methods, this technique provides a quick understanding of user assumptions, and efficiently gathers information from a sizeable number of end-users.

- Surveys: This method is suited for collecting quantitative data about user opinions about a product or service (including its GUI). The main purpose of surveys is to rate user experience and user preferences regarding some system features. Because the evaluator is not present when the user is filling a survey, the questions in surveys (and their possible answers) should be developed under careful considerations in order to produce results that are relevant for improving the design. Traditionally, survey forms are mailed to users by post, but that process is gradually getting replaced by internet surveys.

- User feedback: This method provides users the opportunity to directly send feedback about a product or a service. Possibilities for user feedback is often introduced in the GUI itself, in form of text-fields and a submit-button, or as a link to a third party service specialized in collecting user feedback. The user is free to choose whether or not to participate, and the information gathered might be surprisingly useful, in generating new design ideas, as well as in voicing any frustration the user might be experiencing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In order to design a better GUI, it's important to focus more on what users do, rather than on what users say (because users do not always comprehend their real wants and needs) [89]. Generally, studies of user opinions about a system are less reliable for usability assessment than studies about user performance [90]. Therefore, inquiry methods should not be the only sources of data for evaluating and improving usability.

The Inspection Method

Usability inspections refer to a group of methods where an evaluator uses a group of rules of thumb, standards, guidelines, or heuristics in order to identify possible usability problems in a GUI. Contrary to usability testing, these methods do not require the presence of actual end-users which makes them considerably cheaper to perform. Usability inspections can be carried out early in development to verify the quality of the produced prototypes, or later on in development to assess the overall usability of a system. Among usability inspection methods are the following:

- Guideline review: In this method, one or more usability experts would check a GUI for conformance with a list of guidelines. As such, many accepted guidelines were developed over the years. For Windows, Icons, Menus, and Pointers (WIMP) applications in general, it's possible to rely on the Smith and Mosier guidelines (that include close to 1000 rules) [26], or in the case of Microsoft in particular, there's the online user experience guidelines for Windows-based desktop applications [91]. Mobile applications as well have several different guidelines that are mainly proposed by manufacturers. Some instances are the iOS human interface guidelines [92], the Android user interface guidelines [93], and the UI guidelines for Windows Mobile [94]. When it comes to web applications, several source material for reviewing exist. They deal with different contexts of use and vary in level of quality. Some general references for performing a guideline review on websites, are Nielsen's 113 guidelines for homepage usability, and the ISO 9241-151 standard for guidance on world wide web user interfaces [95]. By contrast, context-specific web guidelines might contain some significant differences and might focus more on other quality attributes such as privacy in the case of medical and health online services, or trustworthiness in the case of e-commerce.
- Cognitive walkthrough: In this method, the usability expert simulates the user's problem solving process while performing tasks himself. This can be applied to a completed GUI, a prototype, or even a paper mock-up. At each step of each task, the evaluator has to examine, judge, and document how well (or how bad) the GUI guides the user toward the next correct step. Cognitive walkthrough have evolved over time into a variations of techniques and developed many extensions, however, in all of them, some methodological weaknesses remain when preparing for the evaluation such as difficulties in proper task analysis, evaluator training, and context considerations [96].
- Heuristic evaluation: This involves having one or more evaluators examine a GUI independently and check it against an accepted set of usability principles. The informal analysis helps in identifying usability issues cheaply and quickly compared to other methods. Figure 3.2 represents a heuristic evaluation of a banking system case study, during which 19 evaluators identified 16 usability problems with varying detection difficulty [97]. Every black square in the figure represents an issue discovered by an evaluator. Judging from the non-overlap in the visual distribution of those squares, it's safe to assume that even the most skilled evaluator would not be enough to identify all usability problems, nor would he necessarily uncover the issues that are the most difficult to find. Therefore, it's preferable to have multiple evaluators conduct heuristic evaluations to identify the highest number of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

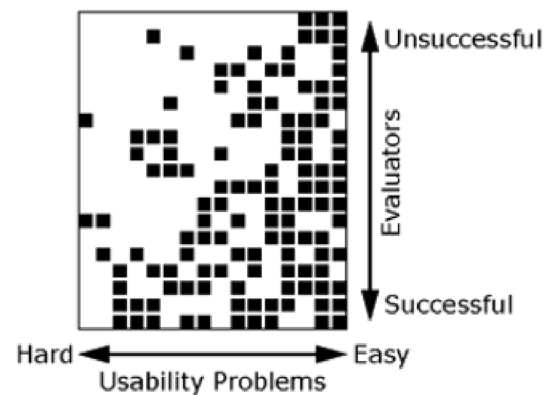

Figure 3.2: Reperesentation of usability problems discovered by evaluators in a heuristic evaluation case study (see [97])

violations. Figure 3.3 holds a graphical curve that shows how the return on investment performs in relation to the number of evaluators [97]. Consequently, relying on about three to five evaluators delivers the most beneficial outcome, because using only one evaluator would probably not uncover all issues, while using more than five evaluators might render the cost too expensive. Moreover, in heuristic evaluations, major usability problems have a higher discovery probability than minor issues, which is a valuable advantage in iterative designs because major problems would be fixed earlier in development [98]. Typically, the results of the evaluations are aggregated and each problem is attributed a severity rating by an evaluator. This provides a cheap way of quantitative measurement of usability, and also delivers useful insight and valuable improvement suggestions for designers and developers that assist them in prioritising design revisions or new features. It's also important to note that the number of principles examined in a heuristic evaluation are significantly less than the number of principles checked in a guideline review. This is because redundant heuristics are eliminated alongside those that rarely apply in a general context.

- Feature inspection: In the first step of this method, the tasks that user would perform are identified, including the series of features needed for performing each task scenario. Then, in a second step, each feature is evaluated based on its understandability, its usefulness, and its availability for the user. This technique also checks whether the tasks require too long of a feature sequence, or if one step in the process is unnatural or too complicated [99]. In a general sense, this approach focuses on the importance of properly developed features in the overall improvement of usability. Therefore, it emphasizes on prioritizing the inspection of features needed in the most highly critical tasks.

- Consistency inspection: Through this method, evaluators evaluate the consistency of a GUI design within a single screen or between multiple screens. The consistency tests examine the text (such as the spelling, and the font), the graphics (such as icons, color, and layout), and aspects of the interaction (such as command names, or the order of steps in a task). Naturally, this inspection should be exaggerated, thus, aspects that need to be different should be distinguishable. This approach ensures that a company would have enough consistency between its multiple software products [100].

In a study by Tasha Hollingsed and David Novick [101], they found that heuristic evaluations are widely used in practice long after their inception. Furthermore, half of the top ten winners of best intranets in 2005 used heuristic evaluation, while in 2001 and 2002, only 10% of them used

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

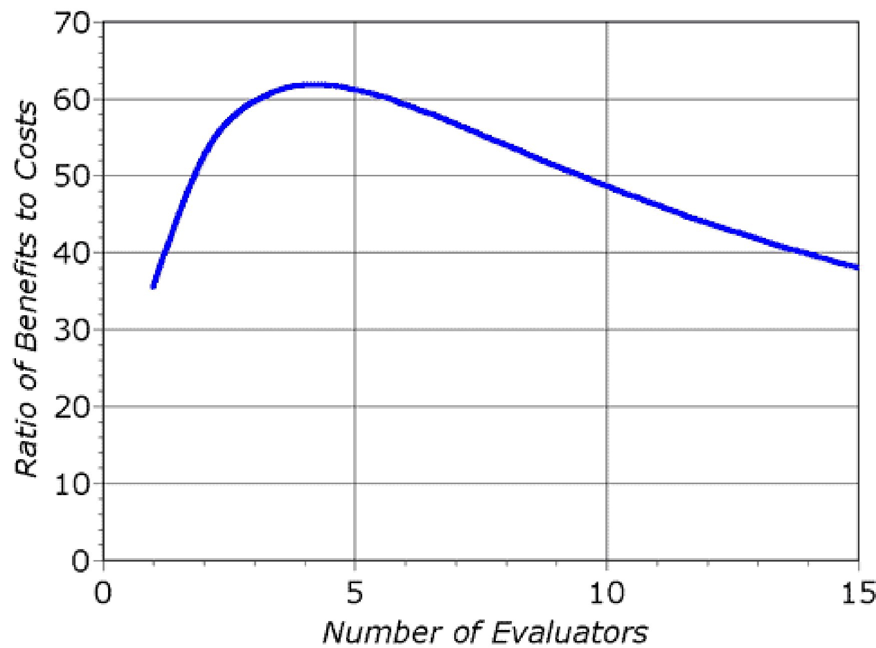

Figure 3.3: Curve showing the return on investment in relation to the number of evaluators in a heuristic evaluation (see [97])

this method [102]. On a more contemporary note, the winners of best intranet in 2016 and 2017 focused on concept reviews, process reviews, style guidelines reviews, combined with inquiry methods such as interviews, and usability testing methods [103] [104]. Continuing with the less popular inspection methods, cognitive and pluralistic usability walkthrough are being used less frequently with time, and formal usability inspection already experienced the peak of its usage in the mid-1990s [101].

The Analytical Modeling Method

Methods of analytical modeling are based on a representational model of the user and the interface. An evaluator would be able to effortlessly generate usability predictions through use of that model. In this category of methods, evaluators would, for instance, write an abstract model of the system under test just once, instead of manually writing hundreds of test cases that assess usability by predicting user behavior. What's more, it's possible to generate a variety of test suites from the same model by simply selecting different test criteria. The models can focus not only on analysing the task environment, or the user's required knowledge, but also on representing the user's performance, or the GUI [105]. Some techniques for usability evaluation through analytical modeling are the following:

- Goals, Operators, Methods, and Selections (GOMS) analysis: A GOMS model is a specialized representation of the human information processor derived through psychological research on human-computer interaction. This model describes the user's cognitive structure, his perception, his short term and long term memory, and his visual and audio information processing. It also introduces a group of goals, operators, methods for meeting the goals, and some selection criteria for deciding which method is better suited for achieving the goals. Each method is composed of a group of steps or operators, while each interface may support multiple methods all leading to the same goal. GOMS analysis can detect

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

potential usability problems by predicting user behavior in the model. In other words, the evaluator can rely on a GOMS model to generate predictions about possible usability issues. However, the model cannot identify usability issues related to aesthetic or linguistic aspects. Because this method takes a goal-oriented view, it is required to analyse tasks in details, and determine all the goals, operators, methods, and selection rules [71].

- Cognitive task analysis: This type of task analysis has the goal of understanding tasks that need rigorous decision-making, problem-solving, memory, and judgement. Cognitive task analysis methods examine and modelize, all the cognitive activities needed from the end-user to perform complex tasks. This can help evaluators in perceiving performance differences between new and expert users, in assessing the mental workload associated with a design, or in analysing information requirements for the GUI of a complex system. Some of instances of the methods are the Applied Cognitive Task Analysis (ACTA), the Critical Decision Method (CDM), the Cognitive Function Model (CFM), and the Task-Knowledge Structures (TKS) [106].
- Task-environment analysis: This method is applied for examining the relationship between how the tasks were originally completed by the user (in his environment), and how these same tasks are mapped in the GUI. Analysing and representing these associations provides the evaluator with information about the learnability, the consistency, and of course the functionality of the GUI. An example of a task-environment analysis method is the External Internal Task Mapping (ETIT) which can be used for assessing the extent of knowledge transfer between multiple designs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

The Simulation Method

The simulation method relies on user and interface models to simulate some user interaction with the GUI. Then, based on this interaction, some usability issues are reported. Simulation techniques might be combined with usability testing methods. In these sorts of evaluations, the usability evaluator would make use of models of users and interfaces aiming to imitate a user interacting with an interface and reports back his findings. Simulators can be run under various parameters to examine different design directions. Some concrete design methods are the following:

- Information processing modeling: This model is based on research in cognitive development and psychology regarding how information is gathered, manipulated, stored, retrieved, and classified by a human. Information processing models can be used to mimic user behavior when interacting with a GUI.
- Information scent modeling: By better comprehending how users seek information, it's possible to improve the usability of a GUI. For instance, humans estimate the amount of useful information they are likely to get on a specific navigation path, and then, they compare the actual value of the information they got with their expected outcome. This can strongly affect user engagement and trust [107]. Information scent models aim to mimic user navigation and discover usability issues.
- Petri net modeling: This method can be used for user performance modeling. When based on previous usage data, it can simulate the future interaction between a user and a GUI, and assist in detecting potential design problems.

3.3.4 The Need for Automating Usability Evaluation

Even though the importance of actual end-users for usability research is undeniable, the most accurate and the most efficient results are produced from usability tests where one or more specialists directly observe representative users as they interact with a GUI to perform task scenarios [90]. However, the majority of usability evaluation methods can be partially or sometimes completely automated. Before examining which aspects can be automated and the degree of automation that can be introduced in the different evaluation types, it's important to first explain the reasons why automation in this field is a must-have. Some of these benefits are briefly discussed in section 3.1.7. Other advantages that are more specific to automated usability testing are the following [79] [108] :

- Simplifying the process of comparing designs: Comparing two designs requires an accurate assessment of both alternatives under the same conditions and using the same metrics. However, measuring usability performance formally demands a high cost in time, effort, and resources (as discussed in section 2.2.3). A solution to this would be automated analytical modeling methods. By relying on models, usability predictions can be effortlessly generated for as many designs as needed. This simplifies greatly the comparison process, especially since this can be integrated in the iterative design process and start early on in development. By comparison, non-automated performance measurements only occur after the GUI have been implemented.

- Decreasing the evaluation cost: For most usability evaluation methods, a considerable portion of the process is repeatable and simple. Automating these types of activities reduces the work time, and by extension the overall cost. Such activities can be, for instance, automated event logging instead of manual logging, or automating the inclusion of surveys before and after an unmoderated remote usability test instead of directly conducting a structured interview.

- Decreasing the dependency on human expertise and specialized knowledge: Some usability inspection methods, for instance, are strongly influenced by the skills and experience of the evaluator (such as in a cognitive walkthrough), or by the knowledge of a domain expert (such as in a standards inspection). By automatically checking a GUI against a fixed set of conditions, it's possible to detect issues early on or even prevent them.

- Simplify cross validation: Performing one single type of evaluation cannot always guarantee acquiring all the data needed to improve usability. Consequently, it's better to combine multiple evaluation methods that complement each other. However, this introduces additional costs and might drain the development budget and resources. Therefore, by relying on a repeatable automated evaluation in an early stage, then performing these tests alongside a non-automated usability test when the GUI is sufficiently developed, then the evaluation results can be cheaply and effectively cross validated. Examining the cross validation results, also simplifies decisions regarding the prioritization of which usability issues to fix.

- Performing tasks that are too complex for manual evaluation: Usability evaluations through analytical modeling or simulation methods provide valuable insights and predictions regarding user interaction with a GUI. Such methods are very appropriate for automation, and should not be performed manually.

- Enabling remote evaluation: Evaluators can rely on software to enable a large number of geographically separated users to participate in usability evaluations. This also saves up on the travel budget and the cost for recruiting participants.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- Supporting and simplifying iterative design: Naturally, automation makes tasks effortlessly repeatable. Therefore, relying on some forms of automated evaluations would support iterative design, since the usability of each iteration can be checked without draining the budget or wasting time.

The purpose of automation is not to replace the user or the evaluator. Its goal is to reduce the overall evaluation cost, and to lessen the burden on the evaluator by supporting him through partial or complete computerization of the evaluation process. However, the distinct usability evaluation methods lend themselves differently to the concept of automation. Therefore, it's important to examine and understand their compatibility degree.

3.3.5 Toward Automating Usability Evaluation

Generally, the taxonomy used for classifying the compatibility between usability evaluation and automation, arranges methods into four categories [79]:

- Non-automatic: This refers to methods that should be performed manually while supervised by human experts. The use of audio or video recording devices is allowed, the same goes for other technologies that assist the user as long as no operation in the activity can be computerised. For instance, logging evaluation observations on a computer does not make the process automated. However, it would be the case if a software tool logs the interaction details without the evaluator's interference.
- Automated capture: This category refers to methods that can depend on software tools for capturing and recording usability-relevant data associated with an interaction session between the user and the system. Instances of captured information can be keyboard presses, mouse movements, navigation paths, and the time needed for task completion.
- Automated analysis: This class refers to methods that are able to automatically detect and locate usability issues. Naturally, it exceeds automatic capture in complexity.
- Automated critic: This methods suggests solutions or improvements to the usability issues detected. This process implies a successful automatic analysis, and is the most challenging to achieve.

These four classes might have provided enough abstraction for previous research, but automation has progressed to a point where it reached a new aspect that was not covered by previous taxonomy. Therefore, a new possible category is proposed in this paper, and it is described in the following:

- Automated planning: Before performing an evaluation, the evaluator needs to make some preparations by performing various activities. These activities can differ depending on the method used and the specified project requirements. They can range from describing the tasks to be performed by users during an evaluation, to selecting the participants that would best represent a user group, or even choosing which questions to include in a survey. Sometimes, the planning phase might incorporate a pilot test with a minimalistic view on the number of users and tasks. Consequently, this sort of planning demands time and effort from the evaluator, thus, it costs more the longer it lasts. Automation can sometimes be used to assist the evaluator during this phase.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Separating planning, data capture, data analysis, and improvement suggestions, allows usability evaluation efforts to be organized and concentrated on one aspect at a time, and for the overall evaluation goals to be achieved in less time, with a better accuracy and a reduced complexity. It's also important to note that the automation difficulty or even feasibility might vary depending on the type of UI evaluated. For simplicity, the discussion is restricted to WIMP, web, and mobile GUIs, with the main focus lying on WIMP interfaces as they have the longest history in research and in practice.

Automating the Testing Method

One role in every testing method cannot be automated, and it is the involvement of the end-user, because replacing the user with a computer program that mimics the user, would by definition change the type of the evaluation into the "simulation" class instead of the "testing" class (both described in section 3.3.3). However, relevant information produced from the interaction between the user and the system can be automatically captured and analysed to a certain extent. In other words, the automation focuses on the responsibilities and activities delegated to evaluators

Starting by the automated capture, its use is most suitable for remote testing, log analysis, and performance measurement. Instead of having the evaluator take notes while directly observing the participant's behavior, or while repeatedly watching a video recording of the participant's interaction with the GUI, it's more efficient to log user activity automatically. While each task is being performed, a usability software tool can easily record relevant information such as button presses, mouse movements, time needed, navigational paths, and errors made. This data can be used as a basis for identifying usability issues. Due to the voluminous nature of such log files, some tools can filer the logged data automatically and classify them according to their relevance to usability. For instance the number of prompted error messages are more important than the distance that a user's mouse has moved during a task.

Automated capture in remote testing gathers information from a much larger user sample than what can be tested in a laboratory. Some online services such as "UserTesting" [109] specialize in unmoderated remote testing of websites and mobile applications, and so far delivered to their clients over one million videos of users performing tasks [109]. Participants are first instructed about the thinking aloud method before the test, and are also asked to give additional feedback following the test. Afterwards, the videos are examined and delivered to clients with relevant notes and observations in a very helpful fashion similar to the screenshot in figure 3.4. The right side of the figure presents the observations made and how they are mapped to their corresponding timestamps in the video, while the area on the left and center shows the recorded video being played. Even though there's no hints about the automation details for producing such detailed notes, the automation process is possibly linked to advanced forms of datalogging and speech recognition, where a large range of observation classes is prepared and arranged in advance according to the context and nature of the interaction and its designated tasks.

In some cases, the automated capture can collect results that exceed the capabilities of an evaluator. For instance, by video recording the user while relying on eye tracking software and emotion recognition APIs, usability can be measured in a manner that cannot be researched by other methods. In other words, automatically measuring the user's emotional response and analysing the implication of his visual focus points, can help in understanding the design elements that attract user attention or trigger his emotional engagement [110]. However, more research on this is needed to increase the accuracy of mental state measurements and interpretation.

Automated analysis can be performed by analysing the log files. For instance, the analysis can be based on metrics, on tasks, or on pattern-matching. The metric-based approach does not necessar-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

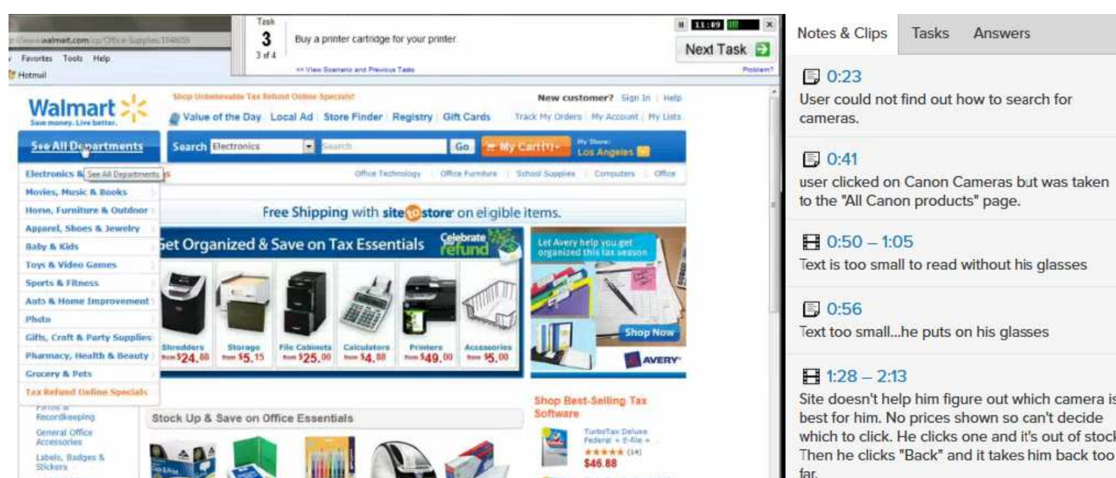

Figure 3.4: Partial snapshot of a user sample test from "UserTesting" delivered with attached relevent notes and observations (see [109])

ily rely on the same metrics discussed in section 2.2.3. Automation can be used to analyse, aggregate, and statistically measure completion rates, completion times, the number of errors made, the number of physical operations required to perform a certain task, event call frequencies, and variations in user attention to different GUI areas during the test. Based on such data, some usability tools might analyse the cognitive or behavioral complexity of the tasks and then associate higher complexity with low usability quality because a high task difficulty hinders the problem solving process and decreases learnability. Therefore, high interaction complexity is associated with presence of usability issues. For instance, Automatic Mental Model Evaluator (AMME) is a program that transforms the recorded log files produced from the user interactions in a form that can be further analysed, then it automatically measures cognitive complexity, behavioral complexity, task complexity, and perceived complexity [111] [112].

The task-based approach compares the user's interactive experience with the designer's perspective on normal task completion. In other words, the log files produced when the designer performs events on the GUI, are examined and compared with the log files generated by the participants in the usability test. Through this analysis, software tools can discover patterns of inefficient or incorrect behaviors, and pinpoint the activities that suffer from usability problems [79].

Software tools based on the pattern-matching approach automatically analyses the captured user logs, to detect repetitive user behavior, such as consecutively calling the same commands, or being continuously confronted with the same error. These patterns are generally the result of usability problems, and therefore, a pattern-matching analysis would locate these issues [79].

If programs take task and pattern analysis further, then usability issues could be arranged in classes, and mapped to possible usability improvements. For example, if users repeatedly make the same mistake while they submit a form, and prompt the same error message each time, then that error message is probably not helpful (provided, of course, that the activity is feasible). Consequently, a possible usability improvement would be, for instance, to make the error message text more concise, and visually highlight the specific fields causing the errors. However, accurate automated critique based only on log files is challenging.

An additional concept whose implementation might improve the automation in usability testing is the use of machine learning. It focuses on studying algorithms and constructing systems that automatically and progressively improve in solving a task through experience [113]. This have great potential for further improving automated analysis and automated critique of usability. By relying on machine learning, the performance of various different evaluation models can be mea-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

sured to find the most suitable one for a specific context of use (such as eLearning systems, or eCommerce sites) [114]. Machine learning techniques can also be suitable for testing usability issues related to the organization and navigation between web pages as shown by the work of Christoffer Korvald, Eunjin Kim and Hassan Rezabut despite their training set being composed of semi-random data [115]. Moreover, it's even possible to use machine learning for analysing and prioritizing which usability improvements are the most needed, by ranking the importance of usability determinative factors in relation to the overall usability of a design for web-based information systems [116].

Automating the Inquiry Method

The automation of this type of usability evaluations is possible through the use of embedded questions that the user would be prompted to subjectivity answer. Naturally, the main goal of inquiry methods is to collect data about user preferences and opinions, more so than identifying existing usability problems. The gathered information would be then used to help improve usability [79]. Therefore, automated inquiry evaluation suffer from the same limitations as their non-automated counterparts, it's challenging to automatically analyse data and locate usability issues, and even more challenging to suggest improvements. However, automation inquiries seem very successful in data capture, and to a lesser degree in automated preparation.

Generally, questionnaires or surveys can be displayed within a GUI, and users can occasionally be prompted or suggested to participate after they finish conducting some of their main tasks. The answers chosen are more accurate when the software asks users to answer immediately after they finish their tasks, because the interactive experience would still be clear in the mind of users.

The frequency at which users are encouraged to participate in an inquiry evaluation should be as low as possible, because asking users multiple times might irritate them, which of course, decreases the usability. In order to avoid that problem, it's preferred to initially present the survey as a link (or a command) instead of a pop-up [79]. Then, if the users would voluntarily choose to participate, their answers would be more thought out than those of users who feel forced into participating. Additionally, using an incentive (or a prize) might significantly increase the motivation of users to participate, but this might also corrupt the validity of the collected results. A considerable portion of users will not necessarily provide honest answers. For instance, websites that specialize in online surveys, and offer participants monetary incentives, are constantly confronted with users who rely on artificially intelligent internet bots to unethically maximize their profits.

Another issue is associated with open questions. Compared to close-ended questions with a fixed set of answers, the answers of open questions are difficult to examine, aggregate and classify automatically. In other words, it's simple to automatically produce relevant statistics based on answers belonging to a limited set of choices [79]. However, users reply to open questions in very different textual forms. Some software tools solve this problem by applying sentiment analysis, text mining, or computational linguistics to derive and quantify the tone and meaning of the written text. This also benefits other inquiry methods that do not have open questions such as the user feedback method (which supports automated capture by default).

Basically, automated capture in inquiry methods can rapidly collect and categorize a great deal of information from a large number of people. However the value and quality of the gathered data depend on the effort put by the evaluators in the planning phase, such as in the preparation and design of the questions. Some of the problems resulting from bad planning are the following [8]:

- Having unneeded questions that do not serve the purpose of improving future designs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- Having inadequate answer options, that do not help in classifying or quantifying information.
- Relying on participants from an incorrect user group or demographic.
- Inconsistent rating options between questions.
- Wrongly assuming some prior knowledge from participants.
- Making the survey too long.

Machine learning algorithms can help in avoiding these issues by laying up the groundwork for inquiry methods based on data from previously performed evaluations, which leads in the reduction of the design cost of a questionnaire and the recruitment of wanted participants [117].

With automated analysis, the gathered data can be used to quantify user expectations, and task level satisfaction. By asking users how difficult they expect a task to be, and then asking them again about the actual difficulty immediately after they perform the task, it's possible to compare both measures and automatically detect problems. A problem is present if the actual difficulty exceeds user expectation. The same can be applied for task level satisfaction in order to detect when one particular task is way more difficult than other tasks with similar activities. Naturally, this works better when based on a large database of previous measurements.

Automating the Inspection Method

Some software tools tried assisting in automated capture by providing evaluators with forms that they have to fill in, and guiding questions that they progressively answer throughout a cognitive walkthrough [118]. However, this approach proved itself counter intuitive as evaluators found it costly in terms of time and energy [96]. As an alternative, it's possible to video record cognitive walkthrough sessions, and use speech recognition software and voice commands to facilitate logging observations and to minimize the effort put in the documentation (but there is no usability tool on the market with this function yet).

To a certain degree, automated capture and automated analysis can be combined and applied when performing guideline reviews, heuristic inspections, standard inspections, consistency inspections, feature inspections, and perspective-based evaluations. The feasibility of automation for checking one specific principle depends directly from the difficulty and complexity associated with the tested conditions.

Two different examples can be considered. The first usability principle would be having short system response time (ideally equal or less than one second) [119]. It is effective to automate a test checking how quickly the system responds to user's initiated commands or events. On the other hand, the second considered principle is having an aesthetically pleasant design. Checking compliance with this principle is challenging to operationalize. Some computational aesthetic judgement methods have been developed to subjectively evaluate photographic images [120], but aesthetically evaluating a GUI design automatically is more complex, and more importantly, it is better suited for other usability evaluation types such as analytical modeling.

From this background, it's possible to assume that, currently, automation of guideline-based usability inspections vary in complexity depending from the nature of the principles that need to be checked. Therefore, the evaluator plays a role that cannot be replaced and automating his work should only be done partially (because checking compliance with some particular principles manually is sometimes more accurate and more cost-effective).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Many software tools and services support the evaluators in their inspections. The GUI of WIMP applications can be automatically checked to examine visual aspects such as the alignment of GUI elements and their symmetry, but support for a complete list of guidelines is relatively limited. On the other hand, web applications benefit the most from such tools, as many can automatically check for guideline violations. For instance, the Varvy SEO tool checks for website compliance with Google guidelines [121], and "Cynthia Says" is a web portal service that can check for compliance with web content accessibility guidelines [122].

To perform an inspection of WIMP application on Windows platforms, a method can be used for structuring ergonomic rules according to user interface objects and evaluating them. This method is called Ergoval and can bridge the gap between the developer's view and presented guidelines [123]. AIDE [124] and Sherlock [125] are two automated analysis tools for Windows interfaces that provided valuable support for analysing the efficiency of a GUI but still, nevertheless, leave significant room for improvements since they together just cover checking aspects of the visual layout and the terminology and do not suggest solutions [79].

Due to the large size of guidelines checked in a guideline review, automating the evaluation process is very beneficial. One way of simplifying this would be to arrange the guidelines into groups such as task-related principles, screen-related principles and widget principles. Despite the abstract level of categorization, these exact classes showed great success when evaluating mobile health applications [126]. Task-related metrics include aspects such as the number of steps or the number of needed inputs to complete a task. Screen-related metrics check, for instance, the number of scrollable GUI elements and the average mouse distance travelled on the screen for performing a task. Widget-related metrics deal with aspects such as appropriate default values in input fields and proper text color contrast ratio.

Heuristics are based on a non-redundant generic set of usability rules of thumb. Heuristic-based automated analysis have shown great success in past studies [127] [17]. One of these studies, performed by Justin Mifsud and Alexiei Dingli, resulted in building a usability framework for automated evaluation of websites called "USEFul" [17]. Conducting experiments with this framework showed that it detects 51.48% of the usability violations manually identified by the usability experts conducting the same heuristic evaluation. Besides, the result increases to 95.86% when the tool and the experts are inspecting the exact same number of heuristics [17]. However, this framework only checks the text and Cascading Style Sheet (CSS) files of a website. Thus, it is limited in the type of heuristics it can check.

Automating the Analytical Modeling Method

All analytical modeling methods support automation, and especially automated analysis. Based on the methods explained in section 3.3.3, the evaluator can produce some accurate usability predictions that assist designers in improving the GUI. For instance, automated GOMS analysis can be used on all types of WIMP applications to effectively detect problems [79].

Even complex psychological factors affecting human interaction with a GUI, can be represented in a model and automatically analysed. For example, the aesthetics of a GUI can be evaluated through the use of a Cognitive-Affective Model of Perceived User Satisfaction (CAMPUS) [128].

With this in mind, it's difficult to pinpoint the potential limits of automation in analytical modeling. Usability evaluator tasks that currently cannot be automated, may possibly become automatable in the future when a model representing that particular expertise or knowledge is developed. Models can collect large quantities of information and make decisions removed from common biases that might affect an evaluator's human judgement.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

However, some downsides to model analysis, are the difficulty and complexity that their use might introduce (as they are rich on variables and parameters), as well as the relatively high initial cost for modeling the different tasks or the graphical elements.

Automating the Simulation Method

Since simulation methods are based on mimicking user behavior when interacting with a GUI, these methods inherently support automated analysis. An example would be to construct a simulation model directly from previous usage data and then analyse the problem solving process [79].

Various simulation models have been created to detect different issues related to software development and cognitive problem solving (some of which are discussed in section 3.3.3). However, the closest model to simulate a usability evaluation is UESim [129]. The concepts associated with the simulation model UESim are represented in a causal loop diagram where the relationships are displayed either as positive or as negative arrows. A positive arrow indicates that if one variable increases, so will the variable it links to. On the hand, a negative arrow, means that the increase of one variable leads to the decrease of the one it links to. This automated analysis is a big advantage for a design team that needs to make many logical and effective design decisions, because the UESim tool is customizable, which means that its variables can be changed dynamically to iteratively observe the impact on usability [129].

An additional automation method which still can be expanded upon, is the use of Artificial Intelligence (AI) bots to mimic the human user. AI bots have a lot of potential but it is difficult to draw concrete conclusions about their effectiveness. More research in this area is needed and once AI bots reach a point where their interaction with an interface is close enough to human users, new possibilities and perspectives on usability engineering might emerge.

Choosing suitable Automation Process

It's not simple to determine which usability evaluation process to automate. Some evaluation classes and in particular analytical modeling and simulation inherently support automation, and should be automated no matter the concrete method (because a manual testing approach in these cases would be difficult and take too much time). The effort and required skills for starting work with methods from these two evaluation categories, is very high, which increases the overall initial cost of development. However, software projects that last over a long period of time benefit significantly from using models, as the tests can be effortlessly repeated and the model can be continuously adjusted inexpensively [130].

On the other hand, automation of other evaluation classes such as testing, inspection, and inquiry cuts on the costs and saves on time and effort, but might not be as accurate as direct evaluation [79]. Automated inquiry methods are much cheaper than the traditional paper alternative, can be distributed to a much larger demographic, and their results can be easily presented in form of textual or graphical statistical reports. Meanwhile, automated testing methods are much cheaper, but still cannot replace the cost-effective high accuracy of directly testing with five users [85].

Unlike testing and inquiry methods, inspection methods do not need users to play a role in the evaluation process. Automated inspections can support evaluators in their work which saves time, or help GUI designers avoid introducing new usability problems. Designers often encounter a challenge when following design guidelines. One study demonstrated that they tend to prefer aesthetically compelling interfaces regardless of their efficiency [131]. Additional studies showed that the average manual search time for a guideline is 15 minutes, that about 42% of designers fail to find guidelines relevant to their problem, that designers do not respect about 11% of these

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

guidelines and that they incorrectly interpret 30% of them [123]. An automated evaluation of common principles and guidelines that can analyse an interface, detect problems and even suggest improvements, cuts the amount of required manual tests in a significant way. Therefore, automation of usability inspections should always be considered, especially if a framework or a tool that supports the process already exists.

3.3.6 Limits of Usability Evaluation

Every engineering effort has its practical limits, and usability evaluation is no exception. Understanding what sort of issues this practice does not focus on, or in which aspects it performs poorly, is important for comprehending how to properly conduct an evaluation.

First of all, the results of usability evaluations, and in particular, those of inquiry methods, should not be confused with market research. Inquiry methods collect user design preferences and opinions. On the other hand, market research analyses market needs, its size, and the competition all in order to adjust the business strategy accordingly.

Similarly, usability does not refer to the extent of compliance with guidelines. Usability principles (or heuristics) are important, but blindly applying all guidelines to a design regardless of the context will not automatically increase usability. Each rule can have some exceptions, and the nature and context of the application needs to be considered. For instance, a major usability guideline for the web that was known since the late nineties is that users do not like scrolling. Therefore, websites were designed in a way that minimized scrolling in order to improve their usability. However, exceptions do exist. An example would be Facebook which is one of the most successful websites in history, despite having a design that requires excessive scrolling.

Another issue is related to the need for aggregating results from one evaluation or also the need for cross validation of results with those from other methods. Contrary to regular software testing, usability testing cannot be performed just once by one single evaluator on one single user. Diversity in evaluation methods is needed for accurate and in-depth discovery of usability issues, as well as for rating their severity. To explain this further, different evaluation methods produce different results, different evaluators find different issues and give them different severity ratings, and the same applies for different users, user groups, contexts of use, and cultures. Even though the results, of course, still share undeniable similarities, the small inconsistencies between results might introduce more complexity and uncertainty when deciding and prioritizing design revisions or feature extensions. Therefore, it's essential to first identify the percentage of usability issues that needs to be detected, then select an appropriate and cost-effective group of evaluation methods that complement each other. Choosing these methods would help avoid an overanalysis of usability, and wasting the design budget. However, there's no clear way for selecting the perfect set of evaluation methods for a certain project as they all differ in budget, in nature, and team experience.

Finally, some aspects of the context of use cannot be simulated in a laboratory environment. For example, users under observation usually dedicate their full attention to the task at hand. However, some usability problems can only be encountered by a user performing the task with a realistic level of concentration in a spontaneous manner. This means that problems associated with the user's focus cannot be accurately assessed.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4 Automating Heuristic-based Usability Inspection with GUI Event Sequencing

Studying the degree in which one particular testing field, such as usability evaluation, can benefit from testing techniques designed for a completely different field such as GUI testing, would require first to thoroughly comprehend the extent at which these two fields are similar, and the same goes for the extent of their differences. Moreover, it's also needed to have an overview of the tools designed and successfully commercialized for assisting GUI testers and usability engineers, in order to understand which testing methods have the best tool support, and which methods aren't assisted as much by the currently available tools. Finally, the impact of certain GUI testing automation techniques on usability evaluation would be studied, in order to suggest possible improvements in the automation of usability evaluation.

4.1 Comparison between GUI Testing and Usability Evaluation

Even though GUI testing and usability evaluation had both been described in details in sections 3.2 and 3.3 respectively, it's still important to examine their similarities and differences when compared with each other. It's also needed to have a basic understanding of the scopes of their respective tools, currently on the market. This would help for comprehending which methods of GUI testing and usability evaluation are being favoured by the software tools industry, and also which methods are being overlooked.

4.1.1 Similarities between GUI Testing and Usability Evaluation

The most apparent aspect shared between GUI testing and usability evaluation is their focus on the GUI. Both testing forms can be performed without any knowledge of the source code, and cannot be started if the GUI is not sufficiently developed.

Furthermore, even though the objectives of GUI testing and usability evaluation are different, some of their methods share some resemblances. For instance, when GUI testing is being performed as a black-box test (which is discussed in section 3.1.4 and in further details in 3.1.5.1), a portion of the tasks performed overlap with those designed for usability evaluation, and in particular, the testing method and the simulation method (which are two evaluation methods described in section 3.3.3.1 and 3.3.3.5 respectively). In other words, GUI testing mainly checks if the software's graphical interface truly allows the user to access the functionalities as specified in the requirements, while on the other hand, the usability testing methods and simulation methods focus on determining the degree of the ease of use concerning each provided functionality [8] [129]. This means that in both fields, the exact same user tasks might be performed throughout the same GUI, where the first tests would check if the function is performed correctly, while the latter ones would assess how easy it is to perform said function [8] [65]. Nevertheless, the same task is being performed twice in this case (but it's still for different reasons).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

It's also significant to clarify that GUI testing might go beyond simply checking the feasibility of each functionality, and into investigating whether basic design principles are being respected such as controllability or error tolerance (as presented in table 2.1 among other principles, and is further explained in section 3.2.1 with various examples). It's possible to analyse the design in the form of checklist-based testing (which is an exploratory testing technique described in section 3.1.3). This approach is very similar to the usability inspection process (defined in 3.3.3), which also relies on guidelines to determine the existence of a problem. Naturally, the guidelines for designing user interfaces do not necessarily overlay with usability guidelines.

By way of explanation, some software companies might specify their own set of design guidelines, that have to be consistently respected across all or some of their software products, and naturally, these specified user interface design guidelines might contradict with some usability principles [132]. However, despite the possible differences in both guidelines, what testers work on during usability inspections is still similar to GUI testing if performed manually in the guise of a checklist-based exploratory test. It should also be reminded that checklist-based testing can even check functional requirements, and is not synonymous with design guidelines verification (which means that its scope is dependent on the nature of the tasks specified in the checklist).

In conclusion, the tasks around which the tests are created for GUI testing and usability evaluation are sometimes identical, but their motivations still differ, as well as the significance of the test results.

4.1.2 Differences between GUI Testing and Usability Evaluation

Since usability is a non-functional quality attribute, uncovered usability problems do not indicate the presence of a deviation in the system's behaviour from what was specified in the requirements [8]. On the other hand, the main goal of GUI testing is to verify that the tasks that can be performed by users are feasible as agreed upon and specified in the requirements [65]. Unlike GUI testing, usability evaluation does not aim to find faults or contribute in increasing the test coverage, no matter the evaluation method used, or the quantity of tests performed. The designed tests do not actively try to uncover defects but rather focus on finding usability problems. However, if a bug is discovered during a usability test, it could be documented and reported by the usability engineer.

Moreover, some GUI tests might focus on verifying the look and feel of the design, and check whether it's aesthetically pleasant across multiple supported environments (such as different display settings, browsers, or various operating systems). These tests might validate image appearances, the correctness of the CSS, the layout positioning, and so forth. On the other hand, usability metrics focus on efficiency, effectiveness, and user satisfaction (as was discussed in section 2.2.3), which are also non-functional quality attributes. However, verifying the look and feel of the GUI is not the purpose of usability evaluation, even if such visual GUI tests might seem similar to usability analytical modeling methods, inspection methods, or to a lesser degree inquiry methods (which were all described in section 3.3.3) [133].

For instance, in both a visual GUI test with a focus on aesthetics, and a usability inspection, it's probable that both tests would include verifying the consistency in the alignment of graphical components. However, even if the same tests are being performed, the motivation for doing such tests is different. This means that such a GUI test would verify the alignment to ensure that the design is aesthetically correct. Meanwhile, a usability inspection also checking alignments, aims to measure how well the design assists and guides the eyes of users to relevant GUI areas during the tasks.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

An additional example is the measurement of user satisfaction through usability inquiry methods. The satisfaction level should not be confused with whether or not the users are satisfied with the visual design. It should be reminded that inquiry methods demand from users their opinions concerning the difficulty of the tasks performed and not how much they like or dislike the visual design. Nevertheless, based on several observed experiments, the opinion of users is prone to the aesthetic-usability effect [134]. This effect describes a bias in which users perceive more-aesthetic designs as user-friendlier than less-aesthetic designs regardless of their actual usability degree [134].

One more difference lies in the amount of knowledge and experience possessed by the actors performing the tests. In the case of usability evaluation, it's strongly encouraged that the users performing the usability tests do not have prior knowledge or experience with the system, so that the measurements can be unbiased. On the other hand, when performing GUI testing, the tester does have prior knowledge of the software, no matter the testing method. Even in case of black-box GUI testing, where the tester has the least amount of information (compared to gray-box and white-box testing), the data provided still includes the list of specified requirements, with details about the expected behaviour of the system.

Furthermore, even in the presence of severe usability issues, GUI tests might still pass positively if the functions are working as expected. The same holds true the other way around, even if a task is user-friendly, it might not conform with specified requirements, and thus would fail one or more of its associated GUI tests [8].

As an extreme case, a script-based GUI test might theoretically require pressing an invisible button in order to successfully perform a task. A script-based GUI test can always access the button through the source code, and as a result, the test would be successful. On the other hand, the presence of the invisible button naturally consists a severe usability issue, since regular users have almost zero chance of performing the task.

A second contrasting example would be a usability test that discovers that every member in the user sample was able to easily change their password in a very short time (without a need for verification), making the task performed very user-friendly. However, in this same example, a requirement would precise that passwords can only be edited after checking the authenticity of the user, by means of sending the new password via a verified email or phone number. Consequently, the GUI tests would prove that the task can be performed without verifying the user's authenticity. In other words, GUI testing would uncover a failure to meet the requirement, even though the task was proven to be highly user-friendly.

Based on the two previously described examples, it can be concluded that an increase in the positive results of GUI testing does not necessary lead to an increase in the levels of user-friendliness, and vice versa. Thus, GUI testing and usability evaluation have no positive correlations.

To examine how the results of both fields further correlate, a project is considered where all GUI tests disastrously failed after a certain upgrade, and as a result, users cannot perform any task. Consequently, usability evaluation would detect a strong decrease in user-friendliness, since tasks can no longer be completed by users.

By way of contrast, one last extreme project is considered, where the design has such severe usability issues, that no one in the user sample was able to finish a single task during evaluations, and a usability inspection proved that all guidelines were violated. In such a worst-case scenario, it's still possible for all functional GUI tests to pass, with the exception of tests verifying the aesthetic correctness (also called the look and feel of the design). Since all usability guidelines were violated, then the visual design must have some aesthetic problems, which means that some of their associated GUI tests would fail if performed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

With these examples in mind, it could be noted that a decrease in the number of passing GUI tests might lead to a decrease in the user-friendliness. Additionally, a low usability level can be accompanied with inferior visual aesthetics and graphical consistency (while having no effect on functional GUI tests).

The differences in the relationship between the results of GUI testing and usability evaluation would require more than informal reasoning or some validating examples. Therefore, there's a need for more research in this area, since currently, no practical study dealing with this issue was performed. All previous examples discussed in this section, examined how an increase or decrease in one variable would affect the other. The reasoning is further simplified in the following:

- A high number of passing GUI tests might be accompanied with both high or low user-friendliness.
- A superior level of usability might be associated with both a high or low amount of passing GUI tests.
- A low number of passing GUI tests, generally means fewer offered functionalities, which means less tasks can be performed by users during usability evaluation, leading to lower measured user-friendliness.
- A poor usability level does not affect functional GUI tests at all. However, if the usability drastically decrease to a very low state, then it's most likely to be accompanied by inferior aesthetics and poor visual consistency, resulting in a decrease in passing GUI tests specializing in checking the look and feel of the design.

In conclusion, GUI tests mainly uncover defects while usability tests find usability issues. Secondly, the correctness of GUI aesthetics is not related to user satisfaction with the design. Subsequently, the amount of prior knowledge about the software differ greatly for both testing fields. Furthermore, the results of GUI tests and the usability level have no positive correlation, but still share some degree of negative correlation. Consequently, it's important to not neglect both forms of testing, as neither can replace the other.

4.1.3 Review of GUI Test Automation Tools

Table 4.1 lists some GUI test automation tools whose development status is still active (as of December 2018), sorted by name in ascending order on its first column. The second column have the name of the developing studio that currently owns and maintains the tool. Thirdly, the column "Supported Platforms" demonstrates what kind of GUI the tool can help in testing. It focuses on whether the tools support automated GUI testing of desktop, mobile and web applications, without listing all the detailed types of supported applications or scripting languages in order to keep the table concise. Lastly, the fourth column shows the type of licence covering the tool.

Additionally, two independent analyst groups Gartner Research and Forrester Research investigated, in 2016, some of the most prominent functional test automation tools, and evaluated them based on data such as user needs, expert interviews, customer interviews, and vendor product demonstrations [156] [157].

The research of Gartner required tools to support testing desktop applications and in particular Windows, as well as testing mobile applications and especially those written in Android and iOS [156]. On the other hand, the Forrester research required tools capable of mobile, GUI and API testing, as well as cross browser testing [157].

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Tool Name	Developers	Supported Platforms	License
Ascentialtest [135]	Zeenyx Inc.	- Desktop: Windows - Web applications	Proprietary
AutoIt [136]	AutoIt Team	- Desktop: Windows	Freeware
Dojo Objective Harness [137]	Dojo Foundation	- Web applications	Academic Free License
EggPlant Functional [138]	TestPlant	- Desktop: Windows, Linux and Mac - Mobile: Android and iOS - Web applications	Proprietary
iMacros [139]	iOpus	- Web applications	Proprietary
Linux Desktop Testing Project [140]	(collaboration)	- Desktop: Windows, Linux and Mac	GNU LGPL
Marveryx [141]	Marveryx srl	- Desktop: Java applications - Mobile: Android	Proprietary
Oracle Application Testing Suite [142]	Oracle	- Desktop: Oracle applications - Web applications	Proprietary
QF-Test [143]	Quality First Software	- Desktop: Java applications - Web applications	Proprietary
Ranorex [5]	Ranorex GmbH	- Desktop: Windows - Mobile: Android and iOS - Web applications	Proprietary
Rational Functional Tester [144]	IBM Rational	- Desktop: Windows - Web applications	Proprietary
Sahi [145]	Tyto Software	- Web applications	Apache
Selenium [4]	(collaboration)	- Web applications	Apache
SkillTest [146]	Micro Focus	- Desktop: Windows and Linux	Proprietary
SOAtest [147]	Parasoft	- Web applications	Proprietary
Squish GUI Tester [148]	froglogic GmbH	- Desktop: Windows and Linux - Mobile: Android and iOS - Web applications	Proprietary
Test Studio [149]	Telerik	- Desktop: Windows - Mobile: Android and iOS - Web applications	Proprietary
TestComplete [150]	SmartBear Software	- Desktop: Windows - Mobile: Android and iOS - Web applications	Proprietary
Tosca [151]	Tricentis	- Desktop: Windows - Mobile: Android and iOS - Web applications	Proprietary
Unified Functional Testing [152]	Micro Focus	- Desktop: Windows - Web applications	Proprietary
Visual Studio Test Professional [153]	Microsoft	- Desktop: Windows (in particular Visual Studio solutions)	Proprietary
Watir [154]	(collaboration)	- Web applications	MIT license
Xnee [155]	GNU Project	- Desktop: X Window Systems	GNU GPL

Table 4.1: Comparative list of active tools for automated GUI testing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Consequently, the efforts of both research groups yielded very similar results when deriving which tools are the leaders in the test automation field (including GUI test automation). The winning tools that best met the evaluation criteria, were determined by both groups to be: Unified Functional Testing [152] by Micro Focus, Rational Functional Tester [144] by IBM Rational, and Tosca [151] by Tricentis. The only exception is SOAtest [147] by Parasoft which was only selected as an industry leader by Forrester, as it was not among the tools evaluated by Gartner (since Gartner requires the tools to run Windows desktop applications, and SOAtest focuses on Web UI testing).

The industry leading automation tools are all capable of automating GUI tests, but differ in their performance, in their level of support for different testing methods, scripting languages, or platforms. In order to better comprehend the features these tools offer, some of the best of them are briefly examined in the following [158]:

- The tool Unified Functional Testing was first written by Mercury Interactive, then acquired by Hewlett-Packard Enterprise which is currently part of MicroFocus. This tool was formerly known as QuickTest Professional [152]. It supports the capture and replay method. Besides, it can generate test cases and test scenarios using a model-based approach (by relying for instance on the MaTeLo plugin [159]), and it uses the VBScript language for performing script-based GUI testing. Additionally, the Unified Functional Testing tool supports continuous testing (described in section 3.1.5) by allowing integration with other software, so that the created tests can be triggered as part of the build process [160]. Among its limitations, this tool runs only in Windows environments. Furthermore, it does not support all web browsers (since the Opera browser is not accounted for).

- Rational Functional Tester is a tool developed by the Rational Software division of the IBM company [144]. As the name implies, it focuses on automated functional testing, but it can also automate regression testing (explained in section 3.1.5). It supports well the capture and replay method of GUI testing, and goes beyond that by providing visual application screenshots to be edited for representing tests in a storyboard format [144]. It also supports both the Java language and the Visual Basic .NET language for creating, editing, and executing script-based tests. However, as a drawback, support for model-based test automation seem currently lacking.

- Tosca is an automation tool by the Austrian software company Tricentis. Model-based GUI testing is one core feature of this tool, that reduces test maintenance cost to a minimum. The Tosca Recorder supports the capture and replay method, even in multiple windows [151]. Moreover, this tool allows continuous integration with various software options that include even Selenium and SoapUI, but it should be considered that sometimes, integration with third party components may be challenging. Also, it only runs under Windows.

Even though GUI testing tools can provide assistance for all GUI test automation methods, the focus still remains on functional tests. Visual verification of the look and feel of the GUI is a totally different issue that brings its own set of challenges, such as recognition of a visual problem in the same way a human tester would. Some software solutions specialise in solving these issues such as Applitools [161] and Sikuli [162]. The first of them, Applitools, relies on artificial intelligence for emulating the human eye and brain in order to process complex and dynamic pages, and detect layout issues [161]. Some tools created through academic research raise the bar even higher by going beyond checking for visual problems and into validating that the design is aesthetically pleasant, such as QUESTIM (Quality Estimator Tool using Metrics) [163].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

4.1.4 Review of Usability Evaluation Tools

Over the years, various usability automation tools have been produced. Table 4.2 offers a list of some of the popular tools among designers and usability experts. The focus is on tools whose development team still support their product (as of December 2018), the list is sorted in ascending order by tool name, and the column "Supported Methods" describes the usability evaluation methods best supported by the software. Then, the various automation categories associated with said methods are presented in a parenthetical statement. These categories are automated capture, analysis, critique, and planning (which were all defined in section 3.3.5). Furthermore, the list does not include third party services that completely perform the usability evaluation themselves, as this can be considered a form of outsourcing. However, services providing remote usability testing are still accounted for.

Among the first tools to specialize in automating usability is Morae [178], it's also among the very few that assists in testing desktop applications, with most of the market focusing on web applications and mobile apps. However, it's significantly more expensive compared to its competition. Other alternatives include UserTesting [109], TryMyUI [174], and WhatUsersDo [176] which all specialize in affordable remote unmoderated usability testing for web applications. On the other hand, the tool LookBack [168] supports both moderated and unmoderated testing, but only for mobile devices.

Based on the background information from the tools presented in table 4.2, it could be observed that commercial usability tools focus on usability evaluation features relating to testing methods and inquiry methods (which were described in details in section 3.3.3 along with other evaluation classes). Tool support for these methods range from automated planning (such as Naview [172]) to automated critique (as for instance Appsee [164]). Furthermore, the category these tools target the most is the automated capture of usability-relevant data (since all tools were capable of at least this much).

Commercial usability tools have sort of marginalised the automation of evaluation methods around inspections, analytical modeling, and simulation. On the other hand, in the academic community, researchers and usability engineers have proven that tools could be built to automate part of the evaluation process. For instance, GLEAN is a tool demonstrating the utility of analytical modeling methods (and in particular the GOMS model) [179]. Secondly, VRUSE is a usability simulation tool developed to pinpoint problematic areas in an interface from a user's perspective [180]. Also, the InfoScent^{TM} Bloodhound Simulator is another usability simulation tool specializing in navigation analysis to discover web usability issues [18]. One final example is the USEFUL framework, which focuses on maintaining web usability through an automated inspection [17]. Compared to an inspection made by human evaluators, the framework resulted in the discovery of the majority of usability issues, and even found some violations that went unnoticed by the human testers [17].

Even though table 4.2 include an extensive list of usability tools, the automation of usability inspections might also benefit from tools that focus on one particular aspect of the interaction with the design. For instance, one usability rule of thumb is that the GUI should load fast, and have users wait as little as possible [181]. Therefore, if it's needed to check the GUI's responsiveness under different internet bandwidth connection speeds and from different geographical locations, then it's possible to rely on the tool GTmetrix, developed by GT.net [182], in order to find and analyse potential speed problems, and get recommended some viable improvements. Consequently, such tools can support the automated evaluation of one or more particular guidelines in a usability inspection. However, referring to such services as usability tools might be some kind of a stretch because the primary goal of such tools is performance testing (described in section 3.1.5).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Tool Name	Offered By	Supported Platforms	Supported Methods
Appsee [164]	appsee.com	- Mobile	- Testing methods (capture, analysis, and critique)
Attensee [165]	attensee.com	- Mobile - Web applications	Attention tracking focus: - Testing methods (capture)
CrowdSignal [166]	Automattic Company	- Desktop - Mobile - Web applications	- Inquiry methods (capture)
Helio [167]	Zurb	- Web applications	- Testing methods (capture)
LookBack [168]	lookback.io	- Mobile - Web applications	- Remote testing methods (capture)
Loop^{11} [169]	Loop11.com	- Mobile - Web applications	- Testing methods (capture and analysis)
Morae [170]	TechSmith	- Desktop: Windows - Web applications	- Testing methods (capture and analysis)
MouseStats [171]	mousestats .com	- Mobile - Web applications	- Testing methods (capture and analysis) - Inquiry methods (capture)
Naview [172]	Volkside	- Web applications	- Testing methods (planning, capture, and analysis)
Silverback [173]	Clearleft Ltd.	- Desktop: Mac	- Testing methods (capture)
TryMyUI [174]	trymyui.com	- Mobile - Web applications	- Remote testing methods (capture and analysis) - Inquiry methods (planning and capture)
Usabilla [175]	Usabilla B.V.	- Mobile - web applications	- Testing methods (capture)
UserTesting [109]	usertesting .com	- Mobile - Web applications	- remote testing methods (planning, capture, and analysis) - Inquiry methods (capture)
WhatUsersDo [176]	WhatUsersDo Ltd.	- Mobile - Web applications	- Remote testing methods (capture and analysis)
Woopra [177]	Woopra Inc.	- Mobile - Web applications	- Testing methods (planning and capture)

Table 4.2: Comparative list of active tools for usability evaluation

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.1.5 GUI Testing Tools Versus Usability Tools

Based on sections 4.1.3 and 4.1.4, it could be observed that GUI testing tools and usability evaluation tools do not share much in common apart of their interest in user interfaces. Most commercial GUI tools generally help validate software functionalities through the GUI, and the resulting tests are usually executed as part of regression testing. The automation process might focus on the capture and replay method, on a model-based approach, or on providing deeper access to the GUI for writing script-based tests (which were discussed in details in section 3.2.4). Besides, more recent GUI tools such as Applitools successfully managed to automate GUI tests checking the visual correctness of various graphical components across an interface [161].

On the other hand, usability tools currently on the market generally focus on evaluation methods that require the presence of users, and these constitute the testing method and the inquiry method. Some other tools support the evaluation of specific guidelines in a usability inspection.

Even though there seem to be no commercial usability tool currently automating analytical modeling methods, the background is similar to automation of model-based GUI testing. Both concepts analyse models and draw conclusions without the need for observing and processing user actions.

Another interesting similarity lies in script-based GUI testing and automated usability inspections. Even though the number of usability tools covering the inspection of multiple guidelines is limited, it's easy to conclude that the automation of a complete inspection would often require accessing the internal attributes of the various graphical components presented in the interface. This aspect makes the inspection process a little bit similar to script-based GUI tests, which also require the same access level. However, validating guidelines and checking functionalities are two entirely distinct processes, and the support for their automation by tools is widely different.

In other words, script-based GUI testing tools have progressed to a point where they support different scripting languages, cover different forms of user interaction, and might even allow the integration of an additional third party script-based testing tools (such as how Tosca [151] supports the integration of Selenium [4]). On the other hand, usability inspection tools currently do not support any scripting languages, and have limited capabilities, which leaves a lot of potential for future growth.

In conclusion, GUI testing tools mainly focus on checking the functionalities through the interface, while usability tools generally target capturing usability-relevant data derived from user experiences. Support for automation efforts is present more in the GUI testing field than in the usability engineering field, mostly due to the importance of the human factor in usability testing.

4.2 Evolution of Heuristics in Software Development

As previously discussed in section 2.2.4, usability heuristics are guidelines or principles describing how a system can best reach a high level of user-friendliness. Some general guidelines can be applied on any type of GUI such as the ten usability rules of thumb by Jakob Nielsen (which are also described in section 2.2.4) [8]. Other guidelines might be suited for a specific context. Thus, they cannot be applied on any GUI. Such heuristics can be, for instance, guidelines specific to touchscreen-based mobile devices, or e-learning applications designed for children [183] [184]. However, no matter the scope of the designated heuristic, it's still important to comprehend how it's first derived, how it's updated, and how its relevance gradually change over time.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.2.1 Ergonomic Roots of Usability Heuristics

Design principles precede usability engineering by far. Among the earliest design guidelines created are Vitruvius's three core design principles dating way back to the first century BC (Before Christ) [185]. His work later influenced various historical figures such as Leonardo da Vinci (who got inspired into drawing the now famous Vitruvian Man) [186]. These three design principles are briefly explained below [187]:

- Firmitas: The design's strength and durability, including its reliability, stability and robustness.
- Utilitas: The convenience and suitability of the design for the needs of its intended users, or in other sense, the design's efficiency and effectiveness.
- Venustas: The perceived beauty of the design.

Design guidelines gained more importance starting world war one and two, with the development of complex new machines and weaponry. The need for lowering the time and cost of training personnel in the use of complex machinery, as well the frequent necessity of replacing said personnel, led to more research and development in the field called Human Factors and Ergonomics (HF&E) [188]. This discipline is concerned with comprehending the data, principles and methods associated with the interaction between humans and other system elements, and the practice of designing systems to optimize human well-being and proper system performance [189]. Usability engineering is a descendent of HF&E, inheriting many of its evaluation techniques such as the cognitive walkthrough, the thinking aloud protocol, and the use of questionnaires (described in section 3.3.3) [190].

With the dawn of information technologies, and the emergence of the first personal computers, usability gradually became a key goal in GUI design so that computers could be used by average users rather than only trained technical specialist. In 1990, authors Jakob Nielsen and Rolf Molich publish their seminal paper titled "Heuristic Evaluation of User Interfaces", in which they present the heuristic evaluation method as a discount usability solution [191].

Consequently, more research was funded for deriving more refined usability guidelines to better suit the context of use. These studies have produced numerous influential heuristics over the years. Some of which are listed below:

- The 944 guidelines for designing user interface software by Smith and Mosier [26].
- The official user experience guidelines for Windows-based desktop applications by Microsoft [91].
- The iOS human interface guidelines by the Apple company [92].
- The Android user interface guidelines by Google LLC [93].
- The user interface guidelines for Windows Mobile by Microsoft [94].
- The 113 guidelines for homepage usability by Jakob Nielsen [192].
- The 247 web usability guidelines by David Travis [193].

Usability heuristics have been developed through both academic and commercial research. Some principles describe general preferences that can be applied on any user interface. While other guidelines deal with a certain environment, or focus on a particular context of use such as a specific mobile device, or the age and background of the targeted users.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.2.2 Traditional Versus Modern Heuristics

Among the many guidelines produced over time, some differences began to surface between the older more traditional heuristics and the newer more recent ones. In general, the more software and technology progresses, the more context-specific the published heuristics become. For instance, one traditional design principle by Smith and Mosier, dating back to 1986, is to warn users of potential data loss (by displaying an explicit warning message to prompt appropriate user action) [26]. Despite this rule being intended for WIMP applications, it's general enough to still be currently relevant for any type of GUI across any platform. Nowadays, even the GUI of a mobile game is expected to ask the user if he's sure before deleting his save file. On the other hand, more recently deduced heuristics mainly have the goal of covering additional areas untouched by previous work such as researching the GUI of mobile devices, of tablets, or of augmented virtual reality headsets [194]. In other cases, the context of the interaction does not only change on the machine side, but also, on the human side. This means that the guidelines can also differ along with the age of users, or their technical expertise [195].

By way of deduction, the continued emergence of new information technologies keeps introducing new interaction possibilities which brings forth sometimes different usability challenges. Therefore, the relevance of some traditional usability heuristics might be put into question the more the interaction between the user and the design is evolving. In order to validate older usability guidelines or derive new ones, it's possible to follow a methodology consisting of the following six steps [196]:

1. Gather literature related to the topic of research, and explore applications specific to that topic.
2. Describe the most important common characteristics and concepts from the previously gathered data.
3. Relate the identified topic characteristics to the best possible usability heuristics, based on case studies analysis and previous guidelines.
4. Formally specify and explain the proposed usability heuristics in the form of a standard template.
5. Validate the new guidelines through additional experiments performed on relevant case studies.
6. Refine and improve the list of heuristics based on the feedback gained from previous validation efforts.

The aforementioned methodology is effective in identifying which traditional heuristics didn't age well, but this is not the only method possible. A more informal approach would be to examine some illustrative examples and conclude whether applying the guideline can still result in an improvement, or whether a better option is available. Or simpler yet, it's possible to examine literature and deduce if a certain traditional guideline is outdated by a more modern one.

For instance, according to David Travis, one usability guideline specific to the search function of a website, is to allow the user to configure the number of search results to appear per page [193]. On the other hand, Jakob Nielsen states that offering a single default number of search results per page is usually better than letting the user choose how many they want to see [197]. However, he still believes that pagination in general is well suited for long listings including photo galleries [197].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The figure 4.1 shows a partial screenshot of the bottom area of a common search result page retrieved from the "Ebay" website. The image contains the pagination options and a drawn red arrow bringing attention to a dropdown menu with a fixed default value representing the number of displayed results. Based on figure 4.1, it can be noted that the Ebay website is following Travis's guideline, and not taking Nielsen's advice.

Due to the constant improvement of software and technology, pagination options have already evolved beyond the need for restrictions on the number of search results. Infinite scrolling is a pagination technique allowing users to continuously scroll down for retrieving and displaying more content [198]. The page would load more results and grow longer the more the user scrolls down. All the while, no pagination section is displayed anywhere on the screen.

Comparing the traditional pagination from figure 4.1 with the infinite scrolling technique shows that the first method would require at least two user actions before loading more results, which are a scroll down and a click on the page number. On the other hand, the second method only requires one user action which is scrolling down (and then, the results are loaded dynamically). Therefore, since the infinite scroll requires less steps, it's less cognitively complex, and by extension, easier for users to perform. Furthermore, studies by Van Deursen and Van Dijk [199], indicate that 91% of users performing a search do not go beyond the first page of search results, which means that they don't click on anything offered in the pagination area [199]. This means that continuously scrolling down might currently be user-friendlier than traditional techniques. However, additional research and user studies should examine and compare both options to draw a more definitive conclusion and refine all usability heuristics advising designers on how to offer search results.

Moreover, it should also be reminded that the infinite scrolling technique is directly contradicting another web usability heuristic by Travis stating that a site should require minimal scrolling and clicking. Therefore, that particular guideline should be revised as well.

Another example is included in the web usability guidelines for smartphones, where it was advised to display navigational components only on the homepage and avoid repeating them on other pages [200]. This modern guideline (dating from 2011) directly contradicts with a traditional one (dating from 1999), which was stated by Jakob Nielsen in his book "Designing web usability: The practice of simplicity" where he encouraged web designers not to consistently display the navigation on all the web pages, and to preferably highlight the specific navigational component indicating the user's location in the site map. [77]. Even though the mentioned modern guideline states the opposite of the traditional one in the context of smartphones use, Nielsen's guideline is still applicable for desktop applications, which is a different context.

Based on the previous examples, it can be deduced that the relevance of traditional usability heuristics is subject to change, and therefore needs to be regularly updated to take into account the potential brought by more modern design options or newer contexts of use. However, in some situations, even two modern usability guidelines might contradict each other [201]. For instance, one heuristic might state that similar menu options should be grouped together, while a second heuristic claims that a menu should not have more than five elements. Theoretically speaking, if more than five menu options are similar, the two aforementioned heuristics would contradict each other. Therefore, it's important for usability experts to take into account the context of use before selecting which heuristics to include in a usability inspection, and it's important for GUI designers to choose well which guidelines to prioritise in their work.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Figure 4.1: Screenshot of a pagination area under search results retrieved from the Ebay website

4.3 Value and Potential of GUI Event Sequencing in automating Heuristic-based Usability Inspection

Since usability inspections do not require the presence of a user sample, the cost of a usability heuristic evaluation is among the cheapest alternatives. Consequently, a successful automated inspection would be even cheaper than the traditional approach, and would greatly contribute to simplifying an iterative design process.

Moreover, based on the review of usability tools, discussed in section 4.1.4, it was observed that automation efforts focus the most on capturing usability relevant data collected from actual users. On the other hand, usability inspections do not currently have notable tool support despite the efforts of a few academic studies in changing that [13] [15] [16] [17] [18]. Therefore, it can be theorized that automating the detection of more guideline violations would close the gap more with manual heuristic evaluation. To achieve that, the use of GUI event sequencing is considered.

A sequence of GUI events represents an ordered chain of related events that follow each other such as method calls and data inputs. Their goal is to emulate a usage scenario during an automated GUI test [64]. Script-based GUI testing requires support for sequencing GUI events in order to properly check if the different graphical components are behaving correctly while a task is performed on the GUI.

In this thesis, it is hypothesized that GUI event sequencing can improve the automation of a heuristic-based usability inspection. Despite being a GUI testing technique, GUI event sequencing can be considered for automating the inspection of some heuristics that involve following an ordered series of events, such as filling a form. However, guidelines evolve over time and might change with the context. Therefore, before checking the potential extent of event sequencing in a usability inspection, it's important to first select a stable set of heuristics, and specify a particular context of use.

Since desktop applications have a longer history than their web and mobile counterparts, research in design guidelines for WIMP applications started first, and continues to this day. Furthermore, Microsoft Windows is the most widely used operating system on desktop devices (holding 90.77% of the market share as of October 2017 [202]). Therefore, for testing the correctness of the hypothesis that an event-driven approach can be applied in heuristic evaluation in order to automate it, the considered context of use is restricted to Windows desktop applications, and the selected set of heuristics would be derived from the 944 design guidelines for user interface software by Smith and Mosier [26]. Under these circumstances, the study would provide a minimalistic and clear set of processed heuristics and indicate how many guidelines can be automated with event sequencing, how many guidelines can be automated through other means, and finally, how many of them should only be checked manually.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5 Proof of Concept for Automating Heuristic-based Usability Inspection with GUI Event Sequencing

To demonstrate that automation in the specific field of usability inspections can benefit from a technique usually reserved for script-based GUI testing, called GUI event sequencing, it's first needed to derive a minimalistic and relevant set of heuristics from a sizeable and widely accepted list of guidelines. Then, the practical automation of an arranged sample of those heuristics has to be verified in a tool selected particularly for the purpose of testing the validity of the claim that GUI event sequencing improves automation efforts of a heuristic evaluation.

5.1 Summary of the Problem Addressed

Heuristic evaluation, being a cheap usability inspection method, is generally performed manually by a usability expert (and is described more in section 3.3.3). The list of guidelines inspected in an evaluation varies in their amount, their focus, and even their durability, depending on the context of use, and the state of the art in information technology at the time in which the guidelines were derived (as explained in section 4.2). In order to facilitate analysing the automation of these guidelines, it is first required to derive a reasonable set of heuristics, classify them according to the feasibility of their automation, and their compatibility with GUI event sequencing. Consequently, a testing tool and a couple of software applications that can run under Windows are selected, to check if a sample of the previously gathered guidelines can be automated in a practical experiment.

5.2 Deriving and Structuring adequate Usability Heuristics

In an effort to grasp a minimalistic set of precise heuristics, the context of use is restricted to the usability of Windows desktop applications. Deriving such a set is possible by starting from an existing source of traditional design principles. One of the best candidates is the list of design guidelines for user interfaces by Smith and Mosier, sponsored by the Electronic System Division (ESD) of the United States Air Force [26]. The list includes 944 guidelines and their objective was to promote efficiency and learnability by minimizing the memory load required from the user, as well as supporting users with different skill levels. This makes their list one of the largest sources of traditional design heuristics. Moreover, even though the report dates back to 1986, the durability of these guidelines is praiseworthy [203]. In a 2005 study by Jakob Nielsen, where he arbitrarily selected ten guidelines from each one of the six sections (by randomly choosing six pages in the document, each belonging to a different section of the list), he concluded that 54 out of the 60 guidelines he reviewed were still valid [204]. However, the results might have greatly changed had he chosen other guidelines (or in other words, if he had randomly picked another page in the document).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Therefore, the 944 design guidelines for user interfaces by Smith and Mosier are selected as the starting point for deriving a minimalistic and relevant set of usability heuristics. This list covers six different functional areas of user-system interaction briefly described in the following [26]:

1. Data Entry: The data entry section has 199 guidelines, all describing how to assist user actions involving input of data in the system, and how the system should respond to such input. (Also, all their reference codes in the document begin with "1.").

2. Data Display: The data display section holds 298 guidelines. It explains how to output data to the user, including what additional information to include, or exclude from such output. (All the reference numbers of these guidelines start with "2.").

3. Sequence Control: The section for sequence control include 184 guidelines. Sequence control refers to the logic behind the transition from one user-system transaction to the next (with a transaction referring to the smallest functional unit of user-system interaction, meaning any user action which is followed by a response from the system). This part covers both user and system actions that start, interrupt, or terminate user-system transactions. (The reference codes of these guidelines always begin with "3.").

4. User Guidance: The user guidance part has 110 guidelines. This section describes how to present data such as error messages, prompts, labels, and even formal instructions in the help material. (Additionally, their references start with "4.")

5. Data Transmission: The list of data transmission guidelines holds 83 principles. They cover email communication, and discuss how to present user interfaces sending and receiving data to and from other users. (These guidelines are referenced with codes that start with "5.")

6. Data Protection: The data protection section include 70 guidelines. This part focuses on the security of the information by advising on how to protect it from destructive user actions, system failures, and unauthorised access. (Also, the prefix "6." precedes the reference number for each of its guidelines)

The report was originally developed for user interfaces of mainframe computers, and each single guideline has its own unique reference number. However, according to Smith and Mosier, not all of their proposed guidelines can be applied when designing any particular software [26]. In other words, some of the guidelines would be relevant while some would not, for any specific application. In fact, through one of their own surveys, they discovered that designers found about 40% of the guidelines applicable in their context of use [26]. Therefore, it's necessary to select currently relevant guidelines for Windows desktop applications, and exclude guidelines which are either too context-specific, or became intuitive for GUI designers, as well as those which turned obsolete.

5.2.1 Filtering Usability Heuristics for Windows Applications

The process behind filtering which heuristics to remove from consideration is based on the following criteria:

Redundancy: The original guidelines have some necessary overlap in their coverage from one section to another, because every section was intended to include a complete list [26]. Consequently, almost identical guidelines would be offered in multiple sections, or even different subsections of the same functional area. For instance, the guideline referenced in their paper as 1.3/32,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

titled "Confirming Actions in DELETE Mode" focuses on the same subject and gives the same advice as the guideline with reference 6.0/18, called "User Confirmation of Destructive Actions". It is therefore sufficient to include such guidelines only once in the classification proposed in this paper.

Inapplicability on a desktop application's GUI: A large group of the offered heuristics do not apply to a modern GUI setting in a regular Windows application. It is therefore necessary to exclude guidelines such as those describing appropriate commands in a command line application, or ensuring better user-system interaction in a voice-based application.

Presently being a common practice: Information technology progressed so far that many heuristics from the report are now endorsed by default, because Windows operating systems currently support users much better than mainframe computers used to do in the 1980s. For example, the guideline referenced 3.1.3/6 "Dual Activation for Pointing" describes the predecessor of what is now commonly known as the double-click. It is not needed to include such guidelines as they are automatically covered by modern operating systems. Besides, some other guidelines became widely accepted over time, and are currently intuitive for GUI designers. An example would be the guideline coded 1.4/7 "Protected Labels", as it is not necessary anymore to advise designers that labels cannot be edited by users. Therefore, it is not needed to include such guidelines in future classifications.

Having a highly specific context: The list has a high number of guidelines designed for various specific contexts of use. For instance, the report offers 19 guidelines for drawing, 12 guidelines for flowcharts alone, and 18 guidelines on how to display maps. In order to minimize the set of heuristics, only general guidelines that can be applied in most Windows applications are considered. The same logic applies for rules that are security-oriented, whether they cover simple data validation or complex encryption of messages, they are not the main focus of usability (and were probably included in the report due to the importance of security and secrecy in software systems designed for military use). Such guidelines are also omitted from the classification.

Turning obsolete: Heuristics that became obsolete over time are naturally dropped from consideration. For instance, the guideline with reference 1.0/9 "Explicit ENTER Action" states that a user is always required to take an explicit ENTER action to start processing entered data, and to not initiate processing as a side effect of some other action. This is no longer correct. For example, during an initial user registration task, the system can check if the chosen username is available before the user submits a registration request. The same applies for the auto-complete feature, that activates as the user is typing text. Another example would be providing search suggestions while the user is typing his search query. Therefore, the automation of any guideline no longer valid is not needed.

Retaining the most relevant guidelines and excluding the rest reduced the number of these rules from 944 to 85. This result is evaluated in details in section 5.4.1. The accepted and classified heuristics are described in sections 5.2.2, 5.2.3, as well as 5.2.4, with each element being followed by an explanation of the reasoning behind its categorisation.

5.2.2 Listing Heuristics Compatible with Automation by Event Sequencing

The following list consists of heuristics, relevant in the general context of Windows desktop applications, which can be verified automatically with the help of GUI event sequencing, as applied in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

script-based GUI testing. Each heuristic starts with its reference number in the report developed by Smith and Mosier, followed by either a partial or complete description of said guideline, then the reason it is believed that such guidelines belong in this category. All of the tests checking these heuristics share in common the need for a user to interact with the GUI in order to arrive at the GUI state to be assessed. In other words, a script-based GUI test would need to re-enact a certain usage scenario so it could verify compliance with the guideline. Such hypothetical usage scenario would be reconstructed automatically by triggering an ordered series of GUI events from a test script. These guidelines are the following [26]:

1.0/1 Data Entered Only Once: It is recommended to ensure that a user would need to enter into a GUI any particular data only once, and that the system can access that entered data afterwards for the same task or for other tasks [26].

Provided that the task in question is compatible with the guideline, it is possible to verify the validity of the recommendation by creating a sequence of events that mimics user behaviour. Said user can perform one task, and enter data needed. Then, after finishing his first task, and initiating a second one, the user shall not need to re-enter the same data from the previous task again if they are required to perform the second task. A sequence of GUI events could be used to simulate the tasks in an ordered fashion, and verify if the guideline was respected. For instance, if the first part of a task is to search for items, and the second is to filter items, the user should not be expected to re-type the search query for the second part of the task to succeed. A more natural example, would be entering a username and password as part of signing-up in an application. Then, immediately after confirming the registration, the user is asked to enter the same data again for a login. In order to not violate the guideline, it can be possible during the last step in confirming the sign-up process to also sign-in the user (since that's his reason for signing-up anyway). Were these two examples not hypothetical, it would have been feasible to automate a test verifying compliance with the heuristic by calling the sequence of GUI events needed for performing the previously described scenarios, and all the while, checking if the same input was entered twice. In other words, the steps needed for the user to perform these tasks on the GUI can be mimicked by a test script that would trigger the GUI elements on the screen in the same order and the same manner as the user.

1.0/4 Fast Response: The system should acknowledge data entry actions rapidly, so that users are not slowed by delays in computer response. For normal operations, displayed feedback delays should not surpass 0.2 seconds [26].

The verification of this guideline can be automated by using timers. For instance, when the GUI event in question is interacted with, then the timer starts measuring the time needed for the system to respond. Once the system response state is detected, or GUI elements associated with the response appear on the GUI, then the timer would stop. However, the time delay limit should not always be 0.2 seconds [119]. Depending on the nature of the operation measured, the delay allowed can vary. It can be pushed up to one second for operations that should not interrupt the user's flow of thought, and a limit of ten seconds for interactions that should keep the attention of the user [205].

1.0/11 Explicit CANCEL Action: A user should perform an explicit action in order to cancel a data entry. Cancellation should not be the side effect of some other action [26].

A test can check if an interruption in a data entry sequence (by opening the help window for instance), would lead to completely or partially losing the data entered, or whether the user can continue his operations where he stopped. The automation of such a test requires reliance on

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a sequence of GUI events that re-enacts a scenario challenging the GUI's compliance with the guideline.

1.0/12 Feedback for Completion of Data Entry: If a data entry transaction was successful, the system should acknowledge the completion of data entry with, for instance, a confirmation message. If data entry was unsuccessful, an error message should inform the user [26].

This recommendation could be verified with sequencing by detecting a confirmation message (or an error message) displayed as a result of some successful (or failed) user action, or by checking if a text notification on the GUI affirms to the user the state of the ongoing operation. The automation of such tests would have to start a sequence of GUI events leading to either a successful operation or an erroneous one (depending on the type of feedback tested).

1.0/16 Partitioning Long Data Items: If long data must be entered, that data should be partitioned into shorter groups to simplify its entry and its display [26].

This principle can still apply when designing input fields for entering long software license keys, or an International Bank Account Number (IBAN). One way of verifying this guideline can be by using event sequencing to test if the long data is partitioned on multiple text fields, and accepted by the system as one input. However, if this long data has only one corresponding input field. The test script should first fill in that field, then check if the entered data is formatted correctly by the GUI. In other words, the test would check if the format divides the data into smaller groups. No matter the way this guideline is tested, the test script would need to emulate at least one user-system interaction (which is data input). Thus, GUI event sequencing is needed.

1.0/28 Decimal Point Optional: It should be allowed to either include or dismiss a decimal point at the end of an integer value, and treat them as equivalent alternatives [26].

With event sequencing, a written automated test could check whether decimal points are accounted for in the system, by input of various numeric forms, sending the data to the system, and checking how the systems deals with such data.

1.3/10 Upper and Lower Case Equivalent in Search: Unless stated otherwise by a user, upper and lower case letters should be treated as equivalent when searching data items [26].

This guideline can be checked automatically through input of one search query multiple times, while in each iteration, different letters are in upper and lower case form. Finally, the test checks the similarity of search results between iterations.

1.3/11 Specifying Case in Search: If differentiating upper and lower case letters is important for users, it should be allowed to specify the case on the GUI as a selectable option when a search operation is performed [26].

If differentiating lower and upper case letters during search is relevant in a certain context of use, then this option should be provided on the GUI. Evaluating if it works can be automated in a GUI test.

1.3/24 Storing Frequently Used Text: Users should be allowed to store frequently used text segments, and then in a later use, identify and recall the stored data segments [26].

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

This guideline can also be verified by a test script. For example, in a login action, sequencing can test if the system can remember the name previously entered by the user on the second login (for example, by displaying it as a default text), as well as checking if the password can be optionally saved for repeated use. Furthermore, the test can check whether the login works fine with the stored username and password. In this scenario, since the login task consists of multiple steps (input of data and submitting it), testing it can be done through event sequencing.

1.4/2 Flexible Interrupt: When various items are entered as one single transaction, such as in form filling, the user should be allowed to review, cancel, save, and change any particular item before finalising his data entry action [26].

Other than in form filling (which require event sequencing to be tested), an instance of an interface abiding by this recommendation, would be a well-designed GUI for a software installation task. The window would have "Next" and "Back" buttons for advancing with the task, or reviewing previous choices. It could also have a "Cancel" button to invalidate the process. No matter the series of actions taken, a test script evaluating compliance with this heuristic needs to simulate user actions by starting a corresponding sequence of GUI events.

1.4/15 Explicit Tabbing to Data Fields: Users should take explicit tabbing action in order to move from one data entry field to the next. The computer should not provide such tabbing navigation automatically without the user requesting it [26].

The process can be tested automatically on a GUI, for instance, by filling a form (with multiple fields) in an automated manner, while triggering the tab key, and checking the location of the cursor. The focus of this principle is not only related to the presence or absence of tabbing navigation, but also on checking if the expected key action controls it. Naturally, since the scenario contains a sequence of user and system actions, the test script would also need to rely on a GUI sequence to simulate the scenario.

1.6/5 Zooming for Precise Positioning: If data entry depends on the exact placement of some graphic elements on the GUI, users should be allowed to expend the part of the GUI critical of the task. Enlarging the display area would simplify the positioning task [26].

This guideline is oriented toward the way GUI elements are displayed, and answers questions such as whether the window can be resized, or certain graphics can be made bigger. For testing this recommendation, the user action of resizing or expending a GUI element has to be simulated in the test, and the properties of the graphical element has to checked before and after the decisive user action. Therefore, a GUI sequence is required for the automated evaluation.

1.7/3 Non-Disruptive Error Messages: When data validation identifies a possible error, an error message should be displayed to the user at the completion of data entry. The ongoing transaction should not be interrupted by error messages [26].

A test could wait for the appearance of an error message event after an erroneous data is entered. If the error message is displayed before the completion of the task, the test would fail since the guideline is then violated. Otherwise, if the user is only notified of the error once he finishes data entry (without being interrupted), then the test would pass. Either way, a GUI sequence is needed for automating the test.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

1.7/7 Optional Item-by-Item Validation: In the case of novice users, an optional item by item data validation could be provided, and their tasks could be divided into multiple entry transactions. This option might improve the learnability for novice users who are not confident about the requirements associated with each data item. On the other hand, such an item-by-item validation process would slow down experienced users. Therefore, making this capability optional would help beginners without getting in the way of experienced users [26].

This guideline is highly compatible with automated testing through GUI event sequencing. The incremental validation of data can be accompanied by corresponding series of tests checking every single validation step.

1.8/9 User Review of Prior Entries: If data entered in one transaction are used in another one, the system should retrieve and display any relevant information so that users can review it, instead of requiring the user to remember those data [26].

In such a scenario, the sequence of events that would help automate the guideline, would start by the input of data, followed by submitting it to the system, and then finally checking if the entered data is displayed in the GUI for review, during the next task.

1.8/10 Automatic Entry of Redundant Data: When accessible data is logically related to previous data entries, the system should be able to retrieve other relevant data without needing the user to repeatability ask for each data attribute separately. As a negative example, a user should not be required to enter both an item's name and its identification number if both attributes are unique in the system. Only entering one of them should be enough to display both of them [26].

Entering multiple data means activating multiple GUI events. Therefore automated testing of the process requires GUI event sequencing, as well as prior understanding of how data items logically relate to each other.

2.3/5 Items Paired for Direct Comparison: When a pair of data items have to be compared, the GUI should display one directly next to the other. It is important to note that users will not always be accurate during comparisons. Therefore, providing an automated analysis would better assist users [26].

Testing if two particular items can be compared and whether their differences are highlighted in some manner to the user, can be performed automatically. The event sequence would start with the selection of the two items under test, initiating the comparison, then checking if different elements in both items are marked in a particular manner to attract user attention.

2.5/4 Paging Crowded Displays: If a GUI displays a large amount of data items to the point that they cannot be presented in a single frame, then those data items should be partitioned into separately displayable pages. Furthermore, a convenient navigation procedure should be provided to enable users to easily move from one page to another [26].

Pagination requires by definition dividing a large data set on multiple pages, presented to the user one page at the time. No matter the type of pagination used (traditional or dynamic, as described in section 4.2.2), the list of data displayed has to be scrolled down if necessary, and the GUI event responsible for calling new pages has to be periodically activated multiple times to ensure its correct behaviour (which is to display the next content page). Therefore, automating this test can be done with event sequencing.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

2.6/2 Removing Highlighting: When highlighting is employed to emphasize on the importance of certain display elements, such highlighting should be dismissed the moment it is no longer required [26].

If for an instance an error is highlighted on the GUI, the highlighting should be removed once the error is corrected. To check this guideline, an automated test can first perform actions to cause the error to be highlighted, then fix the issue and check if the graphical element is still highlighted. If it is no longer attracting the attention of the user, then the test passes.

2.7.3/9 Prediction Display: In order to assist users in understanding and effectively responding to complex data changes, it is advised to consider, when appropriate, displaying predicted upcoming data states based on an automated analysis of certain models representing data dynamics. Depending on the context of use, some examples would be to display the predicted flight path of an airplane, or to display expected market fluctuation [26].

Checking whether the prediction function works correctly should not be done manually. Verifying the correct implementation of this recommendation is done by comparing the actual measured prediction values with the expected values for that particular test setting. As a first step, the GUI event responsible for sending the relevant data to the system is called, then the GUI event responsible for displaying the prediction results listens for the system response and catches it, so the data can be compared with the expected data.

2.7.5/3 User-Specified Windows: When it cannot be known in advance that many different types of data would need to viewed jointly at the same time, then the user should be allowed to specify and choose separate data windows or perspectives that would have to be shared on a single display [26].

An automated test can check if different types of data can be displayed in separate window elements simultaneously. As a result, in the simplest test possible, at least two GUI events need to be taken into account at the same time, which means that a sequence of events is required.

3.0/1 Flexible Sequence Control: The sequence of needed user actions should be flexible. A user should have some freedom when performing transactions related to data entry and display, or he can ask for guidance associated with any transaction he's engaged in [26].

Testing different transaction sequences automatically requires using GUI event sequencing, with each transaction having one or more corresponding GUI events that are associated with it.

3.0/20 Indicating Control Lockout: When the data entry must be delayed in order to give the system time to process prior entries, then the GUI should indicate that processing delay to the user [26].

Considering, for instance, a submit button action that demands more than one second to result in a system response, and that this user action should not be repeated until the first operation is handled. A possible sequence to test this can check if the button is briefly disabled until the result of the last submit action is displayed (or processed). A second example would be to display a progress bar while the system is processing a complex operation, and in the meantime, disable user interaction with the GUI until the progress bar indicates the completion of the task. Either way, a sequence of GUI events is needed for automating these tests.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1.3/7 Menu Selection by Keyed Entry: It is recommended to allowing users to accomplish menu selection by keyed entry. For example, simultaneously pressing two keys would select a menu option faster, and better assist experienced users [26].

Checking this recommendation can be automated by simulating pressing the shortcut keys associated with a menu option, and checking whether the corresponding result (or feedback) of that action appeared on the GUI. Even though simulating key entries in a test is not dependent on GUI elements, the corresponding feedback is. Therefore, event sequencing techniques are needed to wait for and catch graphical elements associated with the result of key presses, then access the properties of those graphical elements to check if they indeed conform with what is expected.

3.2/10 Only Available Options Offered: Users should only be provided options that are actually applicable and relevant for their current task [26].

In this case, an automated test can verify if various options are disabled (or even absent) when the user is in a context that does not allow him to perform those commands, then, when the context changes, the availability of such options are examined again, in accordance with the requirements. These scenarios all require performing a sequence of user actions. Therefore, GUI event sequencing can help automate the tests checking compliance with this guideline.

3.3/1 User Interruption of Transaction: It should be allowed for a user to interrupt a current transaction, in the manner appropriate for that task's requirements [26].

Since the task in question involves having multiple steps within one transaction, and a user action interrupting that transaction. This scenario can be tested by relying on event sequencing.

3.3/3 CANCEL Option: When appropriate to sequence control, the option to cancel a task should be provided to users. This will have the effect of invalidating any changes that were made and restoring the current display to its previous state [26].

As the task under test involves a sequence of operation (starting, then cancelling), it can of course be tested with GUI event sequencing.

3.3/4 BACKUP Option: When applicable and appropriate, the GUI should provide the option to persist in the system all data states related to the work performed by the user (without terminating the transaction or closing the application). The backup action allows returning to the display associated with the backed-up data transaction [26].

Just like the two previous guidelines, this recommendation is about performing a series of operations, and can therefore be tested automatically by relying on GUI event sequencing.

3.3/6 RESTART Option: If possible, the option to restart a transaction should be provided. This will first result in cancelling any entries made in the current transaction sequence. Afterwards, the GUI would return to the beginning of the sequence. If restarting would cause data loss, then users should confirm the restart action [26].

This task can be tested with event sequencing due to it consisting of multiple user actions. It is similar to the previously described guidelines.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

3.3/7 END Option: When permitted, the GUI should provide the option to end a transaction, and most notably, when said transaction is repeatedly performed as part of a loop. The effect of the ending action should be the concluding of the current transaction sequence [26].

It is possible to test if an ongoing task can be ended through the GUI. The test would need to check the GUI before and after ending the task. Therefore, a sequence of GUI events is needed in the test.

3.3/8 PAUSE and CONTINUE Options: If sequence control allows it, the GUI should provide options for pausing and continuing a transaction. This would have the result of first interrupting and then later resuming a transaction sequence. The pause and continue actions should not interfere or modify the data entries or control logic associated with the interrupted transaction [26].

If it's appropriate to pause a certain GUI task, the automated GUI test checking if it can be done would have to rely on event sequencing.

3.3/10 SUSPEND Option: If appropriate, the GUI should provide users with the means to suspend their work on a task. The result of suspending a task would be the preservation of the current transaction status when the user exits the system. Then, when the user returns to the system at a later time, he is permitted to resume the suspended task [26].

An automated test can check the GUI before preserving relevant transaction states, trigger the suspend action, then restart the application, and check if the task can be resumed where it was suspended.

3.5/10 UNDO to Reverse Control Actions: The immediate reversal of user actions should be possible by invoking the undo command [26].

A sequence of events can simulate the reverse control action after the completion of a preceding task (provided the task is reversible according to the requirements), and then check if the behaviour of the GUI is as expected.

3.5/11 Preventing Data Loss at LOG-OFF: Before a user logs off from the application, the system should check pending transactions and if any is found, the GUI should inform the user with an advisory message that a task was not yet completed, or that unsaved data will be lost [26].

With event sequencing, a pending transaction can be simulated, then as a logout action is requested, the test would check if a message appears, notifying the user of possible data loss and asking for confirmation before proceeding with the logout.

3.5/12 Immediate Data Correction: When a data entry transaction has been completed by the user but some errors in the data were identified, the GUI should allow users to immediately correct the information directly [26].

Due to the nature of the task, event sequencing is suitable for checking this heuristic. A possible sequence would have the test first complete an erroneous transaction, receive the error message, then proceed by correcting the mistake, followed by committing the transaction a second time, then finally checking the successful execution of the action.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

4.0/24 Flexible User Guidance: If techniques or design choices associated with user guidance might slow down experienced users, the GUI should present alternative paths or modes that would allow users to by-pass default guidance procedures [26].

The behaviour of skilled users cannot be predicted with confidence. However, if a shorter user action path is offered for experienced user, then it can be represented in a sequence of GUI events and tested.

4.1/1 Indicating Status: When the status of the system is relevant to users, it should be indicated on the GUI at all times [26].

Checking the status with automation would require having a GUI event listening for changes in the status, while other GUI events are triggering different modifications to the system status. The tests are successful if the GUI event listening, can correctly detect all the changes inflicted on the status.

4.3/8 Multiple Error Messages: If multiple errors are identified inside one combined data entry, then the GUI should notify the user without displaying together or consequently complete error messages for every single error [26].

This can be tested by checking how many error messages are prompted by the system, when many errors occurred. If multiple messages are identified, then the test fails. However, if only one error message is returned, it should be the first error, and it could hint to the number of additional errors found while omitting the details (by including for instance the wording "and 4 more errors" at the end of the first error message, instead of displaying 4 additional messages boxes). In this context, detecting the appearance of only one error message would be enough to make an automated test pass. Additionally, it is also possible for the test to check if the message content shows the number of additional errors whose details were omitted.

4.3/13 Cursor Placement Following Error: After an error message is prompted and handled by the user, the cursor focus should be positioned at the GUI element that resulted in an error [26].

Since a series of user and system actions is needed for checking this guideline, it is possible to use event sequencing to test if the cursor is located in the first erroneous field after dismissing an error message.

4.3/15 User Editing of Entry Errors: Once an error is identified and the user is asked to correct it, he should only have to fix the data portion that caused the error, and should not have to repeat an entire task [26].

Taking for instance the context of form filling, a possible sequence of events testing this guideline can start by completing data entry with just one erroneous input. After the appearance and dismissal of the error message, the test can correct the problematic input without touching the other data fields, and then finally submit the data a second time. If the task is completed successfully, the test would pass. However, if the GUI requires re-entering more than just the corrected data, then the test would fail, as this constitutes a violation of the guideline.

4.4/22 Record of Past Transactions: It should be permitted for users to ask for records of previous transactions to be displayed. This would help them review past actions [26].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

A first sequence of GUI events could represent the past transactions, and a second sequence of GUI events could check if data related to previous operations could be displayed. For instance, showing a list of last opened files means this recommendation is being followed by the GUI.

5.2/5 Aids for Directory Search: When users are searching for an address in a directory, the system should assist them by allowing the search with partial names [26].

From a contemporary angle, this guideline does not have to only apply to an address directory. Any search operation would be more user-friendly if search queries can handle a partial name. The sequence needed to test this guideline would start with initiating the search action by specifying part of an address, and then examining the search results.

5.2/14 Automatic Address Checking: The system should validate an email address by checking its content and format, and users should be notified of possible mistakes before sending a message [26].

In this particular data transmission context, the moment the cursor leaves the address field, the computer can start checking the email. The attention of the user should be brought back to that field as long as the address format is invalid. This described system behaviour can be tested with event sequencing.

6.0/5 Protection from Interrupts: If it's noted that a user action would stop the current user transaction sequence in a manner that would result in loss of data, then the GUI should first notify the user of the potential data loss and ask the user to confirm the interrupting action [26].

Checking this guideline automatically would require recreating the same scenario, which cannot be done without relying on GUI event sequencing.

6.0/18 User Confirmation of Destructive Actions: Users should be required to explicitly take an additional action for confirmation of a destructive operation [26].

The chain of events needed for testing this guidelines is simple and can be recreated with event sequencing. A test would delete an object, and wait for the confirmation message to appear. If it does, then the test would pass. Otherwise, the test would fail.

6.1/6 Limiting Unsuccessful LOG-ON Attempts: There should be a maximal rate and number of failed login operations in order to protect the system and its users from persistent attempts at illegitimate access [26].

A GUI test verifying compliance with this guideline can have a sequence of user actions attempting repeatedly to login with a wrong username or password, and check if the number of login attempts in a short time period is limited.

6.3/18 Cross Validation of Related Data: When a logically related data set is entered, the system should cross validate that data to ensure the logical consistency of the entered information [26].

Provided the tester fully understands how data is logically related, this recommendation can be verified through a GUI test that simulates a series of user actions, and checks if the GUI is capable of regular validation and cross validation.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

5.2.3 Listing Heuristics Fitting for Automation without Sequencing

This list includes heuristics, relevant in the general context of Windows desktop applications, which can be verified automatically but without the need for GUI event sequencing. Each heuristic starts with its reference number in the report developed by Smith and Mosier, followed by either a partial or complete description of said guideline, then it's followed by the reason it is believed that such a guideline belongs in this category. Inclusion in this class is based on the nature of the property to be assessed in the test. If the needed property is catchable by the GUI testing tool independently from potential user actions, then that state can be accessed and verified directly at the start of the test without needing to mimic a user action by means of a GUI event sequence. The same applies for guidelines whose focus is on attributes that are persisted in the system's back end. These can be verified independently from the GUI. These guidelines are listed below [26]:

1.4/12 Marking Required and Optional Data Fields: The GUI should clearly and consistently differentiate between optional and required data input fields whenever a form is presented to users [26].

Verifying if a GUI graphically differentiates between required and optional input fields can be done in an automated test. After identifying the manner in which optional and required fields are differentiated, it is then possible for a test to check if that difference is consistent between fields. It is important to note that the test only needs to check for the existence of the same visual mark across the form. Thus, the test does not have to verify if fields with a mark corresponding to "required" are indeed treated as required attributes by the system. In other words, to verify compliance with this guideline, it is not needed to trigger certain GUI events or interact with the interface, because the mark to be checked should always be displayed next to required (or optional) fields when the test starts. Therefore, the test does not need to create a GUI sequence to enact a certain scenario.

1.4/17 Consistent Label Format: A consistent presentation format should be assumed that would relate each label to its input element whenever data fields are dispersed across the GUI [26].

Verifying this guideline can be done automatically in order to check for instance whether labels are always aligned to the left of the fields, or placed immediately above them. These types of visual tests require access to the coordinates of these GUI elements on the screen (among other properties). However, there's no reason for adding to the test a sequence of GUI events when no actual interaction with the GUI needs to be simulated.

1.4/18 Label Punctuation as Entry Cue: For every input field, the label should be followed with a special character, indicating that data entry is allowed [26].

This guideline can be checked automatically by testing whether all labels (that precede input fields) end with the same symbol (such as a colon).

1.4/24 Form Compatible for Data Entry and Display: Entering a data form and later reviewing that same data should seem consistent for users. The labels, the order of elements, and their locations on the GUI should be similar in both data entry and data display [26].

Comparing the visual layout of the data entry form with the data display form can be performed automatically. These kinds of tests don't require a GUI sequence and can benefit from the assistance of testing tools that specialize in visual testing such as Applitools [161]. This guideline is especially relevant when paired with concepts such as responsive design (which readjusts the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

layout of the GUI according to screen size). However, since the focus of this study is on desktop applications, the variations in screen sizes and dimensions is less extreme than in phones and tablets. Nevertheless, the guideline should always be applied.

1.4/26 Minimal Cursor Positioning: When the user is filling a form, the movement of the cursor in the GUI from one field to the other should be minimized [26].

In the context of filling forms, this guideline recommends placing all required fields before any optional ones in order to improve the efficiency in data entry, and also not have the input fields placed too far from each other. Judging whether the visual arrangement of the different fields actually leads to minimal cursor positioning might be tricky to automate. However, provided the tester have access to the properties of those displayed GUI elements, and also sufficient knowledge about the nature of system input (such as what is required or optional), then a test can be automated. The test script would detect guideline violations such as optional fields that precede required ones, and input fields that are placed too far from the rest.

1.4/28 Automatic Cursor Placement: When presenting a form, the GUI has to position the cursor at the start of the first input field [26].

Checking the position of the cursor at default can be automated without the need for creating a sequence of GUI events. Only the first entry field has to be examined to check if it has cursor focus.

1.5/2 Distinctive Labels: When a table is displayed, the format of the column headers should be different from the rows in order to make it easier for users to distinguish data entries [26].

It is possible to automatically retrieve the formats used in the table, and compare them. Since all data needed for the test would be presented on the GUI at the same time, there is no reason to add a GUI sequence to the test.

1.8/4 Display of Default Values: During data entry, the defined standard values could be displayed in advance on their respective input fields. It should not be expected from users to remember default values. Moreover, in order not to confuse the user, those provided data values should be displayed in a different manner from the newly typed data entries [26].

The existence of default values in data fields can be verified with automation. There is no need to create a sequence of GUI events just to check a default value.

2.0/4 Data Display Consistent with User Conventions: "Display data consistently with standards and conventions familiar to users." [26].

An automated test can for instance check if a calendar is presented in a European or an American format depending on the localized version of the GUI. The same can apply for any unit of data that differ geographically, such as the metric system of measurement. Provided the data element to be tested is displayed on the GUI, an automated test can check its consistency with conventions without simulating additional user actions.

2.5/6 Page Labeling: When information is displayed across multiple pages, the label on each one of them should signify its relation to the others[26].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Provided the relation between pages (and even among windows) is consistent between software releases, it is possible to write an automated test that checks if these expected relations are expressed in the page labels. Since only the label element would be accessed by the test without simulating a user interaction, the test does not need to include a sequence of GUI events (but if those labels only appear under a certain usage scenario, then a sequence is needed).

2.6/26 Color Coding for Data Categories: If multiple separate data categories have to be differentiated by users, then it's advised to attribute a unique display color for each type of data [26].

Color detection and comparison of multiple categories of graphical elements can be checked in an automated test without the need for event sequencing.

3.1.8/3 Iconic Menus: If users don't share the same language or technical background, it's advised to present graphical menus with icons symbolising the different possible user actions [26].

The existence of an iconic menu or button on the GUI can be detected automatically by a test. No GUI sequence is required to verify if such a GUI element is displayed on the interface.

4.3/5 Brief Error Messages: Error messages should not have a lot of text but still be informative [26].

A tester can verify the length of error messages at the back end without needing to perform GUI tests. Generally, error messages are well organized and grouped inside the source code of an application, which makes retrieving them directly simpler than performing a series of GUI tests.

4.4/23 HELP: Users should be allowed to request additional guidance by invoking a "Help" command through the GUI [26].

It is possible to use automation to identify and check for the presence of a "Help" menu option, button, or icon on a GUI. No sequence of multiple GUI events is needed to test if an element representing "Help" exists on a GUI. However, an event sequence would be needed if it's to prove that different user guidance options are offered under different usage scenarios. On the other hand, automation cannot assert with confidence the relevance of the provided help to users. In other words, automation (without sequencing) can prove that a help option is offered on a GUI, while automation with event sequencing can check if help is offered under certain scenarios, but no amount of automation can assess how much said help would be useful for users. Therefore, based on how this guideline is described in the original report by Smith and Mosier, checking it could be automated but without needing sequencing.

5.1/5 Automatic Message Formatting: If a text message has to conform with a defined standard format, or if some parts of the message can be predicted by the system, then the GUI should assist the users by providing automatic means for checking or preparing a message [26].

The test evaluating the format of text messages does not even need to be a GUI test. Checking if the message is properly formatted can be performed automatically without the need for re-enacting a scenario through a series of GUI events.

6.0/13 Safe Defaults: During data entry operations, when standard values are offered to users, the defined default values should not contribute to the risk of data loss or make the task unnecessarily more complicated [26].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

These types of tests do not even need to be GUI tests. Automating the evaluation of this guideline can happen in the system's back end by testing if the default values lead to data loss situations.

6.1/5 Private Entry of Passwords When a user enters a password, the GUI should ensure the privacy of the input. In other words, the content of the password should not by displayed on the GUI [26].

Only the properties of the password input element has to be accessed by the test script in order to check how entered data would be displayed. There's no need to create a sequence or simulate a user action. Therefore, the test can be automated without event sequencing.

5.2.4 Listing Heuristics Suitable for Manual Testing

This list holds heuristics, relevant in the general context of Windows desktop applications, which should be verified manually. Just like the previous lists, each heuristic starts with its reference number in the original report, followed by either a partial or complete description of the guideline in question, then it's followed by the reasoning under which such a guideline came to be in this category. In general, these include heuristics that focus on attributes that are challenging to assess automatically, and that are currently not caught by a GUI automation tool. Such guidelines are listed in the following [26]:

1.4/11 Prompting Field Length: When a minimal, maximal, or an exact character string length is associated with a data input field, that information should be conveyed to users on the GUI [26].

Checking compliance with this guideline does not mean verifying if input fields have a maximal or minimal length, but rather means verifying if the label conveys a certain meaning correctly. Understanding the semantics behind a label and judging its significance for a particular field is a challenging task for automation because the same information can have various representations (such as "at least ten characters", "minimum 10 characters", "10 or more characters", or "characters $\geqslant$ 10"), especially when considering automated tests that have to pass all localised versions. As a result, the usefulness of additional information included in labels should be verified manually. The same logic applies for other guidelines verifying user guidance provided in a label such as date formats or units of measurement. Therefore, those highly similar recommendations will not be included again in this section (as they are judged to be duplicates).

1.4/19 Informative Labels: The labels used across the GUI should be descriptive, or else utilize default, predefined terms, or abbreviations [26].

Since it is challenging to automatically judge how informative a label is, verifying this guideline is best done manually.

1.4/27 Data Items in Logical Order As long as it is not stated otherwise in the requirements, the sequence order of data items should be the same order in which a user would think of them. [26].

Checking this guideline should be done manually because the perspective of an actual human user is required for performing the test.

2.0/2 Only Necessary Data Displayed: The amount of displayed data should be adjusted to user needs. The only data displayed to users should be the immediately usable information that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is necessary for performing a task. The data quantity should not overload the GUI and become a burden on the user. If the amount of needed data cannot be anticipated in advance, the GUI should be customizable enough to permit users to select which data to display and which ones to hide [26].

Since it may prove difficult for a machine to understand which data is necessary for the user, and which one is not (and by extension which data should be displayed and which one should be hidden), it is more appropriate to check this guideline manually.

2.0/15 Consistent Grammatical Structure: Across the whole GUI, the same linguistic and grammatical structure has to refer to the same data elements in a consistent manner, so as to not confuse users [26].

This linguistic and grammatical consistency has to be checked manually. Even though a program can check for simple grammatical errors, the problem becomes more complex when checking variations in linguistic and grammatical wording that refer to the same concept. This is especially noticeable when that concept is expressed in multiple words. In other words, testing the grammar is not the problem, but rather testing the consistency across the entire interface. Therefore, it is more appropriate to manually check if the displayed data and labels are linguistically consistent.

2.1/23 Logical List Ordering: Lists should be ordered in accordance to a logical principle. However, if no principle could be applied to the list, then that list should be ordered alphabetically. Nevertheless, the perspective of the user should always prevail over the perspective of the designer [26].

Verifying whether a list is ordered logically from the perspective of the user, is a task that should always be performed manually (and ideally by an expert).

2.5/16 Data Grouped by Importance: In case the presented data items are especially important for users, they should be grouped and clearly displayed at the upper area of the GUI. Such data items could represent for instance any data requiring immediate user response, or any information critical for the success or failure of the current task at hand [26].

Judging the importance of items for users cannot be determined, and validated in an automated manner, it is therefore better to rely on a manual evaluation to determine whether this recommendation was violated.

2.6/24 Brightness Inversion: Inverting the brightness of a graphical element so that dark characters on a bright background can be changed to bright on dark, or vice versa, can be used for highlighting critical data items or areas that have to attract user attention [26].

Compliance with this recommendation can be checked manually because it is challenging to automatically determine the importance of graphical elements for a user. Thus, it is better to have a usability expert decide if a GUI element needs to be highlighted, or if an already highlighted element shouldn't be.

2.6/28 Conservative Use of Color: Coloring should be done moderately and conservatively when designing the interface, and the GUI should contain a relatively few number of colors [26].

The correct use of color on a GUI can currently only be verified by an expert. Therefore, such tests should be performed manually (but at a future date, it might be possible to rely on machine learning to automatically detect color-related design violations [128]).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.0/2 Minimal User Actions: Control actions should favour simplicity, especially when the task performed needs a quick response from the user. The logic employed in the transaction sequence should contain a minimal number of user actions, while also being appropriate with the user's abilities and experience [26].

The verification of this recommendation is difficult to conduct automatically, it is simpler to rely on an expert to check manually if user actions are really kept to a minimum, or if some improvements are still possible.

3.0/16 Compatibility with User Expectations: The results of an operation should be displayed in a manner compatible with user expectations, and should seem natural [26].

Judging the compatibility between user actions and system responses is an important concept that should be checked manually because a human perspective is required.

3.1.3/11 Menu Options Worded as Commands: Menu options have to be worded clearly and consistently in a manner that represents system commands rather than questions directed at the user [26].

The semantics behind a word, used on a menu or a button are better understood by a human than a machine. Therefore, this heuristic should be evaluated manually.

3.1.3/22 Logical Grouping of Menu Options: A menu has to be presented in a way that relates to how logically connected a group of its options are [26].

The correct grouping of options, or their presentation in a logical order cannot be checked automatically. An expert has to manually verify if a menu violates this recommendation.

3.4/1 Defining Context for Users: The context of use has to be maintained while the user is going through a task consisting of a sequence of transactions. For instance, the result of a previous transaction has to be displayed if it can affect the current step of the task. Also, another example would be to only show available options during each point of the action sequence [26].

Even though the display of previous operations, and the availability of options can be checked with automated tests, the context of use is challenging to assess with a machine. After a design change or an updated software release, the context of certain tasks might change drastically. It is more appropriate to have a usability expert manually evaluate the context of use, and decide whether the amount of information and options offered to the user are appropriate.

4.0/17 Task-Oriented Wording: The label wording, prompted messages, and user guidance has to be task-oriented. In other words, the terms employed should be understood by the user, and should not be, for instance, technical expressions used by software developers [26].

Checking how effective the wording is in guiding the end user toward successfully performing the task at hand should be done manually and preferably by an expert.

4.3/1 Informative Error Messages: In case the system identifies an error made by the user, the error message prompted should notify the user of what went wrong, and inform him on how to correct the mistake [26].

Judging how informative an error message is for a user in a specific context cannot be measured by an automated test. These sorts of tests should be performed manually.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3/6 Neutral Wording for Error Messages: The text of error messages should always be neutral. That is to say, the text should not be funny, not contain irony, not blame the user, and not personalize the computer [26].

The evaluation of the text in error messages can benefit from machine learning techniques used in sentimental analysis, to categorize the wording used into positive, negative and neutral classes. However, it is preferred to perform theses tests manually (due to how technical error messages could be, and how challenging it is for a machine to detect humour or irony). Unless the number of error messages in a system is deemed too high for one expert to evaluate, automation is not needed.

4.3/12 Documenting Error Messages: User guidance should include the possibility to view a documented detailed explanation of every possible error message that could be returned by the system [26].

Naturally, this guideline should be checked manually, since an automated test cannot check if an error message is properly and sufficiently explained in the help section or in the documentation.

6.0/8 Appropriate Ease or Difficulty of User Actions: The completion of a task should be made as easy or as difficult as it needs to be. For example, user actions with high frequency or high priority should be performed easily, while actions that could result in data loss should include an additional step requesting user confirmation [26].

The evaluation of task ease or difficulty and whether it fits with the user's cognitive needs attributed to the task at hand, should be performed manually by an expert.

6.1/1 Easy LOG-ON: Logging in to the application should be designed in a manner that favours simplicity as much as possible, all the while balancing the needs to protect the system from unauthorized access [26].

Judging whether the LOG-ON action is easy or difficult should be assessed by a usability expert, since the ease of use of GUI elements is not a property directly observable (or catchable) by GUI testing tools.

6.5/2 Protection from Design Change: Any modification made to the design of the GUI should not hinder or weaken functions assisting the areas of data entry, data display, sequence control, user guidance, data transmission, and data protection [26].

This guideline has one of the largest scopes in the report. Even though it is classified as a data protection recommendation, it covers all six functional areas of user-system interaction. In particular, it focuses on detecting if a design change might decrease the usability of an interface at any level. In other words, it is reasonable to assume that this particular guideline encapsulates the vast majority (if not all) of the other guidelines in the report. Therefore, this broad guideline includes inner heuristics that can be automated and also those that should be done manually (as was described throughout this section). Nevertheless, the expertise of a usability professional is needed to assess the overall effect of a design change on the usability of an interface (but if some automated guideline evaluation tests are implemented, they can save a lot of time for the expert, since that would leave him only the manual tests).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.3 Development of Heuristic Evaluation in a GUI Testing Tool

In order to prove that a GUI testing tool is capable of improving a usability inspection process, the tool should be capable of automating the evaluation of some heuristics. Therefore, some guidelines are selected from the list derived by Smith and Mosier (which was described at the start of section 5.2), to be tested in a practical automation experiment. These all belong to the category of heuristics suitable for GUI event sequencing (previously derived and listed in section 5.2.2). Thus, an adequate GUI test automation tool is also needed to execute the testing scripts. Furthermore, some instances of Windows software applications are required as well to serve as ground for the testing tool to use in the verification of guideline violations. In other words, the experiment requires an automation tool, a GUI to test, and a list of guidelines to verify.

5.3.1 Preparing Needed Development Environment

Among the GUI testing tools presented in section 4.1.3, is Unified Functional Testing (UFT) [152], which was developed by Hewlett Packard Enterprise (HPE) and later on by Micro Focus, and is one of the leading test automation tools for Windows applications on the market. It is therefore selected as the tool under which the testing scripts are written and executed. The version worked with is 14.03, and among its installed features, it offers a variety of add-ins that support optimal performance and reliable object identification when working with applications whose GUI was built on frameworks, languages, and technologies, such as Visual Basic, ActiveX, Java, .Net, and Microsoft UI Automation.

In fact, additional software programs are also needed to serve as applications under test. Any testing script written under UFT has the purpose of checking if a given heuristic is violated in a specified GUI. To that end, two software programs that run under the Windows operating system are selected to be experimented upon. These are Eclipse Photon for Java developers (version 4.8) [206], and MyFlight Sample Application (version 14.03) by HPE and Micro Focus, which is by default incorporated with UFT [152].

First of all, Eclipse is one of the most widely used IDE by software developers. It is built by the Eclipse Foundation, and is continuously supported by the contributions of a globally active community, counting big corporations, some small companies, and even open source enthusiasts [207]. The Eclipse IDE is flexible and rich on features, which makes its GUI a good test subject for checking usability guideline violations with the UFT tool. It is available free of charge under the Eclipse Public License (EPL).

As for the second software, MyFlight Sample Application is a simplified flight reservation simulator built-in with UFT. It was developed with the intention of helping software testers learn how to use the UFT tool for automating GUI and API tests. It mimics typical features offered in software developed for reservation systems, such as booking, browsing, and searching. Since the main purpose of MyFlight is encouraging testers to experiment with UFT, this application does not require changes to its settings to improve the portability of test cases built around it. Furthermore, the MyFlight GUI tests do not require Internet access.

Since Eclipse's GUI is built using the Java programming language, the UFT Java add-in must be loaded in the developing environment before executing any automated GUI tests centered on Java GUI objects, or else the UFT tool cannot identify the displayed GUI elements [208].

Similarly, MyFlight Application is built using the GUI framework of the .Net Windows Presentation Foundation (WPF). Thus, its test scripts require the WPF add-in, which in turn is dependent on the web and .Net add-ins [208]. In other words, it requires the web and .Net add-ins to be installed with UFT, but it does not require them to be loaded each time the UFT tool is starting up.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Reference [26]	Guideline Title in the Original Report [26]	Application Tested
1.0/4	Fast Response	MyFlight Application
1.3/10	Upper and Lower Case Equivalent in Search.	MyFlight Application
1.4/15	Explicit Tabbing to Data Fields.	MyFlight Application
1.7/3	Non-Disruptive Error Messages	MyFlight Application
2.7.5/3	User-Specified Windows.	Eclipse Photon IDE
3.0/1	Flexible Sequence Control.	Eclipse Photon IDE
3.0/20	Indicating Control Lockout.	MyFlight Application
3.1.3/7	Menu Selection by Keyed Entry.	Eclipse Photon IDE
3.2/10	Only Available Options Offered.	Eclipse Photon IDE
3.3/3	CANCEL Option.	Eclipse Photon IDE
3.5/10	UNDO to Reverse Control Actions.	Eclipse Photon IDE
4.3/13	Cursor Placement Following Error.	MyFlight Application
6.0/5	Protection from Interrupts.	Eclipse Photon IDE
6.0/18	User Confirmation of Destructive Actions.	MyFlight Application

Table 5.1: Usability guidelines selected for practical experiments

On a side note, in case the test steps are being executed too fast for the human eye to keep up with, and thus, for an observer to comprehend, it is possible to slow down the execution by having the tool wait some time between individual programming steps within the same test script. This is achieved in UFT by selecting the "Tools" menu list, and clicking the "Options" menu item. Once the option window opens, it's possible to select the "GUI Testing" tab, then "Test Runs". Finally, under "Run Mode", it's advised to check the "Normal" radio button, and specify the delay time in milliseconds.

5.3.2 Instances of Usability Heuristics Evaluation in a GUI Test Automation Tool

The complete developed source code for automating all the tests detailed below is publicly available on the GitHub platform [1], and is published under the GNU General Public License (GPLv3) [209]. The selected guidelines to be experimented upon with UFT, in the two applications under test, are stated in table 5.1. The choice of which guideline to test on which application is partly based on the degree of guideline applicability on the GUI, and partly based on the need to balance the number of tests equally between the two applications. At least one heuristic is selected from each one of the different functional areas covered by the work of Smith and Mosier except the fifth area which focuses on data transmission. The areas tested are: Data entry, data display, sequence control, user guidance, and data protection, which were all explained in the beginning of section 5.2). The automation of each guideline in table 5.1 was already briefly discussed in section 5.2.2 preceded by a description of the particular guideline. Throughout this section, the experimentation with automated heuristic evaluation is examined. The guidelines are discussed in the same order, as their appearance in the table. However, before focusing on individual automated heuristic inspections, it's first needed to explain some UFT-specific testing terms, and briefly describe what most of these tests might have in common.

In order to facilitate understanding the tests, diminish the required effort, and minimize the size of test folders, all tests created in UFT can be divided into multiple logical sections, called actions (which work similarly to regular functions in the VBScript scripting language) [210].

The order in which such actions is executed can be displayed in a convenient flow diagram generated by UFT [211]. Some test steps are designed to be reused (or recalled) during the execution

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

of many different tests. Such test elements can be grouped together into logical sections and are called in UFT external reusable actions [210].

A couple of examples of actions needed by all tests would be starting and closing an application at the beginning and at the end of the test respectively. Therefore, each one of the two applications under test (Eclipse Photon and MyFlight) have a general test case composed only of reusable actions, these two test cases are the only ones that do not check for heuristic violations, but are simply present so that other test cases can avoid duplicate code by calling their actions externally, whenever needed.

The first general test, whose flow diagram is presented in figure 5.1 alongside an expended view of its nested actions, is named "Common Eclipse Behaviour Test". It describes a general eclipse test with four reusable actions (excluding nested actions). In summary, it starts the Eclipse software, creates a Java project, then deletes it, and finally it closes the IDE. The first test action called "Start Application" extracts the user name from the operating system's environment variables, so that it can form a string representing the installation path of the application. Once Eclipse is started, the action would wait for the pre-launch window that asks the user which workspace to open, then it creates a new workspace, and have it be located in the same parent folder as the last opened workspace (or as the default one if it is the first Eclipse run). This prevents testing efforts from interfering with potential existing work, and isolates the testing environment from interferences by some unforeseen factors. On a later test execution, if the workspace prompted is not the one which was created, the script in the "Start Application" action would look for the newly created workspace in the parent folder of the current workspace, and create it again if it was not found. Finally, if the pre-launch window is not prompted on Eclipse startup, the action would change the settings to have that window appear during startup, and then it would restart Eclipse (which explains why the "Start Application" action has a nested action called "Change Setting to Display Launcher", which in turn, contains calls to the "Close Application" and "Start Application" actions, so that it could restart the Eclipse IDE).

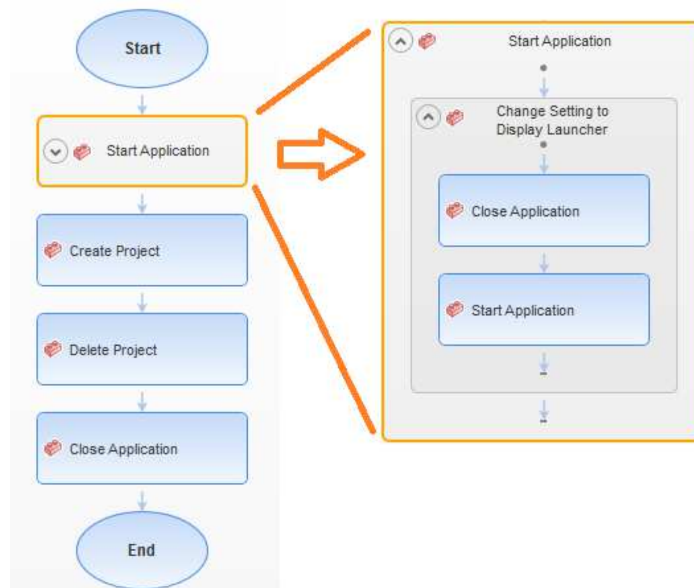

Figure 5.1: UFT flow diagram for a general test case in Eclipse Photon with an expanded view of its nested actions

Secondly, test actions which are common among two or more test cases built around the GUI of MyFlight application, are all grouped under one test case. This test is called "Common MyFlight Behaviour Test" and its UFT flow diagram is shown in figure 5.2. It contains nine self-explanatory

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

actions. First, it starts the application, then it logs in. After that, it searches for a flight, selects one from the search results, and books that particular flight order (which includes entering the passenger name). Once it goes back to the initial window, the test searches for the passenger name that was entered on the last booking order, selects it from the search results, and then deletes it. Finally, the application shuts down.

Figure 5.2: UFT flow diagram for a general test case in MyFlight application

Since the common actions among test cases have been discussed, the tests checking for guideline violations can be examined. It's important to keep in mind that all of these tests are but simplified problem instances to check whether heuristic evaluation can really be automated in practice. Across the following sections, any UFT flow diagram that is not presented alongside its corresponding test description, can be viewed in appendix A where all the additional diagrams are grouped.

Experimenting with Data Entry Guidelines

The first functional area of user-system interaction, according to Smith and Mosier is data entry [26]. Also, the first data entry guideline to be tested is titled "Fast Response". As aforementioned in the previous section, this guideline is still valid, and has a high impact on usability [119].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

It instructs developers that for normal operation, the delay in displayed feedback should not surpass 0.2 seconds. Needless to say, what is considered a normal operation on a mainframe computer during the 1980s is greatly different from a contemporary normal operation for the average modern computer. Nevertheless, there are still three time limits to consider when testing responsiveness which are the following [205] [119]:

- A tenth of a second is about the limit when the goal is for the user to perceive an instantaneous system reaction (such as the time that passes between typing text and the characters being displayed on the screen).
- One second is the upper limit when the user's flow of thought should not be broken (such as when the user is editing data).
- Ten seconds are the maximum when the goal is to keep the attention of a user (as can happen when waiting for the computer to finish processing a task).

Nowadays, displaying typed text is typically handled by the operating system. Therefore, the UFT test checks if the GUI of the MyFlight application reacts in less than one second during a login and a search operation. Figure 5.3 displays the UFT flow diagram of the test, where it is shown that the login and the search occur within the same action. That is because that action has only one focus, which is to measure the system's response time.

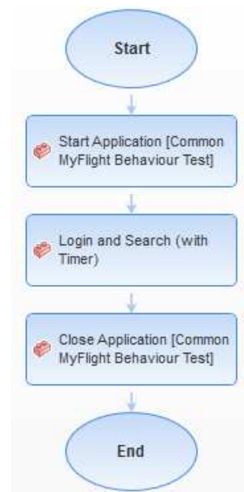

Figure 5.3: UFT flow diagram for guideline 1.0/4 "Fast Response"

The GUI event sequence used for performing the login part of the test can be seen in listing 5.1. First the agent name and password are entered to the text fields (as demonstrated in lines 1 and 2), then a timer logs the starting point (as seen in line 3) right before submitting the form (which occurs in line 4). Once the existence of one GUI element associated with the expected result is confirmed (such as in line 5), another timer logs the ending time (seen here in line 6). The difference between the two timers corresponds to the rounded-up duration of a login (which happens in line 7). If that value is less than one second, the test reports a success (entering thereby the if-branch with lines 8 and 9). Otherwise, the test reports a failure (and executes lines 10 and 11).

```
1 WpfWindow("HPE MyFlight Sample Application").WpfEdit("agentName").
  ↪ Set "john"
2 WpfWindow("HPE MyFlight Sample Application").WpfEdit("password").
  ↪ SetSecure "5b51f6f18e41810734bb"
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
3  loginStarted = timer
4  WpfWindow("HPE MyFlight Sample Application").WpfButton("OK").Click
5  If WpfWindow("HPE MyFlight Sample Application").WpfComboBox("
   ↪ fromCity").Exist Then
6    loginFinished = timer
7    loginTime = loginFinished - loginStarted
8    If loginTime < 1 Then
9      Reporter.ReportEvent micPass, "Login Response Time is less than
       ↪ a second", "The user's flow of thought is uninterrupted"
10   else
11     Reporter.ReportEvent micFail, "Login Response Time is longer
       ↪ than a second", "The user's flow of thought might be broken"
12   End If
13 End If
```

Listing 5.1: Code Snippet with the GUI event sequence verifying login responsiveness [1]

The second data entry guideline to be experimented upon is called "Upper and Lower Case Equivalent in Search". As the name implies, this test would check if the system differentiates between search queries with upper and lower case letters. Most GUI developers nowadays consider dealing with this issue early on without being advised to, so this problem resurfaces less frequently. However, it did not disappear, and it is still a good practice to test a GUI for compliance with this guideline. Figure 5.4 shows the UFT flow diagram of the performed test. During the action "Search the Same Order Differently", the script simulates a search operation of a booking order on the GUI of the MyFlight application, then it checks if the actual result is the expected one during the action called "Display Order Details". The test is repeated four times with a different passenger name in each iteration, and this distinction is only in terms of upper and lower case letters (being "Jane", "jane", "JANE", and "jaNE", which are stored in a global data table).

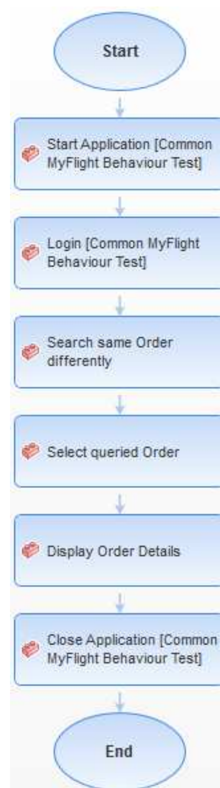

Figure 5.4: UFT flow diagram for guideline 1.3/10 "Upper and Lower Case Equivalent in Search"

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Furthermore, the guideline titled "Explicit Tabbing to Data Fields" is the third data entry guideline to be automated. The test checking this on MyFlight application simulate pressing the tab key, then checks which graphical element has input focus, or in simpler terms, it checks the cursor's position after a tab key entry. Logging-in, searching for flights, and ordering a flight, were all performed with tabbing navigation. The GUI passed this test, since it does not violate this guideline. Listing 5.2 shows the content of the login action. After the simulation of a tabbing event (performed in line 2), the test confirms if the cursor's position is on the first text field (as shown in line 3) by comparing the actual current cursor placement with the predefined expected one. Then, the test proceeds to enter the agent name (seen in line 4), followed by a second tab key press (done in line 5). The second checkpoint checks (in line 6) if the password text field currently has cursor focus. Afterwards, the password is entered (in line 7) and the next tabbing event is executed (in line 8). In a last step, the final checkpoint confirms that the cursor focus property of the "OK" button is true (demonstrated in line 9), followed by a click on that button (shown in the last line). If one of the checkpoints presented in the source code does not pass, the test does not pass as well. The order in which the cursor focus moves from one graphical element to the next, is easy to follow (as exemplified by the numbered GUI components on the right half of figure 5.5, which is further described below).

```
1  'Simulates Pressing the Tab key
2  WpfWindow("HPE MyFlight Sample Application").Type micTab
3  WpfWindow("HPE MyFlight Sample Application").WpfEdit("agentName").
   ↪ Check CheckPoint("AgentName Field has Cursor Focus")
4  WpfWindow("HPE MyFlight Sample Application").WpfEdit("agentName").
   ↪ Set "john"
5  WpfWindow("HPE MyFlight Sample Application").Type micTab
6  WpfWindow("HPE MyFlight Sample Application").WpfEdit("password").
   ↪ Check CheckPoint("Password Field has Cursor Focus")
7  WpfWindow("HPE MyFlight Sample Application").WpfEdit("password").
   ↪ SetSecure "5b51f6f18e41810734bb"
8  WpfWindow("HPE MyFlight Sample Application").Type micTab
9  WpfWindow("HPE MyFlight Sample Application").WpfButton("OK").Check
   ↪ CheckPoint("OK Button has Cursor Focus")
10 WpfWindow("HPE MyFlight Sample Application").WpfButton("OK").Click
```

Listing 5.2: Code Snippet with the GUI event sequence checking tabbing navigation in the login window [1]

This recommendation can be pushed further to evaluate the order in which GUI elements receive cursor focus after tabbing. Generally, after pressing the tab key, cursor focus would move to the next interactive GUI element in order of appearance, starting from left to right, and from top to bottom. However, it is sometimes, more appropriate to move after tabbing to the next most important interactive element to the user. Figure 5.5 shows the login windows of two applications, being MyFlight on the right, and Steam [212] on the left. Each window contains numbers corresponding to the tabbing order of GUI elements (which are enclosed in a red color). As can be seen in the figure, the sixth and seventh elements of the Steam window go in order of importance rather than in order of appearance. Provided the tester knows in which GUI elements the user is most interested, an automated test can be written to check if the tabbing order of elements conforms with their logical merit for the user.

The last data entry guideline is named "Non-Disruptive Error Messages". The test checking this guideline on MyFlight's GUI tries to search for flights using a date in the past, which is not allowed. Then it checks for the exact moment in which the user is notified of his error. If the GUI displays an error message the moment the mistake occurs, meaning the moment an erroneous date

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

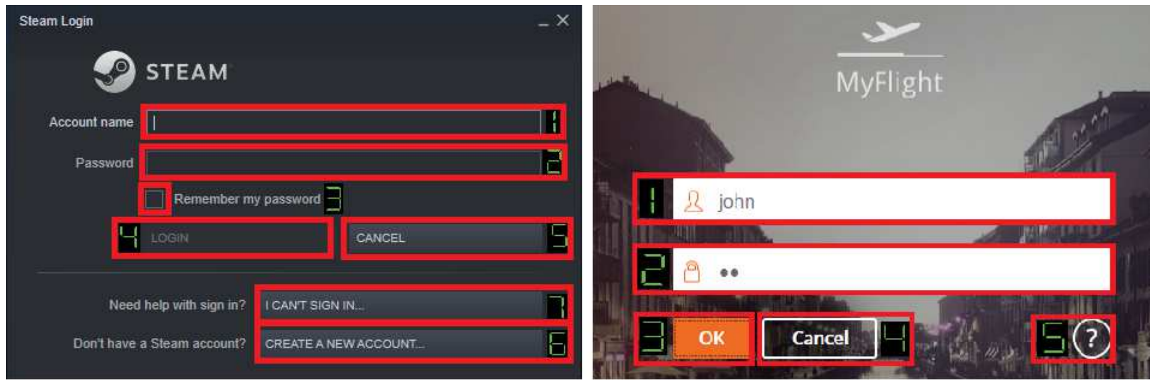

Figure 5.5: Side by side view of the tabbing numeric order in the login windows of Steam (left), and MyFlight application (right)

is selected and before the submit button is pressed. Then, the test would fail, because it disrupts the user. However, if the GUI gives enough time to the user and only displays the error message on submit. Then, the test passes, which it does in this case since MyFlight application abides by this guideline.

Experimenting with Data Display Guidelines

Data display is the second functional area to be studied. Even though most of its automation-friendly guidelines do not need event sequencing for their tests, some of them can only be tested with a GUI event sequence. The heuristic called "User-Specified Windows" is one of them. As already described in the previous section, in order to check if multiple types of data can be viewed together, the properties of the graphical elements displaying such data need to be accessed by the test, all the while, the script simulates the actions of a user creating a special window, and precising the different kinds of content he expects to be displayed.

The Eclipse IDE offers the possibility of viewing a variety of data on the same interface. This makes it an ideal candidate for testing if checking the guideline "User-Specified Windows" can be automated. As seen in figure 5.6 representing the UFT flow diagram of such test, the main action is named "Customize Window", and holds two more nested actions called "Save Perspective" and "Reset Window". First of all, the test script displays on the same window, incrementally more GUI tab elements, each corresponding to a different view. Once all views are displayed in the same layout, the test checks if the earlier tabs still exist on the same window at the same time, then proceeds by closing all views. This portion of the test passes as long as it is possible to add and remove different kinds of views, to be displayed simultaneously on the same window.

Once the layout is completely empty except for the editor (meaning all the views were closed), a new test starts. Three views are recalled, which are named "Ant", the "Package Explorer", and the "Console". This particular layout can then be saved as a custom perspective called "Minimalistic Java Perspective", which the test does. Afterwards, the test verifies if this created perspective is added as a new choice among the list of all offered perspectives. Figure 5.7 shows how the described layout looks like on the left side, while on the right side, it shows the name of this perspective listed with the rest. As can be observed on the figure, the old Java perspective also still exist (which is called "Java (default)"). Subsequently, the test deletes the minimalistic Java perspective, and restores the layout to its default setting, before finally closing the IDE. Since Eclipse obviously abides by the guideline, the test passes in its entirety.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

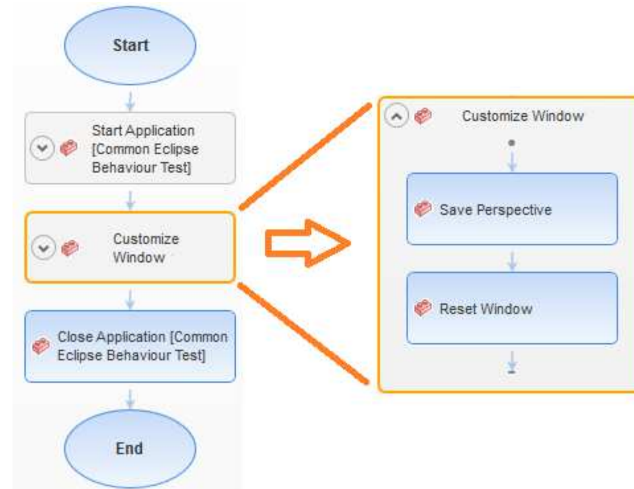

Figure 5.6: UFT flow diagram for guideline 2.7.5/3 "User-Specified Windows" with an expanded view of its nested actions

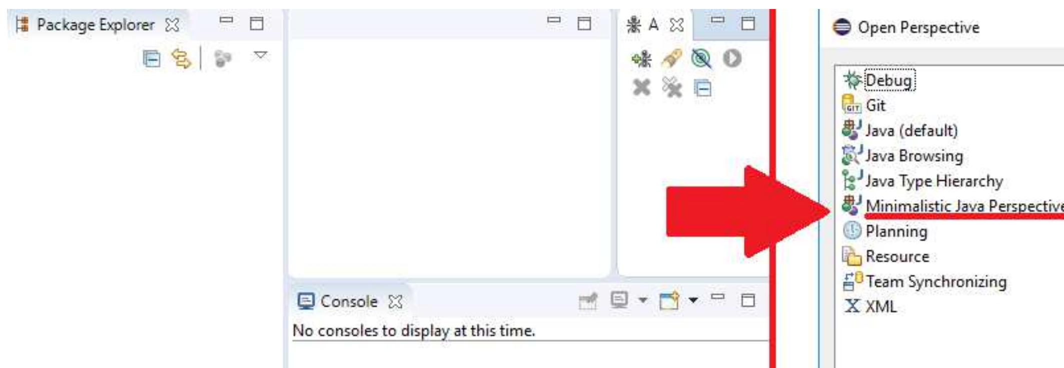

Figure 5.7: An instance of an Eclipse custom Java perspective

Experimenting with Sequence Control Guidelines

The third functional area of system-user interaction is sequence control, and its first guideline to be automated is named "Flexible Sequence Control". Generally, testing flexibility of GUI operations, requires going through a high number of possible sequence combinations [43]. As was described in section 1.1, even a GUI of an application as simple as Microsoft WordPad contains over 324 possible GUI operations [2]. Checking the flexibility can be challenging to automate, as the possible end to end paths the user might take are verified in a similar manner to scenario testing (which is a specification-based testing technique explained in section 3.1.3). However, since a heuristic evaluation does not aim at uncovering code defects or increasing code coverage but rather identify usability issues, the test can be reduced to checking the sequence paths that users are most likely to go through. To that end, the experience of a usability professional is needed to specify the exact sequence of operations that should better be tested. For instance, on a supposed hypothetical login window, a user can initially start a sign-up action, then stop midway through, and try a sign-in action. Once that fails, he can return to the sign-up action, where it would be expected that he would continue the process where he stopped. Another example would be simply starting a certain operation, asking for guidance relevant to the task at hand (such as by pressing the "help" icon), then proceeding with the operation that was interrupted. This particular scenario is tested in the Eclipse IDE.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The test checking if compliance with the guideline "Flexible Sequence Control" can be challenging to automate. A variety of scenarios can be enforced. In case of the Eclipse IDE one particular usage scenario was evaluated, the script starts by creating a new project, then in the middle of the operation, it presses the "Java Build Path Help" icon (symbolised with a question mark) which is provided in the second page of the window "New Java Project" (accessed after clicking the "Next" button). Once the test script confirms the existence of the help window, it closes it and proceeds to continue creating the project. If asking for user guidance does not break the operation sequence initiating the help request, then that part of the GUI complies with the guideline.

On a side note, this test of the Eclipse IDE accidentally helped uncover a different usability issue than the one it was checking. The help window called during testing does not provide any actual guidance for users, since it simply displays the message "Topic not found", as can be viewed in the figure 5.8. However, since this automated test is only checking for compliance with the guideline "Flexible Sequence Control", it passes.

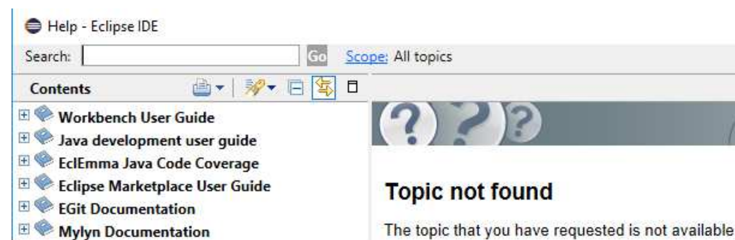

Figure 5.8: The help window of the Eclipse IDE, prompted while testing guideline 3.0/1 Flexible Sequence Control

Another sequence control guideline is called "Indicating Control Lockout". The test script checking this heuristic on the MyFlight application, focuses on the "Order" button which is present on the page displaying order details. The changes that this button passes through are shown in figure 5.9. The first and default button state is to be disabled, this continues as long as the passenger name's text field is empty. If that text field is not empty, then the button enters the enabled state and can be pressed. Finally, once the button is clicked, it would change to its third and final state and become invisible. However, the change from the second to the third state is not instantaneous. The GUI first displays a progress bar at the bottom of the window, and the "Order" button becomes invisible only after the progress bar is filled completely. In other words, while the progress bar is active, the "Order" button is still enabled. Therefore, if the button were to be pressed before becoming invisible, the booking order operation would be carried on again, and the progress bar would reset. This behaviour constitutes a guideline violation, and carries a risk of failure in preventing errors.

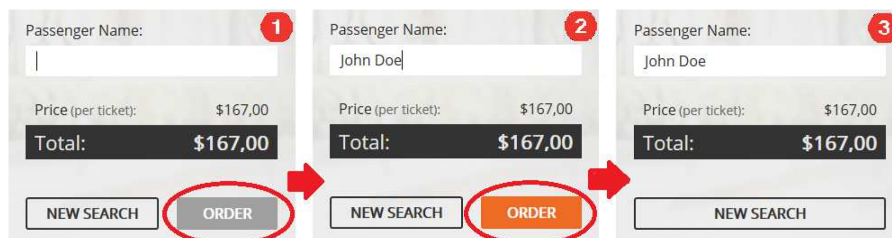

Figure 5.9: The three states of the "Order" button ("disabled" on the left, "enabled" in the middle, and "invisible" on the right)

With this in mind, the test automatically checking this guideline observes the "Order" button states, and verifies if they are in line with the ongoing operation. Its respective flow diagram, which is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

conveyed in figure 5.10, show that the only non-reusable action in this test is the action "Order Same Flight Twice in a Row". Since the GUI does not comply with this guideline the test fails and returns two errors: The first error is because the "Order" button was still enabled after being clicked, and the second error is because the application did not stop the additional booking order after a second click.

Figure 5.10: UFT flow diagram for guideline 3.0/20 "Indicating Control Lockout"

Another sequence control heuristic is titled "Menu Selection by Keyed Entry". As the name implies, an automated GUI test checking this guideline would simulate key presses and check if the desired outcome was achieved. Said test was performed on the Eclipse IDE. Through a key press combination, a new Java project is created, then a new Java class is made and saved. Afterwards, the whole project is deleted, similarly with only key presses. After performing each action, the script accesses the properties of GUI elements to validate the behaviour of the interface. Naturally, this test passes as the Eclipse IDE offers keyboard shortcut keys for its most repetitive and sought after features. Therefore, checking the guideline "Menu Selection by Keyed Entry" can be automated thanks to GUI event sequencing.

Furthermore, the guideline "Only Available Options Offered" also belongs to the sequence control functional area. A test checking compliance with it on the GUI of the Eclipse IDE is represented in figure 5.11, where its UFT flow diagram can be seen. The test verifies the availability of the main buttons in the "New Java Project" window in accordance with the different steps performed in the first non-reusable action titled "Create Project (while checking availability)". These buttons being labelled "Back", "Next", "Finish", and "Cancel", are disabled whenever appropriate, and enabled otherwise. Therefore, this portion of the test passes. Secondly, the action "Delete Project (while checking availability)" checks the state of the "Delete" button after pressing the navigational buttons on the prompted deletion confirmation window. Similarly to the first action, this portion of the test passes as well.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

On the other hand, the last non-reusable action "Search Empty Project", initiates a search operation on an empty workspace. Figure 5.12 gives an idea of how the availability of the "Search" button is handled by the GUI. Despite the search text field being empty, the search button is enabled, all the while, it becomes disabled if the scope of the search is not set. Pressing the button while the search text field is empty always results in no matches being found. Therefore, the search operation in Eclipse violates the guideline "Only Available Options Offered", which explains why the automated test fails, and returns the error: "Search" button is available for empty strings in an empty workspace.

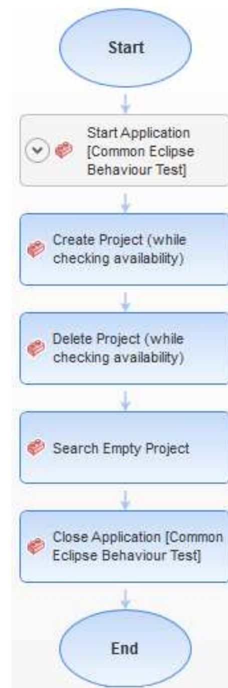

Figure 5.11: UFT flow diagram for guideline 3.2/10 "Only Available Options Offered"

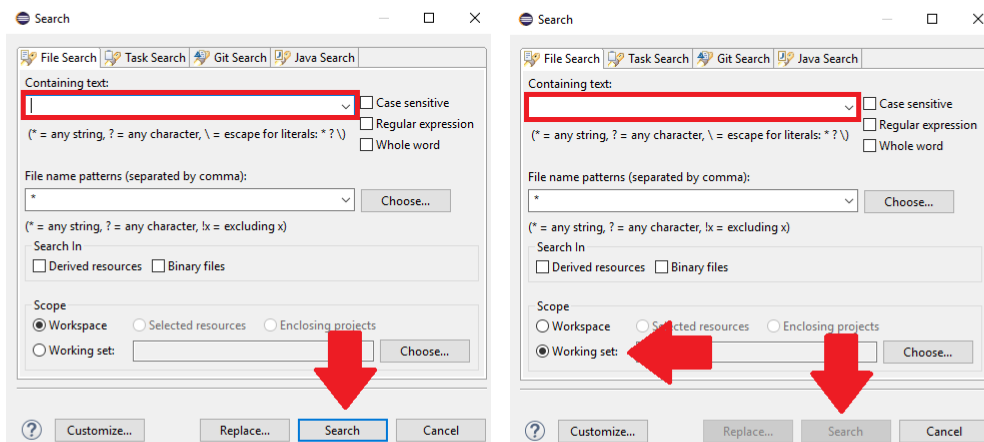

Figure 5.12: The availability of the "Search" button in the "Search" window of Eclipse Photon

Furthermore, an additional sequence control heuristic judged suitable for automation with event sequencing is titled "Cancel Option". The test checking the feasibility of a cancel operation in the Eclipse IDE is rather simple. The only new action in this test is called "Write Program (Include a Cancel)", where the script creates a new "HelloWorld" Java class, have it contain some code, then tries to close the editor tab without saving. This leads Eclipse to ask the user not only if he wishes

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

to save or not save, but also if he wants to cancel closing the tab. The test verifies the existence of the "Cancel" button, and since the Eclipse IDE complies with the guideline, the test passes.

Providing users the choice to cancel an operation might seem one of the most intuitive guidelines to follow. However, it is not rare to find programs that violates this heuristic, such as Netflix desktop application (which can be downloaded and installed directly from the Microsoft store). Netflix is a subscription-based streaming service, providing access to a huge library of films and series [213]. However, the whole application does not seem to contain a cancel button. The login and sign-up operations cannot be cancelled. Also, a rather strange instance can be seen in the figure 5.13. When the Netflix application is started while the operating system is not connected to the Internet. The notification message provides only a "Try again" button, next to the text "(No)" which naturally cannot be interacted with. Therefore, it is advised not to neglect checking a GUI's compliance with this guideline when applicable, since the effort needed for automating testing this heuristic is minimal (that is to say, it only requires testing the presence of a GUI element corresponding to the cancel option).

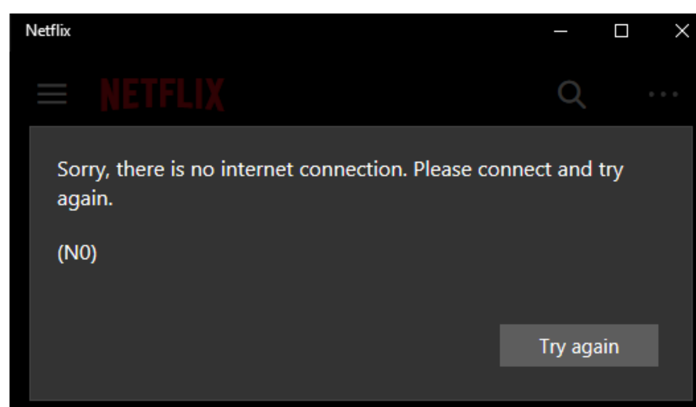

Figure 5.13: Screenshot of the Netflix desktop application when started up without Internet connection

Lastly, the sequence control heuristic "UNDO to Reverse Control Actions" can also be checked on the GUI of the Eclipse IDE. To that end, the test script creates a Java Class called "HelloWorld", types a program in the editor panel, then proceeds to undo the last changes made in that editor. This last step is done by pressing the "Undo" Menu item under the "Edit" Menu option. As a result, the method last added to the "HelloWorld" class is removed from the editor. Finally, the result of the undo operation is reversed again, by clicking this time the "Redo" Menu option. This cancels the effect of the last undo instruction. Successfully performing these undo and redo operations means the GUI complies with the guideline, and thus, the automated test performing the operations passes.

Experimenting with User Guidance Guidelines

The next functional area in Smith and Mosier's guidelines is user guidance [26]. The guideline, belonging to this area, to be tested with automation is called "Cursor Placement Following Error". Its UFT flow diagram is shown in figure 5.14. The only non-reusable action here is "Search a Flight with Past Date". This test is very similar to the test of the data entry guideline "Non-Disruptive Error Messages". The same error is being reproduced which is searching for a flight while the selected date is in the past. However, this time, the focus of the test is not on the moment in which the error message is prompted but rather on the cursor's placement after the error message is dismissed.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In MyFlight application, after an error message is prompted and handled by the user, the cursor position is reset to the first graphical element on the window, located in the upper left corner of the window. Thus, the test fails because it expects the cursor focus to be positioned on the date picker element. This violation is detected, and the error returned is labelled "Cursor Focus is NOT Placed on Erroneous Field".

Figure 5.14: UFT flow diagram for guideline 4.3/13 "Cursor Placement Following Error"

Experimenting with Data Protection Guidelines

Data Protection is the last functional area of user-system interaction presented in the work of Smith and Mosier [26]. One of the heuristics belonging to this category is called "Protection from Interrupts", and simply focuses on preventing unexpected data loss when a task is abruptly interrupted. The automated test checking the compliance of the Eclipse IDE with this rule is very similar to the test checking the guideline "Cancel Option". In other words, both tests create a "HelloWorld" Java class, and attempt to close that class tab in the editor before saving the work. In particular, the step that closes the tab is the interrupting event. From this angle, the test aims at verifying if the GUI can notify the user that unsaved data will be lost (which in this case is represented by the added code in the Java class). The flow diagram of the described program behaviour is represented in figure 5.15. As can be seen on the diagram, the main action in this test is called "Write a Program and Interrupt it Without Saving". Its concrete GUI event sequence in code form is shown in listing 5.3. First, cursor focus is put on the sample project by selecting it (as performed in line 1). Afterwards, the main menu option "File" is selected, followed by the sub-menu "New", then the option "Class" in order to prompt the window responsible for creating a new Java class (which is all encapsulated in line 2). In the newly introduced window, the text "HelloWorld" is entered in the name field (as done in line 3), and the "Finnish" button is clicked (as seen in line 4). Following that, the sample project is selected again (as shown in line 5), then the "HelloWorld" class tab in the editor is given cursor focus (which is done in line 6). Subsequently, the Java code to display the text "Hello World" on console is entered (which corresponds to line 7). As an interrupting action, that "HelloWorld" class tab is closed (as can be seen in line 9). Finally, the test checks if, as a result, a warning window appeared offering the user to save his changes, and reports back the success or failure of the test accordingly (which matches in code lines 10 to 15). The test passes, since the GUI does in fact abide by the guideline.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
1  JavaWindow("Eclipse").JavaTree("Tree").Select "Sample Project"
2  JavaWindow("Eclipse").JavaMenu("File_2").JavaMenu("New").JavaMenu("
   ↪ Class").Select
3  JavaWindow("Eclipse").JavaWindow("New Java Class").JavaEdit("Name:")
   ↪ .Set "HelloWorld"
4  JavaWindow("Eclipse").JavaWindow("New Java Class").JavaButton("
   ↪ Finish").Click
5  JavaWindow("Eclipse").JavaTree("Tree").Expand "Sample Project"
6  JavaWindow("Eclipse Local").JavaEdit("StyledText").SetFocus
7  JavaWindow("Eclipse Local").JavaEdit("StyledText").Set "" + vbCrLf +
   ↪  "public class HelloWorld {" + vbCrLf + "" + vbCrLf + "    public
   ↪  static void main(String[] args) {" + vbCrLf + "         // Prints
   ↪  out ""Hello, World"" on the Output Screen." + vbCrLf + "
   ↪ System.out.println(""Hello, World: The Guideline titled
   ↪ Protection from Data Loss is being tested"");" + vbCrLf + "    }"
   ↪  + vbCrLf + "" + vbCrLf + "}"
8  'Closing the tab is an INTERRUPT Action
9  JavaWindow("Eclipse Local").JavaTab("CTabFolder_2").CloseTab "*
   ↪ HelloWorld.java"
10 If JavaWindow("Eclipse Local").JavaWindow("Save Resource").Exist
   ↪ Then
11   Reporter.ReportEvent micPass, "Data Loss was prevented when
     ↪ closing the tab", "The user is offered to save his file before
     ↪ closing the tab"
12   JavaWindow("Eclipse Local").JavaWindow("Save Resource").JavaButton
     ↪ ("Save").Click
13 else
14   Reporter.ReportEvent micFail, "Data loss occured", "The Tab was
     ↪ closed before the user was prompted to save his data"
15 End If
```

Listing 5.3: Source code with the GUI event sequence verifying data protection from an interrupting action [1]

The final guideline, whose verification is automated, is titled "User Confirmation of Destructive Actions" and belongs to the functional area of data protection. Its test script simply enters a new booking order in the MyFlight application, then deletes it. Automation efforts are centered around detecting the presence of the notification message asking to confirm the delete operation, right after pressing the delete button. This message is shown in figure 5.16, which proves the compliance with the guideline. However, the presence of a totally different usability problem is apparent. The buttons are displayed in a different language than the message text (one being German, and the other is English), which is probably due to the Windows operating system under which the program is installed being in the German language. This localisation issue might interfere with the ability to confirm the delete operation for a small percentage of users, but since this is outside the scope of this heuristic, the test passes.

5.4 Evaluation of the Results

Before analysing the results of the study, it is important to be briefly reminded of the steps taken in the study. First, a minimal set of heuristics was derived from a pre-existing well-accepted list of usability guidelines developed by Smith and Mosier over 30 years ago. In the second step, the guidelines in the derived list were categorised into three classes: Ones suitable for automation with GUI event sequencing, ones suitable for automation without the need to simulate user inter-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar. The approved original version of this thesis is available in print at TU Wien Bibliothek. TU Bibliothek WIEN Your knowledge hub

Figure 5.15: UFT flow diagram for guideline 6.0/5 "Protection from Interrupts"

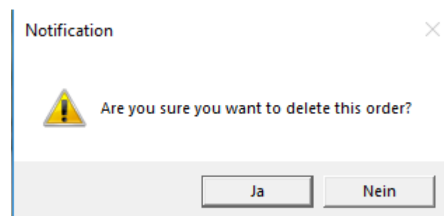

Figure 5.16: Confirmation message of a to be deleted booking order in the MyFlight application

actions, and those that should be checked manually. Finally, in the last step, a sample of guidelines belonging to the first category are automated in a practical experiment. The results of each step in this study are summarised, examined, and relied upon to comprehend whether heuristic evaluation could benefit from automation in general and automation with GUI event sequencing in particular, or whether this usability inspection method should continue to be performed manually in its entirety (as currently done in practice).

5.4.1 Assessment of the outcome of the Heuristics Filtering Process

Examining the list of design guidelines for user interfaces, led to filtering 944 guidelines [26]. After dismissing recommendations deemed either intuitive, redundant, obsolete, too context-specific, or non-applicable in a Windows desktop environment, the result consisted of 85 guidelines (which were all detailed in section 5.2). In other words, the minimal set of heuristics which was derived withheld 9% of the guidelines presented in the original report.

Table 5.2 shows the results of the guideline filtering process, by presenting the number of derived guidelines in each functional area of user-system interaction as originally divided in the aforementioned report by Smith and Mosier [26]. The table also offers the percentage of selected guidelines compared to their original number in each functional area, and it labels this data as the retrieval rate. Most functional areas have similar retrieval rates being over but close to 10% of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

original guideline amount in each respective area. The only exceptions are data display and data transmission with guideline retrieval rates lower than 5%.

In the case of the data display area, the reason for the low retrieval rate is due to the large amount of very specific and somewhat repetitive guidelines that would rarely apply in the general context of a common Windows desktop application. These guidelines advise GUI designers about how to properly present different elements such as diagrams, charts, maps, and even auditory signals. However, the main lessons behind these recommendations are similar (which consist of favouring attributes such as consistency, clarity, and simplicity). Moreover, the data display area holds the highest number of guidelines, making up close to one third of the entire report (31.5% of it to be precise). Thus, even if the number of derived guidelines is not too far from that of other areas, the retrieval rate would still be lower in comparison.

Concerning data transmission which is one of the smallest functional area, the cause of its low guideline retrieval rate is due to how outdated the area is. By extension, almost all its recommendations are either obsolete, or became intuitive for GUI designers. In the document segment describing this area, the report even include the following line: "Computer-mediated data transmission is sometimes called electronic mail" [26]. Therefore, it's safe to consider these recommendations to be oriented towards email interfaces. However, when Jakob Nielsen revisited these guidelines in 2005, he argued that many of them were relevant for systems other than email interfaces, where users could for instance share data or notify each other [203]. Nevertheless, during the heuristic derivation process, only three guidelines were selected from this area, as can be observed in table 5.2. This is because the derived heuristics should not be too entangled with one particular type of applications.

Functional Area [26]	**Original Guidelines [26]**	**Derived Guidelines**	**Retrieval Rate**
Data entry	199	27	13.5%
Data display	298	14	4.6%
Sequence control	184	20	10.8%
User guidance	110	12	10.9%
Data transmission	83	3	3.6%
Data protection	70	9	12.8%

Table 5.2: Guidelines selection data grouped by functional interaction area

At the time of this study, the heuristics developed by Smith and Mosier are over 30 years old. Yet, if it weren't for the need to minimize the derived set of heuristics (for the reasons explained in section 5.2), the amount of selected guidelines would have comprised much more than 9% of the original report. This is because redundant guidelines, or ones that are too context-specific would have been included. It is important to note that a big portion of guidelines in the report are still valid and relevant for contemporary desktop applications, despite the fact that these guidelines were originally aimed at old-fashioned mainframe computers.

In the software development world, requirements change with time, new features are implemented regularly, and new technologies appear constantly. Compared to the frequency of change in modern software and compared to the fast rate of evolution in technologies, heuristics covering user-

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

system interaction can be fairly sturdy in the face of time. Once a set of guidelines (approved by usability experts) are selected and bound to an application, these guidelines are expected to stay valid and relevant over a long period of time. They are also less likely to be subject to change than functional requirements.

Given this background, if a set of design guidelines is enforced early on in development, the number of iterations in a design phase would be reduced. As a result, the overall development cost would go down, and a moderate usability degree might be reached from the start, which would yield some of the usability benefits discussed in section 3.3.2. Naturally, performing a heuristic evaluation in every single design iteration might get counter-intuitive, which is why automating part of this process is so appealing. Other reasons why the automation of this usability inspection method is studied are to gain some of the benefits associated with the field of GUI test automation which are described in section 3.2.3. Furthermore, the similarities and differences between these two fields are covered in details in section 4.1. With this in mind, the suitability of the derived set of heuristics with automation is analysed, and especially, when automation re-enacts a sequence of user actions, in the same manner as a typical automated script-based GUI test.

5.4.2 Analysis of Automation Feasibility in Heuristic Evaluation

The derived minimal set of usability heuristics is divided into three lists according to their compatibility degree with automation. The first category holds the guidelines suitable for automation with GUI event sequencing (in the same manner as in script-based GUI testing), and they are all listed in section 5.2.2. The second category contains heuristics also suitable for automation, except that they don't require a sequence of GUI events for performing the test. These are listed in section 5.2.3. Finally, the third class has guidelines that ought better be evaluated manually, and they are described in section 5.2.4. The proportions of each category are expressed in percentages and presented in a pie chart shown in figure 5.17. As can be observed in the chart, over half of the guidelines can be automated with event sequencing, and three quarters of it can be automated in general (which can be noted by summing up the two first categories). These results mean that for this particular set of heuristics, automated usability inspections could decrease up to 75% of the effort needed for a complete manual heuristic evaluation (as long as these tests require a reasonable maintenance effort). Moreover, it could also be deduced that a GUI testing tool can be used to assist in the automation of over half of the derived guidelines.

More details about automated evaluation feasibility can be examined through the data presented by table 5.3. It can be noted that the functional areas of user-system interaction called data entry and sequence control are the two most suited for automated evaluation with event sequencing. The reason for this is due to the high amount of guidelines in these areas that involve a particular usage scenario. In other words, these guidelines can only be evaluated by interacting with the GUI directly, and cannot be observed by launching the application and looking at the GUI. As a result, their automated evaluation has to reproduce the same GUI interactions in order to check for guideline violations.

Another aspect that can be observed is that in all functional areas, the overall number of guidelines suitable for automation is always superior to those appropriate for manual testing. Despite that, one particular functional area distinguishes itself from the others in its high compatibility with manual testing. This area is data display. The reason behind this result is due to the high proportion of data display guidelines that assess properties that cannot be captured by a GUI automation tool. For instance, automated GUI testing cannot measure the importance of one graphical element compared to others, or the correctness of its logical placement.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

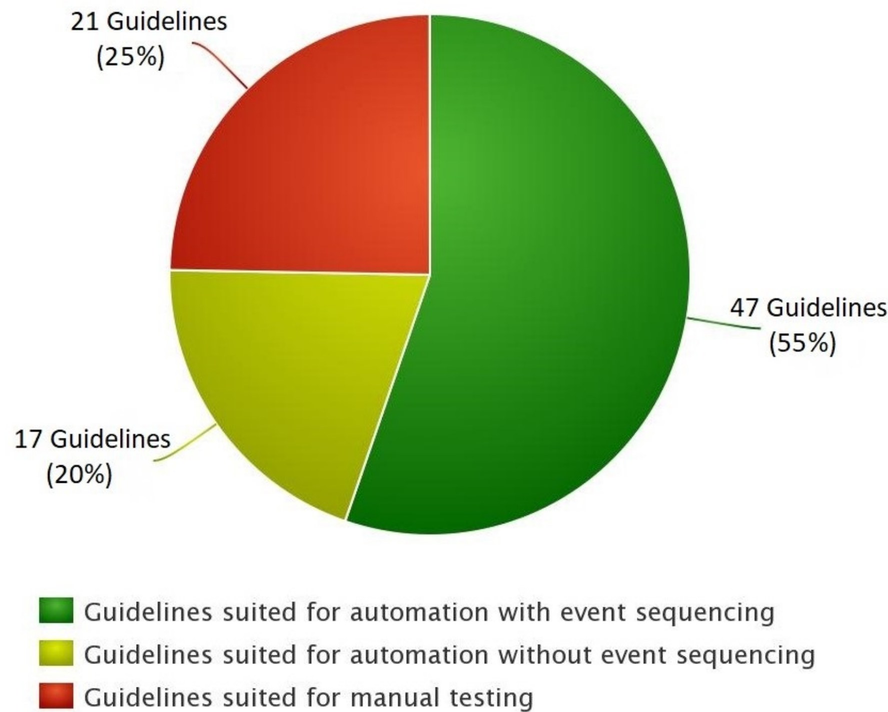

Figure 5.17: Overall classification results after analysis of test automation feasibility on a derived set of adequate heuristics

Functional Area [26]	Guidelines Suited for:		
	Automation with Event Sequencing	Automation without Sequencing	Manual Testing
Data entry	16	8	3
Data display	5	3	6
Sequence control	14	1	5
User guidance	6	2	4
Data transmission	2	1	0
Data protection	4	2	3

Table 5.3: Feasibility of heuristic evaluation automation grouped by functional area

Based on the presented classification results, and in particular, based on the nature of the guidelines in each particular listing (from section 5.2), a pattern can be observed. This means that analysing automation feasibility of heuristic-based usability inspections leads to the observation of common traits between the guidelines in each one of the three categories no matter what functional area they belong to.

First of all, every single guideline suited for automation with GUI event sequencing has to re-enact at least one user-system interaction in order to check if the GUI abides by a guideline. In

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

other words, the test evaluating each one of these guidelines would examine the GUI while it goes through at least two states: The first GUI state is observed before the interaction begins, and the second starts after the user interacts with the GUI. Additionally, a GUI can pass through more states if the interaction being evaluated comprises multiple steps. Therefore, the evaluation of a heuristic can be automated with GUI event sequencing, if the verification of compliance with that guideline requires checking the properties of the displayed GUI elements before and after performing (or simulating) a user-system interaction.

Subsequently, there are also some common traits between all guidelines suitable for automation without a GUI events sequence. They all involve checking observable GUI properties whose assessment do not depend on an external user action. For example, an automated test can check if some GUI elements are visually aligned without requiring event sequencing. In other words, GUI elements under test only go through one state, and the capture of their specific properties do not depend on user-system interaction. In some cases, testing compliance with a guideline can be performed in the back end of the system because the attribute to be checked can be accessed and verified before it is even displayed on the GUI, such as checking the length of error messages, or the side-effects of working with the predefined default values.

Upon examination of the guidelines suited for manual testing, a common trait among all of them is observed. They all require capturing a property that is not directly observable on the GUI, but rather one that can be assessed through a human cognitive deduction process. This means that the desired properties cannot be captured by a GUI automation tool. Some examples would be to judge the aesthetic value of a GUI element, its importance for the user, or its linguistic consistency with similar elements. A human perspective is needed for performing these tests, thus they ought to be tested manually. However, in a future date, machine learning techniques might solve (or at least alleviate) this automation challenge [128].

The validity of the classification results was challenged in practice by automating some heuristic-based inspections in an experiment. As a result, 14 GUI tests were created that check for usability violations instead of identifying defects (and all of these automated instances were described in section 5.3.2). The existence of such programs in practice prove that heuristic-based usability inspection of Windows applications using an event-driven approach is feasible, and that using a GUI testing tool (such as UFT) greatly simplifies the automation process.

Moreover, some observations were only made evident thanks to the practical experiment, and these are the following:

- In some cases, to check compliance with a guideline across an entire GUI, the amount of possible operations or GUI elements to test can be quite high, and by extension, the initial effort required for the automation can also be extensive. Therefore, a usability professional can work together with testers in order to prioritize GUI aspects with stronger impact on usability, or to specify which hypothetical scenarios to consider. For instance, checking guidelines dealing with GUI flexibility can become a process similar to scenario testing (which is a specification-based technique defined in section 3.1.3). It might be challenging for a tester to choose with confidence which scenarios ensure a satisfying level of flexibility, but it is hard to imagine that anyone could be better at selecting realistic usage scenarios than an experienced usability professional who observed users over a long period of time, and extensively studied user behaviour. Therefore, software testers and usability experts could work together to make the most out of automation. Just like GUI developers can work together with usability professionals to improve the quality of their interfaces, software testers might also work together with usability experts to efficiently automate heuristic evaluations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

- The evaluation of many heuristics can be automated without the need for a usability expert to guide testers, or review the test cases. This is because these heuristics are formulated clearly, and leave almost no room for interpretation. For instance, checking if a user is notified of unsaved data when he is trying to exit, or checking if multiple error messages are prompted consecutively (instead of just one message), are the sort of guidelines that can be easily converted into event-driven GUI tests, without requiring prior knowledge about usability.

- Even if heuristic evaluation is fully automated. All the tests can do is check if the GUI abides by a set of guidelines. They do not assess the severity of the individual usability violations uncovered. Even in the general sense, an automated heuristic evaluation cannot be used to measure usability. Measuring the usability attribute of an interface was discussed in section 2.2.3, and in summary, the appraisal process always requires a user sample.

- Some guidelines should not be applied on a GUI at the same time, even if their evaluation can be automated on the same GUI. For instance, there are distinct guidelines that check if data is grouped either by its function, importance, frequency of use, or alphabetical order. It is sufficient to check compliance with only one grouping logic. Choosing one guideline among many conflicting ones can mean a trade-off is needed between two desirable qualities. The decision to enforce a certain guideline over another should be coming from the perspective of users and not from the perspective of designers.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6 Conclusion

Heuristic evaluation is a cheap and simple usability inspection method that is being generally performed manually in its entirety even though the evaluator does not need to observe a user sample. It has been theorized in this thesis that GUI event sequencing, despite being a technique normally used in script-based GUI testing, would have a positive effect on the automation of heuristic-based usability inspection. Therefore, it was needed to analyse the compatibility of heuristic evaluation with automation in general, and with GUI event sequencing in particular.

First, the literature and vocabulary relating to GUI, usability engineering, and software testing fundamentals was considered. Helpful taxonomy was presented to assess the compatibility of usability evaluation methods with automation, including discussing the current progress and success degree of said automation efforts. It was deduced that among all methods presented, inspection methods have the greatest automation potential as they do not require user presence. Afterwards, an in depth comparison of usability evaluation and GUI testing was provided alongside a review of available tools assisting in the automation of each respective field. Finally, after discussing heuristics and their durability, a reputable source of usability guidelines was selected and tailored to fit a specific context of use, which is set to be modern Windows desktop applications. Analysing automation feasibility of the derived heuristics, on a case by case basis, led to classifying 55% of them to be suitable for automation with GUI event sequencing, while 75% of the guidelines were compatible with automation in general. Only 25% of them were more suited for manual testing. The validity of the provided classification was challenged in practice through the implementation of a number of heuristic evaluations that has been judged as appropriate for automation with GUI event sequencing. The experiment involved checking the compliance of two software applications with fourteen different guidelines, and was performed on an event-driven testing tool normally used in GUI testing. Each automated guideline verification was successfully converted into a script-based GUI test whose objective is to identify usability issues instead of software defects.

Through the performed work, a common pattern was uncovered between all heuristics belonging to each category. In other words, a guideline can be verified automatically with event sequencing as long as its manual test requires the evaluator to interact with the GUI, triggering at least one event. On the other hand, automation without event sequencing is more appropriate if the manual evaluation of a guideline involves assessing easily observable attributes such as visual alignment of graphical elements, or if the focus of the recommendation is on aspects that do not depend on the GUI itself and can be verified even in some unit tests, such as when checking the text length of stored error messages, or when testing if the offered default values might contribute to the risk of data loss. Finally, manual evaluation is advised for every guideline revolving around GUI attributes that cannot be captured by a GUI test automation tool. The assessment of these GUI qualities is deduced through a human cognitive process, and sometimes requires the expertise of a specialist. For instance, such attributes are manually evaluated in guidelines judging the aesthetic sense of the GUI, or examining the logic by which data items are ordered and menu options are grouped.

Since not all heuristics can be applied at once on the same application, the percentage of heuristics whose evaluation is suitable for automation and those that should be checked manually may differ according to the software project and depending on the nature of the selected guidelines that the development team decides to abide by. Therefore, by relying on a GUI testing tool and the event sequencing technique, it is possible for a team to select and conform with a heuristic set consisting

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

entirely of guidelines whose evaluation can be automated. As a result, heuristic-based usability inspection can be performed easily and repeatedly during an iterative design process or alongside regression tests after committing some design changes. This kind of automated evaluation does not aim to identify all existing usability issues, but rather has the goal of quickly achieving and cheaply sustaining an appropriate degree of usability. However, even when the set of selected guidelines is 100% compatible with automation, the role of the usability evaluator cannot be replaced. The expertise of the evaluator is needed to guide testing efforts such as by providing reasonable usage scenarios and relevant action sequences for performing a task as a beginner or as an expert user. His presence is also required for overseeing the automated evaluation process, validating the overall results, and deducing the severity of identified issues.

Deeper analysis can be done in future extensions where case studies would be observed in which a development team has its testers collaborate with usability experts to automate a considerable portion of a heuristic evaluation. The result of such studies can be examined to confirm and assess the impact of an automated usability inspection on the quality of the designed GUI in relation to its cost. It would also enable the researcher to observe and measure how effective automation is in assisting the evaluator in practice. Furthermore, an additional potential lies in machine learning. Eventually, it might be possible to rely on machine learning techniques to automate the evaluation of heuristics that can currently only be tested manually such as judging the aesthetic value of the interface, or the correctness of the logical grouping of menu options. All those efforts would further close the gap between automated and manual heuristic evaluation.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

References

[2] A. M. Memon, M. E. Pollack, and M. L. Soffa. „Using a goal-driven approach to generate test cases for GUIs“. In: *Software Engineering, 1999. Proceedings of the 1999 International Conference on*. IEEE. 1999, pp. 257–266.

[3] A. K. Ames and H. Jie. „Critical paths for GUI regression testing“. In: *University of California, Santa Cruz* (2004).

[7] I. DIS. „9241-11: 1998. Ergonomic requirements for office work with visual display terminals-Part 11: Guidance on usability“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[8] J. Nielsen. *Usability engineering*. Elsevier, 1993. ISBN: 9780080520292.

[9] J. Cardello. *Nielsen Norman Group, Three Uses for Analytics in User-Experience Practices*. https://www.nngroup.com/articles/analytics-user-experience. Accessed: 2019-05-27.

[10] J. Cardello. *Nielsen Norman Group, Five Essential Analytics Reports for UX Strategists*. https://www.nngroup.com/articles/analytics-reports-ux-strategists. Accessed: 2019-05-27.

[11] J. Nielsen. „Enhancing the Explanatory Power of Usability Heuristics“. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’94. Boston, Massachusetts, USA: ACM, 1994, pp. 152–158. ISBN: 0-89791-650-6. DOI: 10.1145/191666.191729. URL: http://doi.acm.org/10.1145/191666.191729.

[12] R. Inostroza et al. „Usability heuristics for touchscreen-based mobile devices“. In: *Information Technology: New Generations (ITNG), 2012 Ninth International Conference on*. IEEE. 2012, pp. 662–667.

[13] L. M. Tobar, P. M. L. Andrés, and E. L. Lapena. „WebA: A Tool for the Assistance in Design and Evaluation of Websites.“ In: *J. UCS* 14.9 (2008), pp. 1496–1512.

[14] M. B. J. Machate. „Creative design of interactive products and use of usability guidelines-a contradiction?“ In: *Human-Computer Interaction: Theory and Practice* 1 (2003), p. 43.

[15] A. Beirekdar, J. Vanderdonckt, and M. Noirhomme-Fraiture. „KWARESMI–Knowledge-based Web Automated Evaluation Tool with Reconfigurable Guidelines Optimization“. In: *Proc. of 2nd International Conf. on Universal Access in Human-Computer Interaction UAHCI’2003*. 2003.

[16] A. Beirekdar et al. „Flexible reporting for automated usability and accessibility evaluation of web sites“. In: *Human-Computer Interaction-INTERACT 2005* (2005), pp. 281–294.

[17] J. Mifsud and A. Dingli. *USEFul: A Framework to Mainstream Web Site Usability Through Automated Evaluation*. LAP LAMBERT Academic Publishing, 2012.

[18] E. H. Chi et al. „The bloodhound project: automating discovery of web usability issues using the InfoScentπ simulator“. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 2003, pp. 505–512.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[19] J. Nielsen. *Nielsen Norman Group, Return on Investment for Usability*. https://www.nngroup.com/articles/return-on-investment-for-usability/. Accessed: 2019-05-27.

[20] R. Oppermann. „User-interface design“. In: *Handbook on information technologies for education and training*. Springer, 2002, pp. 233–248. ISBN: 978-3-540-74155-8.

[21] R. Unger and C. Chandler. *A Project Guide to UX Design: For user experience designers in the field or in the making*. New Riders, 2012, pp. 1–15.

[22] D. Saffer. *Designing for interaction: creating innovative applications and devices*. New Riders, 2010, pp. 1–29.

[23] I. DIS. „9241-110: 2006. Ergonomics of human system interaction-Part 110: Dialogue principles“. In: *International Standardization Organization (ISO). Switzerland* (2006).

[24] I. DIS. „9241-12: 1998. Ergonomic requirements for office work with visual display terminals-Part 12: Presentation of information“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[25] I. DIS. „9241-13: 1998. Ergonomic requirements for office work with visual display terminals-Part 13: User guidance“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[26] S. L. Smith and J. N. Mosier. *Guidelines for designing user interface software*. Mitre Corporation Bedford, MA, 1986.

[27] S. Fowler. *GUI design handbook*. McGraw-Hill, Inc., 1998. ISBN: 978-0070592742.

[28] M. Habibi, J. Patterson, and T. Camerlengo. „The Graphical User Interface“. In: *The Sun Certified Java Developer Exam with J2SE 1.4*. Springer, 2002, pp. 189–262.

[29] I. DIS. „9241-14: 1998. Ergonomic requirements for office work with visual display terminals-Part 14: Menu dialogues“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[30] I. DIS. „9241-15: 1998. Ergonomic requirements for office work with visual display terminals-Part 15: Command dialogues“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[31] I. DIS. „9241-16: 1998. Ergonomic requirements for office work with visual display terminals-Part 16: Direct manipulation dialogues“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[32] I. DIS. „9241-17: 1998. Ergonomic requirements for office work with visual display terminals-Part 17: Form filling dialogues“. In: *International Standardization Organization (ISO). Switzerland* (1998).

[33] J. D. Gould and C. Lewis. „Designing for usability: key principles and what designers think“. In: *Communications of the ACM* 28.3 (1985), pp. 300–311.

[34] S. Möller. *Quality engineering: Qualität kommunikationstechnischer Systeme*. Springer, 2010, pp. 57–74.

[35] I. DIS. „9241-210: 2010. Ergonomics of human system interaction-Part 210: Human-centred design for interactive systems“. In: *International Standardization Organization (ISO). Switzerland* (2010). Reviewed and Confirmed in 2015.

[36] I. DIS. „ISO/IEC 25062: 2006. Software engineering, Software product Quality Requirements and Evaluation (SQuaRE), Common Industry Format (CIF) for usability test reports“. In: *International Standardization Organization (ISO). Switzerland* (2006).

[37] J. R. Lewis. „An after-scenario questionnaire for usability studies: psychometric evaluation over three trials“. In: *ACM SIGCHI Bulletin* 23.4 (1991), p. 79.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[38] M. McGee. „Usability magnitude estimation“. In: *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*. Vol. 47. 4. SAGE Publications Sage CA: Los Angeles, CA. 2003, pp. 691–695.

[39] J. Nielsen. *Nielsen Norman Group, Usability Metrics*. https://www.nngroup.com/articles/usability-metrics/. Accessed: 2019-05-27.

[40] A. Spillner, T. Linz, and H. Schaefer. *Software testing foundations: a study guide for the certified tester exam*. Rocky Nook, Inc., 2014.

[41] I.-S. S. Board. „IEEE standard classification for software anomalies“. In: *IEEE Std* 1044 (2009).

[42] I. DIS. „ISO/IEC/IEEE 24765:2010. Systems and software engineering, Vocabulary“. In: *International Standardization Organization (ISO). Switzerland* (2010). Reviewed and Confirmed in 2016.

[43] G. J. Myers, C. Sandler, and T. Badgett. *The art of software testing, Third Edition*. John Wiley & Sons, 2011, pp. 21–42.

[44] P. C. Jorgensen. *Software testing: a craftsman's approach*. CRC press, 2016. ISBN: 978-1466560680.

[45] A. M. Hass. *Guide to advanced software testing*. Artech House, 2014.

[46] I ISTQB. „Glossary of Testing Terms“. In: (2015).

[47] I. DIS. „ISO/IEC/IEEE 29119-4:2015. Software and systems engineering, Software testing, Part 4: Test techniques“. In: *International Standardization Organization (ISO). Switzerland* (2015).

[48] P. Bourque, R. E. Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[49] J. Cem Kaner. „An Introduction to Scenario Testing“. In: (2003).

[50] A. M. Hass. *Guide to advanced software testing*. Artech House, 2014, pp. 1–23.

[51] M. E. Khan, F. Khan, et al. „A comparative study of white box, black box and grey box testing techniques“. In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 3.6 (2012).

[52] M. Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[53] S. R. Jan et al. „An Innovative Approach to Investigate Various Software Testing Techniques and Strategies“. In: *International Journal of Scientific Research in Science, Engineering and Technology (IJSRSET), Print ISSN* (2016), pp. 2395–1990.

[54] W. Wei et al. „From source code analysis to static software testing“. In: *Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on*. IEEE. 2014, pp. 1280–1283.

[55] R. E. Fairley. „Tutorial: Static analysis and dynamic testing of computer software“. In: *Computer* 11.4 (1978), pp. 14–23.

[56] K. Wiegers and J. Beatty. *Software requirements*. Pearson Education, 2013.

[57] C. R. Camacho, S. Marczak, and D. S. Cruzes. „Agile Team Members Perceptions on Non-functional Testing: Influencing Factors from an Empirical Study“. In: *Availability, Reliability and Security (ARES), 2016 11th International Conference on*. IEEE. 2016, pp. 582–589.

[58] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[59] D. Saff and M. D. Ernst. „Reducing wasted development time via continuous testing“. In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE. 2003, pp. 281–292.

[60] T. Xie et al. „Towards a framework for differential unit testing of object-oriented programs“. In: *Proceedings of the Second International Workshop on Automation of Software Test*. IEEE Computer Society. 2007, p. 5.

[61] R. C. Martin. *Clean Coder: A Code of Conduct for Professional Programmers*. mitp Verlags GmbH & Co. KG, 2014, pp. 77–84.

[62] S. Berner, R. Weber, and R. K. Keller. „Observations and lessons learned from automated testing“. In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE. 2005, pp. 571–579.

[63] E. Dustin, T. Garrett, and B. Gauf. *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education, 2009.

[64] C. S. Jensen, M. R. Prasad, and A. Møller. „Automated testing with targeted event sequence generation“. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 67–77.

[65] K. Li and M. Wu. *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons, 2006.

[66] A. Adamoli et al. „Automated GUI performance testing“. In: *Software Quality Journal* 19.4 (2011), pp. 801–839.

[67] Z. Juan et al. „Test case reusability metrics model“. In: *Computer Technology and Development (ICCTD), 2010 2nd International Conference on*. IEEE. 2010, pp. 294–298.

[68] *Microsoft Developer Network, Model-Based Testing*. https://msdn.microsoft.com/en-us/library/ee620469.aspx. Accessed: 2019-05-27.

[69] V. Entin et al. „Combining model-based and capture-replay testing techniques of graphical user interfaces: An industrial approach“. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*. IEEE. 2011, pp. 572–577.

[70] O. Taipale et al. „Trade-off between automated and manual software testing“. In: *International Journal of System Assurance Engineering and Management* 2.2 (2011), pp. 114–125.

[71] J. Scholtz. „Usability evaluation“. In: *National Institute of Standards and Technology* (2004).

[72] H. R. Hartson, T. S. Andre, and R. C. Williges. „Criteria for evaluating usability evaluation methods“. In: *International Journal of Human-Computer Interaction* 15.1 (2003), pp. 145–181.

[73] M. Theofanos and W. Quesenbery. „Towards the design of effective formative test reports“. In: *Journal of usability studies* 1.1 (2005), pp. 27–45.

[74] A. Fernandez, E. Insfran, and S. Abrahão. „Usability evaluation methods for the web: A systematic mapping study“. In: *Information and Software Technology* 53.8 (2011), pp. 789–817.

[75] J. Nielsen. „Iterative user-interface design“. In: *Computer* 26.11 (1993), pp. 32–41.

[76] N. Bevan. „Cost benefits evidence and case studies“. In: *Cost-justifying usability: An update for the internet age. San Francisco: Morgan Kaufmann* (2005).

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[77] J. Nielsen. *Designing web usability: The practice of simplicity*. New Riders Publishing, 1999.

[78] P. J. Clarkson et al. *Inclusive design: Design for the whole population*. Springer Science & Business Media, 2013.

[79] M. Y. Ivory and M. A. Hearst. „The state of the art in automating usability evaluation of user interfaces“. In: *ACM Computing Surveys (CSUR)* 33.4 (2001), pp. 470–516.

[80] J. Nielsen. *Nielsen Norman Group, Thinking Aloud: The Number One Usability Tool*. https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/. Accessed: 2019-05-27.

[81] P. R. Vora and M. Helander. „A teaching method as an alternative to the concurrent think-aloud method for usability testing“. In: *Advances in Human Factors/Ergonomics* 20 (1995), pp. 375–380.

[82] J. Gerken et al. „How to use interaction logs effectively for usability evaluation“. In: *BELIV*. 2008.

[83] A. Schade. *Nielsen Norman Group, Remote Usability Tests: Moderated and Unmoderated*. https://www.nngroup.com/articles/remote-usability-tests/. Accessed: 2019-05-28.

[84] J. Nielsen and T. K. Landauer. „A mathematical model of the finding of usability problems“. In: *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*. ACM. 1993, pp. 206–213.

[85] J. Nielsen. *Nielsen Norman Group, Why You Only Need to Test with 5 Users*. https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users/. Accessed: 2019-05-27.

[86] R. J. Fisher. „Social desirability bias and the validity of indirect questioning“. In: *Journal of consumer research* 20.2 (1993), pp. 303–315.

[87] H. Beyer and K. Holtzblatt. *Contextual design: defining customer-centered systems*. Elsevier, 1997.

[88] J. Nielsen. *Nielsen Norman Group, The Use and Misuse of Focus Groups*. https://www.nngroup.com/articles/focus-groups/. Accessed: 2019-05-28.

[89] J. Nielsen. *Nielsen Norman Group, First Rule of Usability? Don't Listen to Users*. https://www.nngroup.com/articles/first-rule-of-usability-dont-listen-to-users/. Accessed: 2019-05-28.

[90] J. Nielsen and J. Levy. „Measuring usability: preference vs. performance“. In: *Communications of the ACM* 37.4 (1994), pp. 66–75.

[91] *Microsoft Developer Network, Guidelines (Desktop)*. https://msdn.microsoft.com/en-us/library/windows/desktop/dn688964(v=vs.85).aspx. Accessed: 2019-05-28.

[92] A. Developer. „iOS Human Interface Guidelines“. In: *User Experience Documentation* (2012).

[93] *Android Developers, User Interface Guidelines*. https://developer.android.com/guide/practices/ui_guidelines/index.html. Accessed: 2019-05-28.

[94] *Microsoft Developer Network, Guidelines (Windows Mobile 6.5)*. https://msdn.microsoft.com/en-us/library/bb158602.aspx. Accessed: 2019-05-28.

[95] I. DIS. „9241-151: 2008. Ergonomics of human system interaction-Part 151: Guidance on World Wide Web user interfaces“. In: *International Standardization Organization (ISO). Switzerland* (2008).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[96] T. Mahatody, M. Sagar, and C. Kolski. „State of the art on the cognitive walkthrough method, its variants and evolutions“. In: *Intl. Journal of Human–Computer Interaction* 26.8 (2010), pp. 741–785.

[97] J. Nielsen. *Nielsen Norman Group, How to Conduct a Heuristic Evaluation*. https://www.nngroup.com/articles/how-to-conduct-a-heuristic-evaluation/. Accessed: 2019-05-28.

[98] J. Nielsen. „Finding usability problems through heuristic evaluation“. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1992, pp. 373–380.

[99] J. Nielsen. „Usability inspection methods“. In: *Conference companion on Human factors in computing systems*. ACM. 1994, pp. 413–414.

[100] D. Wixon et al. „Inspections and design reviews: framework, history and reflection“. In: *Usability inspection methods*. John Wiley & Sons, Inc. 1994, pp. 77–103.

[101] T. Hollingsed and D. G. Novick. „Usability inspection methods after 15 years of research and practice“. In: *Proceedings of the 25th annual ACM international conference on Design of communication*. ACM. 2007, pp. 249–255.

[102] J. Nielsen. *Nielsen Norman Group, 10 Best Intranets of 2005*. https://www.nngroup.com/articles/10-best-intranets-of-2005/. Accessed: 2019-05-28.

[103] K. Pernice, A. Schade, and P. Caya. *Nielsen Norman Group, 10 Best Intranets of 2016*. https://www.nngroup.com/articles/intranet-design-2016/. Accessed: 2019-05-28.

[104] K. Pernice, A. Schade, and P. Caya. *Nielsen Norman Group, 10 Best Intranets of 2017*. https://www.nngroup.com/articles/intranet-design/. Accessed: 2019-05-28.

[105] G. de Haan. „An ETAG based approach to the design of user interfaces“. In: *Proceedings of the 15th Interdisciplinary Workshop on Informatics and Psychology, Interdisciplinary Approaches to System Analysis and Design (Schaerding, 1994)*. 1994.

[106] B. Crandall, G. A. Klein, and R. R. Hoffman. *Working minds: A practitioner's guide to cognitive task analysis*. Mit Press, 2006.

[107] J. Nielsen. *Nielsen Norman Group, Deceivingly Strong Information Scent Costs Sales*. https://www.nngroup.com/articles/wrong-information-scent-costs-sales/. Accessed: 2019-05-28.

[108] J. Grigera et al. „Automatic detection of usability smells in web applications“. In: *International Journal of Human-Computer Studies* 97 (2017), pp. 129–148.

[110] D. Bacic. „Understanding Business Dashboard Design User Impact: Triangulation Approach Using Eye-Tracking, Facial Expression, Galvanic Skin Response and EEG Sensors“. In: (2017).

[111] M. Rauterberg. „A Petri net based analyzing and modeling tool kit for logfiles in humancomputer interaction“. In: *Proceedings of Cognitive Systems Engineering in Process Control* (1996), pp. 268–275.

[113] M. I. Jordan and T. M. Mitchell. „Machine learning: Trends, perspectives, and prospects“. In: *Science* 349.6245 (2015), pp. 255–260.

[114] A. Oztekin et al. „A machine learning-based usability evaluation method for eLearning systems“. In: *Decision Support Systems* 56 (2013), pp. 63–73.

[115] C. Korvald, E. Kim, and H. Reza. „Evaluation and implementation of machine learning techniques in usability testing for web sites“. In: (2014).

[116] A. Oztekin. „A decision support system for usability evaluation of web-based information systems“. In: *Expert Systems with Applications* 38.3 (2011), pp. 2110–2118.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[117] E. García et al. „Machine Learning Techniques in Usability-Evaluation Questionnaire Systems“. In: *Proceedings of 2nd International Con-d ference Learning ICDL* 2 (2003).

[118] J. Rieman et al. „An automated cognitive walkthrough“. In: *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM. 1991, pp. 427–428.

[119] J. Nielsen. *Nielsen Norman Group, Response-times 3 Important Limits*. https://www.nngroup.com/articles/response-times-3-important-limits/. Accessed: 2019-05-28.

[120] T. O. Aydın, A. Smolic, and M. Gross. „Automated aesthetic analysis of photographic images“. In: *IEEE transactions on visualization and computer graphics* 21.1 (2015), pp. 31–42.

[123] C. Kolski and J. Vanderdonckt. *Computer-Aided Design of User Interfaces III: Proceedings of the Fourth International Conference on Computer-Aided Design of User Interfaces 15–17 May 2002, Valenciennes, France*. Springer Science & Business Media, 2012, pp. 280–286.

[124] A. Sears. „A step toward metric-based interface development tools“. In: *Proceedings of UIST-95*. 1995, pp. 101–110.

[125] R. Mahajan and B. Shneiderman. „Visual and textual consistency checking tools for graphical user interfaces“. In: *IEEE Transactions on Software Engineering* 23.11 (1997), pp. 722–735.

[126] J. Xu et al. „A pilot study of an inspection framework for automated usability guideline reviews of mobile health applications“. In: *Proceedings of the Wireless Health 2014 on National Institutes of Health*. ACM. 2014, pp. 1–8.

[127] A. Sivaji, S.-T. Soo, and M. R. Abdullah. „Enhancing the effectiveness of usability evaluation by automated heuristic evaluation system“. In: *Computational Intelligence, Communication Systems and Networks (CICSyN), 2011 Third International Conference on*. IEEE. 2011, pp. 48–53.

[128] C. K. Coursaris and W. van Osch. „A Cognitive-Affective Model of Perceived User Satisfaction (CAMPUS): The complementary effects and interdependence of usability and aesthetics in IS design“. In: *Information & Management* 53.2 (2016), pp. 252–264.

[129] N. Hurtado et al. „Using simulation to aid decision making in managing the usability evaluation process“. In: *Information and Software Technology* 57 (2015), pp. 509–526.

[130] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010, pp. 26–31.

[131] A. Sears. „AIDE: A step toward metric-based interface development tools“. In: *Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM. 1995, pp. 101–110.

[132] W. O. Galitz. *The essential guide to user interface design: an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

[133] J. C. Silva, J. C. Campos, and J. A. Saraiva. „GUI inspection from source code analysis“. In: *Electronic Communications of the EASST* 33 (2010).

[134] W. Lidwell, K. Holden, and J. Butler. *Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design*. Rockport Pub, 2010, pp. 20–21.

[156] J. Herschmann and T. Murphy. „Gartner Magic Quadrant for Software Test Automation“. In: (2016).

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[157] D. Lo Giudice and C. Mines. „The Forrester Wave: Modern Application Functional Test Automation Tools, Q4 2016“. In: *The 11 Providers That Matter Most And How They Stack Up* (2016).

[158] K. Shaukat et al. „Taxonomy of Automated Software Testing Tools“. In: *International Journal of Computer Science and Innovation* 1 (2015), pp. 7–18.

[163] M. Zen and J. Vanderdonckt. „Towards an evaluation of graphical user interfaces aesthetics based on metrics“. In: *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*. IEEE. 2014, pp. 1–12.

[179] D. E. Kieras et al. „GLEAN: A computer-based tool for rapid GOMS model usability evaluation of user interface designs“. In: *Proceedings of the 8th annual ACM symposium on User interface and software technology*. ACM. 1995, pp. 91–100.

[180] R. S. Kalawsky. „VRUSE—a computerised diagnostic tool: for usability evaluation of virtual/synthetic environment systems“. In: *Applied ergonomics* 30.1 (1999), pp. 11–25.

[181] K. E. Schmidt, Y. Liu, and S. Sridharan. „Webpage aesthetics, performance and usability: Design variables and their effects“. In: *Ergonomics* 52.6 (2009), pp. 631–643.

[183] R. Inostroza et al. „Usability heuristics for touchscreen-based mobile devices: update“. In: *Proceedings of the 2013 Chilean Conference on Human-Computer Interaction*. ACM. 2013, pp. 24–29.

[184] A. Alsumait and A. Al-Osaimi. „Usability heuristics evaluation for child e-learning applications“. In: *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*. ACM. 2009, pp. 425–430.

[185] M. Candi. „Design as an element of innovation: Evaluating design emphasis in technology-based firms“. In: *International Journal of Innovation Management* 10.04 (2006), pp. 351–374.

[186] V. Brophy and J. O. Lewis. *A green vitruvius: principles and practice of sustainable architectural design*. Routledge, 2011.

[187] N. Tractinsky and M. Hassenzahl. „Arguing for aesthetics in human-computer interaction“. In: *I-com* 4.3/2005 (2005), pp. 66–68.

[188] M. R. Lehto and S. J. Landry. *Introduction to human factors and ergonomics for engineers*. Crc Press, 2012.

[189] W. Karwowski. *International encyclopedia of ergonomics and human factors*. Vol. 3. Crc Press, 2001.

[190] G. Salvendy. *Handbook of human factors and ergonomics*. John Wiley & Sons, 2012.

[191] J. Nielsen and R. Molich. „Heuristic evaluation of user interfaces“. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM. 1990, pp. 249–256.

[192] J. Nielsen. *Nielsen Norman Group, 113 design guidelines for homepage usability*. http://www.nngroup.com/articles/113-design-guidelines-homepage-usability/. Accessed: 2019-05-28.

[193] D. Travis. *UserFocus, 247 web usability guidelines*. https://www.userfocus.co.uk/resources/guidelines.html. Accessed: 2019-05-28.

[194] B. Shneiderman et al. *Designing the user interface: strategies for effective human-computer interaction*. Pearson, 2016.

[195] P. Zaphiris, M. Ghiawadwala, and S. Mughal. „Age-centered research-based web design guidelines“. In: *CHI'05 extended abstracts on Human factors in computing systems*. ACM. 2005, pp. 1897–1900.

TU Bibliothek, Your knowledge hub. WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[196] C. Rusu et al. „A methodology to establish usability heuristics“. In: *Proc. 4th International Conferences on Advances in Computer-Human Interactions (ACHI 2011), IARIA*. 2011, pp. 59–62.

[197] J. Nielsen. *Nielsen Norman Group, Users’ Pagination Preferences and "View All"*. https://www.nngroup.com/articles/item-list-view-all/. Accessed: 2019-05-28.

[198] S. Parker, S. Odio, and A. Mosseri. *Infinite Scrolling*. US Patent App. 12/833,901. 2010.

[199] A. J. Van Deursen and J. A. Van Dijk. „Using the Internet: Skill related problems in users’ online behavior“. In: *Interacting with computers* 21.5-6 (2009), pp. 393–402.

[200] D. Lobo et al. „Web usability guidelines for smartphones: a synergic approach“. In: *International journal of information and electronics engineering* 1.1 (2011), p. 33.

[201] J. Grudin. „The case against user interface consistency“. In: *Communications of the ACM* 32.10 (1989), pp. 1164–1173.

[202] M. S. R. Net. *Desktop Operating System Market Share, Market Share Statistics for Internet Technologies*. http://www.netmarketshare.com/. Accessed: 2019-05-28.

[203] J. Nielsen. *Nielsen Norman Group, Sixty Guidelines From 1986 Revisited*. https://www.nngroup.com/articles/sixty-guidelines-from-1986-revisited/. Accessed: 2019-05-28.

[204] J. Nielsen. *Nielsen Norman Group, Durability of Usability Guidelines*. https://www.nngroup.com/articles/durability-of-usability-guidelines/. Accessed: 2019-05-28.

[205] R. B. Miller. „Response time in man-computer conversational transactions“. In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*. ACM. 1968, pp. 267–277.

[207] S. Draxler. „The appropriation of a software ecosystem: a practice take on the usage, maintenance and modification of the eclipse IDE“. In: (2015).

[208] E. Hewlett Packard. *Unified Functional Testing Add-ins Guide, Software Version 14.03*. Micros Focus, 2018.

[210] E. Hewlett Packard. *Unified Functional Testing Tutorial, Software Version 14.03*. Micros Focus, 2018.

[211] S. Tarun et al. „Keyword Based Testing of Windows Application“. In: *Biometrics and Bioinformatics* 8.4 (2016), pp. 92–96.

[213] T. Böttger et al. „Open connect everywhere: A glimpse at the internet ecosystem through the lens of the netflix cdn“. In: *ACM SIGCOMM Computer Communication Review* 48.1 (2018), pp. 28–34.

Tool-Related Web References

[1] A. Banaouas. *GitHub, Complete Source Code for a Practical Experiment on Automated Heuristic Evaluation with GUI Event Sequencing*. Accessed: 2019-05-28. URL: \url{https://github.com/Amir-DA/automated-usability-inspection}.

[4] *SeleniumHQ, Browser Automation*. Accessed: 2019-05-27. URL: https://www.seleniumhq.org.

[5] *Ranorex, Test Automation for GUI Testing*. Accessed: 2019-05-27. URL: https://www.ranorex.com.

[6] *Android Studio, The Official IDE for Android*. Accessed: 2019-05-27. URL: https://developer.android.com/studio/index.html.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[109] *UserTesing, Usability testing from UserTesting*. Accessed: 2019-05-28. URL: https://www.usertesting.com.

[112] M. Rauterberg. *AMME, Home page of AMME*. Accessed: 2019-05-28. URL: http://www.idemployee.id.tue.nl/g.w.m.rauterberg/amme.html.

[121] *Varvy, Varvy SEO tool and optimization guide*. Accessed: 2019-05-28. URL: https://varvy.com/.

[122] *Cryptzone, Cynthia Says Portal*. Accessed: 2019-05-28. URL: http://www.cynthiasays.com/.

[135] *Zeenyx, What is AscentialTest?* Accessed: 2019-05-27. URL: https://www.zeenyx.com/AscentialTest.html.

[136] *Auto It Script, AutoIt*. Accessed: 2019-05-27. URL: https://www.autoitscript.com/site/autoit.

[137] *DOJO , Dojo Toolkit*. Accessed: 2019-05-27. URL: https://dojotoolkit.org/.

[138] *EggPlant , Delivering True Test Automation*. Accessed: 2019-05-27. URL: https://eggplant.io/.

[139] *iMacros , Web Browser Scripting Data Extraction and Web Testing An ipswitch project*. Accessed: 2019-05-27. URL: https://imacros.net/.

[140] *FreeDesktop.org, lcdp*. Accessed: 2019-05-27. URL: https://ldtp.freedesktop.org/wiki/.

[141] *Marveryx, Test it simple*. Accessed: 2019-05-27. URL: https://www.maveryx.com/.

[142] *Oracle Entreprise Manager, Application Testing Suite*. Accessed: 2019-05-27. URL: https://www.oracle.com/technetwork/oem/app-test/index.html.

[143] *QF-Test is Agile, GUI Testing Tool for Java and the Web*. Accessed: 2019-05-27. URL: https://www.qfs.de/en.html.

[144] *IBM, Rational Functional Tester*. Accessed: 2019-05-27. URL: https://www.ibm.com/us-en/marketplace/rational-functional-tester.

[145] *Sahi, Automation Testing Tool for Web Applications*. Accessed: 2019-05-27. URL: https://sahipro.com/.

[146] *Micro Focus, Silk Test*. Accessed: 2019-05-27. URL: https://www.microfocus.com/de-de/products/silk-portfolio/silk-test/.

[147] *Parasoft , Web UI Testing*. Accessed: 2019-05-27. URL: https://www.parasoft.com/capability/web-ui-testing/.

[148] *Squish GUI Tester, Automated GUI Testing Tool*. Accessed: 2019-05-27. URL: https://www.froglogic.com/squish/.

[149] *Telerik Test Studio , Software Testing Tools*. Accessed: 2019-05-27. URL: https://www.telerik.com/teststudio.

[150] *Test Complete , Automated Software Testing Made Simple*. Accessed: 2019-05-27. URL: https://smartbear.com/product/testcomplete/overview/.

[151] *Tricentis, Tosca Automate UI*. Accessed: 2019-05-27. URL: https://www.tricentis.com/resource-assets/tosca-automate-ui/.

[152] *Unified Functional Testing , Automated Testing Software*. Accessed: 2019-05-27. URL: https://www.microfocus.com/en-us/products/unified-functional-automated-testing/overview.

[153] *Visual Studio Test Professional, Software testing for professional*. Accessed: 2019-05-27. URL: https://visualstudio.microsoft.com/de/vs/test-professional/.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[154] *Watir Project, Web Application Testing in Ruby*. Accessed: 2019-05-27. URL: http://watir.com/.

[155] *GNU Xnee, Record and Replay for Linux and Other X11 Based Systems*. Accessed: 2019-05-27. URL: https://xnee.wordpress.com/.

[159] *MaTeLo integrations , MaTeLo Plugin by All4Tech*. Accessed: 2019-05-28. URL: http://www.all4tec.com/matelo.

[160] *Unified Functional Testing Features, Automated Testing Software*. Accessed: 2019-05-28. URL: https://software.microfocus.com/en-us/software/uft/features.

[161] *Applitools, Automated Visual Testing*. Accessed: 2019-05-28. URL: https://applitools.com/.

[162] *SikuliX, Automated Anything You See*. Accessed: 2019-05-28. URL: http://sikulix.com/.

[164] *Appsee, Understand Mobile App Analytics*. Accessed: 2019-05-28. URL: https://www.appsee.com/.

[165] *Attensee, First Impression Testing for UX Desingers*. Accessed: 2019-05-28. URL: http://www.attensee.com/.

[166] *CrowdSignal , Online Survey Software*. Accessed: 2019-05-28. URL: https://crowdsignal.com/.

[167] *Helio , Rapidly Revealing Key User Behaviors*. Accessed: 2019-05-28. URL: https://zurb.com/helio/.

[168] *LookBack, Simple and Powerful User Research*. Accessed: 2019-05-28. URL: https://lookback.io/.

[169] *Loop 11, Online User Testing Tool*. Accessed: 2019-05-28. URL: https://www.loop11.com/.

[170] *TechSmith, Usability Testing with Morae*. Accessed: 2019-05-28. URL: https://www.techsmith.com/morae.html.

[171] *MouseStats , UX Analysis Suite*. Accessed: 2019-05-28. URL: https://www.mousestats.com/.

[172] *Naview , Create easier Navigation through Prototyping and Testing*. Accessed: 2019-05-28. URL: https://www.naviewapp.com/.

[173] *Silverback , Guerrilla Usability Testing on the Mac Made Easy*. Accessed: 2019-05-28. URL: https://silverbackapp.com/.

[174] *TryMyUI , Website Usability Testing*. Accessed: 2019-05-28. URL: https://www.trymyui.com/.

[175] *Usabilla , The Standard User Feedback*. Accessed: 2019-05-28. URL: https://usabilla.com/.

[176] *WhatUsersDo , Usability and User Testing*. Accessed: 2019-05-28. URL: https://www.whatusersdo.com/.

[177] *Woopra, Real Time Customer Analytics*. Accessed: 2019-05-28. URL: https://www.woopra.com/.

[178] *Marae , Usability Testing with Marae by TechSmith*. Accessed: 2019-05-28. URL: https://www.techsmith.com/morae.html.

[182] *GTmetrix, Website Speed and Performance Optimization*. Accessed: 2019-05-27. URL: https://gtmetrix.com/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[206] *Eclipse , Enabling Open Innovation and Collaboration*. Accessed: 2019-05-28. URL: https://www.eclipse.org/.

[209] License. *GNU General Public, Free software foundation*. Accessed: 2019-05-28.

[212] *Steam , Welcome to Steam Buy once Play everywhere*. Accessed: 2019-05-28. URL: https://store.steampowered.com/.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A Additional UFT Flow Diagrams of automated heuristic evaluation instances

When it comes to the practical test instances evaluating heuristics (as described in section 5.3.2). Some additional UFT flow diagrams for those guidelines are presented in this appendix. Every guideline whose flow diagram was non included in section 5.3.2 is offered in this appendix in the same order as the heuristics they correspond to. The figures can be viewed below:

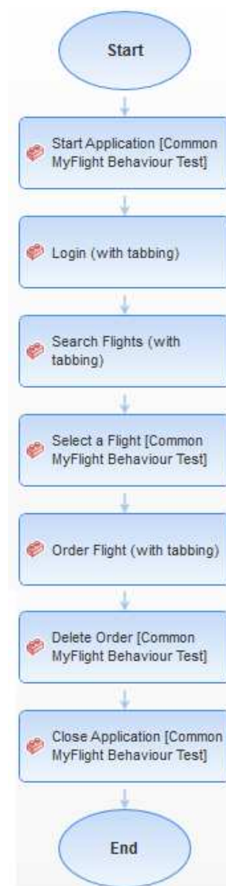

Figure A.1: UFT flow diagram for guideline 1.4/15 "Explicit Tabbing to Data Fields"

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Figure A.2: UFT flow diagram for guideline 1.7/3 "Non-Disruptive Error Messages"

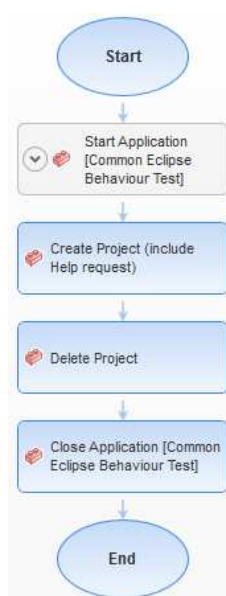

Figure A.3: UFT flow diagram for guideline 3.0/1 "Flexible Sequence Control"

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure A.4: UFT flow diagram for guideline 3.1.3/7 "Menu Selection by Keyed Entry"

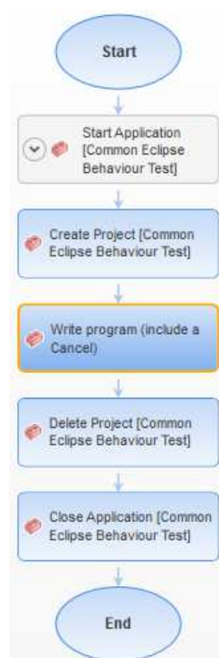

Figure A.5: UFT flow diagram for guideline 3.3/3 "Cancel Option"

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Figure A.6: UFT flow diagram for guideline 3.5/10 "UNDO to Reverse Control Actions"

Figure A.7: UFT flow diagram for guideline 6.0/18 "User Confirmation of Destructive Actions"

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.