Towards Collaborative GUI-based Testing

Andreas Bauer

Department of Software Engineering
Blekinge Institute of Technology

Licentiate Dissertation Series no. 2023:10
Towards Collaborative GUI-based Testing

Andreas Bauer
Towards Collaborative GUI-based Testing

Andreas Bauer

Licentiate Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN
Abstract

Context: Contemporary software development is a socio-technical activity requiring extensive collaboration among individuals with diverse expertise. Software testing is an integral part of software development that also depends on various expertise. GUI-based testing allows assessing a system’s GUI and its behavior through its graphical user interface. Collaborative practices in software development, like code reviews, not only improve software quality but also promote knowledge exchange within teams. Similar benefits could be extended to other areas of software engineering, such as GUI-based testing. However, collaborative practices for GUI-based testing necessitate a unique approach since general software development practices, perceivably, can not be directly transferred to software testing.

Goal: This thesis contributes towards a tool-supported approach enabling collaborative GUI-based testing. Our distinct goals are (1) to identify processes and guidelines to enable collaboration on GUI-based testing artifacts and (2) to operationalize tool support to aid this collaboration.

Method: We conducted a systematic literature review identifying code review guidelines for GUI-based testing. Further, we conducted a controlled experiment to assess the efficiency and potential usability issues of Augmented Testing.

Results: We provided guidelines for reviewing GUI-based testing artifacts, which aid contributors and reviewers during code reviews. We further provide empirical evidence that Augmented Testing is not only an efficient approach to GUI-based testing but also usable for non-technical users, making it a promising subject for further research in collaborative GUI-based testing.

Conclusion: Code review guidelines aid collaboration through discussions, and a suitable testing approach can serve as a platform to operationalize collaboration. Collaborative GUI-based testing has the potential to improve the efficiency and effectiveness of such testing.
Acknowledgements

First and foremost, I am particularly grateful to my supervisor Emil Alégroth, for his invaluable expertise and continuous support throughout this thesis. I would also like to thank my other supervisors, Mikael Svalhberg and Tony Gorschek.

Furthermore, I would also like to thank my many colleagues who enriched my work and made it an enjoyable journey. In particular, Waleed Abdeen, Michael Dorner, Ehsan Zabardast, Julian Frattini, Davide Fucci, Umar Iftikhar, Daniel Mendez, and Svetlana Zivanovic.

Finally, I would like to thank my family for their support.
Overview of Papers

Publications in this Thesis


Contribution Statement

We used CRediT (Contributor Roles Taxonomy) [11] to lay out the individual author contributions to the included papers.

Chapter 2:

- *Andreas Bauer*: Conceptualization (lead), Methodology (lead), Software, Investigation (lead), Writing – original draft (lead), Writing – review and editing (equal), Visualization (lead), Project administration (lead)
• *Emil Alégroth*: Conceptualization (supporting), Methodology (supporting), Writing – review and editing (equal), Supervision (lead)

Chapter 3:

• *Andreas Bauer*: Conceptualization (lead), Methodology (equal), Formal analysis (lead), Investigation (equal), Data curation (lead), Writing – original draft (equal), Project administration (lead)

• *Riccardo Coppola*: Conceptualization (supporting), Methodology (equal), Investigation (equal), Data curation (supporting), Writing – original draft (supporting)

• *Emil Alégroth*: Conceptualization (supporting), Methodology (supporting), Writing – original draft (equal), Writing – review and editing (supporting), Supervision (lead)

• *Tony Gorschek*: Writing – review and editing (supporting), Supervision (supporting), Funding acquisition (lead)

Chapter 4:

• *Andreas Bauer*: Conceptualization (lead), Methodology (lead), Software (equal), Formal analysis (equal), Data curation (lead), Writing – original draft (lead), Writing – review and editing (lead), Visualization (lead), Project administration (lead)

• *Julian Frattini*: Software (equal), Formal analysis (equal), Writing – original draft (supporting), Writing – review and editing (supporting), Visualization (supporting)

• *Emil Alégroth*: Conceptualization (supporting), Methodology (supporting), Resources (lead), Writing – review and editing (supporting), Supervision (lead)

Other Publications not in this Thesis


Funding

We would like to acknowledge that this work was supported by the KKS foundation through the SERT Research Profile project (research profile grant 2018/010) at Blekinge Institute of Technology.
# Contents

Abstract v

Acknowledgements vii

Overview of Publications ix
  Publications in this Thesis xi
  Other Publications not in this Thesis x

List of Abbreviations xv

1 Introduction 1
  1.1 Overview 1
  1.2 Background 3
  1.3 Related Work 6
  1.4 Research Gaps and Questions 8
  1.5 Research Methodology 16
  1.6 Overview of Chapters 20
  1.7 Synthesis 23
  1.8 Future Work 24
  1.9 Conclusion 27

2 We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing 29
  2.1 Introduction 30
  2.2 Background 32
  2.3 Results 36
  2.4 Challenges 44
  2.5 Conclusion 47
# CONTENTS

3 Code Review Guidelines for GUI-based Testing Artifacts 49
  3.1 Introduction ......................................................... 50
  3.2 Background and Related Work ................................... 52
  3.3 Study Design ......................................................... 56
  3.4 Results ................................................................. 68
  3.5 Discussion ............................................................ 92
  3.6 Conclusion ............................................................ 96

4 Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study 105
  4.1 Introduction .......................................................... 106
  4.2 Related work ........................................................ 109
  4.3 Method ................................................................. 111
  4.4 Results ................................................................. 127
  4.5 Discussion ............................................................ 144
  4.6 Threats to validity ............................................... 147
  4.7 Conclusion ............................................................ 149

References 151
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT</td>
<td>Augmented Testing</td>
</tr>
<tr>
<td>BDA</td>
<td>Bayesian Data Analysis</td>
</tr>
<tr>
<td>CSP</td>
<td>Collaborative Software Process</td>
</tr>
<tr>
<td>CST</td>
<td>Crowdsourced Software Testing</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>E2E</td>
<td>End to End</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MCR</td>
<td>Modern Code Review</td>
</tr>
<tr>
<td>SUT</td>
<td>System Under Test</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>VGT</td>
<td>Visual GUI Testing</td>
</tr>
<tr>
<td>XP</td>
<td>Extreme Programming</td>
</tr>
</tbody>
</table>
## List of Figures

1.1 Demonstration of Augmented Testing (AT). The SUT’s widgets are superimposed with information through the augmentation layer.  
1.2 Goals, gaps, and research questions  
1.3 Proposed future work and their relationship to studies included in this thesis  
2.1 Demonstration of Augmented Testing (AT). The SUT’s widgets are superimposed with information through the augmentation layer. Green rectangles represent valid checks and blue rectangles actions. In the middle, a graph that represents a path through the GUI is presented  
2.2 Using a merge request to introduce changes to the main repository  
2.3 Collaborative workflow for GUI-based tests  
2.4 Approach towards a collaborative GUI-based testing workflow  
2.5 Example of a code review discussion on GitHub. Comments can be attached to specific code fragments  
2.6 Example of a simple merge scenario of two test cases  
3.1 Summary of the research approach  
3.2 White and gray literature publications per year  
4.1 Demonstration of Augmented Testing (AT). The SUT’s widgets are superimposed with information through the augmentation layer.  
4.2 Screenshot of SUT  
4.3 Three examples of augmentation superimposing the SUT’s GUI  
4.4 Overall test duration of all test cases per subject, measured in seconds
4.5  Distribution of test duration per test case (both treatments combined) .................................................. 129
4.6  Distribution of test duration per test case with a separation of treatments .............................................. 130
4.7  Directed acyclic graph (DAG) ................................................. 130
4.8  Visualization of the interaction between the two predictor variables 132
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Constellations of changes for one GUI state and how they are handled in the demonstrator</td>
<td>46</td>
</tr>
<tr>
<td>3.1</td>
<td>Search results by library</td>
<td>61</td>
</tr>
<tr>
<td>3.2</td>
<td>Quality assessment questionnaire for gray literature. We use a 3-point Likert scale (yes=1, partly=0.5, and no=0) for the authority and outlet criteria</td>
<td>65</td>
</tr>
<tr>
<td>3.3</td>
<td>Overview of code review guidelines and their mapping to GUI testing</td>
<td>73</td>
</tr>
<tr>
<td>3.4</td>
<td>Overview of code review guidelines used for source and test artifact review (WL=white literature, GL=gray literature, TS=test specific)</td>
<td>90</td>
</tr>
<tr>
<td>3.5</td>
<td>Categories of code review guidelines used for source and test artifact review (WL=white literature, GL=gray literature)</td>
<td>91</td>
</tr>
<tr>
<td>3.6</td>
<td>White literature sources where guidelines were taken</td>
<td>98</td>
</tr>
<tr>
<td>3.7</td>
<td>Gray literature sources where guidelines were taken</td>
<td>102</td>
</tr>
<tr>
<td>4.1</td>
<td>Application of GQM to this experiment</td>
<td>113</td>
</tr>
<tr>
<td>4.2</td>
<td>Description of independent (ind) and dependent (dep) variables</td>
<td>115</td>
</tr>
<tr>
<td>4.3</td>
<td>Professional experience of participants in industry and mapping to their companies</td>
<td>117</td>
</tr>
<tr>
<td>4.4</td>
<td>Assignment of treatments to participants. Subjects get both treatments in a randomized order (M = Manual Testing, A = Augmented Testing)</td>
<td>126</td>
</tr>
<tr>
<td>4.5</td>
<td>Mean time to complete each test case per treatment, measured in seconds (M = manual GUI-based regression testing, A = Augmented Testing)</td>
<td>131</td>
</tr>
</tbody>
</table>
4.6 Feedback regarding AT’s benefits and drawbacks from interview questions ................................. 141
Chapter 1

Introduction

1.1 Overview

Software development today is a socio-technical activity that demands extensive collaboration among professionals with diverse technical skills and domain knowledge to cope with increasing system complexities [35, 123, 185]. In addition to the increasing complexity of software systems, software teams are expected to deliver software faster to the market, in a continuous way, and with high quality. This software development process hinges on effective team communication and coordination. Software development includes activities such as requirements engineering, modeling, development, and testing. The significance of software testing is widely accepted [44, 46, 80]. Like software development, software testing requires technical and non-technical skills [81]. Testing can be categorized by how it is performed, either automated or manual [45]. Depending on the target (i.e., function, component, subsystem, or system), tests can also be classified based on the level of abstraction, such as unit, integration, or system testing [25]. System testing evaluates the system’s holistic behavior and can be conducted through GUI-based testing, allowing the assessment of a system’s GUI and its behavior through its GUI.

In software development, collaboration among professionals is facilitated through the adoption of processes and supported by tools [53], enabling developers to coordinate and integrate software artifact contributions into one system. Collaborative software development platforms, such as GitHub [109] or GitLab [110], have centralized various tools and processes, simplifying their use.
Agile methodologies, which have gained traction in the industry, underscore the usage of collaborative practices [174, 176]. For instance, SCRUM introduces team-wide practices to involve and inform the whole development team, such as sprint reviews and retrospectives [175]. The KANBAN method also includes practices enabling the improvement of a software product collaboratively [178]. Additionally, in extreme programming (XP), the practice of team code ownership empowers every developer to modify any part of the team’s code [177].

A commonly team-centric activity is code review, wherein peers evaluate code changes prior to integration into the main codebase [12, 23, 79]. Although the primary goal of reviewing artifacts is to enhance quality by identifying errors, it also serves as an activity for developers to exchange technical and domain knowledge [12, 39]. The discussions that emerge during the review of artifacts between the reviewer and the contributor of an artifact not only inform team members about modifications to the system but also foster a common understanding and collective responsibility for it. Hence, collaborative practices like code review offer benefits beyond quality improvements [12, 39].

Collaborative practices in software development, like code reviews, have been shown to improve development efficiency and software quality [12, 17, 23, 39, 79, 173]. Thus, it is assumed similar benefits could be extended to other areas of software engineering, such as software testing. Especially when people with domain and technical knowledge that have different levels of experience could exchange information in an effective way. However, Spadini et al. [83] showed that test code is far less reviewed than source code. Discussions during test code review often pivot to topics like code coverage and assertion handling. This emphasizes that test code reviews differ from those of source code, requiring a distinct approach. The study analyzed test code in three projects but did not cover GUI-based tests, leaving a gap in knowledge.

To the best of our knowledge, the literature lacks descriptions of collaborative tool-supported practices for GUI-based testing.

Software testing is an essential practice in software engineering, and GUI-based testing is a common part of it. The lack of guidelines on how to facilitate collaboration between different people with different skill sets on GUI-based testing artifacts leads to ad hoc practices. These ad hoc practices may lack rigor and thus affect the quality of testing artifacts. In consequence, poor test artifact quality leads to increased maintenance issues and costs [92, 93, 94]. Moreover, the reusability of ad hoc practices across different software teams, companies, or domains can be infeasible. Alternatively, if applicable, the practices are inefficient, resulting in further costs. Hence, these ad hoc practices present a
1.2 Background

challenge for companies incorporating GUI-based testing as part of their quality assurance strategy.

To mitigate the lack of rigorous collaborative practices in the context of GUI-based testing, this thesis contributes to the overarching goal of a tool-supported approach enabling collaborative GUI-based testing. The aim is to improve the efficiency and effectiveness of creating and maintaining GUI-based tests through collaborative efforts. Efficiency could be improved by coordinating and reusing tests, or test steps, preventing redundant work. Sharing both technical and domain knowledge within a team would allow less experienced people, or those lacking knowledge in a specific topic, to learn from others within the team. For instance, better testing strategies or insights into the domain can be shared to improve the quality of GUI-based testing artifacts. Such improvements in efficiency and quality can decrease the costs associated with GUI-based testing, making it more accessible for industry adoption.

To guide our research efforts, we formulated two distinct goal statements that contribute to the overarching goal of this thesis:

G1 Understanding: identifying approaches to enable collaboration on GUI-based testing artifacts

G2 Operationalizing: developing a tool to facilitate collaboration on GUI-based testing artifacts

The remainder of this chapter is structured as follows: Section 1.2 provides relevant background information. Section 1.3 presents related work. Section 1.4 describes the goals and its derived research question for our overarching goal of collaborative GUI-based testing. Section 1.5 outlines the research methods employed in this thesis. Section 1.6 presents an overview of the studies included in this thesis. Section 1.7 provides a synthesis of the results and outlines future research. Section 1.8 outlines possible future research. We conclude this chapter in Section 1.9.

1.2 Background

In this section, we provide definitions of the concepts of GUI-based testing, Augmented Testing, and modern code review that are used throughout this thesis.

For clarity, we use the following terminology (based on Zabardast et al. [184]): Source code refers to code written in a human-readable and comprehensible form that encompasses the project’s logic and operates in production. While
test code refers to test cases or automated scripts that validate the application’s functionality. We use the terms test artifact and test code interchangeably.

1.2.1 GUI-based testing

GUI-based testing refers to end-to-end (E2E) functional testing of a defined system under test (SUT) through its graphical user interface (GUI) [30]. In the realm of GUI-based testing, several classifications exist. We use the classification by Alégroth [3] where GUI-based testing methodologies are distinct in three chronological generations.

First-generation GUI testing employs a coordinate-based technique where the software testing tool interacts with the SUT using precise screen coordinates [5]. Coordinate-based tools utilize a capture-and-replay model, where coordinates from an initial human interaction are recorded and subsequently used to replay a regression test. The drawback of this technique is its high maintenance requirements due to its dependence on exact screen coordinates. Any modifications, such as a change in screen resolution, can potentially result in mismatched recorded coordinates. For this reason, coordinate-based techniques are not used in practice, although actions on coordinates are possible in most GUI testing tools.

Second-generation GUI testing adopts a component-based technique, which enables the testing tool to interact with the SUT by accessing its layout model or widget properties. In the context of web applications, this can be achieved by accessing the document object model (DOM). This approach proves to be more robust compared to coordinate-based techniques. However, a notable limitation is its requirement for access to the SUT libraries, thereby tying it to the technology utilized in the SUT. GUITAR is an academic example of a second-generation GUI testing tool [68]. Examples of tools that are used in practice are Selenium [95], Playwright [96], and Cypress [97].

Third-generation GUI testing employs an image recognition-based technique where the tool uses computer vision to identify on-screen widgets based on their visual appearance [27]. This technique involves comparing bitmaps of the targeted widgets with the regions currently displayed on the screen. Testing based on image recognition is also known as Visual GUI Testing (VGT) [4]. Examples of third-generation tools are SikuliX [90] and JAutomate [4], which is now called EyeAutomate.
1.2 Background

1.2.2 Augmented Testing

Augmented Testing (AT) is a novel GUI-based testing technique applicable to both manual and automated testing, with the aim of improving the tests’ effectiveness and efficiency. [65, 66, 67]. AT introduces an intermediate layer between the SUT’s GUI and the tester. This augmented layer overlays the GUI with test-relevant augmentations—augmentations are visual highlights that can be freely placed on the screen but primarily placed on widgets—to recommend user actions, show previous interactions, identify issues, show comments, and perform checks. Figure 1.1 demonstrates the augmented layer of AT. All GUI interactions are channeled through this augmented layer, allowing precise recording of user interactions into a state model. The most important types of augmentations provided by Scout are checks on widget properties, e.g., is the text value of the widget as expected, and actions to perform to progress within the test, e.g., enter a specific input value into the input widget or click on the next button. In Figure 1.1, checks are represented by green rectangles and actions as blue rectangles around the targeted widgets. AT is a tool-driven technique, which has been realized through multiple demonstrators, including the tool Scout [67]. Scout, developed in the programming language Java, is designed with a plugin architecture, enabling the addition of new augmentations and features through plugins. Due to Scout’s reliance on Selenium WebDriver [156], it is currently limited to web-based applications. In the classification by Alégroth [3], Scout represents a second-generation GUI-based testing tool.

In a previous study [66], AT with Scout was compared, via a quasi-experiment, against two state-of-practice GUI-based testing tools, Protractor [114] and Selenium [156]. Participants demonstrated increased efficiency in test case creation with AT compared to both Protractor and Selenium. Additionally, they reported that AT requires less programming and automation expertise than the other tools.
1.2.3 (Modern) code review

Fagan [118] described code inspections, an early form of code reviews, as a process where developers manually inspected other developers’ source code to improve its quality and verify their correctness [116, 117]. This form of code inspection was a formal waterfall-like process that resulted in additional overhead. Code inspection was demonstrated to be effective in identifying bugs and improving the source code [118], yet its adoption has been constrained by overhead caused by the highly structured process [129].

Code reviews, unlike Fagan’s code inspections, are characterized as informal, tool-supported, and lightweight processes, known as modern code review (MCR) [12, 22, 24, 74]. Moreover, this form of code review is widely adopted in the industry and open-source projects [12, 31, 179, 180]. Empirically evaluation demonstrated its effectiveness in improving code quality and sharing knowledge among developers [12, 39, 180]. To support the code review process, dedicated tools, such as Gerrit [107] or Phabricator [108] are used. Modern software platforms, such as GitHub [109] or GitLab [110], have later integrated code review functionality as a fundamental part of their platform, facilitating collaborative software development workflows.

1.3 Related Work

“any software project with more than one person is created through a process of collaborative software engineering” [189]
1.3 Related Work

Collaboration in software engineering manifests across various phases, such as requirements engineering, architecture design/modeling, and development/maintenance [189]. Beginning with requirements engineering, collaboration happens on the elicitation of requirements from stakeholders, e.g., describing use cases.

In the design phase, these requirements are transformed into architectural or detailed component models. The prevalent adoption of the unified modeling language (UML) means that documentation typically takes the form of UML diagrams facilitated by a range of UML modeling tools. Established collaborative practices, such as line-by-line comparisons of changes through version control systems (VCS), like Git, prove challenging, as UML’s graphical nature hinders understanding through its file representation [69, 70]. Given that version control is applied to the underlying file representation, absent of any graphical visualization, human comprehension of changes is hardly possible through version control tools. For example, changes in the structure of a graph due to added or deleted nodes can have substantial discrepancies between its graphical visualization and the underlying representation. The need for version control for software design models, or some form of history functionality, is recognized as significant in industrial collaborative software design [193]. Consequently, tools for collaborative UML design have introduced additional built-in methods to manage and graphically visualize changes instead of relying on external tools like Git, enabling multiple people to collaboratively work on UML models [193]. These integrated methods underscore the necessity for additional consideration when fostering collaboration on artifacts that are not strictly text-based and represent content at a higher level of abstraction.

Finally, the architecture and its associated models are then implemented as source and test code. Collaboration in development can occur synchronously through shared workspaces or asynchronously via a VCS [189]. Development methodologies like Extreme Programming (XP) incorporate collaborative practices such as pair programming [177]. In pair programming, two developers work concurrently on a single machine, with one coding while the other reviews in real-time.

Early on, software inspections used collaborative practices for quality assurance to facilitate the communication needs between different and distributed teams [190, 191]. This type of collaboration mostly builds on synchronous inspection activities, such as meetings. Asynchronous inspections are facilitated by sharing documents about test activities between testers.

The most related research domain to collaborative GUI-based testing is crowdsourced software testing (CST), which applies crowdsourcing—employing
an unspecified online group for voluntary tasks in an open call [14, 15]—–to software testing [58]. CST is a task-based activity, mainly performed by letting the crowd execute and examine test code concerning the resulting output for a given input. Access to the related source code is not provided to the crowd worker. CST does not aim for collaboration within teams, i.e., foster a shared understanding of the target system, as it primarily gathers individual solutions from unidentified contributors who do not know about other contributors. This misalignment makes CST and its specific practices not suitable as a template for collaborative practices within teams.

In a doctoral thesis by Williams [192], a collaborative software process (CSP) was developed, incorporating practices such as pair programming. The thesis demonstrated that CSP improves problem-solving skills, fosters better design, augments learning, and improves team-building for collaborative pairs in comparison to single developer practices. The thesis primarily addresses the collaborative practice of pair programming for software development. Regarding testing, reviews are facilitated by a checklist for collaborative workers. This checklist is of a general nature, focusing on ensuring the completeness and correct implementation of the design and verifying code conformity to standards. A discussion regarding the differentiation in handling low and high-level testing was not covered in the thesis.

In summary, although collaborative practices are established and have a positive effect on software engineering, their application to a specific area, like GUI-based testing, presents notable challenges. In particular, there is a lack of tool support that incorporates the graphical context of the GUI under test and practices distinctly tailored towards GUI-based testing. Therefore, this underscores the importance of our research efforts presented in this thesis.

1.4 Research Gaps and Questions

Figure 1.2 presents an overview of the relationship between this thesis's goals, gaps, and research questions.

Two distinct goals that drive our research efforts were introduced in Section 1. Each goal's outcome directly contributes to separate cornerstones of the overarching goal: a tool-supported approach enabling collaborative GUI-based testing. Achieving this goal demands, first, processes and guidelines for collaboration on GUI-based testing artifacts. Next, the application of these processes and guidelines should be supported by tools. Collaborative development requires tool support [53]. Hence, we need a platform that allows us to de-
velop adequate tools to support the collaborative development of GUI-based tests. Augmented Testing, implemented as the tool Scout, presents itself as a promising candidate for such a platform due to its testing efficiency, usability for non-developers, and availability. Nonetheless, to determine this with certainty, further evaluation of its usability and efficiency is required. By combining the insights from the two goals—the comprehension of collaborative processes and guidelines with the assessment of suitable GUI-based testing approaches—a more defined pathway emerges to achieve the overarching goal.

In the remainder of this section, we describe the research gaps we aim to bridge and the corresponding research questions to address the gaps. Bridging these research gaps contributes towards the two goals and, consequently, to the overarching goal. Each gap is elaborated in individual subsections related to the chapters of this thesis.
Introduction

Figure 1.2: Goals, gaps, and research questions
1.4 Research Gaps and Questions

1.4.1 Gap 1: Guidelines and processes for collaboration on GUI-based testing artifacts

Team collaboration can be performed in various manners, each with its advantages and disadvantages. For example, communication among people can be done synchronously or asynchronously [98], affecting collaboration activities distinctively. Synchronous communication is a real-time interaction among people, such as face-to-face meetings, phone calls, or video chats. In asynchronous communication, in contrast, interactions occur without immediate response, often through emails or messaging platforms. Synchronous communication necessitates upfront coordination among participants, while asynchronous communication takes place individually, interactions need to be coordinated throughout the communication process. Deciding on synchronous or asynchronous communication for collaboration would affect all subsequent steps in the selection of suitable guidelines, practices, and tool support. For example, collaboration in asynchronous form can be realized through version control systems (VCS) tools, whereas synchronous collaboration can be supported through shared workspaces and application sharing [181, 189].

When practitioners perform collaborative practices such as code review on test code, they discuss better testing practices, untested paths, or high-level testing issues [83]. Due to the absence of test-specific information and the missing context of the related source/test code in code review tools, testers often resort to their local development environments to review test code. The different information needs may require additional considerations and modifications when applying general software development practices to test code. Nevertheless, test code can contain defects, similar to source code, and would benefit from practices like code review [85]. The most frequent bug in test code is inconsistent synchronization, causing false alarms during test execution, known as flaky tests [85].

The literature does not cover collaborative practices for GUI-based testing [13, 44, 48], emphasizing a gap in knowledge. Crowdsourced software testing, as discussed in the previous sections, encompasses practices for GUI-based testing but is not suitable as a template for collaborative practices for teams.

A knowledge gap exists regarding collaborative practices in GUI-based testing, particularly regarding processes and tools. This uncertainty raises questions about the direction of future research to achieve our overarching goal. Processes are crucial to guide efficient collaboration, while tools should automate checks to reduce human effort. We address this gap by answering the following research question:
12 Introduction

RQ1: What are the processes and tools of a collaborative workflow for GUI-based testing artifacts?

1.4.2 Gap 2: Collaboration through discussions around artifact changes

Collaborative work is a shared effort to work on a task [182]. Apart from the actual collaboration (working on the task), it includes the characteristics of informing, coordinating, and cooperating (team goals overrule those of individuals). In software engineering, collaborative development methods are present for requirements engineering, software design and modeling, and software implementation and testing [53]. For software implementation and testing, collaboration is highly tool-oriented and primarily centers around source and test code.

As presented in Section 1.1, code reviews constitute such a collaborative practice centered on discussing artifact changes. For testing, code reviews allow experienced testers to improve the quality of the test and to share their knowledge about testing techniques and domain knowledge with less experienced colleagues. These reviews provide a forum to debate alternative solutions and enhance the quality of test artifacts through collective feedback. Particularly, contributions with complexity or substantial system impact benefit from discussions among individuals with both technical and domain knowledge. Finally, the informing characteristic of collaboration [182] is realized by sharing information without a designated recipient, as the information remains valuable for broader use. In other words, informing the whole team about changes and decisions.

Guidelines for code review assist both contributors and reviewers. For contributors, guidelines help to prepare a contribution in a way that is easy to comprehend. This can be done by reducing the cognitive effort to comprehend changes and providing all information supporting the comprehension of the change so that a reviewer does not have to spend additional time gathering the required information [71]. Examples of provided information are the rationale of the change, an explanation of the change’s impact, documentation for used functionality, and previous decisions of similar changes. For reviewers, these guidelines can be used as a checklist to ensure checking changes for specific aspects, such as the usage of established design principles and patterns, utilization of metrics, and the correct usage of assertions in test cases.
1.4 Research Gaps and Questions

Key studies in this field did not address guidelines tailored for GUI-based test artifacts. Most literature, such as Dong et al. [33], predominantly focuses on source and low-level test code.

Pascarella et al. [71] present a taxonomy of information needs in code reviews, identifying seven primary high-level information needs. These information needs can be interpreted as reverse guidelines from a reviewer’s standpoint. For instance, the need for a justification of the change can be interpreted as a guideline to provide a rationale for the change. Yet, its applicability to testing or GUI-based testing artifact review remains undiscussed.

Bacchelli and Bird [12] highlights developers’ motivations for code reviews and discusses expected outcomes compared to actual outcomes of code reviews. The primary motivations for developers were: finding bugs, code improvement, alternative solutions, knowledge transfer, and team awareness. All of these motivations can be assumed to be reasonable for testers working on GUI-based testing as well. Again, the applicability of these findings to GUI-based test artifacts is undiscussed.

Sadowski et al. [79] examined Google’s code review practices, underscoring the knowledge-sharing benefits of code reviews. Code reviews at Google were introduced to ensure code readability and maintainability. Solving problems and the identification of defects are not the main expectations of code reviews. Developers at Google perceive the educational aspect of code reviews as beneficial. Given the potential for source code review practices to offer similar benefits to test code, such as fostering knowledge sharing, further exploration into adapting these practices is necessary.

In contrast, Spadini et al. [83] investigated code reviews in the context of software testing. Their findings show that test code is less discussed during reviews and highlights reviewers’ distinct needs when reviewing test code changes. For instance, the main concern of reviewers reviewing test code is to understand whether the test covers all paths of the system’s source code and to ensure the maintainability of tests. Further, reviewing test code requires developers to also be aware of the related source code of the system under test. When discussing defects in test code, most of the comments are about high-level testing issues, whereas for source code, the comments are more about low-level concerns. This supports our assumption that GUI-based tests, due to their high level of abstraction and distinct requirements, might necessitate dedicated review guidelines.

Alegroth et al. [10] proposed guidelines for developing GUI-based tests with the aim of reducing maintenance overhead. However, their initiative resulted in negative results. Tests developed with these guidelines resulted in higher maintenance costs compared to tests developed without these guidelines. Their find-
ings underscore an underlying distinction, through an “unseen” factor, between GUI-based test scripts and source code, necessitating additional consideration when adapting guidelines between them.

Existing guidelines for source code review are not generally applicable to test code or GUI-based testing artifacts. Differences in the needs for source code and test code reviews, as well as additional considerations needed when adapting guidelines for GUI-based testing, point out a gap in knowledge. Given the observed benefits of code reviews, one might hypothesize that these benefits could extend to GUI-based testing when informed by tailored guidelines. To test this hypothesis, such guidelines must first be developed, tailored to GUI-based testing, and evaluated. For efficiency purposes, these guidelines then need to be supported with appropriate tools.

We address this gap of how to discuss GUI-based artifact changes through code review guidelines by answering the following research questions:

**RQ2.1:** What are existing guidelines for reviewing GUI-based test artifacts?

**RQ2.2:** Which code and test code review guidelines can be mapped to GUI-based test artifacts?

### 1.4.3 Gap 3: Suitable test methodology for collaborative GUI-based testing

For the operationalization of collaboration on GUI-based testing artifacts, an appropriate testing approach must be selected. This approach should be tool-supported to minimize manual intervention, thereby reducing associated costs. In alignment with our overarching goal of enabling collaboration among individuals with diverse technical and domain knowledge, the testing method must prioritize user-friendliness, catering to both technical and non-technical people.

We identify two primary pathways for the development of tool-supported testing approaches, each with its own benefits and drawbacks. The first option involves employing a GUI-based testing approach that leverages established industry-recognized tools like Playwright [96] or Selenium [156]. These script-based tools necessitate proficiency in a specific programming language, which may differ from the system’s language. While this requirement for programming skills may exclude participation from non-technical individuals, opting for a collaborative approach with Playwright or Selenium could facilitate more rapid adoption in an industrial context. The alternative pathway involves adopting a novel testing approach that, while not widely embraced, aligns closely with our
1.4 Research Gaps and Questions

Specifications. Such a GUI-based testing approach may present the advantage of supporting collaborative practices through innovative functionalities, such as augmentations. However, this choice may introduce challenges related to industrial adoption.

A good candidate for a suitable novel testing approach is Augmented Testing (AT). Detailed information about AT and its functionalities can be found in Section 1.2.2. In previous research, Nass et al. [65] evaluated AT through two industrial workshops, seeking insights into its perceived benefits and drawbacks. Participants perceived AT as beneficial and reported benefits such as the reduction of manual work and a better understanding of what is tested and what is not. Most identified drawbacks were linked to the demonstrator’s constraints rather than inherent flaws in the AT approach. In a later study, Nass et al. [66] compared AT with state-of-practice script-based GUI testing methods in a quasi-experiment, finding that AT is more efficient in creating automated test cases. Moreover, testers required less programming and automation expertise than the other tools. Which is important to us to accommodate contributions from non-technical individuals.

Alternative testing methods similar to AT, such as those proposed by Chen et al. [28] for crowd testing and Liu et al. [59] for Android applications, support the claims of AT-like approaches being easy to use for both trained and untrained testers.

AT and Scout, as an academic tool, is not an established tool used in the industry. This makes evaluating collaborative practices with AT in the industry challenging, as it necessitates companies to re-create existing test cases with AT. Despite potential challenges with industry adoption, we selected AT as a candidate due to its reported benefits of being efficient, usable for non-experienced testers, and availability.

However, existing literature provides insufficient insights into AT’s usability. Moreover, evaluations of AT (and AT-like approaches) predominantly center on exploratory testing, creating ambiguity about its efficiency in contexts like regression testing. To address this gap, we formulated the following research question:

RQ3.1: Is Augmented Testing an efficient approach to GUI-based testing?

RQ3.2: What are the usability issues of Augmented Testing?
1.5 Research Methodology

Research methodologies provide a systematic and scientific approach to generating new knowledge by addressing research challenges [183]. They aim to ensure the research process remains objective, rigorous, and replicable, producing findings that are both credible and reliable. Within software engineering, methodologies such as design science and action research are pivotal. These methodologies encompass activities that can be conceptualized as an engineering research cycle, including diagnosing the problem, planning and action-taking, and evaluation followed by learning or specification [183]. Various research methods, such as surveys, interviews, and case studies, can fit within a single methodology.

**Exploratory** studies observe subjects in their natural environments, with findings arising from these observations [88]. The data from such studies are predominantly qualitative and can be gathered through observation, interviews, and surveys [183]. The required flexible design favors research methods such as case study research. For example, to understand a phenomenon that can be observed as part of an industrial practice within specific organizations.

**Explanatory** studies aim to quantify cause-effect relationships, often through the comparison of groups or treatments [88]. Such studies primarily gather quantitative data through a more fixed design. Experiments serve as an effective method for conducting explanatory research, as they enable a controlled and isolated evaluation of one practice’s effect over another by examining its impact on the outcome.

The research methodologies used in this thesis are a systematic literature review and a controlled experiment.

1.5.1 Experience Report

Chapter 2 reflects on our initial research efforts toward a collaborative workflow for GUI-based testing that allows active collaboration. The aim of the study is to propose not only a conceptual workflow but also a technical demonstrator to evaluate the workflow. We faced challenges throughout various stages of workflow development, which forced us to halt development for both the desired workflow and its demonstrator. As a consequence, we presented the challenges encountered and the decisions that were taken as lessons learned. While this presents negative results, it helps other researchers to circumvent similar pitfalls.

While the significance of negative results is well-acknowledged in several research disciplines, it is less common within computer science and software engineering [50]. For instance, a publication bias—a phenomenon where studies
with positive results are more likely to be published than negative ones [34]—can be observed in the field of software engineering [55, 73, 76]. Disclosure of negative results provides valuable insight to fellow researchers, saving them from spending resources on similar approaches and avoiding the same pitfalls. Further, these negative results offer insights into the limitations and constraints of existing technologies.

From this study, the challenges we identified related to collaborative processes and tools for GUI-based testing put us in a better position to achieve such a collaborative workflow successfully. For example, our second study is motivated by the identified challenge of discussing artifact changes.

### 1.5.2 Systematic Literature Review

Kitchenham and Charters [56] define a systematic literature review (SLR) as “[a] form of secondary study that uses a well-defined methodology to identify, analyze and interpret all available evidence related to a specific research question in a way that is unbiased and (to a degree) repeatable.”

In Chapter 3, we conducted an SLR following the guidelines by Kitchenham and Charters [56]. The goal of the study was to gather a comprehensive overview of the existing evidence related to guidelines for reviewing source and test code artifacts. Another reason to conduct an SLR is to identify gaps in current research [56].

The guidelines Kitchenham and Charters [56] structure the SLR process into three main phases: planning, conducting, and reporting the review. In the conduction phase, we performed the following steps:

**Identification.** We select five digital libraries commonly used in software engineering literature reviews to gather a comprehensive list of relevant sources [44]. Our search string is formulated around the primary term “code review” and its synonym “software review” in conjunction with six guideline-related terms. The terms “testing” or “GUI-based testing” are not included due to the limited sources they yielded. We modify the query search string to comply with each library’s specific query syntax.

**Data Extraction.** All the identified sources are compiled into a data extraction form, as described by Kitchenham and Charters [56], subsequently reviewed by the authors.

**Quality Assessment.** We evaluate the quality of gathered sources in several stages. Initially, we formulate a set of inclusion criteria to filter sources
relevant to our research question, ensuring consistent source evaluation among the authors. Subsequently, we establish inclusion and exclusion criteria for the extracted guidelines from the sources to narrow the focus on guidelines that can be applied to GUI-based test artifacts. Our study intentionally excludes guidelines related to processes, such as reviewer selection, as they are not relevant to our focus. Finally, we perform data triangulation on the identified guidelines, necessitating a minimum of three sources reporting a specific guideline. Data triangulation uses different information sources to increase a study’s validity [145].

Synthesis. We choose coding to analyze the collected sources, a method that allows us to extract guidelines from the sources in a systematic way. For coding, we utilize the Straussian grounded theory approach [84]. The two coding authors debate the codes until a consensus is reached to create the final code system, mitigating researcher and selection biases [18]. Lastly, we perform the synthesis by mapping the coded guidelines to GUI-based testing artifacts using logical inference.

A systematic mapping study (SMS) is an alternate method for systematic reviews closely related to SLRs. The key difference between an SLR and an SMS lies in the research questions’ scope and the analysis’s depth. An SMS typically has broader research questions resulting in a broad coverage of a topic, thus contrasting the narrow focus of an SLR [56]. Further, an SMS lacks an in-depth analysis and synthesis of the gathered sources. An SLR is more suitable for our research questions due to its narrowed focus on guidelines for artifact reviews, excluding guidelines related to human factors, processes, and environments, and the necessary depth of analysis required to map findings to GUI-based testing.

Snowballing is another review method to systematically identify primary studies using references from previously recognized papers [82]. This technique is a trade-off between finding all literature sources and excluding irrelevant ones [88]. Given the volume of our gathered literature and the theoretical saturation from the initial round, we deemed snowballing unnecessary. However, after conducting the SLR on white literature sources, we applied backward snowballing on the literature to identify relevant gray literature.

In Chapter 3, we extended our literature review to include gray literature sources to complement the missing information from the academic literature. Gray literature is material not published in an academic outlet, hence lacking peer review, like news articles, reports from companies, Wikipedia, or blogs [38]. Introducing gray literature raises validity concerns, and reviewers must be more
critical of gray literature sources. To address this, we applied distinct quality
criteria to gray literature sources and indicated which guidelines were extracted
from gray literature. The usage of gray and white literature classifies our study
as a multivocal literature review (MLR) [43].

1.5.3 Experiment (Chapter 4)

Robson [75] defines an experiment as “measuring the effects of manipulating
one variable on another variable”. For software engineering, Wohlin et al. [88]
defines an experiment similarly as “an empirical enquiry that manipulates one
factor or variable of the studied setting”. Compared to empirical methodologies
like case studies, experiments allow a high level of control and are mostly done
in a laboratory environment [88]. An exception is quasi-experiments, where con-
founding factors like the assignment of treatments cannot be randomly assigned
to subjects. The inability to deal with confounding variables makes it hard
to determine the causal effects between independent and dependent variables,
thus introducing an internal validity threat. However, when conducted with in-
dustrial participants in an industrial setting, it could strengthen the construct
validity of a study.

In Chapter 4, we conduct an experiment and apply two treatments in a
regression test scenario: Augmented Testing vs. manual GUI-based regression
testing as the baseline. An experiment allows us to measure the effect of AT on
the test case duration with fewer confounding factors.

Prior to executing an experiment, good planning and appropriate scoping
are crucial. Consideration must be given to various trade-offs and their conse-
quent impact on study results. The planning process encompasses the selection
of context, participant sampling, hypothesis formulation, and the choice of de-
pendent and independent variables [88]. It also involves selecting a standard
design type, establishing the instrumentation, and hypothesis testing [63].

We employ Bayesian data analysis (BDA) to analyze the collected data, fol-
lowing the workflow proposed by Gelman et al. [49]. While frequentist statistics
continue to be the prevailing approach in empirical software engineering, e.g.,
the usage of p-values for null hypothesis statistical testing, it also inherits a
number of issues and has faced criticism [16]. For instance, the analysis is over-
simplified by framing hypothesis testing as a binary decision between null and
alternative hypotheses and utilizing the p-values to evaluate statistical signif-
icance [41]. Contrarily, inferences from BDA are more comprehensive in their
underlying assumptions and are explicit about uncertainties.
Descriptive statistics are also an option for the analysis to present interesting aspects of the gathered data [88]. For example, measuring the central tendency (mean, median, and mode), the dispersion of the data, and a graphical visualization in the form of plots and graphs. To visualize the characteristics of the gathered data, we used violin plots to gain insights, e.g., about the distribution of values.

As an alternative method to assess AT’s efficiency, case study research presents a viable option. This research approach seeks to investigate contemporary phenomena within their natural, real-world contexts [91]. Case studies offer high realism but inherently lack controllability due to taking place in a real-world setting. While this research approach can potentially offer valuable insights into the functionality of AT within an industrial context, it poses a challenge in differentiating the effects attributable to the testing technique from those linked to the implementation tool. Such difficulties in differentiating effects prompted our decision to conduct an experiment, ensuring a clearer distinction between the merits of the AT approach and its tool, Scout.

1.6 Overview of Chapters

In this section, we summarize the studies that contribute to this thesis. Each individual subsection corresponds to a specific chapter within the thesis. We first present the contributions of each study, followed by a synthesis of the results and how they contribute to the goals of our research.

1.6.1 Chapter 2: We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

Contemporary software development is a collaborative effort from a group of individuals with varying skills and roles. The rapid evolution and increasing complexity of software systems make it unrealistic for a single developer to comprehend and maintain them. To assist teams in managing source code and low-level testing, developers facilitate established processes and tools that are usually integrated directly into the software development workflow. Thus, constituting a collaborative workflow for teams. In contrast, software testing that is performed on a higher level of abstraction introduces new challenges, such as the reflection of user behavior or the missing context of the system’s GUI, which itself is a product of the entire system’s integration.
In this study, we propose a tool-supported workflow that enables active collaboration on GUI-based tests among testers and developers. This workflow was meant to support efficient and effective team-based development of GUI-based tests.

However, due to several unforeseen challenges that we encountered, we halted the development of the workflow and its demonstrator. Instead of presenting a working workflow, this paper presents challenges, as well as technical and non-technical decisions that lead to this negative outcome. We argue that this work still provides several valuable insights to guide future research and thereby claim the following contributions: a) an experiences report of attempting to develop a workflow and technical demonstrator for collaborative GUI-based testing; b) the steps of our intended workflow and a record of the decisions and challenges that were taken, which culminated in the negative outcome of the project; c) suggestions for future research, building upon our experiences and taking our decisions into account.

1.6.2 Chapter 3: Code review guidelines for GUI-based testing artifacts

In software development and testing, artifacts play a pivotal role in collaborative practices [53]. As we highlighted in Chapter 2, we see a challenge in how discussions around artifact changes for GUI-based testing can be enabled.

Code review is a widely adopted practice to discuss and review code artifact contributions before integrating them into a software product [12, 186, 187, 188]. Reviews help to catch errors, improve sub-optimal solutions, enable knowledge and experience sharing, and, maybe most importantly, allow active collaboration within a team [12, 39]. Nowadays, code reviews are practiced in the form of a lightweight process with tool support, serving as a communication platform for developers [12, 22, 24, 74].

To investigate guidelines that can be used to discuss and review artifact changes, we performed a systematic literature review (SLR) [56]. Additionally, we incorporated gray literature sources to complement the findings from white literature.

The study revealed an absence of guidelines in both white and gray literature that specifically address GUI-based testing artifacts. As a response, we mapped the identified source and test code guidelines to GUI-based testing artifacts, yielding 33 guidelines. Nonetheless, it is crucial to note that the guidelines presented here lack empirical evaluation in practice. With further empirical
evaluation, these guidelines could provide an industrial contribution, filling the void of comprehensive guidelines for the review of GUI-based testing artifacts.

1.6.3 Chapter 4: Augmented Testing to Support Manual GUI-based Regression Testing: An Empirical Study

As noted in Section 1.4, Goal 2, exploring a GUI-based testing approach, does not align intuitively with Goal 1, which concerns the understanding of collaborative processes, but contributes to our overarching goal. Augmented Testing (AT) is a novel tool-supported approach for GUI-based testing. We intend to utilize AT and Scout (a tool implementing AT) as a platform to develop tool support for the envisioned collaborative guidelines and processes. AT is a promising candidate for such a platform due to testing efficiency, usability for non-developers, and its availability. Although prior studies have perceived AT as beneficial, further empirical investigation is required to select it for our purpose [66, 67].

We conducted an experiment to measure the efficiency of AT compared to manual GUI-based regression testing following the guidelines by Wohlin et al. [88]. The study involved 13 participants from six companies, executing regression test cases with AT against a baseline of manual GUI-based regression testing. We measured the time participants needed to finish each test case, observed participants to identify usability issues, and gathered feedback on Augmented Testing’s benefits and drawbacks.

Our results show a significant improvement in testing efficiency when employing AT. In addition, we synthesized interviews with participants, reporting nine benefits and eleven drawbacks of AT. We categorized these benefits and drawbacks as either essential, meaning that they can be attributed to the testing technique AT itself, or as accidental when related to the technical demonstrator of AT. Most identified drawbacks were categorized as accidental and thus can be mitigated through tool improvements. Lastly, by observing the participants, we gathered additional insights about the usability issues of AT and its demonstrator. Similar to the drawbacks, usability issues were relegated to the demonstrator and not the testing approach itself. In conclusion, the usage of AT (with augmentations) shortens the test execution time in 70% of the cases on average, making it more efficient compared to manual GUI-based regression testing (without augmentations).
1.7 Synthesis

In this thesis, we identified three gaps that need to be addressed to achieve our overarching goal of a *tool-supported approach enabling collaborative GUI-based testing*, illustrated in Figure 1.2. Each subsequent chapter addresses a specific gap as follows.

**In Chapter 2, we address Gap 1**, focusing on guidelines and processes for collaborative GUI-based testing artifacts. Our initial approach aimed to establish a workflow for collaboration on GUI-based testing artifacts, incorporating necessary elements like guidelines, processes, and tools. Encountering several unforeseen challenges, we halted the development of the workflow and reported negative results. Instead of presenting a functioning workflow, we reported our steps toward a workflow and explained the decisions that were taken so that other researchers could avoid the same pitfalls. This study underscores several research avenues demanding deeper exploration beyond the capacity of a single study. Nonetheless, the new insights we gathered on the work of a collaborative workflow allow us to define a better pathway to achieve our overarching goal. Thus, Chapter 2 contributes to the overarching goal by identifying processes and tools that can be used to enable efficient collaboration for GUI-based testing artifacts. We see tool-supported processes for code reviews, visualization of artifact changes, and artifact integration as essential parts of collaboration on GUI-based testing artifacts. The potential benefit of these processes is to guide contributors, both technical and non-technical, in contributing to testing artifacts. Further research into these areas can foster robust processes, mitigating the reliance on ad hoc strategies that might lack rigor.

**In Chapter 3, we address Gap 2**, building upon the challenges identified in Chapter 2. The observed benefits of code review practices that can be found in the literature motivated us to investigate code review guidelines specific to GUI-based testing. We conducted a systematic literature review (SLR) that verified our preliminary assumption of a lack of code review guidelines specific to GUI-based testing. In response, we used the identified guidelines for source and test code and mapped them to create actionable and applicable guidelines for GUI-based testing artifacts. This mapping resulted in a set of 44 guidelines mappable to GUI-based testing artifacts.

These guidelines contribute to the overarching goal by guiding contributors and reviewers through discussions around changes of GUI-based testing artifacts. While these guidelines provide an initial set specific to GUI-based testing artifacts, empirical validation in real-world settings remains necessary to ascertain their true value.
If these guidelines prove effective, they may foster collaboration among testers. Such collaboration could facilitate the exchange of technical and domain knowledge, refining test practices and elevating the quality of test artifacts. Moreover, these guidelines may inform subsequent tool development, ensuring relevant information is available to contributors and reviewers during the review phase.

In Chapter 4, we address Gap 3, focusing on a suitable GUI-based testing approach. We conducted an experiment with industry participants comparing Augmented Testing (AT) to manual GUI-based regression testing to further evaluate AT. Our results confirm previous findings on AT’s efficiency and benefits, such as “monitor test execution”, “test case reusability”, and “usable for non-experienced testers”. The majority of identified drawbacks were linked to the tool demonstrating AT and not the testing approach itself. One aspect that emerged from the study is whether AT is better suited for smaller or bigger test cases. Participants expressed contrasting opinions in this regard, but our analysis of the measured test duration suggests a greater benefit in terms of efficiency for bigger test cases. The unique findings of this experiment highlight the usability issues of AT. While augmentations were seen as beneficial for guiding the testing process, distinguishing between various augmentations and deriving a test case’s intent proved problematic. Such insights can steer further AT’s tool development. In terms of collaborative GUI-based testing, introducing new augmentation types to support collaborative tasks must be accompanied by thoughtful design considerations. For example, the introduction of augmentations that summarize previous discussions about a specific GUI widget or highlight contributions to a test case by others.

Chapter 4 contributes to the overarching goal by strengthening the empirical evidence of AT efficiency for GUI-based testing. An otherwise inefficient testing approach could increase manual effort and, consequently, associated costs. Crucially, our findings affirmed the usability of the testing approach for non-experienced testers, emphasizing the capability to reuse and share manually performed GUI-based tests.

1.8 Future Work

In this section, we discuss potential future work, as summarized in Figure 1.3.

(S4) Usability study on augmentation comprehension for GUI-based testing.

In Chapter 4, we revealed usability issues of AT, such as distinguishing between various augmentations and deriving a test case’s intent from
1.8 Future Work

 augmentation. These issues could amplify when more types of augmentations are introduced to support users in different contexts, e.g., visually contrasting contributions of other users with own. A dedicated usability study on augmentation comprehension in the context of GUI-based testing could address these challenges. Specifically, exploring an effective communication of the purpose of different types of augmentations, ensuring clarity and reducing potential confusion for novice testers, and, thus, enhancing the user experience.

(S5) Empirical evaluation of code review guidelines. In Chapter 3, we propose a set of code review guidelines for GUI-based testing artifacts that have yet to be empirically evaluated. Given that a related study found that proposed GUI-based test development guidelines were ineffective in prac-
tice [10], it is crucial to evaluate the utility of our guidelines in practice. Are these guidelines effective in improving the quality of testing artifacts, and are there drawbacks, such as introduced process overhead, that might render these guidelines impractical?

(S6) Visualize changes of GUI-based testing artifacts. One building block of collaborative GUI-based testing is the visualization of changes, in our case, changes in GUI-based testing artifacts. While established tools to visualize differences rely on simple line-by-line text comparisons, suitable primarily for script-based tests, they overlook the intricacies of more complex representations, i.e., model representations. GUI-based tests introduce additional complexity, such as the textual description of widget properties (i.e., ID or label) used in test scripts is not the same as the visual representation of what is shown on the screen. Additionally, dependencies between GUI states are not represented. This inherent complexity, in combination with often a non-textual test representation, further underscores the need for a robust method to effectively visualize and discuss GUI-based artifact changes.

(S7) Merging of GUI-based testing artifacts. Concurrent testing activities necessitate a mechanism for merging (integrating) test cases and data from different contributors into a common representation. Otherwise, upfront coordination among all participants is necessary. The complexity introduced by GUI-based tests, as described for S6, suggests that employing traditional tools may be challenging. The development of a suitable merging mechanism could be targeted towards a completely automated solution or as a support system by providing users with selectable merging options. Even without a collaborative process, merging can simplify the general use of version control systems for GUI-based testing artifacts. Moreover, a foundational test bench is necessary to assess and refine various merging strategies.

(S8) Empirical evaluation of collaborative GUI-based testing. Upon developing essential components, such as visualization of changes, integration of testing artifacts, and augmentation support, it is necessary to empirically evaluate the resulting solution as a whole. Such a study will reveal whether the proposed solution of enabling collaboration will address the challenges of incorporating contributions from diverse people with different skill sets to improve the efficiency and effectiveness of GUI-based testing. In other
words, will the benefits observed in collaborative software development extend to GUI-based testing.

1.9 Conclusion

This thesis contributes to a tool-supported approach enabling collaborative GUI-based testing. Our initial efforts were directed toward enabling collaborative GUI-based testing through the development of a collaborative workflow. This workflow was intended to integrate and adapt collaborative practices specifically for GUI-based testing. However, unforeseen challenges arose, preventing the realization of a functional workflow. Consequently, we highlighted several challenges that demand a deeper exploration in further studies. These challenges provide us with a better understanding of what is necessary to achieve the overarching goal of collaborative GUI-based testing.

Building upon the identified challenge of discussing GUI-based testing artifact changes, we investigate guidelines to aid discussions between contributors of GUI-based testing artifacts and reviewers. Reviews are a verified collaborative practice that can also be used for testing artifacts, where guidelines can aid both reviewers and contributors during reviews. A systematic literature review revealed the absence of code review guidelines specific to GUI-based testing artifacts. In response, we mapped identified guidelines for source and test code review that could be used for GUI-based testing artifacts. Moreover, we provided examples of how these guidelines can be used for GUI-based testing. These guidelines address the gap in how collaboration through discussions on an artifact level can be realized, thereby contributing to the understanding of how collaboration for GUI-based testing can be established. In software development, code reviews promote knowledge exchange within teams and improve the quality of the artifacts under review. However, an empirical evaluation of these guidelines is necessary to determine whether the benefits reported in software development are observable when applied to GUI-based testing.

To facilitate collaborative GUI-based testing with tool support, a suitable GUI-based testing approach is necessary. Such a testing approach and its tooling should be not only efficient but also easy to use to allow engagement from different people. We selected Augmented Testing (AT) as a promising testing approach that, together with the AT tool Scout, could be used as a platform for tool development. Before we finalized our selection of a suitable testing approach, we saw the need for further research regarding AT’s efficiency and potential usability issues. A controlled experiment with industrial participants
comparing AT with manual GUI-based regression testing demonstrated AT’s efficiency. Moreover, participants reported benefits, such as the usability for non-experienced testers and the ability to share manually performed test cases. The accessibility to non-experienced testers may reduce barriers, enabling individuals with both technical and non-technical knowledge to collaboratively contribute to GUI-based testing. Future developments in tools based on AT could incorporate support for collaborative practices, such as code reviews, by providing necessary information and automating tasks to minimize manual effort.

In summary, our research efforts aim to understand processes and guidelines and operationalize tool support for collaborative GUI-based testing, laying a foundation for future developments in collaborative software testing practices. A functioning approach to collaborative GUI-based testing could improve the efficiency and effectiveness of creating and maintaining GUI-based tests. Apart from improving the quality of artifacts, sharing both technical and domain knowledge within a team may have a lasting effect on individuals’ capability to develop good tests.
Chapter 2

We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

This chapter is based on the following paper:

Abstract

Modern software development is a team-based effort supported by tools, processes, and practices. One integral part is automated testing, where developers incorporate automated tests on multiple levels of system abstraction, from low-level unit tests to high-level system tests and Graphical User Interface (GUI) tests. Furthermore, the common practices of code reviews allow collaboration on artifacts based on discussions that improve the artifact’s quality and to share information within the team. However, the characteristics of GUI-based tests, due to the level of abstraction and visual elements, introduce additional requirements and complexities compared to code or lower-level test code review, delimiting the practice benefits.

The objective of this work is to propose a tool-supported workflow that enables active collaboration among stakeholders and improves the efficiency and effectiveness of team-based development of GUI-based tests.

To evaluate the workflow, and show proof of concept, a technical demonstrator for merging of GUI-based tests was to be developed. However, during its development, we encountered several unforeseen challenges that forced us to halt its development. We report the negative results from this development and the main challenges we encountered, as well as the rationale and the decisions we took towards this workflow.

In conclusion, this work presents a negative research result on a failed attempt to propose a tool-supported workflow that enables active collaboration on GUI-based tests. The outcome and learnings of this work are intended to guide future research and prevent researchers from falling into the same pitfalls we did.

2.1 Introduction

Modern software systems are too complex, and evolve too fast, for a single developer to overview. Such development instead requires a team of people with different roles and expertise to collaborate. To aid teams with source and low-level test code management, processes and tools are available that are typically integrated in the development workflow.

Automated testing has become an integral part of modern software development [92, 99]. With automated tests, developers can verify the conformance of software artifacts to their requirements. These tests run quickly and at low expense, providing continuous and frequent feedback to developers about the
2.1 Introduction

Software’s quality during development. Developers and testers incorporate automated tests on multiple levels of a system’s abstraction, from low-level unit tests to high-level system tests and Graphical User Interface (GUI) tests. In particular, GUI-based tests can verify the behavior of a system through interactions with its GUI, similar to what a user would do [7, 30].

Nevertheless, similar to source code, automated test artifacts can contain defects [85]. To identify such defects, one proposed practice is code inspection, or reviews, of test code. A study by Spadini et al. [83] showed that code review, as a collaborative practice, had a positive effect on reducing test code defects. Further, they showed that reviewers discuss better testing practices, test path coverage, and assertions during the code review, which enables knowledge and experience sharing within the team. Despite these benefits, test code reviews for tests happen significantly less frequently than source code reviews.

However, to the best of our knowledge, the majority of research in this area has been devoted to lower-level tests, leaving a gap in knowledge on how to perform reviews and inspections of GUI-level tests. The characteristics of GUI-based tests, being on a higher level of abstraction (i.e., user behavior) and the additional context of the GUI (i.e., a product of the entire system’s integration), introduce additional requirements and complexities to code review. These characteristics introduce new challenges which can lead to friction in a collaborative environment. Challenges that could be mitigated through a supportive workflow tailored to GUI-based test inspection and review.

In an attempt to support efficient and effective team-based development of GUI-based tests, this work aimed to propose a workflow that would enable active collaboration among stakeholders of GUI-based tests. To the best of our knowledge, no such workflow has been proposed for GUI-based testing, but could similarly benefit GUI-based testing in the same way as collaborative activities for source code and low-level test code. For instance, experienced testers could share information about how and what to test, the domain, and the SUT, with less experienced testers. Educating testers on test practices, design principles, and design patterns could improve efficiency by enabling the reusability and maintainability of tests. For effectiveness, testers could improve the detection of defects and increase test coverage through a better understanding of the system’s behavior and its edge cases. A higher test coverage correlates positively with a higher defect-revealing ability [62, 64]. In addition, similarities to modern software development processes and tools, e.g., integrating changes through a version control system, could lower the barrier to adopting GUI-based tests when developers are also responsible for tests.
Finally, we aim to realize and partially automate, the workflow through a technical demonstrator. However, during the development of the demonstrator, we encountered several, unforeseen, challenges that forced us to halt its development. This paper will present these challenges and the technical and non-technical decisions that lead us to this outcome. The intent of this manuscript is to guide future research and prevent researchers from falling into the same pitfalls we did. As such, we classify the outcome of this work as a negative research result. Traditionally, negative results are uncommon in Software Engineering research compared to other research disciplines. Still, we argue that this work provides several valuable insights and thereby the following contributions:

- A report of our experiences of attempting to develop a workflow and technical demonstrator for collaborative GUI-based testing.
- The steps of our intended workflow and a record of the decisions and challenges that were taken, which culminated in the negative outcome of the project.
- Suggestions for future research, building upon our experiences and taking our decisions into account.

These contributions are important since additional research is still warranted in the area of collaborative GUI testing to mitigate its challenges.

2.2 Background

To understand the implementation of our proposed workflow for GUI-based testing, we must first understand the core technologies and practices incorporated into the workflow. In this section, we provide an introduction to Git and the practice of merge requests. Further, we present the testing tool Scout and its testing technique Augmented Testing, which our demonstrator aimed to extend to support the workflow.

2.2.1 Augmented Testing with Scout

GUI-based testing is the practice of testing a SUT’s conformance to its functional requirements through its GUI [7, 30]. In automated GUI-based testing, tests are written as test scripts or code, or are based on a visual, textual, or formal model [100].
However, a novel technique to automated GUI-based testing is Augmented Testing (AT), where a visual layer between the tester and the SUT’s GUI superimpose information on top of the GUI is provided [65]. In more detail, in an AT workflow, information about a GUI’s widgets or bitmaps is gathered to augment widgets and bitmaps with actions, checks, defects, and suggestions to build a layer between the user and the SUT. All user interactions with the SUT go through the augmented layer, which are recorded and used to update the model for testing. Because of these characteristics, AT is an automated GUI-based testing technique that uses a model-based approach for testing. An example of AT is shown in Figure 2.1. In an experiment by Nass et al. [66], results indicate that AT is a promising GUI-based test automation technique.

Researchers at Blekinge Institute of Technology implemented a prototype tool named Scout that demonstrates AT (also shown in Figure 2.1). Scouts’ architecture consists, among other components, of an augmented GUI that is presented to the user instead of the SUT’s GUI, a plugin system that allows users to extend Scout’s functionality, different drivers that translate user interactions to the corresponding SUT interactions, and a model to store actions, checks, suggestions and issues. Testing with Scout is similar to record and replay (R&R) approaches, but it mitigates some drawbacks, such as significant maintenance efforts [66]. For example, Scout can create new branches of existing tests and highlight existing paths through the SUT to simplify exploratory testing. Team collaboration features have not been investigated in Scout, but are identified as a possible benefit by practitioners which indicates the need for further research [65].

2.2.2 Git

A version control system (VCS) is a tool that manages and tracks different versions of software artifacts, which are primarily source code [60]. This ability to track different versions of an artifact enables concurrent and collaborative workflows. The most popular VCS is Git [101], which was created by Linus Torvalds to improve the development of the Linux kernel [60]. In Git, changes are distributed in a decentralized manner and the entire history of changes is present for each client, eliminating the need for a central server for coordination. Hence, Git is a distributed version control system (DVCS).

A typical development workflow with Git consists of branching, committing changes, and merging changes which we explain in the remainder of this section.

Branching: Changes are not performed directly in the main repository. To allow concurrent development, developers create a new branch—A branch is
Figure 2.1: Demonstration of Augmented Testing (AT). The SUT’s widgets are superimposed with information through the augmentation layer. Green rectangles represent valid checks and blue rectangles actions. In the middle, a graph that represents a path through the GUI is presented.
like an isolated environment for artifacts—from a version of the repository to work on a new non-trivial contribution in isolation. Branches are a good way to group changes and allow cohesive contributions. As Barr et al. [19] observed, branches are significantly more cohesive than just a sequence of commits. In terms of collaboration, branching covers important characteristics like being \textit{isolated} and \textit{concurrent}, which are required for the type of collaboration we had in mind for the demonstrator.

The next step is to \textit{commit changes} in the new branch. Artifact changes are performed as Git commits which are atomic operations. Therefore, all changes of a commit are applied, or none of them [60]. Each Git commit contains a message describing the commit’s changes, a reference to the author, and the differences between the current and previous versions of the artifacts. Well-designed commit messages help developers to understand the change and the reasoning behind the change [32]. Further, commit messages help to understand the evolution of the whole software project since they form the history of changes by summarizing them. Regarding collaboration, commits enable \textit{incremental} and \textit{traceable} artifact changes.

Finally, changes need to be integrated back into the main repository. Integration is done via a \textit{merge} where two or more branches are unified into one [60]. When no conflicts between the branches exist, e.g., two branches modify the same line of an artifact with different content, Git can merge those branches automatically.

\subsection{Merge Requests}

Git itself has no straightforward process to propose changes to the main repository. Software forges like GitHub [109], GitLab [110], and BitBucket [111] identified this problem and introduced merge requests as an essential part of their functionality. Merge requests (MR), also known as pull requests, build on top of Git’s functionality and enhance collaboration by enabling discussions on the changes by providing an easy and web-based way to propose changes.

Instead of merging changes directly into the main repository, changes are proposed by creating a new merge request. In a merge request, all changes are visually highlighted and open for discussions and automated checks. Code reviews are a form of discussion that can be performed during a merge request. Based on the feedback of others on the team, the proposed change can be reworked and improved. After the proposed change is approved and no conflicts exist, the change will be merged into the main repository. Figure 2.2 summarizes a typical Git-based workflow using merge requests.
We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

Figure 2.2: Using a merge request to introduce changes to the main repository

2.3 Results

Figure 2.3: Collaborative workflow for GUI-based tests

Figure 2.4: Approach towards a collaborative GUI-based testing workflow

In this chapter, we present the collaborative workflow we envisioned and the decisions we took to create a demonstrator that implements this workflow. An overview of this workflow is shown in Figure 2.3. Whereas Figure 2.4 summarizes the steps with the decisions we took and information about each step’s technical implementation to implement such a workflow.
We define the recurring terms workflow, demonstrator, and collaboration as follows:

*The goal is to establish a workflow for GUI-based tests enabling active collaboration within a team to improve the efficiency and effectiveness of tests. The workflow is a process based on established software development practices, supported by tools, and tailored for GUI-based tests. Its demonstrator is developed to realize the workflow, identify shortcomings that come with its implementation, and refine the workflow based on the insights of the demonstrator. Active collaboration in the context of GUI-based testing, is for us the ability to work jointly on test artifacts and to share a person’s domain or testing experience with others.*

Each step is then presented as a structured text with the following format: (1) a description, (2) the rationale, and (3) its technical implications.

### 2.3.1 Synchronous vs asynchronous communication

*Description:* Communication, the exchange of information among people, is a fundamental prerequisite for collaboration. In distributed teams, collaborative work can be performed synchronously or asynchronously [98]. Synchronous communication is a real-time interaction among people, for instance, as an in-person meeting, by phone, or by video chat. This requires upfront coordination of all participants. In contrast, asynchronous communication takes place individually (the individual decides when to respond), usually in e-mails or other messaging platforms, and does not require upfront coordination.

*Rationale:* In open-source communities and global organizations, collaboration between people is done without sharing the same office or facilities that would allow face-to-face communication without coordination.

One example of how asynchronous communication is successfully used to develop software is the Linux kernel, where communication is done via the Linux kernel mailing list (LKML)\(^1\). In the LKML, technical and non-technical discussions on the Linux kernel design and Linux kernel bugs take place using e-mails to exchange information. The exchanged e-mails contain discussions and artifact changes (a patch) that shall be integrated into the main repository. On top of that, the mailing list serves as an archive of all design discussions and, even more important, decisions that help developers to understand the current design of the Linux kernel. This asynchronous way of communication allows people to read up on previous discussions and contribute to the Linux

\(^1\)https://lkml.org/
We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

kernel using e-mail messages regardless of their physical location, time zones, or preference of working hours. This flexibility is required for the Linux kernel since its development is voluntarily done by different organizations and individuals worldwide.

In global organizations, teams must collaborate with other teams in different countries with different time zones, making it difficult to find overlapping time slots for synchronous communication. Further, software development practices like merge requests with code review are practiced in an asynchronous and lightweight fashion.

We decided to use an asynchronous communication approach for the workflow to allow a more flexible collaboration within and between teams. With a communication approach close to software development practices, we could lean on experiences and knowledge of existing practices and tools for source code.

Technical implication(s): The workflow is not a stand-alone entity but rather builds upon several loosely connected technologies. As such, the workflow becomes essential since the human will be the principal coordinator.

To aid an asynchronous development process, tools have been developed to create, modify, or delete artifacts independently and synchronize artifacts afterward. One of the central tools is Git, the de facto standard for version control of artifacts. Therefore, we chose Git as the central component for test artifact management. Using git will impact future technical decisions of the demonstrator since these decisions have to be compatible with the way Git works and the tool chain that builds on top of Git.

If we had chosen a synchronous way of communication instead, the technical implications would lean towards a central system to manage artifact changes. For example, artifact changes would be discussed in sessions, and parallel changes need to be synchronized during the session by a central system, as it is the case with online collaboration tools like Google Docs for documents. In this way of collaboration, communication between team members would occur more as "human communication", whereas asynchronous communication can be performed via information represented in models or structured information [98].

2.3.2 Discussions around changes

Description: This step aims to increase the quality of tests by incorporating feedback from other testers and developers based on a discussion about the proposed artifact or its changes. During these discussions around artifacts, further information about testing approaches or the domain can be shared. With
discussions, experienced testers can share their knowledge with less experienced testers, with the artifact under discussion as a specific example.

Different practices exist to facilitate discussions around artifacts. One discussion practice is having meetings and workshops. These meetings and workshops can be organized to discuss current topics of the development state or for general education purposes. Current topics can be the development progress, challenges, problems, or the future direction of development. For education purposes, people meet to share their experiences about the domain, new technologies, and development and testing techniques to educate each other. Since we chose an asynchronous way of communication, meetings and workshops do not fit into the workflow but could have been a consideration for a synchronous workflow.

A more artifact-centric practice is software inspections, where other developers inspect artifacts to improve an artifact’s quality [116, 117]. Previous forms of software inspections, as Fagan [118] described it, were formal and highly structured processes, which hindered its broader adoption. Over time, software inspections transformed towards a lightweight, tool-supported, and informal process known as modern code review (MCR) [12, 22, 24, 74]. This modern form of reviewing artifacts is common practice in open-source software projects and software development companies [12, 31].

**Rationale:** Code reviews fit well into our workflow to discuss artifact changes before integrating them into the main repository. It is an active approach to sharing information within the team as new changes are proposed. This practice allows testers to discuss aspects that are not directly observable from the artifact itself, e.g., why an artifact change is required or why a particular testing practice is more favorable. For simple changes that do not require feedback from an experienced tester, a code review can serve as a sanity check. Moreover, testers can learn retrospectively from code reviews of other people or previous code reviews, which represent the history of changes and decisions.

**Technical implication(s):** To support code reviews from a technical point of view, either code review capabilities must be integrated into the demonstrator, or the code review tool, already established as part of the development process, needs to be used to review a model’s file representation (artifacts). We chose to use existing code review tools because a better integration into existing development processes could reduce the barrier to adoption. Code review tools and software forges like GitHub support the code review process by creating a traceable link between review comments, the authors of comments and code changes, and a reference to a particular artifact version in the version control system. Figure 2.5 shows a discussion about a code fragment with traces to users
We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

and Git commits in GitHub. This choice implies that the artifacts can be, or needs to be prepared to be, reviewed directly, independent of the demonstrator.

Figure 2.5: Example of a code review discussion on GitHub. Comments can be attached to specific code fragments

2.3.3 Human-readable artifacts

Description: Scout and other model-based GUI testing tools use a model to visualize the test scenarios, as a specification of the test behavior, and as the foundation for automated tests and, thereby test execution. When those models are stored as a file, the file can be represented as a binary or a human-readable textual file. With a binary file representation, a program can directly serialize object structures without a complex parsing process, resulting in a propriety format that requires deep insights into how to read and write data. This closed-format property makes it hard to compare different versions of artifacts using common difference tools to integrate changes.

On the contrary, general-purpose human-readable artifact file formats are program independent and thus often used for exchanging data between different tools and systems. Due to the encoding as human-readable text, data needs to
be parsed for each read or write operation, which causes an overhead but allows the inspection by common tools.

Examples of general-purpose human-readable artifact formats are JSON\(^2\) (JavaScript Object Notation), YAML\(^3\) (YAML Ain’t Markup Language), and XML (eXtensible Markup Language). Various tools are available to inspect or modify these widely used artifact formats, e.g., compare or filter artifacts.

Rationale: For our demonstrator, we chose to represent GUI tests in a human-readable artifact format that is an open standard instead of a binary format. The highest impact on this decision has the previous step of the workflow of having code reviews to share information. Effects on performance caused by choice of file representation are negligible in the context of GUI-based testing tools where the execution of the SUT presents a huge resource usage impact. It is more important that humans can inspect artifacts and their differences presented by tools. Even if the artifact representation does not show the full picture of the model, it allows the user to determine minor changes, like renaming a property that does not change the behavior of the test. Such minor changes could be handled without deeper and time-consuming inspections through the model-based test tool. Further, Git and tools built on top of Git are designed for text-based artifacts.

We chose JSON as the specific format for a human-readable artifact. The decisive factor for this decision is the format’s widespread use and its simplicity, which, nevertheless, can represent the demonstrator’s data structures. Whereas XML would be more verbose and YAML is more complex without benefits for our demonstrator.

Finally, some other GUI-based testing tools use JSON for artifact representation, demonstrating the usability of JSON for GUI-based testing, e.g., Selenium Web IDE\(^4\) or Graphwalker\(^5\).

Technical implication(s): Due to the good availability of JSON parsing libraries for all common programming languages, there are no significant challenges with supporting JSON for file representation. Not covered by the JSON specifications, and therefore not enforced by the format, are timestamp values or relationships between elements, e.g., using an identifier to link to another element within the JSON file. Within the JSON file, we store all identified widgets of a model as a flat list and link to these widgets within the model using an identifier so that a quick widget lookup without traversing hierarchical

\(^2\)https://www.json.org/
\(^3\)https://yaml.org
\(^4\)https://www.selenium.dev/selenium-ide/
\(^5\)https://graphwalker.github.io/
structures in the model. The integrity of this links can not be ensured by the format itself and needs to be check by the demonstrator.

In general, open standards have many benefits, such as fostering interoperability between systems, allowing independence from a single vendor, aligning more with the needs of its customer, and enabling innovation [122]. We see a potential benefit for our workflow and demonstrator in allowing the inspection with other tools. However, it introduces the threat that other tools can corrupt or cause defects if not aligned with the standard or aspects not enforced by it are interpreted differently, like with timestamps in JSON. In this case, Git could revert a corrupted file state to a previous defect-free state.

From a technical perspective, JSON covered our needs for human-readable artifacts that can be inspected by other tools and can easily be implemented in a Java-based demonstrator.

2.3.4 Incremental changes

Description: Agile methodologies have become the norm for software development [1]. With an agile process, an incremental approach allows building the application in small steps. Short and iterative development cycles enable fast verification and corrections, which minimizes risks by detecting problems early in the development process [1].

To facilitate this iterative way of working, guidelines for source code development recommend keeping the size and the number of artifacts small [33, 119, 120].

Rationale: Our rationale for having incremental changes does not refer to the creation and modification of tests themselves but to the ability to collaboratively contribute small changes and exchange information like the rationale for a change. We assume that GUI-based tests will benefit from being able to provide small changes in a similar way as it does for source code and lower-level test code. Similar to source code, small and incremental changes could be used to discuss and share information about a specific aspect of the test within the team in a practice like a code review. Further, incremental changes could help to separate concerns so that maintenance tasks on tests and the creation of new tests can be provided as separate incremental changes instead of one big change. Not all artifact changes have the same impact on the codebase and therefore do not need the same level of inspection. For instance, updates on metadata such as timestamps or auto-generated identifiers.

Technical implication(s): Incremental changes are proposed to the main repository using merge requests, an established software development practice.
2.3 Results

With a merge request, artifact changes are open for an inspection and discussion by others. Through these discussions, the author of the change can incorporate feedback into the proposed change. Finally, one or more team members will either approve or reject the change to be integrated into the main repository.

Merge requests build on top of Git’s functionalities (commits, branches, and merges), and software forges like GitHub, GitLab, and BitBucket integrated merge requests as an essential part of their platforms. For the demonstrator, this means that the file representation of the model should be usable with Git. This usability with Git was achieved using human-readable file representation for the model in the previous step. Hence, no additional development is required for the demonstrator to incorporate merge requests.

2.3.5 Integrate changes

Description: Since multiple developers and testers can work independently on GUI-based test artifacts, changes in artifacts need to be integrated into one consistent representation after the discussion and approval of artifact changes. Otherwise, the integration of changes could render an artifact faulty.

But concurrent changes in artifacts in a distributed environment can cause conflicts when integrating these changes in one artifact. These conflicts are known as merge conflicts in the context of a VCS. While merge conflicts will not occur every time they are guaranteed to occur at some point, even with modern version control systems in place [121].

When a merge conflict occurs, which means that automatic merging strategies failed, a person needs to decide which changes should be integrated to resolve the conflict since it is not clear what the expected outcome of the merge should be. All artifact changes that do not cause merge conflicts and are approved should be integrated (merged) into the codebase.

We see two main strategies to integrate changes on artifacts, on a pure artifact representation level or within the tool that created the artifact.

Rationale: We decided to integrate artifact changes within the tool that created the artifact for the following reasons. Semantic relationships between GUI stats (pages, screens, etc.) and widgets can be better handled within the tool when working on the model layer. For instance, if a GUI state is deleted, all its subsequential GUI states and widgets must be deleted as well. The testing tool already implemented the handling of such a case, whereas a generic solution only working on the artifact’s representation does not have this ability.

Technical implication(s): In the technical implementation of this step, the integration of concurrent changes is performed by the demonstrator before the
artifacts are integrated into the main repository. This does not remove the risk of merge conflicts but instead moves the responsibility of resolving a merge conflict to the demonstrator. Therefore, conflict resolution is managed by the demonstrator.

2.4 Challenges

In this section, we discuss the challenges that forced us to halt the development of the demonstrator. These challenges did not emerge from one particular step but rather accumulated through the different steps and surfaced later in the development of the demonstrator.

2.4.1 Code review guidelines

For source code, code review guidelines aid developers who provide code changes and those who review them to improve the outcome of code reviews. On the one side, code review guidelines help a developer to prepare code changes in a way that allows the reviewer to better understand the change and its impact on the codebase. In providing such information, a reviewer does not have to spend time gathering it for themselves, which positively affects the review’s efficiency [71]. On the other side, code review guidelines help the reviewer to know what to cover in a code review, e.g., to ensure the readability and maintainability of the codebase. However, code review guidelines are not an explicit part of the workflow, but the demonstrator aims to assist the collaborative workflow. Established guidelines could provide insights on how to assist the code review process with the demonstrator in an effective way, e.g., gather information that is often requested by reviewers, automatically.

We could not identify any guidelines that were explicitly stated for GUI-based testing artifacts. This led to a dedicated study where we investigated existing literature on code review guidelines for use on GUI-based testing artifacts [21]. In the study, we mapped identified guidelines for code and test artifacts to GUI-based test artifacts. The results of the study are 33 guidelines, summarized in nine guideline categories, that are successfully mapped as applicable to GUI-based testing artifacts. These nine categories of guidelines were perform automated checks, use checklists, provide context information, utilize metrics, ensure readability, visualize changes, reduce complexity, requirements and follow design principles and patterns. While this previous work provides an important stepping stone for code review guidelines for GUI-based testing,
future work is needed to evaluate these guidelines empirically. Based on these guidelines, an attempt could be made to support the code review process.

2.4.2 Visualize changes for code review

Another challenge regarding code reviews emerges because code review tools are built on top of Git to present artifact changes. In Git, differences between different versions of an artifact are determined by a line-by-line comparison. This line-by-line comparison lacks the context of a graphical representation of the model and its differences, like a difference in widget checks, interactions with widgets, or transitions to other GUI states. These differences are, in our case, differences in the JSON file representation, which is not understandable in the same way as the visual test model would be.

Even for non-GUI test code, code review tools do not support a side-by-side view of the source and test code to understand the full context of test artifacts [83]. Relying on the GUI testing tool to review artifact changes would require the testing tool to replicate code review functionalities and integrate deeply with the VCS. In addition, this would break existing source and test code development processes that rely on a particular code review tool.

Similar problems occur where a visual model is a primary artifact, and its file representation encodes the visual model as text, as with UML diagrams or Simulink projects [69, 70]. For instance, layout changes of UML diagrams are irrelevant for most diagram types and should be ignored, whereas the identification of attribute changes in class diagrams is particularly challenging. The algorithm to deal with structural changes in UML diagrams, which was presented by Ohst, Welle and Kelter, resulted in a document type-specific solution [69]. This forces users to use the current demonstrator next to a code review tool to understand and discuss changes.

2.4.3 Integrate changes

Integrating (merging) concurrently updated artifacts changes the consistency of the final artifact in terms of the logical consistency of the representation and sequence of test operations needs to be ensured. Automated merging does not work for all changes, especially complex changes can cause a merge conflict. During merging, automated approaches for merging can not cover all possible changes, especially complex changes. Thus, merge conflicts are not avoidable completely and will occur at some point [121]. When merge conflicts occur, human intervention is required to decide which or how changes should be integrated.
since a tool cannot determine it automatically, or multiple merge outcomes are valid options.

Our demonstrator automatically handles simple types of changes when it comes to merges, e.g., the addition, modification, and deletion of GUI states in a test case. Table 2.1 summarizes how the demonstrator deals with different constellations of concurrent changes by two people. Changes of more than two people can be performed as a sequence of two changes.

<table>
<thead>
<tr>
<th>Change A</th>
<th>Change B</th>
<th>Integration strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>If both additions are from the same widget interaction, then merge both GUI states, otherwise add both</td>
</tr>
<tr>
<td>add</td>
<td>modify</td>
<td>Accept modifications and add a transition to new GUI state</td>
</tr>
<tr>
<td>add</td>
<td>delete</td>
<td>Add transition to new GUI state and mark the GUI state and all subsequent states as deleted</td>
</tr>
<tr>
<td>delete</td>
<td>modify</td>
<td>Accept modifications and mark the GUI state and all subsequent states as deleted</td>
</tr>
<tr>
<td>modify</td>
<td>modify</td>
<td>Accept modifications (the last update wins)</td>
</tr>
</tbody>
</table>

Table 2.1: Constellations of changes for one GUI state and how they are handled in the demonstrator

One example from Table 2.1 is when two persons independently added a transition to a new GUI state in a test case. If the added GUI states represent the same GUI state, they will be merged as one; otherwise, two separate transitions to different GUI states are added. This scenario of concurrent changes is shown in Figure 2.6. Our demonstrator determines two GUI states as the same state if the same interaction with a widget, e.g., click a specific button, is performed.

Deleting a part of the test case is done by marking the related GUI state and all subsequent GUI states as deleted in a meta-data field in its JSON representation. Marking instead of permanent deletion can avoid merge conflicts since changes from other persons can be incorporated into the test cases marked as “deleted”, and a person can restore the deleted scenarios if needed.

However, for the integration of changes, we encountered three challenges during the implementation of our demonstrator.
First, changes that result in the same GUI state but are based on different widget interactions cannot be detected as the same GUI state. For instance, two people using different values for an input field can result in seeing the same GUI view but with different widgets displayed, and therefore, it would represent another state of the GUI. This identification problem makes it harder to implement merging algorithms and support systems for model-based GUI test cases.

Second, to evaluate a merging algorithm, requirements on how different types of changes should be integrated, or a dataset of test cases with concurrent modifications, are needed.

Third, when merge conflicts occur, and a person needs to decide which and how changes should be integrated, the demonstrator should support the person by providing visual information about the changes. Plenty of tools exist for text-based artifacts that allow a side-by-side comparison of artifacts and visually highlight changes. Additionally, such tools allow the integration of the highlighted change by clicking on them. This type of support does not exist for model-based GUI test cases, which increases the manual effort and time spent handling merge conflicts.

Figure 2.6: Example of a simple merge scenario of two test cases

2.5 Conclusion

Collaborative GUI testing, and tools for this practice, are lacking in industrial practice. Such tools and ways of working are available for lower-level tests, but for GUI testing more work is required to benefit from it. In this study, an attempt was made to create a workflow and a technical demonstrator for such a practice.
We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing

However, due to several unforeseen challenges during the development of the technical demonstrator, we were forced to halt its development. Thus, we discussed the main challenges and provided a rationale for the steps of our intended workflow and a record of the decisions that were taken.

For future research, we suggest (1) further development of the workflow and the demonstrator to identify their actual benefits and drawbacks, (2) empirical evaluation code review guidelines suitable for GUI-based tests to assist testers, (3) visualizing changes of model-based tests to allow discussions within code review tools, and (4) exploration of integration strategies for concurrent changes on models for GUI-based tests.

In conclusion, this work presents a negative research result on a failed attempt to provide a workflow that enables active collaboration on GUI-based tests, combined with a demonstrator that implements this workflow. This manuscript intends to guide future research and prevent researchers from falling into the same pitfalls we did.
Chapter 3

Code Review Guidelines for GUI-based Testing Artifacts

This chapter is based on the following paper:

Abstract

Context: Review of software artifacts, such as source or test code, is a common practice in industrial practice. However, although review guidelines are available for source and low-level test code, for GUI-based testing artifacts, such guidelines are missing.

Objective: The goal of this work is to define a set of guidelines from literature about production and test code, that can be mapped to GUI-based testing artifacts.

Method: A systematic literature review is conducted, using white and gray literature to identify guidelines for source and test code. These synthesized guidelines are then mapped, through examples, to create actionable, and applicable, guidelines for GUI-based testing artifacts.

Results: The results of the study are 33 guidelines, summarized in nine guideline categories, that are successfully mapped as applicable to GUI-based testing artifacts. Of the collected literature, only 10 sources contained test-specific code review guidelines. These guideline categories are: perform automated checks, use checklists, provide context information, utilize metrics, ensure readability, visualize changes, reduce complexity, check conformity with the requirements and follow design principles and patterns.

Conclusion: This pivotal set of guidelines provides an industrial contribution in filling the gap of general guidelines for review of GUI-based testing artifacts. Additionally, this work highlights, from an academic perspective, the need for future research in this area to also develop guidelines for other specific aspects of GUI-based testing practice, and to take into account other facets of the review process not covered by this work, such as reviewer selection.

3.1 Introduction

The development and evolution of software is a complex undertaking dependent on a high degree of collaboration between many professionals with varying skill sets and expertise ranging from different technical skills to domain knowledge [35]. This has led to a rise and reliance on team-based activities to enable this type of collaboration. One commonly used practice of team-enabling activities is code reviews [23, 79]. Code review is a software engineering practice where peers review a code contribution before additions or changes are integrated into the code base [12, 29]. As a practice, code reviews help to catch errors and improve sub-optimal solutions on different types of artifacts, but also enable
3.1 Introduction

knowledge and experience sharing and, maybe most importantly, allow active collaboration in the team [12, 39].

Another practice that has become ubiquitous in modern software development is automated testing [92]. Automated testing is used to verify the conformance of the developed software to its requirements in an automated fashion and is run frequently to provide continuous feedback to developers. Test automation can be performed on multiple levels of system abstraction, from low-level unit tests to high-level system tests and Graphical User Interface (GUI) tests. Notably, GUI-based tests can verify the behavior of a system through interactions with its GUI in a similar way a user would [7, 30]. Thus, GUI-based testing constitutes a powerful tool to find regression defects that user would otherwise encounter [7].

Guidelines within Software Engineering aim to provide actors in the software development process with best practices for high-quality software development. But most existing guidelines that seek to improve the effectiveness and efficiency of code reviews focus on source code and lower-level test code [33]. In fact, to the best of our knowledge, no review guidelines have been proposed for GUI-based tests, even though similar quality benefits can be assumed to source code due to the GUI test code’s similar characteristics. This observation is supported by results from interviews with GUI-based testing experts from the industry, who state that they use ad hoc guidelines due to the lack of general guidelines in the literature. Coupled with reports that GUI-based tests often suffer quality issues, we see a concrete need for research in this area to both aid industrial practitioners and bridge a gap in knowledge in academia.

In this study, we investigate existing literature on guidelines for code reviews of software development artifacts for the purpose of mapping such guidelines for use on GUI-based testing artifacts. For this investigation, we performed a Systematic Literature Review (SLR) complemented by a gray literature review to extract review guidelines for source and test code. These results are then synthesized into a set of nine code review guideline categories. Using expert judgment and experience with GUI-based test techniques, these categories are then mapped, through examples of application, to GUI-based testing to create review guidelines for GUI-based testing artifacts. We stress that the study is delimited to guidelines explicitly related to review of artifacts, e.g., code files. Thus, omitting guidelines associated with less tangible aspects of the review process or review environment, such as reviewer selection or effort allocation [124, 125, 126].

Based on our results, we claim the following contributions:
• A macro-analysis of review guidelines of source and test code in existing white and gray literature;

• A set of nine code review guideline categories that are mapped as applicable to GUI-based test artifacts;

• Examples of how to apply the identified code review guideline categories to GUI-based test artifacts.

The remainder of the paper is organized as follows: in Section 3.2 we provide background information about GUI-based testing and Modern Code Reviews; in Section 3.3 we illustrate the methodology that we employed to conduct our literature review; in Section 3.4 we report the results of the review; in Section 3.5 we discuss the results and the threats to validity of the present study; in Section 3.6 we conclude the paper by summarizing the main findings and providing future research directions.

3.2 Background and Related Work

The following subsections provide definitions of the concepts of GUI-based Testing and Modern Code Review and references to literature related to Code Review of production and test artifacts.

For the sake of clarity, in the remainder of the manuscript we will adopt the following definitions: we will call production artifacts the parts of the system containing the logic of the project and running in production. We will call as test artifacts the parts of the system containing the test cases which verify if the application works as expected. The nature of test artifacts depend on the specific testing technique that is applied to the project.

3.2.1 GUI-based Testing

Graphical user interface (GUI) testing can be utilized for different test purposes, categorized on a general level of abstraction into two types; (1) GUI tests and (2) GUI-based tests [7]. GUI testing is defined as the practice of verifying the correctness of the GUI’s visual appearance according to the system under tests (SUT) requirements, e.g., that components have the right shape, color, and positioning. GUI-based tests, on the other hand, are defined as the practice of testing a SUT’s conformance to its functional requirements through its GUI [7,
These tests are thereby system-level tests that are performed through End-to-End (E2E) scenarios—Sequences of events performed against the GUI [13]—that are executed with test drivers that can interact with the SUT’s visual GUI, the GUI model or other GUI-related interfaces. These tests can be manual or automated and are commonly used for regression testing [92]. However, for this work, we primarily discuss automated GUI testing with tests written as testware (e.g., test scripts or test code) or model-based tests (e.g., visual, textual, or formal models) [100].

GUI-based testing can be categorized in different ways. One categorization is based on the way the test cases are defined where three categories of tests are usually discussed, i.e., scripted tests, capture and replay and model-based tests. Scripted GUI-based testing is based on the development of test scripts, or test code, with the usage of dedicated automation APIs, tools and frameworks (e.g., Selenium in the web application domain [95]). Capture & Replay testing resorts on providing instruments, usually in the form of tools, to record the operations performed by a tester, or user, on the GUI of the system, to generate re-executable test sequences [127]. Model-based testing is based on the, either manual or automated, generation of models of the SUT’s intended behavior from a GUI level of abstraction. These models are then used to generate test sequences that provide coverage of the different states of the GUI [128]. These models may be visual (i.e., defined with nodes and vertexes) but can also be textual (e.g., utilized in behavioral-driven development) or formal (e.g., mathematical models).

Another categorization of GUI-based test techniques, provided by Alégroth et al. [6], is based on the GUI-based tests means of interacting with the SUT, i.e. their test drivers. In this categorization, three different generations of GUI testing approaches are identified. Coordinate-based, or first generation GUI testing, is an approach driven by tools that identify the elements of the GUI through their coordinates on the screen. These tools generally utilize the capture and replay paradigm, where coordinates to GUI elements are provided by a human user during the recording of test sequences. Component-based, or second generation testing, is an approach driven by tools that perform interactions through the GUI’s layout model or properties, e.g., through access to an application’s document object model (DOM) on web applications or other GUI interfaces. Lastly, Image recognition-based, or third generation GUI testing, is an approach driven by tools that utilize computer vision to identify elements on the screen based on their visual appearance [27].

Whilst these classifications differ and outline a plethora of different types of automated GUI-based testing approaches, they are all used for the same test
purpose, i.e., system-level testing. However, whilst research into the technology used by the approaches is ubiquitous, research into guidelines and ways of using the approaches is less common, leaving a gap in knowledge. Development guidelines for GUI-based tests have been proposed [10], but guidelines for reviews of GUI-based tests are, to the best of our knowledge, not covered in the body of knowledge. Such guidelines are perceived important to aid improve the effectiveness and efficiency of both individual and team-based GUI testing.

### 3.2.2 Modern Code Review

Code reviews started as a waterfall-like procedure back in the 1980s as software inspections, a practice where other developers manually inspect artifacts to improve an artifact’s quality by verifying their correctness[116, 117]. Fagan [118] described this early form of software inspection as a formal and highly structured process. As such, this practice has been shown to be effective in finding both errors and improvement potentials, applied throughout the development process of any product. Nevertheless, the formal nature of this approach and the resulting overhead hindered the adoption of software inspections in the past [129].

Nowadays, code reviews have transformed towards informal, tool-supported, lightweight processes that build a communication platform for developers [12, 22, 24, 74]. This current form of code reviews is a common practice in most agile software development companies and open source software projects, also known as modern code review (MCR) [12, 31]. Dedicated code review tools such as Gerrit [107], Phabricator [108], or CodeFlow [115] support the code review process by providing context-specific information. Modern software forges such as GitHub [109], GitLab [110], or Bitbucket [111] even integrate the functionalities of these dedicated tools as a part of their collaborative software development workflows.

In this work, we use the term code change to refer to added, deleted or changed code that is inspected through a code review. This practice is instigated by the change provider, e.g., a developer, who submits an artifact for review by a reviewer. Other terms, used interchangeably in literature for this practice, are changeset or patch. The latter is a common term in the Open Source Software community to describe artifact contributions [22].

Davila and Nunes [31] performed an SLR on the topic of modern code reviews and proposed a taxonomy for the approach. Through this work, the authors identified studies that propose strategies for MCR, which are categorized into phases of the MCR process that are; (1) review planning and setup, (2) code
review, and (3) process management and support. In the planning category, most studies focus on reviewer recommendation and automated selection of reviewers. Only one study in this category is related to the code change itself, proposing a tool to add a narrative to the code and multimedia resources to support code change documentation. Reviewers can then reproduce the change and provide feedback based on a replay of these multimedia comments. In the code review category, the main identified strategies to support the code review process were to provide visualizations of code changes, present properties associated with the changes, and support for the analysis of change impact. Of the 22 included studies, 15 studies propose code-checking tasks for the purpose of understanding code changes. In 10 of these 15 studies, approaches to code change visualization are proposed.

In a study by Dong et al. [33], 57 practices and 19 code review pains from Open Source Software and industrial communities are summarized. From these results, best practices are derived and organized in 5 steps based on the lifecycle of the MCR process. One conclusion of their study is that the code context is one of the most difficult things for reviewers to understand.

Analysis of the included papers in the aforementioned work indicates that MCR is mostly studied in the context of software development. However, in a study by Spadini et al. [83], test code reviews were also investigated, in particular how developers discuss such code. Results show that reviewers mainly discuss code improvements, suggesting better testing practices and some generic code quality practices to ensure the maintainability and readability of test code. These discussions also served to provide code improvement to understand whether a test covers all paths of the production code, i.e., provide coverage. This is an interesting outcome of the study since coverage is a relevant attribute for test code reviewers that is not relevant for source code.

Another practice that was observed was that test code reviewers requested clarification of the implementation intention by asking for the rationale of the change. For knowledge transfer, reviewers link to external resources containing documentation or example to solve a problem that is part of the code change. In other cases, reviewers directly provide an example of how to tackle an issue that is discussed during a code review. The outcomes from the study of reviews of test cases overlap with the findings of Bacchelli and Bird [12]. For instance, the challenges that reviewers of both source and test code face are due to a lack of context and reasonable navigation possibilities within code review tools. Such functionality is valuable since, when reviewing test code, reviewers must often switch between the test and production code back and forth to understand the impact of a change. The paper concludes with features for future
code review tools, such as ways of providing context information to more easily understand and inspect the classes that are under test and their dependencies, enable easy navigation between test and production code, and provide detailed code coverage information for tests.

## 3.3 Study Design

The objective of this work is to identify guidelines for the review of GUI-based testing artifacts by mapping guidelines from source and test code review to GUI-based testing. These guidelines, from existing literature, are acquired through a systematic literature review complemented with a gray literature review. Before that, we conducted six interviews with industrial GUI testing experts to ensure that there is an industry need for guidelines. The mapping is performed by looking at characteristics that are common to GUI test artifacts compared to production and non-GUI test artifacts and what review guidelines target each characteristic. Logical inference is then used, together with the author’s expert judgment and experience with GUI testing (over 30 years of combined knowledge from industry and academia), to map each guideline as applicable to GUI testing. Using logical inference, a person (or machine) goes beyond available evidence to form a conclusion [130]. Thus, a conclusion is valid if the premises (a guideline can be applied to GUI testing) is true but do not follow any specific logic. In detail, this mapping is performed in a systematic top-down approach, where we (1) evaluate the purpose of each guideline, (2) group guidelines in categories and (3) provide examples of how these guidelines apply to GUI test artifact review. Thereby providing a chain of logic to tailor the original guideline for source or test code to GUI test artifacts.

The goal of this work is thereby to acquire a set of guidelines for GUI-based test artifact review, motivated by a stated need for such guidelines from industrial practice. This work is further motivated by an identified gap in guidelines for GUI-based testing, including GUI test review, in academic literature.

### 3.3.1 Research Questions

To achieve the research objective, and meet the research goal, the objective has been broken down into two research questions to guide the research.

- RQ1. What are the guidelines, from white and gray literature, for source code artifact review?
3.3 Study Design

![Diagram of research approach]

**Figure 3.1: Summary of the research approach**
The rationale for this question is to acquire an overview of the most discussed review guidelines such that they can be used as input for the mapping to GUI-based testing artifacts. The answer to this research question will serve as a preliminary body of knowledge about generalizable code review guidelines (i.e., including generic test and source code review) and not specific to GUI-based test artifacts. This preliminary step was necessary because, after performing iterative tuning of the search string, we did not find a sufficiently large body of evidence to support the search by only looking for “GUI-based test” or “test” in the search string.

Due to the study’s objective of providing an industrial contribution, both white and gray in literature will be reviewed. The synthesised guidelines are intentionally kept on a higher level of abstraction, referred to as guideline categories, to ease the mapping to the specific characteristics of GUI-based tests. Furthermore, the literature review will only focus on artifact guidelines, omitting to review guidelines connected to processes, teamwork, and other non-tangible aspects.

- RQ2. To what extent can source and test artifact review guidelines be mapped for GUI-based test artifact review?

The rationale for this research question is to meet the research objective by mapping the guidelines found to answer RQ1 to GUI-based testing artifacts. The mapping is conducted based on GUI test artifact characteristics, stated pre-conditions for review activities connected to the guidelines identified for RQ1 as well as the authors’ expert judgement and experience with GUI testing (in excess of 30 years of combined knowledge from both industry and academia).

### 3.3.2 Methodology

The methodology used for this work is divided into five phases:

1. Interviews with experts;
2. Literature collection;
3. Data extraction and analysis;
4. Elicitation and categorization of the guidelines through coding;
5. Mapping of identified guidelines to GUI-based test artifacts.
3.3 Study Design

All five phases of the study were conducted consecutively, where the output of one phase was used as input to the next phase in iteration. As such, most of these work sessions were held online.

The collaborative sessions were conducted according to the mini-Delphi method for face-to-face meetings, where meeting sessions were closed once a consensus was reached between the authors [134]. These sessions were utilized for (1) the design of the literature review (2) the literature search (3) the analysis of acquired literature sources and (4) the mapping of results to GUI-based test artifacts.

We used an online spreadsheet (Google Sheet) to organize the data through all the phases of the methodology. The spreadsheet also enabled collaborative, asynchronous, work whilst retaining traceability of guidelines back to their original sources.

In the continuation of this section, the detailed activities of each phase of the study will be presented. Wherever suitable, practices that affect the research validity have been added. For a longer discussion about the study’s threats to validity, we refer the reader to Section 3.5.1.

Interviews with Experts

As a starting point for this study and to ensure that there is an industry need for code review guidelines for GUI-based test artifacts we conducted expert interviews. These interviews were part of a general research interest in how GUI-based testing is integrated into an overall development process and are not solely for this study. Thus, we did not consider the interview results as primary data sources, but rather as complementary sources that motivate the study’s importance. Of the 18 questions asked, three were used for this study since they were explicit regarding the usage of tools and reviews of test cases. These interviews were semi-structured with six GUI testing experts from three Swedish companies in the domains of consulting, financial technology and music streaming.

The interviewees were sampled through convenience sampling—Convenience sampling is a non-random sampling where members meet certain practical criteria [37]—from our industrial network. To fulfill the practical criteria, an interviewee must (1) be a professional engineer working in the industry, (2) have more than 5 years of industrial experience with GUI testing, (3) speak English or Swedish, (4) be available when we conduct the study. As a tool-related ques-

\footnote{The dataset is available at the following URL: https://doi.org/10.5281/zenodo.7248201}
tion, we asked the interviewees “What GUI-based testing approaches do you have experience with?” with the follow-up question “What were the tools you used for it?”. We used these mentioned tools as additional gray literature sources to extract guidelines from a tool’s website, manuals, and wikis. The interviewees were also asked if they adopted code reviews of GUI test for quality assurance and if they had used or knew of any general guidelines for GUI-based testing or reviews. However, although the interviewees use code reviews, they knew of no general guidelines, stating instead that the guidelines they had used were developed ad hoc to fit specific contextual needs. The interviewees also stated that general guidelines would, based on their own experiences, be of both industrial interest and benefit.

Prior to the interviews, the interviewees were sent our questions and information about the study’s purpose to prepare them thematically. The interview guide was structured as follows: (1) preamble with explanations regarding confidentiality and consent to audio record the interviews, (2) background information about the interviewee’s industrial experience, (3) the interviewee’s experience with GUI-based testing, (4) team aspects of GUI-based testing, including use of reviews, (5) troubleshooting of failing tests, (6) closing thoughts and asking for topics we did not cover during the interview but seen as important by the interviewee.

We conducted all interviews remotely via video chat and recorded the audio. After the interviews, all audio recordings were transcribed for further analysis.

White Literature Collection

The first phase of the study aimed to find a suitable set of literature sources for data extraction to answer RQ1. This need arose from a preliminary search of explicit reviewing guidelines for GUI-based testing artifacts which resulted in no tangible results. As such, instead of synthesizing existing guidelines, a mapping approach was chosen, taking best practices and guidelines for review of related artifacts as input.

The SLR was conducted following a subset of Kitchenham’s guidelines to conduct Systematic Literature Reviews [56]. Specifically, we applied all the steps of the Planning and Conducting phase in Kitchenham’s guidelines, with the exception of the prescribed forward and backward snowballing steps after the first phase of collecting literature sources. We did not deem it necessary to perform snowballing because of the size of the set of gathered literature and because of the theoretical saturation of concepts elicited from the first round.
Table 3.1: Search results by library

<table>
<thead>
<tr>
<th>Library</th>
<th>URL</th>
<th>#Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE Xplore</td>
<td><a href="https://ieeexplore.ieee.org">https://ieeexplore.ieee.org</a></td>
<td>321</td>
</tr>
<tr>
<td>ACM Digital Library</td>
<td><a href="https://dl.acm.org">https://dl.acm.org</a></td>
<td>122</td>
</tr>
<tr>
<td>Science Direct</td>
<td><a href="https://www.sciencedirect.com">https://www.sciencedirect.com</a></td>
<td>112</td>
</tr>
<tr>
<td>Springer Link</td>
<td><a href="https://link.springer.com">https://link.springer.com</a></td>
<td>628</td>
</tr>
<tr>
<td>Google Scholar</td>
<td><a href="https://scholar.google.com">https://scholar.google.com</a></td>
<td>23</td>
</tr>
<tr>
<td>Duplicates</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>Total (without duplicates)</td>
<td></td>
<td>1160</td>
</tr>
</tbody>
</table>

Although this represents a threat to the validity of the result, due to the reasons stated, we find this threat to be minor.

The white literature collection, as stated in Figure 3.1, involved the following steps:

**Selection of Digital Libraries:** Five digital libraries were used for our search; IEEE Xplore, ACM Digital Library, Science Direct, Springer Link, and Google Scholar. These were chosen because of their common use in software engineering literature reviews and their perceived complementary overlap of the literature [131].

**Search String:** A search string was then formulated and applied to the databases. The search string was, as mentioned, discussed through the mini-Delphi method, and iterated until a suitable base sample of papers was found. We formulated our search string to be as broad as possible, including the mandatory terms *Code Review* (or *Software Review*) and one of a set of six synonyms of *Guidelines*. Therefore, based on the applied search string, the terms “code review” and “software review” are considered as synonyms throughout all of the present manuscript. We opted to not include the *testing* or *GUI-based testing* at this research phase, because of the very limited number of sources that could be found by applying such keyword in the search string.

The final, general, search string is defined as follows:

("code review" OR "software review") AND (Guideline OR Information OR Data OR Framework OR Practice OR “Best Practices”)

This general search string was then adapted to the specific syntax of each of the selected libraries. To evaluate the dependability of the search string,
throughout its iteration, we tested the string and, through random sampling, checked the validity of the found sources in pilot searches where resulting papers were randomly selected and reviewed for validity.

Because of the high number of returned results, we limited the search on Science Direct to title, abstract and keywords, on ACM Digital Library to abstract and title, and on SpringerLink and Scholar to titles. To enable reproducibility of the study, we limited our search results to the end of June 2022. Table 3.1 lists the number of sources extracted from each library by applying the search string.

To facilitate the continued data extraction and analysis phases, we organized all found sources in a spreadsheet (data extraction form [56]) and marked each source with the following attributes: unique identifier, library, title, authors, year of publication, article URL, venue, exclusion reason, and comments. Before data extraction, the form was discussed by all authors.

**Duplicate Removal:** After the sources had been extracted, we applied an automated text analysis script on the elicited CSV files to remove all the duplicated papers from our initial set (i.e., papers with the same name and same set of authors). If a source was listed in multiple digital libraries, we kept only a single instance of the paper in our final paper pool, keeping the source from the library where the source was first encountered (according to the order in Table 3.1).

**Inclusion Criteria:** To evaluate the relevance of sources to our research objective, and enabling consistent evaluation among the authors, we defined the following inclusion criteria:

- **IC1:** The source is related to production or test artifact review;
- **IC2:** The source is an item of white literature available in full-text, and published in a peer-reviewed journal or conference (companion) proceedings;
- **IC3:** The source is written in a language comprehensible by the authors: English, Italian, German, or Swedish;
- **IC4:** The source has been published before April 2022.

These criteria were applied to either the title and source of the identified sources or its meta-data.
3.3 Study Design

We do not explicitly list exclusion criteria as we consider them as the direct negation of the Inclusion Criteria.

At the end of the white literature collection phase, we collected a total of 1160 literature sources (without duplicates). On this set, we applied the inclusion criteria for literature which resulted in 137 white literature sources for further analysis.

Gray Literature Collection

After a preliminary analysis of the acquired sources, we got further support for our previously stated observation that there is a lack of academic literature about explicit guidelines for reviews of GUI-based testing artifacts. Hence, whilst guidelines are available for source and test artifacts, guidelines for GUI-based testing artifacts are not available.

In an attempt to address this lack of information, we extended the literature review to include also gray literature sources. As such, the study can be classified as a Multivocal Literature Review (MLR) [47].

Gray literature (GL) is defined as what is produced on all levels of government, in academia, business or industry in print and electronic formats, but which is not controlled by commercial publishers, i.e., where publishing is not the primary activity of the producing body [38].

Adams et al. classify gray literature into three different categories: 1st tier (or high credibility), which includes books, magazines, government reports, and white papers; 2nd tier (or moderate credibility), which includes annual reports, news articles, presentations, videos, question and answers on websites; 3rd tier (or low credibility), which includes blogs, evidence from e-mails, posts on social networks [2]. We used this classification of GL as a template for our quality assessment questionnaire (Table 3.2) with the following adjustments to aid our research goals: 1st tier, Books, magazines, reports or documentation from companies; 2nd tier, News articles, presentations, videos, Wiki articles; 3rd tier, Blogs, emails, tweets, Q/A sites (such as StackOverflow).

The use of gray literature served two distinct goals: (1) to attempt to complement the missing information from the academic literature and (2) to provide guidelines based on an industrial perspective [43].

The gray literature was collected and filtered by performing the following steps:

Snowballing from White Literature: We applied Backward Snowballing, defined by C. Wohlin as using the reference list of a paper to identify ad-
Backward Snowballing was applied only to the final set of collected white literature sources, i.e., after the application of inclusion criteria. However, the snowballing was limited to only gray literature sources. This is perceived as a possible threat to the search, i.e., that relevant white literature sources may have been missed. However, due to the comprehensive set of found sources, this threat is considered minor.

As with white literature, we organized the gray literature sources in a form that consists of unique identifier, title, article URL, source (Google, interviews, snowballing), white literature reference if from snowballing, publication year, quality attributes (see Table 3.2).

**General-purpose Search Engine:** The search was performed with the same search string as for the white literature, applied to the Google search engine. To limit the search in terms of the number of found sources, we applied the Bounded Effort strategy to the Google search result, meaning that only the first 100 Google search hits were investigated.

**Duplicate Removal and Inclusion Criteria:** After collecting gray literature sources, we filtered the initial pool by applying duplicate removal and IC1, IC3 and IC4 from the set of inclusion criteria defined for white literature. IC2 was not applied since it would have excluded all the identified sources at this stage.

**Quality Assessment:** To evaluate the quality of the collected gray literature sources, we applied a quality assessment questionnaire, which is presented in Table 3.2. We used a 3-point Likert scale (yes=1, partly=0.5, and no=0) for the authority and outlet criteria. For the backlinks, we used a 4-point scale based on the percentile of the number of backlinks in our sample ($\geq 75$th percentile=1, $\geq 50$th percentile=0.75, $\geq 25$th percentile=0.5, $< 25$th percentile=0).

The questionnaire was adapted from the quality assessment questionnaire proposed in [47], where all the sources that had a score higher than 2.5, in a range between 0 and 5, were kept in the final set of sources.

Since these criteria do not directly apply to the interviews we instead relied on the interviewees’ stated industrial experience as a criterion for their credibility.

The described steps conducted for gray literature search allowed us to obtain 378 gray literature sources (from snowballing, search on the Google search...
3.3 Study Design

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authority</td>
<td>Is the publishing organization reputable? E.g., a research-, software- or technology organization.</td>
</tr>
<tr>
<td>Authority</td>
<td>Is the author reputable? (e.g. experience, job title, or other publications in the field)</td>
</tr>
<tr>
<td>Outlet</td>
<td>1st tier GL (measure=1): High credibility: Books, magazines, reports or documentation from companies</td>
</tr>
<tr>
<td>Outlet</td>
<td>2nd tier GL (measure=0.5): Moderate credibility: News articles, presentations, videos, Wiki articles</td>
</tr>
<tr>
<td>Outlet</td>
<td>3rd tier GL (measure=0): Low credibility: Blogs, emails, tweets, Q/A sites (such as StackOverflow)</td>
</tr>
</tbody>
</table>

Impact | Number of backlinks (using https://www.seoreviewtools.com/valuable-backlinks-checker/) or citations. |

Table 3.2: Quality assessment questionnaire for gray literature. We use a 3-point Likert scale (yes=1, partly=0.5, and no=0) for the authority and outlet criteria.

...engine, and tool mentions in expert interviews) without duplicates. After applying the inclusion criteria and the final quality assessment we were left with 105 gray literature sources that are used to extract candidate guidelines in the next step.

Data Extraction

After the literature source extraction, the elicited sources were analyzed through the following steps:

Guideline Collection: All the sources were read in their entirety to identify possible guidelines for code review. For efficiency, the potential sources were split among the authors and read independently. Each time a guideline or best practice for code review was found in the paper, it was added as a candidate guideline in a separate tab from the white literature source of the data collection form.

For each candidate guideline we collected the following information:

- The full text in the literature item describing the guideline;
- The nature of the guideline, i.e., if it is defined for source or test code;
- If it is an explicit guideline, i.e., an actionable practice or interpreted as such;
Inclusion/Exclusion Criteria for guidelines: We defined a set of inclusion and exclusion criteria for the guidelines to be kept in our final set of sources. These criteria were used to filter the candidate guidelines and ensure consistent inclusion by all authors.

- **IC1**: The guideline must be related to human-readable artifacts;
- **IC2**: The guideline must explicitly mention which type of information it requires, i.e., its prerequisites, to be applied;
- **IC3**: The guideline must be usable for informal code review, i.e., it should not require a highly structured process;
- **EC1**: The guideline is specific to an approach that does not produce human-readable test artifacts;
- **EC2**: The guideline is tool-specific and cannot be generalized to other tools or frameworks;
- **EC3**: The guideline does not provide information about the artifacts but refers to the test process, environment, or actors.

The collection phase allowed us to obtain a set of 1128 candidate guidelines (624 from WL, 504 from GL) that include duplicates. After applying the inclusion criteria to the original set, we obtained a final set of 539 candidate guidelines (246 from WL, 293 from GL).

Among the Inclusion Criteria, IC3 (or the related EC3) proved to be the most selecting, leading to the rejection of 84% of guidelines extracted from WL and 90% of guidelines extracted from GL.

Data Analysis and Coding

After collecting the WL and GL sources, the final phase of the research design was to analyze the collected sources. This was done through coding to (1) identify the most mentioned review guidelines for source and test code and (2) perform synthesis and mapping of the identified guidelines for GUI-based testing artifacts.

The procedure used in this phase was inspired by the Straussian Grounded Theory approach [132], more specifically the coding procedure they propose. Grounded Theory is a data-driven approach to construct a theory from raw data (i.e., data mined from the sources in our final set). The Straussian definition of Grounded Theory differs from the original definition of the technique
3.3 Study Design

(i.e., Glaserian Grounded Theory [133]) since it allows the definition of research questions up-front, before starting analysis of the raw data.

The Straussian definition of coding consists of two separate and consecutive steps, open and axial coding. For this work, coding was described and performed as follows:

**Open Coding.** In the first step of this approach, the captured text data is analyzed, line by line, to capture the main concepts of a theory under construction and identify possible overlaps between data from different sources in the analyzed set. The application of open coding allows the researcher to create categories, defined by codes associated with guidelines, and then cluster the guidelines under these categories. This clustering is made based on the semantic meaning of the data, i.e. guidelines that have the same or similar semantic content are clustered together. The result of this coding is, therefore, a set of common definitions of guidelines.

We applied the following operations to each guideline in our final set of sources: (1) we searched for a code in the current set of defined categories that is semantically compatible with the current guideline under analysis. If a suitable code is present, the guideline is assigned the existing code; (2) if no code in our current code set is semantically suitable for the analyzed guideline, we create a new code and assign this code to the guideline. Thus, the set of codes grows naturally as more guidelines are analyzed.

For guidelines that were semantically ambiguous, the authors resorted to expert judgement to determine if (1) an existing code was suitable, (2) if more than one code was suitable or (3) that no code was suitable. Open coding was performed independently by two of the paper’s authors through individual passes of the filtered set of guidelines. After the individual passes, meetings were held to obtain a consensus on code assignment to mitigate researcher bias.

**Axial Coding.** In the second step of the approach, as described in the Straussian Grounded Theory, the purpose is to understand how individual codes and related concepts are linked together. Hence, the goal is to identify a structure in the theory that is being built. In this study, after the codes were assigned to categories, axial coding was performed to find macro-categories of related codes and define a hierarchy of codes. These hierarchies were then used for the synthesis of the final set of guidelines.

We applied the following operations to each coded guideline in our set of categories: (1) we searched for an existing macro-category suitable to
include the currently analyzed code. If such macro-category is present, the category is assigned to it; (2) if no suitable macrocategory exists to include the currently analyzed code, a new macro-category is created.

Similar to the first step (i.e., open coding step), two of the paper’s authors performed individual passes on the codes and assigned a macro-category to each code. After the individual passes, meetings were performed by using the mini-Delphi approach to obtain a consensus on what macro-category to assign for each code.

In this work, the open coding phase resulted in a set of 33 codes. These codes, after axial coding, were organized into nine higher-level codes, representing artifact review guideline categories in this work.

Finally, using the hierarchical structure of raw data (i.e., guideline definitions), codes (i.e., clusters of semantically equivalent definitions) and categories (i.e., clusters of types of common guidelines) the mapping was done to GUI-based testing artifacts. This was achieved by using the aforementioned information to formulate examples of how the guidelines apply to GUI-based testing. These examples were taken from literature or formulated by the authors based on their knowledge and experience with GUI-based testing. They were later presented as examples of application in the result section. For instance, for guideline G9.2 Don’t repeat yourself (DRY), the page-object design pattern was identified in the literature and provided as an example to apply this guideline on GUI-based testing artifacts. Both DRY and the page-object pattern aim to avoid code duplication to improve the maintainability of production and test artifacts.

3.4 Results

In this section, we will present our results on used guidelines for source and test artifacts to answer our research questions. At first, we provide a macro analysis of the gathered white and gray literature sources, which includes the distribution of literature per year and a description of how guidelines from white and gray literature overlap. These results, combined with test-specific guidelines that are marked in Table 3.3, provide an answer to RQ1.

Next, we present the categories of used guidelines in Table 3.4, followed by a structured textual format with descriptions and examples of the application to GUI-based test artifacts, summarized in Table 3.5, to answer RQ2.
Figure 3.2: White and gray literature publications per year
3.4.1 Macro Analysis

In this section, we present some bibliometric data and other macro observations based on the literature that was collected during the literature review.

After applying the search string and removing duplicates, a total of 1160 white literature sources and 378 gray literature sources were acquired. After further filtering, by applying the inclusion and exclusion criteria for the two sets, we were left with 539 (246 white literature sources and 293 from gray literature sources) candidate guidelines.

An observation made on the number of sources from white versus gray literature was that the number of supporting sources was greater from gray literature. One reason for this was that the gray literature sources often provided concrete lists of guidelines associated with reviewing, thereby complementing several of the guideline categories at once. In contrast, many of the white literature sources did not necessarily focus on guidelines, i.e., only provided guidelines as a byproduct of other research, or only focused on single guidelines. Thus providing more in-depth results compared to the gray literature.

Figure 3.2 presents the distribution of the included white and gray literature publications per year. As can be seen, the first source that was sampled was published in 1995, implying that this has been an area of research for more than 20 years. We also note that this type of research saw an increase in interest around 2013. We also note an increasing trend in the number of published papers every year, indicating that it is still an area of interest. Notably is that the number of gray literature references is also increasing over time, implying that there is also an industrial interest in guidelines of this type. This conclusion is based on the fact that many of the gray literature sources were industrial blogs and from tool vendors’ own documentation rather than academic sources.

It is worth highlighting that the application of our IC and EC for literature and guidelines excluded from our final pool of sources the sources proposing or discussing guidelines about the code review process, the related best practices, and involved actors. A relevant amount of the works available in the literature about code review is in fact related to the code review process. Therefore, the numbers visualized in the graphs only represent a subset of the whole corpus of literature discussing code review.

Furthermore, it must be considered that IC1 for literature required the explicit mention of code review for a source to be included in our final pool. This means that sources about the traditional practice of code inspection, that did not mention (modern) code review, are not considered in Figure 3.2.
3.4 Results

Regardless, the trend indicates a growing interest in both industry and academia to provide guidelines for modern code review.

The literature review resulted in 33 guidelines, categorized into nine guideline categories applicable to GUI-based testing artifacts. Out of the nine categories, only seven were supported by white literature, implying that there are practices in the industry not conceived by the academia. The two categories of guidelines that were introduced through gray literature are *check conformity with the requirements* and *follow design principles and patterns*. Furthermore, the gray literature represents a super-set of the sample, including all guidelines from white literature.

Analysis of the content of identified sources also showed that the vast majority of guidelines refer to reviews of source code. In fact, only 10 of the white literature sources, out of 137, mention test-specific review guidelines. Whilst it can be argued that code guidelines on a higher level of abstraction are also applicable to test code, this result indicates a lower research focus on test-specific guidelines.

References for all sources where guidelines were taken from are documented in Table 3.6 for white literature and Table 3.7 for gray literature in the appendix.

3.4.2 RQ1: What are guidelines used for source and test artifact review?

Synthesis of the two literature sets, i.e., white and gray literature, acquired from our SLR, resulted in 33 guidelines in nine categories for review of artifacts. These guidelines are presented in Table 3.3, and a categories-only summary is shown in Table 3.4. In Table 3.3, we report which guideline was defined as specific to test artifacts. As a notable result, we find that only 10 sources of those we analyzed specifically described guidelines for the review of test artifacts. Further, the table presents how many occurrences of each guideline appeared in white and gray literature with references to the sources.

The main purpose of these guidelines is to guide both the contributor of code changes and the reviewer to improve the effectiveness and efficiency of the code review process. On the one hand, efficiency is improved by providing all information required by the reviewer to comprehend the code changes. Thus, mitigating the need for reviewers to spend time gathering this information themselves. Moreover, automated checks reduce the manual effort and prevent low-level discussion about personal code style opinions. On the other hand, effectiveness is improved by summarizing relationships, metrics, and behavioral changes of code changes in a way that allows the reviewer to provide
more useful feedback. Combined, these guidelines aid the reviewer by reducing cognitive complexity and effort, allowing the reviewer to focus on identifying faults and improvement potential. Thus, providing the code contributor with better feedback, which results in a better quality of the resulting artifact.

Notably, we see that the most mentioned guideline category (Table 3.4) from both literature sets is to provide context information to a review request. This result can be explained by the cost-value of adding such information, where cheap practices, e.g., linking a review request to the implemented requirements, can significantly improve the reviewer's understanding of the review. A similar explanation of cost-value can be given for the second most mentioned item, reducing complexity. Whilst this is not necessarily a cheap practice, since it may be complicated to achieve, its impact in terms of value is high, since complexity affects the readability, understandability and maintainability of the artifact. Notably, the guideline category with the most significant variance in reference support between white and gray literature is the one to ensure readability. This result is surprising since this guideline is an important aspect of dealing with complexity. From the gray literature only guideline categories, the one regarding using established software engineering design principles has a high reference support. As described earlier, white literature often provides guidelines as a byproduct of other research and which could explain why established design principles or patterns are mentioned as a contribution.

Answer to RQ1: All identified code review guidelines used for source and test artifact review are presented in Table 3.3, and a categories-only summary is shown in Table 3.4. We consolidated our findings of 33 guidelines in nine categories, where seven categories emerged from white literature and two from gray literature. Nonetheless, the main purpose of these guidelines is to guide the contributor of code changes and the reviewer to improve the effectiveness and efficiency of the code review process.

3.4.3 RQ2: To what extent can source and test artifact review guidelines be mapped for GUI-based test artifact review?

In the continuation of this section, we present a detailed description of each guideline. Table 3.5 presents the mapping of the identified guidelines to GUI-based test artifacts. We structure this section according to the categories presented in Table 3.4 to answer RQ2. For generalizability, the level of abstraction
of each guideline has been kept on a higher level to make them applicable to different GUI testing techniques and tools. The presented guidelines are inherently generalizable to all GUI-based testing artifacts, since they are provided as a generalization of general-purpose code review guidelines originally specified for source or generic test code. Code reviews for specific aspects of GUI-based testing are therefore not to be found in the results discussed in this section. Each guideline category is presented according to a structured textual format that explains (1) the purpose of the guideline category, (2) guidelines of the category and (3) examples of application for GUI-based tests. In the explanation of each guideline, a reference to the guidelines in Table 3.5 is provided by adding the corresponding ID in parentheses. Each guideline has also been ranked in terms of if it’s a guideline that is suggested, recommended, or strongly recommended to be followed, based on reference materials. Suggested guidelines are in this context viewed as beneficial but not necessary, recommended guidelines are rated as beneficial by multiple sources and could have adverse effects if not applied and strongly recommended are suggested by multiple sources to have significant benefits or tangible adverse effects if not applied.

Table 3.5: Overview of code review guidelines and their mapping to GUI testing

<table>
<thead>
<tr>
<th>ID</th>
<th>Guideline</th>
<th>Mapping to GUI-based test artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td><strong>Perform automated checks</strong></td>
<td></td>
</tr>
<tr>
<td>G1.1</td>
<td>Perform automated checks on the change</td>
<td>Apply static code analysis tools to identify unsuitable or incorrect patterns in the code</td>
</tr>
<tr>
<td>G1.2</td>
<td>Perform automated checks for code style</td>
<td>Apply static code analysis tools to review and format code according to a style guide</td>
</tr>
<tr>
<td>G2</td>
<td><strong>Use checklists</strong></td>
<td></td>
</tr>
<tr>
<td>G2.1</td>
<td>Provide/use a checklist</td>
<td>Provide/use a checklist that is tailored to the company or team that cover aspects related to GUI element localization, test oracles or synchronization</td>
</tr>
<tr>
<td>G3</td>
<td><strong>Provide context information</strong></td>
<td></td>
</tr>
<tr>
<td>G3.1</td>
<td>Provide additional context information</td>
<td>Provide contextual information about the code change as a comment during code review or as a comment or annotation to the artifact</td>
</tr>
<tr>
<td>G3.2</td>
<td>Provide rationale for the change</td>
<td>Provide contextual information about the motivation why the changes are valid for the purpose of the test</td>
</tr>
<tr>
<td>G3.3</td>
<td>Provide context information about the impact of the change</td>
<td>Provide contextual information as a comment during the code review about how the change will impact other reusable test cases or components that are dependant on the change, e.g., how many test cases depend on a changed functionality</td>
</tr>
<tr>
<td>G3.4</td>
<td>Provide information about the design and architecture of code affected by the change</td>
<td>Provide contextual information about the logical and chronological behavior of the GUI tests</td>
</tr>
</tbody>
</table>

Continued on next page
### ID | Guideline | Mapping to GUI-based test artifacts
--- | --- | ---
G3.5 | Provide links to related resources and documentation | Provide list of references (links) as comment during code review, e.g., about a new introduced testing technique
G3.6 | Provide information about the dependencies between test and production code | Provide a reference between test cases and the corresponding production code
G3.7 | Provide context information about the history of changes | Provide a list of references (links) as a comment during code review to previous code reviews or merge requests if an artifact has been changed many times
G3.8 | Provide information about test edge cases | Provide contextual information about edge cases as comment during the code review to inform the reviewer about critical paths of the test case
G3.9 | Provide a prioritization of the files/classes of the change | Provide prioritization of test cases as a comment during code review if multiple test cases are part of the code review and review time is limited

### G4 Utilize metrics
- G4.1 Measure and monitor code metrics
- G4.2 Provide metrics about execution time (for efficiency)
- G4.3 Provide test coverage metrics (for effectiveness)

- Measure and monitor code metrics on test cases like the complexity, coverage, or run-time
- Measure execution time of test cases
- Measure feature-, scenario- or GUI-element coverage

### G5 Ensure readability
- G5.1 Ensure readability of the change
- G5.2 Provide comments
- G5.3 Follow coding style and naming conventions
- G5.4 Follow coding style and naming practices in test writing
- G5.5 Avoid code comments if they are not clear and useful
- G5.6 Ensure proper usage of techniques for testing and exception handling
- G5.7 Ensure correctness of assertions in test cases

- Ensure readability by following code styles, norms, and conventions
- Comment and annotate GUI-based test artifacts to explain difficult aspect and give the reviewer insights into the test creator’s reasoning behind the tests
- Script-based GUI tests should follow a set coding style, norms, and naming convention
- Script-based GUI tests should follow a set coding style, norms, and naming convention
- Ensure comments and annotations are helpful to the reviewer and provide additional information that can not be easily derived from the test artifact itself
- Check for improper use of testing techniques, such as incorrect usage of mocks, testing on the wrong level of abstraction or wrong variable initialization
- Check that assertions are following established test patterns

### G6 Visualize changes
- G6.1 Provide a visualization of the change
- G6.2 Provide a visualization of the change regarding its impact on the code base
- G6.3 Allow traceability and easy navigation between artifacts

- Provide a graph representation of visited GUI states of the SUT
- Presenting screenshots, or screen recordings, of the GUI states that are covered by the test
- Provide references to dependent tests and libraries and allow following the references as links if possible

### G7 Reduce complexity
- G7.1 Keep size of a change as low as possible

- Apply patterns and to minimize the use of branching scenarios to keep scripts as short and focused as possible

Continued on next page
### 3.4 Results

Table 3.5 – continued from previous page

<table>
<thead>
<tr>
<th>ID</th>
<th>Guideline</th>
<th>Mapping to GUI-based test artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>G7.2</td>
<td>Keep complexity of a change as low as possible</td>
<td>Apply patterns and to minimize the use of branching scenarios to keep scripts as short and focused as possible</td>
</tr>
<tr>
<td>G7.3</td>
<td>Avoid unrelated and unstructured changes</td>
<td>Avoid mixing code changes with unrelated changes that do not fit into the scope of the artifact change for review, e.g., code style changes</td>
</tr>
<tr>
<td>G8.1</td>
<td>Ensure conformance with the requirements</td>
<td>Review the changed test scenario in terms of conformance with the requirements</td>
</tr>
<tr>
<td>G9.1</td>
<td>Apply established design principles and patterns</td>
<td>Apply established design principles and patterns for source code and explicit ones for GUI testing</td>
</tr>
<tr>
<td>G9.2</td>
<td>Don’t repeat yourself (DRY)</td>
<td>Extract common functionality into reusable components and reuse them throughout the test. Apply design patterns such as page-object pattern</td>
</tr>
<tr>
<td>G9.3</td>
<td>Avoid hardcoded values</td>
<td>Avoid hardcoded values, such as hardcoded identifiers</td>
</tr>
<tr>
<td>G9.4</td>
<td>SOLID principle</td>
<td>Avoid unrelated changes to confirm with the single responsibility principle</td>
</tr>
</tbody>
</table>

### G1 Perform automated checks

**Purpose and description:** The purpose of performing automated checks is to improve the efficiency of the review process by reducing manual effort. This allows the reviewer to focus on the understandability and maintainability of the code changes instead of getting distracted by revising low-level code issues [79]. These automated checks are either done by the submitter before the code review or handled by a Continuous Integration (CI) infrastructure when code changes are submitted for a code review.

**Guidelines:** Automated checks can be applied in different ways, for instance, through the application of static code analysis (G1.1). Static code analysis is an approach that seeks to find unsuitable or incorrect patterns in the code. In addition, most static code analyzers also provide metrics that can be used to determine the quality of the code being analyzed, e.g., cyclomatic complexity. The code analyzers come with default rules and patterns that they analyze, but since these may not align with the code developed in the applied context, they often require context-dependent tailoring. Failure to do such tailoring, which requires both domain and technical knowledge, can otherwise result in false positive results.

A specific type of static code analysis is regarding the analysis of the code style (G1.2) of source code and script-based tests. An inconsistent code style, like different indentations for code blocks, reduces the readability and obscures behavioral changes when reviewed as one set of changes. Since good tool support exists to detect and automatically fix code style-related issues, the developer
should prevent these low-level issues before an artifact is submitted for a code review.

Ad hoc automated checks can also be created, for instance through context-dependent scripts, e.g., automatically check that all test-related artifacts are available and that testing libraries are up to date. These ad hoc solutions can also be required when the tests are not script-based, e.g., model-based with visual models or capture-replay tools.

Regardless, it is strongly recommended to automate as many of the low-level review tasks as possible to reduce manual effort. However, and additionally, care shall be taken to how the results of the automated checks are presented to the reviewer—Cognitive complexity of the resulting analysis shall be kept to a minimum.

Example(s) of application: In many instances, existing tools can be applied to assist reviews for GUI-based test, with similar benefits to source code. For example, similar to source code, GUI-based tests shall be consistent to improve their readability. This is normally achieved by following test code patterns like the setup-test-teardown or page-object patterns (G1.1). A static code analyzer can be instructed to find such patterns in the code and warn the reviewer if test code strays from the defined pattern. Similarly, naming conventions can be targeted and automatically presented to the reviewer, or even developer, to avoid human error of them being overlooked (G1.2).

G2 Use checklists

Purpose and description: The purpose of a checklist is to guide the reviewer during the code review, outlining either specific activities to be carried out during the review or artifacts to be reviewed. Guidelines thereby provide a powerful, context-dependent, tool for reviews and are therefore strongly recommended for most types of artifact reviews, including GUI-based tests. The exception to this rule is when the reviewing activity has been so thoroughly embedded in the organization that its need becomes superfluous. Especially junior developers, without code review experience, benefit from checklists [77]. However, the guidelines shall take the domain, company, team culture and other environmental aspects into account [40].

Guidelines: Providing and using a checklist (G2.1) may have different origins, depending on the purpose of the checklist. For instance, if providing guidance to what activities to perform during the review, similar to a definition of done, the guideline can be set on an organizational level, providing a roadmap for the review with links to further reading. As an alternative, the checklist can be provided by the submitter of a code change, listing specific artifacts or
explicit aspects of the code that – in the submitter’s perception – require a more detailed review. Checklists can also be automatically generated based on code changes.

However, based on our included literature, we could not identify any specific items that a checklist should always contain for source- and test code artifacts. Neither could we identify any example checklists for non-GUI or GUI-based testing, although checklists for source code are available. One such example is proposed by GitLab, which encourages checking for performance, security, reliability, observability, and maintainability risks [102].

*Example(s) of application:* As stated, no generic checklist items could be identified in the review. However, checklists focused on activities could include analysis of set naming conventions, compliance of test patterns, and aspects of modularity and reuse, but should also include analysis of code complexity, readability and maintainability. Thus, complementing review aspects to the automated checks.

Checklists instead derived by test developers can outline aspects related to GUI element localization, test oracles or synchronization (G2.1). For instance, the reviewer could be tasked to verify that the test scripts execute as intended in the viewer’s development environment. Alternatively, the reviewer could be tasked with the verification that the chosen strategies to identify the chosen GUI element are suitable as oracles, in terms of test robustness or compliance to requirements. At a high-level perspective, it can be asked if the GUI element used in an assertion is suitable given the purpose of the test. At a low-level, it can be asked if the XPath locator is valid in a changed context.

### G3 Provide context information

*Purpose and description:* The purpose of contextual information is to aid the reviewer in understanding the changes made to the project, be it requirements, code or tests. This type of information is argued to reduce the reviewer’s cognitive load and thereby improve the effectiveness—Identification of errors or improvements—of the code review [71]. Additionally, providing such information has positive effects on the efficiency of the review, since it mitigates the need for the reviewer to spend time gathering this information for themselves [71].

*Guidelines:* The test developer is responsible for providing sufficient contextual information for the review to understand the change G3.1. This information belongs to one of two categories, either technical knowledge—Information about the test code that has been developed or its artifacts—or domain knowledge—Information about requirements, the application or its usage.
From the literature, the main type of contextual information that shall be provided is a rationale for the change (G3.2)—A motivation for the change, its cause and effect. This information aims to address the knowledge gap that is presented when a change occurs, since what has changed is easily seen, but why the change occurred may be less evident. Hence, the rationale serves to provide an answer to why a change has been made, which for GUI-level tests generally relates to domain aspects, such as changed requirements. These requirements may however be of both a functional, e.g., features or functions of the SUT, or non-functional nature, e.g., quality attributes of the SUT such as its performance or security or, for GUI-level tests, also the SUT’s visual appearance or behavior—Changed bitmaps or modes of interaction.

The test developer also must provide information about the impact of the changes (G3.3). For instance, if the changes have impacted the test coverage, test design or the architectural design of the testware (G3.4). The latter aspect is seldom discussed for GUI-level tests, but, as discussed by Sutherland et al. in the context of software, a lack of such discussion in reviews can lead to technical or architectural debt in the codebase [137].

Another type of information that can help the reviewer understand the context of a change includes documentation (G3.5), or links, to the tested functionality (G3.6) or similar tests—Alternative test solutions—, previous code reviews of the artifact (G3.7), testing edge cases (G3.8), and other resources related to the code change. These resources shall aim to provide the reviewer with insights into how the test developer produced the solution to more easily judge its correctness and quality.

For larger changes, which may stretch across several test artifacts, it is also suitable that the test developer provides a prioritized list of the files that have been changed (G3.9). This allows the reviewer to focus on the important or critical parts first in their review, which provides additional contextual insights. For time-critical reviews this guideline also helps improve review effectiveness since the more important changes are reviewed first.

We rank this guideline category as recommended and its guideline of providing a rationale as strongly recommended.

Example(s) of application: For GUI testing, from a domain perspective, the test developer shall describe the changes to the requirements that lead to the test code change (G3.1). This includes motivating why the changes are valid for the purpose of the test, i.e., motivation for why the new test is suitable to test the changed requirement(s) (G3.2).

From a technical perspective, two aspects need to be considered; (1) the logical and (2) the chronological behavior of the GUI tests (G3.4). For instance,
if the GUI of the SUT has been updated, new GUI elements may have been required in the test and they should be reviewed for contextual correctness. Similarly, the changed test scenario needs to be reviewed in terms of conformance to the requirements, including edge cases (G3.8). The use of external, or reusable, artifacts shall also be reviewed (G3.3, G3.5, and G3.6). In cases where test scenarios were changed often, the test developer shall provide references to previous reviews to reveal its history as additional context (G3.7).

For changes to the chronological behavior—Chronological behavior is defined by synchronization between the SUT and the test cases— it is necessary to review that they are suitable. Suitability in this context concerns that the synchronization steps do not wait too long, which would introduce unnecessary overhead to the test execution. However, the waits must also not be too short, since this would increase the probability of false positive test results. When many test scenarios are changed, a prioritization of critical test scenarios should be provided as a comment, assuming that the reviewer’s time is limited (G3.9).

**G4 Utilize metrics**

*Purpose and description:* The purpose of utilizing metrics in code reviews is to monitor the effects of change, predict the impact of code changes on the code base and support decision-making in the review process. Metrics are thereby an additional source of information to gain insights into the effects of the changes of artifacts—Effects include changes in complexity, performance or breach of patterns. This measured information enables a fast feedback loop for the contributor and allows the reviewer to give more detailed feedback based on this quantifiable information.

*Guidelines:* Measuring and monitoring different types code metrics can be performed during the development or review process to give inputs to the artifact reviewer (G4.1). These metric types can be roughly clustered in terms of change impact on artifact complexity, efficiency, and effectiveness. Metrics regarding the impact of a change on the complexity of the reviewed artifact can be acquired from static code analysis tools—Static code analysis is performed as automated checks—and are valuable to provide insights regarding the readability, understandability and maintainability of the modified artifact. Examples of possible complexity metrics include cyclomatic complexity, the number of files in a solution, the size of single files or functions, sum of added and removed lines of code (i.e., code churn) and dispersion of modified lines across files. In
the sampled literature, these metrics were also used as predictors of introducing defect-prone code to the code base [135].

One way to monitor the efficiency of changes, be it source or test artifacts, is to monitor the execution time (G4.2) of artifacts—By monitoring we include execution time before and after change. For test artifacts, the execution time can be measured on different levels of granularity. On a high level, the execution time of the overall test suite provides an overview, whilst lower-level, more detailed measurements of time spent in functions, test cases, or single GUI screens, can help identify bottlenecks. For source code, poor performance can be an indicator of poor code quality [24]. In the same way, poorly performing test cases should be investigated by the reviewer for test quality issues.

Measuring coverage metrics such as code, path, or GUI coverage allows insights into the effectiveness of test artifacts (G4.3). Increased coverage is correlated with increased fault-finding behavior and is therefore one of the main attributes the reviewer of a test artifact must investigate. This entails acquiring an understanding of whether the test artifacts cover all, or at least all necessary, paths of the production code [83].

Example(s) of application: The complexity of GUI-based tests shall also monitored (G4.1). Depending on the frameworks used to develop the tests, this monitoring can be done with the same tools as for source code. For instance, tools like Selenium allow the test developer to write test cases in source cod, or use test libraries such as XUnit [95, 112]. Such tests can be analyzed using the same static-code analyzers as the source code and provide similarly effective results. For tools with custom scripting languages or IDEs, such as Graphwalker [113] or Scout [67], conventional solutions to measure complexity are not applicable. Instead, the reviewer needs to manually evaluate the perceived complexity, in particular taking into account the perceived readability and understandability of the tests. These attributes are important since they affect the maintainability of the tests, which, in turn, affects the tests’ longevity.

The execution costs of automated GUI-based tests are more expensive compared to low-level unit tests since the tests are more computationally heavy. In fact, the execution time of a GUI-based test is generally several orders of magnitude more costly than, for instance, a unit test [9]. Therefore, information about the execution time and the required resources of these test cases are important to estimate the scaleability of the test approach (G4.2). In particular, when these tests are executed as part of a continuous integration environment, where allotted time for test execution is limited and test prioritization or selection is therefore required.
Furthermore, in the same way as for poorly performing source code, a GUI test case with a long execution time, compared to other similar GUI test cases, could indicate inefficiencies in its design. Inefficiency could include an unnecessarily long or complex test scenario, unsuitably long synchronization checkpoints, or inefficient GUI element identification. Hence, quality attributes of the test artifact that the reviewer must investigate.

Measuring test effectiveness in terms of coverage is also an important aspect of GUI-based tests (G4.3). However, due to the level of abstraction these tests operate, low-level coverage metrics, like code coverage, can be difficult to calculate. Solutions include instrumentation of the SUT or using third-party measurement tools, e.g., JaCoCo [105] or Cobertura [106]. More commonly used coverage metrics are instead feature, scenario, or GUI element coverage. Whilst these metrics don’t provide insights of the same granularity, they are useful to determine the effects of a change to the tests.

Whilst many of the aforementioned metrics provide direct insights into the quality characteristics of the GUI tests, some may not. The former types can be acted upon immediately, e.g., if coverage is found to be low, whilst the latter instead serve to monitor trends, or changes, in the GUI tests’ quality. Both are useful for the reviewer as they provide insights into the effects of the reviewed change that may not be acquired from reviewing the artifact in isolation.

Historical data on the number of found faults, false positives and negatives, shall also be used. These metrics are unique to tests, but give insights into the tests’ behavior and priority. For instance, it can be assumed that tests of older age, but still have a high failure rate, cover important or central parts of the SUT that are also subject to continuous change. Since such code is central to the SUT, more effort shall be spent on reviewing these tests.

We rank this guideline as recommended since it provides a more objective input to base feedback on.

G5 Ensure readability

*Purpose and description:* The purpose of ensuring readability is to improve the ease of understanding, and thereby maintaining, an artifact over time. More formally, Buse and Weimer [138] defines readability “as a human judgment of how easy a text is to understand”. As such, the readability of an artifact has a direct impact on the reviewing process itself, where an artifact with low readability is perceived as more difficult to review than an artifact with high readability. Code reviews shall strive to identify the level of readability of an artifact and suggest improvements. For example, at Google, code reviews were introduced to ensure code readability and maintainability, where readability is
supported by the use of a consistent code style within the code base Sadowski et al. [79]. Therefore, we rank this guideline as strongly recommended.

Guidelines: One way to improve the readability of code is to follow common coding styles, norms and naming conventions (G5.1). A team should enforce a coding style within the team to prevent low-level discussions about individual styling preferences during a code review (G5.2 and G5.3). A style guide is a set of conventions for how code shall be structured in a project, including how indentations shall be done, variable naming convention (i.e., use camel-case), restrictions on line length, or the ban of specific operators. For instance, Google has style guides for all major programming languages that developers must follow for all application developments [103]. Code consistency also lowers cognitive complexity for the reviewer, making it easier to focus on the artifacts conformance to the intended behavior. Consistent code can also more easily be automatically checked through, for instance, static code analysis or other automated checks.

Another way to improve readability is to annotate important or more difficult parts of the artifact. Annotations can take the form of comments or other supplementary materials (G5.4). An example of the value of annotation is provided by An et al. [144], where crash-prone code with a low ratio of code comments was found more difficult to review. However, annotations shall only be added when they are useful (G5.5), i.e., they must be helpful to the reviewer by providing additional information that is difficult to derive from the code itself. Mindless addition of annotations can otherwise lead to “clutter” in the artifact, which decreases its readability. The annotations must also be fit-for-purpose, since they can otherwise add ambiguities that confuse the reviewer, e.g., if supplementary materials are added that are not consistent with the reviewed artifact itself.

For test artifacts it is also important to consider the fit-for-purpose of the artifact, and to use the most suitable technique or pattern for the test purpose (G5.6). Similarly to coding conventions, using familiar techniques or patterns makes it easier for the reviewer to comprehend the changes. Furthermore, improper use of testing techniques, such as incorrect usage of mocks, testing on the wrong level of abstraction or wrong variable initialization, may lead to faulty test results [83]. For instance, if a technique is on the wrong level of abstraction for the test purpose, it may not be able to identify a fault. In addition, test assertions should also follow suitable patterns for the test purpose (G5.7). It is therefore important that the reviewer also evaluates the adequacy of the artifact for the purpose, and provide recommendations if this is not aligned.
3.4 Results

**Example(s) of application:** Due to the benefits of coding conventions, script-based GUI tests should follow a coding style, norms, and naming convention (G5.1). These conventions can, if applicable, be the same as the tested software to maintain consistency and readability across the entire code base (G5.2 and G5.3). When existing conventions can not be reused, new conventions shall be implemented. Examples where new conventions may be required are GUI-based tests with custom scripting languages or tests that are derived from models. In these instances, some coding conventions may not be applicable, but consistent naming conventions should at least be upheld by all test-related artifacts.

Similar to source code, annotating GUI-based test artifacts can help explain non-intuitive parts of the tests (G5.4). These comments shall also provide the reviewer with insights into the test creator’s reasoning behind the tests. However, authors should not add comments if the comment does not provide additional insights since this can lead to “clutter” (G5.5). Supplementary materials, e.g. requirements, shall also be provided when necessary to aid the reviewer.

A unique aspect of test code that reviewers must investigate is the test assertions (G5.6 and G5.7). These need to be clear, from a readability perspective, so that the reviewer can determine their correctness and fit-for-purpose. The reviewer must also evaluate that the assertions are placed in suitable sections of the test scenario [83]. Additionally, the reviewer must evaluate the assertions are placed following established test patterns.

**G6 Visualize changes**

**Purpose and description:** The purpose of visualizing code changes is to ease the understanding of the change and its impact. Visualization also allows for easier navigation between related artifacts and their dependencies. A study by Baum et al. [22] identified that reviewers often miss features in IDEs (Integrated Development Environments) for code review of non-trivial changes. They conclude that tool support would mitigate this challenge and improve both the effectiveness and efficiency of code reviews. Furthermore, for review of test artifacts, many code review tools lack test-specific information and only present changes as line-by-line comparisons. This forces the reviewer to open the artifacts in their local environments, e.g., IDEs, to be able to navigate through the dependencies and acquire a full picture of the change [83]. Using tools that have stronger visualization solutions helps to mitigate these challenges.

**Guidelines:** First, a visualization shall be provided of the files affected by the change and the dependencies between these files and their changes (G6.1). These dependencies between files help a reviewer to understand which code change(s) may have the biggest side effects due to its depending components.
Thus, providing insights into what files shall be prioritized in the review and the order they should be reviewed in. Common to both source and test code, changes in a file whose functionality is reused in multiple instances can potentially cause more side effects. This visualization should also indicate the impact of the change on the rest of the system, e.g., a Change Impact Analysis [139]. Hence, although visualization of changed technical artifacts is a minimum, additionally affected artifacts, e.g. source code, test code, requirements and libraries, shall also be noted.

For more assistance, semantic changes and relationships shall also be presented in addition to the textual line-by-line changes [139] to determine the impact of a change (G6.2). Line-by-line differences are usually based on the underlying GIT or Unix tooling, and although they allow the reviewer to pinpoint the exact changes, they do not provide any insight into the behavior change caused by such changes. For instance, renaming a variable doesn’t affect the behavior of a program, and therefore it doesn’t change the semantic relationship. However, changing the algorithm that uses said variable may result in completely different software behaviors. Behaviors that may be non-compliant and not visible in the code.

Visualization helps the reviewer to more easily navigate between different artifacts, but it still requires the reviewer to open these artifacts, often in different tools. Using review tools that more easily help the reviewer navigate, and help ensure traceability, between different artifacts are therefore helpful (G6.3). For instance, mitigating the need for a reviewer to switch between source and test code in different windows during a review. As pointed out by Spadini et al. [83], artifacts are usually coupled to multiple other artifacts and contextual switches between sources thereby creating additional cognitive load.

The type of visualization also matters but what visualization is the most suitable is often context dependent. For instance, when dependent changes have occurred to multiple files, a class-like diagram may be suitable. However, when changes instead affect the behavior, a state-diagram, or tree-like structure, may be more adequate.

*Example(s) of application:* For GUI-based tests, a graph representation of visited GUI states of the SUT could help the reviewer to get an overview of the changes to the GUI test scenarios (G6.1). Such visualization can also help the reviewer understand semantic changes to the tests. One example use case is when the test cases are generated in a stochastic manner by the GUI-based testing tool based on some stopping condition. This functionality means that repeated tests may result in slightly different outcomes. Hence, an inherent function of MBT-based GUI tests where the graph acts as a meta-model for
the test cases. Such visualization would be beneficial for scripted test solutions, especially when they are data-driven.

Furthermore, for scripted solutions that rely on technical locators, e.g., XPath locators, it may be difficult for the reviewer to relate the locators to visual elements. Presenting screenshots, or screen recordings, of the GUI states that are covered by the test is therefore a useful practice (G6.2). This type of visualization is inherent to computer vision-based testing (Third generation) tools, but for first- or second-generation tools such information is generally omitted.

Common to both source and test code, convenient navigation capabilities between different related artifacts—Artifacts are in this case other test cases or dependent libraries—would reduce the mental effort to find the different artifacts. Whilst tooling can make the navigation very simple, an initial step is to reference-dependent artifacts in comments of the test code (G6.3). In conjunction with suitable naming practices, this practice can improve review efficiency. Similarly, references to dependent tests and libraries shall be presented when a new code change is submitted for review, i.e., change impact of the test code change.

We rank this guideline as suggested. While we see the benefits of visualizing changes, its implementation will depend highly on tools used for GUI testing.

**G7 Reduce complexity**

*Purpose and description:* The purpose of reducing artifact complexity is multi fold. First, reduced complexity is associated with increased understandability of the artifact. Second, complexity has a well-known impact on the defect rate of a software, where more complex components are more likely to have defects [142].

*Guidelines:* The main approach to reduce complexity is to keep code changes for a review as small as possible (G7.1). Small code changes are preferred by reviewer and they can provide more useful feedback compared to big code changes [143]. In addition, complexity should be avoided (G7.2) by having a narrow and well defined scope [143]. Although a development practice, these practices reduces the effort during code review, improving its efficiency and effectiveness.

Another complexity reducing approach is avoiding mixing code changes with unrelated changes that do not fit into the scope of the artifact change for review (G7.3). An example of an unrelated change is to update a test case that is not in the same scope, nor related to the other test cases of the code change. This scenario may occur because the test developer noticed a possible improvement, or applied an improvement from one test to another with a similar issue. In these
circumstances, these changes shall be submitted as individual code changes.

Furthermore, included non-code artifacts like configuration files, test or build results in less useful feedback [23]. Pure style changes—Style changes include changes to naming convention or arrangement of test cases—can be integrated into the code base with a less exhaustive review, since these changes should not introduce any changes in behavior of the code. Mixing such style changes with code changes makes it harder for the reviewer to identify the actual change in behavior.

Example(s) of application: The size and complexity of GUI-based test cases is strongly correlated with the test scenarios that they aim to verify. However, to reduce this complexity and the size, it is suitable to apply patterns and to minimize the use of branching scenarios to keep scripts as short and focused as possible (G7.1 and G7.2), e.g., unrelated features must not be tested in the same scenario. The reviewer of such tests should evaluate that these criteria are fulfilled, which includes checking that patterns are followed, that reusable components are used and that there are no unnecessary dependencies between test artifacts.

Similar to other artifacts, unrelated changes shall be dealt with separately and reviewed independently (G7.3). This implies that, for instance, style-changes or updates to GUI-locators for other tests shall not be submitted together with the tests that are the focus for a particular review.

Based on the number of sources that contributed to this guideline and the known negative implications of complex components, we rank this guideline as strongly recommended.

G8 Check conformity with the requirements

Purpose and description: The purpose of requirements is to provide a specification for the system that is developed. These requirements are thereby the inputs to development but also for the creation of test cases that aim to verify that the implemented system conforms to the specification. Thus, although not the only purpose of a review, reviewers use requirements to check for such conformance and that the code, reasonably, fulfils the intended functional and non-functional requirements.

Although reviewers verify these attributes of an artifact change, it is the authors’ responsibility to ensure that the artifacts that are proposed for a review fulfil the requirements. In modern development environments, the artifacts to go into the review are not restricted to source code, but also automated tests and other artifacts as well, e.g., dependent libraries, models or design descriptions.
3.4 Results

*Guidelines:* The first step of a review, which is in relation to requirements, is to ensure that the submitted code changes are traceable to a requirement (G8.1). Such traces shall be submitted with the review request and/or be stated in the submitted changes to provide contextual information for the reviewer. As part of this analysis, the reviewer shall verify that the requirement that has been implemented is still up to date, i.e., that no changes have been made to the requirement during the development process.

If the requirement is up to date, for new requirements, the reviewer shall check that the submitted code reasonably complies with the requirement. For changed requirements, the reviewer instead verifies that the delta, i.e., the changes, are compliant. This analysis is not restricted to only source code, but should also cover test code or other supplementary materials.

Alternatively, code changes can sometimes result in changed requirements, i.e., requirements are updated after a change to the code. Similarly to code, the requirements shall be submitted for review and verified for correctness. Once verified, the review of code artifacts proceeds as described above, taking any supplementary materials into account.

Specific to test reviews, the reviewer must verify that the test suitably verifies the implementation’s conformance with the requirement. Dependent on the level of abstraction of the requirement, this may entail a set of different actions. The first action is to verify that the test is on a suitable level of abstraction to test the requirement, i.e., fit-for-purpose. Second, the quality of the test itself needs to be reviewed, e.g., whether it provides suitable coverage and if suitable test data are used.

*Example(s) of application:* GUI-level tests, similar to all other tests, shall be traceable to a requirement, or requirements. For instance, if aligned with a use case, a test case can be stand-alone, but in other instances, a test case may cover multiple, related, features. The reviewer shall check for completeness of the test case to cover the intended feature(s) and that traceability information is available (G8.1).

Next, the suitability of the GUI test scenario is evaluated in relation to the requirement. This includes checking both the functional and chronological behavior of the test to verify that it is compliant with the requirement as well as synchronized with the behavior of the SUT.

Note that for GUI-level requirements, unlike lower-level tests, visual requirements may also be consulted to ensure that the GUI elements used in the test case, e.g., for assertions, are up to date and used in a suitable manner. This is particularly important for computer-vision based test cases since the use of incorrect visual elements may otherwise lead to false positive test results. For
older generations, i.e., first or second generation, the reviewer should instead verify that the element locators are correctly defined for the current version of the SUT. For example, for second-generation scripts that use element IDs as locators, these shall be checked that they are up to date.

Since this guideline does not contribute to the efficiency and effectiveness of code reviews in the same way as other guidelines do we rank this one as suggested.

G9 Follow design principles and patterns

Purpose and description: The purpose of design principles and design patterns are to provide consistency among various types of artifacts. Another purpose is to prevent the degradation of the artifacts. For instance, software architectures and libraries may degrade over time as the code is rewritten, maintained or otherwise updated. Principles for software design can prevent these issues and thereby prevent source code fragility and improve reusability, maintainability and scalability of the software [140]. Design patterns are thereby blueprints for building reusable solutions for common problems. These patterns are well proven and have evolved over time [141].

In contrast to the ensure readability guideline, which is based on both white and gray literature, this guideline is based explicitly on gray literature.

Guidelines: Common design principles, including Don’t Repeat Yourself (DRY), SOLID for object-oriented design, and avoiding hardcoded values, should be followed if the context allows it (G9.1). Applying the DRY principle favors the (re-)use of components instead of a duplication of code and effort (G9.2). Hardcoded values, such as hardcoded identifiers, can hinder the deployment of the application in different environments [104] and thus should be avoided (G9.3). The SOLID principle consists of single-responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion. Applying SOLID principles to object-oriented design helps to improve the maintainability of the codebase [140]. Although these are patterns that the developer must consider, it is part of the reviewers’ responsibility to verify that they are followed. These mentioned principles and patterns are only a subset of what is available, and we restrict ourselves to these examples since the patterns themselves are out of the scope of this work. Additionally, their application are context-dependent, where some principles may be used in some companies but not in others. Hence, it is important that both developers and reviewers are aware of context-applicable patterns and if/how they change over time.

Example(s) of application: For GUI-based testing, principles like the usage of dynamic (i.e., wait for a specific visual state) synchronization checks to
avoid unnecessary test code maintenance or automatically capture screenshots of failed actions or state transitions when a test fails for failure replication can be applied (G9.1) [10]. If tests are script-based, principles for source code like DRY and avoiding hardcoded identifiers can be applied. For example, common functionality such as logging into an application is common and should be extracted and reused throughout the test suite rather than to be repeated (G9.2). Further, hardcoded values, such as widget identifier, should also be avoided (G9.3). The reason is that if this functionality or values changes, it will have to be updated in multiple scripts. Reviewers can aid the script authors in identifying reusable functionality that can be extracted, or point the authors to reusable components when such are missed during development.

There are also specific patterns for testing. One such pattern, explicit for GUI testing, is the page-object pattern (G9.2). The pattern ensures that functionality is tested in isolation, considering one page view at the time. This practice helps to reduce the coupling between test cases and the SUT [136]. But the identified design patterns are not applicable to GUI-based tests that are on a higher level of abstraction, like image recognition-based GUI tests.

Lastly, from the SOLID principle, the single responsibility principle overlaps to some degree with the guideline to avoid unrelated changes G9.4. Hence, from a review standpoint, the reviewer shall ensure that each test is focused on only one test aspect. However, it is not clear how the remaining SOLID principles can be mapped to GUI-based test cases.

We rank this guideline as strongly recommend and see a high value in following established design principles and patterns.

**Answer to RQ2:** We present a mapping of source and test artifact review guidelines to GUI-based test artifacts by providing examples on its application. A summary of this mapping is shown in Table 3.5. In addition, we explained the purpose of each guideline category and ranked each guideline as suggested, recommended, or strongly recommended based on reference materials.
## Code Review Guidelines for GUI-based Testing Artifacts

<table>
<thead>
<tr>
<th>ID</th>
<th>Guideline</th>
<th>#WL</th>
<th>#GL</th>
<th>TS</th>
<th>WL Sources</th>
<th>GL Sources</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1.1</td>
<td>Perform automated checks on the change</td>
<td>21</td>
<td>21</td>
<td>no</td>
<td>S1, S2, S3, S9, S15, S18, S19, S20, S25, S26, S27, S28, S30, S36, S39, S42, S46, S60, S65, S75, S79</td>
<td>S1, S5, S4, S6, S7, S9, S15, S18, S20, S25, S26, S27, S28, S30, S36, S39, S42, S46, S60, S65, S75, S79</td>
</tr>
<tr>
<td>G1.2</td>
<td>Perform automated checks for code style</td>
<td>6</td>
<td>11</td>
<td>no</td>
<td>S36, S47, S56, S73, S76, S77</td>
<td>S36, S47, S56, S73, S76, S77</td>
</tr>
<tr>
<td>G2.1</td>
<td>Provide/use a checklist</td>
<td>8</td>
<td>19</td>
<td>no</td>
<td>S2, S6, S16, S19, S34, S47, S61, S76</td>
<td>S2, S6, S16, S19, S34, S47, S61, S76</td>
</tr>
<tr>
<td>G3.1</td>
<td>Provide additional context information</td>
<td>17</td>
<td>19</td>
<td>no</td>
<td>S12, S14, S17, S22, S35, S42, S46, S48, S50, S54, S62, S68, S71, S72, S73, S74, S77</td>
<td>S12, S14, S17, S22, S35, S42, S46, S48, S50, S54, S62, S68, S71, S72, S73, S74, S77</td>
</tr>
<tr>
<td>G3.2</td>
<td>Provide rationale for the change</td>
<td>10</td>
<td>14</td>
<td>no</td>
<td>S3, S19, S21, S22, S35, S36, S75, S77, S78, S79</td>
<td>S3, S19, S21, S22, S35, S36, S75, S77, S78, S79</td>
</tr>
<tr>
<td>G3.3</td>
<td>Provide context information about the impact of the change</td>
<td>1</td>
<td>1</td>
<td>no</td>
<td>S22</td>
<td>S22</td>
</tr>
<tr>
<td>G3.4</td>
<td>Provide context information about the history of changes</td>
<td>6</td>
<td>13</td>
<td>no</td>
<td>S7, S10, S22, S29, S69, S79</td>
<td>S7, S10, S22, S29, S69, S79</td>
</tr>
<tr>
<td>G3.5</td>
<td>Provide links to related resources and documentation</td>
<td>1</td>
<td>0</td>
<td>yes</td>
<td>S77</td>
<td>S77</td>
</tr>
<tr>
<td>G3.6</td>
<td>Provide context information about the dependencies between test and production code</td>
<td>4</td>
<td>0</td>
<td>no</td>
<td>S18, S35, S62, S75</td>
<td>S18, S35, S62, S75</td>
</tr>
<tr>
<td>G3.7</td>
<td>Provide context information about the test edge cases</td>
<td>1</td>
<td>2</td>
<td>yes</td>
<td>S77</td>
<td>S77</td>
</tr>
<tr>
<td>G4.1</td>
<td>Measure and monitor code metrics</td>
<td>7</td>
<td>7</td>
<td>no</td>
<td>S35, S36, S37, S40, S45, S51</td>
<td>S35, S36, S37, S40, S45, S51</td>
</tr>
<tr>
<td>G4.2</td>
<td>Provide metrics about execution time (for efficiency)</td>
<td>1</td>
<td>2</td>
<td>no</td>
<td>S47</td>
<td>S47</td>
</tr>
<tr>
<td>G4.3</td>
<td>Provide test coverage metrics (for effectiveness)</td>
<td>3</td>
<td>2</td>
<td>yes</td>
<td>S55, S57, S77</td>
<td>S55, S57, S77</td>
</tr>
<tr>
<td>G5.1</td>
<td>Ensure readability of the change</td>
<td>5</td>
<td>9</td>
<td>no</td>
<td>S1, S39, S42, S47, S76</td>
<td>S1, S39, S42, S47, S76</td>
</tr>
<tr>
<td>G5.2</td>
<td>Ensure conformance with the requirements</td>
<td>2</td>
<td>7</td>
<td>no</td>
<td>S47, S78</td>
<td>S47, S78</td>
</tr>
<tr>
<td>G5.3</td>
<td>Follow coding style and naming conventions</td>
<td>2</td>
<td>23</td>
<td>no</td>
<td>S38, S77</td>
<td>S38, S77</td>
</tr>
<tr>
<td>G5.4</td>
<td>Follow coding style and naming practices in test writing</td>
<td>2</td>
<td>0</td>
<td>yes</td>
<td>S67, S77</td>
<td>S67, S77</td>
</tr>
<tr>
<td>G5.5</td>
<td>Avoid code comments if they are not clear and useful</td>
<td>0</td>
<td>5</td>
<td>no</td>
<td>S48, S71, S17, S19, S34, S35</td>
<td>S48, S71, S17, S19, S34, S35</td>
</tr>
<tr>
<td>G5.6</td>
<td>Ensure proper usage of techniques for testing and exception handling</td>
<td>2</td>
<td>5</td>
<td>yes</td>
<td>S47, S77</td>
<td>S47, S77</td>
</tr>
<tr>
<td>G5.7</td>
<td>Ensure correctness of assertions in test cases</td>
<td>1</td>
<td>2</td>
<td>yes</td>
<td>S77</td>
<td>S77</td>
</tr>
<tr>
<td>G6.1</td>
<td>Provide a visualization of the change</td>
<td>9</td>
<td>1</td>
<td>no</td>
<td>S11, S13, S31, S42, S49, S52</td>
<td>S11, S13, S31, S42, S49, S52</td>
</tr>
<tr>
<td>G6.2</td>
<td>Provide a visualization of the code regarding its impact on the code base</td>
<td>3</td>
<td>0</td>
<td>no</td>
<td>S5, S13, S40</td>
<td>S5, S13, S40</td>
</tr>
<tr>
<td>G6.3</td>
<td>Allow traceability and easy navigation between artifacts</td>
<td>2</td>
<td>0</td>
<td>no</td>
<td>S42, S80</td>
<td>S42, S80</td>
</tr>
<tr>
<td>G7.1</td>
<td>Keep size of a change as low as possible</td>
<td>15</td>
<td>15</td>
<td>no</td>
<td>S14, S18, S19, S22, S33, S36, S39, S42, S43, S62, S64, S68, S70, S75, S78</td>
<td>S14, S18, S19, S22, S33, S36, S39, S42, S43, S62, S64, S68, S70, S75, S78</td>
</tr>
<tr>
<td>G7.2</td>
<td>Keep complexity of a change as low as possible</td>
<td>13</td>
<td>9</td>
<td>no</td>
<td>S4, S8, S18, S22, S39, S41, S47, S62, S68, S77, S78, S77, S78, S80</td>
<td>S4, S8, S18, S22, S39, S41, S47, S62, S68, S77, S78, S77, S78, S80</td>
</tr>
<tr>
<td>G7.3</td>
<td>Avoid unrelated and unstructured changes</td>
<td>6</td>
<td>8</td>
<td>no</td>
<td>S14, S19, S23, S24, S47, S58</td>
<td>S14, S19, S23, S24, S47, S58</td>
</tr>
<tr>
<td>G8.1</td>
<td>Ensure conformance with the requirements</td>
<td>0</td>
<td>3</td>
<td>no</td>
<td>S17, S35, S38</td>
<td>S17, S35, S38</td>
</tr>
<tr>
<td>G9.1</td>
<td>Apply established design principles and patterns</td>
<td>0</td>
<td>9</td>
<td>no</td>
<td>S21, S29, S31, S34, S35, S36, S42</td>
<td>S21, S29, S31, S34, S35, S36, S42</td>
</tr>
<tr>
<td>G9.2</td>
<td>Don’t repeat yourself (DRY)</td>
<td>0</td>
<td>4</td>
<td>no</td>
<td>S21, S31, S42, S43</td>
<td>S21, S31, S42, S43</td>
</tr>
<tr>
<td>G9.3</td>
<td>Avoid hardcoded values</td>
<td>0</td>
<td>2</td>
<td>no</td>
<td>S31, S34</td>
<td>S31, S34</td>
</tr>
<tr>
<td>G9.4</td>
<td>SOLID principle</td>
<td>0</td>
<td>3</td>
<td>no</td>
<td>S21, S31, S42</td>
<td>S21, S31, S42</td>
</tr>
</tbody>
</table>
### 3.4 Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Guideline Category</th>
<th>WL</th>
<th>GL</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>Perform automated checks</td>
<td>27</td>
<td>32</td>
<td>Perform automated checks to reduce the effort of the reviewer and avoid discussion about low-level code issues</td>
</tr>
<tr>
<td>G2</td>
<td>Use checklists</td>
<td>8</td>
<td>19</td>
<td>Create and use a checklist to prepare code changes for review and guide the reviewer during the code review</td>
</tr>
<tr>
<td>G3</td>
<td>Provide context information</td>
<td>44</td>
<td>54</td>
<td>Providing contextual information to aid the reviewer in understanding the code changes, e.g., the rationale of the proposed changes</td>
</tr>
<tr>
<td>G4</td>
<td>Utilize metrics</td>
<td>11</td>
<td>11</td>
<td>Utilize metrics on the code changes to monitor the effects of changes, predict the impact of code changes on the code base, and support decision-making in the review process</td>
</tr>
<tr>
<td>G5</td>
<td>Ensure readability</td>
<td>14</td>
<td>51</td>
<td>Ensure the readability and understandability of code changes</td>
</tr>
<tr>
<td>G6</td>
<td>Visualize changes</td>
<td>14</td>
<td>3</td>
<td>Support the reviewer with a visual representation of code changes to understand the impact of code changes and allow easier navigation between related artifacts</td>
</tr>
<tr>
<td>G7</td>
<td>Reduce complexity</td>
<td>34</td>
<td>32</td>
<td>Avoid and reduce the complexity of code changes due to their negative impact on reviewability</td>
</tr>
<tr>
<td>G8</td>
<td>Check conformity with the requirements</td>
<td>0</td>
<td>3</td>
<td>Ensure code changes are aligned with requirements and test specifications so that the changes are not implementing or testing the “wrong thing”</td>
</tr>
<tr>
<td>G9</td>
<td>Follow design principles and patterns</td>
<td>0</td>
<td>18</td>
<td>Use established software engineering design principles and design patterns</td>
</tr>
</tbody>
</table>

Table 3.4: Categories of code review guidelines used for source and test artifact review (WL=white literature, GL=gray literature)
3.5 Discussion

In this study, we have identified nine categories of code review guidelines for source and test artifacts that can be mapped as applicable to GUI-based test artifacts. We restrict the guidelines to artifacts to make them as tangible as possible. Thus, omitting less tangible guidelines regarding processes, principles and human factors associated with the code review process. The purpose of the proposed guidelines is to aid practitioners in improving the effectiveness and efficiency of code reviews. These improvements are perceived based on the proven value of guidelines for artifact review in other areas of software engineering. The motivation for this work stems from an empirically identified need for general code review guidelines for GUI-based tests in the industry as well as an identified gap for guidelines within academic literature.

As such, this work provides a tangible industrial contribution in an initial set of general code review guidelines for GUI-based testing. We perceive that our results can be used as a starting point for companies that seek to start using GUI-based testing practices, or companies that seek to improve upon their current practices. In companies that already conduct code reviews, we expect that there may be an overlap between existing guidelines and the proposed guidelines. However, from our empirical analysis, we have identified that existing guidelines are developed ad hoc. The results of this work provide a nomenclature to and provide insights into the purpose of each guideline, which may allow practitioners to more easily discuss and thereby motivate, or understand, the practices they use today.

Thus, although the list of nine categories including 33 guidelines we present is a good starting point, we do not perceive this list to be comprehensive. This assumption is motivated by the study’s focus on artifacts, not covering additional practices of the review process associated with, for instance, reviewer selection or effort allocation. The assumption is also motivated by the analysis of gray literature, where we found multiple practices that have not been covered in white literature. Thus, implying that there may be practices used in industry that academia is not aware of. In addition, we have only mentioned guidelines that could be data triangulated—Data triangulation is the practice of using different sources of information to increase the validity of a study’s results [145]—with at least three sources. Hence, additional guidelines were identified, but due to a lack of support for their validity, they were not incorporated into our results. An example of such a guideline was to 'provide information to reproduce identified faults' [36]. This guideline could possibly belong to a category of guidelines regarding how to provide reviewer feedback. However, due
to the lack of additional such guidelines in our sample, no such category was added.

The literature review also provides an academic contribution in showing gaps in knowledge about GUI-based testing, explicitly about review practices. However, looking at the body of knowledge on GUI-based testing, we note that the majority of work is focused on technical aspects of the approach. This observation also explains why we choose artifacts as the center point of our literature review since artifacts are more closely connected to the technical aspects of GUI-based testing. Regardless, there is a general need for more practice and human-focused research in the area of GUI-based testing. This work highlights one such area, where additional research is also required to extend the set of guidelines to be more comprehensive for the entire reviewing process.

Furthermore, although the guidelines presented in this work are perceived as valuable to GUI testing, due to their mapping to guidelines valuable for other software engineering artifacts, the guidelines have not yet been empirically tested. Initial screening has been made by presenting the guidelines to practitioners, but any feedback given from the practitioners is purely based on the perception of value. As such, future research is required to evaluate the actual value of the suggested guidelines.

Empirical evaluation of the guidelines is important since, in a related study, guidelines for the development of GUI-based tests were suggested but when evaluated in practice did not provide a successful result [10]. Although the study is limited in scope, it suggests that best development practices for source code are not necessarily transferable to GUI test code. The reasons stated in the related work were increased cognitive load as well as lack of applicability of some practices. Due to the level of abstraction of our suggested guidelines, focusing on what to look at in GUI-based test reviews and not how to do so, we do not perceive the same concerns. However, there is still a possibility that this unknown factor that set GUI-based tests apart from source code could play a role in the applicability of the proposed guidelines. Thus, once more, stressing the need for future empirical evaluation of the guidelines. Despite this potential issue, we do not perceive that it takes away anything from the contribution of this work. The reason is, as stated, the current omission of any general guidelines for reviews of GUI-based tests in the academic body of knowledge.

The guidelines presented in this work are for both the contributor and reviewer, but more focused towards the reviewer, and highlight practices that they shall perform. However, several of the practices can also be viewed from the contributor’s perspective, meaning that they give inputs on how to create better GUI test artifacts. This can be viewed as a natural progression of adopting
guidelines of this type. Meaning that feedback from reviewers naturally affects how contributors work and what they provide in their review requests. For example, if a reviewer provides feedback that contextual information is missing for a review. The contributor would append additional information and, likely, in the future, remember what information to add. Hence, the guidelines have a broader positive effect than for just the review process itself.

However, the caveat of making these guidelines valuable is not to overdo them. Seven of the nine guideline categories can be argued to be focused around supplying the reviewer with additional information for the review process. Whilst additional information is central to forming a complete picture of a situation, too much information can have adverse effects by increasing the reviewer’s cognitive load. From the review, we identified that adding context information is necessary to reduce the effort for the reviewer that would otherwise have to gather this information themselves. A similar circumstance can happen if the reviewer is provided with too much additional information that they need to siphon through. As such, a lean mindset must be applied to the type and amount of additional information that is provided in a review request. In particular, ensuring that all additional information provides value to the reviewer, omitting information that can be considered “nice to have”. Such information, although possibly useful, is likely to add overhead since it requires the reviewer to go through it and find what is important and not. In the worst case, such information is a pure waste, since it does not serve a purpose for a given review. Exactly how to determine what information to supply is however context-dependent, where in some cases the additional information can be very useful but, in other circumstances, it is not. Since different reviews vary in terms of size and focus, it is perceived that no general rule can be identified regarding what information is the best for each context.

3.5.1 Threats to Validity

The threats to the validity of this work have been divided into four parts; internal validity, external validity, construct validity and reliability, following the guidelines of Runeson and Höst [78].

**Internal validity**: Internal validity concerns the ability of the study design to conclude a correct relationship between factors, where multiple factors may have confounding, but unknown, effects on the investigated factor. For this study, one internal validity threat is that it is possible that literature sources with additional guidelines may have been overlooked. Although this could affect the guideline categories presented in this work, it is unlikely due to the high
level of abstraction they are presented at. Another threat is that the mapping between the source- and test code guidelines are invalid because we have failed to take factors of the original guideline into account. Once more this is unlikely, since the mapping was done through examples that show the guidelines use case. This mapping approach has been used previously for GUI-based testing research, for instance, by Alegroth and Gonzalez-Huerta [8].

A final threat about literature inclusion is about the possible utilization of code inspection as a synonym for code review in literature items. We tailored our inclusion criteria to include only literature items explicitly mentioning code review; as well, we did not provide any explicit exclusion criterion for the presence of code inspection as a keyword in the elicited literature. This set of criteria creates two threats if authors of searched literature used one of the term in place of the other: there is therefore a possibility that literature about code inspection is included in our results, and that literature about (modern) code review is excluded. Even though this threat can have an impact on the more quantitative results (i.e., number of sources and mentions per guideline or category) we do not expect impacts on the qualitative results (i.e., individual guidelines and categories description) given that modern code review are an evolution of code inspection and we expect guidelines in both fields to be similar.

The objective of RQ2 was to provide a mapping from unspecialized code review guidelines to guidelines for review of GUI-based test artifacts. In addition of the possibility of overlooking sources discussed previously, a threat to the validity of the result is the possible existence of facets of GUI-based test artifacts that should be code-reviewed, but that have no correspondence with normal production code. GUI-based testing artifacts include in fact several elements that strongly differ from regular code artifacts, e.g. models or screen captures of the visual content of the SUT. Guidelines to address these specific objects cannot be deduced by mapping code review guidelines for traditional code review, thereby limiting the comprehensiveness of the answer provided to RQ2 of the present manuscript. A final construct validity threat is related to the selection of a broad search string, including all code and software-related guidelines. The selection of such string can have an impact on the final count of the guideline occurrence that was measured as an answer to RQ2, providing measures differing significantly than the figures that would have been obtained including only test-related sources in the final pool.

**External validity:** External validity concerns the generalizability of the results to other areas or domains. The study’s scope is on GUI-based testing in general, regardless of test driver, GUI element localization technique or test case representation. Whilst there is a threat that some of the guidelines are less
applicable in some of these permutations of approaches, due to the high level of the guidelines, such threats are perceived as low.

**Construct validity:** Construct validity concerns if the studied phenomenon is the right phenomenon to meet the research objective. In this study, the objective was to map guidelines from source- and test code to guidelines for GUI-based tests. The assumption behind this objective is that the guidelines are transferable due to the common attributes between source and test code [8]. However, as discussed by Alegroth et al. [10], not all practices seem to be transferable. Hence, there is a threat to the study’s results that the characteristics used for the mapping are not representative, which may lower the applicability of some of the guidelines. Due to the example-based approach to the mapping, we consider this threat to be lower, but we can not conclude that all guidelines are applicable without empirical validation. Regardless, this threat does not affect the contribution of this work since there is currently a complete lack of guidelines for GUI-based testing reviews in the academic body of knowledge.

**Reliability:** Reliability concerns how reliant the study, and its results, are on the researchers. For the data collection, we have mitigated this threat by outlining the research procedure in detail. In addition, a replication package has been provided that presents all the acquired papers and the intermediate steps of the analysis. A larger threat lies in the synthesis of the results, common to coding-based research, where researcher and selection biases may have been introduced. These biases have been mitigated through the use of continued discussions among the authors of the paper and cross-validation of the results. However, due to the size of the data set, this threat can not be completely discarded. A common threat associated to Systematic Literature Reviews is researcher’s fatigue, i.e. the possibility of introducing biases in the analysis of large data tests, where the application adopted methodology becomes less rigorous towards the end of the analysis. This threat is amplified in the context of the present work by the adoption of a rather broad search string, as motivated in Section 3.1. The authors mitigated this threat by adopting a careful division of the reviewing tasks and by performing analysis sessions in fixed time windows.

### 3.6 Conclusion

Code reviews are a common practice in modern software development, used to identify faults and find improvements but also to share domain and technical knowledge within a software development team. These reviews are common for
source code and lower-level testing, but for GUI-based testing artifacts, there are no general guidelines.

In this work, we have performed a systematic literature review of guidelines for source and test code and mapped these to GUI-based testing artifacts. The review is classified as multi-vocal because in addition to incorporating white literature we also used gray literature.

From the synthesis of the results, nine categories of code review guidelines were identified, which were perform automated checks, use checklists, provide context information, utilize metrics, ensure readability, visualize changes, reduce complexity, requirements and follow design principles and patterns. The resulting mapping provides a contribution in terms of guidelines of general value to reviewing software development artifacts, but explicitly for the area of GUI-based testing. In addition to presenting the guidelines themselves, each guideline is demonstrated, through examples, how it can be applied for review of GUI-based testing. Notably, we see that the proposed guidelines can also have a positive impact on the development of GUI-based tests. It is, however, important to mention that the provided set of guidelines cannot be considered complete nor comprehensive for the practice of reviewing GUI-based test artifacts, since many specific aspects of GUI-based testing exist that can hardly be considered when providing general-purpose software review guidelines. Since our mapping was performed with general-purpose software review guidelines as a starting point, it is by construction possible that some of these specific aspects are missed. The results of these reviews should, therefore, be complemented in future work by the definition of code review guidelines that are exclusive to GUI-based test artifacts. Given the lack of such guidelines in the literature, we foresee the utilization of surveys and unstructured interviews with professionals as the means for collecting such evidence.

Future work based on this pivotal research includes empirical validation of the guidelines in practice, as well as extending the guidelines with non-artifact focused guidelines, e.g. guidelines for reviewer selection of GUI-based tests and review-process related aspects.

In conclusion, this work provides an important stepping stone for review guidelines for GUI-based testing. However, more work is required in the future to address the current needs from the industry, and challenges in this area.
Table 3.6: White literature sources where guidelines were taken

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Authors</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>A Community of Practice Around Peer Review for Long-Term Research Software Sustainability</td>
<td>Kalyan, Akshay, et al.</td>
<td>2019</td>
</tr>
<tr>
<td>S2</td>
<td>A reflective practice of automated and manual code reviews for a studio project</td>
<td>Oh, Jun-Suk, and Ho-Jin Choi</td>
<td>2005</td>
</tr>
<tr>
<td>S3</td>
<td>A Secure Code Review Retrospective</td>
<td>Buttner, Andrew, et al.</td>
<td>2020</td>
</tr>
<tr>
<td>S5</td>
<td>Aiding Code Change Understanding with Semantic Change Impact Analysis</td>
<td>Hanam, Quinn, et al.</td>
<td>2019</td>
</tr>
<tr>
<td>S7</td>
<td>An Empirical Study of Link Sharing in Review Comments</td>
<td>Jiang, Jing, Jin Cao, and Li Zhang</td>
<td>2019</td>
</tr>
<tr>
<td>S8</td>
<td>Analyzing Involvements of Reviewers through Mining a Code Review Repository</td>
<td>Liang, Junwei, and Osamu Mizuno</td>
<td>2011</td>
</tr>
<tr>
<td>S10</td>
<td>Automatic patch linkage detection in code review using textual content and file location features</td>
<td>Wang, Dong, et al.</td>
<td>2021</td>
</tr>
<tr>
<td>S12</td>
<td>Can peer code reviews be exploited for later information needs?</td>
<td>Sutherland, Andrew, and Gina Venolia</td>
<td>2009</td>
</tr>
<tr>
<td>S13</td>
<td>ChangeViz: Enhancing the GitHub Pull Request Interface with Method Call Information</td>
<td>Gasparini, Lorenzo, et al.</td>
<td>2021</td>
</tr>
<tr>
<td>S15</td>
<td>Code review analysis of software system using machine learning techniques</td>
<td>Lal, Harsh, and Gaurav Pahwa</td>
<td>2017</td>
</tr>
<tr>
<td>S16</td>
<td>Code review and cooperative pair programming best practice</td>
<td>Fu, Qiang, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S19</td>
<td>Code Reviewing in the Trenches: Challenges and Best Practices</td>
<td>MacLeod, Laura, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S20</td>
<td>Code Reviews With Divergent Review Scores: An Empirical Study of the OpenStack and Qt Communities</td>
<td>Hirao, Toshiki, et al.</td>
<td>2022</td>
</tr>
<tr>
<td>S21</td>
<td>Communicative Intention in Code Review Questions</td>
<td>Ebert, Felipe, et al.</td>
<td>2018</td>
</tr>
</tbody>
</table>
Table 3.6 – continued from previous page

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Authors</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>S23</td>
<td>CoRA: Decomposing and Describing Tangled Code Changes for Reviewer</td>
<td>Wang, Min, et al.</td>
<td>2019</td>
</tr>
<tr>
<td>S26</td>
<td>Evaluating how static analysis tools can reduce code review effort</td>
<td>Singh, Devarshi, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S27</td>
<td>Expectations, outcomes, and challenges of modern code review</td>
<td>Bacchelli, Alberto, and Christian Bird</td>
<td>2013</td>
</tr>
<tr>
<td>S28</td>
<td>Fix-it: An extensible code auto-fix component in Review Bot</td>
<td>Balachandran, Vipin</td>
<td>2013</td>
</tr>
<tr>
<td>S30</td>
<td>Impact of Coding Style Checker on Code Review - A Case Study on the OpenStack Projects</td>
<td>Ueda, Yuki, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S31</td>
<td>Interactive Code Review for Systematic Changes</td>
<td>Zhang, Tianyi, et al.</td>
<td>2015</td>
</tr>
<tr>
<td>S33</td>
<td>Investigating code review quality: Do people and participation matter?</td>
<td>Kononenko, Oleksii, et al.</td>
<td>2015</td>
</tr>
<tr>
<td>S34</td>
<td>Java code reviewer for verifying object-oriented design in class diagrams</td>
<td>Jinto, Kanit, and Yachai Limpiyakorn</td>
<td>2010</td>
</tr>
<tr>
<td>S35</td>
<td>Lessons Learned from Building and Deploying a Code Review Analytics Platform</td>
<td>Bird, Christian, et al.</td>
<td>2015</td>
</tr>
<tr>
<td>S36</td>
<td>LightSys: Lightweight and Efficient CI System for Improving Integration Speed of Software</td>
<td>Lim, Geunsik, et al.</td>
<td>2021</td>
</tr>
<tr>
<td>S38</td>
<td>Mining Source Code Improvement Patterns from Similar Code Review Works</td>
<td>Ueda, Yuki, et al.</td>
<td>2019</td>
</tr>
<tr>
<td>S39</td>
<td>Modern Code Review: A Case Study at Google</td>
<td>Sadowski, Caitlin, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S40</td>
<td>Multi-Perspective Visualization to Assist Code Change Review</td>
<td>Wang, Chen, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S41</td>
<td>Natural Language Insights from Code Reviews that Missed a Vulnerability</td>
<td>Munaia, Nuthan, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S43</td>
<td>On the understanding of programs with continuous code reviews</td>
<td>Bernhart, Mario, and Thomas Grechenig</td>
<td>2013</td>
</tr>
<tr>
<td>ID</td>
<td>Title</td>
<td>Authors</td>
<td>Year</td>
</tr>
<tr>
<td>----</td>
<td>-------</td>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>S45</td>
<td>Guiding Developers to Make Informative Commenting Decisions in Source Code</td>
<td>Huang, Yuan, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S46</td>
<td>Practical aspects on the assessment of a review process</td>
<td>Kiiskila, Janne</td>
<td>1998</td>
</tr>
<tr>
<td>S48</td>
<td>RAID: Tool Support for Refactoring-Aware Code Reviews</td>
<td>Brito, Rodrigo, and Marco Tulio Valente</td>
<td>2021</td>
</tr>
<tr>
<td>S49</td>
<td>Refactoring in Code Review Considered Harmful: A Large-Scale Empirical Investigation</td>
<td>Paixao, Matheus, and Paulo Henrique Maia</td>
<td>2019</td>
</tr>
<tr>
<td>S51</td>
<td>Review Dynamics and Their Impact on Software Quality</td>
<td>Thongtanunam, Patanamon, and Ahmed E. Hassan</td>
<td>2021</td>
</tr>
<tr>
<td>S52</td>
<td>RSTrace+: Reviewer suggestion using software artifact traceability graphs</td>
<td>Sülün, Emre, et al.</td>
<td>2020</td>
</tr>
<tr>
<td>S53</td>
<td>Salient-class location: help developers understand code change in code review</td>
<td>Huang, Yuan, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S54</td>
<td>SCRUB: a tool for code reviews</td>
<td>Holzmann, Gerald</td>
<td>2010</td>
</tr>
<tr>
<td>S55</td>
<td>Semantics-assisted code review: An efficient tool chain and a user study</td>
<td>Menarini, Massimiliano, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S57</td>
<td>Static Security Analysis Based on Input-Related Software Faults</td>
<td>Nagy, Csaba, and Spiros Manourtzis</td>
<td>2009</td>
</tr>
<tr>
<td>S62</td>
<td>The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects</td>
<td>McIntosh, Shane, et al.</td>
<td>2014</td>
</tr>
<tr>
<td>S64</td>
<td>The influence of non-technical factors on code review</td>
<td>Baysal, Olga, et al.</td>
<td>2013</td>
</tr>
<tr>
<td>S65</td>
<td>The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more</td>
<td>Hentschel, Martin, et al.</td>
<td>2019</td>
</tr>
</tbody>
</table>
### Table 3.6 – continued from previous page

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Authors</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>S68</td>
<td>Towards a taxonomy of code review smells</td>
<td>Dogan, Emre, and Eray Tüzün</td>
<td>2021</td>
</tr>
<tr>
<td>S69</td>
<td>Understanding shared links and their intentions to meet information needs in modern code review</td>
<td>Wang, Dong, et al.</td>
<td>2021</td>
</tr>
<tr>
<td>S71</td>
<td>Using Paragraph Vectors to improve our existing code review assisting tool-CRUSO</td>
<td>Kapur, Ritu, et al.</td>
<td>2021</td>
</tr>
<tr>
<td>S73</td>
<td>What Are They Talking About? Analyzing Code Reviews in Pull-Based Development Model</td>
<td>Li, Zhi-Xing, et al.</td>
<td>2017</td>
</tr>
<tr>
<td>S74</td>
<td>What Design Topics do Developers Discuss?</td>
<td>Viviani, Giovanni, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S75</td>
<td>What makes a code change easier to review: an empirical investigation on code change reviewability</td>
<td>Ram, Achyudh, et al.</td>
<td>2018</td>
</tr>
<tr>
<td>S78</td>
<td>Why Did This Reviewed Code Crash? An Empirical Study of Mozilla Firefox</td>
<td>An, Le, et al.</td>
<td>2018</td>
</tr>
</tbody>
</table>
### Table 3.7: Gray literature sources where guidelines were taken

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>URL</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS2</td>
<td>Best Practices for Peer Code Review - SmartBear</td>
<td>smartbear.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS3</td>
<td>How to do a code review</td>
<td>eng-practices - Google</td>
<td>google.github.io</td>
</tr>
<tr>
<td>GS4</td>
<td>5 code review best practices - Work Life by Atlassian</td>
<td>atlassian.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS5</td>
<td>Code Review Best Practices - Palantir Blog</td>
<td>blog.palantir.com</td>
<td>2018</td>
</tr>
<tr>
<td>GS7</td>
<td>How to Make Good Code Reviews Better - Stack Overflow Blog</td>
<td>stackoverflow.blog</td>
<td>2019</td>
</tr>
<tr>
<td>GS8</td>
<td>16 Tech Leaders Share Smart Best Practices For Reviewing Code</td>
<td>forbes.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS9</td>
<td>Reviewing Code - Best practices and techniques for code review</td>
<td>codegrip.tech</td>
<td>2021</td>
</tr>
<tr>
<td>GS10</td>
<td>How To Review Someone Else’s Code: Tips and Best Practices</td>
<td>codecademy.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS11</td>
<td>Code Review good practices: guide for beginners</td>
<td>medium.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS12</td>
<td>5 code review best practices. Make others like your code review</td>
<td>tsh.io</td>
<td>2020</td>
</tr>
<tr>
<td>GS13</td>
<td>Code Review Guidelines - GitLab Documentation</td>
<td>docs.gitlab.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS14</td>
<td>Proven Code Review Best Practices from Microsoft</td>
<td>michaelagreiler.com</td>
<td>2019</td>
</tr>
<tr>
<td>GS15</td>
<td>Code review best practices - DeepSource</td>
<td>deepsource.io</td>
<td>2019</td>
</tr>
<tr>
<td>GS16</td>
<td>13 Code Review Standards Inspired by Google</td>
<td>betterprogramming.pub</td>
<td>2020</td>
</tr>
<tr>
<td>GS17</td>
<td>What is Code Review? - Guidelines and Best Practices</td>
<td>blog.ndepend.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS19</td>
<td>Code Review - Open Practice Library</td>
<td>openpracticelibrary.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS20</td>
<td>6 code review best practices for a happier codebase and team</td>
<td>educative.io</td>
<td>2022</td>
</tr>
<tr>
<td>GS21</td>
<td>Code Review Best Practices - Trisha Gee</td>
<td>trishagee.com</td>
<td>2018</td>
</tr>
<tr>
<td>GS23</td>
<td>5 Best Practices For Code Review - GeeksforGeeks</td>
<td>geeksforgeeks.org</td>
<td>2022</td>
</tr>
<tr>
<td>GS24</td>
<td>How to improve your code review: tips and best practices</td>
<td>belvo.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS25</td>
<td>Investigating the effectiveness of peer code review in ...</td>
<td>jserd.springeropen.com</td>
<td>2018</td>
</tr>
<tr>
<td>GS26</td>
<td>Better code, better applications - every time</td>
<td>walkingtree.tech</td>
<td>2022</td>
</tr>
<tr>
<td>GS27</td>
<td>How we do it: peer code review - DataMiner Dojo</td>
<td>community.dataminer.services</td>
<td>2022</td>
</tr>
<tr>
<td>GS28</td>
<td>Code Review Guidelines for Data Science Teams</td>
<td>tdhopper.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS29</td>
<td>Gerrit Code Review Product Overview</td>
<td>gerrit-review.googlesource.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS31</td>
<td>Code review checklist - Apex Hours</td>
<td>apexhours.com</td>
<td>2021</td>
</tr>
</tbody>
</table>

Continued on next page
Table 3.7 – continued from previous page

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>URL</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>GS32</td>
<td>The Art of Code Review - Towards Data Science</td>
<td>towardsdatascience.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS34</td>
<td>Best practices code review test automation by Anton Smirnov</td>
<td>itnext.io</td>
<td>2021</td>
</tr>
<tr>
<td>GS36</td>
<td>Code Review Best Practices - GitKraken</td>
<td>gitkraken.com</td>
<td>2021</td>
</tr>
<tr>
<td>GS37</td>
<td>Best Practices for Effective Code Review</td>
<td>leobit.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS38</td>
<td>Best Practices for Effective and Efficient Agile Code Reviews</td>
<td>queue-it.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS40</td>
<td>Creating Simple and Effective Guidelines for Code Reviews</td>
<td>newrelic.com</td>
<td>2018</td>
</tr>
<tr>
<td>GS42</td>
<td>Code Review Best Practices - Programmer Friend</td>
<td>programmerfriend.com</td>
<td>2018</td>
</tr>
<tr>
<td>GS43</td>
<td>How rOpenSci uses code review to promote reproducible science</td>
<td>ropensci.org</td>
<td>2017</td>
</tr>
<tr>
<td>GS44</td>
<td>Where is the value in package peer review?</td>
<td>ropensci.org</td>
<td>2018</td>
</tr>
<tr>
<td>GS45</td>
<td>Recommended C Style and Coding Standards. Pocket reference guide</td>
<td>gnu.org</td>
<td>2005</td>
</tr>
<tr>
<td>GS46</td>
<td>ChangeViz materials</td>
<td>doi.org/10.5281/zenodo.5175927</td>
<td>2021</td>
</tr>
<tr>
<td>GS47</td>
<td>Ilya Sabanin Contributor to Beanstalk Guides</td>
<td>guides.beanstalkapp.com</td>
<td>2019</td>
</tr>
<tr>
<td>GS48</td>
<td>Atlassian Crucible features</td>
<td>atlassian.com</td>
<td>2022</td>
</tr>
<tr>
<td>GS49</td>
<td>CodeFlow</td>
<td>getcodeflow.com</td>
<td>2014</td>
</tr>
<tr>
<td>GS50</td>
<td>Intel Open Source Technology Center – Patch Review</td>
<td>blog.ffwll.ch</td>
<td>2020</td>
</tr>
<tr>
<td>GS51</td>
<td>Apache Spark</td>
<td>spark.apache.org</td>
<td>2020</td>
</tr>
<tr>
<td>GS52</td>
<td>Chromium coding style</td>
<td>dev.chromium.org</td>
<td>2014</td>
</tr>
<tr>
<td>GS53</td>
<td>Pep 8: style guide for python code</td>
<td>peps.python.org.pep-0008</td>
<td>2001</td>
</tr>
<tr>
<td>GS54</td>
<td>10 faulty behaviors of code review</td>
<td>speakerdeck.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS55</td>
<td>How we do code review — app center blog</td>
<td>devblogs.microsoft.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS56</td>
<td>Code review best practices by Palantir</td>
<td>medium.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS57</td>
<td>Pull request best practices - the pragmatic engineer</td>
<td>blog.pragmaticengineer.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS58</td>
<td>The wireshark wiki - Development/SubmittingPatches</td>
<td>wiki.wireshark.org</td>
<td>2020</td>
</tr>
<tr>
<td>GS59</td>
<td>Sharma S. How to write a good pull request description – and why it’s important URL</td>
<td>freedecodecamp.org</td>
<td>2020</td>
</tr>
<tr>
<td>GS60</td>
<td>What is code review?</td>
<td>smartbear.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS61</td>
<td>Code reviews at google are lightweight and fast</td>
<td>michaelagreiler.com</td>
<td>2020</td>
</tr>
<tr>
<td>GS63</td>
<td>Firefox code review</td>
<td>wiki.mozilla.org</td>
<td>2016</td>
</tr>
</tbody>
</table>
Chapter 4

Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

This chapter is based on the following paper:

Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

Abstract

Context: Manual graphical user interface (GUI) software testing presents a substantial part of the overall practiced testing efforts, despite various research efforts to further increase test automation. Augmented Testing (AT), a novel approach for GUI testing, aims to aid manual GUI-based testing through a tool-supported approach where an intermediary visual layer is rendered between the system under test (SUT) and the tester, superimposing relevant test information.

Objective: The primary objective of this study is to gather empirical evidence regarding AT’s efficiency compared to manual GUI-based regression testing. Existing studies involving testing approaches under the AT definition primarily focus on exploratory GUI testing, leaving a gap in the context of regression testing. As a secondary objective, we investigate AT’s benefits, drawbacks, and usability issues when deployed with the demonstrator tool, Scout.

Method: We conducted an experiment involving 13 industry professionals, from six companies, comparing AT to manual GUI-based regression testing. These results were complemented by interviews and Bayesian data analysis (BDA) of the study’s quantitative results.

Results: The results of the Bayesian data analysis revealed that the use of AT shortens test durations in 70% of the cases on average, concluding that AT is more efficient. When comparing the means of the total duration to perform all tests, AT reduced the test duration by 36% in total. Participant interviews highlighted nine benefits and eleven drawbacks of AT, while observations revealed four usability issues.

Conclusion: This study makes an empirical contribution to understanding Augmented Testing, a promising approach to improve the efficiency of GUI-based regression testing in practice. Furthermore, it underscores the importance of continual refinements of AT.

4.1 Introduction

Manual software testing is labor-intensive, error-prone, and demands a combination of technical proficiency and domain knowledge. It is, therefore, associated with high cost [51, 150]. To address these challenges, the adoption of automated testing has become an essential part of software development practices [92, 99]. By leveraging automated tests, developers can effectively verify software artifacts against their specified requirements. This approach allows for frequent and
cost-effective test executions, providing continuous feedback on the software system’s quality throughout the entire development lifecycle [151]. Furthermore, automated testing minimizes the occurrence of human errors, thereby enhancing the overall reliability of the testing process [152].

Despite various research efforts in the area of automated testing, manual testing presents a significant portion of the overall practiced testing in industry [44, 81]. According to a survey conducted in 2020 [147], the automation rate of test cases within an agile and DevOps development environment was reported to be only 6%. Interestingly, but not related, 6% of practitioners also expressed confidence in achieving complete automation in software testing in a survey from 2012 [72]. While the rate of automated test cases and the confidence in complete automation can not be compared directly, these numbers indicate the importance of manual testing in practice. Moreover, the findings between these two studies highlight a persistent challenge spanning nearly a decade.

Manual testing can be carried out at varied levels of abstraction, employing different test session strategies and test execution techniques [153]. One such technique is GUI-based testing, where a system’s behavior is verified through its GUI at a higher level of abstraction, mirroring user interactions [7, 30]. Interactions with the GUI allow not only the verification of the appearance of the GUI but, more importantly, the verification of the behavior of the system under test (SUT) [7].

Augmented testing (AT) is a novel testing approach that has shown promise to aid manual testers [65, 66, 67]. The core of this approach regards using tools to create a virtual intermediary layer between the tester and the SUT’s GUI to superimpose test information that can aid the tester in verifying the SUT’s behavior according to its specifications. This superimposed information includes visual highlights on widgets or bitmaps of previous, or suggested, user interactions, checks/assertions, identified issues, or comments, which all aim to improve the tests’ effectiveness. Figure 4.1 demonstrates the concept of AT on a web GUI.

In addition to improving tester effectiveness, AT shows promise in achieving improvements in the efficiency of GUI testing by reducing the time required to perform the tests. This is achieved since all user interactions with the SUT go through the augmented layer and are recorded in a state model, referred to as a test model. Recording tests through the augmented layer reduces the time for test development and maintenance and ensures that recorded tests can be replayed in the same way as they were recorded. The augmentations, e.g. test data, also reduce cognitive complexity and the need for context switching, for instance, for running regression tests with step-by-step instructions that are
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

commonly documented in separate documents. Furthermore, the recorded test data in the test model can be used to derive suggestions and information relevant to further testing.

A technical demonstrator of AT is provided in the tool Scout [67], which, in previous work, has been evaluated in two industrial workshop studies [65]. In the first study, the tool’s perceived benefits and drawbacks were evaluated, further supported by results from a second empirical study on the industrial applicability of AT [66]. More details regarding these two studies are provided in the related work section.

We define AT as a tool-supported testing approach that employs GUI augmentations, such as visual highlights on widgets, to assist testers in navigating the SUT’s GUI, asserting widgets, and providing test-relevant information. Hence, we classify other testing approaches as AT if they align with our definition.

![Figure 4.1: Demonstration of Augmented Testing (AT). The SUT’s widgets are superimposed with information through the augmentation layer.](image)

**Research gap.**

Previous studies highlight the potential benefits of Augmented Testing (AT) in software testing [65, 66]. However, an empirical evaluation of its usability remains unexplored but would allow us to further understand its practical application. In addition, a deeper analysis of its efficiency could demonstrate positive effects on the high costs associated with the current deployment of manual GUI-based test practices.

This study aims to address this gap by empirically evaluating the efficiency of AT compared to manual GUI-based regression testing. Through interviews and observations, we identify AT’s usability issues, benefits, and drawbacks. Given the prominence of manual GUI-based testing in current testing practices,
4.2 Related work

It’s crucial to explore methodologies that enhance efficiency and offset the substantial costs of manual testing.

Based on our results, we claim the following contributions:

- An empirical evaluation of AT compared to manual GUI-based regression testing in regards to its efficiency;

- A synthesis of reported benefits and drawbacks of AT compared to manual GUI-based regression testing, categorized as essential (related to the testing approach itself) or accidental (related to the tool implementing AT);

- A synthesis of observed usability issues of AT, categorized as essential or accidental, to serve as a recommendation of future research directions to improve AT tools.

**REPLICATION PACKAGE: https://doi.org/10.5281/zenodo.8328166 [146].** This replication package includes the description of test cases (as given to the participants), binaries of the testing tool, source code of the SUT, source code used for the statistical analysis, observation notes, and measurements.

The remainder of this paper is organized as follows: in Section 4.2 we summarize work related to Augmented Testing; in Section 4.3 we describe the method that we employed to conduct our experiment in detail; in Section 4.4 we report the results of the experiment; in Section 4.5 we discuss the results; in Section 4.6 we discuss the threats to validity of the study; in Section 4.7 we conclude the paper by summarizing the main findings and providing future research directions.

4.2 Related work

Liu et al. [59] introduced a novel tool-supported approach called NaviDroid, which assists testers in enhancing the efficiency and effectiveness of manual GUI-based testing for mobile applications. This approach involves highlighting the next operations of the test to guide testers through specific paths of the SUT’s GUI or explore new ones. The concept of NaviDroid fulfills our definition of AT, particularly in terms of providing visual indicators (augmentations) on the GUI to facilitate navigation through a specific path of the GUI or exploring various states of the GUI that are not covered. Utilizing augmentations allows testers
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

to avoid redundant repetitions of test cases. Navidroid, designed for exploratory testing, has shown improved effectiveness through increased test coverage and a higher amount of identified bugs. Furthermore, NaviDroid enhanced testing efficiency by reducing the time required to cover all pages of an app and helps testers avoid the creation of duplicated actions within a test. Testers’ feedback further confirmed the usefulness of NaviDroid in assisting manual GUI-based testing of Android applications and liked the interaction design with the visual indicators as guidance. The study’s results support the value of AT but are limited to exploratory testing for the mobile application platform. The authors expect similar results for other platforms but require additional studies to show its extensibility. We see the need for coverage of other platforms and a consideration of manual regression testing. In the study, drawbacks and usability issues also remained uncovered. Notably, the study highlights the usefulness of visual indicators but does not address potential issues arising from guiding testers with visual cues rather than textual descriptions. Hence, there remains a need for future studies to explore and address these concerns.

Chen et al. [28] proposed two techniques to improve crowd-testing efficiency: the interactive event-flow graph and GUI-level guidance. The interactive event-flow graph consolidates all navigation paths followed by testers in an application into a single model. GUI-level guidance utilizes overlays on the GUI to highlight previously visited paths based on the event-flow graph, which fulfills our definition of AT. GUI-level guidance is advantageous in crowd testing, where multiple testers engage in independent GUI-based testing activities. By preventing redundant test cases, GUI-level guidance proves especially beneficial as crowd workers often follow common paths while working in parallel [154, 155]. To measure test coverage, the authors adopted event-interaction coverage, which examines the permutation of input events, as a metric to evaluate the tests’ effectiveness [61]. Other top-level categories in this taxonomy are functional-level, GUI-level, and code-level. Notably, GUI-level guidance solely concerns navigation and does not provide widget assertion of the behavior of the SUT. Detecting defects is accomplished through interactions spanning various user intents, which serve as a behavior template. To evaluate the effectiveness of the two presented techniques, an experiment involving 30 crowd testers was conducted. The results showcased a significant increase in test coverage for both trained and untrained testers who utilized the presented techniques. Relying on unknown crowd workers rather than industrial practitioners constrains the insights into the approach’s industrial application. Industrial participants might offer insights into potential drawbacks that could render the approach infeasible in an industrial setting.
Nass et al. [65] introduced the concept of AT and conducted two workshops involving industrial participants to gather information about the perceived benefits and drawbacks of AT. These workshops, where a total of 10 software developers participated, followed a structured approach consisting of four distinct steps: 1) An introduction where the prototype tool for AT was demonstrated, 2) participants were given the freedom to evaluate the tool by themselves, 3) participants documented their observations of both the benefits and drawbacks they identified during their evaluation. 4) participants were asked to envision a state-of-the-art AT-based tool and identify achievable benefits of AT. In summary, the gathered feedback highlighted the prevalence of perceived benefits associated with AT, with many of the mentioned drawbacks being primarily attributed to limitations of the prototype tool rather than the AT testing approach itself. Examples of perceived benefits included the ability to understand what has been tested and what has not, as well as a reduction in manual workload. This study represents an initial step towards empirically evaluating AT, laying the foundation for further investigation. As part of future research, the authors mentioned the importance of conducting an industrial evaluation to assess the efficiency and effectiveness of AT in practical settings.

In a later study by Nass et al. [66], the authors investigated the industrial applicability of the tool Scout with AT. The creation of test cases with AT was evaluated in an industrial context via a quasi-experiment, where Scout was compared against two other state-of-practice tools regarding its efficiency. Here, six practitioners created tests with Scout and Protractor [114], and six other practitioners created tests with Scout and Selenium [156] for automated GUI-based testing. After the experiment, a questionnaire survey was performed to understand how much experience in test automation and programming is perceived to be required to create tests using AT successfully. Their study showed that Scout with AT requires less programming and automation expertise than the compared tools, thus making it more accessible to users. Further, creating effective test cases is more efficient than the tools Protractor and Selenium. While this study highlights the ease of use and the enhanced efficiency and effectiveness of test case creation, further evaluation of AT in diverse contexts is necessary to determine the generalizability of the results.

4.3 Method

The objective of this study is to evaluate the efficiency of AT compared to manual GUI-based regression testing and report its benefits, drawbacks, and
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

usability issues. This research stems from the need for testing methodologies that enhance efficiency and offset the substantial costs of manual testing. AT demonstrated promising results in improving the efficiency of GUI-based testing, but previous work did not cover manual testing and/or focused more on the perceived values of the approach. Thus leaving a gap in knowledge of the testing approach’s actual value, effectiveness, and efficiency in the context of manual regression testing with the potential to reduce costs when used by industrial practitioners. To achieve this objective, we have broken it down into a series of research questions.

- RQ1: To which extent can Augmented Testing improve the efficiency of manual GUI-based regression testing?
- RQ2: What are the benefits and drawbacks of Augmented Testing reported by practitioners?
- RQ3: What are the observed usability issues of Augmented Testing?

To answer RQ1, we measure the test duration—the time required to complete each test case—as the metric for efficiency. Through a statistical analysis of these measurements, we can then infer whether the usage of AT results in shorter test durations and, thus, is more efficient.

Important to note is that we aim to understand whether AT can aid and improve the efficiency of a manual GUI-based testing process. The objective is thereby not to automate the testing process but to assist the tester in their current process through the augmentation layer.

RQ2 aims to identify areas for improvement and generate ideas for further research. The insights gained from this investigation will play a vital role in refining AT and its demonstrator, aligning them more with the demands of an industrial context.

RQ3 aims to identify usability issues emerging from AT or its demonstrator. Usability issues that are independent of a particular implementation are categorized as AT issues, while issues that can be mitigated through improved tooling belong to the demonstrator. Given that AT is a novel approach to testing, it is unclear if it is intuitive to testers and if the augmented layer between the tester and the SUT causes usability concerns. For instance, can testers complete a task without asking questions to continue with the task? This is known as the learnability attribute of usability, implying that a system should be easily [157, 158].
4.3 Method

We conduct an experiment to answer the research questions, following the guidelines by Wohlin et al. [88]. In this experimental study, participants will perform multiple manual GUI-based regression test cases, both with and without the application of AT as the treatment. The experiments will be conducted individually, with participants participating in one-on-one sessions with the researchers. This approach allows for detailed observations and the collection of valuable insights.

Based on the goal template by Basili and Rombach [20], shown in Table 4.1, the goal of this study’s experiment is to analyze manual GUI-based regression testing and Augmented GUI-based regression testing with respect to its efficiency from the point of view of researchers and testers in the context of web applications.

<table>
<thead>
<tr>
<th>Object of study</th>
<th>GUI-based regression testing of web applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Compare AT with manual GUI-based testing</td>
</tr>
<tr>
<td>Focus</td>
<td>Efficiency</td>
</tr>
<tr>
<td>Context</td>
<td>Web-based application</td>
</tr>
<tr>
<td>Perspective</td>
<td>Tester, Researchers</td>
</tr>
</tbody>
</table>

Table 4.1: Application of GQM to this experiment

4.3.1 Context selection

Selecting the context of the experiment is a choice between different trade-offs, as described by Wohlin et al. [88].

- Student vs. professional: We choose professionals as subjects. As Höst et al. [54] showed, students could be used if the task size is relatively small. However, students would not allow us to gather additional data on the benefits and drawbacks of Augmented Testing from a practitioner’s perspective to identify obstacles preventing an industrial adoption. Using professionals will, however, introduce higher costs for the companies providing employees for this experiment, which reduces the number of subjects we can request.

- Toy vs. real problems: We use a web-based project designed for managing food recipes in schools as a SUT. Although this project does not belong to an industry setting, its complexity surpasses that of a toy-sized
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

project or educational demonstrator, and it is actively utilized by a school, thereby positioning it as a “real” project. However, the GUI tests that were performed for this specific SUT were created specifically for the experiment, as no pre-existing GUI tests existed within the project. Since the tests for the experiment were ad hoc, they could be considered more as a “toy” problem. While toy problems may have limitations in terms of generalizability to real-world issues, they offer advantages in enabling more controlled comparisons of results across participants, which was necessary in this case since participants came from various companies.

- Specific vs. general: A specific context is chosen by narrowing the scope to specific types of augmentations.

4.3.2 Hypothesis formulation

To support RQ1, we formulated the following hypotheses.

Null hypothesis ($H_0$): The use of Augmented Testing has no effect on the average test duration to complete each test case.

Alternative hypothesis ($H_1$): The use of Augmented Testing has an effect on the average test duration to complete each test case by either reducing or increasing it, considering the size of test cases and a potential learning effect.

4.3.3 Variable selection

In our experimental design, we specify three independent variables: treatment type (either AT or manual GUI-based regression testing), relative test case size, and the learning effect associated with the usage of the AT interface. Independent variables are those that we can control, anticipating their influence on dependent variables.

The dependent variable, which measures the treatment’s effect based on our hypothesis, pertains to the efficiency of manual GUI-based regression testing. Table 4.2 details the variable selection, including each variable’s data type and range.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Variable</th>
<th>Type</th>
<th>Data type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Treatment</td>
<td>Treatment of Augmented Testing vs. manual GUI-based regression testing (baseline)</td>
<td>AT</td>
<td>ind</td>
<td>boolean</td>
<td>{true, false}</td>
</tr>
<tr>
<td>Size</td>
<td>Relative size of a test case</td>
<td>size</td>
<td>ind</td>
<td>boolean</td>
<td>{big, small}</td>
</tr>
<tr>
<td>Learning</td>
<td>Learning effect of repeatedly using the AT interface. A linear factor increasing one unit with each use of AT</td>
<td>learn</td>
<td>ind</td>
<td>count</td>
<td>[0, 4]</td>
</tr>
<tr>
<td>Efficiency</td>
<td>Test efficiency; measured as the average time in seconds taken to complete each test case per treatment</td>
<td>duration_scaled</td>
<td>dep</td>
<td>float</td>
<td>$x &gt; 0$</td>
</tr>
</tbody>
</table>

Table 4.2: Description of independent (ind) and dependent (dep) variables
4.3.4 Selection of subjects

The primary selection criteria for participants (subjects) is that they have industrial experience as developers or testers with GUI-based testing. For all sampled participants, we collected detailed data on their professional experience after each experimental session. Specifically, their experience, measured in years, in the software development industry, industrial experience in testing (manual or automated), and experience with GUI-based testing. Specific competencies, like testing and GUI-based testing experience, that are relevant to the experiment are more informative than collecting only the overall industrial experience Ko et al. [57].

Subjects were sampled from different companies in our network, where the mapping of participants to companies is presented in Table 4.3, together with a summary of each participant’s industrial experience. Of the six companies, two agreed to be named in this study.

- **Visma** is a large-scale software development company that delivers purchasing solutions, payment services, in-store data solutions, and consulting activities to small and large businesses.

- **Company C2** is a large-scale software development company specializing in cloud-based financial and accountancy services based in Sweden. C2 presents a mature company in terms of development practices, with a portfolio of well-established products.

- **Company C3** is a software consultancy company based in Sweden specializing in system development, architecture, and quality assurance solutions.

- **Company C4** is a software development provider specializing in Java, .Net, and cloud technologies and based in Sweden.

- **QESTIT** is a large European software consultancy company specializing in software testing and IT security.

- **Company C6** is a medium-sized company specializing in the use of AI and machine-learning solutions for various domains. The company develops its own solutions and provides consultancy, primarily in Sweden.
4.3 Method

Table 4.3: Professional experience of participants in industry and mapping to their companies

<table>
<thead>
<tr>
<th>Participant</th>
<th>Company</th>
<th>Session</th>
<th>Industry</th>
<th>Testing</th>
<th>GUI testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Visma</td>
<td>in-person</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>P2</td>
<td>Visma</td>
<td>in-person</td>
<td>25</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>P3</td>
<td>Visma</td>
<td>in-person</td>
<td>10</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>P4</td>
<td>C2</td>
<td>online</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>P5</td>
<td>C2</td>
<td>online</td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>P6</td>
<td>C3</td>
<td>online</td>
<td>3</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>P7</td>
<td>C4</td>
<td>online</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>P8</td>
<td>QESTIT</td>
<td>online</td>
<td>15</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>P9</td>
<td>QESTIT</td>
<td>online</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>P10</td>
<td>C6</td>
<td>online</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P11</td>
<td>QESTIT</td>
<td>online</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>P12</td>
<td>QESTIT</td>
<td>online</td>
<td>27</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>P13</td>
<td>C6</td>
<td>online</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

4.3.5 Experiment design

In this subsection, we provide detailed information about the general design principles, the object (SUT), and the instrumentation of the experiment.

We addressed the general design principles as follows:

- **Randomization**: All participants will undertake tasks with both treatments in a counterbalanced order, each starting with the treatment opposite to the previous participant. The participants independently selected their preferred time slots from an available list. We consider this a randomized order, as no pre-defined allocation of participants to treatment was done beforehand.

- **Blocking**: Blocking eliminates undesired effects in a study from additional factors, enhancing the experiment’s precision by not studying the effects between blocks [88]. We ensured a similar level of test case sizes, measured in test steps, in both treatments to eliminate effects from varying sizes.

- **Balancing**: In a balanced design, each treatment involves an equal number of participants which strengthens the statistical analysis of the data [88].
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

To balance the data set, each participant performs both treatments, and the treatment order is changed with each iteration. Balancing based on factors such as experience or company affiliation was not done.

- **Standard design type**: One factor with two treatments.

**Object application**

We chose the following requirements to select a suitable SUT.

- **Compatibility with the test environment**: The SUT should be able to run locally on the test environment to reduce the delays when loading pages. Since our test environment uses an ARM CPU, it can cause incompatibilities with dependencies in some technology stacks. Section 4.3.6 provides more details about the test environment.

- **Web-based application**: Our testing tool, Scout, is currently limited to web-based GUIs due to its reliance on Selenium WebDriver, an automation framework for web-based applications, to interface with the SUT’s GUI [156].

- **Manipulation of the source code**: Modifications to the SUT’s source code should be feasible to introduce potential defects, bring the SUT into a specific state for testing, deactivate features, and address minor issues.

- **Manageable level of complexity**: The SUT should have a manageable level of complexity that would not necessitate specialized knowledge for understanding and testing purposes.

- **Variety of features**: The SUT should have a feature set that would allow the creation of different test cases. For instance, a simple toy application would be inadequate.

To identify a suitable SUT, we referred to a study conducted by Coppola et al. [159], in which the authors used a list of open-source web-based projects available in grey literature\(^1\). Although the grey literature list presented several other potential candidates, none of them fully satisfied our requirements. Consequently, we made the decision to use the web-based project Make-Your-Menu developed by the main author for the experiment. Projects were primarily excluded from this list due to setup challenges in our testing environment, [1]https://github.com/unicodeveloper/awesome-opensource-apps
such as incompatible dependencies. Additionally, a lack of diverse features led to exclusions. For example, content management system (CMS) software like Mezzanine, suitable for exploratory testing through content creation, fell short in feature diversity for regression tests.

Make-Your-Menu (MYM) is a web-based recipe management application specifically designed for educational institutions that offer cooking as part of their curriculum. This application serves as a resource for teachers, supporting them in creating, preparing, and sharing recipes while also providing nutrition and allergy information. For instance, teachers have the capability to create new recipes or modify existing ones, comprising multiple food items along with detailed preparation instructions and serving sizes. Based on the included food items, a summary of nutritional facts and allergy information is automatically generated. These recipes can be conveniently reused by others for meal planning purposes within their own courses. Furthermore, the application facilitates the compilation of a shopping list, which presents a printable overview of all required ingredients arranged according to their supermarket sections and prices. To facilitate efficient food item searching, the application provides users with a user-friendly search field as well as the option to perform advanced searches within specific categories. Moreover, Make-Your-Menu offers seamless language switching between English and German to accommodate teachers’ language preferences. Screenshots of MYM are shown in Figure 4.2.

We formulated eight test cases for the SUT. Each test case description includes a list of the actions to perform with the required input values and the expected results. Detailed test case descriptions are available in the replication package. Below is a concise summary of each test case:

- **TC1**: Verify login into the system with an email address and password is possible given the correct credentials.

- **TC2**: Verify that switching the system’s language between English and German is possible. All visible labels that are not content should switch language.

- **TC3**: Verify that a new victual can be created and deleted afterward.

- **TC4**: Verify that a recipe that contains at least two victuals can be created and deleted afterward.

- **TC5**: Verify that a victual can be found via the search and that detailed information about the victual is displayed.
• TC6: Verify that feedback can be sent to and read by the product owners.

• TC7: Verify that victuals and recipes can be added to the shopping list. Added items should be visible in the shopping list view, the sidebar, and as a notification badge in the top bar.

• TC8: Verify that a copy of a recipe can be created and that it shows up in the list of only your recipes. Afterward, delete the recipe.

The SUT utilized in this study was well-tested to maximize the chance of it being free of bugs that could affect any of the prescribed tests. This deliberate choice enables us to focus on evaluating the efficiency of AT itself without unexpected behaviors caused by the faulty SUT.

Participants in the study do not require to have specialized domain knowledge in order to understand and use MYM. We base this assumption on the commonality of features in MYM to most e-commerce systems. For example, creating, deleting, searching for items, or adding items to shopping lists. Additionally, MYM adheres to the Material Design guidelines established by Google [148], which is the standard design language for Google applications and is commonly used in Android applications. Following the Material Design guidelines should ensure a familiar and intuitive user experience. Since none of the participants had prior experience with this particular web-based project, all participants started with the same level of familiarity with the SUT.

Technology-wise, Make-Your-Menu uses the JavaScript web framework Vue.js\(^2\) for the frontend and Firebase\(^3\) as a backend in the cloud, also known as backend-as-a-service. The backend system undertakes various essential tasks, including user authentication, data storage, and the execution of cloud functions to compute meta-data. In total, the project consists of 50k lines of code (LOC). For comparison, most empirical studies work on rather small SUTs up to 12.5k LOC [48]. The GUI of MYM consists of 23 different views, i.e., the shopping list view, and 16 reusable GUI components, i.e., the language switcher.

4.3.6 Instrumentation

We set up a testing environment using a MacBook Pro (Apple M1 Max CPU, 64 GB RAM) running macOS version 13.4 that we used for all sessions. Scout\(^4\)

\(^2\)https://vuejs.org
\(^3\)https://firebase.google.com
\(^4\)The specific Scout version used during the experiment had the Git commit hash 3fd8f40
was executed with Java in the version openjdk 1.8.0_362. In cases where in-person sessions were not feasible, we used the video conference tool Zoom and its remote control feature that allows online participants to control the testing environment. Table 4.3 provides an exact overview of which sessions were conducted in-person vs. online. Regardless of session type, online and in-person sessions, subjects were presented with the same amount of visible content, but the perceived size of the content can vary depending on the participant’s display setup. Our SUT, MYM, was run locally on the testing environment hardware while cloud functionality and content data remained in the Firebase cloud. We recorded the screen and audio for subsequent analysis, enabling precise measurement of test case execution and transcription of interview questions. No noticeable performance impacts on the testing environment or Scout were detected as a result of the recording process.

With its modular design, Scout facilitates regression and exploratory GUI-based testing, providing diverse support to testers. It offers functionality to suggest checks and actions or adapt the test model to changes in the SUT. However, we limited the feature set of Scout in this study and only enabled relevant plugins to study the effect of augmentations on manual GUI-based regression testing. These augmentations primarily involve highlighting checks and actions to be performed. Figure 4.3 demonstrates these augmentations. Green rectangles around widgets represent checks performed automatically on a selected widget, e.g., to check if the text property of a widget is equal to a specific text (Figure 4.3a). Whereas blue rectangles around widgets represent actions required to progress within the test case, like clicking on a button to progress within the test (Figure 4.3b) or typing text into an input field (Figure 4.3c).
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

Before the experiment started, we explained the confidential handling of interview answers and measurements to ensure the participants that traceability between individuals, or their affiliation, to single data points is impossible. We then asked for explicit consent to record the experimental session. The purpose of these records is to ensure accurate data collection and analysis. Participants have the right to withdraw from the study at any time without any consequences. Further, we allow all participants to review all materials before they are made publicly available.

The execution of the experiment is divided into five steps and takes in total 90 minutes.

**Step 1) Introduction to Augmented Testing (AT).** The initial stage of the experiment involves providing participants with an introduction to AT with Scout. During this introduction, all relevant concepts of AT are explained, accompanied by a demonstration of Scout, the testing tool, and its features. The explanation covers the various ways in which augmentations are superimposed on the widgets of the SUT. Additionally, participants are introduced to the distinct interaction approach with the SUT in Scout. Notably, Scout records interactions before handing them to the SUT, resulting in a workflow that differs in some ways from the familiar usage of a browser. For example, rather than clicking into a text field and then typing, users are instructed to first type the text and subsequently click into the text field. Similarly, scrolling through a page is achieved solely using the side scroll bar, as opposed to the customary mouse scrolling method. Nevertheless, differences in interactions might potentially affect the overall results, e.g., participants could spend more time in the beginning. It is important to note that this effect is not limited to a specific treatment, as Scout was used in both treatments instead of a web browser for manual GUI-based regression testing.

**Step 2) Warm-up phase.** The warm-up phase aims to familiarize participants with the tool, particularly emphasizing aspects that deviate from conventional user interactions with a web GUI, such as scrolling and text input.

Figure 4.3: Three examples of augmentation superimposing the SUT’s GUI
Wikipedia\(^5\) serves as the SUT for the warm-up exercises, comprising four simple tasks that participants are required to perform. These tasks consist of searching for the Wikipedia page dedicated to the participant’s company, the page of our research institute, switching the language of an article, and visiting an article suggested on the main page of Wikipedia. Participants first perform the tasks without any augmentations, like a manual GUI-based testing process. Subsequently, a second round of the same tasks is performed, this time with augmentations derived from the interactions of the previous run. Participants are instructed to employ the think-aloud protocol, articulating their thoughts and perceptions throughout the task execution without engaging in an immediate analysis [160]. After this warm-up phase, we were confident that all participants understood the task and the tool’s usage.

**Step 3) Description of SUT.** A verbal explanation of the SUT’s features was provided, encompassing the primary use cases, the target user group, and the features relevant to the upcoming testing tasks. The given explanation is similar to the information presented in Section 4.3.5.

**Step 4) Test execution.** The description of the test cases is distributed to the participants either in printed form or as a PDF document for online sessions. The researcher orally reads the test case description, ensuring clarity and understanding by allowing the participants to ask clarifying questions before the test starts. Subsequently, the researcher prepares the run of a test case in Scout by selecting pre-planned test cases according to the specific type of treatment for each participant. The sequence in which the test cases are executed and the particular treatment are presented in Table 4.4. Like with the warm-up tasks, participants were instructed to employ the think-aloud protocol and notify the researcher once they had completed each test case. During the test execution, the researcher stays in the background and only interferes when technical problems arise. For example, participants occasionally interacted with incorrect widgets, resulting in an overlay that left them unsure of how to close it.

We use ISO Central Secretary [161] definition of efficiency, which is *“the extent to which time, effort or cost is well used for the intended task or purpose”*. The efficiency measurement is determined by the time required to complete each test case. Less time spent per test case is considered to be better, thus, more efficient. To ensure consistent time measurements, we use the moment when participants initiate mouse movement as the starting point. Additionally, participants were instructed to signal us once they had completed a test case,
enabling us to utilize this signal as the endpoint for time measurement. Waiting for a signal by the participants adds some uncertainty in the time measurement, but this overhead time is negligible compared to the total test execution time and, therefore, not considered a confounding variable. Exact measurements of the duration of each test are later derived from the video recordings.

**Step 5) Interview about benefits and drawbacks.** To address RQ2, we conducted semi-structured interviews with the participants after each session. The interviews focus on the participants’ reflections and observations from the session to understand how they see AT compared to traditional manual GUI-based testing. Explicitly, the following questions were asked of each participant:

- Do you see benefits of using AT for manual GUI-based regression testing?
- Do you see drawbacks of using AT for manual GUI-based regression testing?
- Do you see an aspect that would prevent the usage of AT in an industrial context?
- Would you use AT for manual GUI-based regression testing?

The responses largely reflect individual perceptions based on limited exposure to the testing tasks and may not indicate participants’ likelihood of adopting AT in practice. After the interview, we gathered the experience levels of the participants in *industrial experience, experience with manual or automated testing, and experience with GUI-based testing.*

**Application of treatment and no treatment.** Both treatments, Augmented Testing (AT) and manual GUI-based regression testing (baseline) are performed using our testing tool, Scout. In the no-treatment scenario, all support augmentations, i.e., indications of the next widget to interact with or checks, are disabled. Participants rely solely on the textual description of test cases, which detail input and expected output, to perform manual tests. Table 4.4 presents the precise mapping of participants, treatments, and test cases. The execution order of the test cases follows a fixed sequence: $TC1 \rightarrow TC3 \rightarrow TC5 \rightarrow TC7 \rightarrow TC2 \rightarrow TC4 \rightarrow TC6 \rightarrow TC8$ Treatment transition occurs after half of the test cases have been executed. We selected this order deliberately to account for the increasing complexity in each feature set, which is determined by the number of test steps. For instance, $TC3 \rightarrow TC5 \rightarrow TC7$ covers victuals features of the SUT, while $TC4 \rightarrow TC6 \rightarrow TC8$ focuses on recipe features. Test cases for both feature sets have comparable numbers of steps and
demand similar cognitive effort, an intended design to make them comparable. Although the execution order of test cases remains constant, the treatment of each half alternates for each run. This design aims to mimic a crossover arrangement, effectively delineating feature sets without introducing a separate SUT. Nonetheless, the fixed order introduces a potential learning bias due to carry-over effects between the test cases. Such effects could influence participants’ performance in subsequent test cases based on their prior experience. For example, TC5 and the subsequential TC7 both include the search for a specific victual. In our data analysis, we incorporated an independent variable to address potential learning effects.

During the test case execution, augmentations, as shown in Figure 4.3, were provided to guide the participants through the test cases when AT was used as a treatment. Here, the participants can interact with a widget, highlighted through augmentations, by clicking on it to continue to the next test step. Scout derives these augmentations from a pre-recorded run of the test cases by the researcher. Participants also had access to the textual description of the test cases either as PDF or printed copies, serving as a reference in case they encountered any uncertainties regarding the next steps within the test. This approach aims to minimize confounding factors arising from the tool’s operation, such as tool constraints or different ways to interact with the SUT’s GUI.

All relevant material, description of test cases, interview questions, measurements, observation notes, source code of SUT, and the source code of the statistical analysis are provided as a replication package.

4.3.7 Analysis Method using Bayesian Data Analysis (BDA)

We employ Bayesian data analysis (BDA) for causal inference following an established workflow by Gelman et al. [49]. We opt for BDA over the more common frequentist statistical methods like null-hypothesis tests against a fixed significance level [88] for several reasons as summarized by Furia et al. [41]. Most importantly, the inferences made using BDA are more detailed and remain explicit about uncertainty [163]. The application of BDA became accessible with the computational availability of Markov Chain Monte Carlo (MCMC) randomized algorithms [166] and has already been successfully applied to software engineering research [42, 162].

To begin, we make our causal assumptions explicit in a directed, acyclic graph (DAG). The DAG visualizes all variables of interest as nodes and all assumed causal relationships as directed edges between these nodes [164]. The
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

<table>
<thead>
<tr>
<th>Participant</th>
<th>Treatment per test case and number of steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P2</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P3</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P4</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P5</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P6</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P7</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P8</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P9</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P10</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P11</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
<tr>
<td>P12</td>
<td>A  A  A  A  M  M  M  M</td>
</tr>
<tr>
<td>P13</td>
<td>M  M  M  M  A  A  A  A</td>
</tr>
</tbody>
</table>

Table 4.4: Assignment of treatments to participants. Subjects get both treatments in a randomized order (M = Manual Testing, A = Augmented Testing)

DAG serves two main purposes: (1) it makes all causal assumptions explicit, which clearly delineates our horizon of considered variables and enables future research to scrutinize and extend these assumptions; and (2) based on four graph criteria [163], it allows determination of which variables to include in the statistical evaluation and which to exclude based on their role as confounders.

After the selection of appropriate variables according to the DAG [163], we perform a regression analysis to model the relationship between the independent and dependent variables using the brms package in R [26], which in turn relies on the probabilistic programming language Stan [167]. We model the dependent variable, normalized time for completing a test case, as a Gaussian distribution based on the maximum entropy criterion [168] and ontological assumptions. For the coefficients of each independent variable with an assumed causal effect on the dependent variable, we select prior distributions and confirm their appropriateness using graphical prior predictive checks [169]. Upon confirming the appropriateness of these priors, we train the Bayesian model using the data from the experiment. In this process, the model updates the variable coefficients according to Bayes’ Theorem such that each coefficient reflects the
impact of its independent variable on the dependent variable [163]. After training, we confirm the appropriateness of the trained model via graphical posterior predictive checks [169].

Once confirmed, we evaluate the model by sampling from the posterior distribution of variable coefficients [165]. To this end, we construct a synthetic data set in which only the treatment (the use of AT) varies, while all other independent variables are fixed to representative values (i.e., the mean for continuous variables and the mode for discrete variables). The only exception concerns the independent variable of test case size, which we uniformly distributed in these synthetic data sets as well. We performed 12,000 predictions for each of the two data points (using AT, not using AT), simulating the isolated effect of only our factor of interest. This way, we maintain the model uncertainty encoded in every variable coefficient. We compare the predicted values of the dependent variable between the two treatments and summarize the likelihood distribution that the existence of the treatment has a positive, negative, or no effect on the dependent variable.

4.4 Results

This section reports the experimental results involving 13 participants, detailing AT’s efficiency, benefits, drawbacks, and usability issues.

4.4.1 RQ1 - Efficiency

We begin by presenting the collected data through diagrams and a table, then employ BDA to address RQ1. First, we examine the total duration it took each participant to complete all test cases, combining both treatments, as presented in Figure 4.4. This helps understand the variation among participants. Next, Figure 4.5 illustrates the duration to complete each test case by all participants as a distribution, combining both treatments. As detailed in Section 4.3.6 and outlined in Table 4.4, test cases were executed in the sequence: $TC1 \rightarrow TC3 \rightarrow TC5 \rightarrow TC7 \rightarrow TC2 \rightarrow TC4 \rightarrow TC6 \rightarrow TC8$. The treatment changed after the completion of half the test cases. Figure 4.5 confirms that the time needed to complete both halves of all test cases is balanced. To achieve this balance, the design involved the creation of pairs of test cases with a similar number of steps (see Table 4.4) while covering different features of the SUT.

Table 4.5 displays the differences in the mean time to complete each test case per treatment. Performing all test cases took an average of 1222 seconds
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

Participants

<table>
<thead>
<tr>
<th>Participant</th>
<th>Test Duration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>983</td>
</tr>
<tr>
<td>2</td>
<td>858</td>
</tr>
<tr>
<td>3</td>
<td>759</td>
</tr>
<tr>
<td>4</td>
<td>724</td>
</tr>
<tr>
<td>5</td>
<td>808</td>
</tr>
<tr>
<td>6</td>
<td>955</td>
</tr>
<tr>
<td>7</td>
<td>1112</td>
</tr>
<tr>
<td>8</td>
<td>1303</td>
</tr>
<tr>
<td>9</td>
<td>1123</td>
</tr>
<tr>
<td>10</td>
<td>901</td>
</tr>
<tr>
<td>11</td>
<td>858</td>
</tr>
<tr>
<td>12</td>
<td>724</td>
</tr>
<tr>
<td>13</td>
<td>1554</td>
</tr>
</tbody>
</table>

Figure 4.4: Overall test duration of all test cases per subject, measured in seconds

Using the conventional manual method, compared to 779 seconds with AT, a reduction of 36% in average completion time. Offering an additional perspective on the gathered data, Figure 4.6 displays a probability density distribution of the time required to finish each test case.

By referring to Figure 4.6 alongside Table 4.5, we can observe that six out of the eight test cases exhibit shorter times to complete the test cases, indicating higher efficiency. An exception to this trend is TC1 and TC2, where the difference between the two treatments is small and in favor of the baseline treatment. Notably, TC1 and TC2 represent the test cases with the fewest steps (see Table 4.4) and thus result in a shorter duration to complete them with lower differences.

Bayesian data analysis (BDA). In the remainder of this section, we present the Bayesian data analysis (BDA) results as described in Section 4.3.7. Figure 4.7 visualizes the DAG, which makes the assumed causal relationships of all variables of interest explicit. We consider the scaled test duration to be impacted by the use of AT, the size of the test case, and the learning effect of repeatedly using the AT interface, as listed in Table 4.2.
After model exploration and comparison, we moved forward with the model described in Equation 4.1, which models the collected data best according to the PSIS-LOO information criterion [87]. Detailed information about all models and their comparisons can be found in the replication package.

\[
\text{duration\_scaled} \sim \text{Skewed\_Norm}(\mu_i, \sigma, a) \\
\mu_i = \alpha_i + \beta_{AT} AT_i + \beta_{\text{size}} size_i + \beta_{AT \times \text{size}} AT_i \times size_i + \beta_{\text{learn} \times \text{learn}i} \times AT_i 
\]

(4.1)

The response variable \text{duration\_scaled} is modeled with a skewed normal distribution, the shape of which is defined by its mean \(\mu_i\). Hence, \(\mu_i\) is modeled by the independent variables. The terms equating to \(\mu_i\) (second row in Equation 4.1) represent the causal assumptions from the DAG in Figure 4.7 in the following way:

- \(\alpha\): the intercept. It represents the average mean of the normal distribution in the absence of all factors. Since the response variable is normalized, this is expected to be close to zero.
- \(\beta_{AT} AT_i\): the influence of augmented testing.
- \(\beta_{\text{size}} size_i\): the influence of the test case size.
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

Figure 4.6: Distribution of test duration per test case with a separation of treatments

Augmented Testing (AT)

- $\beta_{AT \times size} AT_i \times size_i$: the interaction between the use of AT and the test case size (in case the use of AT has a different effect on the response variable depending on the size of the test case).

- $\beta_{learn \times AT_i} learn_i \times AT_i$: the learning effect of repeatedly using the augmented testing interface.

For each coefficient, we selected appropriate priors. During the training process, these coefficients are updated based on the observed data and Bayes’ Theorem. After the training, each coefficient represents the strength and direction with which an independent variable influences the mean of the response variable.
4.4 Results

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Mµ</th>
<th>Aµ</th>
<th>Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>64</td>
<td>73</td>
<td>+14%</td>
</tr>
<tr>
<td>TC2</td>
<td>36</td>
<td>41</td>
<td>+14%</td>
</tr>
<tr>
<td>TC3</td>
<td>256</td>
<td>180</td>
<td>-30%</td>
</tr>
<tr>
<td>TC4</td>
<td>246</td>
<td>136</td>
<td>-45%</td>
</tr>
<tr>
<td>TC5</td>
<td>101</td>
<td>54</td>
<td>-47%</td>
</tr>
<tr>
<td>TC6</td>
<td>101</td>
<td>51</td>
<td>-50%</td>
</tr>
<tr>
<td>TC7</td>
<td>196</td>
<td>139</td>
<td>-29%</td>
</tr>
<tr>
<td>TC8</td>
<td>222</td>
<td>105</td>
<td>-53%</td>
</tr>
<tr>
<td>Total</td>
<td>1222</td>
<td>779</td>
<td>-36%</td>
</tr>
</tbody>
</table>

Table 4.5: Mean time to complete each test case per treatment, measured in seconds (M = manual GUI-based regression testing, A = Augmented Testing)

Figure 4.8 visualizes the strength of the interaction effects ($\beta_{AT \times size}$, $AT_i$ and $\beta_{learn}$) which the model picked up after training. As shown in Figure 4.8a, the use of AT has a stronger benefit on large test cases, as the difference between the mean of the response variable duration_scaled for size = big is larger than for size = small. Figure 4.8b shows that the repeated use of the AT interface slightly reduces the average duration_scaled, confirming the existence of a learning effect.

Finally, the sampling from the posterior distribution reveals that in 70% of all cases across both test case sizes, the use of AT results in a lower value for duration_scaled. This means that considering all uncertainty encoded in the probability distributions of the model, the use of AT results in a shorter test duration in 70% of all cases on average compared to manual GUI-based regression testing. Consequently, we reject the null hypothesis $H_0$ and accept the alternative hypothesis $H_1$.

**Answer to RQ1:** Accounting for all uncertainties in our BDA model, we found that AT decreases the test duration in 70% of all cases on average compared to manual GUI-based regression testing, making it more efficient in those cases. When comparing the means of the total duration to perform all tests, AT reduced the time spent by 36%.
Augmented Testing to support Manual GUI-based Regression

Figure 4.8: Visualization of the interaction between the two predictor variables
4.4.2 RQ2 - Benefits and drawbacks

This section presents our interview findings on AT’s benefits and drawbacks, mirroring the structure of Table 4.6 that consolidates these results.

During the interview process, as outlined in Section 4.3.6, we asked questions that specifically focused on AT as the testing approach rather than the tool Scout, which serves as its demonstrator. However, due to participants’ exposure to AT exclusively through Scout, they may be unable to fully differentiate between the testing approach and the tool. Therefore, we need to categorize the interview results as related to the testing approach (AT) or the tool. For instance, Brooks and Kugler [172] distinguishes between essential and accidental software engineering difficulties. Essential difficulties are inherent in the nature of software itself and are often unsolvable due to their fundamental nature. In contrast, technical or other means can address or mitigate accidental difficulties. These difficulties can emerge from utilizing software but are not fundamental to the software.

We apply this classification of essential and accidental difficulties to our identified benefits and drawbacks of AT. Thus we define essential benefits and drawbacks as those that emerge from the testing approach AT and accidental benefits and drawbacks as those that emerge from the tool utilizing AT.

Benefits of using AT for manual GUI-based regression testing

**B1 Efficient (faster) test execution.** The primary reported benefit is the faster execution of test cases, confirming results from previous studies [65, 66]. Other benefits outlined in this section contribute to this increase in efficiency. Participants reported reduced time in understanding and executing the subsequent step in a test case due to augmentations. Further time savings were achieved because input fields, such as a search field, were pre-populated with input values. This benefit likely arises from the cumulative benefits of AT and its associated tool; hence, we do not classify it separately.

**B2 Visual cues indicating next action.** Participants appreciated the ability of AT to provide visual cues atop the SUT’s GUI through augmentations, spotlighting the next action necessary for test case progression. As one participant stated “I really liked that each component that forwards the test is highlighted” (P3). As presented in Figures 4.3b and 4.3c, these augmentations guide understanding of the next expected action and prevent context switches to read test case descriptions. Further, augmentations can eliminate context switches to other applications or media to determine the next action when per-
forming a test case. Since using augmentations as a form of visual guidance is a core part of AT, we categorize this benefit as essential.

**B3 Elimination of typos.** Augmentations on widgets auto-fill the input fields with the correct values, reducing the chance of typos or incorrect entries. Participants managed to avoid errors during their inputs, as these values appeared when hovering over the widgets and could be easily submitted to the SUT with a single mouse click. During the experiment, when the pure manual GUI-based testing treatment was employed, some participants encountered issues with typos while searching for specific victuals and recipes. This benefit can be categorized as accidental. Prepared input values are not enforced and can be overridden or ignored by the tester. A different implementation of AT could just recommend values but require a tester to type the values themselves.

**B4 Monitor test execution.** Augmentations that highlight performed checks on widgets, as shown in Figure 4.3a, can be used to monitor which widgets are checked or need to be checked for each screen of the SUT. Participants valued this feature for offering a clear overview of the tested SUT components. For instance, participant P4 remarked, “It is a good way to monitor what you have done and tested and what you have not tested”. Previous studies highlighted this benefit in the context of exploratory testing, where this aspect is more essential. Participant P7 further highlighted its utility for retrospective test analyses, stating, “You interacting with what you are testing, and you could go back and look into it more and see what happened”.

**B5 Reduced effort for non-primary actions.** In such cases, augmentations assist a tester with navigation and preparing required input values. Some activities during testing are not the primary objective of a test case but are required to bring the SUT into a specific state for the test. For example, participant P11 stated “It reduces unnecessary steps. You don’t have to log in every time.”. Logging in to the SUT is essential to progress in certain test cases. Therefore, augmentations can assist in fulfilling this precondition. This assistance was seen as a reduction in effort required by the tester. However, this benefit emerges from augmentations and how they support testers with navigation through the SUT. Thus, we categorize it as an essential benefit.

**B6 Helps not to forget steps or checks.** Participants see a benefit in AT’s ability to guide a tester through the SUT and ensure no essential actions or verifications are overlooked. This was observable during the experiment when larger forms to create a new recipe were required to fill out. AT assisted in remembering the tester to fill these fields during the tests. The guidance through the SUT using augmentations is essential to AT; thus, we categorize this benefit as essential.
B7 Test case reusability. Augmentations in AT, generated by recording the interactions with the SUT, allow the reusability of test cases in an easy way. Once recorded, test cases can be shared with other testers, enabling them to perform test cases using exactly the same steps by following the augmentations. Participants reported the ability to reuse manually performed GUI-based tests by other testers as beneficial: “If I created the steps and someone else can click through the steps and verify that everything is where it is supposed to be” (P1). We categorize this benefit as essential to AT since a recording of performed test cases is a byproduct of AT, independent of its specific implementation or its representation of test cases.

B8 Usable for non-experienced testers. In addition to the previously mentioned benefit of reusable test cases, one participant reported that test cases with AT could be performed by people who are not experienced testers. This confirms a previous study by Nass et al. [66], where AT with Scout was also perceived to require less programming and test automation experience compared to a script-based technique. We categorize this as an essential benefit, as other testing approaches fitting the AT definition also reported the usability for non-experienced testers.

B9 Preferable to textual step descriptions. One participant reported that he/she would prefer to prepare a test case with AT instead of writing down a textual description of the exact steps within a test case. They found recording a test case using AT and reusing it by other testers would be easier. This benefit is related to B7 “Test case reusability” and can be seen as an extension of B7.

Drawbacks of using AT for manual GUI-based regression testing

D1 Neglecting non-augmented widgets. Participants reported a risk of ignoring non-augmented widgets during AT testing, potentially leading to missed defects. Although augmentations do not prevent checking other widgets or SUT exploration, the tendency to follow a specific path and inspect specific widgets during the test was noted. This flexibility to deviate from the test was deemed important even in regression testing scenarios. We categorize this drawback as essential since it emerges from the usage of augmentations and not a specific implementation. In our experimental setting, participants were explicitly instructed to follow the given regression test cases. Whereas in a real-world setting, such instructions would not necessarily be in place or be as rigid, potentially rendering this drawback less significant.

D2 Issues of augmentation purpose comprehension and distinguishability. Participants encountered difficulties comprehend the purpose
of augmentations and distinguishing between various types. In the experiment, two specific augmentations, distinguished by their color, were utilized: green rectangles symbolizing checks on widgets and blue rectangles indicating actions to be performed on widgets. Additional text such as Check, Click, or Type is presented when hovering over an augmented widget, as demonstrated in Figure 4.3. Nevertheless, participants reported ambiguity during the testing phase regarding the exact purpose of a given augmentation and the expected interaction with a widget. We consider this issue an accidental drawback that can be mitigated by altering the visual representation of augmentations, such as incorporating symbols alongside the color highlights. One participant proposed displaying a pop-up screen before the test starts, briefly describing the different augmentations.

D3 Invisibility of off-screen augmentations. When widgets are located outside the visible screen area, their augmentations remain unseen. Participants reported that they missed an indicator when augmentations were outside the visible screen area, and they had to search for them actively. For example, the screen to create a new recipe contains a form with multiple inputs that exceed the visible screen area. At the end of this form, the button to save a new recipe is located and can be overlooked when not scrolling down. This presents an accidental drawback of AT. The tool could provide additional indicators for augmentations that are located outside of the visible screen area. In manual testing, a tester also lacks indicators to locate targeted widgets on the SUT’s GUI.

D4 Color visibility issues. One participant reported that augmentations are hard to see. In some instances, augmentation colors matched widget or background colors, making them hard to see. Participant P5 stated, “Some colors are similar to each other, and sometimes you don’t know where to click.” We categorize this drawback as accidental since it can be addressed through adjustments in the visual representations of augmentations. For example, adopting accessibility guidelines for limited-vision users such as “Don’t rely on hover for critical information” or “Avoid color-exclusive meaningful functionality” [149].

D5 Test case goal can not be derived purely from augmentations. As reported by one participant, solely following the augmentations does not provide an understanding of the overall goal of a test. Thus, augmentations that represent the single steps of a test do not provide context information that allows one to understand the goal of the test. A pop-up screen summarizing the test’s overall goal, presented before the test starts, would address this drawback, making it an accidental drawback.
D6 Non-intuitive SUT interaction. Some participants expressed that interacting with the SUT is not intuitive. As highlighted in Section 4.3.6, how a tester interacts with a SUT in Scout differs from standard browser interactions, especially regarding scrolling and data input. As a participant reported: “But for some things you have to get used to. Scrolling and input of text.” (P4). Scrolling is not enabled through the mouse wheel and is solely feasible via the sidebar scrollbar. The way of inputting data was reported as not intuitive and is in contrast to what is commonly expected when interacting with an input widget, like a search field. Participant P3 stated “Since I’m used to navigating by first clicking and then typing, it was kind of hard to first type and then click.”. These drawbacks can be directly attributed to the tool and are hence classified as accidental drawbacks. However, it remains uncertain whether this challenge will persist as users become more familiar with the tool, especially given that participants noted the need for familiarization.

D7 Input values limitations. Input values for input fields are provided during AT from previous interactions with the particular widget. A participant noted a limitation and reported that utilizing a particular input value or a narrow set of such values hampers the identification of defects during a test. However, in the experiment, augmentations guide the tester through a specific regression scenario by following exactly one path through the SUT for each test case. We categorize this drawback as accidental, given that AT does not inherently limit input values allowing testers to deviate in this regard. Additionally, AT could be used to suggest new input values.

D8 Utility limited to small tests. Regarding the usefulness of AT, one participant reported that he/she deemed AT’s usefulness only for small test cases with expected results. This could be attributed to previously described drawbacks of D5 “Test case goal can not be derived purely from augmentations” and D7 “Input values limitations”. We categorize this limitation as an accidental drawback. The complexity of test cases with AT is not limited to a specific size. In addition, the reported benefits and the analysis of measurement using BDA indicate increased usefulness for more complex test cases.

D9 Unclear mistake correction process. One participant reported that it was unclear how to correct mistakes during the test. Mistakes are accident interactions with widgets. For example, adding unnecessary checks by clicking on a widget or typing wrong input values into input fields. Deviations from the planned steps in the test case will create a new branch within the test case that does not include the augmentations of the main branch. In Scout, there is a way to remove augmentations from widgets and branches from test cases. However, an introduction to this functionality was not part of the experiment
Augmented Testing to support Manual GUI-based Regression Testing: An Empirical Study

since modification of test cases was not planned and to avoid additional overhead in learning this feature during the warm-up phase. We see this as a clear accidental drawback. For instance, adding indicators when deviating from the planned steps and giving a possibility to go back directly could mitigate this drawback. Additionally, an introduction to the main controls, next to a short description of the types of augmentations, could be provided as a pop-up when a test case is started. Familiarity with the tool could further mitigate this drawback. Finally, this challenge is also present in manual testing, where reverting changes and restoring a SUT’s state can become complex.

D10 High tool reliance. AT aims to support testers during manual GUI-based regression testing. Participants voiced concerns that testers might overly rely on AT and the tool, potentially reducing intellectual participation during the test. This could hinder considering diverse input values or exploring alternative strategies to achieve a test case’s goal. Should testers lessen their engagement and intellectual participation, negatively affecting the detection of defects, it would render this aspect of AT an essential drawback.

D11 Overlays cover large screen area. Hovering over a widget displays additional information like the text label value to be checked. For large text values, this overlay becomes large and obstructs a significant portion of the SUT’s GUI. One participant reported large overlays as a drawback. Limiting or disabling the overlay for large text values would mitigate this drawback and thus can be categorized as an accidental drawback.

Aspects that would prevent the use of AT in an industrial context

Out of the 13 participants, six did not identify any barriers that would hinder the adoption of AT in an industrial context. However, two participants expressed concerns regarding data privacy if AT requires storing information of input values or displayed content of a SUT. They specifically raised concerns about data protected by the General Data Protection Regulation (GDPR) or sensitive data from governmental organizations.

One participant highlighted that testing non-production-ready products with AT would require additional effort in recording AT test cases in a fast-paced development environment. Due to the rapid changes frequently encountered in such environments, these test cases may quickly become invalid or outdated. However, the participant also clarified that this issue does not pose a problem when testing production-ready products.

Another factor that may avoid the use of AT in an industrial context is the associated costs related to the necessary tooling. Certain companies may
hesitate to explore AT for quality assurance activities if these costs are high. This study, however, focuses on the AT approach and does not cover the business models of the required tools implementing AT.

The last concern is regarding the documentation of test cases. In some organizations, documentation in the form of reports detailing the execution of test cases is a requirement. Moreover, there is a preference for integrating these reports into other systems, such as the issue tracker Jira, to facilitate comprehensive test tracking within the broader development workflow. This concern about reporting results also belongs to the tooling and not AT itself. Future tool development should address this to enhance industrial applicability.

All concerns raised that would prevent the use of AT in an industrial context can be classified as accidental, as they primarily stem from the tooling rather than the testing approach itself. Therefore, these concerns can be mitigated by making adjustments or implementing improvements to the tooling of AT.

Behavioural intent to use

Among the 13 participants, 10 expressed their intention to utilize or explore AT for manual GUI-based regression testing. Interestingly, one participant articulated an intention to utilize AT specifically for complex test cases while opting for pure manual GUI-based testing for smaller ones. This perspective contrasts with a previously mentioned drawback, where a participant suggested that AT would benefit smaller test cases with predictable outcomes. Conversely, only one participant stated that they would not use AT due to its detachment from the development workflow, as it may introduce friction when using a separate testing tool that is not integrated into the integrated development environment (IDE). This is an accidental issue which that emphasizes the need for seamless integration of testing tools within the existing development workflow. Finally, one participant offered an ambiguous response, failing to clearly indicate their stance on AT usage.
**Answer to RQ2:** Through interviews with the experiment participants, we identified 9 benefits and 11 drawbacks associated with Augmented Testing (AT). We further categorized these drawbacks based on their origins: those inherent to AT itself were classified “essential”, while those stemming from the tool implementing AT were classified as “accidental”. Interestingly, most participants did not identify any factors that would prevent the industrial usage of AT. Instead, the majority expressed willingness to either use or explore AT in their testing procedures. We summarized our findings addressing RQ2 in Table 4.6.
4.4 Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Question/Answer</th>
<th>Benefits of using AT for manual GUI-based regression testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Efficient (faster) test execution</td>
<td>P1, P2, P3, P6, P10, P11, P12</td>
</tr>
<tr>
<td>B2</td>
<td>Visual cues indicating next action</td>
<td>P3, P7, P8, P10, P11, P12</td>
</tr>
<tr>
<td>B3</td>
<td>Elimination of typos</td>
<td>P2, P7</td>
</tr>
<tr>
<td>B4</td>
<td>Monitor test execution</td>
<td>P4, P7</td>
</tr>
<tr>
<td>B5</td>
<td>Reduced effort for non-primary actions</td>
<td>P11</td>
</tr>
<tr>
<td>B6</td>
<td>Helps not to forget steps or checks</td>
<td>P11, P12</td>
</tr>
<tr>
<td>B7</td>
<td>Test case reusability</td>
<td>P1</td>
</tr>
<tr>
<td>B8</td>
<td>Usable for non-experienced testers</td>
<td>P12</td>
</tr>
<tr>
<td>B9</td>
<td>Preferable to textual step descriptions</td>
<td>P13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Question/Answer</th>
<th>Drawbacks of using AT for manual GUI-based regression testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Neglecting non-augmented widgets</td>
<td>P1, P2, P10</td>
</tr>
<tr>
<td>D2</td>
<td>Issues of augmentation purpose comprehension and distinguishability</td>
<td>P5, P6, P7, P13</td>
</tr>
<tr>
<td>D3</td>
<td>Invisibility of off-screen augmentations</td>
<td>P1, P11</td>
</tr>
<tr>
<td>D4</td>
<td>Color visibility issues</td>
<td>P1</td>
</tr>
<tr>
<td>D5</td>
<td>Test case goal can not be derived purely from augmentations</td>
<td>P13</td>
</tr>
<tr>
<td>D6</td>
<td>Non-intuitive SUT interaction</td>
<td>P3, P4, P8, P9</td>
</tr>
<tr>
<td>D7</td>
<td>Input values limitations</td>
<td>P12</td>
</tr>
<tr>
<td>D8</td>
<td>Utility limited to small tests</td>
<td>P8</td>
</tr>
<tr>
<td>D9</td>
<td>Unclear mistake correction process</td>
<td>P3</td>
</tr>
<tr>
<td>D10</td>
<td>High tool reliance</td>
<td>P5, P11</td>
</tr>
<tr>
<td>D11</td>
<td>Overlays cover large screen area</td>
<td>P6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Question/Answer</th>
<th>Aspect that would prevent the use of AT in an industrial context</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>No or Not really</td>
<td>P2, P3, P4, P6, P8, P10</td>
</tr>
<tr>
<td>-</td>
<td>GDPR concerns or sensitive data</td>
<td>P5, P11</td>
</tr>
<tr>
<td>-</td>
<td>When complex verifications like calculations are required and not just text comparison</td>
<td>P1</td>
</tr>
<tr>
<td>-</td>
<td>Usage in a fast-paced environment</td>
<td>P7</td>
</tr>
<tr>
<td>-</td>
<td>If the tool for AT is very expensive</td>
<td>P11</td>
</tr>
<tr>
<td>-</td>
<td>Lack of documentation of test cases</td>
<td>P12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Question/Answer</th>
<th>Would you use AT for manual GUI-based regression testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Yes or yes I would try it</td>
<td>P1, P2, P3, P4, P6, P8, P9, P10, P11, P13</td>
</tr>
<tr>
<td>-</td>
<td>Yes, for complicated manual test cases</td>
<td>P5</td>
</tr>
<tr>
<td>-</td>
<td>No</td>
<td>P7</td>
</tr>
<tr>
<td>-</td>
<td>Unclear answer</td>
<td>P12</td>
</tr>
</tbody>
</table>

Table 4.6: Feedback regarding AT’s benefits and drawbacks from interview questions
4.4.3 RQ3 - Usability issues

The following section presents our findings on the usability issues of Augmented Testing (AT), which were identified through participant observations during the experiment session. This analysis is important to understand usability concerns and limitations of AT that participants may not have directly reported during the interviews. Additionally, it can further identify potential areas for improvement.

The process of identifying usability issues in a software product with appropriate users is known as usability testing, which aims to address the identified issues [170, 171]. In our study, we focused primarily on the usability issues arising from the testing approach itself, while also considering secondary usability issues related to the used tool, i.e., Scout. Similar to the reported drawbacks, we categorize usability issues as essential or accidental.

One significant usability metric is a user’s ability to complete a given task fully. To ensure participants could accomplish the test cases and consequently assess AT’s efficiency, we provided a textual description of each test case detailing all necessary steps. This description serves as a backup during AT. Thus, the number of participants who would complete the test cases solely with the aid of AT remains undetermined. Nevertheless, two participants successfully performed the test cases using AT without referring to the textual test case description. Other participants utilized the textual description to cross-verify individual steps of the test case.

**Unclear how to verify non-existing items.** A recurrent issue observed across six instances was participants’ uncertainty in verifying the absence of an item, such as after deleting a victual or recipe. During AT, participants had difficulty determining whether an augmentation was expected to denote the absent item or confirm its deletion. Some participants resorted to using the search field to confirm the proper deletion of the item. We consider this to be an essential issue with AT, as augmentations are always associated with widgets and cannot be set to represent something that does not exist in the form of a widget, such as a missing entry in a list of items.

**Unclear if the test case is completed.** The issue of verifying non-existing items raises the additional problem of uncertainty regarding the completion of a test case. As the deletion of an item serves as the final step in certain test cases to clean up test artifacts, participants experienced uncertainty regarding the completion status of the test case. We noted this occurrence in four instances during AT testing and in one instance during pure manual GUI-based regression testing. Distinct from the issue of uncertainty in verifying non-existing items,
we classify this issue as accidental. For instance, an augmentation could display the overall test coverage to indicate the completion of a test case. Such a test coverage augmentation is already implemented in Scout but was disabled during the experiment to prevent the introduction of confounding factors. Another potential mitigation strategy is to include a system logout or visiting the initial view as the final step in the test case. By incorporating this step, participants can have a clear indication that the test case has been successfully completed.

**Unclear how to proceed with the test.** In our analysis, we encountered four instances where participants expressed uncertainty regarding the continuation of the test. Notably, one of these instances occurred during AT. In this particular case, an augmentation was intended to guide the participant toward visiting the shopping list view, yet it went unnoticed. One potential explanation for this incident is visibility issues when conducting this particular session online via Zoom. The participant mentioned facing challenges due to the small screen size on his/her device, making some widgets difficult to see. Other potential explanations are the reported accidental drawbacks of D2 “Issues of augmentation purpose comprehension and distinguishability” and D4 “Color visibility issues”. Considering the absence of a definitive signal indicating a fundamental usability issue with AT, as well as its resemblance to the mentioned accidental drawbacks, we classify this issue as accidental in nature.

**Not intuitive interaction with SUT.** The previously reported drawback, D6 “Non-intuitive SUT interaction”, despite being explicitly reported by four participants, manifested across almost all test case executions by the entire participant group. This happened frequently, even after familiarizing the participants with the SUT during the warm-up phase. In most instances, this occurrence merely constituted an inconvenience and did not significantly hinder test execution. However, complications arose when multiple text input widgets were adjacent, and the tool could potentially direct the text input to the incorrect widget. For instance, on the login view, where the email address and password input widgets are located next to each other, some participants mistakenly click on the email address widget, input the email address, and then click on the password field, resulting in the email address being sent to the password widget. It is important to note that this scenario exclusively occurs during manual testing and not during AT, as the input values are auto-filled through the augmentations. We consider this issue as an observation of the reported accidental drawback F16, consequently classifying it as an accidental usability issue.
**Answer to RQ3:** Among the four identified usability issues, only one is fundamental to AT and not specific to the tool implementing AT: the verification of non-existing items. This particular usability issue is likely not only a limitation of AT but also applicable to other approaches that rely on augmentations or visual cues.

### 4.5 Discussion

Our study affirms the increased efficiency of AT found in prior research while also providing new insights into its benefits, drawbacks, and usability issues. The motivation for this work stems from a need to investigate AT in the context of manual GUI-based regression testing, as other studies focus on exploratory testing scenarios.

**Overlap with other studies.** We showed that AT has a statistically significant positive effect on the efficiency of manual GUI-based regression testing. A positive effect through additional support during the testing process may not be surprising but provides an empirical contribution to the body of knowledge and complements other findings related to AT. For instance, Nass et al. [66] observed a boost in efficiency when employing AT with Scout for the creation of test cases. For exploratory testing, Liu et al. [59] demonstrated an improvement in efficiency when utilizing a test approach for Android applications that fall into the definition of AT.

Despite the increased efficiency, there’s a potential risk of reduced testing rigor, possibly causing testers to operate in an “autopilot“ mode. Reported drawbacks, specifically D2 and D5, suggest this risk, which has been unaddressed in prior literature.

These improvements across diverse testing domains strengthen the confidence in a generally positive impact of AT on testing efficiency. Notably, this study distinguishes itself from related work in several aspects: it incorporates usability issues based on observations made during testing sessions; focuses on web applications as opposed to Android applications; rather than sourcing participants from a crowd of unknown individuals, it engages testers from established companies.

Benefits reported in this study, such as B1 “Efficient (faster) test execution”, B4 “Monitor test execution”, B7 “Test case reusability”, B8 “Usable for non-experienced testers”, and the drawback D5 “Test case goal can not be derived purely from augmentations” confirm reported benefits and drawbacks of previous studies [28, 59, 66].
4.5 Discussion

Unique findings of our study. Unique findings from our study include the usability issues related to the visual representation of augmentations for GUI-based testing, which were not reported in previous papers. The visualization techniques for augmentations presented by Liu et al. [59] and Chen et al. [28], employing colored rectangles around widgets, share similarities with our way of visualizing augmentations. However, our interviews and observations revealed that this representation of augmentations can cause confusion when augmentations serve multiple purposes, such as checks and actions. Additionally, drawbacks D3 “Invisibility of off-screen augmentations” and D4 “Color visibility issues” emphasize the importance of designing augmentations in a manner that ensures clarity, distinguishability, and usability for individuals with visual impairments. Investigating a good design of augmentations to mitigate its usability issues could improve the quality of AT’s tooling. Studies indicate that usability issues can directly or indirectly affect software quality [158].

In addressing essential drawbacks or usability issues of AT, none of the literature in our related work section included the limitation of AT in verifying non-existing items. We hypothesize that this limitation might have been overlooked in other studies, especially those involving exploratory testing setups. Although this issue was not identified as a drawback by participants and emerged from our observations, we deem this concern as an essential drawback that cannot be straightforwardly addressed via enhancements to the tooling system. In our test cases, this drawback was identified during the deletion of items (victuals and recipes). This scenario is not exclusive to our SUT and is likely to be encountered in numerous web applications. For test cases requiring the assertion of item deletion, this limitation implies that the test may either be infeasible or demand an indirect approach to assert the item’s absence. For instance, an item counter could be employed as a proxy to assert a missing item. In contrast, script-based GUI-based testing approaches can assert the absence of a widget based on its properties.

Test case size. An intriguing aspect to explore is whether the utility of AT varies with the size of test cases. Our study revealed contrasting reports among participants; one participant noted he/she would utilize AT primarily for smaller and predictable test cases, whereas another participant highlighted the utilization for larger test cases. This disparity prompts an investigation into whether AT is equally suited for both smaller and larger test cases. We found a trend of improved efficiency in larger test cases by analyzing the measurements using Bayesian data analysis. Notably, this analysis adjusts for a potential learning effect, considering that larger test cases are executed after small test cases.
Deviation from planned test steps. We observed deviations from the planned test steps during our experiment, with 21 in the manual treatment and 11 in the AT treatment. One notable example is the deletion of recipes, which can be accomplished either by clicking on the delete icon when the list of all recipes is displayed or within the detailed view of a specific recipe. Participants often used the delete icon they first encountered during the manual treatment. Another instance involved the creation of a new recipe, where participants overlooked adding a description for the recipe. Interestingly, participants also deviated from the planned steps during AT. For example, in some cases, during the execution of TC1, participants additionally visited the victual or recipe view to confirm the success of the login before proceeding to log out. In other cases, the deviation occurred accidentally by clicking slightly outside the augmentation on an adjacent widget.

We can interpret these deviations in two ways. On one hand, augmentations at the widget level can reduce deviations, particularly accidental ones, fostering more deterministic tests. On the other hand, AT does not entirely inhibit deviations, thereby enabling a flexible non-deterministic test execution. As noted by Haas et al. [52], non-exploratory manual testers often deliberately underspecify non-exploratory test cases to retain flexible execution paths and cover entire equivalence classes of test cases. Should this flexibility be preserved, a clear route back to the guided path through SUT is required. We observed instances of participant confusion when augmentations disappeared when deviating from the planned test steps, underscoring the need for clear return pathways in AT tools.

Privacy-sensitive data For aspects that would prevent the usage of AT in an industrial context, two participants reported concerns regarding the handling of privacy-sensitive data during test case processing. In our AT implementation, all data processing and storage occur solely on the device, without reliance on external cloud services. Although this is currently not an issue, it is imperative to consider the handling of privacy-sensitive data when expanding the capabilities of AT in future research and development endeavors.

Essential vs. accidental benefits and drawbacks. We distinguish between essential and accidental drawbacks and usability issues, with the majority falling into the accidental category. Of the three essential drawbacks identified, two are inherent to any tool-supported testing approach, while the one concerning the verification of non-existent elements is specifically linked to AT. Additional research is necessary to pinpoint contexts in which this drawback might impose a significant limitation and render AT inapplicable, particularly in various industry settings.
4.6 Threats to validity

However, AT’s identified accidental drawbacks and usability issues that can be mitigated through improved tooling. Nass et al. [66] also highlighted that most perceived drawbacks were linked to the limitations of the tool implementing AT. Therefore, further development of tools presents a significant opportunity to overcome the majority of drawbacks, maximizing the potential benefits of AT.

4.6 Threats to validity

We discuss the threats to validity based on the checklist presented in the guidelines by Wohlin et al. [88].

**Conclusion validity.** Conclusion validity concerns the relationship between the treatment and the observed outcomes. The small sample size of 13 could cause a threat to validity, particularly if the effect size of Augmented Testing (AT) is small. As stated in Section 4.3.1, we decided to have professional testers from the industry as participants. With their experience in testing, they do not require additional training to perform testing tasks. However, the decision to have professional testers as participants limits the number of participants due to the associated costs for the companies. While the limited number of participants has an impact on the conclusion, the collected data is relevant to understand the novel testing approach. To mitigate human errors in measuring, all measurements of test duration were consistently determined using video recordings of the sessions. Furthermore, the recordings allowed a detailed re-analysis of each session for the observations of usability issues.

**Internal validity.** Internal validity concerns to which extent the causal conclusion based on a study and the extracted data is assured. Participants performed tasks under both treatments, thereby serving as their own control group. Repeated measures for both treatments with the same participants require fewer people and account for differences between participants but do not account for order effects [86]. To mitigate learning effects, participants were randomly assigned to treatment orders. Nonetheless, the fixed sequence of test case execution, regardless of treatment, may introduce a learning effect. For instance, test cases performed last might appear to be more efficient than those performed first due to more practice with the SUT and tool. In our BDA model, we incorporated a variable to account for the learning effect during analysis. There is also a general threat through the “learning by practice” effect, wherein participants exhibited performance improvements over time for a given task due to their accumulated practical experience. Test case complexity was
evenly distributed between the two halves of all test cases, ensuring comparable levels of difficulty and average time spent per treatment. None of the participants had prior experience with the SUT, eliminating any advantage through familiarity with the SUT one participant might have over others. While the same testing environment was used for all experimental sessions, it is important to acknowledge that online participation introduced certain confounding factors, such as unpredictable delays, variations in the perceived screen sizes, and potential video compression artifacts. These factors could hinder task performance and increase time spent, but their equal impact on both treatments mitigates the issue. As an incentive to participate in our study, we highlighted the contribution to research and promised participants access to the testing tool afterward. We did not offer any monetary compensation for participation.

**External validity.** External validity concerns to which extent the results can be generalized to industrial practice. As with all empirical studies, it is essential to replicate our experiment in various environments and contexts before generalizing the results. Specifically, conducting the experiment with a different SUT and test cases, ideally from an industry software project, would strengthen the robustness of our findings. We argue that our selected SUT possesses real-world characteristics since it is in active use and its size exceeds the scale of those in most empirical studies [48]. Whereas the test cases created by the researcher might be tailored to specific functionalities or scenarios of the SUT, introducing a bias. Despite the real-world characteristics, our SUT may not represent an industrial software system or encompass the full complexity and diversity found in different software domains.

Another potential external validity threat lies in the interaction with the SUT through Scout in the non-treatment scenario. On the one hand, this minimizes confounding factors that could arise from using different tools, such as Scout versus a web browser. However, on the other hand, participants may make input errors when providing textual values (human errors) due to the distinct nature of interacting with the SUT, which they would not encounter when using a web browser. While the warm-up phase aimed to familiarize participants with this different mode of interaction, we still observed such mistakes during the actual experiment run, thus presenting a threat to the generalizability to normal web browsers.

Finally, the results of this study are a contribution to the existing body of knowledge of empirical GUI-based testing.

**Construction validity.** Construction validity concerns if the measures in a study represent the constructs in the real world and it is investigated according to the research questions. We quantified the duration for each test case
in seconds, drawing directly from the recordings of the experimental sessions. Therefore, these data constitute direct measurements that are not derived from other variables, making them a reliable indicator of the test approach’s efficiency. The data gathered on benefits and drawbacks are inherently subjective, encompassing individual interpretations and may vary among participants. To disclose the reliability of our findings, we present the number of participants who reported specific benefits and drawbacks, allowing data triangulation. Additionally, we present observations of participants and highlight instances where these observations support reported benefits or drawbacks. Nonetheless, it is important to recognize that our study might be prone to mono-operation bias due to the exclusive use of one SUT for both treatments. To mitigate this bias, we try to ensure diversity in our analysis by covering different feature sets of our SUT. Moreover, we include participants from six companies to provide a broader perspective and minimize the influence of any single organization’s practices.

4.7 Conclusion

Manual testing constitutes a substantial portion of overall testing practices in industry [44, 81], despite its labor-intensive and error-prone nature, requiring a blend of technical expertise and domain knowledge, consequently incurring significant costs [51, 150]. Augmented Testing (AT), a novel approach for GUI-based testing, shows promise in aiding manual testing [65, 66, 67].

While previous studies have either focused on exploratory GUI-based testing or compared AT against automated approaches, our research uniquely investigates the utility of AT within the context of manual GUI-based regression testing.

In this study, we evaluated the efficiency of AT compared to manual GUI-based regression testing. We conducted an experiment involving 13 industry participants across six companies who performed AT compared to manual GUI-based regression testing.

Employing BDA, our study shows that AT is more efficient in 70% of all cases on average. AT reduced the time spent by 36% when comparing the means of the total duration to perform all tests.

Additionally, we collected feedback on the benefits and drawbacks of AT and observed usability issues that participants did not report during interviews. Through data analysis, we categorized the reported benefits and drawbacks as essential or accidental, with most of the drawbacks falling into the accidental
category, thus potentially mitigable through improved tooling. Interestingly, most participants expressed a willingness to use AT in an industrial context.

For future research, we propose exploring different methods and designs for presenting augmentations, focusing on ease of understanding and differentiation to mitigate the identified drawbacks. Further empirical studies in an industrial context are required to address the industry’s current needs and address AT’s challenges.

In summary, this work provides an important contribution by providing empirical evidence for Augmented Testing (AT), a novel testing approach, in the context of manual GUI-based regression testing.

**Data availability**

The datasets generated during and/or analysed during the current study are available at the following URL: https://doi.org/10.5281/zenodo.8328166 [146].
References


[186] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International


Context: Contemporary software development is a socio-technical activity requiring extensive collaboration among individuals with diverse expertise. Software testing is an integral part of software development that also depends on various expertise. GUI-based testing allows assessing a system’s GUI and its behavior through its graphical user interface. Collaborative practices in software development, like code reviews, not only improve software quality but also promote knowledge exchange within teams. Similar benefits could be extended to other areas of software engineering, such as GUI-based testing. However, collaborative practices for GUI-based testing necessitate a unique approach since general software development practices, perceivably, cannot be directly transferred to software testing.

Goal: This thesis contributes towards a tool-supported approach enabling collaborative GUI-based testing. Our distinct goals are (1) to identify processes and guidelines to enable collaboration on GUI-based testing artifacts and (2) to operationalize tool support to aid this collaboration.

Method: We conducted a systematic literature review identifying code review guidelines for GUI-based testing. Further, we conducted a controlled experiment to assess the efficiency and potential usability issues of Augmented Testing.

Results: We provided guidelines for reviewing GUI-based testing artifacts, which aid contributors and reviewers during code reviews. We further provide empirical evidence that Augmented Testing is not only an efficient approach to GUI-based testing but also usable for non-technical users, making it a promising subject for further research in collaborative GUI-based testing.

Conclusion: Code review guidelines aid collaboration through discussions, and a suitable testing approach can serve as a platform to operationalize collaboration. Collaborative GUI-based testing has the potential to improve the efficiency and effectiveness of such testing.