Code Review of GUI-based Tests

Andreas Bauer

Department of Software Engineering
Blekinge Institute of Technology

Doctoral Dissertation Series no. 2025:14

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:14
ISSN: 1653-2090
ISBN: 978-91-7295-515-8

Code Review of GUI-based Tests

Andreas Bauer

Doctoral Dissertation

for the degree of Doctor of Philosophy at Blekinge Institute of Technology, to be publicly defended on 2025-11-24 at 09:00 in room J1630, Valhallavägen 1, 37 140 Karlskrona, Sweden

Supervisors	Emil Alégroth, Blekinge Institute of Technology, Sweden Mikael Svahnberg, Blekinge Institute of Technology, Sweden Tony Gorschek, Blekinge Institute of Technology, Sweden
Faculty Opponent	Alberto Bacchelli, University of Zürich, Switzerland
Grading Committee	Mika Mäntylä, University of Helsinki, Finland Nicole Novielli, University of Bari, Italy Sebastian Baltes, Heidelberg University, Germany

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:14

Code Review of GUI-based Tests

Andreas Bauer

Doctoral Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
Sweden

Blekinge Institute of Technology, Sweden
Department of Software Engineering

Doctoral Dissertation Series No. 2025:14

ISSN: 1653-2090
ISBN: 978-91-7295-515-8
urn:nbn:se:bth-28772

Printed in Sweden by Media-Tryck, Lund University, Lund 2025.

“All things are difficult before they are easy”

—Thomas Fuller

Abstract

Background: Modern software systems are large and complex, requiring collaboration among developers with diverse skills to manage this complexity. Code review is an essential collaborative software engineering practice in which changes are discussed before they are integrated into the codebase, enhancing code quality and promoting knowledge sharing. GUI-based testing is a technique that verifies and validates a system's behavior through its GUI by simulating user interactions. Like production code, it requires collaboration, as tests are created, reviewed, and maintained alongside production code. Code review practices for tests differ from those for production code. Omitting reviews of tests can lower their quality and increase maintenance costs. However, practices for reviewing GUI-based tests are not well understood, as academic literature mainly focuses on production code and low-level tests.

Objective: We aim to advance the understanding and practice of code review of GUI-based tests by (a) identifying code review guidelines; (b) investigating the specific practices, challenges, and information needs; (c) finding empirical evidence supporting the proposed guidelines; and (d) providing an outlook on how code review may evolve in the future.

Methods: First, we conduct a literature review of white and gray literature to identify guidelines for source and test code, and synthesize them for GUI-based tests. Next, we perform qualitative interviews with software testing professionals to identify practices, challenges, and information needs when reviewing GUI-based tests. To find empirical evidence for the proposed guidelines, we mine open-source software repositories. Finally, we conduct a questionnaire survey to gather practitioners' expectations about the future importance of code reviews.

Results: We synthesized 33 guidelines for GUI-based tests from literature sources. In analyzing code review comments from open-source repositories, we found empirical evidence supporting 25 out of the 33 proposed guidelines. Practitioners acknowledge the importance of code reviews, but lack defined practices for reviewing GUI-based tests. We identified four practices, six challenges, and four information needs related to reviewing GUI-based tests. The survey results indicate that code review will remain an essential practice with an anticipated increase in code review activities, including those for GUI-based tests.

Conclusion: This thesis advances the understanding and practice of code review for GUI-based tests to improve both review effectiveness and the quality of the tests under review. We present a set of empirically grounded guidelines derived from literature and refined through the analysis of code review comments of open-source repositories. Our research investigates current practices for reviewing GUI-based tests, highlighting the specific challenges and information needs that distinguish these reviews from those for production code. Finally, we highlight the relevance of our research for the future.

Declaration

Publications

The chapters of this cumulative thesis are based on the following four publications.

Chapter 2

Andreas Bauer, Riccardo Coppola, Emil Alégroth, and Tony Gorschek. "*Code Review Guidelines for GUI-based Testing Artifacts*". In: *Information and Software Technology* 163 (November 2023), p. 107299. DOI: 10.1016/j.infsof.2023.107299

Chapter 3

Andreas Bauer, Tomas Helmfridsson, Emil Alégroth, and Georg-Daniel Schwarz. "*When GUI-based Testing Meets Code Reviews*". Manuscript submitted to *Software Testing, Verification and Reliability*. 2025

Chapter 4

Andreas Bauer, Florian Angermeir, Emil Alégroth, and Seth Anglert. "*The Prevalence of Code Review Guidelines for GUI-Based Testing in Open-Source*". Manuscript submitted to *Information and Software Technology*. 2025. DOI: 10.2139/ssrn.5547512

Chapter 5

Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript submitted to the *Journal of Systems and Software (New Ideas and Trends Papers)*. 2025

Contributions

Andreas Bauer is the main author of the first three publications and the second author on the fourth one, compiled in this thesis. As the main author, he took the main responsibility for conceptualization, methodology, software, formal analysis, investigation, data curation, and writing of all chapters. Although the author takes primary responsibility for the introductory chapter of this thesis (Chapter 1), it is presented in the plural form to highlight the collaborative nature of all research efforts. The contributions of the author and the co-authors to each chapter are detailed using the contributor role taxonomy *CRediT*:

Chapter 2

Andreas Bauer: Conceptualization, Methodology, Formal analysis, Investigation, Data curation, Writing – original draft, Project administration; *Riccardo Coppola:* Conceptualization, Methodology, Investigation, Data curation, Writing – original draft; *Emil Alégroth:* Conceptualization, Methodology, Writing – original draft, Writing – review and editing, Supervision; *Tony Gorschek:* Writing – review and editing, Supervision, Funding acquisition.

Chapter 3

Andreas Bauer: Conceptualization, Formal analysis, Data curation, Investigation, Methodology, Project Management, Resources, Writing – original draft; *Emil Alégroth:* Conceptualization, Methodology, Supervision, Writing – review and editing; *Tomas Helmfridsson:* Formal analysis, Resources; *Georg-Daniel Schwarz:* Formal analysis, Validation.

Chapter 4

Andreas Bauer: Conceptualization, Formal analysis, Data curation, Investigation, Methodology, Project Management, Resources, Software, Writing – original draft; *Florian Angermeir*: Formal analysis, Data curation, Investigation, Writing – original draft; *Emil Alégroth:* Conceptualization, Methodology, Supervision, Writing – review and editing; *Seth Anglert:* Data curation, Resources.

Chapter 5

Michael Dorner: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization, Supervision, Project administration; *Andreas Bauer:* Software, Validation, Data curation, Writing – review & editing, Visualization; *Darja Šmite:* Conceptualization, Writing - original draft, Writing - review & editing; *Lukas Thode:* Writing - original draft, Writing - review & editing; *Daniel Mendez:* Conceptualization, Writing - review & editing; *Ricardo Britto:* Data curation, Writing - review & editing; *Stephan Lukasczyk:* Data curation, Writing - review & editing; *Ehsan Zabardast:* Data curation, Writing - review & editing; *Michael Kormann:* Validation, Writing - review & editing.

Other publications not part of this thesis

Georg-Daniel Schwarz, Andreas Bauer, Dirk Riehle, and Nikolay Harutyunyan. "*A Taxonomy of Microservice Integration Techniques*". In: *Information and Software Technology* 183 (July 2025), p. 107723. DOI: 10.1016/j.infsof.2025.107723

Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Fabiola Moyón C., Daniel Mendez, and Tony Gorschek. "*Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies*". Accepted at *International Conference on Software Engineering (ICSE '26)*. 2025

Michael Dorner, Andreas Bauer, and Florian Angermeir. "*No Free Lunch: Research Software Testing in Teaching*". Under revision at *Journal of Open Research Software*. 2024

Andreas Bauer, Julian Frattini, and Emil Alégroth. "*Augmented Testing to Support Manual GUI-based Regression Testing: An Empirical Study*". In: *Empirical Software Engineering* 29.6 (August 2024), p. 140. DOI: 10.1007/s10664-024-10522-z

Julian Frattini, Jannik Fischbach, and Andreas Bauer. "*CiRA: An Open-Source Python Package for Automated Generation of Test Case Descriptions from Natural Language Requirements*". In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. Hannover, Germany: IEEE, September 2023, pp. 68–71. DOI: 10.1109/REW57809.2023.00019

Andreas Bauer and Emil Alégroth. "*We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing*". In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Dublin, Ireland: IEEE, April 2023, pp. 1–9. DOI: 10.1109/ICSTW58534.2023.00015

Andreas Bauer, Nikolay Harutyunyan, Dirk Riehle, and Georg-Daniel Schwarz. "*Challenges of Tracking and Documenting Open Source Dependencies in Products: A Case Study*". In: *Open Source Systems*. Ed. by Vladimir Ivanov, Artem Kruglov, Sergey Masyagin, Alberto Sillitti, and Giancarlo Succi. Vol. 582. Cham: Springer International Publishing, 2020, pp. 25–35. DOI: 10.1007/978-3-030-47240-5_3

Nikolay Harutyunyan, Andreas Bauer, and Dirk Riehle. "*Industry Requirements for FLOSS Governance Tools to Facilitate the Use of Open Source Software in Commercial Products*". In: *Journal of Systems and Software* 158 (December 2019), p. 110390. DOI: 10.1016/j.jss.2019.08.001

Funding

This research was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

Open Science

We follow the principles of open science to ensure transparency and reproducibility. All publications are made freely accessible without paywalls (*open access*). Research data are shared in reusable, editable, and interoperable formats, enabling secondary analysis, replication, and integration with future studies (*open data*).

All data related to human participants, including interview transcripts and company associations, were anonymized to ensure confidentiality and protect the privacy of individuals involved in the study. Sensitive information from industry partners was managed under confidentiality agreements. Research materials were only released with explicit consent from the participants. These measures strike a balance between transparency and the ethical responsibility to protect privacy and trust.

We upload our research materials to Zenodo[1] for permanent archiving, ensuring their availability with a digital object identifier (DOI) for reference.

By committing to open science, we enable independent validation, support cumulative knowledge building, and foster collaboration across the software engineering community. Table 1 provides a summary of our open science initiatives for each chapter.

Table 1: Summary of our open science initiatives for each chapter of this thesis

Chapter	Study	Open Access	Open Data
Chapter 2	[32]	yes	https://doi.org/10.5281/zenodo.7248201
Chapter 3	[35]	yes	https://doi.org/10.5281/zenodo.17158220
Chapter 4	[31]	yes	https://doi.org/10.5281/zenodo.17104677
Chapter 5	[69]	yes	https://doi.org/10.5281/zenodo.17359762

[1] https://zenodo.org

Acknowledgements

This thesis would not have been possible without the support, guidance, and encouragement of many individuals I am deeply grateful to.

First and foremost, I would like to express my sincere gratitude to my supervisor, Emil Alégroth, for his invaluable expertise, insightful guidance, and continuous support throughout this work. His feedback and encouragement have been instrumental to the completion of this thesis. I am also thankful to my co-supervisors, Mikael Svahnberg and Tony Gorschek, for their support.

Furthermore, I am grateful for the many wonderful people I have had the pleasure to work with. I am not only appreciative of their invaluable support but also of the many enjoyable moments we shared, which made this journey far more rewarding. In particular, I am thankful for my fellow Ph.D. colleagues Michael Dorner, Waleed Abdeen, Ehsan Zabardast, Julian Frattini, Felix Jedrzejewski, Lukas Thode, Florian Angermeir, Umar Iftikhar, and Michel Nass, the senior researchers Daniel Méndez and Davide Fucci, and my ever-helpful colleagues Svetlana Živanović, Anna Eriksson, and Monique Johansson.

The Department of Software Engineering at Blekinge Institute of Technology (BTH), led by Michael Mattsson, and the SERT research project, led by Tony Gorschek, provided a supportive research environment that made this work possible. Furthermore, I owe thanks to the company partners of the SERT project and the individuals who collaborated with us, providing valuable contributions to this research. I also want to acknowledge my collaboration with Riccardo Coppola and Tomas Helmfridsson.

From the very beginning of my academic journey, I am grateful to have worked with and become friends with Georg-Daniel Schwarz and Nikolay Harutyunyan.

Outside of my academic life, I thank Anja and Ráhel for making life in Karlskrona more enjoyable.

Lastly, I would like to thank my family for their support.

Andreas Bauer
Karlskrona, October 2025

Contents

1

Introduction

1

1.1 Motivation

Modern software systems are large and fast-evolving. No single developer can oversee every concern or anticipate the effects of each change. Effective development, therefore, depends on collaboration among professionals with complementary technical skills and domain expertise [73, 140, 166]. In other words, *"any software project with more than one person is created through a process of collaborative software engineering"* [163]. Such collaboration spans over all phases of the software development lifecycle: requirements, architecture, implementation, and maintenance [163]. During the implementation of source and test code, code review has become a central mechanism for collaboration and quality assurance [42, 190]. Code review is a software engineering practice where other developers or testers discuss code contributions before changes (additions, modifications, or deletions) are integrated into the codebase [20, 53]. This practice helps to improve the quality of artifacts, decrease the likelihood of post-release defects, promote consistent coding practices, and enable knowledge transfer [20, 27, 61, 83, 109, 156]. Additionally, code reviews offer socio-technical benefits, such as providing justification for changes, keeping the team informed about the project's evolution, and providing a record of design decisions. The adoption of code review can be seen in open-source software [178] as well as in leading software companies such as Microsoft [20] and Google [190].

Software testing is a systematic process for the verification and validation of software against its specifications [164, 204, 221]. Verification[1] ensures that the software meets its defined requirements, while validation[2] ensures the software aligns with the customer's expectations. It involves executing software with defined inputs and assessing the resulting outputs against expected outcomes. As software systems grow in complexity and scale, testing has become essential for maintaining reliability and enhancing confidence in evolving software [204]. Automated testing plays a crucial role in addressing the scalability challenge, enabling frequent and consistent verification throughout the development process [181]. As a crucial part of the software development process, software testing is the most widely used method for ensuring software quality in the industry [216]. Testing can account for half of the development budget, making it also an expensive part of software development [9].

Testing for the verification of a system can be divided by the level of abstraction of the tests: unit, integration, and system testing [204, 221]. Unit testing represents the lowest level of abstraction and verifies the behavior of a "unit" of code in isolation, such as a function or class [48, 174, 221]. These tests are typically performed as automated tests. Integration testing verifies the interactions among elements of the system, like components, modules, or subsystems [221]. System testing verifies the behavior of the system as a whole [221], usually through a scenario-based execution [189]. System

[1] Verification: *"Are we building the product right?"*[40]
[2] Validation: *"Are we building the right product?"* [40]

tests may be performed manually or as automated tests, targeting either the system's application programming interface (API) or its graphical user interface (GUI).

In the metaphor of the testing pyramid [54], unit tests form the base of testing efforts. They are numerous, fast to execute, and cover small, isolated functionalities. Developers typically create these tests alongside production code, making them the primary focus of testing efforts during development. System-level tests occupy the top of the pyramid. They are few, resource-intensive, and developed later in the lifecycle. Only a subset of system tests target the GUI, yet these GUI-based tests are crucial for verifying and validating conformance to specifications and user expectations [8]. We argue that, due to their high cost and limited number, ensuring the quality of those GUI-based tests that are created is essential, which requires substantial technical and domain knowledge.

GUI-based testing, a type of system testing, verifies a system's behavior through interactions with its GUI, in the same way a user would use the system [8, 55]. By exercising end-to-end user interactions with the GUI, GUI-based testing enables the assessment of both the visual appearance and functional correctness of the system under test (SUT).

Automated software tests require collaborative efforts similar to production (source) code [191], as test artifacts are created, reviewed, and maintained alongside production code. Sub-bar test results can arise from human error or a lack of technical or domain knowledge. Omitting the review of test code can lower its quality and lead to increased maintenance costs [39, 93, 138].

Existing research on code review largely targets production code and low-level tests [61, 205, 206]. There are best practices and guidelines for code review available in both open-source and industrial communities [66], indicating their importance for these groups. However, the survey conducted by Dong et al. [66] does not provide information on best practices specifically related to test code in general, or GUI-based tests in particular. Previous research has identified differences in the review processes for different artifacts, e.g., production code and test code [205]. When reviewing test code, reviewers mainly discuss code improvements, suggesting better testing practices and some generic code quality practices to ensure the maintainability and readability of test code.

Conducting effective code reviews is a complex and cognitively demanding task. Reviewers need to understand not only the quality of the specific changes but also their broader implications for the system, such as whether these changes might break other code [212]. Developers often find it difficult to comprehend code modifications during the review process, even when they are supported by tools. Another issue is the cognitive load that reviewers experience. Developers and testers, for instance, apply mental effort to understand the structures of the source code [58] or to interpret

various levels of abstraction [162]. A high cognitive load can negatively impact performance on a given task [153], thus affecting review quality.

Given the distinction between code review for production code and test code, along with the lack of established practices for reviewing GUI-based tests, and the general efforts required to perform good code review, we believe it is essential to understand how code review practices can be effectively applied to GUI-based tests. Code review is known to improve artifact quality. However, at the starting point of this research, the extent of existing support for reviewing GUI-based tests was unclear. Supporting this activity, for instance, with clear guidelines is therefore critical. Combined with the substantial resource demands of developing and maintaining such tests, this gap highlights the importance of investigating and developing review guidelines specifically for GUI-based tests to detect issues early on. Such guidelines could enhance not only the quality of the tests themselves but also that of the software systems, since their purpose is to provide assurance of the system's quality.

In this thesis, we aim to enhance our understanding of the specific aspects involved in code review of GUI-based tests and to provide tailored guidelines that will support this practice. This could improve both the effectiveness of the review process and the quality of the tests being reviewed, as it is crucial for delivering high-quality software [216].

1.2 Background

This section explains the concepts of GUI-based testing and code review used throughout this thesis.

For clarity, we use the following terminology that we adopted from Zabardast et al. [229]: *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*, *test code*, and *test scripts* interchangeably. We also use the terms *source code* and *production code* interchangeably.

1.2.1 GUI-based Testing

Graphical User Interface (GUI)-based testing verifies and validates system behavior through direct interaction with its GUI, similar to how a user would interact with a system's GUI [8, 55]. Depending on the test's purpose, GUI-based testing can assess both the visual appearance of the GUI and the functional correctness of the underlying system [8].

GUI-based testing can be classified in several ways [10]. The first classification relates to how test cases are specified. The three dominant approaches are script-based,

model-based, and capture-and-replay. *Script-based* testing defines test cases as code. Automation frameworks (e.g., Selenium [103]) provide APIs for interacting with GUI elements, enabling integration with continuous integration pipelines. *Model-based* testing (MBT) utilizes explicit models that represent the expected behaviors of a SUT and its surrounding environment [217]. These models are more abstract and simpler than the SUT itself. These models may be visual (e.g., state machines defined as nodes and edges), textual (as in behavior-driven development), or formal (e.g., mathematical specifications). *Capture-and-replay (C&R)* testing involves recording user interactions with the GUI and then replaying those interactions as executable tests [144]. Creating test cases is very easy and does not require advanced testing skills. However, this approach can be fragile, as even minor changes to the interface may render the recorded sequences invalid.

Another classification, described by Alégroth, Feldt, and Ryrholm [10], distinguishes automated GUI-based testing generations by their interaction mechanism with the system under test (SUT). First-generation (coordinate-based) testing identifies GUI elements by screen coordinates, typically recorded through capture-and-replay tools. Relaying on fixed coordinates is fragile, as it fails under layout changes. Second-generation (component-based) testing uses tools that interact with the GUI using the layout model or properties of the application. This interaction often involves accessing an application's Document Object Model (DOM) in web applications or other GUI interfaces. Third-generation (image-based) testing leverages computer vision to recognize GUI elements by their visual appearance [46]. While more resilient to structural changes, it is computationally demanding and sensitive to rendering variations.

GUI-based testing applies across different platforms that provide a GUI, including web applications, mobile applications [56, 57], and desktop applications with native GUIs [194]. Each of these environments comes with its own set of tools and ways to incorporate a system's environment. For example, the Android platform provides Espresso[3], a testing framework for Android applications, which can be executed either in an emulator or on a real device. A third-generation approach is an exception, being agnostic to various platforms due to its image-based interactions with the SUT, but as described, sensitive to rendering variations.

Finally, GUI-based tests serve different testing objectives. Exploratory testing uncovers new and previously unknown faults in software, by testers utilizing their domain knowledge and technical expertise [126, 128, 180]. These tests can be conducted manually and in an ad hoc manner. Regression testing verifies that previously tested functionality remains intact after changes, typically relying on automated tests [221]. As regression tests evolve with the system being tested, qualities such as readability, maintainability, modularity, and reusability become important.

[3] https://developer.android.com/training/testing/espresso/

1

There is no universal testing technique that meets all testing needs. The choice of technique depends on the specific requirements of the project, including the complexity of the application, the expertise of the team, and the goals of the testing. In this thesis, we focus on scripted automated tests for regression testing, which represent the majority of GUI-based tests for web applications (as reflected in the testing tools used for this purpose [113]).

Test Cases A GUI test case consists of two parts: (a) an initial GUI state in which the test case is executed, and (b) a sequence of events that serve as the test input to the GUI [157]. Multiple instances of initial GUI states and sequences of interactions can be combined to form a larger test case.

Listing 1.1 presents an example test case using Playwright to test the functionality of a to-do list. The initial GUI state for this test case is achieved by navigating to a specific URL using `page.goto(...)`. In this example, no further setup is required, e.g., to ensure that a user is logged in or exists. After reaching the initial state, various actions are performed to interact with the GUI and create a new user through a form available on the interface. This code listing, although simple, presents challenges in terms of understandability and readability. The context of the GUI in its current state is not provided, making it difficult to understand each interaction and how each of them affects the GUI. As a result, to comprehend the test case, we need to either observe its execution or have domain knowledge about the SUT.

Most automated GUI-based tests are implemented as code that employs known programming constructs such as branching and looping, similar to production code. However, unlike conventional source code, these tests emulate user interactions and execute in parallel with the SUT. This imposes unique requirements on their design, as they must handle asynchronous, scenario-based actions and synchronize with system responses that involve both chronological and logical dependencies.

Testing frameworks utilize locators to identify and target elements on the GUI to emulate external inputs from users. For web-based applications, the document object model (DOM) is employed to target specific elements using HTML tags, identifiers, and CSS properties. In Listing 1.1, elements are located using `page.getByRole(...)`, which provides a loose coupling between the test code and the GUI of the SUT. Modifications made to the GUI can lead to failing test cases, necessitating maintenance efforts from testers [57]. The selection of robust locators is crucial and depends on identifiable elements present in the GUI. Additionally, system responses must either be synchronized using static delays via wait statements or handled through the functions provided by the frameworks.

Assertions are performed using the `expect(...)` to check for a specific behavior. Typically, test cases assert whether a GUI element is present, visible, or equal to a specific value. However, the context of the GUI makes it difficult to determine the

exact interactions with it without prior knowledge. For instance, in heavily nested structures of different types of GUI elements.

Even in the simple example shown in Listing 1.1, several potential issues may arise depending on the implementation of the system. When submitting a to-do item (line 13), it may be displayed directly on the GUI, but is still processed in the background. The item is then loaded asynchronously after it has been successfully processed in the backend. Thus, an immediate check (line 17) would not confirm whether the item was processed successfully. Furthermore, if many to-do items are presented in a list or other structure, the test logic becomes more complicated due to the potential for ambiguous locators. Additionally, input fields that accept free text introduce edge cases that need to be considered, such as how to handle special characters, empty strings, or excessively long inputs. In such situations, code reviews by domain experts and test automation specialists could help identify potential issues early to improve the quality of tests.

```
1  test.describe('Adding New Todos', () => {
2    test('Add Valid Todo', async ({ page }) => {
3      await page.goto('https://demo.playwright.dev/todomvc');
4
5      // 1. Click in the "What needs to be done?" input field
6      const todoInput = page.getByRole('textbox', { name: 'What needs to be done?' });
7      await todoInput.click();
8
9      // 2. Type "Buy groceries"
10     await todoInput.fill('Buy groceries');
11
12     // 3. Press Enter key
13     await todoInput.press('Enter');
14
15     // Expected Results:
16     // - Todo appears in the list with unchecked checkbox
17     await expect(page.getByText('Buy groceries')).toBeVisible();
18     const todoCheckbox = page.getByRole('checkbox', { name: 'Toggle Todo' });
19     await expect(todoCheckbox).toBeVisible();
20     await expect(todoCheckbox).not.toBeChecked();
21
22     // - Counter shows "1 item left"
23     await expect(page.getByText('1 item left')).toBeVisible();
24
25     // - Input field is cleared and ready for next entry
26     await expect(todoInput).toHaveValue('');
27     await expect(todoInput).toBeFocused();
28
29     // - Todo list controls become visible (Mark all as complete checkbox)
30     await expect(page.getByRole('checkbox', { name: '>Mark all as complete' })).
       toBeVisible();
31   });
32 });
```

Listing 1.1: Example test case from the Playwright repository, written in TypeScript, asserting that items can be added to a to-do list. Source: add-valid-todo.spec.ts

Challenges with GUI-based Test Automation Manual execution of GUI-based tests is labor-intensive, error-prone, requires both technical expertise and domain knowledge, and is associated with high costs [111, 112]. Automated testing reduces human error and increases reliability [133]. However, the execution of automated GUI-based tests is still significantly slower and more resource-intensive than lower-level tests since test execution depends not only on the underlying functionality but also on updates and rendering of the GUI [5]. Therefore, enhancing the quality of tests before they are integrated into the codebase could help optimize the impact of these resource-intensive tests.

Transitioning from manual testing to automated GUI-based testing can yield a positive return on investment (ROI) [9]. The time it takes to achieve ROI depends on the extent of prior testing efforts. However, maintenance costs associated with script-based GUI tests are a significant challenge for industrial adoption. When test maintenance is performed frequently, the maintenance costs are lower than those of big-bang test maintenance overhaul.

Nass, Alégroth, and Feldt [168] identifies 24 distinct challenges in GUI-based test automation. Among these, four are seen as essential challenges that may not be resolved through improved tools or processes. These key challenges are: application changes break test execution, test automation of dynamic applications, tool challenges, and state-space explosion in model-based testing. These challenges impact individual test steps, test cases, test suites, tools, and reviews.

The challenges that are most relevant for this thesis are the synchronization between tests and the SUT, and the robust identification of GUI elements via locators. Synchronization refers to the need for tests to wait for the SUT and its GUI to be ready to receive interactions [168]. For example, a test step (such as a button click) initiates a transition to another state (changing the SUT), and the test must synchronize, either through static or dynamic delays, before the next test step can be executed. Otherwise, the consecutive test step can fail, e.g., if the GUI elements for the next interaction are not rendered.

Robust identification of GUI elements refers to the ability of a locator to reliably target the intended GUI element for interactions or checks, even when changes to the GUI occur [168]. Modern testing frameworks offer a variety of locator methods, ranging from general string searches to specific test identifiers, enabling the identification of GUI elements based on their properties. Studies by Leotta et al. [146] and Nass et al. [170] have proposed approaches that utilize multiple locators to enhance the robustness of test cases. However, modern testing frameworks such as Playwright and Cypress, which are widely used for testing web applications, did not adopt multi-locators. Furthermore, whether single-locators or multi-locators are utilized, locators are selected by developers and testers based on their knowledge, experience, and preference.

The challenges of synchronization and robust locators are interconnected [168]. A robust locator must be synchronized to ensure that the GUI element is in the correct state before any interaction can occur. This can practically be observed in the *auto-waiting*[4] functionality of Playwright, where a simple `locator.click()` triggers checks to ensure that exactly one element is located, that it is visible, enabled, stable, and not obscured by other elements. As such, synchronization has been recognized as a critical issue by several tool vendors, and mechanisms have been built in to support most general cases. However, this doesn't prevent developers and testers from introducing their own waiting mechanisms, either due to a lack of knowledge about the inner workings of the frameworks used or to address edge cases. In both cases, this should be covered and discussed during a code review.

1.2.2 Code Review

Code review is a collaborative practice in which developers examine and discuss code changes before integration. It is among the most widely adopted practices in modern software engineering and integrates seamlessly with contemporary development workflows. The origins of code review can be traced to formal software inspections, introduced by Fagan [78]. These inspections were highly structured and rigorous, involving predefined roles, checklists, and meetings. While effective in defect detection, their complexity and cost limited their adoption in practice [198]. Contemporary code review practices have evolved to become more lightweight, informal, and supported by tools, commonly referred to as modern code review (MCR) [20, 53]. While code reviews are not tied to specific tools, they are frequently facilitated by dedicated platforms such as Gerrit or by integrated systems like GitHub and GitLab, which embed review functionality directly into version control workflows. This integration allows for asynchronous discussions, inline commenting, and iterative refinement, making the review process both scalable and collaborative. Figure 1.1 illustrates such a development workflow. In such a development workflow, changes to the codebase are usually made in separate branches. Once the changes are ready to be merged into the main branch, a review is initiated and assigned to one or more reviewers. Reviewers examine the proposed modifications, provide feedback, and may request revisions or additional context to fully understand the purpose and impact of the changes. This process often involves multiple iterations until the reviewers are satisfied. At that point, the changes are either approved and merged into the main branch or rejected if they do not meet the required standards.

Code reviews are effective in improving software quality and facilitating knowledge sharing [22, 82, 156]. Research has shown that defects identified and resolved during code reviews often enhance the software's understandability and modifiability, rather than just its visible functionality [150]. This is particularly relevant for GUI-based tests, in particular due to the associated maintenance costs, as well as for tests in general.

[4] https://playwright.dev/docs/actionability

1

Code reviews are widely adopted in both industry and open-source projects [20, 27, 61, 156].

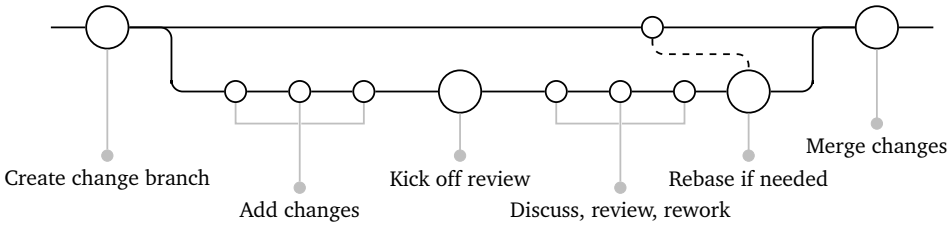

Figure 1.1: Development workflow of integrating changes with code reviews

Code Review of Test Code Spadini et al. [205] investigated whether and how developers review test code during code reviews. This study is particularly relevant to us for two main reasons. First, it reveals that developers tend to discuss test files significantly less than production files, raising the question of whether the code review of tests, and in our case, GUI-based tests, is a less valuable practice. Second, the study indicates that the process of reviewing test code differs from that of reviewing production code. This motivated our research and this thesis, which aims to investigate the distinct practices and challenges when reviewing GUI-based tests.

Findings by Spadini et al. [205] also show that when reviewers review test code, they focus on test-specific aspects, such as better testing practices, identifying tested and untested paths, and evaluating assertions within the test code. Reviewing test code requires not only understanding the context of the test but also the production code under test. Further, the absence of test-specific information in code review tools often compels developers to inspect tests locally in their integrated development environment (IDE), leveraging navigation and other features unavailable in code review tools.

In Chapter 3, we present our findings on how testers conduct code reviews of GUI-based tests, and we highlight how these findings overlap with those of Spadini et al. [205].

Best Practices Best practices in software development are essential for standardizing processes that produce high-quality products on predictable schedules [176]. In this thesis, the terms *guidelines* and *best practices* are used interchangeably, as both aim to improve the code review process.

A study by Dong et al. [66] summarizes 57 best practices and 19 pain points related to code reviews within open-source software and industrial communities. The best practices encompass how to prepare for and discuss the code under review through comments, as well as how to conduct the review process itself. These practices focus on

ensuring code quality before the review, selecting appropriate reviewers, incorporating automation, prioritizing reviews, providing clear, reasonable, actionable, and positive feedback, iteratively responding to code changes, and more. The study also identifies 19 pain points associated with code reviews, which include challenges related to tools, achieving consensus, geographical distance, and the workload involved in reviews.

One key conclusion from the study is that understanding the code context is one of the most difficult things for reviewers. Given that GUI-based tests involve a broader context, including the GUI and the overall behavior of the system, missing context information during review could become even more critical. Similar to our research objective, the authors hope that both researchers and practitioners will gain better insight into effective code review practices, facilitating more efficient code reviews.

Organizations like Google [108] and GitLab [19] provide publicly available best practices for conducting code reviews for contributing to their open-source projects. These resources outline the overall review process, how to prepare changes for review, and the principles to follow during the review, as summarized by Dong et al. [66].

In conclusion, code reviews are an effective way to improve the quality of artifacts, but their success depends on the human judgment and technical expertise of the reviewers. Unstructured or informal review practices can lead to the omission of important contextual information, which diminishes the effectiveness of the review or may even result in overlooking significant quality issues. Establishing review guidelines can help address these challenges by providing reviewers with specific evaluation criteria and authors with information to prepare artifacts for a review. This approach enhances both the efficiency and quality of the review process.

1.3 Research Objectives and Questions

The overarching objective of this research is to *advance the understanding and practice of code review of GUI-based tests to improve both the effectiveness of the review process and the quality of the tests under review*. We consider this crucial to deliver high-quality tests due to the differences between reviewing production code and test code, the lack of established practices for reviewing GUI-based tests, and the overall effort needed for effective code reviews. Based on this overarching objective, we derive four specific objectives and corresponding high-level research questions, which we present in the remainder of this section.

1.3.1 Objective 1: Identify Code Review Guidelines

We aim to identify code review guidelines from scientific literature and gray literature to synthesize them into sets for production, test, and GUI-based test code. As noted earlier, general guidelines can help make code reviews more efficient and effective,

and have been established for production code in both the industry and open-source software communities [66].

In this objective, we are focusing on guidelines to facilitate the discussion of artifacts under review, omitting aspects related to less tangible elements of the code review process, such as reviewer selection or effort allocation [49, 231]. Although further research should examine additional factors, we believe that the main challenges in reviewing GUI-based tests stem from the tests themselves rather than these other factors. Therefore, our research focuses on the tests under review.

Our analysis of literature serves two purposes. First, to identify the extent to which existing guidelines address test code in general and GUI-based tests in particular. Second, to identify a set of code review guidelines that can be mapped as perceivably applicable to GUI-based tests, even when not explicitly defined for GUI-based tests.

Code review guidelines could benefit both contributors of tests and reviewers. For contributors, they provide direction on how to structure and document code changes to facilitate comprehension. Clear explanations of a change's rationale, its expected impact, and relevant documentation reduce the reviewer's need to seek additional contextual information [178]. For reviewers, guidelines function as structured checklists to ensure systematic evaluation of contributions. They can promote consistency in reviewing aspects such as adherence to design principles, appropriate use of metrics, and correctness of assertions in test cases.

This objective is addressed through the following research question:

RQ1: What existing code review guidelines can be identified and adapted for the review of GUI-based tests?

1.3.2 Objective 2: Practices, Challenges, and Information Needs

We aim to advance understanding of the code review process for GUI-based tests by examining practitioners' specific practices, challenges, and information needs. While the previous objective focuses on identifying review guidelines, this one explores how practitioners conduct reviews of GUI-based tests in practice, what challenges they face, and what information they require to perform effective reviews.

Building on prior work by Spadini et al. [205] and Pascarella et al. [178], this investigation seeks to determine whether the code review of GUI-based tests differs from that of production or lower-level test code. We assume such differences, as there are established distinctions in how test code is reviewed compared to production code [205]. This is further reinforced by the additional requirements of GUI-based tests, which must manage both chronological and logical sequences through the use of locators and synchronization mechanisms.

This objective encompasses three interrelated aspects. For *practices*, we are interested not only in how practitioners conduct code reviews of GUI-based tests, but also in whether they view these reviews as beneficial and to what extent they are aware of and adhere to established practices. For *challenges*, we want to understand the concrete challenges reviewers encounter, clarifying whether GUI-based tests introduce unique difficulties or amplify those already known in general code review. For *information needs*, we want to understand what contextual information reviewers require to review GUI-based tests effectively. Investigating these aspects would not only advance understanding of effective code review for GUI-based tests but could also complement the guidelines (Objective 1) by addressing aspects that are absent in the literature.

This objective is addressed through the following research question:

RQ2: What practices, challenges, and information needs do testers encounter during code review of GUI-based tests?

1.3.3 Objective 3: Empirical Evidence for Guidelines

We aim to find empirical evidence supporting the relevance of the guidelines identified and mapped to GUI-based testing (Objective 1). So far, no empirical evidence has been provided to show that these guidelines are relevant and could be recommended to practitioners. We examine the extent to which code review comments on GUI-based tests correspond, either explicitly or implicitly, to the concerns represented in these guidelines. This will provide an initial evaluation of their practical validity. Explicit alignment occurs when a review comment directly addresses a concern described by a guideline, whereas implicit alignment refers to a comment that reflects an issue that a guideline could help mitigate. Identifying such alignments offers insights into how well the proposed guidelines capture the issues practitioners encounter during real-world reviews. From real-world reviews, we could provide specific examples and refine guidelines that are too ambiguous.

This objective is addressed through the following research question:

RQ3: What empirical evidence exists that supports the use and relevance of the proposed guidelines in real-world code reviews of GUI-based tests?

1.3.4 Objective 4: Anticipating the Future of Code Review

We aim to explore how practitioners perceive the current state of code review and what changes they anticipate in the near future. As discussed earlier, code review is an essential practice in collaborative software engineering. This is why we seek to improve our understanding of code review for GUI-based tests.

Code review practices have evolved from the synchronous, formal, highly structured code inspections of the 1980s [78] to asynchronous, lightweight, and tool-supported

approaches [53]. Future changes to code review are likely as the software industry and engineering continue to evolve. In particular, advances in large language models (LLMs) are beginning to influence how code reviews are conducted, potentially transforming the reviewer's role and the overall review process.

We want to understand whether practitioners anticipate code review to be more or less essential for their development process and to what extent code review shall be delegated or supported by LLM-based systems. While we believe in our approach to improve the quality of GUI-based tests by enhancing the code review process for it, we remain open to adapting it in response to technological advancements and evolving development practices. Furthermore, we will investigate whether artifacts that are not production code, such as GUI-based tests, will be subject to review in the future.

This objective is addressed through the following research question:

RQ4: What changes do practitioners anticipate for code review in the future?

1.4 Methods

Research methodologies offer a systematic and scientific way to generate new knowledge by addressing research challenges [227]. Their purpose is to maintain objectivity, rigor, and replicability in the research process, leading to findings that are credible and reliable. In software engineering, methodologies like action research and design science are commonly used [208, 223]. While a *methodology* refers to the overall research approach, a *method* relates to the specific techniques and procedures employed for data collection [4]. A single methodology can consist of multiple research methods.

To address our research questions, we conduct various empirical studies utilizing methods such as qualitative interviews, a literature review, software repository mining, and a survey. Table 1.1 summarizes the methods used in each chapter of this thesis, and Figure 1.2 visualizes this relationship.

Table 1.1: Overview of research methods employed in each chapter

Chapter	RQ	Objective	Method
Chapter 2	RQ1	Identify Code Review Guidelines	Literature Review
Chapter 3	RQ2	Practices, Challenges, and Information Needs	Qualitative Interviews
Chapter 4	RQ3	Empirical Evidence for Guidelines	Mining Software Repositories
Chapter 5	RQ4	Anticipating the Future of Code Review	Questionnaire Survey

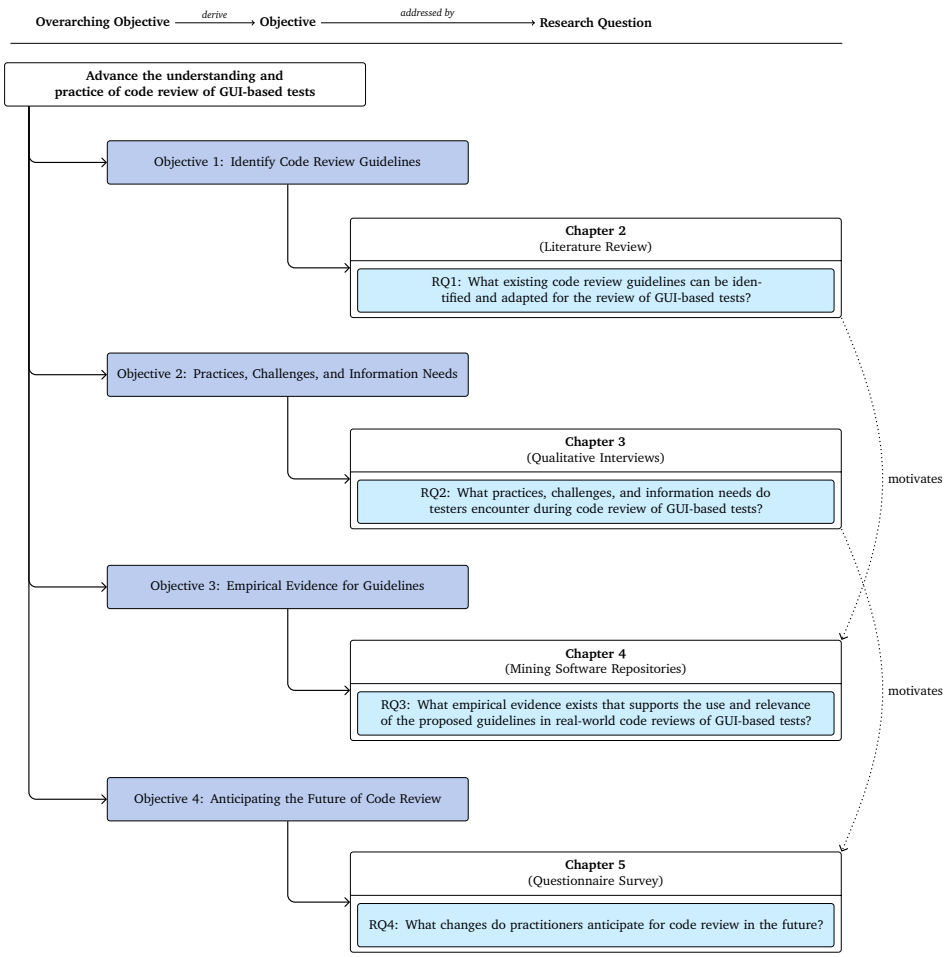

Figure 1.2: Research objectives and research questions

1

A pragmatic view on empirical software engineering research is described by Méndez Fernández and Passoth [158] and reflects a cyclic process of knowledge creation. Figure 1.3 shows an adaptation of the pragmatic research cycle. A single empirical perspective rarely provides a comprehensive understanding of complex software engineering phenomena. Therefore, an iterative approach that alternates between exploration and confirmation is required. This cyclical process integrates both inductive and deductive reasoning to build, refine, and confirm empirical knowledge. Multiple iterations and sources of evidence are essential to draw conclusions through the triangulation of methods and data.

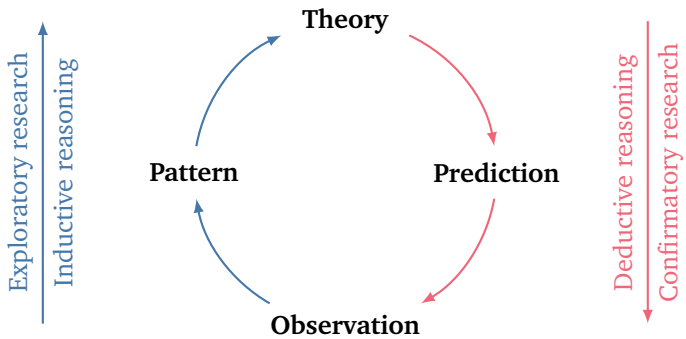

Figure 1.3: The empirical research cycle involves two main approaches: exploratory research, which generates theories using inductive reasoning (starting with observations), and confirmatory research, which tests theories through deductive reasoning (starting with a theory). Adopted from [158] and [72].

Induction describes the inference of a general rule from a particular case [158]. Inductive, or exploratory, studies are particularly valuable when limited theory exists, as they enable researchers to observe and describe emerging patterns in real-world contexts. The data collected from such studies is primarily qualitative and can be obtained through case study research or methods such as interviews, surveys, and observations [227].

In contrast, deduction applies general rules or theories to specific cases [158]. Deductive, or confirmatory, studies test hypotheses and theoretical predictions. These studies primarily collect quantitative data using a more structured design, often through (quasi-)experiments.

Across all phases of the empirical research cycle, researchers must make design decisions concerning methods, measurement, and analysis approaches. Each decision involves trade-offs that affect the validity of the results for a given research question. Experiments, for instance, provide strong internal validity because they allow strict control over study conditions, making them suitable for comparative and confirmatory research. However, this control may reduce construct validity, as experiments often fail to capture the full complexity of real-world contexts. Case studies, often based on

observational methods, provide high construct validity by being conducted in a real-world context, where the boundary between the phenomenon and the environment is not clearly defined. Yet, this advantage comes at the cost of internal validity, since sample selection and size may be limited, reducing the ability to isolate and verify causal relationships.

In this thesis, the research cycle is employed twice to iteratively advance understanding of code review for GUI-based tests. The first cycle focuses on exploring and confirming code review guidelines. In Chapter 2, we explore existing literature to synthesize general code review guidelines and adapt them to GUI-based testing. In Chapter 4, we confirm the adoption of these guidelines and their manifestation in open-source repositories. This cycle ensures that the proposed guidelines are both theoretically grounded and empirically observable in practice.

The second cycle addresses the practice and perception of code review for GUI-based tests. In Chapter 3, we explore specific challenges, practices, and information needs associated with code review of GUI-based tests and whether practitioners consider code review relevant in this context. In Chapter 5, confirm that code review is a relevant practice for GUI-based tests, but also explore how practitioners anticipate the future of code review. This cycle helps establish the practical relevance of code review for GUI-based tests and clarifies its evolving role within modern software development.

1.4.1 Literature Review

We conducted a systematic literature review (SLR) in Chapter 2 to gain a comprehensive overview of existing literature that provides evidence for guidelines on reviewing source and test code artifacts. To perform the SLR, we followed the guidelines established by Kitchenham and Charters [137]. According to Kitchenham and Charters [137], a SLR is defined 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"* . The process of conducting an SLR can be structured into three main phases: planning, conducting, and reporting the review. During the conducting phase, we selected five digital libraries to compile a comprehensive list of relevant sources. We performed data extraction from these sources and conducted a quality assessment in multiple stages using inclusion and exclusion criteria. To increase the validity of our findings, we utilized data triangulation, requiring a minimum of three sources to confirm each guideline candidate. Data triangulation is the practice of using different sources of information to increase the validity of a study's results [115].

This comprehensive overview of literature allows us to identify gaps in current research. In our case, we sought to determine whether there are any code review guidelines specifically tailored for GUI-based testing.

We synthesized the collected sources to systematically extract guidelines. For this purpose, we utilized coding based on the Straussian grounded theory approach [213]. This synthesis enabled us to map guidelines stated for general source and test code to guidelines for GUI-based testing through logical inference.

In contrast to related methods for literature review, such as a systematic mapping study (SMS) [137], a SLR is more suitable for our research goal 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.

To complement our findings with guidelines from non-academic sources, we broadened our literature review to include gray literature. Gray literature refers to materials that are not published in academic journals and therefore do not undergo peer review, like news articles, company reports, Wikipedia entries, and blogs [80]. We conducted the gray literature search using the same search string as the one used for the white literature search, but we utilized the general-purpose search engine Google. To manage the results, we employed a bounded effort strategy, limiting our search to the first 100 results. We applied distinct quality criteria to evaluate gray literature sources and ensured a clear separation between guidelines derived from gray literature and those from traditional academic sources, commonly referred to as white literature. This separation is important because incorporating gray literature can raise validity concerns. By utilizing both gray and white literature, our study is categorized as a multivocal literature review (MLR) [95].

Another method used to identify relevant literature in our review was snowballing. Snowballing is a reference-based search strategy that systematically identifies primary studies by tracing citations from an initial "take-off" paper [200, 226]. It represents a trade-off between completeness and selectivity [228], as no search strategy can achieve both perfect recall and precision [200]. Snowballing can be conducted in two forms: forward snowballing, which identifies new literature citing the take-off paper, and backward snowballing, which explores the reference lists of the take-off paper to identify new literature. In this study, we applied backward snowballing, using the final set of white literature sources as a basis for identifying additional gray literature. Snowballing could have been used to complement the white literature search, but given the number of literature sources we gathered and the theoretical saturation achieved from the initial round, we deemed snowballing unnecessary.

1.4.2 Qualitative Interviews

To explore code review practices, challenges, and information needs related to the code review of GUI-based tests, we conducted a qualitative interview study in Chapter 3. This study involved software testing professionals from various companies. A qualitative interview study is a type of basic qualitative research [160]. Inter-

views are the most common method of data collection in qualitative studies [160] and serve as an effective way to gather firsthand and in-depth information about a phenomenon [197].

The interviews were designed and semi-structured to allow exploration and in-depth discussions of emerging topics during the interview sessions [122]. For instance, we use open-ended questions like "How do you conduct code reviews?", which allows a follow-up about the practices shared by the interviewee. In contrast, structured interviews have very specific objectives and questions [197]. In a structured interview, a question would be more in the form of "Are you using method Y to conduct code reviews?" This does not allow for the exploration of other methods that the interviewee might mention, but it does help quantify responses to the specific questions asked. An interview guide was used to steer the discussions, following the recommendations outlined by Seaman [197].

The qualitative data analysis of the interviews was conducted using thematic analysis, as outlined by Braun and Clarke [43]. Thematic analysis was selected for its effectiveness in analyzing qualitative data, given its flexibility and capacity to yield unexpected insights while summarizing the key findings in the data [43]. The analysis began with familiarization, involving a manual review and cleanup of the interview transcriptions. Next, systematic coding was conducted by assigning preliminary codes to meaningful text segments, forming the initial code set. These preliminary codes were then examined and grouped into broader themes based on conceptual and semantic similarities. A theme, as defined by Braun and Clarke [43], captures "something important about the data in relation to the research question." To improve the reliability of the analysis, three additional authors independently coded a subset of the interviews and offered feedback on the themes and codes. Thematic analysis depends on the individual knowledge and perspective of the researcher, but this can be mitigated through inter-coder reliability sessions with additional coders [173]. Based on this feedback, ambiguities in the codes and themes were resolved, resulting in a refined and consistent thematic structure.

In the field of software engineering, case studies are prevalent [225], and the term *case study* is often used interchangeably with *qualitative research* [160]. However, as noted by Wohlin [225], 47% of the investigated articles labeled as case studies do not meet the criteria for an actual case study. Wohlin defines a case study as follows: *"A case study is an empirical investigation of a case, using multiple data collection methods, to study a contemporary phenomenon in its real-life context, with the investigator(s) not taking an active role in the investigated case."* [225]. This issue is also raised by Stol and Fitzgerald, who noted that the term *case study* is particularly problematic [210], and by Easterbrook et al. [74] pointed out that *"There is much confusion in the SE literature over what constitutes a case study."*

Since our research utilized only one data collection method (interviews), this does not qualify as a proper case study and would therefore be a misuse of the term *case study*.

1.4.3 Mining Software Repositories

In Chapter 4, we employ a mining software repositories (MSR) [119] method to collect code review data from open-source projects hosted on GitHub. We want to identify the extent to which concerns in code review comments from real-world projects align, thereby providing empirical evidence to support the proposed guidelines.

MSR enables researchers to extract and analyze software artifacts produced throughout the development lifecycle [120]. These artifacts are stored in a variety of repositories. *Historical repositories* capture records of past development activities, such as issue trackers or archived communication. *Run-time repositories* contain logs from deployed systems, recording operational behavior and failures. *Code repositories* maintain source code and its evolution over time, traditionally on platforms such as SourceForge or, more recently, GitHub. Modern platforms like GitHub integrate multiple repository types. In addition to hosting source code, GitHub records issue discussions, feature changes via pull requests, review comments, and general discussions. This integration provides a comprehensive view of both the technical and social processes of software development.

Our study focuses on code review comments in pull requests in GitHub repositories. To investigate code review comments related to GUI-based testing, we restrict our analysis to repositories that employ modern GUI-based testing frameworks, namely Playwright and Cypress. These two frameworks are gaining popularity, while older frameworks like Selenium, which are more widely covered in academic literature, are losing popularity among web developers [113]. While this choice limits the ability to compare results with previous academic work, it provides better examples of observable guidelines, making it more valuable for practitioners starting with a modern technology stack.

Following the ABC framework proposed by Stol and Fitzgerald [210], this work constitutes a sample study, as we aim to draw conclusions applicable to a larger population of open-source software projects. Sample studies are commonly employed to analyze extensive collections of software development artifacts or projects, especially within the realm of open-source software.

1.4.4 Questionnaire Survey

In Chapter 5, we conducted a questionnaire survey with software practitioners from different companies. We wanted to examine attitudes towards code review and to explore the changes practitioners anticipate in the near future.

A survey is a systematic method for collecting and summarizing data from a large representative group of the overall population of interest to obtain insights into specific phenomena [134]. Surveys are particularly useful in software engineering research because they enable the collection of data at scale and increase the potential to generalize findings across organizational and industrial contexts [134, 210].

A survey may refer to a range of data collection approaches, including interviews, questionnaires, or mixed methods, intended to capture quantitative or qualitative data on attitudes, behaviors, experiences, demographics, or opinions. The term *survey* is frequently used inaccurately in literature, and the term *questionnaire* is often treated as synonymous. However, as Kasunic [134] clarifies, a *survey* refers to a process, while the instrument for conducting a survey is the *questionnaire*. To avoid ambiguity, we adopt the term *questionnaire survey* to emphasize that our study relies exclusively on structured questionnaires as the instrument for data collection. In contrast, surveys based on interviews, which allow for richer qualitative insights but at a smaller scale, have been discussed extensively by Jansen [129]. By clarifying this methodological distinction, we ensure that our results are interpreted within the correct empirical frame.

In our context, a questionnaire survey is not suitable to investigate the challenges and practices of individual companies, but rather to identify broader trends that emerge across multiple organizations.

1.4.5 Other Methods

As an alternative to the MSR method used for gathering empirical evidence for the proposed guidelines, a (quasi-)experiment could be employed as an interventional method. In software engineering, Wohlin et al. [228] defines an experiment as "an empirical inquiry that manipulates one factor or variable of the studied setting." An experiment allows for a high level of control and is mostly done in a laboratory environment [228]. A quasi-experiment differs in that certain confounding factors, such as the assignment of treatments, cannot be randomly assigned to subjects.

To evaluate the proposed code review guidelines, we initially conducted a pilot study in the form of a quasi-experiment with a small group of four practitioners (Chapter 4). The study aimed to assess whether applying the guidelines to GUI-based tests improved the quantity and quality of review comments. A quantitative analysis of the number of code review comments could assess the effect size, while a qualitative analysis of the comments could provide insight into their quality and relevance. However, as detailed in Chapter 4, practical constraints limited the feasibility of this approach, leading us to pivot to MSR as the research method.

1.5 Contributions

In this section, we outline our contributions to this thesis, which are based on four objectives to answer the four high-level research questions.

1.5.1 Identify Code Review Guidelines (RQ1)

In Chapter 2, we conduct a systematic review of code review guidelines for both production and test code, of white and gray literature. With this systematic review, we want to understand how code review is currently supported across different artifact types (i.e., production code, test code, and GUI-based tests) and to determine whether specific guidelines exist for GUI-based tests. Our search yielded 1160 white literature sources and 378 gray literature sources. After applying inclusion and exclusion criteria, 137 white and 105 gray literature sources were deemed relevant. We expected to find guidelines mentioned for different types of artifacts. However, only 10 of the 137 white literature sources mentioned test-specific review guidelines, and none explicitly addressed GUI-based tests.

This notable absence of code review guidelines specific to GUI-based tests constitutes our first contribution, highlighting a gap in research. The lack of established guidelines for reviewing GUI-based tests demonstrates that existing knowledge cannot be directly reused or extended without adaptation. This finding motivates the need to contextualize and refine general code review guidelines for GUI-based testing.

To address this gap, we systematically mapped the identified guidelines for production and test code to GUI-based tests. This process resulted in 33 guidelines, organized into nine categories, as shown in Table 1.2. The categories “Provide context information” and “Ensure readability” not only contain the highest number of guidelines but are also well-supported by both white and gray literature sources. Of these, we labeled six guidelines as test-specific, as they were originally proposed for general testing. These test-specific guidelines are: “Provide information about the dependencies between test and production code”, “Provide information about test edge cases”, “Provide test coverage metrics (for effectiveness)”, “Follow coding style and naming practices in test writing”, “Ensure proper usage of techniques for testing and exception handling”, and “Ensure correctness of assertions in test cases.”

The final set of 33 guidelines represents our second contribution, providing a structured and comprehensive foundation for supporting code review of GUI-based tests. While these guidelines have not yet undergone empirical validation, this set of guidelines represents a starting point and was then refined through the continuation of the thesis work.

Table 1.2: Categories of code review guidelines for GUI-based tests, along with the number of guidelines (#G) and the supporting sources in each category (#WL = white literature, #GL = gray literature)

ID	Guideline category	#G	#WL	#GL
G1	Perform automated checks	2	27	32
G2	Use checklists	1	8	19
G3	Provide context information	9	44	54
G4	Utilize metrics	3	11	11
G5	Ensure readability	7	14	51
G6	Visualize changes	3	14	3
G7	Reduce complexity	3	34	32
G8	Check conformity with the requirements	1	0	3
G9	Follow design principles and patterns	4	0	18

1.5.2 Practices, Challenges, and Information Needs (RQ2)

In Chapter 3, we identify the practices, challenges, and information needs of code review when reviewing GUI-based tests. This was accomplished through a qualitative interview study with 14 software testing professionals from six different companies.

Participants in our study view code review as essential. However, they often lack clear, shared practices for reviewing GUI-based tests. Among the four practices reported, the most frequently mentioned is running tests on the local machine. Practitioners execute tests locally during the review process to ensure that they function as expected and to utilize the navigation and debug features offered by their development environment. Additionally, the limited navigation capabilities in code review tools are acknowledged by Spadini et al. [205], highlighting the difficulties in comprehending GUI-based tests through these tools, as all interviewees preferred to inspect the tests locally.

Participants reported several challenges related to the unique demands of GUI-based testing. A primary concern is ensuring the correct verification of the SUT. This involves understanding the relationship between the test code and the GUI, as well as using robust locators to guarantee that the correct elements are being verified. In some cases, it may also be necessary to observe the tests in real-time. Additional challenges include mixed levels of abstraction, such as simple sequences of test steps in combination with the abstractions of whole pages (often implemented using the page-object pattern). Test robustness is another concern. Tests should not fail when the SUT undergoes minor changes, which is also related to the usage of robust locators. Moreover, practitioners have noted a lack of testing experience, as GUI-based testing activities are performed infrequently. This infrequency leads to a slower accumulation of knowledge and experience with various testing frameworks and techniques. In these instances, where knowledge-sharing constraints exist, guidelines could help to promote good testing practices.

For addressing information needs, context information is essential. This includes the rationale behind changes, relationships to related artifacts, and the visual appearance

of the GUI. For test reviews specifically, information about test data is important, as variations in input data can significantly affect test outcomes and the accuracy of system verification.

Overall, our findings contribute to the understanding of code review practices and highlight the need for tailored review practices and tools specifically designed for GUI-based testing. This contribution offers a different perspective on the code review process for GUI-based tests compared to our previous work, which focused on guidelines.

1.5.3 Empirical Evidence for Guidelines (RQ3)

In Chapter 4, we gather empirical evidence on the prevalence of our guidelines in a sample of 100 popular open-source repositories that contain Playwright or Cypress tests. In total, we gathered data from 878 002 pull requests, of which 16 866 PRs (1.9%) modify GUI-based tests, and just 2084 PRs (0.24%) contained code review comments or discussions. The manual analysis of 1000 PRs revealed support for 25 out of the 33 proposed guidelines, drawing insights from review comments, review templates, and GitHub Actions. The analysis indicates that many concerns addressed by the proposed guidelines are reflected in practice, demonstrating their practical relevance and coverage for GUI-based test reviews.

The guidelines offer insights into *what* practitioners should focus on, rather than mandating specific methods on *how* to address them. This allows guidelines to be adapted to different contexts, but does not provide explicit support on how to achieve better quality discussions or artifacts. In practice, repositories utilize automated checks and continuous integration (CI) workflows, such as GitHub Actions and bots, to enforce various syntactic and structural standards. Additionally, 71 of the 100 repositories have implemented pull request templates that require contributors to provide contextual information before a review begins. Together, automation and templates complement the guidelines by reducing reviewers' cognitive burden and ensuring that review discussions focus on higher-level concerns.

The most frequently observed guideline of "Proper usage of testing techniques and exception handling" appeared in 277 PRs. To better capture the concerns associated with this guideline in real-world code review comments, we proposed the following refined sub-guidelines: "Select robust locators, like role-based or test-specific locators", "Avoid explicit waits and define timeouts on a test-case level", and "Avoid test annotations `skip` and `only`". The second most observed guideline of "Prioritization of files" appeared in 203 PRs, in which contributors highlighted files, or specific lines of a file, for review by placing a comment to inform or start a discussion with the reviewer.

This research contributes to the thesis by offering empirical evidence as a preliminary form of evaluation for the proposed guidelines. Additionally, we have provided

concrete examples of how guidelines are reflected in code review comments and have refined the most common guideline.

1.5.4 Anticipating the Future of Code Review (RQ4)

Code reviews, as a collaborative practice, may change in the future due to the increasing adoption of technologies such as large language models (LLMs).

In Chapter 5, we investigate how practitioners anticipate the future of code review, focusing on how emerging technologies may reshape review processes and responsibilities.

Our survey of 92 professional software developers from four large software-driven companies shows that practitioners continue to view code review as an essential practice. They expect to spend more time on reviews in the coming years. Over the next five years, they anticipate a growing interest in reviewing a wider variety of artifacts, including production code, test code, configuration files, documentation, and GUI-based tests. As shown in Figure 1.4, there is a notable growth in the anticipated review activities for GUI-based tests, compared to the increase of other types of artifacts. This indicates a rising recognition and integration of this practice in future software development methodologies. The results indicate that while LLMs are expected to become widespread and undertake more review tasks, code review continues to be a human-centered activity, grounded in human understanding and oversight.

This contributes to the thesis by highlighting the anticipated importance of code review as a collaborative practice in general, particularly regarding the review of GUI-based testing files in the future.

A high-level summary of the contributions outlined in this section is presented in Figure 1.5.

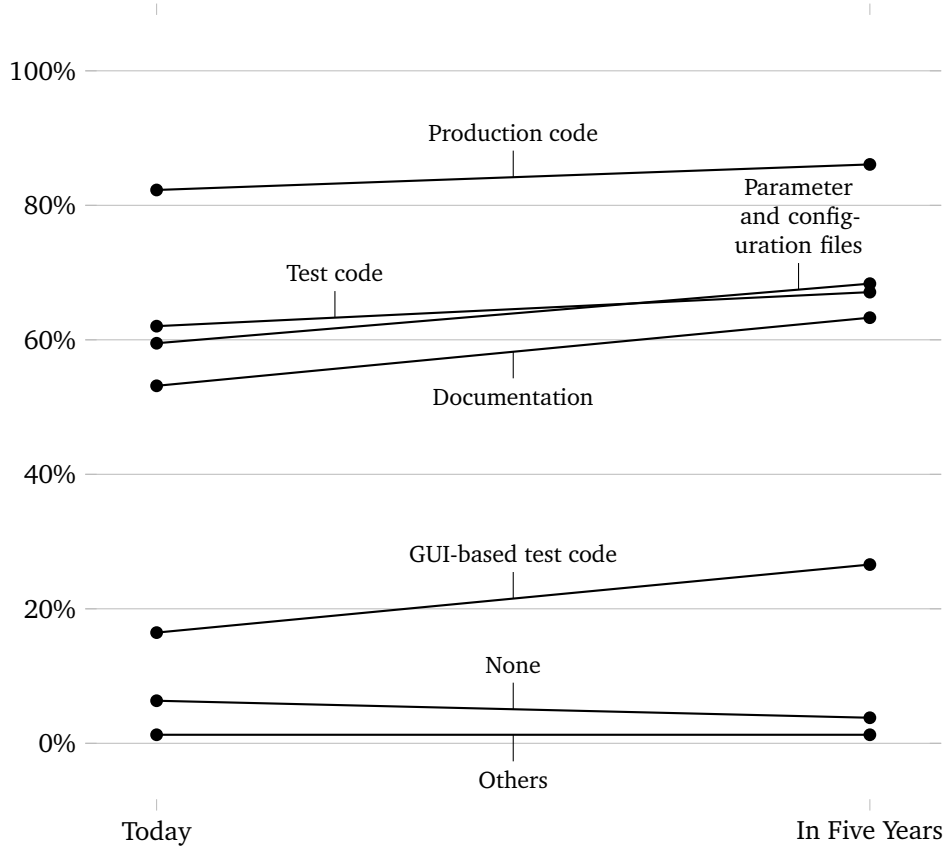

Figure 1.4: Practitioners anticipate reviewing a broader range of artifacts in the future, with notable growth in attention to GUI-based test code and a decline in those reviewing no artifacts at all. Y-axis scale representing the number of responses.

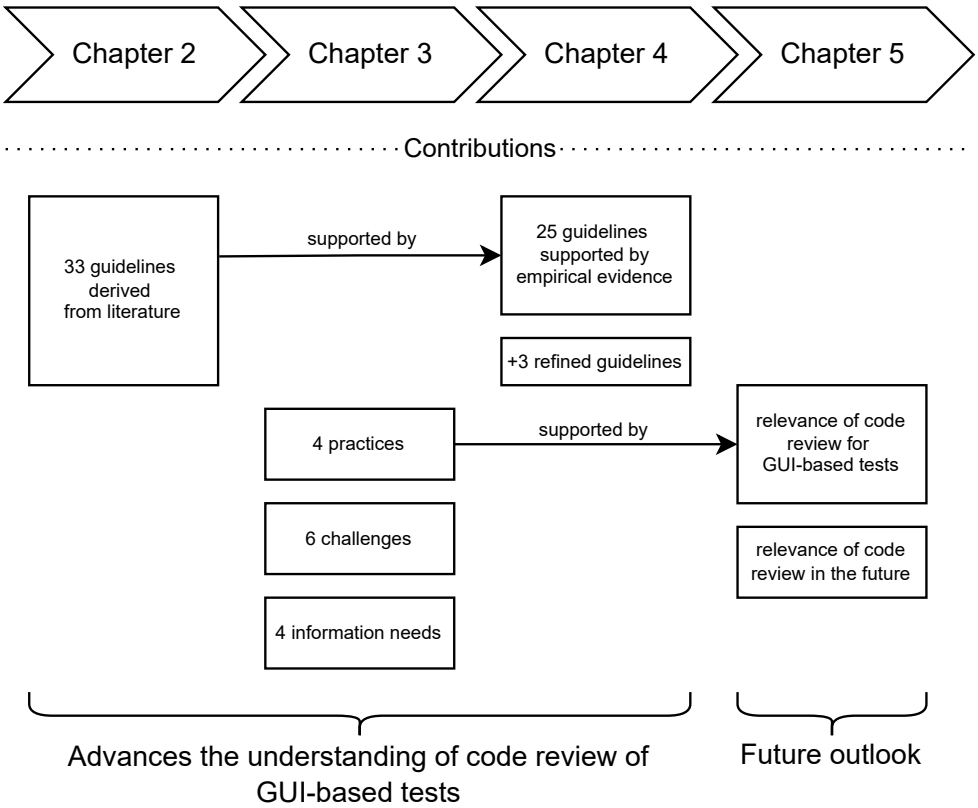

Figure 1.5: High-level summary of contributions per chapter

1.6 Discussion

In the following sections, we will discuss the contributions made in both the scientific and practical contexts.

Code Review Guidelines We see the proposed empirically grounded guidelines for code review of GUI-based tests as the central contribution of this thesis. Through the contributions of Chapter 2 and Chapter 4, we improved the understanding and practice of discussing GUI-based tests during code review. Transferring general code review guidelines to GUI-based tests may not be effective due to the unique challenges these tests present, e.g., the level of abstraction, the context of the GUI, and its synchronization. Therefore, guidelines need to be tailored specifically for GUI-based tests and supported by empirical evidence before they can be recommended to practitioners and introduced in an industrial context.

Given that GUI-based tests are created less frequently, expertise within mixed teams is less common, execution is more resource-intensive, and maintenance costs are higher compared to unit tests, the tests introduced into the codebase should be of high quality. This ensures they effectively contribute to overall quality assurance efforts. High-quality software depends on equally high-quality tests that prevent regression defects and ensure that new functionality does not degrade the user experience. Well-conducted code reviews with the support of guidelines could help achieve this goal by identifying issues early, improving maintainability, and promoting shared understanding among team members. Moreover, it encourages knowledge sharing within teams, enabling testers and developers to adopt effective testing techniques and consistent design principles.

While the elicitation of guidelines from academic literature resulted in more abstract terms (e.g., use of mocks, incorrect abstraction levels, or faulty variable initialization), our analysis of real-world code review comments revealed concrete concerns unique to GUI-based testing. For instance, the use of robust locators, the avoidance of explicit waits, and the risk of overusing annotations to skip tests. This shows how empirical evidence can refine abstract principles into more actionable ones for practitioners.

There are eight guidelines for which we could not find empirical evidence. Notably, the guideline concerning the dependencies between test code and the production code it tests was not reflected in the comments during code reviews. One possible explanation for this is that when both production code and the related test code are included in the same pull request, reviewers may find the overall summary of the changes to be sufficient. However, the practitioners we interviewed (Chapter 3) reported a lack of contextual information regarding the relationship between the tests and the GUI as information needs, which aligns with the study by Spadini et al. [205]. For other guidelines, such as providing information about the history of a change or the adherence to requirements, we assume that GitHub, as a code

review platform, already provides sufficient access to the commit history and allows the discussion about features and their requirements as GitHub issues. Comments regarding SOLID[5] principles could not be observed, or visualizing changes, assuming they are less relevant for GUI-based tests when using Playwright or Cypress. The remaining guidelines for using metrics and checklists are not reflected in code review comments, but they are already addressed through automation and pull request templates.

It is important to note that this work does not establish a causal relationship between adherence to the guidelines and quantifiable improvements in the code review process or test quality. This limitation is further discussed in Section 1.6.1.

Are Findings Unique to GUI-based Testing? Some of the reported review practices, challenges, and information needs are not unique to GUI-based tests. For example, the practice of reviewing test code alongside production code or the SUT, the challenges related to the understandability and readability of changes, and the need for more contextual information are all relevant across different types of artifacts. However, they had not been previously studied in this context. By systematically examining them for GUI-based tests, this thesis contributes to extending the body of knowledge into a previously underresearched area. This not only validates the relevance of established code review principles in a new setting but also clarifies how these principles manifest differently when applied to GUI-based tests. For instance, what practitioners mean when reporting understandability and readability as a recurring challenge during reviews.

Potential for Industry Adoption GUI-based test files are reviewed and modified less frequently than production code. As prior studies indicate, this reduced frequency limits reviewers' opportunities to develop comparable expertise in this area, with long-term consequences for test quality and maintainability [35, 205]. Our guidelines, therefore, serve as a valuable resource for less experienced contributors and reviewers, offering structured support to ensure quality even in the absence of deep expertise.

From a practical standpoint, guidelines can reduce the cognitive load of review by providing explicit criteria. Nevertheless, prior research shows that programming guidelines can, under some circumstances, increase cognitive load instead of reducing it [14]. Automation plays a key role in mitigating this risk. Our analysis revealed widespread adoption of CI-based automation, such as linting, structural checks, and coverage analysis, alongside pull request templates that prompt authors to explain rationale and validation steps. When integrated thoughtfully, such automation allows reviewers to focus on higher-level aspects while offloading routine checks to tools.

[5] Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion

This echoes broader evidence that sensible automation improves the efficiency and effectiveness of code reviews [21].

Taken together, the findings suggest that industry adoption of these guidelines is feasible and beneficial, provided organizations invest in complementary automation and training to support their effective use.

Is Code Review a Relevant Practice for GUI-based Tests Our empirical studies strongly emphasized the relevance of code review for GUI-based tests. In interviews with practitioners (Chapter 3), all participants see code review as an essential collaborative practice. Our survey results (Chapter 5) provide further support in favour of code review. Practitioners anticipate an increase in code review activities overall and especially for GUI-based tests in the near future. This trend underscores the growing recognition of GUI-based tests within software projects.

This finding is interesting, as GUI-based tests represent only a small fraction of overall testing efforts. Spadini et al. [205] demonstrated that tests in general are reviewed less frequently than production code. Our analysis of open-source repositories revealed that only 1.9% of all pull requests modify GUI-based tests, and just 0.24% actually discuss them. However, the low numbers do not imply that GUI-based tests are not valuable, but could support our proposition that reviewers require further support to thoroughly review those tests that will be integrated into the code base.

Accordingly, the contributions of this thesis are both timely and forward-looking. By establishing the relevance of code review in GUI-based testing, identifying specific practices and challenges, and proposing empirically grounded guidelines, this work provides a foundation for future research and improved industry practice.

1.6.1 Limitations

The main limitation of this thesis is the empirical approach used to evaluate the proposed guidelines for reviewing GUI-based tests. The research utilized an observational design to analyze how well existing code review comments align, either explicitly or implicitly, with the proposed guidelines. While this approach enabled the identification of naturally occurring concerns in code review comments, it does not establish a causal relationship between adherence to the guidelines and quantifiable improvements in the code review process or test quality. Therefore, this thesis cannot determine the impact of the guidelines when they are introduced to a project.

Interventional studies, such as controlled experiments or field experiments [210], would be required to establish such causal relationships. However, conducting these studies in industrial settings presents considerable challenges. Code review is an essential collaborative practice in contemporary software development workflows. Companies are understandably reluctant to modify or temporarily disrupt these

established processes for experimental purposes. Consequently, the feasibility of conducting field experiments is limited. Similarly, while controlled experiments in this context of code review are possible, as demonstrated by [206], the inclusion of GUI-based tests as artifacts under review introduces additional limiting factors. For instance, a smaller number of developers or testers with experience in GUI-based testing, comprehending whole testing scenarios of the SUT, instead of single functions, technical details required of the SUT's GUI to determine suitable techniques (i.e., locators). Nevertheless, as demonstrated in the doctoral thesis by Dorner [67], observational studies are a valid and valuable approach to theory evaluation in the context of code review research.

A further limitation is related to the sampling strategy used. In Chapter 3, where we investigate the practices and challenges of code review for GUI-based tests, we focus primarily on practitioners whose primary role is related to testing or quality assurance. This sampling omits insights from developers who occasionally perform GUI-based testing, which can bias the data in favor of GUI-based testing activities. Especially when it comes to the importance of performing code review of GUI-based tests. The survey presented in Chapter 5 shows that general developers expect to increase the review activities for GUI-based tests in the future. These findings only highlight an anticipated increase in the importance of such reviews and do not offer insights into specific practices, challenges, or information needs that may arise in mixed-role teams.

Finally, the generalizability of the results related to the guidelines is constrained due to the analysis of only open-source repositories, which differ from industrial companies that develop closed-source products. Open-source projects are usually developed by distributed teams that depend on asynchronous, text-based communication rather than in-person interactions. Contributors to open-source projects may participate sporadically since they are not bound to the project through employment agreements [25, 26, 141]. Studies by Rigby and Bird [184] and Bosu et al. [41] found substantial similarities in the code review processes of open-source and closed-source systems. However, relationships among code review participants are particularly important in open-source projects, whereas closed-source projects focus more on knowledge transfer [41]. Thus, we can not generalize our findings directly to an industrial closed-source context without further research.

1.6.2 Opportunities for Future Work

We see two main opportunities for future research and development.

First, the empirically grounded and now concretely exemplified guidelines provide a better foundation for experimental and interventional studies that assess their causal impact on software quality and review performance. Future work should investigate whether the explicit adoption of these guidelines improves the quality, maintainability,

and reliability of GUI-based tests, as well as the efficiency of the review process itself. Despite the mentioned challenges of conducting experiments, controlled field experiments or longitudinal case studies could provide quantifiable empirical evidence over the observational approach adopted in this thesis.

Second, the guidelines can be used as an input for tool-supported automation and developer assistance. They could be operationalized within continuous integration (CI) pipelines to automatically detect guideline violations or generate review reports. Alternatively, integration as an IDE-based linter or review assistant could provide immediate feedback during test development or before submission for review.

By making our research transparent and our artifacts publicly available on Zenodo[6], we enable the research community to replicate, build upon, and enhance this research.

1.7 Conclusion

Code review is an essential practice for collaborative software engineering. Guidelines to facilitate code review are established in both open-source and industrial communities, but there are currently no guidelines specifically for GUI-based tests. Furthermore, academic research does not cover specific practices, challenges, and information needs. This thesis advances the understanding and practice of code review for GUI-based tests to improve both review effectiveness and the quality of the tests under review.

First, our systematic review showed a lack of code review guidelines mentioned for GUI-based tests. To address this, we mapped existing guidelines for GUI-based tests, resulting in 33 guidelines organized into nine categories.

Second, our interview study with software testing professionals identified the specific practices, challenges, and information needs when reviewing GUI-based tests.

Third, our repository mining study of open-source projects demonstrated that the concerns of real-world code review comments align with 25 of the 33 proposed guidelines. We refined the most prevalent guideline on testing techniques and exception handling into concrete sub-guidelines, which resulted in more actionable guidelines.

Fourth, our survey of 92 professional developers from four large software-driven companies indicated that code review will remain essential. Practitioners expect to increase review activities and to review a broader set of artifacts, with notable growth anticipated for GUI-based tests.

The proposed guidelines offer specific recommendations for practitioners that can be used during the review process or incorporated into pull request templates and continuous integration pipelines. For tool developers, the findings can serve as an

[6] https://zenodo.org

input for future tool development aimed at better supporting the review of tests and GUI-based tests, e.g., provide more relevant contextual information. Researchers can use this work to evaluate guidelines in real-world industrial settings, assess their impact on review effectiveness and efficiency, and explore how automation and new technologies such as LLMs can further support code review for GUI-based tests.

A better understanding of code review for GUI-based tests with its specific challenges and information needs, along with clear guidelines, could improve the review process. This improvement fosters more productive discussions and better outcomes when evaluating these artifacts. Effective reviews improve the quality of the GUI-based tests being reviewed and, in consequence, the overall quality of the software systems they verify. Additionally, these reviews promote knowledge sharing between testers and developers.

2

Code Review Guidelines for GUI-based Testing Artifacts

This chapter is based on Andreas Bauer, Riccardo Coppola, Emil Alégroth, and Tony Gorschek. "*Code Review Guidelines for GUI-based Testing Artifacts*". In: *Information and Software Technology* 163 (November 2023), p. 107299. DOI: 10.1016/j.infsof.2023.107299.

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 the 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 the 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.

2.1 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 [73]. 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 [42, 190]. Code review is a software engineering practice where peers review a code contribution before additions or changes are integrated into the code base [20, 53]. As a practice, code reviews help to catch errors and improve sub-optimal solutions on different types of artifacts, but also enable knowledge and experience sharing and, maybe most importantly, allow active collaboration in the team [20, 83].

Another practice that has become ubiquitous in modern software development is automated testing [39]. 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 [8, 55]. Thus, GUI-based testing constitutes a powerful tool to find regression defects that users would otherwise encounter [8].

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 [66]. 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 the 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 [6, 49, 231].

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 2.2 we provide background information about GUI-based testing and Modern Code Reviews; in

Section 2.3 we illustrate the methodology that we employed to conduct our literature review; in Section 2.4 we report the results of the review; in Section 2.5 we discuss the results and the threats to validity of the present study; in Section 2.6 we conclude the paper by summarizing the main findings and providing future research directions.

2.2 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 the parts of the system containing the test cases that verify if the application works as expected *test artifacts*. The nature of test artifacts depends on the specific testing technique that is applied to the project.

2.2.1 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 [8]. GUI testing is defined as the practice of verifying the correctness of the GUI's visual appearance according to the system under test (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 [8, 55]. These tests are thereby system-level tests that are performed through End-to-End (E2E) scenarios—Sequences of events performed against the GUI [24]—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 [39]. 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) [12].

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 [44]). *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 [171]. *Model-based*

testing is based on 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 [199]. These models may be visual (i.e., defined with nodes and vertices) 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, Feldt, and Ryrholm [10], 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 [46].

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 [14], 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 as important to aid in improving the effectiveness and efficiency of both individual and team-based GUI testing.

2.2.2 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[1, 2]. Fagan [78] 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 [198].

Nowadays, code reviews have transformed towards informal, tool-supported, lightweight processes that build a communication platform for developers [20, 36, 41, 184]. 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) [20, 61]. Dedicated code review tools such as Gerrit [100], Phabricator [125], or CodeFlow [207] support the code review process by providing context-specific information. Modern software forges such as GitHub [124], GitLab [101], or Bitbucket [17] 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 [36].

Davila and Nunes [61] 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 recommendations 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 the analysis of change impact. Of the 22 included studies, 15 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. [66], 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. [205], 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 examples 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 [20]. 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, enabling easy navigation between test and production code, and providing detailed code coverage information for tests.

2.3 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, were 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 [131]. Thus, a conclusion is valid if the premises (a guideline can be applied to GUI testing) are 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.

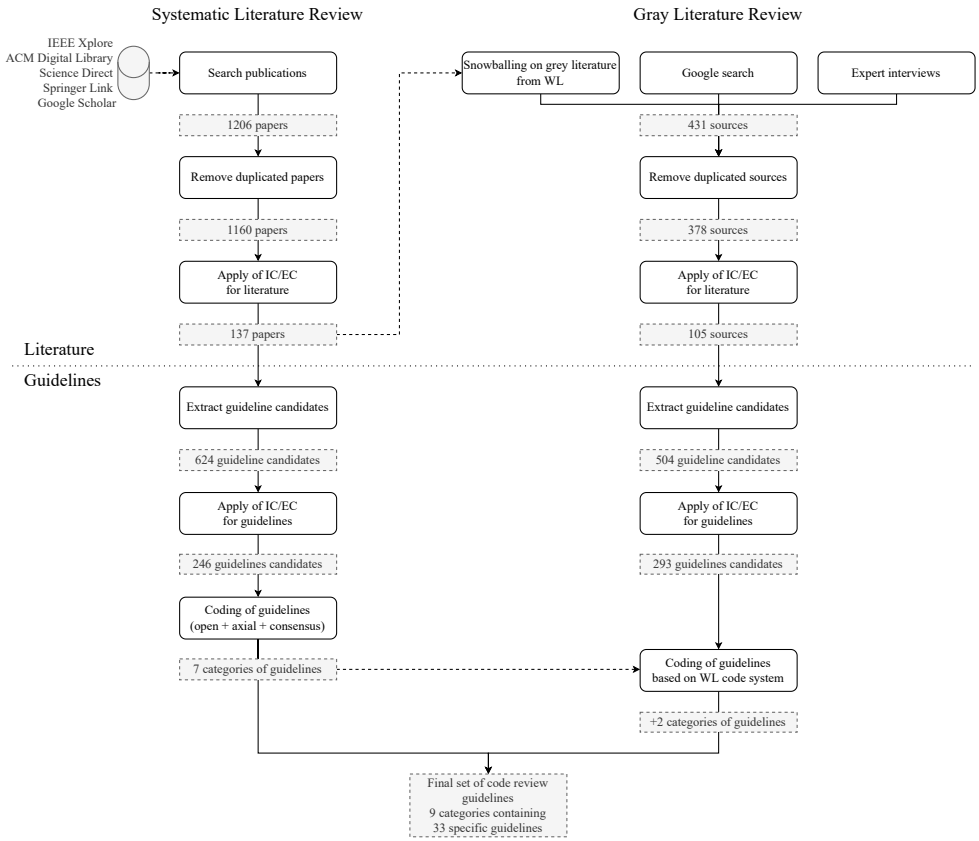

Figure 2.1: Summary of the research approach

This work is further motivated by an identified gap in guidelines for GUI-based testing, including GUI test review, in academic literature.

2.3.1 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?

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 judgment and experience with GUI testing (in excess of 30 years of combined knowledge from both industry and academia).

2.3.2 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.

All five phases of the study were conducted consecutively, where the output of one phase was used as input to the next phase in the 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 [177]. 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[1].

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 2.5.1.

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 [77]—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 question, 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 the 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 tests 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

[1]The dataset is available at the following URL: `https://doi.org/10.5281/zenodo.7248201`

Library	URL	#Studies
IEEE Xplore	`https://ieeexplore.ieee.org`	321
ACM Digital Library	`https://dl.acm.org`	122
Science Direct	`https://www.sciencedirect.com`	112
Springer Link	`https://link.springer.com`	628
Google Scholar	`https://scholar.google.com`	23
Duplicates		46
Total (without duplicates)		**1160**

Table 2.1: Search results by library

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 [137]. 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. 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 2.1, 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 [97].

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 the present manuscript. We opted not to 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 keywords 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 2.1 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 [137]) 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 2.1).

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 their metadata.

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) [94].

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* [80].

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 [3]. We used this classification of GL as a template for our quality assessment questionnaire (Table 2.2) 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 [95].

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 additional papers* [226]. 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 2.2)*.

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 2.2. 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 (>=75th percentile=1, >=50th percentile=0.75, >=25th percentile=0.5, <25th percentile=0).

The questionnaire was adapted from the quality assessment questionnaire proposed in [94], 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 engine, and

Criteria	Question
Authority	Is the publishing organization reputable? E.g., a research, software, or technology organization.
Authority	Is the author reputable? (e.g., experience, job title, or other publications in the field)
Outlet	1st tier GL (measure=1): High credibility: Books, magazines, reports, or documentation from companies 2nd tier GL (measure=0.5): Moderate credibility: News articles, presentations, videos, Wiki articles 3rd tier GL (measure=0): Low credibility: Blogs, emails, tweets, Q/A sites (such as StackOverflow)
Impact	Number of backlinks (using `https://www.seoreviewtools.com/valuable-backlinks-checker/`) or citations.

Table 2.2: 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

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 selective, 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 [214], 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 (i.e., *Glaserian* Grounded Theory [102]) 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 passed, 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 a macro-category is present, the category is assigned to it; (2) if no suitable macro-category 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 passed, meetings were performed by using the mini-Delphi approach to obtain a consensus on what macro-category to assign to 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.

2.4 Results

In this section, we will present our results on the guidelines used 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 2.3, provide an answer to RQ1.

Next, we present the categories of used guidelines in Table 2.4, followed by a structured textual format with descriptions and examples of the application to GUI-based test artifacts, summarized in Table 2.5, to answer RQ2.

2.4.1 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

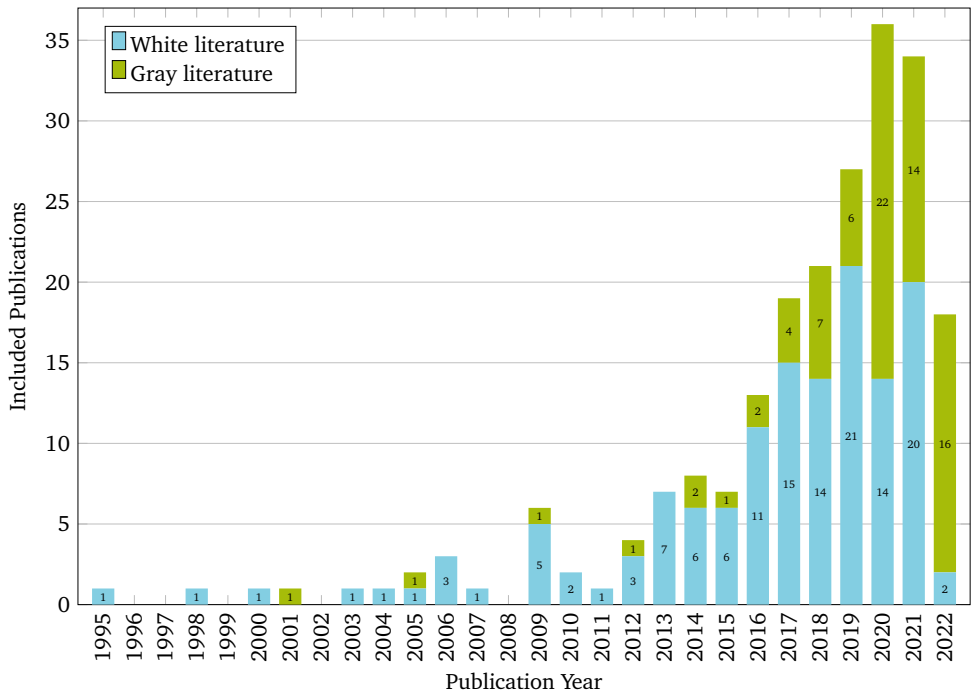

Figure 2.2: White and gray literature publications per year

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 a single guideline. Thus, providing more in-depth results compared to the gray literature.

Figure 2.2 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, 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 2.2.

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 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 superset 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 2.6 for white literature and Table 2.7 for gray literature in the appendix.

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

Table 2.3: Overview of code review guidelines used for source and test artifact review (WL=white literature, GL=gray literature, TS=test specific)

ID	Guideline	#WL	#GL	TS	WL Sources	GL Sources
G1	**Perform automated checks**					
G1.1	Perform automated checks on the change	21	21	no	S1, S2, S3, S9, S15, S18, S19, S20, S25, S26, S27, S28, S30, S36, S39, S42, S46, S60, S65, S75, S79	GS1, GS4, GS6, GS7, GS8, GS9, GS11, GS13, GS17, GS18, GS21, GS26, GS30, GS36, GS37, GS47, GS49, GS50, GS55, GS61, GS62
G1.2	Perform automated checks for code style	6	11	no	S36, S47, S56, S73, S76, S77	GS5, GS11, GS14, GS15, GS17, GS21, GS45, GS50, GS52, GS53, GS54
G2	**Use checklists**					
G2.1	Provide/use a checklist	8	19	no	S2, S6, S16, S19, S34, S42, S61, S76	GS2, GS4, GS8, GS9, GS11, GS13, GS14, GS15, GS20, GS22, GS26, GS30, GS37, GS39, GS42, GS47, GS54, GS56, GS62
G3	**Provide context information**					
G3.1	Provide additional context information	17	19	no	S12, S14, S17, S22, S35, S42, S46, S48, S50, S54, S62, S68, S71; S72, S73, S74, S77	GS2, GS8, GS9, GS13, GS16, GS24, GS27, GS30, GS35, GS36, GS37, GS39, GS42, GS46, GS51, GS54, GS58, GS59, GS63
G3.2	Provide rationale for the change	10	14	no	S3, S19, S21, S22, S35, S36, S75, S77, S78, S79	GS5, GS12, GS13, GS14, GS24, GS26, GS28, GS36, GS40, GS42, GS47, GS50, GS57, GS59
G3.3	Provide context information about the impact of the change	1	1	no	S22	GS3

Continued on next page

Table 2.3: (continued)

ID	Guideline	#WL	#GL	TS	WL Sources	GL Sources
G3.4	Provide information about the design and architecture of code affected by the change	2	4	no	S58, S63	GS5, GS21, GS54, GS59
G3.5	Provide links to related resources and documentation	6	13	no	S7, S10, S22, S29, S69, S79	GS3, GS5, GS13, GS17, GS19, GS21, GS35, GS42, GS43, GS44, GS50, GS51, GS59
G3.6	Provide information about the dependencies between test and production code	1	0	yes	S77	
G3.7	Provide context information about the history of changes	4	0	no	S18, S35, S62, S75	
G3.8	Provide information about test edge cases	1	2	yes	S77	GS43, GS59
G3.9	Provide a prioritization of the files/classes of the change	2	1	no	S53, S55	GS36
G4	**Utilize metrics**					
G4.1	Measure and monitor code metrics	7	7	no	S35, S36, S37, S40, S45, S51, S62	GS2, GS4, GS26, GS31, GS39, GS37, GS48
G4.2	Provide metrics about execution time (for efficiency)	1	2	no	S47	GS21, GS63
G4.3	Provide test coverage metrics (for effectiveness)	3	2	yes	S55, S57, S77	GS31, GS34
G5	**Ensure readability**					
G5.1	Ensure readability of the change	5	9	no	S1, S39, S42, S47, S76	GS5, GS13, GS21, GS23, GS31, GS41, GS42, GS50, GS57
G5.2	Provide comments	2	7	no	S47, S78	GS3, GS5, GS6, GS22, GS26, GS29, GS47
G5.3	Follow coding style and naming conventions	2	23	no	S38, S77	GS3, GS5, GS8, GS10, GS16, GS17, GS19, GS20, GS21, GS23, GS29, GS31, GS32, GS33, GS34, GS35, GS36, GS40, GS41, GS42, GS43, GS51, GS63
G5.4	Follow coding style and naming practices in test writing	2	0	yes	S67, S77	
G5.5	Avoid code comments if they are not clear and useful	0	5	no		GS8, GS17, GS19, GS34, GS35
G5.6	Ensure proper usage of techniques for testing and exception handling	2	5	yes	S47, S77	GS21, GS23, GS31, GS35, GS51
G5.7	Ensure correctness of assertions in test cases	1	2	yes	S77	GS23, GS31
G6	**Visualize changes**					
G6.1	Provide a visualization of the change	9	1	no	S11, S13, S31, S42, S49, S52, S55, S66, S67	GS57

Continued on next page

Table 2.3: (continued)

ID	Guideline	#WL	#GL	TS	WL Sources	GL Sources
G6.2	Provide a visualization of the change regarding its impact on the code base	3	0	no	S5, S13, S40	
G6.3	Allow traceability and easy navigation between artifacts	2	0	no	S42, S80	
G7	**Reduce complexity**					
G7.1	Keep size of a change as low as possible	15	15	no	S14, S18, S19, S22, S33, S36, S39, S42, S43, S62, S64, S68, S70, S75, S78	GS5, GS6, GS11, GS14, GS15, GS25, GS27, GS28, GS31, GS35, GS50, GS54, GS60, GS61, GS63
G7.2	Keep complexity of a change as low as possible	13	9	no	S4, S8, S18, S22, S39, S41, S47, S62, S68, S77, S78, S79, S80	GS3, GS19, GS22, GS28, GS29, GS31, GS35, GS42, GS51
G7.3	Avoid unrelated and unstructured changes	6	8	no	S14, S19, S23, S24, S47, S68	GS5, GS6, GS13, GS14, GS15, GS23, GS34, GS56
G8	**Check conformity with the requirements**					
G8.1	Ensure conformance with the requirements	0	3	no		GS17, GS35, GS38
G9	**Follow design principles and patterns**					
G9.1	Apply established design principles and patterns	0	9	no		GS21, GS29, GS31, GS34, GS35, GS36, GS42, GS50, GS51
G9.2	Don't repeat yourself (DRY)	0	4	no		GS21, GS31, GS42, GS43
G9.3	Avoid hardcoded values	0	2	no		GS31, GS34
G9.4	SOLID principle	0	3	no		GS21, GS31, GS42

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 2.3, and a categories-only summary is shown in Table 2.4. In Table 2.3, 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 contributors of code changes and the reviewers to improve the effectiveness and efficiency of the code review process. On the one hand, efficiency is improved by providing all the information required by the reviewer to comprehend the code changes. Thus, it mitigates the need for reviewers to spend time gathering this information themselves. Moreover, automated checks reduce the manual effort and prevent low-level discussion about

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

Table 2.4: Categories of code review guidelines used for source and test artifact review (WL=white literature, GL=gray literature)

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 2.4) 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 2.3, and a categories-only summary is shown in Table 2.4. 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 contributors of code changes and the reviewers to improve the effectiveness and efficiency of the code review process.

2.4.3 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 2.5 presents the mapping of the identified guidelines to GUI-based test artifacts. We structure this section according to the categories presented in Table 2.4 to answer RQ2. For generalizability, the level of abstraction of each guideline has been kept at 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) the 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 2.5 is provided by adding the corresponding ID in parentheses. Each guideline has also been ranked in terms of whether 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 2.5: Overview of code review guidelines and their mapping to GUI testing

ID	Guideline	Mapping to GUI-based test artifacts
G1	**Perform automated checks**	
G1.1	Perform automated checks on the change	Apply static code analysis tools to identify unsuitable or incorrect patterns in the code
G1.2	Perform automated checks for code style	Apply static code analysis tools to review and format code according to a style guide
G2	**Use checklists**	
G2.1	Provide/use a checklist	Provide/use a checklist that is tailored to the company or team that covers aspects related to GUI element localization, test oracles, or synchronization
G3	**Provide context information**	
G3.1	Provide additional context information	Provide contextual information about the code change as a comment during code review or as a comment or annotation to the artifact
G3.2	Provide rationale for the change	Provide contextual information about the motivation why the changes are valid for the purpose of the test
G3.3	Provide context information about the impact of the change	Provide contextual information as a comment during the code review about how the change will impact other reusable test cases or components that are dependent on the change, e.g., how many test cases depend on a changed functionality
G3.4	Provide information about the design and architecture of code affected by the change	Provide contextual information about the logical and chronological behavior of the GUI tests
G3.5	Provide links to related resources and documentation	Provide list of references (links) as a 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 comments 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**	

Continued on next page

Table 2.5 – continued from previous page

ID	Guideline	Mapping to GUI-based test artifacts
G4.1	Measure and monitor code metrics	Measure and monitor code metrics on test cases like the complexity, coverage, or run-time
G4.2	Provide metrics about execution time (for efficiency)	Measure execution time of test cases
G4.3	Provide test coverage metrics (for effectiveness)	Measure feature-, scenario- or GUI-element coverage
G5	**Ensure readability**	
G5.1	Ensure readability of the change	Ensure readability by following code styles, norms, and conventions
G5.2	Provide comments	Comment and annotate GUI-based test artifacts to explain difficult aspects and give the reviewer insights into the test creator's reasoning behind the tests
G5.3	Follow coding style and naming conventions	Script-based GUI tests should follow a set coding style, norms, and naming convention
G5.4	Follow coding style and naming practices in test writing	Script-based GUI tests should follow a set coding style, norms, and naming convention
G5.5	Avoid code comments if they are not clear and useful	Ensure comments and annotations are helpful to the reviewer and provide additional information that can not be easily derived from the test artifact itself
G5.6	Ensure proper usage of techniques for testing and exception handling	Check for improper use of testing techniques, such as incorrect usage of mocks, testing on the wrong level of abstraction, or wrong variable initialization
G5.7	Ensure correctness of assertions in test cases	Check that assertions are following established test patterns
G6	**Visualize changes**	
G6.1	Provide a visualization of the change	Provide a graph representation of visited GUI states of the SUT
G6.2	Provide a visualization of the change regarding its impact on the code base	Presenting screenshots, or screen recordings, of the GUI states that are covered by the test
G6.3	Allow traceability and easy navigation between artifacts	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 to minimize the use of branching scenarios to keep scripts as short and focused as possible
G7.2	Keep complexity of a change as low as possible	Apply patterns to minimize the use of branching scenarios to keep scripts as short and focused as possible
G7.3	Avoid unrelated and unstructured changes	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
G8	**Check conformity with the requirements**	
G8.1	Ensure conformance with the requirements	Review the changed test scenario in terms of conformance with the requirements
G9	**Follow design principles and patterns**	
G9.1	Apply established design principles and patterns	Apply established design principles and patterns for source code and explicit ones for GUI testing
G9.2	Don't repeat yourself (DRY)	Extract common functionality into reusable components and reuse them throughout the test. Apply design patterns such as the page-object pattern
G9.3	Avoid hardcoded values	Avoid hardcoded values, such as hardcoded identifiers
G9.4	SOLID principle	Avoid unrelated changes to conform with the single responsibility principle

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 [190]. 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 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 tests, 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 [187]. However, the guidelines shall take the domain, company, team culture, and other environmental aspects into account [88].

Guidelines: Providing and using a checklist (G2.1) may have different origins, depending on the purpose of the checklist. For instance, if providing guidance on 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 [19].

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 the 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 reviewer'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 [178]. 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 [178].

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 [211].

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 their 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 of 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, the 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 [155].

One way to monitor the efficiency of changes, be it source or test artifacts, is to monitor the execution time (G4.2) of artifacts. For monitoring, we include execution time before and after the 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 [41]. 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 [205].

Example(s) of application: The complexity of GUI-based tests shall also be 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 code, or use test libraries such as XUnit [44, 84]. 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 [110] or Scout [167], 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 [13]. Therefore, information about the execution time and the required resources of these test cases is important to estimate the scalability 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, the 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 at, 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 [136] or Cobertura [52]. 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 [45] 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. [190]. 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 conventions (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 [108]. 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. [15], 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 [205]. 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 provides recommendations if this is not aligned.

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 [205]. Additionally, the reviewer must evaluate whether 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. [36] 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 [205]. 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 their dependent 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 [117]. Hence, although visualization of changed technical artifacts is minimal, 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 [117] 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 navigate more easily 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 is therefore helpful (G6.3). For instance, it mitigates 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. [205], 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 which 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., the change impact of the test code change.

We rank this guideline as *suggested*. While we see the benefits of visualizing changes, their implementation will depend highly on the 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 software, where more complex components are more likely to have defects [159].

Guidelines: The main approach to reducing complexity is to keep code changes for review as small as possible (G7.1). Small code changes are preferred by reviewers, and they can provide more useful feedback compared to big code changes [65]. In addition, complexity should be avoided (G7.2) by having a narrow and well-defined scope [65]. Although a development practice, these practices reduce 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 [42]. 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 the 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 are 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 of 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 being 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 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.

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 to ensure 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 is 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 [152]. Design patterns are thereby blueprints for building reusable solutions for common problems. These patterns are well proven and have evolved over time [90].

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 [202] 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 [152]. Although these are patterns that the developer must consider, it is part of the reviewers' responsibility to verify that they are followed. The 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) [14]. 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 being repeated (G9.2). Further, hardcoded values, such as widget identifiers, should also be avoided (G9.3). The reason is that if this functionality or these values change, 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 a time. This practice helps to reduce the coupling between test cases and the SUT [143]. 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 of their application. A summary of this mapping is shown in Table 2.5. In addition, we explained the purpose of each guideline category and ranked each guideline as *suggested*, *recommended*, or *strongly recommended* based on reference materials.

2.5 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, it omits 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 [115]—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" [75]. 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 [14]. Although the study is limited in scope, it suggests that the 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 the 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 sets 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 that, as stated, there is a 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 are 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 input 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 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 on 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 best for each context.

2.5.1 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 [188].

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 at which they are presented. Another threat is that the mapping between the source and test code guidelines is invalid because we have failed to take the 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 [11].

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 terms 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 is 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 to 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 a 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 from 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 whether 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 [11]. However, as discussed by Alegroth, Petersen, and Tinnerholm [14], 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 with Systematic Literature Reviews is researchers' fatigue, i.e., the possibility of introducing biases in the analysis of large data sets, 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.

2.6 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, on how it can be applied to the 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 or 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 of the industry and challenges in this area.

Table 2.6: White literature sources where guidelines were taken

ID	Title	Authors	Year
S1	A Community of Practice Around Peer Review for Long-Term Research Software Sustainability	Kalyan, Akshay, et al.	2019
S2	A reflective practice of automated and manual code reviews for a studio project	Oh, Jun-Suk, and Ho-Jin Choi	2005
S3	A Secure Code Review Retrospective	Buttner, Andrew, et al.	2020
S4	Accept or Not? An Empirical Study on Analyzing the Factors that Affect the Outcomes of Modern Code Review?	Wang, Dandan, et al.	2021
S5	Aiding Code Change Understanding with Semantic Change Impact Analysis	Hanam, Quinn, et al.	2019
S6	An Effective Source Code Review Process for Embedded Software	Hirayama, Masayuki, et al.	2006
S7	An Empirical Study of Link Sharing in Review Comments	Jiang, Jing, Jin Cao, and Li Zhang	2019
S8	Analyzing Involvements of Reviewers through Mining a Code Review Repository	Liang, Junwei, and Osamu Mizuno	2011
S9	Assisting the Code Review Process Using Simple Pattern Recognition	Farchi, Eitan, and Bradley R. Harrington	2006
S10	Automatic patch linkage detection in code review using textual content and file location features	Wang, Dong, et al.	2021
S11	Can Formal Methods Improve the Efficiency of Code Reviews?	Hentschel, Martin, et al.	2016
S12	Can peer code reviews be exploited for later information needs?	Sutherland, Andrew, and Gina Venolia	2009
S13	ChangeViz: Enhancing the GitHub Pull Request Interface with Method Call Information	Gasparini, Lorenzo, et al.	2021
S14	Characteristics of Useful Code Reviews: An Empirical Study at Microsoft	Bosu, Amiangshu, et al.	2015
S15	Code review analysis of software system using machine learning techniques	Lal, Harsh, and Gaurav Pahwa	2017
S16	Code review and cooperative pair programming best practice	Fu, Qiang, et al.	2017
S17	Code Review Knowledge Perception: Fusing Multi-Features for Salient-Class Location	Huang, Yuan, et al.	2020
S18	Code Review Quality: How Developers See It	Kononenko, Oleksii, et al.	2016
S19	Code Reviewing in the Trenches: Challenges and Best Practices	MacLeod, Laura, et al.	2018
S20	Code Reviews With Divergent Review Scores: An Empirical Study of the OpenStack and Qt Communities	Hirao, Toshiki, et al.	2022
S21	Communicative Intention in Code Review Questions	Ebert, Felipe, et al.	2018
S22	Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies	Ebert, Felipe, et al.	2019
S23	CoRA: Decomposing and Describing Tangled Code Changes for Reviewer	Wang, Min, et al.	2019
S24	Decomposing Composite Changes for Code Review and Regression Test Selection in Evolving Software	Guo, Bo, et al.	2019
S25	Effects of Adopting Code Review Bots on Pull Requests to OSS Projects	Wessel, Mairieli, et al.	2020
S26	Evaluating how static analysis tools can reduce code review effort	Singh, Devarshi, et al.	2017
S27	Expectations, outcomes, and challenges of modern code review	Bacchelli, Alberto, and Christian Bird	2013

Continued on next page

Table 2.6 – continued from previous page

ID	Title	Authors	Year
S28	Fix-it: An extensible code auto-fix component in Review Bot	Balachandran, Vipin	2013
S29	Generating Code Review Documentation for Auto-Generated Mission-Critical Software	Denney, Ewen, and Bernd Fischer	2009
S30	Impact of Coding Style Checker on Code Review - A Case Study on the OpenStack Projects	Ueda, Yuki, et al.	2018
S31	Interactive Code Review for Systematic Changes	Zhang, Tianyi, et al.	2015
S32	Investigating code reading techniques for novice inspectors: an industrial case study	Rong, Guoping, et al.	2014
S33	Investigating code review quality: Do people and participation matter?	Kononenko, Oleksii, et al.	2015
S34	Java code reviewer for verifying object-oriented design in class diagrams	Jinto, Kanit, and Yachai Limpiyakorn	2010
S35	Lessons Learned from Building and Deploying a Code Review Analytics Platform	Bird, Christian, et al.	2015
S36	LightSys: Lightweight and Efficient CI System for Improving Integration Speed of Software	Lim, Geunsik, et al.	2021
S37	Mining Peer Code Review System for Computing Effort and Contribution Metrics for Patch Reviewers	Mishra, Rahul, and Ashish Sureka	2014
S38	Mining Source Code Improvement Patterns from Similar Code Review Works	Ueda, Yuki, et al.	2019
S39	Modern Code Review: A Case Study at Google	Sadowski, Caitlin, et al.	2018
S40	Multi-Perspective Visualization to Assist Code Change Review	Wang, Chen, et al.	2017
S41	Natural Language Insights from Code Reviews that Missed a Vulnerability	Munaiah, Nuthan, et al.	2017
S42	On the Need for a New Generation of Code Review Tools	Baum, Tobias, and Kurt Schneider	2016
S43	On the understanding of programs with continuous code reviews	Bernhart, Mario, and Thomas Grechenig	2013
S44	Patch Review Processes in Open Source Software Development Communities: A Comparative Case Study	Asundi, Jai, and Rajiv Jayant	2007
S45	Guiding Developers to Make Informative Commenting Decisions in Source Code	Huang, Yuan, et al.	2018
S46	Practical aspects on the assessment of a review process	Kiiskila, Janne	1998
S47	Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft	Bosu, Amiangshu, et al.	2017
S48	RAID: Tool Support for Refactoring-Aware Code Reviews	Brito, Rodrigo, and Marco Tulio Valente	2021
S49	Rebasing in Code Review Considered Harmful: A Large-Scale Empirical Investigation	Paixao, Matheus, and Paulo Henrique Maia	2019
S50	Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox	AlOmar, Eman Abdullah, et al.	2021
S51	Review Dynamics and Their Impact on Software Quality	Thongtanunam, Patanamon, and Ahmed E. Hassan	2021
S52	RSTrace+: Reviewer suggestion using software artifact traceability graphs	Sülün, Emre, et al.	2020

Continued on next page

Table 2.6 – continued from previous page

ID	Title	Authors	Year
S53	Salient-class location: help developers understand code change in code review	Huang, Yuan, et al.	2018
S54	SCRUB: a tool for code reviews	Holzmann, Gerald	2010
S55	Semantics-assisted code review: An efficient tool chain and a user study	Menarini, Massimiliano, et al.	2017
S56	Share, But be Aware: Security Smells in Python Gists	Rahman, Md Rayhanur, et al.	2019
S57	Static Security Analysis Based on Input-Related Software Faults	Nagy, Csaba, and Spiros Mancoridis	2009
S58	Supporting Automatic Code Review via Design	He, Jiantao, et al.	2013
S59	Test-Driven Code Review: An Empirical Study	Spadini, Davide, et al.	2019
S60	Testing Web Enabled Simulation at Scale Using Metamorphic Testing	Ahlgren, John, et al.	2021
S61	The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study	Rong, Guoping, et al.	2012
S62	The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects	McIntosh, Shane, et al.	2014
S63	The Impact of Code Review on Architectural Changes	Paixao, Matheus, et al.	2021
S64	The influence of non-technical factors on code review	Baysal, Olga, et al.	2013
S65	The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more	Hentschel, Martin, et al.	2019
S66	Tool Support for Code Change Inspection with Deep Learning in Evolving Software	Ayinala, Krishna Teja, et al.	2020
S67	Tool Support for Managing Clone Refactorings to Facilitate Code Review in Evolving Software	Chen, Zhiyuan, et al.	2017
S68	Towards a taxonomy of code review smells	Dogan, Emre, and Eray Tüzün	2021
S69	Understanding shared links and their intentions to meet information needs in modern code review	Wang, Dong, et al.	2021
S70	Using Metrics to Track Code Review Performance	Izquierdo-Cortazar, Daniel, et al.	2017
S71	Using Paragraph Vectors to improve our existing code review assisting tool-CRUSO	Kapur, Ritu, et al.	2021
S72	VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits	Perl, Henning, et al.	2015
S73	What Are They Talking About? Analyzing Code Reviews in Pull-Based Development Model	Li, Zhi-Xing, et al.	2017
S74	What Design Topics do Developers Discuss?	Viviani, Giovanni, et al.	2018
S75	What makes a code change easier to review: an empirical investigation on code change reviewability	Ram, Achyudh, et al.	2018
S76	What Types of Defects Are Really Discovered in Code Reviews?	Mäntylä, Mika V., and Casper Lassenius	2009
S77	When Testing Meets Code Review: Why and How Developers Review Tests	Spadini, Davide, et al.	2018
S78	Why Did This Reviewed Code Crash? An Empirical Study of Mozilla Firefox	An, Le, et al.	2018
S79	Why Do Developers Reject Refactorings in Open-Source Projects?	Pantiuchina, Jevgenija, et al.	2021

Continued on next page

Table 2.6 – continued from previous page

ID	Title	Authors	Year
S80	Why Security Defects Go Unnoticed During Code Reviews? A Case-Control Study of the Chromium OS Project	Paul, Rajshakhar, et al.	2021

Table 2.7: Gray literature sources where guidelines were taken

ID	Title	URL	Year
GS1	9 Best Practices for Code Review You Really Need	perforce.com	2019
GS2	Best Practices for Peer Code Review - SmartBear	smartbear.com	2022
GS3	How to do a code review | eng-practices - Google	google.github.io	2020
GS4	5 code review best practices - Work Life by Atlassian	atlassian.com	2022
GS5	Code Review Best Practices - Palantir Blog	blog.palantir.com	2018
GS6	8 Proven Code Review Best Practices for Developers - Snyk	snyk.io	2022
GS7	How to Make Good Code Reviews Better - Stack Overflow Blog	stackoverflow.blog	2019
GS8	16 Tech Leaders Share Smart Best Practices For Reviewing Code	forbes.com	2020
GS9	Reviewing Code - Best practices and techniques for code review	codegrip.tech	2021
GS10	How To Review Someone Else's Code: Tips and Best Practices	codecademy.com	2021
GS11	Code Review good practices: guide for beginners	medium.com	2021
GS12	5 code review best practices. Make others like your code review	tsh.io	2020
GS13	Code Review Guidelines - GitLab Documentation	docs.gitlab.com	2022
GS14	Proven Code Review Best Practices from Microsoft	michaelagreiler.com	2019
GS15	Code review best practices - DeepSource	deepsource.io	2019
GS16	13 Code Review Standards Inspired by Google	betterprogramming.pub	2020
GS17	What is Code Review? - Guidelines and Best Practices	blog.ndepend.com	2021
GS18	Secure Code Review: How Secure is Your Code? - DataPrivia	dataprivia.com	2022
GS19	Code Review - Open Practice Library	openpracticelibrary.com	2020
GS20	6 code review best practices for a happier codebase and team	educative.io	2022
GS21	Code Review Best Practices - Trisha Gee	trishagee.com	2018
GS22	Best Practices for Code Review and Pull Requests - Updivision	updivision.com	2021
GS23	5 Best Practices For Code Review - GeeksforGeeks	geeksforgeeks.org	2022
GS24	How to improve your code review: tips and best practices	belvo.com	2021
GS25	Investigating the effectiveness of peer code review in ...	jserd.springeropen.com	2018
GS26	Better code, better applications - every time	walkingtree.tech	2022
GS27	How we do it: peer code review - DataMiner Dojo	community.dataminer.services	2022
GS28	Code Review Guidelines for Data Science Teams	tdhopper.com	2021
GS29	Gerrit Code Review Product Overview	gerrit-review.googlesource.com	2022
GS30	Code Review: Best Practices - Waverley Software	waverleysoftware.com	2019
GS31	Code review checklist - Apex Hours	apexhours.com	2021
GS32	The Art of Code Review - Towards Data Sciencehttps	towardsdatascience.com	2021
GS33	Code Review Best Practices - Level Up Coding	levelup.gitconnected.com	2021

Continued on next page

Table 2.7 – continued from previous page

ID	Title	URL	Year
GS34	Best practices code review test automation by Anton Smirnov	itnext.io	2021
GS35	Best Practices for Code Review	dzone.com	2020
GS36	Code Review Best Practices - GitKraken	gitkraken.com	2021
GS37	Best Practices for Effective Code Review	leobit.com	2020
GS38	Best Practices for Effective and Efficient Agile Code Reviews	queue-it.com	2022
GS39	Best Practices for Peer Code Review	kessler.de	2009
GS40	Creating Simple and Effective Guidelines for Code Reviews	newrelic.com	2018
GS41	Best Practices For Code Review - C# Corner	c-sharpcorner.com	2016
GS42	Code Review Best Practices - Programmer Friend	programmerfriend.com	2018
GS43	How rOpenSci uses code review to promote reproducible science	ropensci.org	2017
GS44	Where is the value in package peer review?	ropensci.org	2018
GS45	Recommended C Style and Coding Standards. Pocket reference guide	gnu.org	2005
GS46	ChangeViz materials	doi.org/10.5281/zenodo.5175927	2021
GS47	Ilya Sabanin Contributor to Beanstalk Guides	guides.beanstalkapp.com	2019
GS48	Atlassian Crucible features	atlassian.com	2022
GS49	CodeFlow	getcodeflow.com	2014
GS50	Intel Open Source Technology Center – Patch Review	blog.ffwll.ch	2020
GS51	Apache Spark	spark.apache.org	2020
GS52	Chromium coding style	dev.chromium.org	2014
GS53	Pep 8: style guide for python code	peps.python.org/pep-0008	2001
GS54	10 faulty behaviors of code review	speakerdeck.com	2020
GS55	How we do code review — app center blog	devblogs.microsoft.com	2020
GS56	Code review best practices by Palantir	medium.com	2020
GS57	Pull request best practices - the pragmatic engineer	blog.pragmaticengineer.com	2020
GS58	The wireshark wiki - Development/SubmittingPatches	wiki.wireshark.org	2020
GS59	Sharma S. How to write a good pull request description – and why it's important URL	freecodecamp.org	2020
GS60	What is code review?	smartbear.com	2020
GS61	Code reviews at google are lightweight and fast	michaelagreiler.com	2020
GS62	The 2019 State of Code Review: Trends and Insights into Collaborative Software Development	static1.smartbear.co	2019
GS63	Firefox code review	wiki.mozilla.org	2016

When GUI-based Testing Meets Code Reviews

3

This chapter is based on Andreas Bauer, Tomas Helmfridsson, Emil Alégroth, and Georg-Daniel Schwarz. "*When GUI-based Testing Meets Code Reviews*". Manuscript submitted to *Software Testing, Verification and Reliability*. 2025.

Abstract

Context: Code reviews are a well-established practice to ensure code quality, identify potential bugs, promote knowledge sharing, and maintain coding standards within a team or organization.

Objective: This study aims to investigate the specific practices, challenges, and information needs encountered when reviewing GUI-based testing files for web applications, which remain poorly understood.

Method: We conducted a qualitative interview study with 14 software testing professionals from six different companies to explore the distinct aspects of reviewing GUI-based testing code.

Results: We identified four practices, six challenges, and four information needs related to reviewing GUI-based testing files. The foremost challenge is the validation of GUI-based tests under review. Furthermore, challenges concerning levels of abstraction and test robustness were not addressed in related studies. Additionally, participants proposed six potential improvements for tools and practices to better support the code review process. Notably, the absence of standardized practices and the need to run tests locally were common themes across participants.

Conclusion: The code review process for GUI-based testing differs from that of production code, highlighting the need for practices and tools tailored specifically to the unique demands of GUI-based testing.

3.1 Introduction

Code review is a software engineering practice in which peers review a code contribution before additions or changes are integrated into the code base [20, 53]. This practice is effective in ensuring the quality of products by catching errors and improving sub-optimal solutions on different types of artifacts [18]. Additionally, it enables knowledge and experience sharing and fosters active collaboration within the team [20, 83].

Testing is the most widely used method for ensuring quality in the industry [216], and thus, test code is commonly subject to code review. Whilst the core benefit of reviewing test code is improved test effectiveness, the practice has also been shown to improve the overall quality of requirements and production code thanks to feedback loops in the development process [139, 179]. Defects found during test case reviews can even help localize problematic areas in the software development lifecycle itself [135]. Adversely, not reviewing test code lowers quality and can result in increased maintenance costs [39, 93, 138]. Consequently, there is evidence to suggest the benefits and effectiveness of test code review.

Previous research has identified differences in the review processes for different artifacts, e.g., production code and test code [205]. This stems from the different purposes, structure, content, and reviewer focus during the evaluation of a specific artifact. The review of the quality of test code differs from that of production code. This underscores the importance of tailoring review criteria and strategies to the specific type of artifact under review.

However, to the best of our knowledge, whilst best practices are available for most types of software development artifacts, there is an absence of academic literature that covers practices for review of graphical user interface (GUI) based testing artifacts.

Software testing is practiced at different levels, from low-level unit tests of single functions to high-level tests conducted through the system's GUI (GUI-based). GUI-based testing is a technique in which the verification and validation of a system's behavior are done through interactions with the system's GUI, mirroring user interactions [8, 55]. This approach enables the verification of both the system's visual elements and its underlying functionality [8]. GUI-based testing is commonly employed in the industry to identify GUI regressions, validate information flow throughout the system under test (SUT), and ensure the functionality of various interconnected systems, to ensure the quality and reliability of complex software applications [168].

GUI-based testing can be carried out in different forms, from manual testing to automated script-based approaches, supporting both exploratory and regression testing. In this manuscript, the term "GUI-based testing" refers explicitly to the automated approach to verify a system's behavior, and "GUI-based test files" are script-based (test code) representations of test cases. Popular tools for this form of GUI-based testing include Selenium [103], Playwright [161], and Cypress [59]. Note that this differs from GUI testing, where the purpose is to verify the GUI's visual correctness.

Due to the user-centered approach of GUI-based testing and the engagement of the entire system under test (SUT), it presents several unique challenges that are not found in other types of software testing [168]. For example, it can be difficult to accurately identify GUI widgets, manage variations in screen resolution, and ensure proper synchronization between the tests and the SUT.

GUI-based tests are associated with high maintenance costs [7, 8, 9, 13, 92, 168]. Implementing practices like code reviews could help to reduce some of the costs by ensuring the understandability and maintainability of tests before they are integrated into the codebase. However, a study by Alegroth, Petersen, and Tinnerholm [14] indicates that applying general programming guidelines may not be effective and can even have negative impacts on the maintenance costs of GUI-based tests and require special consideration. These findings suggest that standard practices used in software development, such as code reviews, cannot be easily applied to GUI-based testing. As a result, there is a significant gap in knowledge regarding how to effectively review

GUI-based testing artifacts. Consequently, there is a lack of common best practices in the industry.

The primary research objective of this work is to identify practices used in industry for the review of GUI-based testing artifacts and synthesize these into a set of generally applicable practices. As a secondary objective, we seek to understand the current challenges with these practices that can be subject to future work, and whether these practices are systemic (the same practices are unknowingly used in all companies) or ad hoc (practices differ between companies) in industrial practice.

Whilst GUI-based testing is performed using both manual and automated practices in industry, in this study, we focus on automated GUI-based testing. Hence, testing where tests are represented as test scripts (code) and related artifacts to support the automation. Thus, this work excludes test automation by other means, such as model-based GUI testing, where tests are represented as visual or textual models. Our motivation for this delimitation is that tests represented as code are the most similar to production code, where code reviews are most commonly practiced. Additionally, we limit the scope of our study to the GUI-based testing of web applications, as they are the primary system under test of this type of testing and offer a wide range of available testing tools. Thus, we exclude GUI-based testing of embedded systems, mobile, and desktop applications.

As such, the main contributions of this work are:

1. An overview of practices for reviewing GUI-based test artifacts for web applications in industrial practice.
2. A synthesis of collected practices into a set of general practices for review of GUI-based testing artifacts.
3. A synthesis of known challenges with the review of GUI-based testing artifacts that represent areas of future work.

3.2 Background

3.2.1 Code review

Code reviews, also known as modern code review (MCR), are characterized as informal, tool-supported, and lightweight processes [53]. This contrasts with early code reviews described by Fagan [78], which were formal, waterfall-like processes that were often associated with overhead costs. Peer code reviews were an earlier form of MCR that is, for instance, practiced in open-source software projects like the Apache server project [185]. These reviews were conducted through emails and mailing lists, and they are performed either before or after a contribution (commit) is integrated.

A generic workflow that incorporates MCR, where peers review a code contribution before integration into the code base, is demonstrated in Figure 3.1.

A common practice in modern development workflows is to modify the codebase in separate branches. Once the changes are ready for integration into the main branch, a code review is initiated and assigned to a reviewer, typically as part of a merge or pull request. The reviewer(s) analyze the code under review and provide feedback, prompting rework if necessary. Optionally, the reviewer(s) may also request additional information to better understand the purpose and impact of the changes. As such, the review process can involve multiple iterations until the changes are either accepted or rejected to be merged into the main branch.

Code reviews have been shown to be effective at raising software quality and promoting knowledge sharing [22, 82, 156]. Defects found and resolved during code reviews often enhance the software's understandability and modifiability, rather than its visible functionality [150]. The practice is therefore widely adopted in both industry and open-source projects [20, 27, 61, 156].

Whilst code review can be a completely manual process, it is primarily facilitated by dedicated tools, such as Gerrit [100], or integrated code review functionality of software platforms, such as GitHub [124] or GitLab [101]. On these software platforms, in particular, code reviews are an integral feature of the collaborative software development workflows. This feature aids developers by highlighting changes to the code and visualizing added, modified, and removed lines of source code compared to the main version in the codebase (see Figure 3.2). However, despite its effectiveness, these features are limited to low-level source code. As such, code review features may not adequately address complexities emerging when code that validates a system's behavior, i.e., GUI-based testing artifacts, is under review.

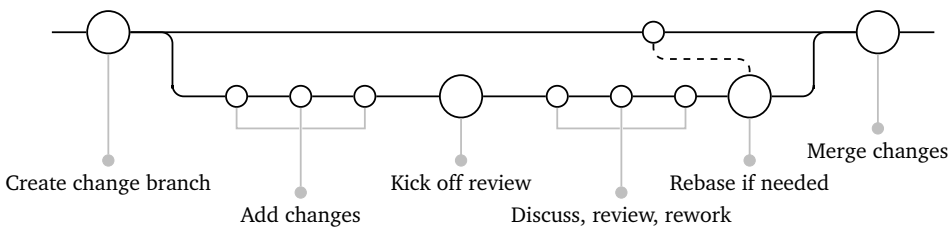

Figure 3.1: Development workflow of integrating changes with code reviews

3.2.2 Test cases for GUI-based testing

A GUI test case consists of two parts: (a) an initial GUI state in which the test case is executed, and (b) a sequence of events that serve as the test input to the GUI [157].

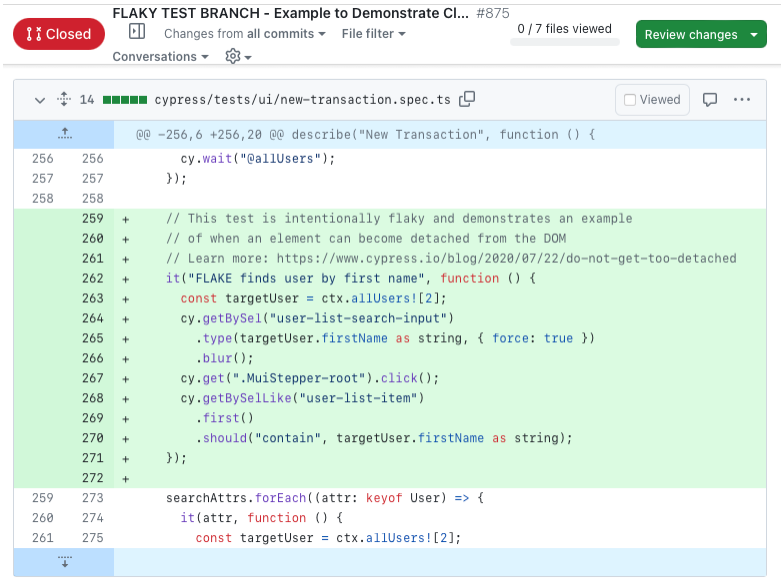

Figure 3.2: Diff of changed lines of code presented in a Pull Request on GitHub (PR #875)

Listing 3.1 displays an excerpt from a test case intended to validate the login functionality of an application. The initial SUT and its GUI state (not shown in the listing) are set up so that the user does not already exist. Subsequently, individual events are performed to interact with the GUI and create a new user through a form present on the GUI.

Testing frameworks utilize locators to identify and target elements on the GUI. For web-based applications, the document object model (DOM) is employed to target specific elements using HTML tags, identifiers, and CSS properties. In Listing 3.1, elements are located using `cy.get(...)`, which provides a loose coupling between the test code and the GUI of the SUT. Modifications made to the GUI can lead to failing test cases, necessitating maintenance efforts from testers [57].

Assertions are performed using `.should(...)`. Typically, test cases assert whether a GUI element is present, visible, or equal to a specific value.

However, the context of the GUI makes it difficult to determine the exact interactions with it without prior knowledge.

```
// description: should add a user
cy.get('[data-e2e="create-user-button"]')
  .should('be.visible').click()
cy.url().should('contain', 'users/create')
cy.get('[formcontrolname="email"]')
  .should('be.enabled').type('dummy@dummy.com')

cy.get('[formcontrolname="userName"]')
  .should('be.enabled').type(user.addName)
cy.get('[formcontrolname="firstName"]')
  .should('be.enabled')
  .type('e2ehumanfirstname')
cy.get('[formcontrolname="lastName"]')
  .should('be.enabled')
  .type('e2ehumanlastname')
cy.get('mat-select[data-cy="country-code"]')
  .click()
cy.contains('mat-option', 'Switzerland')
  .scrollIntoView().click()
cy.get('[formcontrolname="phone"]')
  .should('be.enabled').type('123456789')
cy.get('[data-e2e="create-button"]')
  .click({ force: true })
cy.shouldConfirmSuccess()
let loginName = user.addName
if (user.mustBeDomain) {
  loginName = loginname(user.addName, Cypress.env('ORGANIZATION'))
}
cy.contains('[data-e2e="copy-loginname"]', loginName).click()
cy.clipboardMatches(loginName)
```

Listing 3.1: Excerpt from an actual test case asserting the login functionality using Cypress as the testing framework. Source: humans.cy.ts

3.3 Related work

The most relevant work for our study is a study conducted by Spadini et al. [205], which investigates whether and how developers review test files during code reviews. In the study, 12 expert interviews were performed that were analyzed using both quantitative and qualitative analysis. One of the study's main findings is that reviewing test files differs from reviewing production files. The study also concludes that when reviewers review test files, they focus on test-specific aspects, such as better testing practices, identifying tested and untested paths, and evaluating assertions within the test code.

Despite the importance of ensuring the quality of test files [39, 138], the study finds that test code files are almost twice as likely to be overlooked compared to production code. This discrepancy in review effort between artifacts highlights a gap in how code reviews are adopted and an area of potential improvement to improve overall

software quality. Furthermore, the study explains the lack of test code reviews from a set of challenges. One of these primary challenges is a lack of navigation capabilities within the code review tools, which hinders thorough examination.

However, whilst the study goes into depth into the unique aspects of test code review, the study does not address the particularities of reviewing files for GUI-based testing. As such, there is a lack of clarity on whether the findings can be applied to this specific type of testing.

Another study of interest comes from Pascarella et al. [178], who investigated the specific information needs of reviewers during code review discussions. The study's findings resulted in a taxonomy of reviewers' information needs, consisting of seven distinct high-level categories. The most commonly observed information needs for the code under review concern the suitability of an alternative solution (N1), correct understanding of its functionality (N2), rationale for the change (N3), and the code context (N4). As such, fulfilling these information needs during code review has positive effects on the efficiency of the review process, since it reduces the time reviewers are required to spend collecting this information for themselves. Pascarella et al.'s study is particularly relevant as a benchmark for the categorization of our findings on missing information for reviewing files for GUI-based testing. By utilizing their classification, we can better position our findings within the broader context of code reviews and enable better replicability and analysis of the results.

Dong et al. [66] conducted a comprehensive review of both white and grey literature, leading to the identification of 57 general practices and 19 common pains associated with code reviews. For example, "provide positive comments", "pick the right people to review", or "simplify code changes". Practices explicitly applicable to code artifacts include, "perform static code analysis", "test before submission", or "use automated tools (build and test)". However, despite their perceived relevance and generalizability, it is unclear how these practices apply to the specific needs and challenges associated with reviewing higher-level test code, such as GUI-based tests.

This lack of clarity is underpinned by Söderberg et al. [203], who highlights that a "one size fits all" approach to code reviews causes problems in terms of tooling and the review workflow. In their study, they also identified a mismatch between the scope of data presented to the reviewer (Unit of Analysis) and the scope of the system that the reviewer needs to attend to (Unit of Attention) to perform the review. We assume that this mismatch becomes more pronounced for GUI tests since GUI-based tests operate on a higher level of abstraction that encapsulates the system's behavior rather than individual code components. Inferring that reviewers must understand the broader context and interactions within the system under test (SUT) to provide valid and valuable feedback on the quality and correctness of the test code.

In conclusion, related work has established code reviews as a powerful practice, also for test code. The area is well studied, as evident by the systematic literature

review by Davila and Nunes [61] and systematic mapping studies by Wang et al. [220] and Badampudi, Unterkalmsteiner, and Britto [21]. However, whilst the body of research covers code review practices, there are no or limited insights into the specific challenges and practices related to the review of GUI-based testing artifacts. This omission highlights a gap in the current understanding that necessitates further investigation.

3.4 Study design

The goal of this study is to cover the gap in knowledge of practices, challenges, and information needs when performing code reviews of GUI-based testing artifacts, e.g., test code and associated files.

The study's goal has been broken down into the following research questions.

> **RQ1: Which practices do reviewers in industry use for reviewing GUI-based test artifacts?**

This research question seeks to identify the practices industrial practitioners use when reviewing GUI-based test artifacts and whether these practices are explicit and common among companies (systemic) or implicit and differ between companies (ad hoc). Given that reviews of test code are reported to be less frequent than reviews of production code [205], we also explored practitioners' attitudes towards code reviews, i.e., if they are positively or negatively oriented towards the practice.

> **RQ2: What challenges do developers/testers in industry face when reviewing GUI-based test artifacts?**

This research question aims to identify the problems and challenges developers encounter when reviewing GUI-based test artifacts. By understanding these challenges, we seek to identify shortcomings in current tools and practices that may be subject to further development and/or research.

> **RQ3: What are the information needs when reviewing GUI-based test artifacts in industry?**

This question seeks to provide insights into the needs, including information needs, of the reviewers to perform efficacious code reviews. Hence, what information should test developers append to their test code to facilitate better code reviews? Additionally, we perceive that such information needs provide insights into which and how to improve tools and practices for review of the GUI-based test artifacts in the future.

> **RQ4: What improvements do practitioners envision that can be adopted in future tools and practices for review of GUI-based test artifacts?**

This research question aims to gather practitioners' perspectives on potential improvements for future tools and practices in the context of code reviews. By exploring their insights, we seek to understand whether there is a perceived need for enhancements in the current tooling and practices or if practitioners are generally satisfied with the existing solutions despite the existing challenges.

3.4.1 Research method

To address our research questions, we conduct a qualitative interview study [160] with software testing professionals, henceforth referred to as practitioners. Conducting interviews is an effective method for gathering firsthand and in-depth information about a phenomenon [197]. The findings of our study provide input to the formulation of generalized best practices for the review of GUI-based test artifacts. In addition, the results provide valuable insights for the development of methods and tools to support code reviews of GUI-based tests.

Design

Due to the descriptive nature of this research, we chose to use semi-structured interviews as the primary means of data elicitation. This method is suitable since it allows for exploration and in-depth elicitation of emerging topics during an interview. The interviews were driven by an interview guide, following the recommendations by Seaman [197]. To structure the interview, the interview guide included the following steps:

Informed consent: Before the interview, we explained the confidential handling of responses and obtained explicit consent to record each interview for transcription. We also clarified the study's purpose and defined key terms.

Background/demographic information: We began with questions about participants' industrial experience, including their specific experience in development, testing, and code review practices.

Questions: The main questions of the interview, organized by areas of interest: code review practices, challenges, information needs, and future tools/practice improvements.

Backup questions: If a participant did not have experience with code reviews, what alternative methods do they use for ensuring code quality?

Closing question: Each interview was concluded with an open-ended question where the participant could complement their answers or give additional information not covered by the interview questions.

The complete interview guide with all questions is available in 3.8.

Table 3.1: Interview question in relation to related work

Our interview questions		Interview questions from related work
5	What is the importance of reviewing files for GUI-based testing?	What is the importance of reviewing these files? [205]
6	How do you conduct code reviews?	How do you conduct reviews? [205]
6.a	Are you following any explicit or implicit practice?	Do you have specific practices? [205]
6.b	Are these practices different for reviewing files for GUI-based testing compared to production code?	What are the differences between reviewing test files and production files? [205]
7	What challenges do you face when reviewing files for GUI-based testing?	What challenges do you face when reviewing test files? What are your needs related to this activity? [205]
8	Do you think current code review tools support your information needs (information required to understand a proposed change) when reviewing files for GUI-based testing?	Do you think current code review tools support this need? [178]
9	How would you improve the current tools or practices you use to review files for GUI-based testing?	How would you improve current tools? [178]

Interview questions To contextualize our results and situate them with the existing body of knowledge on test code reviews, the interview questions were aligned with those of Spadini et al. [205] and Pascarella et al. [178]. From Spadini et al. [205], we adapted questions related to code review practices and the differences between the review of production and test code. From Pascarella et al. [178], we adapted questions about information needs during code reviews and tool improvements. Note that the purpose of our study was not to replicate the results from previous work, but rather that we drew inspiration from earlier studies to improve and situate our results. For an overview of the differences between our questions and those from related work, see Table 3.1.

Participants To acquire a suitable sample of interviewees with experience in reviewing GUI-based testing artifacts, we reached out to companies and individuals within our industry-academia collaboration network. Contact was sought via email, in which we explained our preferences for interviewees, a summary of the study's goals, the estimated duration of the interviews, and a list of possible time slots for participation.

To expand the sample further, we also asked participants to recommend others who might be suitable for our study at the end of each interview.

Despite the established sampling parameters, we categorize our sampling as convenience sampling. This approach means that only engineers who specialize in GUI-based testing are included in our study, excluding those who occasionally perform GUI-based testing or other types of testing. Further, participation was self-selected. On one hand, this creates a bias that fosters an overly positive view of GUI-based testing, especially regarding the perceived importance of reviewing these tests compared to other types of artifacts. On the other hand, gathering insights from specialists in GUI-based testing can help uncover the specific practices, challenges, and needs related to reviewing GUI-based testing files, as these topics are not adequately covered in the existing literature.

Qualitative data gathering

The interviews were conducted online using Microsoft Teams, utilizing its integrated recording capabilities for later transcription and subsequent analysis. Interviews were carried out individually with the participants by the first author following the interview guide. Given the narrow focus on code review practices for GUI-based testing and the ambition to synthesize practices from a larger sample, we limited the interview duration to 30 minutes. Thus, limiting the effort required by the industrial practitioners to participate in the study and thereby increasing the potential number of participants that would be willing to engage in the study.

Descriptive data about the companies and participants In total, we interviewed 14 participants from six companies. Table 3.2 presents an overview of the participants, including the company affiliations, their role at each company, and their industrial experience. All participants were primarily involved in testing the GUI of web-based applications, rather than other types, such as those for desktop or mobile applications. Having participants from multiple companies helped ensure that the acquired data is more representative of a more diverse set of contexts and domains, and it was necessary to investigate whether practices are systemic or ad hoc. This analysis was further facilitated by eight of the 14 participants being consultants with experience from several companies and contexts, combining insights from multiple industries, including telecommunications, automotive, finance/e-commerce, insurance, and transportation systems.

Collectively, the participants covered a range of widely used testing and test automation tools for GUI-based testing of web applications, including Selenium [103], Playwright [161], Cypress [59], and Robot Framework [186], reducing bias by focusing on challenges that correlate with only one specific tool.

Table 3.2: Overview of participants

			Industrial experience in years		
ID	**Company**	**Role**	**total**	**dev/tester**	**code review**
P1	C1	Software architect	6	6	6
P2	C1	Developer	7	7	7
P3	C2	Developer / QA consultant	15	12	7
P4	C2	Developer / QA consultant	16	16	9
P5	C2	Tester / QA consultant	11	11	7
P6	C2	QA consultant	36	36	3
P7	C2	Developer / tester	29	29	24
P8	C2	QA engineer / consultant	13	13	5
P9	C3	QA consultant	24	24	22
P10	C4	Developer / QA consultant	10	10	6
P11	C5	QA consultant	17	15	7
P12	C6	Test / infrastructure engineer	5	4	2
P13	C6	Test engineer / lead	19	4	4
P14	C6	Test engineer / developer	25	15	0

Ericsson (C1) is a leading Swedish multinational telecommunications company specializing in networking, telecommunications equipment, and digital services.

Qestit (C2) is a large European software consultancy company specializing in software testing and IT security.

Iceberry (C3) is a specialized IT consulting firm that assists businesses in growth through software development, testing, and quality assurance services.

Zington-VASS (C4) is a large European software consulting company specializing in system development, architecture, and quality assurance solutions.

Company C5 is a freelancer specializing in software testing consulting.

Company C6 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.

Qualitative data analysis

In this section, we describe how we prepared and conducted the analysis of interviews.

Transcription of interviews We initially transcribed the interview recordings using Microsoft Teams' built-in live transcription functionality. The first author then manually reviewed each transcription for accuracy and corrected any transcription errors. Additionally, transcripts were anonymized and cleaned to remove filler words and repetitions. Care was taken at this step not to modify the semantic meaning of the interviewees' statements.

Following the guidelines by Braun and Clarke [43], we conducted a thematic analysis including the six outlined phases as described below. Thematic analysis was chosen for its suitability for qualitative data analysis due to its flexibility and ability to generate unanticipated insights while summarizing key findings in the data [43].

Phase 1: familiarization with the data Familiarization began with a manual review and cleanup of the transcriptions. Additionally, the cleaned transcriptions were read before coding to familiarize the researchers with their contents.

Phase 2: generate initial codes Coding was performed systematically by assigning text segments of interest with preliminary codes. This process was performed in parallel with the interviews, meaning that new codes were incrementally added as more interviews were performed. To consolidate the codes, old interviews were revisited and recoded as needed. The coding was performed using the open-source Taguette [182] tool that helps ensure the traceability of emerging codes and themes to their sources.

Phase 3: search for themes Preliminary codes were analyzed and combined into broader themes based on conceptual or semantic similarity. A theme captures "something important about the data in relation to the research question, and represents some level of patterned response or meaning within the data set" [43]. Examples of themes in our study include "review practices," "challenges," and "information needs", which reflect our research questions. The analysis of the relationship between themes resulted in a hierarchical arrangement of themes, with overarching themes consisting of more specialized sub-themes.

Phase 4: review themes To test the validity of the codes and how well they represented the data set, three additional coders reviewed a subset of the interviews and provided feedback. The review was carried out by the reviewers applying the codes to the interviews. More information about the quality assurance process is presented below. Based on the feedback of the review, several issues of ambiguity were resolved in the themes and codes.

Phase 5: define and name themes In this phase, we finalized the naming of the themes and preliminary codes. We defined the scope and content of each theme by

providing descriptions, examples of when to use and not to use them, and examples of how the codes should be applied. The first and another author then used the finalized code system to recode all interviews. Whilst not considered strictly necessary, the re-coding was deemed prudent to ensure consistency throughout the dataset. The final coding system comprises three high-level themes encompassing 22 distinct codes.

Phase 6: produce the report Apart from reporting our findings in this manuscript, we published an artifact package [28] on Zenodo to enable researchers to validate our analysis. This package includes a summary of all codes and themes, the codebook (codes and their descriptions), anonymized interviews, and an export from the tool to import the interviews with applied codes.

Quality assurance of interview analysis The first author created the initial set of themes. Three additional coders conducted coding on a total of six randomly assigned interviews over two iterations. After each iteration, an intercoder session was performed where codes applied by the other coders were compared to the first author's coding and discussed in a meeting, resulting in an updated version of the code system. Any deviations in the coding were discussed to clarify reasoning and determine whether themes required merging or splitting into sub-themes. For instance, codes that were originally labeled "missingInfo" were changed to "infoNeeds" to highlight the necessity for the information rather than simply indicating its absence. The single theme "challenges" becomes a top-level theme, consolidating various codes that were previously related to tests or tools. If there were ambiguities regarding the application of codes, the codebook was updated with clearer descriptions and additional examples of when to apply the codes. For instance, the code distinguishing explicitly defined from implicitly performed practices was initially applied too broadly, encompassing subjective interpretations of implicitness and explicitness. Other coders noted that the criteria for application of the related code was not clear enough. The revised codebook now specifies that this code should only be applied when a participant explicitly states that a practice is implicit or explicit. Table 3.3 illustrates the changes in the code system after each inter-coder session, highlighting the convergence towards a common understanding as evidenced by the reduced number of changes after the second session. All versions of the code system and the differences between them are represented in the codebook that is provided in the artifact package [28].

Finally, once all interviews were coded by two coders using the final code system, an intercoder session was conducted between the two coders. During this session, the final coding of text segments was discussed by reviewing all differences in coding, where disagreements were resolved through discussion until a consensus was reached on which codes fit the text segment. Many of the differences in coding can be attributed to coded text segments overlapping with the next paragraph (the chosen window size).

We decided not to employ a statistical method to calculate intercoder reliability as a numerical measure of agreement [154, 173]. Instead, we utilized an iterative process where two coders resolved differences through discussion to enhance the coding system. By having two authors code all interviews and then finalize the coding, we are confident in the objectivity and reliability of the results obtained. For transparency, the artifact package [28] includes all iterations of the code system and a comparison of the coding between the first author and the reviewers.

Table 3.3: Inter-coder session changes of the code system

Code system changes	Session 1	Session 2
Added codes	4	4
Changed codes	12	1
Removed codes	5	0
Sum	**21**	**5**

3.5 Results

This section presents the results addressing our research questions.

3.5.1 RQ1. Which practices do reviewers in industry use for reviewing GUI-based test artifacts?

Code reviews provide many benefits, such as improving code quality and sharing knowledge within a team. However, during reviews, test files are less likely to be discussed, and reviewers often perceive reviewing test files as less important than reviewing production code [205]. Therefore, we started by assessing the participants' attitudes toward code reviews for GUI-based testing to ascertain whether testers consider it to be an important and valuable practice. This assessment provides a deeper understanding of the reported practices and challenges, i.e., why certain practices are employed and/or the challenges connected to them.

RQ1 is addressed by first summarizing the reported practices used by practitioners for code reviews of GUI-based test artifacts. These practices are then synthesized, and the derived list of practices is analyzed to determine if they are systemic or ad-hoc solutions.

Importance of reviewing files for GUI-based testing Of the 14 participants, 13 participants consider the review of GUI-based tests and related test files to be important, or more important, than production code. They argue that since the tests are commonly represented as code (in their experience), the tests should be quality-assured

with the same rigorous review practices as production code. This point is argued from the perspective that tests, just like code, serve a purpose for the product that influences its quality and validity. As stated by participant P8, "*Whether it's validating feature code with test automation or the feature code itself, it [the code] still serves a purpose that needs to be validated and examined through both.*" Thus, highlighting the common value-provision of both production and test code.

Only one participant considered review of tests as less important than production code and argued that the impact of sub-par tests is less on customer-facing software compared to the production code. The interviewee assumed that lower-quality test code is less likely to affect the software experience for customers compared to lower-quality production code.

Are there any explicitly defined practices that the participants follow None of the participants mentioned following any explicitly defined practices for reviewing GUI-based testing files. While some participants mentioned guidelines related to the management of reviews, like requiring a minimum number of approvals for each review, there were no explicit guidelines that would support or structure the review process. Consequently, the thoroughness and execution of the review practices depend on individuals or teams, resulting in ad-hoc solutions. Thus, highlighting that reviews of GUI-based test artifacts are heterogeneous, even within teams. While heterogeneous practices of *how* to review the tests may not be a hindrance, the absence of standardized guidelines regarding *what*, *when*, and *why* to review test code represents a potential limitation.

Guidelines for software engineering tasks have mixed efficacy. On the one side, guidelines can promote consistency within the codebase, positively impacting its maintainability [148, 219]. On the other side, guidelines can have detrimental effects on maintenance costs due to the cognitive load they introduce during development or the trade-offs they entail, such as modularity vs readability [14]. Therefore, guidelines should not be imposed on practitioners but rather considered as suggestions for engineering practices.

Further research is needed to evaluate the efficacy of specific guidelines in the context of code reviews for GUI-based testing files. We believe it is important to explore potential guidelines, especially because GUI-based testing files are created and reviewed less often, making it difficult to rely on sufficient expertise within a team.

Running tests on the local machine All participants stated that they run GUI-based tests on their local machines during reviews. Although not every participant's role is responsible for running tests locally, they felt compelled to do so to circumvent limitations in the code review tools to provide feedback on the tests. Several reasons were provided for this practice. Firstly, local testing assures that the tests are working as expected and can run in different environments. Verifying that tests can be migrated

Table 3.4: Overview of code review practices (RQ1)

Practice	Support
Running tests on the local machine	all participants
Side-by-side views	P1, P4, P7, P13
Commit-by-commit analysis	P3, P8
Reading only the diff	P10

between environments is necessary since GUI tests are created on the test developers' local environments. However, they are often run in test environments as part of a continuous integration pipeline (CI) [38]. Thus, if the tests can be run on the reviewer's own local environment, it is assumed that the tests can be migrated. Faults that can be identified this way include accidental local dependencies or configurations. This challenge was explicitly stated by P10: "*Especially with larger changes [of tests], you might have accidentally [introduced] some dependencies on things that are on your [hard drive]; hardware-related or environment-related.*"

Secondly, reviewing the tests locally allows the reviewer to use their own integrated development environment (IDE). The participants stated that IDEs provide functionalities to overview, perform deeper reviews, and debug the tests. Otherwise, reviewers must rely on the information presented in the code review tool, which was reported to often lack GUI-based testing relevant information, like detailed test coverage information. The practitioners explicitly mentioned that for large changes to tests or test suites, the ability to easily navigate or preview related files and functionalities is necessary to comprehend, especially larger changes. Findings by Spadini et al. [205] also highlight the need for better navigation capabilities between test and production files in code review tools.

As participant P5 described: "*But for very large changes [of tests], the diff [between old and new test] becomes meaningless because it's so hard to get an overview of it anyway. So it's better to pull it [the test or test suite] down [to your machine] and just work through it locally.*"

Finally, by running the tests locally, the reviewers can see the tests' interactions with the SUT's GUI. This allows the reviewer to experiment with the tests to provide alternatives or improvements. While such improvements may be provided without running the tests, verifying their correctness can only be achieved through running the tests. Thereby making test execution a necessity.

Side-by-side views An extension of the "running test locally" practice is to have the code review tool window open next to the SUT's GUI such that the test execution can be followed, line by line, during execution. "*I don't see any other solution than to have*

the test cases on one screen [during execution/review]. The application is on the middle screen, and the test code is on the third screen. I don't see any other way.", as mentioned by participant P4. This practice highlights an important distinction between the review of production code and GUI-based testing artifacts. Hence, for GUI-based tests, reviewers need to follow the test code in real-time during test execution instead of using breakpoints in the code or logs. This is necessary since the tests need to align with the tested system's functional and chronological behavior. However, as the tests run asynchronously, only synchronized by static- or dynamic waits (statements used in most tools for synchronization), faulty behavior is best seen through observation.

Commit-by-commit analysis Another practice mentioned by the practitioners involves reading through the change history (pull/merge request) of the SUT before reviewing any new or changed tests. When commit messages are used effectively, they not only describe what has been changed but also explain the purpose and design decisions of the change. Thereby providing additional contextual information that fosters an understanding of the introduced changes in the test cases and the SUT. Participant P3 explained: "*Some people pay great attention to what the commit log looks like, in terms of commit messages and what has been done and what order. Having the [git] commit history tells a story of how the product has been developed.*"

Reading only the diff Another practice is to only review the test in the code review tool without taking any further action. There are two reasons for this: first, if the test code changes are minor, such that validity can be easily determined, and second, if the reviewer has extensive knowledge about the domain, such that they can ensure test validity without further action. However, only one participant provided support for the second reason, i.e., that having extensive domain knowledge is sufficient to understand changes. Thus, while results indicate this practice to be possible and efficient in terms of time, it is also associated with risks that will be discussed further in Section 3.5.2.

Aspects of interest during review When reviewing GUI-based testing artifacts, participants examine multiple aspects to verify that the test provides coverage and that it is of sufficient quality. Similar to production code, reviewers are interested in the **readability and understandability** of the test cases are paramount, as these attributes contribute to the general maintainability of test cases.

One factor specific to test code is understanding the **test flow**, which should ideally encompass an end-to-end scenario, detailing which parts/functions/features of the SUT are being tested. Participant P1 stated, "*I would say that if I were to review a GUI test, I think the primary thing I would be looking at is the end-to-end flow of the test, i.e., what parts of the GUI have you covered.*" As input for such a review, the reviewer would consult the requirements, e.g., use cases, or use their own domain knowledge.

Another important factor is to ensure that test cases align with **coding standards and styles** if such standards are established within the project or organization. Automated static code analysis tools are used to ensure consistency of production and test code. At a higher level, participants aim to maintain consistency in the way they implement end-to-end testing scenarios across different projects. Participant P3 explained "*Code review is an important place to make sure that the code, across the repository, is consistent.*"

The **scope of test cases** is another factor to consider, with a preference for tests that are designed to assess one specific feature or specific functionality, reflecting a single-responsibility principle. While end-to-end test scenarios may require interaction with multiple SUT features in a scenario, the test objective should still be focused. Thus, feedback shall be given from the review if the scenario touches upon unrelated or otherwise unnecessary functionality.

Finally, ensuring that test cases **conform to their defined requirements** is essential and can be effectively achieved through a robust traceability mechanism that links each test case back to its respective requirement. Implementing and maintaining this traceability across all levels of abstraction improved the consistency and alignment with project goals.

Difference between reviewing files for GUI-based testing and production code

Of the 14 participants, six stated that they approach code reviews of artifacts for GUI-based testing in the same way as for production code. Participants explained that since GUI-based tests are often represented as code they can and should be treated the same. For example, the test code should also follow practices for good software development, e.g., follow coding standards, as previously mentioned.

However, while the participants mentioned several practices common to the review of production code, they also highlighted the need for specific practices. One aspect to consider is which **test data set** is used for a particular test, as it impacts the test's ability to cover functionality and edge cases. Especially if data sets are available in multiple sizes and versions. Another aspect is **test-specific design patterns**, such as the use of the page-object pattern, which promotes code reuse and maintainability. Hence, it is suitable to place additional knowledge requirements of such patterns and constructs on both the test developers and reviewers.

Participants also emphasized the importance of **test oracles**, which are mechanisms to determine whether a test has passed or failed; for instance, verifying that the output matches the expected results. Utilizing requirements (e.g., acceptance criteria) and expert knowledge (e.g., colleagues with domain expertise) as inputs can help improve the chosen oracles. Regardless, it is suitable to place additional knowledge requirements, both technical and domain knowledge, on developers and reviewers of the tests.

The **setup and environment for tests** are also critical, as discrepancies between the test and production environments can lead to unreliable test outcomes. These preconditions shall be listed in the tests themselves, thereby mitigating the need for domain and technical knowledge from the reviewer to set up the tests correctly.

Handling credentials, such as passwords or access tokens, in test cases that are used for authentication tasks shall be handled using suitable guidelines and mechanisms. Such management of credentials mitigates the risk of security being compromised, especially in larger organizations where tests are performed by multiple teams. Depending on the context, such guidelines and mechanisms shall be local or distributed and supported by suitable security training.

Finally, while the **performance of test case execution** is considered, it is not considered as critical as the performance of production code, given that GUI-based tests are seldom used to evaluate SUT performance. It is, however, a suitable practice, as part of the review where tests are executed, to note any performance issues and report these as part of the review feedback. This aspect shall especially be considered for tests that require long waiting times for synchronization, since these can be a source of failure and/or unnecessary cost, i.e., unnecessary waiting time.

Summary RQ1: The interviewed practitioners do not follow specific practices for reviewing artifacts for GUI-based testing. As a result, the applied practices are ad-hoc in nature. Furthermore, in contrast to related work [205], participants consider the review of test files to be as important as for production code. Many participants reported that they approach code reviews of files for GUI-based testing in the same way as for production code, as GUI-based tests are often represented as code. Specific concerns during code reviews include test data, test-specific design patterns, test oracles, setup and test environments, an emphasis on credentials handling, and less emphasis on the performance of test case execution. An overview of reported practices is provided in Table 3.4.

3.5.2 RQ2. What challenges do developers/testers in industry face when reviewing GUI-based test artifacts?

In this section, we present the challenges reported by the participants during code reviews of artifacts for GUI-based testing.

Ensuring correct validation Ensuring that test cases under code review target the correct elements on the system's GUI and, thus, cover the underlying functionality presents notable challenges. Below, we present the potential causes of such challenges.

First, understanding the relationship between code for the GUI, tests, and the SUT is notably difficult when different technologies are used. For example, web interfaces

Table 3.5: Overview of challenges during code review (RQ2)

Challenge	Support
Ensuring correct validation	P1, P2, P5, P9, P10, P11
Understandability and readability	P3, P4, P6, P7
Levels of abstraction	P2, P4, P5, P11
Lack of testing experience	P2, P5, P7
Test robustness	P7, P8, P9, P12
Test maintenance	P8, P9, P11, P12, P13

are typically created using HTML and JavaScript, while the underlying backend functionality may be coded in Java, Python, C++, or other languages. In addition, test code can be represented by any of those languages or a domain-specific language, like Gherkin [201]. Effectively reviewing and validating GUI-based test cases, therefore, requires a good understanding of a project's incorporated technologies.

Next, the locators used to identify elements of the GUI and drive automated tests may be difficult for testers to interpret. For instance, when using XPath [50] expressions for HTML-based GUIs, it may not be clear which element of the GUI an action will be performed on, or which element is used for verification. Thus, necessitating the need to observe the test(s) in runtime. Participant P2 mentioned this challenge as follows, "*Just because the text says 'click on the button', it might not click on the button, it might click on the link [associated with the button].*" This challenge underscores the importance of having in-depth knowledge of the testing framework and what the framework's features actually do. In particular, in testing frameworks that include multiple ways to interact with or to assert the SUT, e.g., Selenium allows the user to interact with elements using absolute XPaths, relative XPaths, element IDs, and more. These different modes of interaction have different benefits and drawbacks in terms of understandability, maintenance cost, but also reliability.

Furthermore, gaining an understanding of the overall purpose and assessing the impact of code changes on the GUI-based tests can be difficult based on only the diff presented in the code review tool. Participant P9 said: "*I would say to get the whole picture would be one thing. Basically, or specifically when it's about these [code] diffs, it's hard to understand the coverage [of the test code]*". The challenge stems from the disconnect between the SUT code and the resulting behavior of the SUT, as observed through the GUI. Small changes, e.g., changing an expression, may redirect a scenario completely, while large changes, e.g., refactoring the SUT to improve its performance, may have no behavioral effect at all. This challenge highlights the importance of having both domain and technical knowledge to properly interpret the diff and assess the full impact of the changes without further measurements or observation. Even

so, just reviewing the code may not be sufficient to (1) understand the impact of the change on the SUT and (2) if the changes have an effect on the test's correctness and ability to test the SUT.

In conclusion, testers face challenges in understanding *the relationship between code (GUI, tests, and SUT)*, *locators used to identify GUI elements*, and the *overall purpose and impact based on code diffs* when reviewing artifacts for GUI-based testing. These challenges may not be found in code reviews for production code. Further, code reviews can not be performed in isolation without technical knowledge and domain knowledge of the SUT.

Understandability and readability Understanding the test flow—the sequence of steps and interactions that a tester or automated testing tool performs to validate a SUT's behavior against its specification—is reported to be challenging during a code review and is one of the main concerns of reviewers.

The factors mentioned by participants that negatively impact the ability to understand the test flow during a review are cryptic identifiers and large test cases.

The naming of variables and functions in test code, as well as identifiers in the SUT production code, plays an important role in understanding test cases. When identifiers of GUI elements, functions, and variables do not have descriptive names, it becomes difficult for reviewers to understand which elements are being targeted by locators in the test case. This issue is further complicated by the fact that while testers define locators, it is the SUT's developers who create the identifiers for GUI elements. Poor communication between testers and developers can result in unusable identifiers, making the test cases harder to interpret and maintain. This challenge is particularly pronounced in web-based GUIs, where developers rarely set meaningful identifiers (ID attributes), as noted by Nass et al. [169]. In the absence of reliable IDs, testers often resort to using XPath expressions as locators, which are not inherently human-readable.

Large test cases are harder to review than smaller test cases. This aligns with the general Clean Code principles [147, 151] and guidelines for reviewing production code [32]. Keeping track of many test steps in larger test cases can increase the cognitive load of the reviewer. Maintaining the logical and chronological order of the test steps becomes challenging. Especially when the locators are not intuitively understandable, as mentioned earlier.

The participants emphasized that ensuring the readability of test cases allows for a more efficient review process. For example, participant P7 mentioned, "*When it comes to efficiency, I think that the most important thing is how easy it is to actually start reading someone else's code and understand what it is doing.*"

Levels of abstraction Dealing with abstractions in test cases during code reviews was reported as challenging. Our participants' reports are inconclusive on whether more or less abstraction in test cases is preferable. While the appropriate level of abstraction in test cases remains debated, the methodologies and frameworks used in testing influence these abstractions. Traditionally, test cases were structured as a simple sequence of test steps. However, current testing frameworks and methodologies introduce more layers of abstraction. Techniques such as the page object pattern [143] facilitate interactions with web pages through model representations in test cases. Testing frameworks like Playwright [161] provide methods for interaction and validation. For example, the `locator.click()` method in Playwright performs a range of additional checks, such as ensuring the locator resolves to exactly one element, the element is visible, not in animation, and enabled. In these cases, abstraction is inherently promoted.

Additionally, structuring test cases with a focus on modularity and reusability can lead to higher levels of abstraction, allowing the same test logic to be applied across different parts of the codebase.

Although abstractions in test code simplify functionality, they can obscure the underlying mechanisms, making it harder to review whether the test code correctly validates the system under test's behavior.

Our participants reported that they would prefer to have more insights into the actual test code instead of relying on the provided abstraction. For example, the Robot Framework [186] abstracts test and automation code to a human-friendly syntax, resembling natural language text. Participant P2 said regarding these types of testing frameworks, "*In those sorts of frameworks, it would have been a huge improvement to be able to see what's actually happening. I would rather read the actual code compared to these abstract tests of the text*".

Another aspect of abstraction reported by our participants is mixing different levels of abstraction in one test case or file. For instance, while some test cases may correctly implement the page object pattern, other parts deviate from this approach by directly accessing elements, thus breaking the intended abstraction. Inconsistent levels of abstractions have a negative impact on the reading flow. For example, participant P11 said, "*But if you have: search item, putting into basket, go to check out and then you have like click element XYZ. Those are two very different levels of abstraction, and you shouldn't mix those*". Listing 3.2 demonstrates the issue of mixed levels of abstraction.

```
1 driver = webdriver.Chrome()
2 page = login_page(driver)
3 page.authenticate("admin","1234")
4 # Next line causes an abstraction mismatch
5 findBy("#react-select-2-option-0-0").click()
6 page.logout()
```

Listing 3.2: Example of breaking the level of abstraction by using a low-level *element.click(...)* command (line 5) next to higher-level functions.

On the contrary, some participants reported that a certain level of abstraction is necessary to understand test cases and to improve their maintainability over time. It can be easier to understand the purpose of a test case or step on a higher level of abstraction, especially for testers with less domain or technical knowledge. Participant P11 said, "*It makes the code more difficult to follow. For example, 'put item in basket' is a lot more understandable for someone coming in and trying to understand the code than 'click element X'.*"

Finally, reusable test code could help to avoid inconsistencies and duplicated code in the code base, as mentioned by participant P5, "*But when [the code base] grows, maybe you want to abstract it further and have a shared library instead, between teams. Because otherwise, you have a whole bunch of silos with a lot of duplicated code and different practices and everything.*"

Lack of testing experience Since GUI-based testing activities are performed less frequently, it leads to a slower accumulation of knowledge and experience with testing frameworks and techniques. First, the creation and modification of GUI-based tests occur less frequently compared to other development activities, such as creating production code or low-level unit tests. This leads to fewer opportunities for discussing and sharing knowledge about testing techniques or frameworks in collaborative practices such as code reviews, and thus learning from more experienced testers. As participant P2 (a software developer) pointed out, "*We do [GUI-based testing] so rarely that we are missing a lot of knowledge within the framework itself.*"

Secondly, in cross-functional teams, the number of testers is notably smaller than the number of developers. In small teams, it is possible to have only one tester. This limits or prevents proper code reviews of GUI-based testing files.

Finally, a mismatch in development and testing experience can appear in cross-functional teams. Testers generally have less experience with software development. When both developers and testers contribute to test cases, developers should prioritize simplicity over sophisticated solutions. This approach ensures that testers can provide meaningful feedback during code reviews.

Conversely, developers often lack expertise in testing methodologies, such as determining what to test and how to test it effectively, which could lead to suboptimal test cases [205].

One of the benefits of code reviews is the educational aspect of sharing knowledge with others [190]. However, fewer GUI-based testing activities, fewer testers to review testing files, and less development experience limit the chances of receiving valuable feedback during code reviews on test-specific concerns to learn from more experienced testers.

Test robustness Another challenge is to determine the robustness of test cases. Robust test cases should not fail when the SUT undergoes minor modifications. Participants noted that robust test cases are crucial, as stated by participant P8, "*To me, the stability, reliability, and resilience of tests is, I prioritize that higher than what it actually covers. [...] Especially with GUI-level tests.*"

Participant P7 provides an example of how they avoid some challenges with less robust test cases, "*We used to have quite a lot of problems with [robustness] because certain people love to use XPaths, which is the main source of flakiness, and just having a thorough discussion with the designers. And then actually teaching each other what is the importance of having an ID or test class or whatever, like a unique identifier for each object. This took care of most of the problems that we had.*" The usage of simple XPath locators can lead to less robust test cases, as these locators often fail even on minor changes to the SUT [144].

Another example of potentially less robust test cases involves the intricacies of a testing framework's API used to retrieve specific elements within a collection of elements, such as a row in a table, in a reliable manner. As described by participant P9, "*And one thing could be that when you review if the GUI under test is a table, for instance, your tests would fetch one row. You would like to be 100% sure that the method to fetch that row is completely foolproof so that you're not fetching another row.*"

Test maintenance Over time, the codebase grows and evolves, leading to an increase in the number of test cases. Not all test cases remain relevant or are relevant in the first place. As participant P9 mentioned, "*There are many aspects to review, but one important aspect is if the test is relevant.*" Hence, changes in requirements can render a test case as not relevant, but still can cause maintenance efforts if the test case is not removed. Furthermore, in certain testing scenarios, the issue boils down to choosing the most appropriate testing method. In these cases, simpler techniques like unit testing might have been a better choice over GUI-based tests. Therefore, it is essential to assess the test's suitability for its intended purpose.

Another aspect of maintenance challenges is the lack of proper documentation during code review, which describes aspects such as the test flow and how different compo-

nents of the test fit together. Further, context information about the test's purpose can not be derived from the test cases themselves. Code reviews offer the chance to request more context information, like the purpose of a test, but reducing delays in the review process can be achieved by providing reviewers with the information they need [178].

To mitigate test maintenance challenges, reviews should consider similar practices used for maintaining production code. For instance, keeping test cases and contributions small, narrowing the scope of test cases as much as possible, separating concerns into multiple files and through separate commits, and providing documentation. As participant P8 summarized it, "*I would say one of the big challenges from my experience is Pull Request hygiene. So, a combination of many things, keeping the scope as narrow as possible, and proper documentation. If there is some clear chronological pattern to your implementation, try to segment that in separate commits so I can follow along your trail of thought.*"

Summary RQ2: Ensuring the correct validation of GUI elements is the biggest concern for reviewers. It's challenging to understand the relationship between the test code and GUI elements during code reviews, and reviewers run tests locally to ensure they validate the correct elements. Further, understanding the test flow and ensuring readable identifiers or locators are seen as challenging. The use of abstractions in test cases, along with a lack of testing experience, makes reviews even more challenging. Finally, test robustness and maintenance, e.g., through obsolete tests, is reported as a challenge.

3.5.3 RQ3. What are the information needs when reviewing GUI-based test artifacts in industry?

Table 3.6: Overview of information needs during code review (RQ3)

Information needs	Support
Context information	P1, P2, P3, P4, P5, P7, P9, P11, P12
Performance and coverage metrics	P1, P3, P4, P7, P9, P12
Test data	P7, P9, P12, P13
Logging information	P7, P9, P12

Context information Code review tools primarily display a diff of changed lines of code, often lacking the necessary context information about a changed or created test case, as shown in Figure 3.2. This limited view can impede a thorough understanding of code changes, as reviewers cannot see the broader implications or rationale behind

the changes. Context-relevant information can be provided by the contributor of a change or through tooling as part of the code review to support reviewers.

Reviewers miss context information about the **rationale of a proposed test case** to understand the "why" of a change. This includes information about the purpose of a test case, requirements of the feature that the test should cover, as well as a description of the "big picture" that the test case is contributing towards. Further, it is important to understand the expected results of test cases. Participant P3 mentioned, "*It's important to look back at why the tests were implemented, like what are the requirements that led up to this. What are we trying to verify? So we are always looking back at the feature request or ticket or backlog item or whatever you call it to see what's the feature we're covering here. And what's the related feature code as well? What's the actual production code change that we're covering also comes into play.*"

Another type of **context information is about the relationship to artifacts** of test cases or the SUT that are not present during the code review. For example, a relationship to HTML or JavaScript files that are used to develop a web GUI. Further information on how locators in a test case interact with elements of the GUI of the SUT. In contrast to code review tools, IDEs provide functionalities to navigate through related files in a vast code base.

Finally, **context information about the GUI of the SUT** is missing during the review. Apart from the relationship to the files of the SUT, there is a need for a representation of the SUT's GUI. This representation could take the form of screenshots or videos to demonstrate the exact interactions that a test case would perform, similar to what a user would do. As participant P3 mentioned, "*Yes, I suppose in terms of GUI-based testing, it could definitely be useful to have more context about the system under test. If you looking at a page object for a web app, for instance, to actually be able to see what the page looks like from an end users perspective and see what is the GUI that the test is interacting with. That could definitely be helpful.*"

Performance and coverage metrics Metrics about the test case execution provide insights into the operational aspects of the tests, helping reviewers to assess the efficiency and effectiveness of the testing process.

Test coverage of a test case is the primary metric that participants are interested in during code reviews. Simple test coverage metrics, such as lines of code, do not provide enough insights to understand if a SUT's feature is covered by a test correctly, leading reviewers to run test cases on their local machines. As participant P9 mentioned, "*Basically, or specifically when it's about these diffs, it's hard to understand the coverage.*" Thus, clear and comprehensive coverage metrics for GUI-based tests can help reviewers avoid gathering coverage data during the review and focus on the review itself.

Additionally, information about the **execution time of test cases** is of considerable interest. Understanding how long it takes to run each test case, while not as crucial as the performance of production code, is still important for reviewers. This information helps in identifying tests that may be unnecessarily time-consuming, allowing for their optimization, and aids in scheduling local test runs for the review process.

Lastly, **indicators of the robustness of test code** are vital. As previously discussed, one of the bigger challenges in reviewing GUI-based testing artifacts is assessing the robustness of the test code. Robust test code is essential for reliable testing outcomes when making minor changes to the SUT. Identifying weaknesses in the test code's robustness can lead to improvements that make the testing process more resilient and dependable.

Test data Information about the test data used for a test case is absent during the code review of GUI-based tests. As mentioned by participant P7, "*[. . .] a lot of the test data and this kind of things I need to look up somewhere else.*", reviewers need to actively fetch information about test data from other sources during reviews.

In data-driven test scenarios, considering test data during code reviews is essential to determine whether the data fits the purpose of a test case and the test environment.

Test data should fit the purpose of a test case, ensuring that it adequately covers the intended scenarios and contributes to a robust testing strategy. This may require selecting specific subsets of test data that can cover edge cases. Participant P9 mentioned, "*If you have data in a database or like in a large table or something, you would like the review to understand if that data is valid and if it fits the purpose.*"

Different test data sets are used for different versions of the test environment or SUT. Each environment may have unique requirements and constraints that should be considered when incorporating test data. As described by participant P7, "*Depending on the test environment, we have different setups for those environments because they are different versions of them. Certain [environments] are a couple of versions ahead, and there, we cannot load data in the upper test environment just because they need to be closer to how the production [environment] is looking. Every data set is unique for each test environment.*"

Logging information Log files of test code execution, including those from the SUT, are artifacts that are not typically provided during code reviews. Having access to log files allows for deeper insights into the execution of test cases, which can help identify issues with the robustness of test code. Participant P7 describes it as follows, "*We log quite a lot of the development code, but when it comes to testing that part, when it comes down to debugging, why do things fail? There's quite a poor level of logging when it comes to test code. It seems to be, that this is an area where developers, where people, just forget that it's quite important as well. Just log things when things go wrong,*

Table 3.7: Overview of ideas for future tools and practice improvements (RQ4)

Future improvements	Support
Navigation to related files	P1, P2, P9, P11
Screenshots	P1, P2, P3
Visual representation of the test flow	P4, P10
Static code analysis	P3, P6, P7
Running tests in virtual environments	P1, P5
More code review	P5, P8

especially when it comes to flakiness." Providing access to log files during code reviews necessitates the general development practice of incorporating logging in test code.

Summary RQ3: Information needed by reviewers can be categorized into: context information, performance and coverage metrics, test data, and logging information. Additional context information is the most requested information. This includes missing information about the rationale of a proposed test case, the relationship between test and production code, and information about the GUI of the SUT.

3.5.4 RQ4. What improvements do practitioners envision that can be adopted in future tools and practices for review of GUI-based test artifacts?

In this section, we present ideas for future tools and practice improvements that were suggested by the practitioners for the review of GUI-based test artifacts.

Navigation to related files The first idea for future improvement mentioned by the participants is to highlight and allow navigation to related files. This involves displaying more of the surrounding code that is not part of the diff and thus often omitted by the code review tooling. Specifically, it would highlight the relationship to the SUT's functionality, such as HTML code for web GUIs, aiding in understanding what has been tested. Additionally, integrating navigation features similar to those in IDEs, such as clicking on a variable to jump to its usage, would be beneficial. Apart from the SUT's functionality, the navigation capability should be extended to methods in testing frameworks and common libraries used in testing, allowing a more detailed analysis of the suitability of a chosen testing approach. The need for improved navigation capabilities during code reviews is also recognized in academic literature. For example, Gasparini et al. [99] proposed an extension of GitHub's user interface to improve the navigation between methods of production code in a change.

Screenshots Another suggested improvement involves capturing screenshots for each step within a test case or a video recording during its execution. These screenshots would serve as input for code reviews, reducing the cognitive load of reviewers of deciphering obscure locators and interactions with the SUT's GUI. While some testing tools currently capture screenshots only when a test fails, this functionality should be extended to include all test executions, regardless of the outcome. As noted by participant P1, this enhancement would reduce the need to run tests locally: "*That would be very cool because then you wouldn't even have to run the test locally or follow along manually.*" Participant P2 added, "*As the PR [pull request] has been running and the test has passed. It could have potentially done, for example, screenshots or a video, so we can actually see what has actually been tested. This would be an amazing enhancement because then we can see that you do not test this field at all. Or did this test, which we could see is maybe meaningless and just adding execution time.*"

Visual representation of the test flow Another type of visual aid is an abstract representation of the test flow. This visualization could be in the form of a graph summarizing test steps or actions. Various approaches have already been proposed for production code review, such as a UML-like relationship diagram [23], graph-based representation with code navigation capabilities [87], or a textual summary of code coverage impacts [175].

Static code analysis The code review process should be pre-checked by static code analysis of the test code. As with production code, test code must adhere to established coding standards and project-specific style rules, which static code analysis tools can efficiently enforce. These tools should report any deviations before the code review starts. Additionally, static code analysis could identify and highlight untested areas of the GUI. For instance, if a new button is added to the GUI but is not presented in a list of all widgets of the test suite, the analysis will highlight this gap, ensuring comprehensive test coverage. As participant P7 mentioned, "*We would see that everything that has a test ID also exists in the code. It would be a little bit like a static code review. So to say, to ensure that we haven't missed anything on a page. The second part that we wouldn't really detect if we didn't look at the page is that there might be objects there that we would also work and test with. For example, we would like to detect if there are additional new buttons that pop up.*" This statement is also supported by participant P7, "*Static kind of code review that ensures that we haven't missed things or detected things that might be considered.*"

Further, artificial intelligence (AI) based systems could be used to identify common flaws or violations against established test design patterns.

Running tests in virtual environments Another suggestion is the provisioning of a virtual environment accessible within the code review tool that allows the direct

execution of test cases. Participant P1 noted, "*If I could run the test straight from the browser where I'm doing my code review or such, that would be awesome.*" This approach differs from automated test execution in a Continuous Integration (CI) [38] pipeline, which provides test results or metrics for reviewers as inputs.

More code reviews Finally, participants support having more code reviews for GUI-based testing, along with requesting multiple approvals from different reviewers. As participant P5 mentioned, "*I would push for more peer reviews. Because I think that's a very good practice, code reviewing overall. Because you can have a discussion, that instantly means you get more information than you would get on your own.*"

Summary RQ4: Participants expressed a strong desire for enhanced navigation capabilities within code review tools. Improved navigation would allow reviewers to more easily traverse related files and understand the broader context of the code under review. Additionally, participants emphasized the need for features such as screenshots or visual representations of the test flow to better comprehend interactions with the SUT's GUI.

Other suggested improvements include the integration of more static code analysis checks and the provision of virtual environments to run tests, thereby reducing the reliance on local test execution. Finally, participants advocated for an increase in the frequency of code reviews, underscoring the value these reviews provide to testers.

Table 3.8: Summary of our findings. Type: P=Practice, C=Challenges, I=Information needs, F=Future improvements; #=Supported by number of interviews

Type	Our findings	Description	#
P	Running tests on the local machine	Run tests locally to ensure test portability, identify environment-specific issues, use IDE features, and observe interactions with the SUT directly	14
P	Side-by-side views	View the code review tool next to the GUI during test execution to track test behavior in real time	4
P	Commit-by-commit analysis	Review the commit history to gain context and understand the rationale behind the changes	2

Continued on next page

Table 3.8: Summary of our findings (continued)

Type	Our findings	Description	#
P	Reading only the diff	Some reviewers rely solely on code diffs for minor changes or when they possess strong domain knowledge, though this approach is limited	1
C	Ensuring correct validation	Ensure tests target the correct elements on the system's GUI, unclear locators, and limited context from code diffs, making runtime observation and domain knowledge essential	6
C	Understandability and readability	Understanding the test flow is challenging due to non-descriptive identifiers and large test cases, which increase cognitive load and reduce code review efficiency	4
C	Levels of abstraction	Inconsistent or too abstract test code can obscure test behavior, complicating reviews and requiring a balance suited to reviewer expertise	4
C	Lack of testing experience	Infrequent testing combined with uneven experience levels between testers and developers hinder knowledge sharing and reduce the quality of GUI-based test review	3
C	Test robustness	Concerns the resilience to SUT changes, where practices like using XPaths are a source of flakiness	4
C	Test maintenance	Ensure tests are maintainable, remove irrelevant ones, and adhere to clean code practices to minimize technical debt	5
I	Context information	Context information about the test's rationale, related artifacts, and the GUI it interacts with are missing	9

Continued on next page

Table 3.8: Summary of our findings (continued)

Type	Our findings	Description	#
I	Performance and coverage metrics	Metrics related to execution time and test robustness indicators are important	6
I	Test data	Test data used in GUI-based tests is typically missing during code reviews, though it is crucial for understanding test validity across different environments and scenarios	4
I	Logging information	Logs from test executions are rarely available during reviews, despite their importance for diagnosing issues like test flakiness and understanding test failures	3
F	Navigation to related files	Enabling navigation to related files and code elements, similar to IDE features, to better understand the relationship between test code and the SUT	4
F	Screenshots	Capturing screenshots or videos for each test step during execution would help reviewers visualize interactions with the GUI and reduce the need to run tests locally	3
F	Visual representation of the test flow	Visualizing test flows through abstract diagrams or graphs could improve the understanding of test logic and behavior	2
F	Static code analysis	Use static code analysis to enforce coding standards, detect untested GUI elements, and potentially highlight flaws in tests using AI-based tools	3
F	Running tests in virtual environments	Provisioning of a virtual environment accessible within the code review tool that allows the direct execution of test cases	2

Continued on next page

Table 3.8: Summary of our findings (continued)

Type	Our findings	Description	#
F	More code review	More frequent and collaborative code reviews of GUI-based tests to enhance knowledge sharing and review quality	2

3.6 Analysis

In this section, we present an analysis of our findings in comparison to the findings of Spadini et al. [205] and Pascarella et al. [178]. Table 3.9 summarizes the mapping of our findings to those from related work.

Table 3.9: Mapping between our findings and the findings from related work. Type: P=Practice, C=Challenges, I=Information needs, F=Future improvements

Type	Our findings	Findings from related work
P	Running tests on the local machine	Finding 5. Due to the lack of test-specific information within the code review tool, developer run code locally in their IDE [205]
P	Side-by-side views	Finding 3. Reviewing production and test code [205]
P	Commit-by-commit analysis	Reading commit message or any documentation attached to the review request [205]
P	Reading only the diff	–no mapping–
C	Ensuring correct validation	Finding 4. A main concern of reviewers is understanding whether the test covers all the paths of the production code [...] [205]
C	Understandability and readability	Finding 4. A main concern of reviewers [...] and ensure tests' maintainability and readability[205]
C	Levels of abstraction	–no mapping–
C	Lack of testing experience	Finding 8. Novice developers and managers lack testing experience [205]
C	Test robustness	–no mapping–

Continued on next page

Table 3.9: Mapping of our findings (continued)

Type	Our findings	Findings from related work
C	Test maintenance	Finding 4. A main concern of reviewers [...] and ensure tests' maintainability and readability[205]
I	Context information	N3. Rational [178]; N4 Code context [178]; Finding 6. Reviewing test files requires developers to have context about not only the test, but also the production file under test [205]
I	Performance and coverage metrics	Finding 5. Due to the lack of test-specific information within the code review tool, developer run code locally in their IDE [205]
I	Test data	Finding 5. Due to the lack of test-specific information within the code review tool, developer run code locally in their IDE [205]
I	Logging information	Finding 5. Due to the lack of test-specific information within the code review tool, developer run code locally in their IDE [205]
F	Navigation to related files	Finding 9. Review tools should provide better navigation between test and production files [205]
F	Screenshots	New methods should be devised to not only provide general information on code coverage, but also provide information that is specific to each test method. [205]
F	Visual representation of the test flow	New methods should be devised to not only provide general information on code coverage, but also provide information that is specific to each test method. [205]
F	Static code analysis	–no mapping–
F	Running tests in virtual environments	–no mapping–
F	More code review	Set aside sufficient time for reviewing test files [205]

Importance of reviewing files for GUI-based testing Spadini et al. [205] found that test files are less frequently reviewed and often regarded as less important than production code. In our study, which primarily involved testers rather than developers, we offer a different perspective, valuing the practice of code reviews for GUI-based testing. However, due to our sampling of only having testers, we are not in a position to confirm or reject previous findings or generalize our findings. Despite this limitation in generalizability, the responses still provide valuable insights, as a majority of our participants emphasized the importance of code reviews and did not suggest alternatives to this practice. Further, they expressed support for increasing the frequency of reviews. This indicates a recognition of the critical role of code reviews, as practiced in the industry, in ensuring the quality and effectiveness of GUI-based testing, as well as in facilitating the sharing and discussion of test-specific topics.

Practice or challenge In our study, practices and challenges often reflect two perspectives of the same phenomenon, with each potentially being interpretable as the other.

For instance, the use of a side-by-side view during code reviews was reported as a practice. However, it can also present a challenge of incorporating the real-time behavior of GUI-based test execution into code review tools. Faulty behavior in such tests is often best detected through direct observation, which static views may not fully capture. Similarly, the challenge of handling different levels of abstraction in test cases can also be interpreted as a practice to establish and enforce test code standards across teams or organizations. This practice can be supported through the use of static code analysis tools, such as linters, to ensure consistency, thus mitigating issues that arise from mixed levels of abstraction.

In our reporting, we classify each based on how participants explicitly referred to them, either as a practice or as a challenge.

Code review practices Of the four code review practices, we found support for three in Spadini et al. [205] study. The exception, “only reading the diff,” was mentioned only once by our participants, indicating it is not a common or recommended practice. Effective code reviews typically require a more comprehensive analysis, including examining the broader context and related code, to ensure a thorough evaluation and understanding of the changes.

All of our participants mentioned running tests on their local machines during code reviews to analyze changed test cases further. Consistent with our findings, Spadini et al. [205] argue that platforms like GitHub provide limited overview and navigation capabilities, restricting reviewers’ ability to analyze changed code effectively. Local code execution is further supported by literature on general code review practices [203].

The practice of "side-by-side views" aligns with the concept of reviewing production and test code together. Additionally, reading Git commits and understanding the commit history during reviews were practices also observed by Spadini et al. [205]. Well-crafted commit messages help developers understand the changes and the reasoning behind them [62].

Challenges Similar to the findings of Spadini et al. [205], "ensuring correct validation" and "understandability and readability" were primary concerns for our participants when reviewing artifacts for GUI-based testing.

Although not explicitly stated by our practitioners during the interviews, testers should be aware of test smells and their mitigation strategies when creating test cases. Test smells (build upon the notion of code smells [85]) refer to patterns in test code that hinder its readability, maintainability, and overall quality. These design flaws do not necessarily indicate incorrect functionality, but they often reduce the effectiveness and maintainability of test cases, making them harder to understand, modify, or extend over time [37, 64, 96] In the context of GUI-based testing, Fulcini et al. [89] presented 22 guidelines for GUI testing maintenance, which were drawn from a literature review on test smells. The authors also developed a tool that acts as a linter for Visual Studio Code to detect these test smells. These guidelines can serve as a checklist or be integrated as automated checks during review to help avoid known test smells, ultimately enhancing the maintainability of tests.

Further, testers lack development experience for tests that are represented as test scripts. This leads to situations where experienced developers need to ensure that test code does not exceed a certain level of complexity so testers can review it. Findings by Spadini et al. [205] highlighted a challenge where novice developers and managers are not aware of the impact of poor testing and reviewing on software quality. We argue that our findings can be mapped to the same challenge from this point of view. Instead of developers being more aware of testing practices, testers should be more aware of good development practices to allow a review process involving testers and developers.

In contrast, the challenges of "levels of abstraction" and "test robustness" were not identified in [205]. The issue with levels of abstraction appears to be specific to GUI-based tests, likely due to the inclusion of the entire system and its environment during GUI-based testing, as opposed to isolated functional tests. In cross-functional teams, GUI-based tests can range from sequential, step-based approaches to more programmatic approaches, such as representing pages as page objects. Relatedly, Chen and Wang [47] reported the test smell "inconsistent hierarchy", denoting a mismatch between the GUI and its modeled abstraction. Notably, this test smell also lacks an equivalent code smell in production code, further underscoring the distinct challenges of GUI-based testing.

Robustness also seems to be a bigger concern for GUI-based tests, as it was not mentioned in the study by Spadini et al. [205]. Our participants noted that interactions with GUI elements are challenging and often lead to less robust tests, a problem less prevalent in low-level unit tests. For example, issues with XPath locators have been documented [144].

Information needs Spadini et al. [205] describes a general lack of test-specific information within code review tools, while Pascarella et al. [178] summarizes general information needs during code reviews. Our findings of "performance and coverage metrics," "test data," and "logging information" can be viewed as more specialized forms of test-specific information.

No further need for information or tool support Some participants expressed satisfaction with the current capabilities of code review tools and indicated no need for additional information during code reviews. This satisfaction may not stem from the availability of GUI-based test-specific information but rather from participants' familiarity with the available tooling.

This raises the question of whether further improvements in code review tools are desired by practitioners. For instance, participant P1 mentioned, "*You just get so used to it that you don't think about it,*" while participant P6 stated, "*I'm quite satisfied with both [Bitbucket and Gerrit].*" Similarly, participant P11 noted, "*most of the time, the tooling, I find it to be good enough to understand what's going on.*"

When we asked participants why they run tests locally despite the current state of code review tools, they explained the specific information they seek, such as test runnability, coverage, and correct validation of GUI elements. The separation between code review tools and the testing environment (local machine) was not perceived as a limitation by our participants.

Future tool improvements Current code review tools lack effective navigation capabilities to related files, such as production or test code. This limitation was also identified as a primary area for improvement by Spadini et al. [205]. Enhancing navigation capabilities would not only benefit the review of test files but also improve the overall functionality and usability of code review tools.

For GUI-based testing, specific information such as screenshots and visual representations of the test flow is crucial for understanding test coverage in greater detail. Traditional line coverage metrics are insufficient for capturing the complexities of GUI-based tests. Enhanced tooling incorporating these elements would improve the effectiveness of code reviews in this context.

One example of such tooling is the Playwright Trace Viewer [161], a GUI tool that allows users to explore recorded test runs. It enables testers to navigate through

each action of the test, visually inspecting what occurred at each step. Integrating or linking such results within code review tools could effectively address the need for screenshots and visual representations of the test flow.

Emerging technologies like large language models (LLMs), such as ChatGPT, are also being explored to support the code review process. A study by Watanabe et al. [222] investigated the use of ChatGPT to generate code review feedback for proposed changes. While the tool provided useful suggestions in some cases, 30% of the feedback was met with negative reactions, as the proposed solutions did not offer any significant benefits. These findings indicate that while LLMs have potential, further refinement is needed to ensure they add value to the code review process.

Additionally, GitHub introduced Copilot code review [123] after the interviews were conducted. This AI-powered feature is specifically designed to support code reviews by providing feedback, directly integrating into GitHub. Whether this AI-driven feedback meets the needs of practitioners or proves beneficial for GUI-based testing remains a topic for future research. Since Copilot's training data comes from existing code repositories, and given that GUI-based testing is less frequently created compared to unit tests, it may result in suboptimal suggestions.

Implications for future tooling The ideas articulated by practitioners suggest not a radical departure from current review practices but an evolutionary enhancement of existing code review tooling to better accommodate the demands of GUI-based testing. We argue that future tools must be integrated into platforms like GitHub to minimize workflow fragmentation, which is a barrier to tool adoption in software engineering practice [130]. Deep integration ensures that enhancements are available across environments without requiring context-switching or external dependencies, thereby lowering the cognitive and operational overhead for testers and reviewers.

GitHub Copilot Code Review exemplifies such a tightly integrated approach, offering suggestions directly within the pull request interface.

Researchers and tool developers can leverage GitHub's extensibility through GitHub Actions and the GitHub API to provide advanced visualizations and analytical capabilities during code reviews. For example, Scheibel et al. [195] demonstrates a visual software analysis tool featuring a 2.5D interactive software map and describes its integration into GitHub. Building on this approach, test frameworks such as Playwright, which already produce rich artifacts including screenshots, logs, and execution traces, could be coupled with additional visualizations of the test flow. These visualizations could highlight key transitions, test assertions, and coverage gaps, and be enriched with hyperlinks to related source files and component definitions. Such an approach could help reviewers understand what is being tested without the need to execute the tests on their local machines.

However, determining whether such tooling enhancements would improve the efficiency or effectiveness of code reviews of GUI-based testing files requires further research.

Are the findings unique to GUI-based testing? The reported findings of this study, including practices, challenges, and information needs related to GUI-based testing artifacts, may be generalizable to other test or code artifacts. For example, the identified challenges of understandability and readability, or the need for additional context information, are not limited to GUI-based testing alone. However, although several findings are perceivably not unique to GUI-based testing, this work is the first, to our best knowledge, to address these aspects of GUI-test reviews. This statement is supported by our previous systematic literature study on code review guidelines that found no explicit guidelines for GUI-based testing [32]. This study addresses this gap by making findings related to practices, challenges, and information needs explicit, rather than assuming that insights from other types of tests are transferable.

Furthermore, although there are similarities between our findings and related works for other testing techniques, there is evidence to suggest that there are unique characteristics, or factors, to GUI-based testing that prohibit direct transference of practices. For instance, a study by Alegroth, Petersen, and Tinnerholm [14] showed that many development guidelines for source code development were not directly transferrable to GUI-based tests. Possible characteristics that set GUI-based test code apart include, but are not limited to; (1) the codes' inclusion/dependence on visual artifacts (e.g., DOM-references or images), (2) their dependency on asynchronous chronological behavior, (3) their, often, limited cyclomatic complexity, (4) high maintenance requirements due to external change factors, or (5) other, unknown, factor(s). Identifying these factors is, therefore, an interesting topic of future research.

Comparison with our previous study about guidelines In our previous study on code review guidelines for GUI-based testing [32], we proposed 33 guidelines organized into nine categories, each aimed at improving GUI-based testing artifacts: (G1) perform automated checks, (G2) use checklists, (G3) provide context information, (G4) utilize metrics, (G5) ensure readability, (G6) visualize changes, (G7) reduce complexity, (G8) check conformity with the requirements, (G9) follow design principles and patterns.

For challenges, the challenge of ensuring the correct validation, practitioners' concerns may be mitigated by guideline G6 (visualize changes), which visualizes the relationship between test cases and the SUT using screenshots or graphs representing the different states of the SUT. Understandability and readability challenges could potentially be mitigated by G1 (perform automated checks) to enforce consistent code style, G5 (ensure readability) could be mitigated by following naming conventions and proper exception handling, and G7 (reduce complexity) could help in maintaining

comprehensible test code. Test maintenance challenges may be mitigated by G3 (provide context information), which details the impact of changes on the code base and test cases, and G9 (follow design principles and patterns), which advocates design practices, patterns, and avoiding hardcoded values.

Guidelines that solely focus on artifacts under review overlook factors such as reviewer selection and, thus, are probably limited in mitigating socio-technical challenges such as a lack of testing experience. The challenge of ensuring suitable levels of abstraction in the tests can not be directly addressed by any of our guidelines. Although guideline G9 (following design principles and patterns) appears to be related, it does not specifically address the challenges highlighted by practitioners.

For information needs, the need for context information relates directly to G3 (provide context information), where the need for more information could perceivably be mitigated by providing the rationale of a change, its impact, and references to related resources. Further, G6 (visualize changes) could provide a visual representation of relevant contextual information regarding the environment and the SUT. The need for performance and coverage metrics aligns with G4 (utilize metrics), which mainly concerns providing execution times and test coverage data for changed test cases. Information needs regarding test data and logging information are not directly covered by our guidelines.

While these guidelines could potentially mitigate some of the challenges and information needs revealed in interviews, a comprehensive and detailed mapping between the literature-derived guidelines and the interview findings is beyond the scope of this study. As a potential next step for future research, an empirical evaluation of the guidelines in code reviews from both industry and open-source projects could be conducted. This evaluation would involve observing the use of the guidelines and analyzing the impact of each individual guideline in the reviews. Such research could show the extent to which the guidelines could mitigate the identified challenges and information needs.

3.7 Limitations

We utilize the trustworthiness criteria proposed by Guba [114] to discuss the limitations of this study, which arise from the qualitative nature of the empirically collected data.

3.7.1 Limits to credibility

The credibility criteria is related to the extent to which findings accurately reflect reality.

The interview guide is refined and based on interview questions from previous studies in the related area [178, 205], and thus can be seen as tested. The included questions are open-ended, allowing the participants to elaborate on answers. The researcher's interpretations were not imposed on the participants during the interviews or in the quotes used in this study. When participants' responses lacked clarity or were complex, the interviewer summarized the answers to confirm correct understanding. This approach helps to address the common challenge of language ambiguity in qualitative research [165]. The results and manuscript were sent to all interviewees for feedback, and they were asked for consent to publish the anonymized interview transcriptions. All participants responded, agreed to publish the anonymized interview transcriptions, and made no requests for corrections.

In addition, an experienced tester with 25 years of industrial experience in GUI-based testing reviewed the synthesized results to confirm the accuracy of the interpretations. This process also included providing relevant examples as part of the member checking process, thereby enhancing the validity of our interpretations [114].

Next, transparency in the data analysis process is important to ensuring the credibility of the results. We have therefore appended this manuscript with extensive supplementary materials as an online artifact package [28], which includes the codebook (all codes and their descriptions), anonymized interviews, documentation of intercoder sessions, and an export from the coding tool that can be imported to review the interviews and applied codes. By utilizing detailed descriptions and appropriate citations, we aim to provide a vivid and credible portrayal of each theme [43].

3.7.2 Limits to transferability

Transferability refers to the extent to which study findings can be generalized beyond the specific context of the study.

Recognizing that each participant may have individual preferences in reviewing and handling GUI-based tests, we mitigated this by interviewing participants from different companies. Most of the participants were also consultants with experience from various company assignments, across different domains, providing a broader perspective that goes beyond the context of a single company or domain. This enables them to provide more general insights that are perceived as applicable to multiple contexts and domains.

Our study relies on a convenience sample of professional software testers, which introduces a selection bias that could affect the transferability of our findings by potentially skewing the range of perceptions, tools, and approaches to code reviews. Participants volunteered after we reached out to professional software testers, which may have led to an overrepresentation of individuals with positive attitudes toward GUI-based testing and code reviews. At the same time, the sample includes experiences with all major GUI-based testing tools, reducing biases tied to any single tool. Moreover, our

participants had substantial industrial experience in international companies and were not only experts in GUI-based testing but also knowledgeable testing professionals more broadly. Several consultants had worked on assignments unrelated to GUI-based testing, enabling them to compare practices across domains For example, contrasting GUI-based testing reviews with source code reviews. This cross-domain expertise provides a degree of heterogeneity in testing and review practices and helps mitigate, though not fully eliminate, the contextual homogeneity of our sample.

In addition to individual factors related to the participants, social factors may also restrict the generalizability of these findings to other cultural contexts. According to Fatima, Nazir, and Chuprat [81], several social factors among participants in a code review can influence the process, including trust, the frequency and volume of interactions, relationships, and impressions, which reflect participants' judgments of one another.

While we did not systematically validate theoretical saturation for our sample size—defined as the point where additional interviews are unlikely to uncover new concepts or aspects [121]—the last few interviews did not yield any new insights. Thus, we assume that all important insights are exhausted from the interviews. Hennink and Kaiser [121] demonstrated that saturation can be achieved within a narrow range of interviews (9–17), particularly in studies with relatively homogenous study populations and narrowly defined objectives, as is the case in our study.

Furthermore, there is a tradeoff between shorter and longer interviews, which is an aspect that can impact the validity of the study. We maintained a narrow focus for our interviews and limited their duration to 30 minutes. Shorter interviews potentially increase the number of participants, enhancing the study's generalizability through sample diversity. However, longer, more in-depth interviews can improve the study's internal validity by providing richer data. We chose to keep the interviews concise to attract more participants and gather insights from diverse perspectives and contexts. This design choice is also motivated by our research goal to find general practices, which requires a diverse set of interview subjects.

The study's results are reported in the context of web applications as the system under test (SUT). Thus, the transferability of our findings to the code review of GUI-based testing artifacts on other platforms, such as mobile, desktop, or embedded systems, may be limited due to platform-specific tools and challenges.

Lastly, temporal factors may limit the relevance of the results over time. The findings represent the state of practice at the time of the interviews, which may become outdated as circumstances change. However, while the rapid pace of technological advancements can introduce variability in a tool-dependent process such as code reviews, underlying socio-technical challenges of code reviews, such as a lack of testing experience or the understandability of changes, could remain consistent despite these technological improvements.

3.7.3 Limits to dependability

The dependability criterion evaluates how understandable and reproducible the research design and execution are.

Individual researchers may have influenced the data collection for this study. To minimize researcher biases throughout the overall study, two authors collaborated in the planning and design of the study.

Another important factor is the reliance on a researcher's individual knowledge and perspective when performing thematic analysis. To address this issue, we introduced two inter-coder reliability sessions [173]. During the first two sessions, a subset of interviews was coded by three additional coders. During this session, six interviews were coded by three additional coders and compared to the first author's coding. Any deviations in the coding and interpretation of codes were discussed, leading to an updated version of the code system. Afterward, to address potential biases in the final interpretation of codes, two authors coded all interviews and reached a consensus on the application of codes to the text segments through discussions about coding differences.

The use of a tool-supported QDA approach allows traceability between the raw data (interviews) and the results, enabling other researchers to validate the findings.

3.7.4 Limits to confirmability

The confirmability criteria assess the extent to which the biases and perspectives of the researcher shaped the results.

The interview guides acted as protocols to keep the interviews focused and ensure questions were asked in a consistent manner. However, their design may have introduced a bias toward certain aspects of the phenomenon being studied. For instance, if the interviewers overlook unanticipated elements due to the strict design of the interview guide, important insights could be missed. To reduce this bias, the interview questions were open-ended, allowing participants to elaborate on their answers.

The final QDA (coding) of all interviews was performed by two researchers to reduce the potential threat of subjective judgment by a single researcher. Disagreements in the coding were resolved through discussion until a joint decision on a final code was reached. The codebook used by all coders documents the rationale behind each theme and code, along with the criteria and examples for their application [149].

3.8 Conclusion

In this study, we present our findings on the practices, challenges, and information needs associated with reviewing GUI-based test files, a topic that has been underexplored in existing academic literature. Code reviews are a well-established practice in software engineering, recognized for their ability to catch errors, improve sub-optimal solutions, and facilitate knowledge sharing and collaboration within teams [20, 53, 83]. However, the tools and methodologies currently used for code reviews were primarily designed with production code in mind, leaving a gap in understanding how these practices translate to GUI-based testing.

Our findings reveal that while participants consider code reviews to be an essential practice, they often do not adhere to any explicitly defined practice when reviewing GUI-based tests. The most common practice among participants is running test cases on their local machines to ensure they are executable and to conduct more detailed inspections. This reliance on local test execution indicates a shortfall in the functionality provided by current code review tools, which lack the necessary features to fully support effective reviews of GUI-based tests within the tool itself. Other practices include viewing the test code and the SUT's GUI side by side or analyzing changes on a Git commit basis.

The challenges identified by participants primarily relate to the unique demands of GUI-based testing. Ensuring the correct validation of the SUT is a primary concern. Additionally, "levels of abstraction" and "test robustness" emerged as challenges that were not addressed in related studies, suggesting they may be specific to GUI-based testing environments.

Participants' information needs vary widely, from general context information, such as the purpose of a test, to more specific details like test coverage and related test data. These needs highlight the importance of providing contextual and test-specific information within the code review process to ensure thorough and effective evaluations.

The findings of this study highlight the need for tailored review practices and tools specifically designed for GUI-based testing. Further research should focus on developing such tools and techniques, as well as evaluating code review guidelines, to support thorough and effective reviews of GUI-based test files. Improving the effectiveness and efficiency of code reviews in this context will enhance software quality and reliability. Additionally, an area for potential future work is to evaluate how the findings regarding the code review of GUI-based tests manifest across different testing frameworks. Such a study should investigate the prevalence and impact of our findings, as well as examine whether and how they vary across various frameworks.

Data availability

The datasets generated during and/or analysed during the current study are published on Zenodo: https://doi.org/10.5281/zenodo.17158220 or [28].

Appendix: Interview guide

Before the Interview: Informed consent

- ☐ All answers will be kept anonymous, meaning that no answer can be traced back to you.
- ☐ Do you agree to this interview being recorded? The recordings will be confidential and not shared outside the research team.
- ☐ You will be given the possibility to review all materials before they are publicly published.
- ☐ To ensure a common understanding, GUI-based testing in this interview is defined as testing the SUT's functionality, i.e., its functional conformance to its specification or user needs, through the graphical user interface.
- ☐ Purpose of study: Explore challenges and information needs during code review of files for GUI-based testing to support code reviews in the future with missing information.

General background information about interviewee

1. What is your role (developer, tester, manager, etc.) within the organization?
2. How many years of industrial experience do you have in the software development industry?
3. How many years of industrial experience do you have as a developer and/or tester?
4. How many years of industrial experience do you have with code review practices?

Code review of GUI-based tests

5. What is the importance of reviewing files for GUI-based testing?
   (a) Is the importance of reviewing files different between GUI-based tests and production code?
6. How do you conduct code reviews?
   (a) Are you following any explicit or implicit practice?
   (b) Are these practices different for reviewing files for GUI-based testing compared to production code?
7. What challenges do you face when reviewing files for GUI-based testing?
8. Do you think current code review tools support your information needs (information required to understand a proposed change) when reviewing files for GUI-based testing?
9. How would you improve the current tools or practices you use to review files for GUI-based testing?

(Backup questions, in case the interviewee is not involved in code reviews)

10. How do you ensure the quality of GUI-based tests? For example, test coverage, defect-finding ability, or the alignment with requirements.
11. Do you document and share information about created test cases?
12. Do you maintain other people's test cases?

Closing

13. Are there any important aspects of reviewing files for GUI-based testing that we did not cover?

The Prevalence of Code Review Guidelines for GUI-Based Testing in Open-Source

4

This chapter is based on Andreas Bauer, Florian Angermeir, Emil Alégroth, and Seth Anglert. "*The Prevalence of Code Review Guidelines for GUI-Based Testing in Open-Source*". Manuscript submitted to *Information and Software Technology*. 2025. DOI: 10.2139/ssrn.5547512.

Abstract

Context: Code review has become a core practice in collaborative software engineering, helping ensure code quality, detecting potential bugs, and supporting communication among developers. Prior research has shown that code review practices differ between production and test code, suggesting that established code review guidelines may fall short in the context of test and GUI-based test code. Particularly, GUI-based testing lacks adequate support during the code review process. To address this, we proposed a set of code review guidelines specifically designed for reviewing GUI-based test files, which, however, have not yet been empirically evaluated, limiting their practical relevance.

Objective: This study empirically assesses the extent to which code review comments on GUI-based tests align (explicitly or implicitly) with the concerns captured by the proposed guidelines, and uses the findings to refine the guideline set.

Method: To achieve this, we sampled code review comments discussing GUI-based test files across 100 open-source projects and manually analyzed 1000 pull requests to determine to what extent the reviewers' comments align with the proposed guidelines.

Results: Review comments aligned with the proposed guidelines in 808 of 1000 pull requests. We found empirical evidence for 25 of the 33 guidelines. The most frequently observed guideline concerns the correct use of testing techniques and exception handling, particularly regarding locators, explicit waits, and timeout behavior.

Conclusion: The observed alignment suggests that the proposed guidelines capture concerns articulated in practice, indicating practical relevance for GUI-based test reviews. This represents an initial step towards providing empirical validation of the proposed guidelines, highlighting their potential value in enhancing the quality of GUI-based test reviews.

4.1 Introduction

Modern software systems tend to be large and complex, evolving rapidly to the point where a single developer cannot oversee all aspects of the software or fully understand the implications of every change. Thus, the development of modern software systems requires extensive collaboration among professionals with diverse technical skills and domain knowledge to manage the increasing complexity of developed systems [73, 140, 166]. To facilitate collaborative efforts, many industry and open-source projects have embraced code reviews, as informal and asynchronous discussions about changes and their potential impacts before they are merged into the codebase [20, 27, 61, 109, 156]. In addition to improving the quality of artifacts under review, code reviews facilitate knowledge sharing among developers and testers.

Similarly, software testing is a collaborative effort that relies on contributions from individuals with both technical and non-technical expertise [192]. Among the various testing approaches, graphical user interface (GUI)-based testing stands out as a technique for verification and validation of a system's behavior through its visual interface, simulating real-world user interactions [8, 55].

In our earlier exploratory work, we proposed guidelines for reviewing GUI-based test files to assist contributors and reviewers during the code review process [32]. However, these proposed guidelines have not been empirically evaluated yet.

This previous exploratory research, viewed through the lens of an empirical research cycle (see Figure 4.1), is used to generate a theory that positions guidelines as the formalized version of that theory. In this theory, we propose that guidelines for code review in GUI-based test files can offer benefits similar to those of general guidelines for production code. We believe that tailored guidelines specific to the context of GUI-based testing can promote consistency, reduce ambiguity, and enhance both the effectiveness of the review process and the quality of the reviewed artifacts. For testers, they clarify expectations and explicate the information needs of reviewers. For reviewers, they serve as a checklist to systematically evaluate test artifacts and identify potential quality concerns.

Building on our prior work, we take the next step by asking whether and how the concerns captured by the proposed guidelines appear in real-world code review discussions. This study analyzes the perspectives of both reviewers and developers regarding the guidance provided by these guidelines. We seek to identify an alignment of these guidelines in code review comments on pull requests (PRs) that modify GUI-based tests in popular GitHub repositories. This alignment can be either explicit, meaning a review comment directly reflects a guideline's concern, or implicit, where a review comment could potentially be addressed by a guideline. Observing such alignment would indicate that the guidelines capture concerns that are relevant to practitioners during review and would help us refine them for clarity and coverage. This represents our initial effort to evaluate the guidelines and, consequently, our underlying theory.

We claim the following contribution of this study:

1. An empirical assessment of the alignment between proposed guidelines for reviewing GUI-based test files and code review comments from real-world projects.

2. Concrete examples that illustrate each guideline as observed in review comments, and resulting refinements to the guidelines.

3. A comparative analysis of our findings alongside those reported in other studies.

4. A replication package containing all scripts used for data collection and a set of intermediate artifacts, allowing the replication and extension of our study by other researchers.

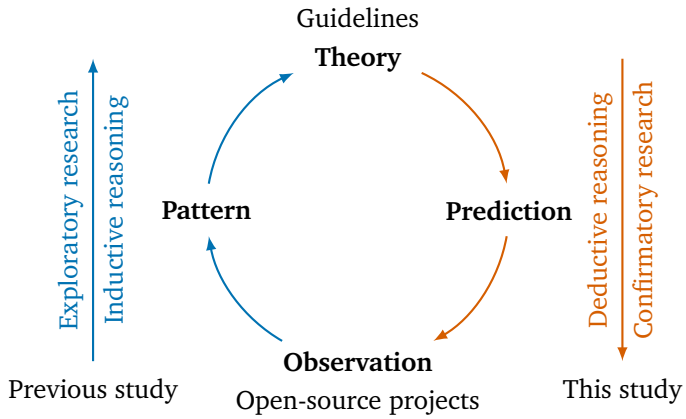

Figure 4.1: The empirical research cycle involves two main approaches: exploratory research, which generates theories using inductive reasoning (starting with observations), and confirmatory research, which tests theories through deductive reasoning (starting with a theory). Adopted from [158] and [72]

4.2 Background and Related Work

4.2.1 Code Review

Code review, as it is practiced nowadays, is characterized as informal, tool-supported, and lightweight processes [53]. In contrast, earlier methods of code review, often referred to as code inspections, followed a more formal, waterfall-like process [78]. This approach was associated with overhead costs, which hindered its adoption.

Code reviews have been shown to be effective at raising software quality and promoting knowledge sharing [22, 82, 156]. Defects found and resolved during code reviews often enhance the software's understandability and modifiability, rather than its visible functionality [150]. The practice is therefore widely adopted in both industry and open-source projects [20, 27, 61, 156].

Code reviews are not tied to any specific tool and can even be conducted manually, though they are most often facilitated by dedicated tools, such as Gerrit[1]. Modern code hosting services like GitHub[2] and GitLab[3] integrate code review directly as a core collaborative workflow, typically through a pull or merge request mechanism.

[1]https://www.gerritcodereview.com
[2]https://github.com
[3]https://about.gitlab.com

4.2.2 GUI-based testing

GUI-based testing is a form of testing, performed at a higher level of abstraction, where a system's behavior is verified through its graphical user interface (GUI), mirroring user interactions [8, 55]. By interacting with the GUI, GUI-based testing enables the assessment of both the visual appearance and functional correctness of the system under test (SUT) [8]. In industry, such testing is performed manually as well as through automated tests [76, 92, 98, 116, 127, 142].

GUI-based testing can be categorized in several ways [10]. One common approach categorizes tests based on how the test cases are defined, generally falling into three categories: scripted tests, capture and replay, and model-based tests. Scripted GUI-based testing involves the development of test scripts or code using specialized automation APIs, tools, and frameworks, such as Selenium, Playwright, and Cypress. Capture and Replay testing relies on tools that record the actions performed by a tester or user on the GUI of the system. These tools create re-executable test sequences from the recorded operations. Model-based testing involves the generation of models that represent the SUT's intended behavior from a GUI perspective. These models are used to generate test sequences that ensure coverage of various states of the GUI. The models created can be visual (defined with nodes and vertices), textual (as used in behavior-driven development), or formal (for instance, in mathematical modeling).

This study specifically focuses on automated GUI-based testing, where test cases are written as scripts along with supporting artifacts that enable automation. Other forms of test automation, such as model-based GUI testing, where tests are defined using visual or textual models, are excluded from this study. This focus is motivated by the fact that code-based test scripts closely resemble production code, which is the typical subject of code review practices.

Additionally, GUI-based testing can be applied to the GUI of various types of applications, including web applications, native desktop applications [194], and mobile applications [56, 57]. Web applications are a common focus for GUI-based tests, and there is a wide range of testing tools available for this purpose. Therefore, this study will concentrate exclusively on web applications and exclude other types. However, our findings could be relevant, to some extent, to both mobile and desktop applications that are built on top of web technologies.

A GUI test case is composed of two components: (a) the initial GUI state in which the test case is executed, and (b) a series of events that act as the test input for the GUI [157].

4.2.3 Related Work

In our previous study, we conducted a multivocal literature review of both academic and gray literature to identify code review guidelines for GUI-based testing files [32].

Software engineering guidelines outline best practices for producing high-quality software. However, most existing code review practices and guidelines aim at enhancing the effectiveness and efficiency of code reviews, and focus primarily on source code and lower-level test code [32, 66].

We discovered that there were no specific code review guidelines for GUI-based testing files, revealing a gap in the academic literature. As a result, we adapted guidelines from production and low-level test code reviews to apply them, where possible and relevant, to GUI-based testing. This process led to the creation of 33 guidelines organized into the following 9 categories: G1: Perform automated checks; G2: Use checklists; G3: Provide context information; G4: Utilize metrics; G5: Ensure readability; G6: Visualize changes; G7: Reduce complexity; G8: Check conformity with the requirements; G9: Follow design principles and patterns.

The most relevant work related to this topic is by
textcitespadini2018WhenTestingMeets. In this study, the authors investigated whether and how developers review test files during code reviews. Through interviews with 12 experienced practitioners, the study revealed that reviewing test files differs from reviewing production code. Specifically, when reviewing test files, reviewers tend to focus on test-specific concerns, such as adherence to testing best practices, coverage of expected and edge-case behaviors, identification of tested and untested paths, and the clarity and correctness of assertions. While the study highlights the difference between production and test code review, the study does not distinguish among different types of tests, i.e., unit, integration, or system tests. Therefore, it remains unclear whether review practices differ even more based on the type of tests. For example, unit tests typically emphasize correctness and isolation of individual components, while system tests focus on the end-to-end behavior of a system and integration across modules. The cognitive load required to review a system's behavior may be greater than that of isolated components, potentially requiring tailored review guidelines or tooling support. Given that cognition plays an important role in software engineering activities [79, 104, 105], research is investigating its impact on, e.g., interpreting different levels of abstraction [162].

In a study conducted by Gonçalves et al. [107] (based on the registered report [106]), the authors investigate the impact of a guidance approach via a checklist on the effectiveness and efficiency of code reviews through experimentation. The research suggests that providing a guidance approach for code reviews is expected to reduce the cognitive load associated with performing these reviews. The experimental setup included three types of conditions: ad hoc review, a checklist, and a guided checklist.

For effectiveness, the researchers measured the number of functional defects identified, while for efficiency, they assessed the number of defects found in relation to the review time taken. The checklist comprised 17 binary checks (indicating either that everything is okay or that defects have been found) and is organized into topics

such as general, classes, methods, arguments, variables, if-then statements, loops, recursion, errors, and a final check. The guided checklist presented the same items as the regular checklist, but did not show all items simultaneously. Instead, it displayed only those relevant to the selected code segment.

The study found no strong correlation between the guidance provided and code review performance. While the checklist has the potential to reduce developers' cognitive load, it was observed that a higher cognitive load might have led to better performance, possibly due to the overall low effectiveness and efficiency of the study participants.

In contrast to our guidelines, their checklist was designed for general production code reviews. The review of test code and GUI-based test code occurs less frequently [35, 205], which limits opportunities to gain expertise through reviews. Thus, we assume that the potential benefits of guidelines supporting code reviews may be greater.

4.2.4 Previous evaluation attempts

In an earlier attempt to evaluate our proposed guidelines, we conducted a pilot study in the form of a quasi-experiment to examine whether code review guidelines specific to GUI-based testing improve the quantity and quality of review comments. This approach was inspired by related research on guideline evaluation. The pilot study indicated that the guidelines derived from the literature were too general, requiring refinement, which this study aims to achieve. Refined guidelines that include concrete examples in modern testing tools establish a stronger foundation for conducting controlled experiments, making future cause-and-effect analysis potentially more effective and feasible.

A brief summary of the pilot study is as follows:

Four testing professionals from a large European consultancy company specializing in software testing and IT security participated. Each participant developed a set of eight test cases and reviewed eight test cases created by other participants. The first four test case reviews were performed using an ad hoc approach, while for the remaining four, we provided our guidelines to the participants. Cypress was the testing tool used to test a meal recipe web application, which does not require specific domain knowledge to understand.

While participants produced more comments when using the guidelines, many were superficial acknowledgments of individual guidelines, such as "the locators seem pretty robust" or "separation of concerns – OK." This experimental setup did not yield meaningful insights about the impact of guidelines on the code review process, partially due to the small sample of testing experts in the experiment.

An experimental study would have enabled us to assess the causal impact of introducing guidelines into the code review process for GUI-based test files. However, two main factors prevented us from carrying out this evaluation as intended.

First, the required effort in terms of participant time was too high. Participants had to familiarize themselves with an unfamiliar system under test, design corresponding test cases, and subsequently review them. The average time spent per participant was 7 hours. This workload made it infeasible to recruit and involve a sufficient number of participants to obtain statistically meaningful effect sizes.

Second, the absence of concrete, illustrative examples of the guidelines for a given testing tool led to ambiguity in the interpretation of certain guidelines. The elicitation of guidelines from academic literature resulted in guidelines on a more abstract level. For instance, guideline G5.6 “proper usage of testing techniques and exception handling” lacks specific instructions on what actions to take or avoid in test cases. Moreover, it was not specified which guidelines could and should be addressed through automation, and which required explicit consideration by the reviewer.

As a result, we needed to adopt a different research approach. In our case, we are taking an observational approach, analyzing a wide range of open-source projects to gain insights into real-world code reviews. While this shift does limit the strength of our conclusions, such as the inability to directly quantify the effect size of a guideline’s impact on the code review process, the observational method can still yield valuable insights.

4.3 Methodology

We conduct a theory-informed, qualitative content analysis of code review comments. Our prior exploratory work suggests the proposition that specific guidelines for reviewing GUI-based tests capture concerns that are relevant for performing code reviews. Our goal is not to reach a binary conclusion through statistical hypothesis testing on the effectiveness of these guidelines, but rather to gain insights using a confirmatory approach. This approach guides our research question:

> **RQ: To what extent, and in what ways, do code review comments on real-world projects align with the proposed guidelines for reviewing GUI-based test files (explicitly or implicitly)?**

According to the ABC framework proposed by Stol and Fitzgerald [210], this study is classified as a sample study, as we aim to draw conclusions applicable to a larger population of open-source software projects. Sample studies are commonly employed to analyze extensive collections of software development artifacts or projects, especially within the realm of open-source software.

A preliminary analysis[4] of three open-source projects revealed that a small number of pull requests ($\leq$ 1%) discuss GUI-based testing files during the code reviews. In total, 35 instances were identified where GUI-based testing files were mentioned, which can be mapped to our guidelines. Most of the code review comments we found focused on concerns related to understandability and readability. Other comments addressed the robustness and maintenance of test cases, the need for proper validation, and requests for additional contextual information.

Building on these initial insights, we expand the scope of our investigation to include multiple repositories, providing a broader and more nuanced picture of code review comments related to GUI-based testing or their absence in real-world code reviews. This analysis will also guide further qualitative exploration and potential refinement of the guidelines.

4.3.1 Mining Software Repositories (MSR)

To collect relevant evidence for the proposed GUI-based testing guidelines, we employ software repository mining as a research method [119], analyzing code review comments from open-source GitHub projects.

In this section, we outline all the steps of the data-gathering process. Figure 4.2 illustrates a visual summary of this process. For transparency and replicability, all scripts used to perform the data gathering are available in the study's replication package [30].

Inclusion of testing tools Before we begin identifying open-source repositories, we clarify our choice of Cypress and Playwright as testing tools for this search. In academic studies, Selenium [103] is frequently referenced as the reference browser automation tool for facilitating automated tests such as GUI-based testing [91, 172]. However, more recent tools have surpassed Selenium in usage and popularity, such as Cypress, Playwright, or Puppeteer [91]. According to the State of JS 2024 survey [113], which gathered responses from 14 015 participants, Cypress ranked third in usage among the JavaScript community, while Selenium has seen a continuous decline, now ranking ninth. Selenium is a general-purpose browser automation library used for testing web-based projects, rather than a pure testing framework, which results in multiple challenges when used for end-to-end testing scenarios, such as flakiness [145]. During the pilot study, our participating company reports that clients hiring test automation consulting staff are increasingly choosing Cypress to fulfill their testing needs, reflecting the growing popularity of this tool in industrial practice. As a result, we decided to focus on repositories that utilize either Cypress or Playwright as their testing tools.

[4]The analysis was conducted to determine if findings from interviews could be complemented by findings from open-source repositories in our previous study [35].

Step 1: Identify repositories We begin by searching for repositories that include files related to testing frameworks designed for GUI-based testing. In particular, our focus is on configuration files for Playwright and Cypress, as these files have a fixed name that can serve as a reliable marker for the search, unlike the test case names or associated metadata. To facilitate future studies, such as extensions or replications, the scripts can be configured to search for configuration files from other tools. To conduct our search, we utilized GitHub's REST API[5] to identify repositories that contain the desired configuration files. The specific search query we used is detailed in Table 4.1. Since the GitHub API limits each request to 1000 results (repositories), we employed a pagination strategy based on file size. By specifying file size ranges for the targeted configuration files, we partitioned the results into manageable subsets of repositories that are below the maximum search limitation. This approach enabled a sliding window mechanism, where each file size range represents a window, to systematically retrieve all matching repositories. As a result of this step, we identified a total of 59 576 repositories.

Table 4.1: Search query to find repositories that include configuration files for Playwright or Cypress. This query locates files including the specified `filename` as part of their name and with the extensions `.js` or `.ts`. Example of files that will be found: `playwright.config.ts`, `playwright.config.js`, and `example.cypress.config.ts`.

API	`GET api.github.com/search/code?q=Query`
Query	`filename:playwright.config or filename:cypress.config extension:ts OR extension:js`

Step 2: Sample top 100 repositories The resulting repository list from Step 1 includes GitHub's star counts, which act as a proxy of popularity. This star count allows us to sort and select the most popular repositories. Popular repositories tend to be widely used, and we assume, therefore, more likely to be well tested compared to those that are less popular. Since not all steps of the analysis process can be automated, and because crawling data through GitHub's API is time-intensive, we need to limit the subsequent analysis to a manageable number of repositories. Thus, we ranked repositories by star count and selected the top 100.

Step 3: Locate GUI-based test directories and pull request templates The first author manually identified directories containing GUI-based test files. These directory paths serve as filters for relevant pull requests in subsequent analysis. To make it easier to look up test files later, we also noted the test file extension (e.g., `*.test.ts`, `*.spec.ts`) for each repository to account for cases where test and production code are organized in the same directory. Repositories that use Playwright or Cypress solely

[5] https://docs.github.com/en/rest/search/search?apiVersion=2022-11-28

for non-GUI tests were excluded from further analysis. This includes scenarios where tests are executed against a web API or where the integration of single components in an isolated context is assessed. Out of the 100 repositories, 84 remain for further analysis.

We also analyzed the template files used for pull requests. When a repository includes pull request templates, contributors automatically see the template's content in the pull request body. They are expected to provide the requested information, e.g., the rationale of the change. On GitHub, pull request templates must be named `pull_request_template.md`. We collected all templates from the top 100 repositories and organized them into a single folder for easy inspection and comparison. Afterward, we manually reviewed each template and counted how often specific topics appeared in them.

Step 4: Crawl all pull requests In this step, we collected all pull request data from 84 of the top 100 repositories[6]. We utilized GitHub's GraphQL API[7], instead of using the REST API, to gather comprehensive data for each pull request, as it enabled us to customize the information included in the responses. This allowed us to extract all relevant information for the study, which included the title, description, comment thread with references to modified files, individual comments, and a list of files that have been changed. For the files modified within a pull request, we will retrieve only the first 100. We do not consider this a limitation, as these pull requests typically involve discussions about broader refactorings, such as changes in file structure, rather than specific discussions related to the modified files. This observation was made during the manual analysis of pull requests in subsequent steps.

Step 5: Filter pull requests modifying GUI-based test files For each repository, we excluded pull requests that did not create, modify, or delete GUI-based test files. This filtering was done automatically and relied on the directory paths identified in Step 3 and the list of changed files associated with each pull request.

Step 6: Filter pull requests discussing GUI-based test files Not all pull requests that modify GUI-based test files will also contain a code review of these files. Therefore, we need to filter the pull requests that actually discuss GUI-based test files via code review comments.

On GitHub, there are two ways to provide comments: threads and general comments. Thread comments consist of one comment or a series of comments related to a specific file being modified, making it easy to identify what is under discussion. In contrast,

[6]We encountered many issues collecting pull request data from the `elastic/kibana` repository as the API responded with an error when resolving the changed files of a pull request. Thus, we substituted it with the repository ranked 101.

[7]https://docs.github.com/en/graphql

general comments do not reference any specific file and typically address the entire pull request.

To inform our filtering criteria, the first author manually reviewed over 500 PRs. From this preliminary analysis, we found that general comments rarely contained substantive discussion of GUI-based test files, instead focusing on merge conflicts or vague mentions of test flakiness without technical justification that could be mapped to one of the proposed guidelines. As a result, we limited inclusion to PRs containing thread comments on GUI-based test files. While this excludes potentially relevant general comments, it allows for a manual, high-fidelity analysis without introducing the complexities and potential inaccuracies of (semi-)automated comment classification. Additionally, we excluded automated bot comments, such as those reporting test metrics or flagging contributor license agreement (CLA) requirements, as well as comments that consist solely of emojis, as they do not represent human review.

The resulting set of 2084 relevant PRs was compiled into a spreadsheet and shared among the authors for further analysis (Step 7).

Step 7: Aligning code review comments with guidelines We mapped comments on GUI-based testing files to the 33 guidelines that were organized in nine categories from our prior study [32]. For the mapping, we use the existing explanation for each guideline about its purpose, a description, and examples of its application to assess whether a comment aligns with a specific guideline.

A comment was considered aligned if it directly or indirectly reflected a guideline. For example, if a comment includes a link to the Playwright documentation, it aligns with guideline G3.5, which states to provide links to relevant resources and documentation. Additionally, a comment may also be aligned if the issue raised can be addressed by a guideline. For example, if a comment requests a better description of a test case to better convey its intent, it could be improved by following guideline G5.4, which emphasizes adhering to coding style and naming practices. We mapped comments to specific guidelines (e.g., G3.5) rather than to broader categories (e.g., G3).

The first two authors performed the mapping collaboratively over multiple sessions, discussing each case to minimize overlooked content and resolve ambiguities. When a comment was unclear, we examined the associated pull request in GitHub to get the whole context and overview of the changed tests. Additionally, decisions regarding the mapping of specific comments were documented to ensure consistent mapping throughout all sessions. Due to the choice of the collaborative process, inconsistencies in mapping between authors did not arise, and thus, we do not have any inter-coder reliability metric.

We limited the analysis to 1000 pull requests due to the substantial manual effort required. In this final sample, we ensured that all repositories containing comments

about GUI-based test files were represented. The mapping of each pull request to the guidelines is made transparent in the replication package. It includes a spreadsheet that lists all pull requests related to GUI-based test files, along with the guideline mappings, additional notes, and URLs to access the pull requests on GitHub.

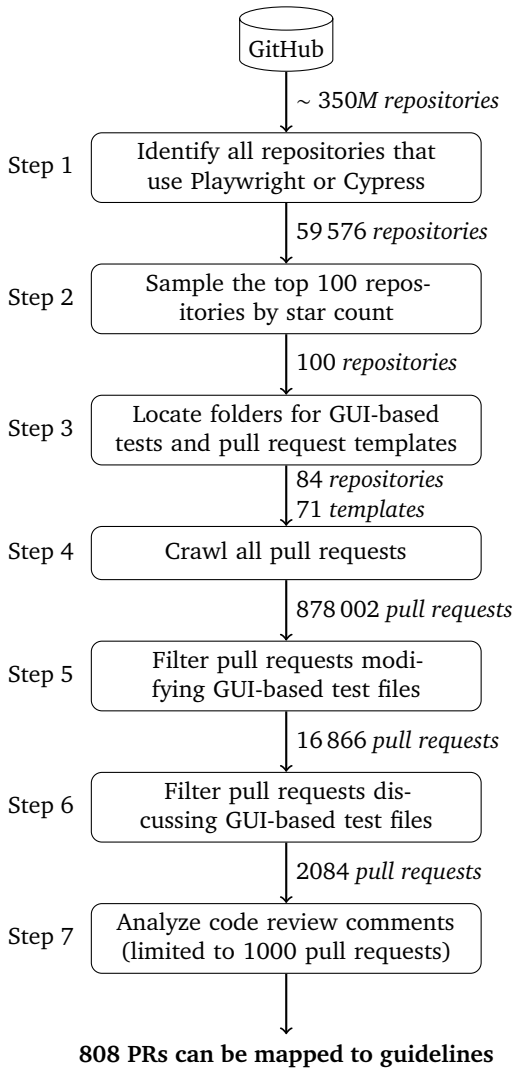

Figure 4.2: Summary GitHub review analysis

4.4 Results

Description of Collected Data

Identified repositories We identified a total of 59 576 repositories that contain configuration files for either Playwright or Cypress and therefore potentially include GUI-based tests. Of these repositories, 67% have no stars, suggesting they are primarily hobby projects, experimental demonstrations, coursework, or early-stage developments.

The most starred repository is `freeCodeCamp`[8], an open-source curriculum for learning programming, with 417k stars. Repositories in the top 100 each have at least 19k stars. Among these repositories, 84 had pull requests (PRs) modifying GUI-based test files, and 56 explicitly discussed GUI-based tests.

Pull requests related to GUI-based test files In total, we gathered data from 878 002 PRs. As shown in Table 4.2, modifications to GUI-based test files are rare, and code review discussions of these files are rarer still. Only 1.9% (16 866 PRs) of all PRs modify GUI-based test files, and just 0.24% (2084 PRs) contained comments or discussions.

[8] https://github.com/freeCodeCamp/freeCodeCamp

Table 4.2: Overview of top 100 repositories and the number of total pull requests, pull requests changing GUI-based test files (w/ tests), and pull requests containing code review comments addressing GUI-based test files (w/ review)

#	Repository	Total	w/ tests	w/ review	#	Repository	Total	w/ tests	w/ review
1	freeCodeCamp/freeCodeCamp	39974	432	154	51	hasura/graphql-engine	3397	0	0
2	facebook/react	17280	13	3	52	vercel/swr	964	10	0
3	tailwindlabs/tailwindcss	3638	0	0	53	remix-run/remix	4696	0	0
4	storybookjs/storybook	13837	254	8	54	floating-ui/floating-ui	1379	68	0
5	n8n-io/n8n	11666	651	117	55	alan2207/bulletproof-react	115	4	0
6	supabase/supabase	14221	8	0	56	cockroachdb/cockroach	73308	11	0
7	mermaid-js/mermaid	2911	0	0	57	refinedev/refine	4838	43	0
8	microsoft/playwright	16537	57	9	58	ianstormtaylor/slate	2488	12	2
9	coder/code-server	1980	66	22	59	backstage/backstage	21820	2	1
10	louislam/uptime-kuma	1741	10	3	60	codex-team/editor.js	1154	124	30
11	grafana/grafana	62133	752	113	61	fabricjs/fabric.js	3195	54	18
12	apache/superset	18373	145	23	62	ueberdosis/tiptap	1492	0	0
13	immich-app/immich	7122	41	5	63	nextcloud/server	31217	366	25
14	facebook/docusaurus	6433	15	0	64	docsifyjs/docsify	1146	0	0
15	nuxt/nuxt	7690	66	11	65	xyflow/xyflow	1632	9	0
16	gatsbyjs/gatsby	22330	0	0	66	menloresearch/jan	2265	18	0
17	remix-run/react-router	4170	0	0	67	pmndrs/react-spring	603	0	0
18	nocodb/nocodb	5464	441	16	68	motiondivision/motion	1160	0	0
19	ionic-team/ionic-framework	8101	0	0	69	AdguardTeam/AdGuardHome	494	0	0
20	withastro/astro	8023	386	34	70	GitbookIO/gitbook	1292	75	5
21	toeverything/AFFiNE	8686	1099	28	71	twentyhq/twenty	6714	56	14
22	marktext/marktext	1075	11	0	72	DioxusLabs/dioxus	1856	46	0
23	TryGhost/Ghost	15999	118	4	73	paperless-ngx/paperless-ngx	2975	32	0
24	laurent22/joplin	5181	72	10	74	keycloak/keycloak	19737	81	11
25	cypress-io/cypress	8518	0	0	75	nextauthjs/next-auth	3199	4	1
26	slab/quill	784	8	0	76	ZuodaoTech/everyone-can-use...	694	16	0
27	FuelLabs/fuels-ts	2383	8	3	77	PostHog/posthog	24958	937	85
28	RocketChat/Rocket.Chat	18199	728	130	78	eslint/eslint	7890	0	0
29	dcloudio/uni-app	522	0	0	79	marmelab/react-admin	6054	204	27
30	tldraw/tldraw	3660	36	10	80	nrwl/nx	15453	111	2
31	expo/expo	19097	46	0	81	goharbor/harbor	9626	1	0
32	tabler/tabler	1041	0	0	82	badges/shields	8145	7	1
33	penpot/penpot	4312	114	11	83	openai-translator/openai-translator	683	8	1
34	gradio-app/gradio	5093	74	10	84	Redocly/redoc	764	31	1
35	eugenp/tutorials	17732	0	0	85	BabylonJS/Babylon.js	13337	19	3
36	trpc/trpc	4572	0	0	86	react-navigation/react-navigation	1950	12	0
37	novuhq/novu	6478	78	5	87	usablica/intro.js	1236	7	3
38	appsmithorg/appsmith	17130	1623	309	88	forem/forem	14840	453	149
39	ray-project/ray	32344	60	6	89	plausible/analytics	2748	37	4
40	slidevjs/slidev	760	12	0	90	remotion-dev/remotion	3355	0	0
41	Kong/insomnia	3972	819	66	91	BerriAI/litellm	5051	18	0
42	logseq/logseq	3410	183	28	92	QwikDev/qwik	4962	1	0
43	calcom/cal.com	12910	775	209	93	facebook/lexical	4445	616	52
44	ToolJet/ToolJet	7699	535	19	94	wandb/openui	55	2	0
45	payloadcms/payload	6394	1917	80	95	handsontable/handsontable	3010	39	2
46	spacedriveapp/spacedrive	2073	10	1	96	eclipse-theia/theia	6595	41	8
47	jaredpalmer/formik	1348	3	0	97	hcengineering/platform	6249	489	19
48	usebruno/bruno	1622	0	0	98	apollographql/apollo-client	6815	3	1
49	harness/harness	1527	0	0	99	sveltejs/kit	6242	573	72
50	mattermost/mattermost	21665	560	108	100	microsoft/fluentui	19899	0	0

Pull request templates Pull request templates help standardize contributions by prompting authors to complete specific steps or include required information, often through checklists. Among our top 100 sampled repositories, 48 included such templates.

4.4.1 Findings from pull request templates

We analyzed 71 PR templates from the 48 repositories and identified recurring elements. The templates generally requested either a text description (e.g., explaining why the change is necessary) or items to be checked from a checklist (e.g., does it contain breaking changes: yes/no). In our report, we only include the elements that appear in templates from at least two repositories to exclude solely project-specific elements.

Links to related resources (53 templates) Contributors are most often asked to link to the relevant GitHub issue, since feature proposals and bug discussions occur there. Some repositories require opening an issue before submitting a pull request to discuss the envisioned changes first, which shifts part of the discussion away from the code review. Templates also frequently request links to specifications, design documents, or other contextual resources, as well as to any dependent PR or issues that must be resolved first.

Summary of changes (45 templates) Many templates require a concise summary of the modifications introduced. Some also mandate a changeset file for version tracking in the release process.

Motivation for the change (27 templates) In addition to the summary of changes, contributors should articulate the rationale: why the change is necessary, what it affects, and which design decisions or trade-offs were taken during its implementation.

Test plan description (24 templates) In one-third of the identified pull request templates, contributors must describe how the changes can be tested. This includes information about the environment configuration, minimal test data, the expected behavior for key scenarios, and any other information that helps to test the pull request.

Visual demonstration (20 templates) When applicable, contributors should include a video, animated GIF, or screenshots that illustrate feature usage or user-interface changes.

Before/after screenshots (10 templates) Some templates specifically request paired screenshots to highlight visual differences in the GUI.

Highlighted changes for reviewers (2 templates) A small number of templates ask contributors to highlight particular code sections or files for reviewers to prioritize their review.

Checklist items Contributors shall check or uncheck items that are presented in Table 4.3 if applicable.

Table 4.3: Checklist items in pull request templates and the number of templates (#) they appear in

Aspect	#
Documentation was added, updated, or checked if it is still up-to-date	29
Tests were added or updated and cover new code	28
Change type specified, such as bug fix, feature, documentation, refactoring, etc.	14
Project-specific contribution guidelines were read and followed	17
Style guidelines were followed	12
Breaking changes were introduced	8
Tests were executed locally	6
Self-review by the author was performed	6
User-facing (GUI) changes were introduced	4
PR scope is as isolated as possible (only one feature) without unrelated changes	2

4.4.2 Mapping Code Review Comments to GUI-based Testing Guidelines

Out of the 1000 analyzed PRs that included discussions related to GUI-based testing, 808 instances (80.8%) were aligned with our proposed guidelines.

In the following analysis, we detail the level of support observed for each guideline, organized by category. Table 4.4 summarizes the found evidence to support our guidelines based on the gathered data. As evidence, we consider the analyzed code review comments at the PR granularity level, as well as insights from the PR templates.

Table 4.4: Overview of guidelines and the supporting evidence found in open-source repositories

ID	Description	PRs	Other evidence
G1	**Perform automated checks**		
G1.1	Automated checks	33	GitHub Actions
G1.2	Automated code style checks	25	GitHub Actions
G2	**Use checklists**		
G2.1	Use a checklist	0	43 PR templates
G3	**Provide context information**		
G3.1	Additional context information	63	GitHub Actions
G3.2	Rationale of change	55	27 PR templates
G3.3	Impact of change	14	–
G3.4	Design & architecture	5	–
G3.5	Links to related resources	72	53 PR templates
G3.6	Dependencies between test and production code	1	–
G3.7	History of changes	2	–
G3.8	Edge cases	9	–
G3.9	Prioritization of files	203	2 PR templates
G4	**Utilize metrics**		
G4.1	Measure code metrics	0	GitHub Actions
G4.2	Measure execution time	0	GitHub Actions
G4.3	Measure test coverage	0	GitHub Actions
G5	**Ensure readability**		
G5.1	Ensure readability	13	–
G5.2	Provide code comments	30	–
G5.3	Follow code/naming style (same as G5.4)		see G5.4
G5.4	Follow code/naming style in test files	80	–
G5.5	Avoid code comments	28	–
G5.6	Proper usage of testing techniques and exception handling	277	–
G5.7	Correct assertions	49	–
G6	**Visualize changes**		
G6.1	Visualization of changes	0	–
G6.2	Visualization of the impact of changes	2	–
G6.3	Traceability and easy navigation	1	–
G7	**Reduce complexity**		
G7.1	Keep size of changes low	13	–
G7.2	Keep complexity low	13	–
G7.3	Avoid unrelated changes	13	2 PR templates
G8	**Check conformity with the requirements**		
G8.1	Adherence to requirements	0	–
G9	**Follow design principles and patterns**		
G9.1	Use of design patterns	36	–
G9.2	Don't repeat yourself	27	–
G9.3	Avoid hardcoded values	10	–
G9.4	SOLID principles	0	–

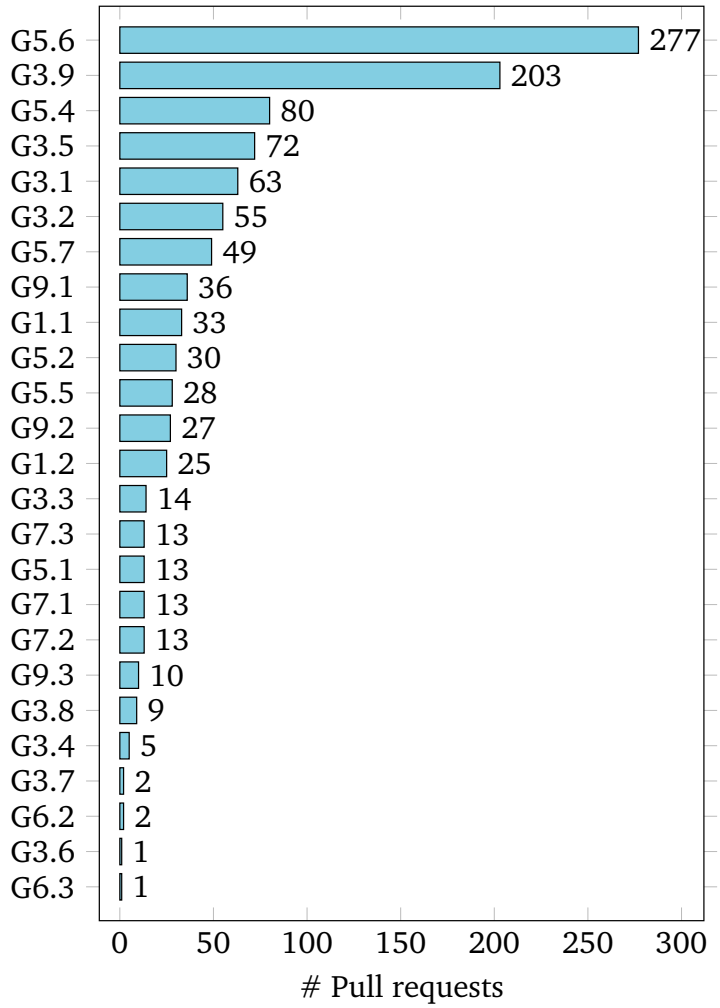

Figure 4.3: Guidelines observed in code review comments found in open-source repositories

G1: Perform automated checks Our guidelines for automated checks focus on two main areas: general checks to identify incorrect code patterns and ensure code style conformity. All 84 analyzed repositories employ automated checks via continuous integration (CI) pipelines (e.g., GitHub Actions) or third-party bots that integrate into GitHub.

Examples of automated checks relevant to GUI-based tests include providing test execution results (such as test coverage and execution time) and checking adherence to code style rules. To report GUI coverage, Cypress provides a report that includes a coverage score based on the ratio of interacted elements to all interactable elements available. Playwright allows users to record complete traces, highlighting interactions with GUI elements, for reporting. AI/LLM-based code review bots are also utilized. Finally, the licensing of contributions through a contributor license agreement (CLA) is requested automatically.

From the code review discussion, we noticed that 33 PRs include comments that were related to automated checks (G1.1), pointing out typos that can be easily fixed with tools. Additionally, automated code style checks (G1.2) were mentioned 25 times. These comments included simple adjustments, such as correcting spacing or adding empty lines between functions, which are typically enforced by linters. In some cases, style issues were introduced after merging other branches into one of the PR, despite a linter being available.

G2: Use checklists Our use checklist guideline refers to aspects like GUI element localization, test oracles, or test synchronization. While there were no code review comments related to the checklist, 43 of the 71 pull request templates included predefined checks, as detailed in Section 4.4.1. The items covered by these checklists were generally applicable, but only one was specifically related to testing: it involved verifying whether a new or modified test was successfully executed in a local environment.

G3: Provide context information Providing context information encompasses a broad category of guidelines aimed at equipping reviewers with additional information to better understand the changes, thus avoiding the need for reviewers to request this information during reviews.

The general guideline of providing context information (G3.1) helps to better understand the change under review. Since all sampled GitHub repositories utilize automated bots, they already provide additional information without specific requests from reviewers. For instance, bots offer links to a deployment or instance of a preview environment for the system under review, provide summaries of changed file sizes, and report the sizes of bundled packages. In 63 PRs reviewer requested more context information to understand a particular aspect of the tests.

Providing a rationale (G3.2) is a common requirement in PR templates. We identified it in 27 templates. In 55 PRs, reviewers requested a rationale for specific tests to clarify changes. Contributors should explicitly state the purpose and motivation, such as why a test case was removed, since these cannot be deduced from the code alone.

Providing information about the impact of a change (G3.3) was observed in 14 PRs. Related discussions addressed the execution time of tests, the effects of parallel execution, and changes needed to prevent test failures or flakiness.

Providing design and architecture details (G3.4) appeared in 14 PRs, where discussions concerned the overall structure of tests and related aspects, such as credential handling.

Linking to related resources (G3.5) can be observed often (72 PRs) as providing links to other test files that serve as references for implementing, documentation of testing tools, and related GitHub issues/PRs.

Providing dependency information between test and production code (G3.6) was not observed in the analyzed PRs.

Providing change history (G3.7) occurred in only 2 PR.

Discussing edge cases (G3.8) appeared in 9 PRs, covering scenarios such as boundary values and inconsistent behavior across browsers.

Prioritizing files under review (G3.9) was present in only two PR templates, but could be observed often in review comments. It was the second most observed guideline, with 203 PRs containing comments by the contributor of a change to draw reviewers' attention to specific files and sections. In these comments, contributors provide context or ask questions to reviewers.

G4: Utilize metrics For the guidelines of utilizing metrics, we found support only in automated reports. Test execution and coverage are provided via GitHub Actions or bot comments. There were no discussions on code metrics between reviewers and contributors.

G5: Ensure readability The guidelines in this category focus on the readability and clarity of test code, addressing reviewers' concerns about code that is confusing or unclear.

Ensure the readability of tests (G5.1) is the general guideline that captures comments that emphasize overall test readability but do not fall under other specific subcategories. In 13 PRs, we could observe the need to improve the readability, where certain test sections were difficult to read and should be refactored.

Adding code comments (G5.2) was requested in 30 PRs to clarify ambiguous code. Examples of the requested comments include explanations for why a variable is set in

the test, descriptions of complex locators, reasons for potential errors and how they are handled, as well as deviations from project-specific practices.

Following a coding style in test writing (G5.3 and G5.4[9]), could also be observed in 28 PRs. Adherence to a specific code style and naming convention is generally enforced through automated checks using a linter. However, there were instances where the reviewer requested that the test code be formatted according to the style guide or that variable names align with the project's naming conventions. In many cases, reviewers expressed concerns about the naming of test case descriptions, noting that they often contained typos, were inconsistent with other test case descriptions, or did not clearly convey the intent of the test case. These issues cannot be enforced through automated checks.

Remove code comments (G5.5) is the inverse of G5.2 and could be observed in 28 PRs. Reviewers request to remove outdated or uninformative comments that no longer aid comprehension due to code changes. A significant concern is the presence of commented-out code that is no longer in use. This not only reduces the readability of the test code but also creates confusion for testers and reviewers about the potential relevance of the test code in the future. Reviewers have requested the removal of this commented-out code or provided an explanation why it should not be deleted. In cases where the code is infrequently used, reviewers recommend employing techniques such as feature flags to enable tests only in specific situations.

The proper usage of testing techniques and exception handling (G5.6) was observed most frequently with its appearance in 277 PRs. Beyond readability, G5.6 also relates to maintainability. The use of familiar testing patterns improves reviewer comprehension and supports long-term code quality. Reviewers frequently highlighted problems such as incorrect use of mocks, testing at inappropriate levels of abstraction, or improper variable initialization, factors known to contribute to test unreliability [205].

We observed that reviewer feedback focused on the correct use of locators and timeout/wait behavior, concerned that an improper use of these techniques would introduce flakiness of tests.

Modern testing frameworks provide various locator methods to identify GUI elements based on their properties. Reviewers often recommended replacing less precise locators with more robust alternatives. For example, in one repository, there was a strong preference for using the `findByRole` locator, which targets elements by semantic role (e.g., buttons) over the more generic `findByText` locators. An example of locator methods is shown in Listing 4.1.

```
1 cy.findByText("confirm").click() // avoid
```

[9]G5.3 and G5.4 are considered the same in this study. In our previous research, we listed G5.4 as a separate guideline because it was mentioned in the source material specific for testing.

```
2 cy.findByRole('button', { name: /confirm/i }).click()
```

Listing 4.1: Example two locators in Cypress targeting the same element on the GUI

Explicit wait statements (e.g., `cy.wait(2000)`), to wait for results/responses of the system under test (SUT), are often not necessary and should be avoided. Reviewers have requested the removal of these explicit wait statements. When timeouts need to be defined, reviewers recommend consolidating the timeout settings either globally or for specific test cases, rather than setting them for individual test steps (see Listing 4.2).

To ensure the correctness of assertions (G5.7), reviewers flagged assertions that were incorrect, unnecessary, or suboptimal. This guideline could be observed in 49 PRs. Unnecessary assertions included those on intermediate steps not relevant to the test objective. Suboptimal assertions included vague checks, for example, verifying a button by label instead of confirming the expected system behavior. In Playwright, the `click()` method automatically waits for and checks the visibility of the targeted element by default. Therefore, additional visibility checks are not needed. In one instance, a reviewer discouraged the use of regular expressions for assertions, stating: "Assertions in our tests should be strict and concise. Using regexes means that we're losing strictness unless your regex is watertight."[10]


```
1 await page.goto('/home')
2 await page.waitForTimeout(5000) //avoid
```

Listing 4.2: Example of an explicit timeout to wait for a transition to another page.

G6: Visualize changes This set of guidelines addresses additional visualizations of test case changes that aid their comprehension.

Visual graph representations (G6.1) were not observable from either code review comments or PR templates.

Screenshots or recordings (G6.2) appeared in PR templates as before-and-after screenshots and in two PR comments highlighting targeted GUI elements. In both instances, the screenshot showed the rendered GUI and the browser's debug view of the DOM, to highlight the nesting of an element within another one, and to propose an element's attribute to locate it.

Traceability and navigation between files (G6.3) were observed in only one PR. The relevant PR template referred to visual changes in the system under test, not to visual representations of covered GUI states by tests or their interactions. Combined with the low number of related comments, support for this guideline category appears limited.

[10] https://github.com/facebook/lexical/pull/1189

G7: Reduce complexity This category of guidelines concerns reducing the complexity of test file changes.

Keep changes small (G7.1) was observed in 13 PR comments, often by avoiding unnecessary tests or checks.

Keep the complexity low (G7.2) appeared in 13 PR comments. This guideline favors simple solutions, which may conflict with design principles such as Don't Repeat Yourself (G9.2) that would introduce abstractions for better reusability.

Avoiding Unrelated Changes (G7.3) was observed in two PR templates and 13 PR comments.

When analyzing the number of files changed per pull request, particularly the distribution between production code and test code files, it becomes evident that test case files are often included alongside production code files rather than as separate modifications. For pull requests containing GUI-based test files, the median number of files modified is 8, while the mean is substantially higher at 41. This suggests that updates to GUI-based test cases are typically made in conjunction with feature changes instead of being addressed independently.

G8: Check conformity with the requirements We could not find any support in the code review comments that discuss the adherence of the test code to its requirements.

G9: Follow design principles and patterns This guideline category promotes the following of design principles and patterns. Specific to GUI-based testing is the page-object model [143], a common abstraction in GUI testing.

Applying design principles and patterns (G9.1) was observed in 36 PRs. Beyond recommending the page-object model, reviewers requested adherence to project-specific design conventions. For instance, as one reviewer commented, "*I'd prefer we kept this code portable by using the JS primitive Object.keys and making decisions about unreachable cases right now*"[11].

Don't Repeat Yourself (G9.2) appeared in 27 PRs, where reviewers encouraged reuse of recurring test steps, complex locators, and assertions.

Avoiding hardcoded values (G9.3) was noted in 10 PRs. Reviewers advocated configuring tests via variables or environment settings, favoring parameterization over hardcoded values. As one reviewer commented, "*Please replace the hard-coded values with fixtures or variables for better maintainability.*"[12]

[11] https://github.com/Kong/insomnia/pull/5076
[12] https://github.com/appsmithorg/appsmith/pull/34925

We did not find any references to the SOLID design (G9.4) principles—Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—in the code review comments.

4.5 Discussion

To answer our research question "*To what extent, and in what ways, do code review comments on real-world projects align with the proposed guidelines for reviewing GUI-based test files (explicitly or implicitly)?*", we discuss our findings from different perspectives.

How well do the proposed guidelines correspond to current practices observed in open-source projects? Current practices observed in open-source projects reflect the proposed guidelines in three ways: (1) implemented via automation, (2) enforced through pull request templates, and (3) discussed during code review.

First, in the repositories we analyzed, several guidelines are already realized through GitHub Actions and bots. Automated checks (G1), the use of code quality metrics (G4), and aspects of providing contextual information (G3) are already incorporated. This suggests that the principles underlying those guidelines not only could provide benefits, but have already been institutionalized. As a result, incorporating them into new projects is recommended, as automation ensures the application of guidelines, and the efforts required to set up automation are low. All analyzed repositories employed GitHub Actions, bots, or both.

Second, some guidelines are reflected in pull request templates. For example, having a checklist for contributions (G2), providing a rationale and a prioritization of files (G3), or avoiding unrelated changes (G7). Although templates do not enforce strict rules, they serve as soft constraints that guide contributors towards consistent submissions that are ready for review and assist reviewers by providing a structure for the review.

Third, our analysis of review comments shows frequent attention to test readability (G5), particularly in relation to the appropriate use of testing techniques (G5.6) and correct assertions (G5.7). These aspects are often difficult to automate and remain dependent on reviewer expertise. For example, choosing a meaningful GUI locator strategy, avoiding explicit wait/timeout statements [89], or asserting behavior based on GUI state rather than raw HTML can enhance maintainability and clarity [63]. Additionally, the most suitable testing technique will vary depending on the technology stack and the aim of the testing efforts for a specific project.

Visualizing changes (G6) only appeared in two code review comments. Our previous study found that reviewers often run test cases locally to verify correct interactions with the SUT [35]. Providing visualizations of interactions with the SUT could reduce this need by making execution behavior more transparent. However, support for such

features depends heavily on the chosen testing framework. For example, Playwright offers the Trace Viewer[13], a GUI tool that enables detailed inspection of recorded traces after test execution, facilitating debugging of failures in CI. Playwright thus illustrates how visualization can address the need to provide visualizations manually, and we assume similar capabilities will be adopted by other tools.

Do the findings indicate an implicit adoption of the guidelines? Our investigation demonstrates extensive coverage of the underlying principles behind the guidelines in nearly all observed repositories. However, some guidelines (G3.6, G3.7, G4, G6, G8, and G9.4) demonstrate comparatively low coverage in code review comments. The overall coverage of many guidelines indicates that the proposed guidelines represent relevant principles in practice. Thus, we argue that an implicit adoption of these guidelines in practice can be inferred.

Can we explain the absence of observations for certain guidelines? Dependencies between test code and the production code it tests (G3.6) did not appear in review comments, and no clear reason could be identified for this absence. One possible explanation is that when production code and the corresponding test code are part of the same change, the overall summary of changes may be sufficient for reviewers. Further, a reviewer's familiarity with the codebase can have an impact on this guideline, but we cannot assess this based on the data we have gathered. This omission is notable, as reviewers frequently reference related tests or documentation (G3.5). Prior work, including our study [35] and that of Spadini et al. [205], identifies such dependencies as a recurring information need for reviewers.

For the supplying of a history of changes (G3.7) guideline, we can assume that the availability of a full commit history and references to prior PRs by GitHub and Git already covers the need described by the guideline. Therefore, this information does not need to be provided or requested, as it is always available for review.

One possible explanation for the lack of comments regarding adherence to requirements (G8.1) is that new contributions to the repository, such as new features and their corresponding tests, should first be discussed in GitHub issues or other platforms before being submitted as pull requests (PRs), as outlined in Section 4.4.1. Thus, potential discussions around requirements are resolved in other ways.

The SOLID principles (G9.4) appear less relevant for GUI-based tests, such as those implemented with Playwright or Cypress. These tests primarily consist of linear sequences of user interactions (e.g., clicking, typing, navigating), which emphasize procedural flows rather than modular design. While additional helper functions used in GUI-based tests could incorporate concepts like inheritance hierarchies or dependency inversion, we did not observe this in the analyzed repositories. As a

[13]https://playwright.dev/docs/trace-viewer

result, we did not find evidence of SOLID principles being applied in this context, and thus, their practical value for GUI-based tests remains uncertain.

Should the proposed guidelines be refined in light of our empirical observations? Although observed in 203 PRs, comments on file prioritization (G3.9) exhibit a cohesive scope, suggesting no refinement is needed. In contrast, the guideline on testing techniques and exception handling (G5.6) appeared most frequently, with 277 PRs. Given that this guideline is specific to GUI-based testing without a counterpart in production code review guidelines and the large number of diverse examples support refining G5.6, we propose the following fine-grained sub-guidelines, as shown in Table 4.5.

Table 4.5: Sub-guidelines for G5.6 and the number of pull requests (PRs) where they were observed

ID	Guideline	PRs
G5.6.1	Select robust locators, like role-based or test-specific locators	100
G5.6.2	Avoid explicit waits and define timeouts on a test-case level	63
G5.6.3	Avoid test annotations `skip` and `only`	37

Select robust locators (G5.6.1): According to best practices recommended by Playwright[14], Cypress[15], and the Testing Library[16], GUI elements should preferably be located either by their role or by a dedicated test identifier. Role-based locators (e.g., `page.getByRole()` in Playwright) align with how users and assistive technologies perceive the GUI under test, for instance, as a button, checkbox, or dropdown. In combination with a text attribute, role-based locators become more stable than pure text-based locators. Alternatively, elements can be identified using test-specific attributes (e.g., `page.getByTestId()` in Playwright). According to the tools' best practices, locating elements by generic tags, classes, or IDs is fragile, as these attributes often change. Test-specific identifiers mitigate this volatility and provide the most resilient way of locating elements. However, they require introducing these test-specific attributes first, and they do not reflect a user's perception of the GUI anymore.

Avoid explicit waits (G5.6.2): Unnecessary explicit wait statements with arbitrary durations constitute an anti-pattern or test smell [89]. Modern frameworks such as Playwright and Cypress automatically handle waiting for events and interactions, eliminating the need for manual waits, which slow down the execution time. For instance, they resolve pages or locate dynamically loaded elements without additional delay instructions. When specific timeout behavior is required, it should be defined at

[14]https://playwright.dev/docs/locators
[15]https://docs.cypress.io/app/core-concepts/best-practices
[16]https://testing-library.com/docs/queries/about

the test-case level rather than for individual statements. However, synchronization of tests and the SUT remains a significant challenge in GUI-based testing [168].

Avoid test annotations `.skip` and `.only` (G5.6.2): Annotations such as `.skip` and `.only` allow testers to focus the test execution on specific tests or temporarily skip failing tests without needing to delete or modify the test code. However, reviewers have noted the presence of these annotations and have requested their removal. Contributors can use these annotations during test development, but sometimes forget to remove them when submitting a PR for code review.

What is the potential practical value of the guidelines for developers? Test files and GUI-based test files are modified and reviewed less frequently than production code, limiting reviewers' opportunities to develop comparable expertise [35, 205]. A lack of experience in code reviews has a negative long-term effect on software quality [156]. Consequently, the guidelines can support less experienced contributors and reviewers to improve the quality of code reviews.

During the review process, these guidelines can help reduce the cognitive load on reviewers by providing clear criteria to follow. However, as demonstrated by Alegroth, Petersen, and Tinnerholm [14], programming guidelines can sometimes increase cognitive load for testers instead. Using automation to prepare and assist in the code review process, as observed in our sampled repositories, could reduce the required cognitive load, although it will not eliminate it. Continuous integration pipelines should enforce syntactic and structural checks, such as linting and test coverage thresholds. Additionally, pull request templates should encourage authors to provide context, rationale, and validation steps for GUI-based tests. In general code reviews, automation, if applied in a sensible manner, has in many cases a positive impact on the review process [21].

The limited discussion observed for guideline G8.1 may indicate a blind spot during review. In particular, the need for additional context (G3.1) and rationale (G3.2) suggests insufficient understanding of the requirements that motivated the changes. One potential explanation is the use of open-source repositories, which may not be created through a software engineering process with proper requirements elicitation.

Comparison of results to other studies Ricca and Stocco [183] surveyed the grey literature to identify best practices in end-to-end web test automation. Several of the practices they identified align closely with our findings from the analysis of code review comments and project guidelines. Table 4.6 shows this alignment.

Among the technical best practices discussed, the proper handling of synchronization between the web application and the test code was highlighted most frequently. This practice aligns with our guideline G5.6, which advises against using explicit waits to wait for events from the system under test. Improper handling of synchronization via

Table 4.6: Overlap between best practices in end-to-end web test automation (as identified by Ricca and Stocco [183] and our guidelines

Best practice	Guideline
Use Continuous Integration (CI)	G1.1
Produce detailed reports	G4.1
Use appropriate naming and code conventions	G5.4
Create robust/proper locators/selectors	G5.6
Manage the synchronization with the app	G5.6
Take/use screenshots	G6.1
Keep the tests atomic and short	G7.1
Use the Page Object Pattern	G9.1
Focus on reusable test code	G9.2

waits can not only cause an unnecessary increase in test execution time, but also may introduce flaky behavior due to the waits being non-deterministic [183].

Fulcini et al. [89] presented guidelines to avoid test smells specific to GUI-based testing. Test smells, which build upon the concept of code smells [85], refer to patterns in test code that hinder readability, maintainability, and overall quality. Based on their guidelines, they developed a linter to identify test smells. Table 4.7 illustrates how our guidelines overlap with theirs. Interestingly, their guidelines, R3 to R7, provide a more detailed set of recommendations for using locators. However, they do not mention the role-based locator, which has often been requested in the code review comments we observed. They found that the most frequently violated guidelines in their sampled GitHub repositories were the use of global variables in test cases, followed by the discouraged use of XPath [50] locators. While issues related to XPath locators have been noted in other studies [144], we only encountered them in one pull request comment. Similarly, discussions about global variables were observed in a few pull requests. A potential explanation for this discrepancy is the selection of repositories using Selenium as the testing tool, rather than our choice of Playwright and Cypress. Nevertheless, highlighting test smells automatically would help identify potential maintainability issues in test code during code reviews.

Table 4.7: Overlap between guidelines for GUI-based testing maintenance by Fulcini et al. [89] and our guidelines

Guideline by Fulcini et al. [89]	**Guideline**
R2: Keep test cases as simple and short as possible	G7.1
R3: Prefer to use locator by Id, CSS locators and Xpath when not available, in that order	G5.6
R4: Use relative XPath in place of absolute XPath	G5.6
R5: Do not use link locators	G5.6
R6: Use tag locators only for multiple elements	G5.6
R7: Use relative URLs locator instead of absolute ones	G5.6
R8: For elements and variables use name that mirrors functionalities	G5.4
R9: Keep names of variables clear to everyone	G5.4
R19: Run linters to detect anti-pattern	G1.1
R20: Adopt Page Object Pattern	G9.1
R21: Avoid Thread.sleep statements by turning fixed-time waits into condition-based ones	G5.6

4.6 Threats to Validity

We discuss the threats to the validity of this study in terms of construct validity, internal validity, external validity, and reproducibility. Further, we discuss the three types of threats that must be addressed in software repository mining studies according to Vidoni [218]: repository bias, data bias, and information bias.

4.6.1 Construct Validity

A central threat to construct validity is that the absence of review comments does not imply that reviewers neglected aspects covered by guidelines or deemed them irrelevant. This illustrates the broader problem that the absence of evidence is not evidence of absence. To strengthen construct validity, we examined pull request templates and information supplied by automation tools (e.g., GitHub Actions, bots) as complementary data sources. Reviewers are unlikely to comment on aspects already covered by contributors or automated feedback, such as test coverage reports. The involved data constraints the extent to which conclusions can be drawn about the value provided by the guidelines. Without a comparative study design, a causal attribution is not feasible. As such, the objective of this manuscript is to gain insights into the presence and use, not to measure the concrete value of the guidelines. A comparative study is envisioned for future work.

4.6.2 Internal Validity

A limitation of this study is the qualitative nature of the code review comment analysis. To mitigate researcher bias, the two main authors jointly mapped comments to guidelines in multiple sessions. Each session was limited to a maximum of 90 minutes to avoid researcher fatigue from analyzing a set of 1000 pull requests. Because this process was performed collaboratively, inconsistencies in mapping did not emerge. We consider this manual mapping a strength relative to automated methods. For example, we evaluated an LLM-based classification approach but rejected it due to its low agreement with the manual results, which would have introduced information bias [218].

Another potential threat arises from ambiguous pull request comments. When a comment's intent was unclear, we inspected the associated pull request in GitHub to recover the full context of the change.

Comments that could not be reasonably mapped to any guideline were not forced into a guideline, thereby preserving the integrity of the mapping.

4.6.3 External Validity

Repository Bias This study collects data exclusively from publicly available open-source repositories hosted on GitHub. Alternative platforms such as GitLab or GitBucket are excluded, which introduces a repository bias [218]. Given GitHub's dominance, with approximately 350 million repositories, this exclusion is unlikely to limit the relevance of our findings when sampling 100 popular repositories for the analysis. GitHub is also the most common source of repositories for MSR studies [218]. It's important to note that these results may vary for less popular repositories. Additionally, this approach indirectly excludes very new repositories, as they have not yet had the opportunity to gain popularity.

In total, we gathered data from 878 002 pull requests across these 100 popular repositories. This reflects the collaboration among many experienced developers and testers in real-world projects, and we believe that this sample is representative of code review practices.

Open-source software projects differ significantly from closed-source, commercial systems, making it inappropriate to directly transfer insights from open-source to closed-source environments. Open-source projects are typically developed by distributed teams that rely on asynchronous, text-based communication rather than face-to-face interactions. Contributors to open-source projects may participate infrequently or irregularly since they are not contractually tied to the project through employment agreements [25, 26, 141].

Studies by Rigby and Bird [184] and Bosu et al. [41] compare aspects of code reviews in open-source and closed-source systems. Generally, both studies found a substantial

overlap between the code review processes in these two types of software development. However, a notable difference lies in the relationships between code review participants, which are more important in open-source projects, while knowledge transfer is a key focus in closed-source projects [41]. We also view knowledge transfer as an important function of code reviews for GUI-based testing. As a result, findings derived from analyzing open-source repositories may not be applicable to proprietary, closed-source environments, where team organization, development workflows, and quality assurance mechanisms can vary considerably.

Data bias A data bias occurs when there are threats to the validity of the data extracted from the repositories [218]. In our study, this threat emerges from the selection of testing tools. We focus on Playwright and Cypress, excluding Selenium. Although Selenium remains prevalent in industry and academic literature, its omission reflects our emphasis on modern frameworks. Tool-specific features may influence the mapping to our proposed guidelines.

Based on the selection of tools and our focus on web-based applications, the diversity of programming languages is narrowed down to JavaScript and TypeScript. While Playwright can be used with other languages, such as Python or Java, our sampled repositories only use JavaScript and TypeScript. This potentially limits the transferability of our results to projects in other languages with different tooling and community practices.

4.6.4 Reproducability

We provide a replication package containing all scripts used for data collection and a set of intermediate artifacts. This package enables researchers to inspect, verify, and repeat the data-gathering process under the same conditions, thereby increasing transparency. By documenting the exact data collection scripts and preserving the raw outputs of intermediate steps, we reduce ambiguity and mitigate threats to reproducibility.

Data volatility Temporal factors introduce data volatility, which may threaten both the validity and reproducibility of our results. First, the GitHub API evolves over time, altering query structures, rate limits, and the availability of metadata. Such changes can render past data collection procedures irreproducible or introduce inconsistencies across studies [132]. Second, popular repositories are in continuous development, as contributors frequently merge pull requests, refactor code, and expand test suites. Consequently, metrics such as the number of open pull requests or the size of the test base may differ substantially between two collection points, even if separated by only a few days.

To mitigate these threats to some extent, we preserved snapshots of time-intensive intermediate steps, including the complete crawling of pull requests and their metadata from 100 repositories. These snapshots provide stable reference points that support reproduction and allow researchers to validate our results despite the inherent volatility of the original data source.

4.7 Conclusion

In this study, we collected code review data from 100 repositories to empirically evaluate proposed guidelines for GUI-based test reviews. In total, we gathered data from 878 002 pull requests, of which only 1.9% (16 866 PRs) of all PRs modify GUI-based test files, and just 0.24% (2084 PRs) contained comments or discussions.

We manually analyzed 1000 pull requests, of which 808 included code review comments that either directly or indirectly reflected our guidelines. We found evidence supporting 25 out of the 33 proposed guidelines, drawing insights from review comments, review templates, and GitHub Actions. Notably, Guideline G5.6, which relates to locators and the behavior of waiting and timeouts, was the most frequently observed, which we then refined.

Additionally, we provide concrete examples of guidelines for the analyzed testing tools, reducing the ambiguity of the guidelines, along with an overview of which guidelines are supported through automation. Finally, we compare our findings with those from other studies.

Our results suggest that many concerns captured by the proposed guidelines are articulated in practice, indicating practical relevance and coverage for GUI-based test reviews. Our refinements of guidelines and examples could help teams adopt more consistent review practices and inform tool builders about automatable checks.

For future work, the concrete examples of the guidelines can inform experimental studies aimed at assessing the impact of explicitly adopting these proposed guidelines on the quality of GUI-based test cases and the overall code review process. Refined guidelines and concrete examples of their application could help mitigate the problems previously encountered with experimental methodologies. Additionally, it would be valuable to explore the potential for automation to assist in the code review process and its effects.

Data availability

The datasets generated during and/or analysed during the current study are published on Zenodo: https://doi.org/10.5281/zenodo.17104677 or [30].

Quo Vadis, Code Review?

5

This chapter is based on Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript submitted to the *Journal of Systems and Software (New Ideas and Trends Papers)*. 2025.

Abstract

Code review has long been a core practice in collaborative software engineering. In this research, we explore how practitioners reflect on code review today and what changes they anticipate in the near future. We then discuss the potential long-term risks of these anticipated changes for the evolution of code review and its role in collaborative software engineering.

5.1 Introduction

> "Nothing is constant except change."
>
> —Heraclitus

During the early days of software engineering, code was often designed and implemented by individual developers working in isolation. As the software systems became more and more complex, software engineering became a collaborative effort requiring many developers with diverse skill sets and specializations, organized in different teams, and later even distributed around the globe. To maintain and enhance a collective understanding of the codebase and the changes to it, developers need to discuss a code change before it gets merged into the codebase. Over the years, these discussions have taken various forms, ranging from the formal and heavyweight *code inspections* in the 1980s [78] to synchronous and steady exchanges between two developers in *pair programming* [224]. Nowadays, the asynchronous and less formal discussions around code changes are known as *code reviews*, a practice that has seen widespread adoption in collaborative software engineering [32, 71].

However, change is constant in software engineering—and code review is no exception. Advances in large language models (LLMs) and new regulatory frameworks, such as the EU Cyber Resilience Act, are beginning to influence how code reviews are conducted.

In this article, we take a glimpse into the future of code review. To that end, we begin by analyzing how practitioners reflect on code review today and what they expect to change, based on a survey of 92 professional software developers from four large software-driven companies, each representing a distinct organizational and software engineering context. Building on this analysis, we discuss potential long-term risks for the future role of code review as a core practice in collaborative software engineering.

5.2 Study Design

To understand how developers experience code review today and how they expect it to change, we surved a sample of 92 practitioners.

Table 5.1: Organizational and Software Engineering Contexts and Participant Numbers by Company

Attribute	**SAP**	**Ericsson**	***A Bank***	**JetBrains**
Industry Domain	Enterprise Software	Telecommunications	Banking and Finance	Developer Tools
Business Model	Software products (B2B SaaS, ERP)	Infrastructure provider (network equipment, telecom software)	Financial services (retail and investment banking)	Commercial software vendor (IDEs and tooling)
Primary Software Focus	Customer-facing enterprise platforms	Embedded and network control software	Internal financial transaction and risk management systems	Developer productivity and language tooling
Number of Employees	>100,000	>100,000	>25,000	>2,200
Number of participants	32	24	25	11

To gain a diverse perspective on the future of code review grounded in real-world practice, we employed *quota sampling*, a non-random, two-stage strategy. First, we purposively selected four relevant companies representing varied organizational and software engineering contexts. Then, we aimed to recruit approximately 25 professional developers from each company, closing recruitment after eight weeks or once the quota was reached.

Our sample includes 92 developers from (1) one of the world's largest enterprise software providers (SAP), (2) a globally leading provider of telecommunications infrastructure (Ericsson), (3) one of Europe's largest banks, which has requested anonymity, and (4) a mid-sized vendor of professional developer tools, including widely used IDEs (JetBrains). The number of participants per company varied slightly. See Table 5.1 for an overview of the participating organizations, their contexts, and participant numbers. Though not statistically representative and with varying participation per company, the sample captures diverse industry perspectives on code review.

To capture developers' current code review practices and their expectations for how the practice may evolve over the next five years, we asked six questions:

1. How many hours on average do you currently review code per week? (numerical response)
2. Do you expect to spend more, less, or the same amount of time on code review in five years (compared to today)? (Multiple choice)
3. What software artifacts do you review today? (Multiple selection)
4. What software artifacts do you anticipate reviewing in five years? (Multiple selection)
5. What major changes do you anticipate in code review in five years? (Open-ended)
6. Based on those changes, what implications for code review do you see? (Open-ended)

The survey covered six types of software artifacts covered in code review: Production code, Test code, Parameter and configuration files, Documentation, and GUI-based test code (end-to-end testing). Respondents could also add further artifact types or choose to select none.

The survey was conducted at the end of 2024 and the beginning of 2025 via an online questionnaire, shared through internal communication channels at the respective companies. Participation was both voluntary and anonymous, allowing respondents to share their perspectives freely. To further protect confidentiality, we deliberately excluded demographic questions such as age, gender, or company affiliation, and

reported the results at the sample level rather than by individual company. Participants were informed about the study's purpose and assured that their participation was voluntary, and their responses would remain anonymous.

Quantitative and multiple-choice responses were analyzed using descriptive statistics. Open-ended responses were examined using *thematic analysis* [51] to uncover common patterns and emerging trends.

All anonymized data and analysis scripts are publicly available at Zenodo[1].

5.3 Code Review Five Years From Now

What do developers actually do in code review today and how do they think that will change? In this section, we report the results of our survey, capturing how developers currently experience code review today and how they expect it to change in the coming years. See Section 5.2 for more details on study design. The analysis is structured around three central questions from our survey: the *time* they currently spend on code review and how they expect that to change; the *types of artifacts* they review today versus those they anticipate reviewing in the future; and *who* they expect to be involved in the code review process going forward.

5.3.1 Not Less Time

Currently, respondents report spending a median of approximately three hours per week on code reviews. To contextualize this finding, we compare it against two external data sources: first, a 2013 survey by Bosu et al. involving 416 Microsoft developers; and second, the 2019 Stack Overflow Developer Survey, which includes responses from 49 790 developers[2]. Figure 5.1 shows the cumulative distributions of weekly review hours across all three data sources. The distribution in our sample is consistently lower compared to both the 2013 Microsoft study by Bosu et al. and the 2019 Stack Overflow Developer Survey, which show strikingly similar distributions with a median of around four hours spent on code reviews. The reasons for this difference are not evident from our data. Possible explanations may include a decline in the perceived importance of code review over the past years, differences in domain-specific practices, or inaccuracies in self-reported estimates.

A vast majority of practitioners do not expect to spend less time on code review in the coming years. According to our survey:

- 42.5 % expect to spend *more time*,
- 32.5 % expect to spend *about the same amount* of time, and
- 25.0 % foresee spending *less time*.

[1] https://doi.org/10.5281/zenodo.17359762
[2] https://survey.stackoverflow.co/2019

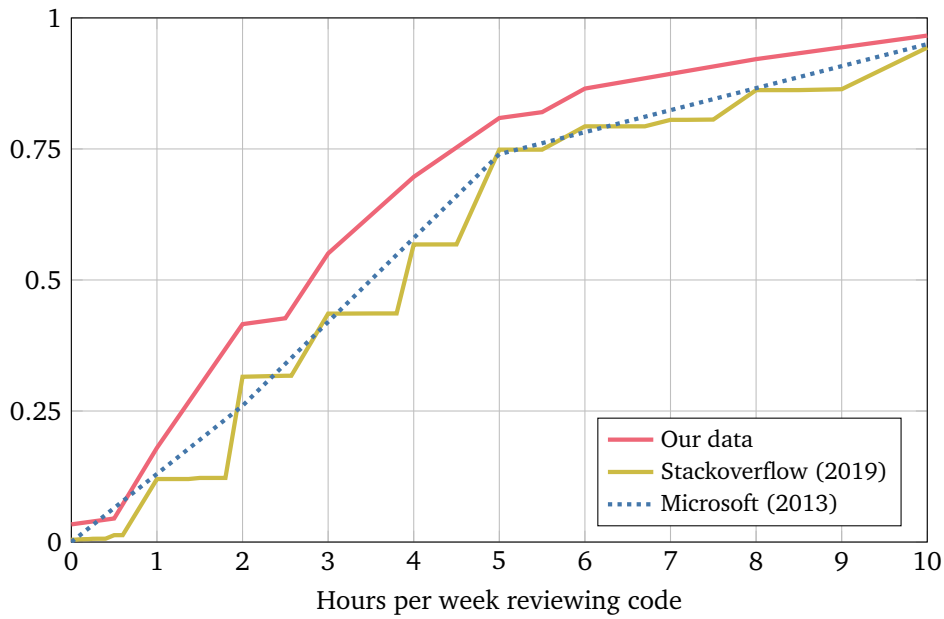

Figure 5.1: Cumulative distribution of hours spent on code review today (≤ 10 hours). In the absence of raw data, we approximated the cumulative distribution from the reported ordinal-scale results at Microsoft [41].

Although those results suggest that code review might play even a more central role in collaborative software engineering in the near future, we are cautious about extrapolating this insight to the entire software industry. The distributions in our sample are consistently lower than those in our reference datasets—though these datasets are older—which raises the possibility that the companies in our study are simply catching up to broader industry standards rather than indicating a widespread shift in code review practices.

5.3.2 Broader Coverage

Practitioners expect to review a broader range of artifacts in the near future in comparison to today. Over the next five years, we see an increasing interest in reviewing more artifact types, including production code, test code, configuration files, documentation, and GUI test code. Notably, the share of practitioners who currently do not review any artifacts (around 6%) is expected to decrease.

Production code will remain the primary artifact under review in the future. There is also a broad consensus that parameter and configuration files, documentation, and test code will continue to be reviewed regularly. A particularly notable trend is the comparatively sharp increase in the future prediction of review focus on GUI-based test code, suggesting a growing recognition and integration of it in future software development practices.

The connected dot graph in Figure 5.2 highlights the expected changes in the coverage of reviewed artifacts in the future.

5.3.3 The New Code Review Participant

Given the current hype around generative AI, it is hardly surprising that nearly all practitioners, when asked about anticipated major changes in future code reviews, expect large language models to become active participants in the process. This trend is also highlighted by Davila, Melegati, and Wiese [60], whose gray literature review identified strong practitioner interest in both proposing novel generative AI–based solutions and adopting established tools such as ChatGPT for code review.

In contrast to the deterministic, rule-based, or pattern-matching bots that long assisted in code reviews [21], survey respondents envision LLMs taking a way more active role: An LLM might act as the code reviewer, assessing the code written by humans and providing feedback through an *automated review*. The anticipated autonomy of LLMs varies and can be viewed along a spectrum, ranging from supporting human reviewers as an additional assistant (“*AI might take over some parts of the code review process.*”) to fully conducting the review independently (“*complete code review with the help of AI*”).

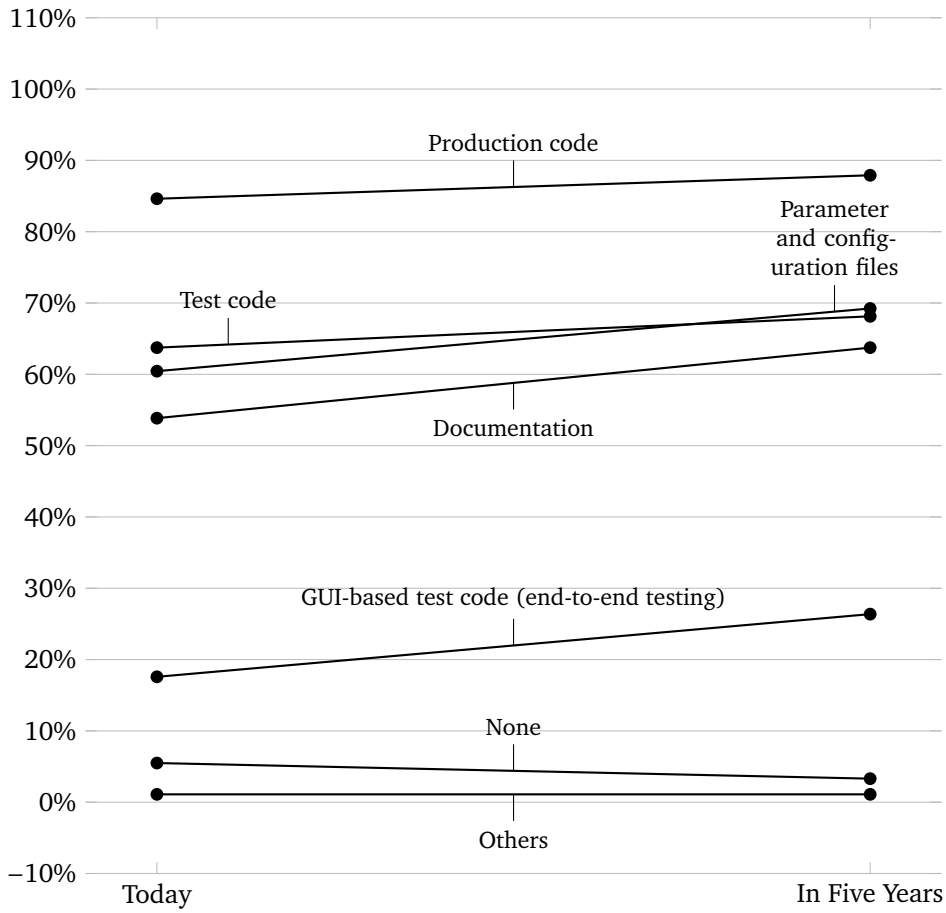

Figure 5.2: Practitioners anticipate reviewing a broader range of artifacts in the future, with notable growth in attention to GUI-based test code and a decline in those reviewing no artifacts at all.

Alternatively, LLMs could serve as the authors, *automatically generating code* that is then reviewed by humans, keeping developers "in the loop": "*Code will be generated by generative-AI based tools, but reviewed by humans. A review by humans will be mandatory to avoid any law suits on the code development organization.*" Again, the depth of involvement in code authoring lies on a spectrum, from assisting developers to autonomously proposing and integrating code changes. The amount and pace of generated code by AI will force us to review the code in a different way. In this scenario, concerns arise about the long-term impact on developer expertise. As one participant warned: "*I am a bit concerned about the future generation of developers as they will have less experience actually coding.*"

Two respondents also considered that automated code review intersects with automated code generation and "*AI [is] everywhere*". We refer to the intersection where both code generation and review are fully handled by LLMs and eliminate the need for human involvement as *unsupervised software engineering*. Yet this shift comes with caveats, as one participant explicitly warned:

> *"We will have to decide if we want to write code with AI or review code with AI [...] I don't think both at the same time will work, because there can be instances, where the programmer generates bad code with the AI, and a mistake is kept in, and because of time shortage the reviewer also uses AI to review the same code, and the AI doesn't catch the mistake because it generated it."*

Expressing concern about the future towards unsupervised software engineering, one developer stated: "*I expect that there will be more generated code of a worsen quality. I expect that there will be a lot of AI code review systems, which will be pretty useless.*"

We also encountered fundamental opposition to the use of AI in code review—whether as author or reviewer—as one participant put it: "*As long as AI is involved, it is likely to be a more tedious process [code review] than before.*" We refer to this stance as human-led software engineering, a position in direct contrast to a future of fully unsupervised software engineering without human oversight.

Based on practitioner input, we summarize the anticipated roles of AI in code review as a spectrum of involvement, gradually ranging from fully human-led to fully LLM-led, in both the roles of code author and reviewer. As illustrated in Figure 5.3, four possible futures of software engineering emerge from this gradual interplay between humans and LLMs, each reflecting a distinct balance in the roles of author and reviewer.

5.4 Doesn't LGTM

Our survey highlights two key expectations from practitioners about the future of code review. First, code review will remain a central practice in collaborative software engineering. Developers anticipate that it will continue to demand the same or even

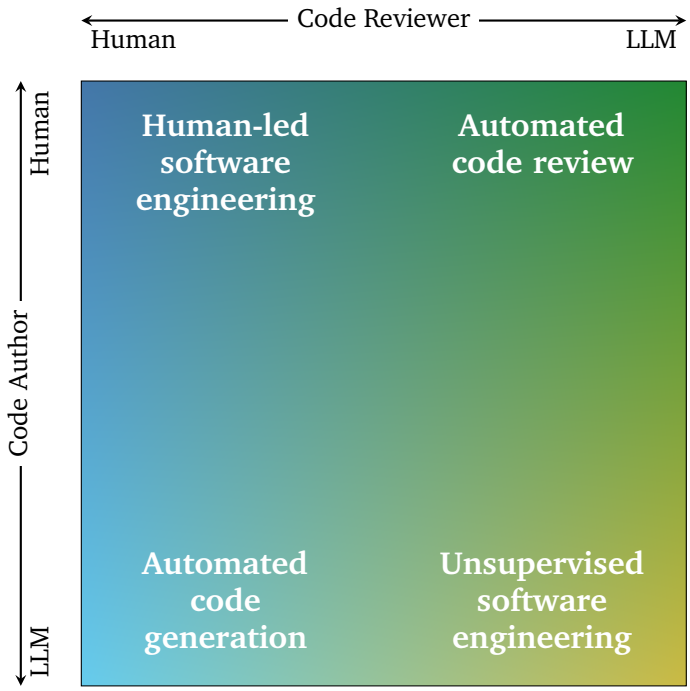

Figure 5.3: Practitioners expect the roles of code author and code reviewer to lie on continuous spectra, each ranging from fully human-led to fully LLM-led. Based on this human–AI interplay in code review, four possible futures of collaborative software engineering begin to take shape.

more time in the coming years, while expanding to include a broader range of artifacts. Second, with the growing integration of LLMs, the traditionally human-driven nature of code review with developers leading discussions, making judgments, and approving changes [21, 72] is expected to shift.

In this section, we invite the reader to envision the future that practitioners anticipate, one where AI takes a central role in code review. Based on this premise, we discuss three forms of erosion that are likely to affect collaborative software engineering broadly, but surface first in code review.

5.4.1 Erosion of Understanding

At its core, code review, and all the many review guidelines that support it [32], ultimately boils down to a single key question: *Would I be able to maintain this code change in the future if the original author were no longer available?* If the answer is no, it signals that the code may be too complex, poorly documented, badly structured, or unclear, and likely needs revision. By introducing a second pair of eyes, code review not only enhances maintainability and consistency but also acts as a mechanism for capturing and preserving human understanding of the software system and the broader implications of ongoing changes.

Now, imagine a future where an LLM handles code reviews and optimizes code to its own capacity to "understand" it. Ironically, this could lead to code becoming less maintainable over time—from a human perspective. What is readable and logical to an LLM may be opaque and not intuitive to a human developer. As a result, human understanding of the changes in the evolving codebase could gradually erode, leaving only other LLMs capable of maintaining it. And this might happen subtle: Even for AI-assisted code reviews where we keep the human in the process, reviewers focus more on the annotated code section and potentially overlooking issues elsewhere which could limit a comprehensive understanding [215].

Should we, then, make it a priority for humans to review code that is authored (partially or entirely) by LLMs to ensure long-term human maintainability of the codebase, not just short-term AI comprehension? This approach also has its cons. When developers routinely defer to LLM-generated solutions—especially without critically engaging with or questioning them—we risk fostering a *passive development culture*. Combined with an increased volume of LLM-generated code and thus review workload, developers may be forced to treat code as a black box—accepting solutions that seem to work, but without fully grasping their structure, intent, or long-term consequences.

Code is more than just functional software. It is a shared language among developers. If only machines can understand it, we risk losing not only our connection to the people we build systems for, but also our ability to meaningfully control the systems we create.

5.4.2 Erosion of Accountability

Code review is not just about finding bugs—it is about reinforcing accountability, ownership, and responsibility in software development [20]. By requiring developers to justify their changes and reviewers to critically evaluate them, code review ensures that every line of code has a clear owner—someone responsible for its quality, security, and long-term maintainability [230]. In contrast, AI systems can neither be held responsible nor accountable (let alone liable) in the same way, creating a gap in ownership of the codebase.

This gap places software companies in a legal limbo in the context of regulatory compliance. Emerging regulations such as the EU's *Cyber Resilience Act*, which mandates effective and regular reviews of software systems with a focus on security, or the *Transfer Pricing Guidelines* by the OECD, which call for a more precise definition of code ownership, both place accountability at the forefront of regulatory priorities and expectations [70].

As AI systems take on a greater role in collaborative software engineering, ensuring clear human accountability will be critical [193], not only for maintaining code quality, but also for meeting the growing legal and regulatory demands placed on software organizations.

5.4.3 Erosion of Trust

Trust is fundamental for code review—just as it is in any form of human collaboration. With the increasing involvement and reliance on LLMs, the foundation for trust is at risk. It is becoming increasingly unclear who truly authored a piece of code or a code review. Was it written by a human? By an LLM? Or, perhaps more worryingly, a human blindly parroting LLM-generated content without critical thought or understanding? This growing uncertainty signals a deeper issue—a gradual *erosion of trust* in the authenticity and accountability of the review process.

The erosion of trust in code review is not a distant threat—it is a problem we already face. In his blog post titled "The I in LLM stands for intelligence," Daniel Stenberg, the maintainer of the curl project, describes how some individuals use LLMs to analyze curl's code and generate security vulnerability reports [209]. These AI-generated reports often appear legitimate but are shallow and inaccurate, forcing maintainers to spend valuable time assessing and eventually dismissing them. This problem highlights the growing challenges to distinguish meaningful contributions from synthetic.

We fear a similar risk in code review. When reviewers cannot distinguish between genuine human insight and automated suggestions, feedback becomes superficial and potentially harmful. Code review loses its purpose and deviates from its critical function, becoming an empty shell—a routine for the sake of routine, rather than being a meaningful safeguard for code quality and collaboration.

5.5 Conclusion

Code review has long been at the heart of collaborative software engineering where experience, reasoning, and thoughtful feedback drive the evolution of software systems. And our results indicate that code review will not disappear in the foreseeable future. In fact, developers expect to spend the same or even more time reviewing code, and reviewing more artifacts in the process.

While code review will remain essential, it will evolve. The results of our survey clearly show that LLMs are expected to become ubiquitous and to take on more tasks. The question remains to what extent code review shall be delegated to AI or kept in human hands? Without unnecessarily demonizing the LLMs, we raise concerns for the consequences of routine over-reliance on LLMs in code review—risks of eroding human understanding, accountability, and trust, ultimately reducing code review to an empty shell.

As software engineering is in constant evolution, raising questions about how to improve efficiency through automation, we advocate for a nuanced approach to code review. Human understanding (even if not very efficient), accountability (even if not on today's agenda), and trust (even if not easily measured) ensure that we keep control of the long-term evolution of our software systems and remain resilient in the face of future changes, whether technical or transformational.

We advocate for further research to explore the implications of integrating LLMs and other AI technologies into code review, as we believe code review is where the effects of LLM adoption in collaborative software engineering are likely to surface first. An over-reliance on AI gradually shifts the focus of understanding from human teams to the LLM—an entity that, despite its capabilities, cannot be held accountable, reason contextually, or justify its decisions beyond probabilistic associations. To preserve the integrity of software engineering as a collaborative, transparent, trustworthy, and responsible engineering practice, we must ensure that code review remains grounded in human understanding and oversight.

Acknowledgement

We thank all participants of the survey. This work was supported by the KKS Foundation through the SERT Project (Research Profile Grant 2018/010) at Blekinge Institute of Technology.

References

[1] A. Frank Ackerman, Priscilla J. Fowler, and Robert G. Ebenau. "*Software Inspections and the Industrial Production of Software*". In: *Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives*. Darmstadt, Germany: Elsevier North-Holland, Inc., 1984, pp. 13–40.

[2] A.F. Ackerman, L.S. Buchwald, and F.H. Lewski. "*Software Inspections: An Effective Verification Process*". In: *IEEE Software* 6.3 (May 1989), pp. 31–36. DOI: 10.1109/52.28121.

[3] Richard J. Adams, Palie Smart, and Anne Sigismund Huff. "*Shades of Grey: Guidelines for Working with the Grey Literature in Systematic Reviews for Management and Organizational Studies: Shades of Grey*". In: *International Journal of Management Reviews* 19.4 (October 2017), pp. 432–454. DOI: 10.1111/ijmr.12102.

[4] Gustavo Jorge Martins De Aguiar. *Distinguishing between Method and Methodology in Academic Research*. 2024. DOI: 10.2139/ssrn.4857923.

[5] Pekka Aho and Tanja Vos. "*Challenges in Automated Testing Through Graphical User Interface*". In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Vasteras: IEEE, April 2018, pp. 118–121. DOI: 10.1109/ICSTW.2018.00038.

[6] Adam Alami, Marisa Leavitt Cohn, and Andrzej Wasowski. "*Why Does Code Review Work for Open Source Software Communities?*" In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1073–1083. DOI: 10.1109/ICSE.2019.00111.

[7] Emil Alégroth. "*On the Industrial Applicability of Visual Gui Testing*". Licentiate Thesis. Chalmers University of Technology, 2013.

[8] Emil Alégroth and Robert Feldt. "*On the Long-Term Use of Visual Gui Testing in Industrial Practice: A Case Study*". In: *Empirical Software Engineering* 22.6 (December 2017), pp. 2937–2971. DOI: 10.1007/s10664-016-9497-6.

[9] Emil Alégroth, Robert Feldt, and Pirjo Kolström. "*Maintenance of Automated Test Suites in Industry: An Empirical Study on Visual GUI Testing*". In: *Information and Software Technology* 73 (May 2016), pp. 66–80. DOI: 10.1016/j.infsof.2016.01.012.

[10] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. "*Visual GUI Testing in Practice: Challenges, Problemsand Limitations*". In: *Empirical Software Engineering* 20.3 (June 2015), pp. 694–744. DOI: 10.1007/s10664-013-9293-5.

[11] Emil Alegroth and Javier Gonzalez-Huerta. "*Towards a Mapping of Software Technical Debt onto Testware*". In: *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Vienna, Austria: IEEE, August 2017, pp. 404–411. DOI: 10.1109/SEAA.2017.65.

[12] Emil Alégroth, Kristian Karl, Helena Rosshagen, Tomas Helmfridsson, and Nils Olsson. "*Practitioners' Best Practices to Adopt, Use or Abandon Model-based Testing with Graphical Models for Software-intensive Systems*". In: *Empirical*

Software Engineering 27.5 (September 2022), p. 103. DOI: 10.1007/s10664-022-10145-2.

[13] Emil Alegroth, Arvid Karlsson, and Alexander Radway. "*Continuous Integration and Visual GUI Testing: Benefits and Drawbacks in Industrial Practice*". In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. Vasteras: IEEE, April 2018, pp. 172–181. DOI: 10.1109/ICST.2018.00026.

[14] Emil Alegroth, Elin Petersen, and John Tinnerholm. "*A Failed Attempt at Creating Guidelines for Visual GUI Testing: An Industrial Case Study*". In: *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. Porto de Galinhas, Brazil: IEEE, April 2021, pp. 340–350. DOI: 10.1109/ICST49551.2021.00046.

[15] Le An, Foutse Khomh, Shane Mcintosh, and Marco Castelluccio. "*Why did this reviewed code crash? An empirical study of mozilla firefox*". In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE. 2018, pp. 396–405. DOI: 10.1109/APSEC.2018.00054.

[16] Florian Angermeir, Maximilian Amougou, Mark Kreitz, Andreas Bauer, Matthias Linhuber, Davide Fucci, Fabiola Moyón C., Daniel Mendez, and Tony Gorschek. "*Reflections on the Reproducibility of Commercial LLM Performance in Empirical Software Engineering Studies*". Accepted at *International Conference on Software Engineering (ICSE '26)*. 2025.

[17] Atlassian. *Bitbucket | Git solution for teams using Jira*. 2022. URL: https://bitbucket.org (visited on 10/17/2022).

[18] Aybuke Aurum, Håkan Petersson, and Claes Wohlin. "*State-of-the-art: Software Inspections after 25 Years*". In: *Software Testing, Verification and Reliability* 12.3 (September 2002), pp. 133–154. DOI: 10.1002/stvr.243.

[19] GitLab B.V. *Code review guidelines*. 2025. URL: https://docs.gitlab.com/ee/development/code_review.html (visited on 10/09/2025).

[20] Alberto Bacchelli and Christian Bird. "*Expectations, Outcomes, and Challenges of Modern Code Review*". In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 712–721. DOI: 10.1109/ICSE.2013.6606617.

[21] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. "*Modern Code Reviews—Survey of Literature and Practice*". In: *ACM Transactions on Software Engineering and Methodology* 32.4 (October 2023), pp. 1–61. DOI: 10.1145/3585004.

[22] Richard A. Baker. "*Code Reviews Enhance Software Quality*". In: *Proceedings of the 19th International Conference on Software Engineering - ICSE '97*. Boston, Massachusetts, United States: ACM Press, 1997, pp. 570–571. DOI: 10.1145/253228.253461.

[23] Faruk Balci, Dilruba Sultan Haliloglu, Onur Sahin, Cankat Tilki, Mehmet Ata Yurtsever, and Eray Tuzun. "*Augmenting Code Review Experience Through*

Visualization". In: *2021 Working Conference on Software Visualization (VISSOFT)*. Luxembourg: IEEE, September 2021, pp. 110–114. DOI: 10.1109/VISSOFT52517.2021.00021.

[24] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. "*Graphical User Interface (GUI) Testing: Systematic Mapping and Repository*". In: *Information and Software Technology* 55.10 (October 2013), pp. 1679–1694. DOI: 10.1016/j.infsof.2013.03.004.

[25] Ann Barcomb, Klaas-Jan Stol, Brian Fitzgerald, and Dirk Riehle. "*Managing Episodic Volunteers in Free/Libre/Open Source Software Communities*". In: *IEEE Transactions on Software Engineering* 48.1 (January 2022), pp. 260–277. DOI: 10.1109/TSE.2020.2985093.

[26] Ann Barcomb, Klaas-Jan Stol, Dirk Riehle, and Brian Fitzgerald. "*Why Do Episodic Volunteers Stay in FLOSS Communities?*" In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 948–959. DOI: 10.1109/ICSE.2019.00100.

[27] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K. Lahiri. "*Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets*". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 134–144. DOI: 10.1109/ICSE.2015.35.

[28] Andreas Bauer. *Artifact package: When GUI-based Testing Meets Code Reviews*. Zenodo, September 2025. DOI: 10.5281/zenodo.17158220.

[29] Andreas Bauer and Emil Alégroth. "*We Tried and Failed: An Experience Report on a Collaborative Workflow for GUI-based Testing*". In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Dublin, Ireland: IEEE, April 2023, pp. 1–9. DOI: 10.1109/ICSTW58534.2023.00015.

[30] Andreas Bauer and Florian Angermeir. *Replication package: The Prevalence of Code Review Guidelines for GUI-Based Testing in Open-Source*. Version 1.0.0. Zenodo, September 2025. DOI: 10.5281/zenodo.17104677.

[31] Andreas Bauer, Florian Angermeir, Emil Alégroth, and Seth Anglert. "*The Prevalence of Code Review Guidelines for GUI-Based Testing in Open-Source*". Manuscript submitted to *Information and Software Technology*. 2025. DOI: 10.2139/ssrn.5547512.

[32] Andreas Bauer, Riccardo Coppola, Emil Alégroth, and Tony Gorschek. "*Code Review Guidelines for GUI-based Testing Artifacts*". In: *Information and Software Technology* 163 (November 2023), p. 107299. DOI: 10.1016/j.infsof.2023.107299.

[33] Andreas Bauer, Julian Frattini, and Emil Alégroth. "*Augmented Testing to Support Manual GUI-based Regression Testing: An Empirical Study*". In: *Empirical Software Engineering* 29.6 (August 2024), p. 140. DOI: 10.1007/s10664-024-10522-z.

[34] Andreas Bauer, Nikolay Harutyunyan, Dirk Riehle, and Georg-Daniel Schwarz. "*Challenges of Tracking and Documenting Open Source Dependencies in Products: A Case Study*". In: *Open Source Systems*. Ed. by Vladimir Ivanov, Artem Kruglov, Sergey Masyagin, Alberto Sillitti, and Giancarlo Succi. Vol. 582. Cham: Springer International Publishing, 2020, pp. 25–35. DOI: 10.1007/978-3-030-47240-5_3.

[35] Andreas Bauer, Tomas Helmfridsson, Emil Alégroth, and Georg-Daniel Schwarz. "*When GUI-based Testing Meets Code Reviews*". Manuscript submitted to *Software Testing, Verification and Reliability*. 2025.

[36] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. "*A Faceted Classification Scheme for Change-Based Industrial Code Review Processes*". In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. Vienna, Austria: IEEE, August 2016, pp. 74–85. DOI: 10.1109/QRS.2016.19.

[37] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. "*An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance*". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. Trento, Italy: IEEE, September 2012, pp. 56–65. DOI: 10.1109/ICSM.2012.6405253.

[38] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[39] Stefan Berner, Roland Weber, and Rudolf K. Keller. "*Observations and Lessons Learned from Automated Testing*". In: *Proceedings of the 27th International Conference on Software Engineering - ICSE '05*. St. Louis, MO, USA: ACM Press, 2005, p. 571. DOI: 10.1145/1062455.1062556.

[40] Barry W. Boehm. "*Verifying and validating software requirements and design specifications*". In: *IEEE software* 1.1 (1984), p. 75.

[41] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. "*Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft*". In: *IEEE Transactions on Software Engineering* 43.1 (January 2017), pp. 56–75. DOI: 10.1109/TSE.2016.2576451.

[42] Amiangshu Bosu, Michaela Greiler, and Christian Bird. "*Characteristics of Useful Code Reviews: An Empirical Study at Microsoft*". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. Florence, Italy: IEEE, May 2015, pp. 146–156. DOI: 10.1109/MSR.2015.21.

[43] Virginia Braun and Victoria Clarke. "*Using Thematic Analysis in Psychology*". In: *Qualitative Research in Psychology* 3.2 (January 2006), pp. 77–101. DOI: 10.1191/1478088706qp063oa.

[44] Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. "*Web application tests with Selenium*". In: *IEEE software* 26.5 (2009), pp. 88–91.

[45] Raymond PL Buse and Westley R Weimer. "*Learning a metric for code readability*". In: *IEEE Transactions on software engineering* 36.4 (2009), pp. 546–558.

[46] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. "*GUI Testing Using Computer Vision*". In: *Proceedings of the 28th International Conference on Human Factors in Computing Systems - CHI '10*. Atlanta, Georgia, USA: ACM Press, 2010, p. 1535. DOI: 10.1145/1753326.1753555.

[47] Woei-Kae Chen and Jung-Chi Wang. "*Bad Smells and Refactoring Methods for GUI Test Scripts*". In: *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Kyoto, Japan: IEEE, August 2012, pp. 289–294. DOI: 10.1109/SNPD.2012.10.

[48] Yoonsik Cheon and Gary T. Leavens. "*A Simple and Practical Approach to Unit Testing: The JML and JUnit Way*". In: *ECOOP 2002 — Object-Oriented Programming*. Ed. by Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, and Boris Magnusson. Vol. 2374. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 231–255. DOI: 10.1007/3-540-47993-7_10.

[49] Aleksandr Chueshev, Julia Lawall, Reda Bendraou, and Tewfik Ziadi. "*Expanding the Number of Reviewers in Open-Source Projects by Recommending Appropriate Developers*". In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Adelaide, Australia: IEEE, September 2020, pp. 499–510. DOI: 10.1109/ICSME46990.2020.00054.

[50] James Clark and Steve DeRose. *XML Path Language (XPath)*. 1999. URL: https://www.w3.org/TR/1999/REC-xpath-19991116/ (visited on 07/10/2024).

[51] Victoria Clarke and Virginia Braun. "*Thematic analysis*". In: *The Journal of Positive Psychology* 12 (3 May 2017), pp. 297–298. DOI: 10.1080/17439760.2016.1262613.

[52] Cobertura. *A code coverage utility for Java*. 2022. URL: https://cobertura.github.io/cobertura/ (visited on 10/15/2022).

[53] Jason Cohen. "*Modern Code Review*". In: *Making Software: What Really Works, and Why We Believe It*. 1st ed. O'Reilly, 2010, pp. 329–336.

[54] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[55] Riccardo Coppola and Emil Alégroth. "*A Taxonomy of Metrics for GUI-based Testing Research: A Systematic Literature Review*". In: *Information and Software Technology* 152 (December 2022), p. 107062. DOI: 10.1016/j.infsof.2022.107062.

[56] Riccardo Coppola, Luca Ardito, Maurizio Morisio, and Marco Torchiano. "*Mobile Testing: New Challenges and Perceived Difficulties From Developers of the Italian Industry*". In: *IT Professional* 22.5 (September 2020), pp. 32–39. DOI: 10.1109/MITP.2019.2942810.

[57] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. "*Scripted GUI Testing of Android Open-Source Apps: Evolution of Test Code*

and Fragility Causes". In: *Empirical Software Engineering* 24.5 (October 2019), pp. 3205–3248. DOI: 10.1007/s10664-019-09722-9.

[58] Igor Crk and Timothy Kluthe. "*Assessing the contribution of the individual alpha frequency (IAF) in an EEG-based study of program comprehension*". In: *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE. 2016, pp. 4601–4604.

[59] Cypress.io, Inc. *Test. Automate. Accelerate.* 2025. URL: https://www.cypress.io (visited on 01/14/2025).

[60] Nicole Davila, Jorge Melegati, and Igor Wiese. "*Tales from the Trenches: Expectations and Challenges from Practice for Code Review in the Generative AI Era*". In: *IEEE Software* (2024), pp. 1–8. DOI: 10.1109/MS.2024.3428439.

[61] Nicole Davila and Ingrid Nunes. "*A Systematic Literature Review and Taxonomy of Modern Code Review*". In: *Journal of Systems and Software* 177 (July 2021), p. 110951. DOI: 10.1016/j.jss.2021.110951.

[62] Brian de Alwis and Jonathan Sillito. "*Why Are Software Projects Moving from Centralized to Decentralized Version Control Systems?*" In: *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 36–39. DOI: 10.1109/CHASE.2009.5071408.

[63] Marco De Luca, Anna Rita Fasolino, and Porfirio Tramontana. "*Investigating the Robustness of Locators in Template-Based Web Application Testing Using a GUI Change Classification Model*". In: *Journal of Systems and Software* 210 (April 2024), p. 111932. DOI: 10.1016/j.jss.2023.111932.

[64] Arie Deursen, Leon MF Moonen, A Bergh, and Gerard Kok. *Refactoring test code*. 2001.

[65] Emre Doğan and Eray Tüzün. "*Towards a taxonomy of code review smells*". In: *Information and Software Technology* 142 (2022), p. 106737.

[66] Liming Dong, He Zhang, Lanxin Yang, Zhiluo Weng, Xin Yang, Xin Zhou, and Zifan Pan. "*Survey on Pains and Best Practices of Code Review*". In: *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. Taipei, Taiwan: IEEE, December 2021, pp. 482–491. DOI: 10.1109/APSEC53868.2021.00055.

[67] Michael Dorner. "*Code Review as a Communication Network*". PhD thesis. Blekinge Institute of Technology, Department of Software Engineering, 2025, p. 188.

[68] Michael Dorner, Andreas Bauer, and Florian Angermeir. "*No Free Lunch: Research Software Testing in Teaching*". Under revision at *Journal of Open Research Software*. 2024.

[69] Michael Dorner, Andreas Bauer, Darja Šmite, Lukas Thode, Daniel Mendez, Ricardo Britto, Stephan Lukasczyk, Ehsan Zabardast, and Michael Kormann. "*Quo Vadis, Code Review?*" Manuscript submitted to the *Journal of Systems and Software (New Ideas and Trends Papers)*. 2025.

[70] Michael Dorner, Maximilian Capraro, Oliver Treidler, Tom-Eric Kunz, Darja Šmite, Ehsan Zabardast, Daniel Mendez, and Krzysztof Wnuk. "*Taxing Col-*

laborative Software Engineering". In: *IEEE Software* (2024), pp. 1–8. DOI: 10.1109/MS.2023.3346646.

[71] Michael Dorner, Daniel Mendez, Krzysztof Wnuk, Ehsan Zabardast, and Jacek Czerwonka. "*The Upper Bound of Information Diffusion in Code Review*". In: *Empirical Software Engineering* (June 2023).

[72] Michael Dorner, Daniel Mendez, Ehsan Zabardast, Nicole Valdez, and Marcin Floryan. *Measuring Information Diffusion in Code Review at Spotify*. 2024.

[73] A. E. Randel and K. S. Jaussi. "*FUNCTIONAL BACKGROUND IDENTITY, DIVERSITY, AND INDIVIDUAL PERFORMANCE IN CROSS-FUNCTIONAL TEAMS.*" In: *Academy of Management Journal* 46.6 (December 2003), pp. 763–774. DOI: 10.2307/30040667.

[74] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. "*Selecting empirical methods for software engineering research*". In: *Guide to advanced empirical software engineering*. Springer, pp. 285–311.

[75] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. "*Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies*". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Hangzhou, China: IEEE, February 2019, pp. 49–60. DOI: 10.1109/SANER.2019.8668024.

[76] Emelie Engström, Per Runeson, and Mats Skoglund. "*A Systematic Review on Regression Test Selection Techniques*". In: *Information and Software Technology* 52.1 (January 2010), pp. 14–30. DOI: 10.1016/j.infsof.2009.07.001.

[77] Ilker Etikan. "*Comparison of Convenience Sampling and Purposive Sampling*". In: *American Journal of Theoretical and Applied Statistics* 5.1 (2016), p. 1. DOI: 10.11648/j.ajtas.20160501.11.

[78] M. E. Fagan. "*Design and Code Inspections to Reduce Errors in Program Development*". In: *IBM Systems Journal* 15.3 (1976), pp. 182–211. DOI: 10.1147/sj.153.0182.

[79] Fabian Fagerholm, Michael Felderer, Davide Fucci, Michael Unterkalmsteiner, Bogdan Marculescu, Markus Martini, Lars Göran Wallgren Tengberg, Robert Feldt, Bettina Lehtelä, Balázs Nagyváradi, and Jehan Khattak. "*Cognition in Software Engineering: A Taxonomy and Survey of a Half-Century of Research*". In: *ACM Computing Surveys* 54.11s (January 2022), pp. 1–36. DOI: 10.1145/3508359.

[80] Dominic J. Farace and Joachim Schöpfel, eds. *Grey Literature in Library and Information Studies*. DE GRUYTER SAUR, September 2010. DOI: 10.1515/9783598441493.

[81] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. "*Individual, Social and Personnel Factors Influencing Modern Code Review Process*". In: *2019 IEEE Conference on Open Systems (ICOS)*. Pulau Pinang, Malaysia: IEEE, November 2019, pp. 40–45. DOI: 10.1109/icos47562.2019.8975708.

[82] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. "*Knowledge Sharing Factors for Modern Code Review to Minimize Software Engineering Waste*". In: *International Journal of Advanced Computer Science and Applications* 11.1 (2020). DOI: 10.14569/IJACSA.2020.0110160.

[83] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. "*Knowledge Sharing, a Key Sustainable Practice Is on Risk: An Insight from Modern Code Review*". In: *2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. Kuala Lumpur, Malaysia: IEEE, December 2019, pp. 1–6. DOI: 10.1109/ICETAS48360.2019.9117444.

[84] .NET Foundation. *xUnit.net*. 2022. URL: https://xunit.net (visited on 10/17/2022).

[85] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[86] Julian Frattini, Jannik Fischbach, and Andreas Bauer. "*CiRA: An Open-Source Python Package for Automated Generation of Test Case Descriptions from Natural Language Requirements*". In: *2023 IEEE 31st International Requirements Engineering Conference Workshops (REW)*. Hannover, Germany: IEEE, September 2023, pp. 68–71. DOI: 10.1109/REW57809.2023.00019.

[87] Enrico Fregnan, Josua Fröhlich, Davide Spadini, and Alberto Bacchelli. "*Graph-Based Visualization of Merge Requests for Code Review*". In: *Journal of Systems and Software* 195 (January 2023), p. 111506. DOI: 10.1016/j.jss.2022.111506.

[88] Qiang Fu, Francis Grady, Bjoern Flemming Broberg, Andrew Roberts, Geir Gil Martens, Kjetil Vatland Johansen, and Pieyre Le Loher. "*Code Review and Cooperative Pair Programming Best Practice*". In: (2017). DOI: 10.48550/ARXIV.1706.02062.

[89] Tommaso Fulcini, Giacomo Garaccione, Riccardo Coppola, Luca Ardito, and Marco Torchiano. "*Guidelines for GUI Testing Maintenance: A Linter for Test Smell Detection*". In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. Singapore Singapore: ACM, November 2022, pp. 17–24. DOI: 10.1145/3548659.3561306.

[90] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[91] Boni García, Jose M. Del Alamo, Maurizio Leotta, and Filippo Ricca. "*Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright*". In: *Quality of Information and Communications Technology*. Ed. by Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini. Vol. 2178. Cham: Springer Nature Switzerland, 2024, pp. 142–149. DOI: 10.1007/978-3-031-70245-7_10.

[92] Vahid Garousi, Wasif Afzal, Adem Çağlar, İhsan Berk Işık, Berker Baydan, Seçkin Çaylak, Ahmet Zeki Boyraz, Burak Yolaçan, and Kadir Herkiloğlu.

"*Visual GUI Testing in Practice: An Extended Industrial Case Study*". In: (2020). DOI: 10.48550/ARXIV.2005.09303.

[93] Vahid Garousi and Michael Felderer. "*Developing, Verifying, and Maintaining High-Quality Automated Test Scripts*". In: *IEEE Software* 33.3 (May 2016), pp. 68–75. DOI: 10.1109/MS.2016.30.

[94] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. "*Guidelines for Including Grey Literature and Conducting Multivocal Literature Reviews in Software Engineering*". In: *Information and Software Technology* 106 (February 2019), pp. 101–121. DOI: 10.1016/j.infsof.2018.09.006.

[95] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. "*The Need for Multivocal Literature Reviews in Software Engineering: Complementing Systematic Literature Reviews with Grey Literature*". In: *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. Limerick Ireland: ACM, June 2016, pp. 1–6. DOI: 10.1145/2915970.2916008.

[96] Vahid Garousi and Barış Küçük. "*Smells in Software Test Code: A Survey of Knowledge in Industry and Academia*". In: *Journal of Systems and Software* 138 (April 2018), pp. 52–81. DOI: 10.1016/j.jss.2017.12.013.

[97] Vahid Garousi and Mika V Mäntylä. "*Citations, research topics and active countries in software engineering: A bibliometrics study*". In: *Computer Science Review* 19 (2016), pp. 56–77.

[98] Vahid Garousi and Mika V. Mäntylä. "*A Systematic Literature Review of Literature Reviews in Software Testing*". In: *Information and Software Technology* 80 (December 2016), pp. 195–216. DOI: 10.1016/j.infsof.2016.09.002.

[99] Lorenzo Gasparini, Enrico Fregnan, Larissa Braz, Tobias Baum, and Alberto Bacchelli. "*ChangeViz: Enhancing the GitHub Pull Request Interface with Method Call Information*". In: *2021 Working Conference on Software Visualization (VISSOFT)*. Luxembourg: IEEE, September 2021, pp. 115–119. DOI: 10.1109/VISSOFT52517.2021.00022.

[100] Gerrit. *Gerrit Code Review*. 2024. URL: https://www.gerritcodereview.com (visited on 08/04/2024).

[101] GitLab B.V. *GitLab*. 2025. URL: https://about.gitlab.com/ (visited on 07/14/2025).

[102] Barney G Glaser, Anselm L Strauss, and Elizabeth Strutzel. "*The discovery of grounded theory; strategies for qualitative research*". In: *Nursing research* 17.4 (1968), p. 364.

[103] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. "*Analysis and Design of Selenium WebDriver Automation Testing Framework*". In: *Procedia Computer Science* 50 (2015), pp. 341–346. DOI: 10.1016/j.procs.2015.04.038.

[104] Lucian Gonçales, Kleinner Farias, Bruno Da Silva, and Jonathan Fessler. "*Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study*". In: *2019 IEEE/ACM 27th International Conference on Program Com-*

prehension (ICPC). Montreal, QC, Canada: IEEE, May 2019, pp. 42–52. DOI: 10.1109/ICPC.2019.00018.

[105] Lucian José Gonçales, Kleinner Farias, and Bruno C. Da Silva. "*Measuring the Cognitive Load of Software Developers: An Extended Systematic Mapping Study*". In: *Information and Software Technology* 136 (August 2021), p. 106563. DOI: 10.1016/j.infsof.2021.106563.

[106] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. "*Do Explicit Review Strategies Improve Code Review Performance?*" In: *Proceedings of the 17th International Conference on Mining Software Repositories*. Seoul Republic of Korea: ACM, June 2020, pp. 606–610. DOI: 10.1145/3379597.3387509.

[107] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. "*Do Explicit Review Strategies Improve Code Review Performance? Towards Understanding the Role of Cognitive Load*". In: *Empirical Software Engineering* 27.4 (July 2022), p. 99. DOI: 10.1007/s10664-022-10123-8.

[108] Google. *Google Engineering Practices Documentation*. 2025. URL: https://google.github.io/eng-practices/ (visited on 10/09/2025).

[109] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "*An Exploratory Study of the Pull-Based Software Development Model*". In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 345–355. DOI: 10.1145/2568225.2568260.

[110] GraphWalker. *GraphWalker, an open-source model-based testing tool*. 2022. URL: https://graphwalker.github.io (visited on 10/17/2022).

[111] Mark Grechanik, Qing Xie, and Chen Fu. "*Creating GUI Testing Tools Using Accessibility Technologies*". In: *2009 International Conference on Software Testing, Verification, and Validation Workshops*. Denver, CO, USA: IEEE, 2009, pp. 243–250. DOI: 10.1109/ICSTW.2009.31.

[112] Mark Grechanik, Qing Xie, and Chen Fu. "*Maintaining and Evolving GUI-directed Test Scripts*". In: *2009 IEEE 31st International Conference on Software Engineering*. Vancouver, BC, Canada: IEEE, 2009, pp. 408–418. DOI: 10.1109/ICSE.2009.5070540.

[113] Sacha Greif. *State of JS 2024 survey*. 2024. URL: https://2024.stateofjs.com/en-US/libraries/testing/ (visited on 04/11/2025).

[114] Egon G. Guba. "*Criteria for Assessing the Trustworthiness of Naturalistic Inquiries*". In: *ECTJ* 29.2 (June 1981), p. 75. DOI: 10.1007/BF02766777.

[115] Lisa A Guion, David C Diehl, and Debra McDonald. "*Triangulation: Establishing the Validity of Qualitative Studies*". In: *Edis* (2011).

[116] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. "*How Can Manual Testing Processes Be Optimized? Developer Survey, Optimization Guidelines, and Case Studies*". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on*

the Foundations of Software Engineering. Athens Greece: ACM, August 2021, pp. 1281–1291. DOI: 10.1145/3468264.3473922.

[117] Quinn Hanam, Ali Mesbah, and Reid Holmes. "*Aiding code change understanding with semantic change impact analysis*". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 202–212.

[118] Nikolay Harutyunyan, Andreas Bauer, and Dirk Riehle. "*Industry Requirements for FLOSS Governance Tools to Facilitate the Use of Open Source Software in Commercial Products*". In: *Journal of Systems and Software* 158 (December 2019), p. 110390. DOI: 10.1016/j.jss.2019.08.001.

[119] Ahmed Hassan. "*Mining Software Repositories to Assist Developers and Support Managers*". In: *2006 22nd IEEE International Conference on Software Maintenance*. Philadelphia, PA, USA: IEEE, September 2006, pp. 339–342. DOI: 10.1109/ICSM.2006.38.

[120] Ahmed E. Hassan. "*The Road Ahead for Mining Software Repositories*". In: *2008 Frontiers of Software Maintenance*. Beijing, China: IEEE, September 2008, pp. 48–57. DOI: 10.1109/FOSM.2008.4659248.

[121] Monique Hennink and Bonnie N. Kaiser. "*Sample Sizes for Saturation in Qualitative Research: A Systematic Review of Empirical Tests*". In: *Social Science & Medicine* 292 (January 2022), p. 114523. DOI: 10.1016/j.socscimed.2021.114523.

[122] S.E. Hove and B. Anda. "*Experiences from Conducting Semi-structured Interviews in Empirical Software Engineering Research*". In: *11th IEEE International Software Metrics Symposium (METRICS'05)*. Como, Italy: IEEE, 2005, pp. 23–23. DOI: 10.1109/METRICS.2005.24.

[123] GitHub Inc. *GitHub: Copilot code review*. 2025. URL: https://docs.github.com/en/copilot/how-tos/agents/copilot-code-review/using-copilot-code-review?tool=webui (visited on 07/10/2025).

[124] GitHub Inc. *GitHub: Where the world builds software*. 2025. URL: https://github.com (visited on 07/10/2025).

[125] Phabricator Inc. *Phabricator: Discuss. Plan. Code. Review. Test*. 2022. URL: https://www.phacility.com/phabricator (visited on 10/17/2022).

[126] J. Itkonen and K. Rautiainen. "*Exploratory Testing: A Multiple Case Study*". In: *2005 International Symposium on Empirical Software Engineering, 2005.* Queensland, Australia: IEEE, 2005, pp. 82–91. DOI: 10.1109/ISESE.2005.1541817.

[127] Juha Itkonen, Mika V. Mantyla, and Casper Lassenius. "*How Do Testers Do It? An Exploratory Study on Manual Testing Practices*". In: *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. Lake Buena Vista, FL, USA: IEEE, October 2009, pp. 494–497. DOI: 10.1109/ESEM.2009.5314240.

[128] Juha Itkonen, Mika V. Mantyla, and Casper Lassenius. "*The Role of the Tester's Knowledge in Exploratory Software Testing*". In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 707–724. DOI: 10.1109/TSE.2012.55.

[129] Harrie Jansen. "*The Logic of Qualitative Survey Research and Its Position in the Field of Social Research Methods*". In: *Forum Qualitative Sozialforschung / Forum: Qualitative Social Research* Vol 11 (March 2010), No 2 (2010): Visualising Migration and Social Division: Insights From Social Sciences and the Visual Arts. DOI: 10.17169/FQS-11.2.1450.

[130] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "*Why Don't Software Developers Use Static Analysis Tools to Find Bugs?*" In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[131] Philip N Johnson-Laird. *Human and machine thinking*. Psychology Press, 2013.

[132] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. "*An In-Depth Study of the Promises and Perils of Mining GitHub*". In: *Empirical Software Engineering* 21.5 (October 2016), pp. 2035–2071. DOI: 10.1007/s10664-015-9393-5.

[133] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. "*Empirical Observations on Software Testing Automation*". In: *2009 International Conference on Software Testing Verification and Validation*. Denver, CO, USA: IEEE, April 2009, pp. 201–209. DOI: 10.1109/ICST.2009.16.

[134] Mark Kasunic. *Designing an effective survey*. 2005.

[135] C.F. Kemerer and M.C. Paulk. "*The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data*". In: *IEEE Transactions on Software Engineering* 35.4 (July 2009), pp. 534–550. DOI: 10.1109/TSE.2009.27.

[136] Mountainminds GmbH & Co. KG. *Java Code Coverage for Eclipse*. 2022. URL: https://www.jacoco.org (visited on 10/15/2022).

[137] Barbara Kitchenham and Stuart Charters. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University, 2007.

[138] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. "*Technical Debt: From Metaphor to Theory and Practice*". In: *IEEE Software* 29.6 (November 2012), pp. 18–21. DOI: 10.1109/MS.2012.167.

[139] Filippo Lanubile and Teresa Mallardo. "*Inspecting Automated Test Code: A Preliminary Study*". In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi. Vol. 4536. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 115–122. DOI: 10.1007/978-3-540-73101-6_16.

[140] P. Layzell, O.P. Brereton, and A. French. "*Supporting Collaboration in Distributed Software Engineering Teams*". In: *Proceedings Seventh Asia-Pacific*

Software Engeering Conference. APSEC 2000. Singapore: IEEE Comput. Soc, 2000, pp. 38–45. DOI: 10.1109/APSEC.2000.896681.

[141] Amanda Lee and Jeffrey C. Carver. "*Are One-Time Contributors Different? A Comparison to Core and Periphery Developers in FLOSS Repositories*". In: *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Toronto, ON: IEEE, November 2017, pp. 1–10. DOI: 10.1109/ESEM.2017.7.

[142] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. "*Reconciling Manual and Automated Testing: The AutoTest Experience*". In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. Waikoloa, HI: IEEE, January 2007, 261a–261a. DOI: 10.1109/HICSS.2007.462.

[143] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. "*Improving Test Suites Maintainability with the Page Object Pattern: An Industrial Case Study*". In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Luxembourg, Luxembourg: IEEE, March 2013, pp. 108–113. DOI: 10.1109/ICSTW.2013.19.

[144] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. "*Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution*". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. Koblenz, Germany: IEEE, October 2013, pp. 272–281. DOI: 10.1109/WCRE.2013.6671302.

[145] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. "*Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey*". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Dublin, Ireland: IEEE, April 2023, pp. 339–350. DOI: 10.1109/ICST57152.2023.00039.

[146] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. "*Using Multi-Locators to Increase the Robustness of Web Test Cases*". In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. Graz, Austria: IEEE, April 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102611.

[147] Kevin Ljung and Javier Gonzalez-Huerta. ""*To Clean-Code or Not To Clean-Code*" *A Survey among Practitioners*". In: (2022). DOI: 10.48550/ARXIV.2208.07056.

[148] Bart Luijten and Joost Visser. "*Faster defect resolution with higher technical quality of software*". In: 2010.

[149] Kathleen M. MacQueen, Eleanor McLellan, Kelly Kay, and Bobby Milstein. "*Codebook Development for Team-Based Qualitative Analysis*". In: *CAM Journal* 10.2 (May 1998), pp. 31–36. DOI: 10.1177/1525822X980100020301.

[150] M.V. Mantyla and C. Lassenius. "*What Types of Defects Are Really Discovered in Code Reviews?*" In: *IEEE Transactions on Software Engineering* 35.3 (May 2009), pp. 430–448. DOI: 10.1109/TSE.2008.71.

[151] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[152] Robert C Martin. "*Design principles and design patterns*". In: *Object Mentor* 1.34 (2000), p. 597.

[153] Gerald Matthews, Ryan W. Wohleber, and Jinchao Lin. "*Stress, Skilled Performance, and Expertise: Overload and Beyond*". In: *The Oxford Handbook of Expertise*. Ed. by Paul Ward, Jan Maarten Schraagen, Julie Gore, and Emilie M. Roth. 1st ed. Oxford University Press, May 2019, pp. 490–524. DOI: 10.1093/oxfordhb/9780198795872.013.22.

[154] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. "*Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice*". In: *Proceedings of the ACM on Human-Computer Interaction* 3.CSCW (November 2019), pp. 1–23. DOI: 10.1145/3359174.

[155] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. "*The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects*". In: *Proceedings of the 11th working conference on mining software repositories*. 2014, pp. 192–201.

[156] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. "*An Empirical Study of the Impact of Modern Code Review Practices on Software Quality*". In: *Empirical Software Engineering* 21.5 (October 2016), pp. 2146–2189. DOI: 10.1007/s10664-015-9381-9.

[157] Atif Muhammed Memon. *A comprehensive framework for testing graphical user interfaces*. University of Pittsburgh, 2001.

[158] Daniel Méndez Fernández and Jan-Hendrik Passoth. "*Empirical Software Engineering: From Discipline to Interdiscipline*". In: *Journal of Systems and Software* 148 (February 2019), pp. 170–179. DOI: 10.1016/j.jss.2018.11.019.

[159] Tim Menzies, Justin S Di Stefano, Mike Chapman, and Ken McGill. "*Metrics that matter*". In: *27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings.* IEEE. 2002, pp. 51–57.

[160] Sharan B. Merriam and Elizabeth J. Tisdell. *Qualitative Research: A Guide to Design and Implementation*. Fourth edition. The Jossey-Bass Higher and Adult Education Series. San Francisco, CA: John Wiley & Sons, 2015.

[161] Microsoft. *Playwright enables reliable end-to-end testing for modern web apps*. 2025. URL: https://playwright.dev (visited on 07/14/2025).

[162] Randall K. Minas, Rick Kazman, and Ewan Tempero. "*Neurophysiological Impact of Software Design Processes on Software Developers*". In: *Augmented Cognition. Enhancing Cognition and Behavior in Complex Human Environments*. Ed. by Dylan D. Schmorrow and Cali M. Fidopiastis. Vol. 10285. Cham: Springer International Publishing, 2017, pp. 56–64. DOI: 10.1007/978-3-319-58625-0_4.

[163] Ivan Mistrík, John Grundy, André Hoek, and Jim Whitehead, eds. *Collaborative Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-10294-3.

[164] Glenford J. Myers, Tom Badgett, and Corey Sandler, eds. *The Art of Software Testing*. 1st ed. Wiley, January 2012. DOI: 10.1002/9781119202486.

[165] Michael D. Myers and Michael Newman. "*The Qualitative Interview in IS Research: Examining the Craft*". In: *Information and Organization* 17.1 (January 2007), pp. 2–26. DOI: 10.1016/j.infoandorg.2006.11.001.

[166] Kwan-Sik Na, Xiaotong Li, James T. Simpson, and Ki-Yoon Kim. "*Uncertainty Profile and Software Project Performance: A Cross-National Comparison*". In: *Journal of Systems and Software* 70.1-2 (February 2004), pp. 155–163. DOI: 10.1016/S0164-1212(03)00014-1.

[167] Michel Nass and Emil Alegroth. "*SCOUT: A Revised Approach to GUI-test Automation*". In: (2020), p. 2.

[168] Michel Nass, Emil Alégroth, and Robert Feldt. "*Why Many Challenges with GUI Test Automation (Will) Remain*". In: *Information and Software Technology* 138 (October 2021), p. 106625. DOI: 10.1016/j.infsof.2021.106625.

[169] Michel Nass, Emil Alégroth, Robert Feldt, and Riccardo Coppola. "*Robust Web Element Identification for Evolving Applications by Considering Visual Overlaps*". In: *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Dublin, Ireland: IEEE, April 2023, pp. 258–268. DOI: 10.1109/ICST57152.2023.00032.

[170] Michel Nass, Emil Alégroth, Robert Feldt, Maurizio Leotta, and Filippo Ricca. "*Similarity-Based Web Element Localization for Robust Test Automation*". In: *ACM Transactions on Software Engineering and Methodology* 32.3 (July 2023), pp. 1–30. DOI: 10.1145/3571855.

[171] Stanislava Nedyalkova and Jorge Bernardino. "*Open source capture and replay tools comparison*". In: *Proceedings of the International C* Conference on Computer Science and Software Engineering*. 2013, pp. 117–119.

[172] M Niranjanamurthy, Sushanth Navale, S Jagannatha, and Sudesha Chakraborty. "*Functional Software Testing for Web Applications in the Context of Industry*". In: *Journal of Computational and Theoretical Nanoscience* 15.11 (November 2018), pp. 3398–3404. DOI: 10.1166/jctn.2018.7632.

[173] Cliodhna O'Connor and Helene Joffe. "*Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines*". In: *International Journal of Qualitative Methods* 19 (January 2020), p. 160940691989922. DOI: 10.1177/1609406919899220.

[174] Michael Olan. "*Unit testing: test early, test often*". In: *J. Comput. Sci. Coll.* 19.2 (December 2003), pp. 319–328.

[175] Sebastiaan Oosterwaal, Arie Van Deursen, Roberta Coelho, Anand Ashok Sawant, and Alberto Bacchelli. "*Visualizing Code and Coverage Changes for Code Review*". In: *Proceedings of the 2016 24th ACM SIGSOFT International*

Symposium on Foundations of Software Engineering. Seattle WA USA: ACM, November 2016, pp. 1038–1041. DOI: 10.1145/2950290.2983929.

[176] Rodrigo Ordoñez-Pacheco, Karen Cortes-Verdin, and Jorge Octavio Ocharán-Hernández. "*Best Practices for Software Development: A Systematic Literature Review*". In: *New Perspectives in Software Engineering*. Ed. by Jezreel Mejia, Mirna Muñoz, Álvaro Rocha, and Yadira Quiñonez. Vol. 1297. Cham: Springer International Publishing, 2021, pp. 38–55. DOI: 10.1007/978-3-030-63329-5_3.

[177] Shi Qing Pan, Maria Vega, Alan J Vella, Brian H Archer, and GR Parlett. "*A mini-Delphi approach: An improvement on single round techniques*". In: *Progress in tourism and hospitality research* 2.1 (1996), pp. 27–39.

[178] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. "*Information Needs in Contemporary Code Review*". In: *Proceedings of the ACM on Human-Computer Interaction* 2.CSCW (November 2018), pp. 1–27. DOI: 10.1145/3274404.

[179] Oksana Petunova and Solvita Bērziša. "*Test Case Review Processes in Software Testing*". In: *Information Technology and Management Science* 20.1 (January 2017). DOI: 10.1515/itms-2017-0008.

[180] Dietmar Pfahl, Huishi Yin, Mika V. Mäntylä, and Jürgen Münch. "*How Is Exploratory Testing Used? A State-of-the-Practice Survey*". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Torino Italy: ACM, September 2014, pp. 1–10. DOI: 10.1145/2652524.2652531.

[181] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mantyla. "*Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey*". In: *2012 7th International Workshop on Automation of Software Test (AST)*. Zurich, Switzerland: IEEE, June 2012, pp. 36–42. DOI: 10.1109/IWAST.2012.6228988.

[182] Rémi Rampin and Vicky Rampin. "*Taguette: Open-Source Qualitative Data Analysis*". In: *Journal of Open Source Software* 6.68 (December 2021), p. 3522. DOI: 10.21105/joss.03522.

[183] Filippo Ricca and Andrea Stocco. "*Web Test Automation: Insights from the Grey Literature*". In: *SOFSEM 2021: Theory and Practice of Computer Science*. Ed. by Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong. Vol. 12607. Cham: Springer International Publishing, 2021, pp. 472–485. DOI: 10.1007/978-3-030-67731-2_35.

[184] Peter C. Rigby and Christian Bird. "*Convergent Contemporary Software Peer Review Practices*". In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. Saint Petersburg Russia: ACM, August 2013, pp. 202–212. DOI: 10.1145/2491411.2491444.

[185] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. "*Open Source Software Peer Review Practices: A Case Study of the Apache Server*". In: *Proceedings of the 13th International Conference on Software Engineering - ICSE '08*. Leipzig, Germany: ACM Press, 2008, p. 541. DOI: 10.1145/1368088.1368162.

[186] Robot Framework Foundation. *Robot Framework*. 2025. URL: https://robotframework.org (visited on 07/16/2025).

[187] Guoping Rong, Jingyi Li, Mingjuan Xie, and Tao Zheng. "*The Effect of Checklist in Code Review for Inexperienced Students: An Empirical Study*". In: *2012 IEEE 25th Conference on Software Engineering Education and Training*. Nanjing, China: IEEE, April 2012, pp. 120–124. DOI: 10.1109/CSEET.2012.22.

[188] Per Runeson and Martin Höst. "*Guidelines for Conducting and Reporting Case Study Research in Software Engineering*". In: *Empirical Software Engineering* 14.2 (April 2009), pp. 131–164. DOI: 10.1007/s10664-008-9102-8.

[189] Johannes Ryser and Martin Glinz. "*A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts*". In: CNAM, December 1999.

[190] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. "*Modern Code Review: A Case Study at Google*". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. Gothenburg Sweden: ACM, May 2018, pp. 181–190. DOI: 10.1145/3183519.3183525.

[191] Mary-Luz Sánchez-Gordón and Ricardo Colomo-Palacios. "*From Certifications to International Standards in Software Testing: Mapping from ISQTB to ISO/IEC/IEEE 29119-2*". In: *Systems, Software and Services Process Improvement*. Ed. by Xabier Larrucea, Izaskun Santamaria, Rory V. O'Connor, and Richard Messnarz. Vol. 896. Cham: Springer International Publishing, 2018, pp. 43–55. DOI: 10.1007/978-3-319-97925-0_4.

[192] Mary-Luz Sánchez-Gordón and Ricardo Colomo-Palacios. "*From certifications to international standards in software testing: mapping from ISQTB to ISO/IEC/IEEE 29119-2*". In: *European Conference on Software Process Improvement*. Springer. 2018, pp. 43–55.

[193] Jaakko Sauvola, Sasu Tarkoma, Mika Klemettinen, Jukka Riekki, and David Doermann. "*Future of software development with generative AI*". In: *Automated Software Engineering* 31.1 (2024). DOI: 10.1007/s10515-024-00426-z.

[194] Marko Savic, Mika Mantyla, and Maelick Claes. "*Win GUI Crawler: A Tool Prototype for Desktop GUI Image and Metadata Collection*". In: *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. Valencia, Spain: IEEE, April 2022, pp. 223–228. DOI: 10.1109/ICSTW55395.2022.00046.

[195] Willy Scheibel, Jasper Blum, Franziska Lauterbach, Daniel Atzberger, and Jürgen Döllner. "*Integrated Visual Software Analytics on the GitHub Platform*". In: *Computers* 13.2 (January 2024), p. 33. DOI: 10.3390/computers13020033.

[196] Georg-Daniel Schwarz, Andreas Bauer, Dirk Riehle, and Nikolay Harutyunyan. "*A Taxonomy of Microservice Integration Techniques*". In: *Information and Software Technology* 183 (July 2025), p. 107723. DOI: 10.1016/j.infsof.2025.107723.

[197] C.B. Seaman. "*Qualitative Methods in Empirical Studies of Software Engineering*". In: *IEEE Transactions on Software Engineering* 25.4 (July-Aug./1999), pp. 557–572. DOI: 10.1109/32.799955.

[198] Forrest Shull and Carolyn Seaman. "*Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion*". In: *IEEE Software* 25.1 (January 2008), pp. 88–90. DOI: 10.1109/MS.2008.7.

[199] Alper Silistre, Onur Kilincceker, Fevzi Belli, Moharram Challenger, and Geylani Kardas. "*Models in graphical user interface testing: Study design*". In: *2020 Turkish National Software Engineering Symposium (UYMS)*. IEEE. 2020, pp. 1–6.

[200] Mats Skoglund and Per Runeson. "*Reference-Based Search Strategies in Systematic Reviews*". In: *13th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 2009. DOI: 10.14236/ewic/EASE2009.4.

[201] SmartBear Software. *Gherkin Reference*. 2024. URL: https://cucumber.io/docs/gherkin/reference/ (visited on 07/10/2024).

[202] Anton Smirnov. *Best practices code review test automation*. 2022. URL: https://itnext.io/best-practices-code-review-test-automation-fb970feeca4c (visited on 09/24/2022).

[203] Emma Söderberg, Luke Church, Jürgen Börstler, Diederick C. Niehorster, and Christofer Rydenfält. "*What's Bothering Developers in Code Review?*" In: *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 341–342. DOI: 10.1145/3510457.3513083.

[204] Ian Sommerville. *Software Engineering*. Tenth edition. Always Learning. Boston Columbus Indianapolis New York San Francisco Hoboken Amsterdam Cape Town Dubai London: Pearson, 2016.

[205] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. "*When Testing Meets Code Review: Why and How Developers Review Tests*". In: *Proceedings of the 40th International Conference on Software Engineering*. Gothenburg Sweden: ACM, May 2018, pp. 677–687. DOI: 10.1145/3180155.3180192.

[206] Davide Spadini, Fabio Palomba, Tobias Baum, Stefan Hanenberg, Magiel Bruntink, and Alberto Bacchelli. "*Test-Driven Code Review: An Empirical Study*". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 1061–1072. DOI: 10.1109/ICSE.2019.00110.

[207] CACM Staff. "*CodeFlow: Improving the Code Review Process at Microsoft*". In: *Commun. ACM* 62.2 (January 2019), pp. 36–44. DOI: 10.1145/3287289.

[208] Miroslaw Staron. *Action Research in Software Engineering: Theory and Applications*. Cham: Springer International Publishing, 2020. DOI: 10.1007/978-3-030-32610-4.

[209] Daniel Stenberg. *The I in LLM Stands For Intelligence*. 2024. URL: https://daniel.haxx.se/blog/2024/01/02/the-i-in-llm-stands-for-intelligence/ (visited on 05/29/2025).

[210] Klaas-Jan Stol and Brian Fitzgerald. "*The ABC of Software Engineering Research*". In: *ACM Transactions on Software Engineering and Methodology* 27.3 (July 2018), pp. 1–51. DOI: 10.1145/3241743.

[211] Andrew Sutherland and Gina Venolia. "*Can Peer Code Reviews Be Exploited for Later Information Needs?*" In: *2009 31st International Conference on Software Engineering - Companion Volume*. Vancouver, BC, Canada: IEEE, 2009, pp. 259–262. DOI: 10.1109/ICSE-COMPANION.2009.5070996.

[212] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. "*How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry*". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. Cary North Carolina: ACM, November 2012, pp. 1–11. DOI: 10.1145/2393596.2393656.

[213] Mai Thai, Li Chong, and Narendra Agrawal. "*Straussian Grounded-Theory Method: An Illustration*". In: *The Qualitative Report* (January 2015). DOI: 10.46743/2160-3715/2012.1758.

[214] Mai Thi Thanh Thai, Li Choy Chong, and Narendra M Agrawal. "*Straussian grounded theory method: An illustration*". In: *The Qualitative Report* 17.5 (2012).

[215] Rosalia Tufano, Alberto Martin-Lopez, Ahmad Tayeb, Ozren Dabic, Sonia Haiduc, and Gabriele Bavota. " *Deep Learning-based Code Reviews: A Paradigm Shift or a Double-Edged Sword? "* In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2025, pp. 597–597. DOI: 10.1109/ICSE55347.2025.00060.

[216] Maneela Tuteja, Gaurav Dubey, et al. "*A research study on importance of testing and quality assurance in software development life cycle (SDLC) models*". In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.3 (2012), pp. 251–257.

[217] Mark Utting, Alexander Pretschner, and Bruno Legeard. "*A Taxonomy of Model-based Testing Approaches*". In: *Software Testing, Verification and Reliability* 22.5 (August 2012), pp. 297–312. DOI: 10.1002/stvr.456.

[218] M. Vidoni. "*A Systematic Process for Mining Software Repositories: Results from a Systematic Literature Review*". In: *Information and Software Technology* 144 (April 2022), p. 106791. DOI: 10.1016/j.infsof.2021.106791.

[219] Joost Visser. *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. Java edition, First edition. Beijing: O'Reilly, 2016.

[220] Dong Wang, Yuki Ueda, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. "*Can We Benchmark Code Review Studies? A Systematic Mapping Study of Methodology, Dataset, and Metric*". In: *Journal of Systems and Software* 180 (October 2021), p. 111009. DOI: 10.1016/j.jss.2021.111009.

[221] Hironori Washizaki, ed. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. Version 4.0. IEEE Computer Society, 2024.

[222] Miku Watanabe, Yutaro Kashiwa, Bin Lin, Toshiki Hirao, Ken'Ichi Yamaguchi, and Hajimu Iida. "*On the Use of ChatGPT for Code Review: Do Developers Like Reviews By ChatGPT?*" In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. Salerno Italy: ACM, June 2024, pp. 375–380. DOI: 10.1145/3661167.3661183.

[223] Roel Wieringa. "*Design science methodology: principles and practice*". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 2010, pp. 493–494.

[224] Laurie Williams. "*Integrating pair programming into a software development process*". In: *Proceedings 14th Conference on Software Engineering Education and Training. 'In search of a software engineering profession' (Cat. No.PR01059)*. IEEE Comput. Soc, pp. 27–36. DOI: 10.1109/CSEE.2001.913816.

[225] Claes Wohlin. "*Case Study Research in Software Engineering—It Is a Case, and It Is a Study, but Is It a Case Study?*" In: *Information and Software Technology* 133 (May 2021), p. 106514. DOI: 10.1016/j.infsof.2021.106514.

[226] Claes Wohlin. "*Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering*". In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14*. London, England, United Kingdom: ACM Press, 2014, pp. 1–10. DOI: 10.1145/2601248.2601268.

[227] Claes Wohlin and Per Runeson. "*Guiding the Selection of Research Methodology in Industry–Academia Collaboration in Software Engineering*". In: *Information and Software Technology* 140 (December 2021), p. 106678. DOI: 10.1016/j.infsof.2021.106678.

[228] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-29044-2.

[229] Ehsan Zabardast, Javier Gonzalez-Huerta, Tony Gorschek, Darja Šmite, Emil Alégroth, and Fabian Fagerholm. "*A Taxonomy of Assets for the Development of Software-Intensive Products and Services*". In: *Journal of Systems and Software* 202 (August 2023), p. 111701. DOI: 10.1016/j.jss.2023.111701.

[230] Ehsan Zabardast, Javier Gonzalez-Huerta, and Binish Tanveer. "*Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution*". In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, March 2022, pp. 30–34. DOI: 10.1109/ICSA-C54293.2022.00013.

[231] Motahareh Bahrami Zanjani, Huzefa Kagdi, and Christian Bird. "*Automatically Recommending Peer Reviewers in Modern Code Review*". In: *IEEE Transactions on Software Engineering* 42.6 (June 2016), pp. 530–543. DOI: 10.1109/TSE.2015.2500238.

Background: Code review is an essential collaborative practice in which changes are discussed before they are integrated into the codebase, enhancing code quality and promoting knowledge sharing. GUI-based testing is a technique that verifies and validates a system's behavior through interactions with its GUI, which also requires collaboration. Code review practices for tests differ from those for production code, and are not understood for GUI-based tests.

Objective: We aim to advance the understanding and practice of code review of GUI-based tests by
(a) identifying code review guidelines;
(b) investigating the specific practices, challenges, and information needs;
(c) finding empirical evidence supporting the proposed guidelines; and
(d) providing an outlook on how code review may evolve in the future.

Methods: We conduct a literature review to identify and synthesize guidelines for the review of GUI-based tests. Next, we conduct a qualitative interview with testing experts to identify practices, challenges, and information needs when reviewing GUI-based tests. To find empirical evidence for the proposed guidelines, we mine open-source software repositories. Finally, we conduct a questionnaire survey to gather practitioners' expectations about the future importance of code reviews.

Results: We synthesized 33 guidelines for GUI-based tests from literature sources, for which we found empirical evidence supporting 25 of them. Practitioners acknowledge the importance of code reviews, but lack defined practices for reviewing GUI-based tests. We identified four practices, six challenges, and four information needs for reviewing GUI-based tests. Participants in the survey anticipate an increase in code review activities, including those for GUI-based tests.

Conclusion: This thesis advances the understanding and practice of code review for GUI-based tests to improve review effectiveness and the quality of the tests under review. It also highlights its relevance in the future.

Printed by Media-Tryck, Lund 2025 NORDIC SWAN ECOLABEL 3041 0903

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2025:14
ISSN: 1653-2090
ISBN: 978-91-7295-515-8