When GUI-based Testing Meets Code Reviews

Andreas Bauer[a], Tomas Helmfridsson[a], Emil Alégroth[a], Georg-Daniel Schwarz[b]

[a]*Software Engineering Research Lab (SERL), Blekinge Institute of Technology, Sweden*
[b]*Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany*

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.

Keywords: GUI-based testing, GUI testing, code review, test review, test inspection

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, 7]. This practice is effective in ensuring the quality of products by catching errors and improving sub-optimal solutions on different types of artifacts [6]. Additionally, it enables knowledge and experience sharing and fosters active collaboration within the team [7, 30].

Testing is the most widely used method for ensuring quality in the industry [78], 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 [49, 68]. Defects found during test case reviews can even help localize problematic areas in the software development lifecycle itself [47]. Adversely, not reviewing test code lowers quality and can result in increased maintenance costs [16, 35, 48]. 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 [77]. 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 [2, 21]. This approach enables the verification of both the system's visual elements and its underlying functionality [2]. 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 [63].

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 [41], Playwright [61], and Cypress [23]. 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 [63]. 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 [37, 63, 1, 2, 4, 3]. 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 Alégroth et al. [5] 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.

2. Background

2.1. Code review

Code reviews, also known as modern code review (MCR), are characterized as informal, tool-supported, and lightweight processes [20]. This contrasts with early code reviews described by Fagan [28], 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 [70]. 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 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 [31, 58, 9]. Defects found and resolved during code reviews often enhance the software's understandability and modifiability, rather than its visible functionality [55]. The practice is therefore widely adopted in both industry and open-source projects [24, 7, 58, 11].

Whilst code review can be a completely manual process, it is primarily facilitated by dedicated tools, such as Gerrit [39], or integrated code review functionality of software platforms, such as GitHub [44] or GitLab [40]. 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 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.

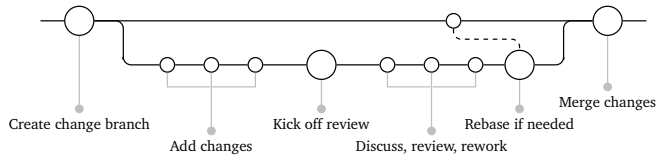

Figure 1: Development workflow of integrating changes with code reviews

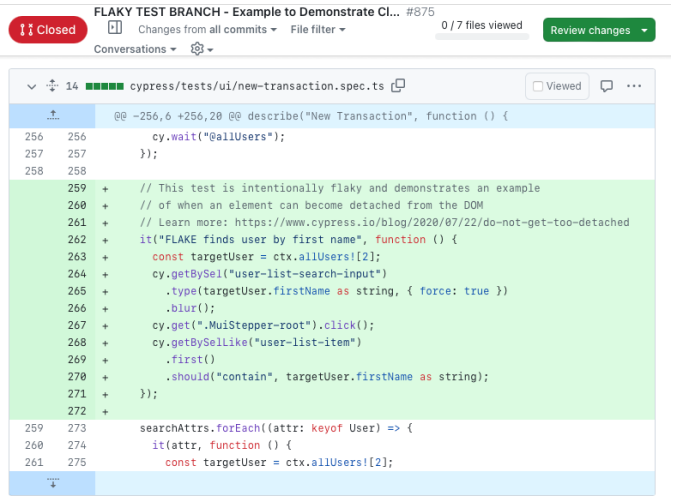

Figure 2: Diff of changed lines of code presented in a Pull Request on GitHub (PR #875)

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 [59].

Listing 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 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 [22].

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

3. Related work

The most relevant work for our study is a study conducted by Spadini et al. [77], 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 [48, 16], 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. [67], 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. [27] 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. [76], 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 [24] and systematic mapping studies by Wang et al. [80] and Badampudi et al. [8]. 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.

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 [77], 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.

4.1. Research method

To address our research questions, we conduct a qualitative interview study [60] 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 [74]. 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.

4.1.1. 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 [74]. 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 Appendix A.

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. [77] and Pascarella et al. [67]. From Spadini et al. [77], we adapted questions related to code review practices and the differences between the review of production and test code. From Pascarella et al. [67], 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 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.

4.1.2. 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 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

Table 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? [77]
6	How do you conduct code reviews?	How do you conduct reviews? [77]
6.a	Are you following any explicit or implicit practice?	Do you have specific practices? [77]
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? [77]
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? [77]
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? [67]
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? [67]

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 [41], Playwright [61], Cypress [23], and Robot Framework [71], reducing bias by focusing on challenges that correlate with only one specific tool.

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.

4.1.3. 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 [17], 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 [17].

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 [69] tool that helps ensure the traceability of emerging codes and themes to their sources.

Table 2: Overview of participants

ID	Company	Role	Industrial experience in years		
			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

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" [17]. 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 [12] 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 subthemes. 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 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 [12].

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 [65, 57]. 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 [12] includes all iterations of the code system and a comparison of the coding between the first author and the reviewers.

Table 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**

5. Results

This section presents the results addressing our research questions.

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 [77]. 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 [79, 53]. 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 [5]. 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.

Table 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

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 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) [15]. 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. [77] 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 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.

5.1.1. 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 [77], 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 4.

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.

Table 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

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 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 [75]. 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 [19] 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. [64]. 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 [56, 52] and guidelines for reviewing production code [13]. 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 [50] facilitate interactions with web pages through model representations in test cases. Testing frameworks like Playwright [61] 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 [71] 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 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 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 [77].

One of the benefits of code reviews is the educational aspect of sharing knowledge with others [72]. 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 [51].

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 components 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 [67].

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.

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

Table 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 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, 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.

Table 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

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.

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. [38] 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 [10], graph-based representation with code navigation capabilities [33], or a textual summary of code coverage impacts [66].

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) [15] 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.

6. Analysis

In this section, we present an analysis of our findings in comparison to the findings of Spadini et al. [77] and Pascarella et al. [67]. Table 9 summarizes the mapping of our findings to those from related work.

Importance of reviewing files for GUI-based testing. Spadini et al. [77] 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. [77] 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. [77] 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 [76].

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. [77]. Well-crafted commit messages help developers understand the changes and the reasoning behind them [25].

Challenges. Similar to the findings of Spadini et al. [77], "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 [32]) 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 [26, 36, 14] In the context of GUI-based testing, Fulcini et al. [34] presented 22 guidelines for GUI

Table 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
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
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
F	More code review	More frequent and collaborative code reviews of GUI-based tests to enhance knowledge sharing and review quality	2

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. [77] 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 [77]. The issue

Table 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 [77]
P	Side-by-side views	Finding 3. Reviewing production and test code [77]
P	Commit-by-commit analysis	Reading commit message or any documentation attached to the review request [77]
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 [...] [77]
C	Understandability and readability	Finding 4. A main concern of reviewers [...] and ensure tests' maintainability and readability[77]
C	Levels of abstraction	–no mapping–
C	Lack of testing experience	Finding 8. Novice developers and managers lack testing experience [77]
C	Test robustness	–no mapping–
C	Test maintenance	Finding 4. A main concern of reviewers [...] and ensure tests' maintainability and readability[77]
I	Context information	N3. Rational [67]; N4 Code context [67]; Finding 6. Reviewing test files requires developers to have context about not only the test, but also the production file under test [77]
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 [77]
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 [77]
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 [77]
F	Navigation to related files	Finding 9. Review tools should provide better navigation between test and production files [77]
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. [77]
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. [77]
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 [77]

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 [18] 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. [77]. 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 [51].

Information needs. Spadini et al. [77] describes a general lack of test-specific information within code review tools, while Pascarella et al. [67] 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' familiar-

ity 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. [77]. 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 [61], 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. [81] 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 [45] 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 [46]. 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. [73] 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 [13]. 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 Alégroth et al. [5] 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 [13], 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.

7. Limitations

We utilize the trustworthiness criteria proposed by Guba [42] to discuss the limitations of this study, which arise from the qualitative nature of the empirically collected data.

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 [77, 67], 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 [62]. 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 [42].

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 [12], which includes the codebook (all codes and their descriptions), anonymized interviews, documentation of inter-coder 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 [17].

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 et al. [29], 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 [43]—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 [43] 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.

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 [65]. 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.

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 [54].

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, 7, 30]. 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.

CRediT authorship contribution statement

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.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the authors used Grammarly and ChatGPT in order to improve and rephrase written paragraphs. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Funding

This work was funded by the KKS foundation through the SERT Research Profile project (research profile grant 2018/010) at Blekinge Institute of Technology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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 [12].

References

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

[2] Emil Alégroth and Robert Feldt. On the long-term use of visual gui testing in industrial practice: A case study. *Empirical Software Engineering*, 22(6):2937–2971, December 2017. ISSN 1382-3256, 1573-7616. doi:10.1007/s10664-016-9497-6.

[3] Emil Alégroth, Robert Feldt, and Pirjo Kolström. Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, 73:66–80, May 2016. ISSN 09505849. doi:10.1016/j.infsof.2016.01.012.

[4] Emil Alégroth, 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)*, pages 172–181, Vasteras, April 2018. IEEE. ISBN 978-1-5386-5012-7. doi:10.1109/ICST.2018.00026.

[5] Emil Alégroth, 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)*, pages 340–350, Porto de Galinhas, Brazil, April 2021. IEEE. ISBN 978-1-72816-836-4. doi:10.1109/ICST49551.2021.00046.

[6] Aybuke Aurum, Hkan Petersson, and Claes Wohlin. State-of-the-art: Software inspections after 25 years. *Software Testing, Verification and Reliability*, 12(3):133–154, September 2002. ISSN 0960-0833, 1099-1689. doi:10.1002/stvr.243.

[7] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 712–721, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-3076-3 978-1-4673-3073-2. doi:10.1109/ICSE.2013.6606617.

[8] Deepika Badampudi, Michael Unterkalmsteiner, and Ricardo Britto. Modern Code Reviews—Survey of Literature and Practice. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–61, October 2023. ISSN 1049-331X, 1557-7392. doi:10.1145/3585004.

[9] Richard A. Baker. Code reviews enhance software quality. In *Proceedings of the 19th International Conference on Software Engineering - ICSE '97*, pages 570–571, Boston, Massachusetts, United States, 1997. ACM Press. ISBN 978-0-89791-914-2. doi:10.1145/253228.253461.

[10] 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)*, pages 110–114, Luxembourg, September 2021. IEEE. ISBN 978-1-66543-144-6. doi:10.1109/VISSOFT52517.2021.00021.

[11] 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*, pages 134–144, Florence, Italy, May 2015. IEEE. ISBN 978-1-4799-1934-5. doi:10.1109/ICSE.2015.35.

[12] Andreas Bauer. Artifact package: When gui-based testing meets code reviews, September 2025. URL `https://doi.org/10.5281/zenodo.17158220`.

[13] Andreas Bauer, Riccardo Coppola, Emil Alégroth, and Tony Gorschek. Code review guidelines for GUI-based testing artifacts. *Information and Software Technology*, 163:107299, November 2023. ISSN 09505849. doi:10.1016/j.infsof.2023.107299.

[14] 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)*, pages 56–65, Trento, Italy, September 2012. IEEE. ISBN 978-1-4673-2313-0 978-1-4673-2312-3. doi:10.1109/ICSM.2012.6405253.

[15] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[16] 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*, page 571, St. Louis, MO, USA, 2005. ACM Press. doi:10.1145/1062455.1062556.

[17] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative Research in Psychology*, 3 (2):77–101, January 2006. ISSN 1478-0887, 1478-0895. doi:10.1191/1478088706qp063oa.

[18] 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*, pages 289–294, Kyoto, Japan, August 2012. IEEE. ISBN 978-1-4673-2120-4. doi:10.1109/SNPD.2012.10.

[19] James Clark and Steve DeRose. Xml path language (xpath), 1999. URL `https://www.w3.org/TR/1999/REC-xpath-19991116/`.

[20] Jason Cohen. Modern code review. In *Making Software: What Really Works, and Why We Believe It*, pages 329–336. O'Reilly, 1st ed edition, 2010. ISBN 978-0-596-80832-7.

[21] Riccardo Coppola and Emil Alégroth. A taxonomy of metrics for GUI-based testing research: A systematic literature review. *Information and Software Technology*, 152:107062, December 2022. ISSN 09505849. doi:10.1016/j.infsof.2022.107062.

[22] Riccardo Coppola, Maurizio Morisio, Marco Torchiano, and Luca Ardito. Scripted GUI testing of Android open-source apps: Evolution of test code and fragility causes. *Empirical Software Engineering*, 24(5):3205–3248, October 2019. ISSN 1382-3256, 1573-7616. doi:10.1007/s10664-019-09722-9.

[23] Cypress.io, Inc. Test. automate. accelerate., 2025. URL `https://www.cypress.io`.

[24] Nicole Davila and Ingrid Nunes. A systematic literature review and taxonomy of modern code review. *Journal of Systems and Software*, 177:110951, July 2021. ISSN 01641212. doi:10.1016/j.jss.2021.110951.

[25] 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*, pages 36–39, Vancouver, BC, Canada, 2009. IEEE. ISBN 978-1-4244-3712-2. doi:10.1109/CHASE.2009.5071408.

[26] Arie Deursen, Leon MF Moonen, A Bergh, and Gerard Kok. Refactoring test code, 2001.

[27] 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)*, pages 482–491, Taipei, Taiwan, December 2021. IEEE. ISBN 978-1-66543-784-4. doi:10.1109/APSEC53868.2021.00055.

[28] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. ISSN 0018-8670. doi:10.1147/sj.153.0182.

[29] 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)*, pages 40–45, Pulau Pinang, Malaysia, November 2019. IEEE.

doi:10.1109/icos47562.2019.8975708.

[30] 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)*, pages 1–6, Kuala Lumpur, Malaysia, December 2019. IEEE. ISBN 978-1-72814-082-7. doi:10.1109/ICETAS48360.2019.9117444.

[31] Nargis Fatima, Sumaira Nazir, and Suriayati Chuprat. Knowledge Sharing Factors for Modern Code Review to Minimize Software Engineering Waste. *International Journal of Advanced Computer Science and Applications*, 11(1), 2020. ISSN 21565570, 2158107X. doi:10.14569/IJACSA.2020.0110160.

[32] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[33] Enrico Fregnan, Josua Fröhlich, Davide Spadini, and Alberto Bacchelli. Graph-based visualization of merge requests for code review. *Journal of Systems and Software*, 195:111506, January 2023. ISSN 01641212. doi:10.1016/j.jss.2022.111506.

[34] 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*, pages 17–24, Singapore Singapore, November 2022. ACM. ISBN 978-1-4503-9452-9. doi:10.1145/3548659.3561306.

[35] Vahid Garousi and Michael Felderer. Developing, Verifying, and Maintaining High-Quality Automated Test Scripts. *IEEE Software*, 33(3):68–75, May 2016. ISSN 0740-7459, 1937-4194. doi:10.1109/MS.2016.30.

[36] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81, April 2018. ISSN 01641212. doi:10.1016/j.jss.2017.12.013.

[37] 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. 2020. doi:10.48550/ARXIV.2005.09303.

[38] 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)*, pages 115–119, Luxembourg, September 2021. IEEE. ISBN 978-1-66543-144-6. doi:10.1109/VISSOFT52517.2021.00022.

[39] Gerrit. Gerrit code review, 2024. URL `https://www.gerritcodereview.com`.

[40] GitLab B.V. Gitlab, 2025. URL `https://about.gitlab.com/`.

[41] Satish Gojare, Rahul Joshi, and Dhanashree Gaigaware. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*, 50:341–346, 2015. ISSN 18770509. doi:10.1016/j.procs.2015.04.038.

[42] Egon G. Guba. Criteria for assessing the trustworthiness of naturalistic inquiries. *ECTJ*, 29(2):75, 1981. ISSN 0148-5806. doi:10.1007/BF02766777.

[43] Monique Hennink and Bonnie N. Kaiser. Sample sizes for saturation in qualitative research: A systematic review of empirical tests. *Social Science & Medicine*, 292:114523, January 2022. ISSN 02779536. doi:10.1016/j.socscimed.2021.114523.

[44] GitHub Inc. Github: Where the world builds software, 2025. URL `https://github.com`.

[45] 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`.

[46] 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)*, pages 672–681, San Francisco, CA, USA, May 2013. IEEE. ISBN 978-1-4673-3076-3 978-1-4673-3073-2. doi:10.1109/ICSE.2013.6606613.

[47] C.F. Kemerer and M.C. Paulk. The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data. *IEEE Transactions on Software Engineering*, 35(4):534–550, July 2009. ISSN 0098-5589. doi:10.1109/TSE.2009.27.

[48] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, November 2012. ISSN 0740-7459. doi:10.1109/MS.2012.167.

[49] Filippo Lanubile and Teresa Mallardo. Inspecting Automated Test Code: A Preliminary Study. In Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors, *Agile Processes in Software Engineering and Extreme Programming*, volume 4536, pages 115–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73100-9. doi:10.1007/978-3-540-73101-6_16.

[50] 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*, pages 108–113, Luxembourg, Luxembourg, March 2013. IEEE. ISBN 978-0-7695-4993-4 978-1-4799-1324-4. doi:10.1109/ICSTW.2013.19.

[51] 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)*, pages 272–281, Koblenz, Germany, October 2013. IEEE. ISBN 978-1-4799-2931-3. doi:10.1109/WCRE.2013.6671302.

[52] Kevin Ljung and Javier Gonzalez-Huerta. "To Clean-Code or Not To Clean-Code" A Survey among Practitioners. *arXiv*, 2022. doi:10.48550/ARXIV.2208.07056.

[53] Bart Luijten and Joost Visser. Faster defect resolution with higher technical quality of software. 2010.

[54] Kathleen M. MacQueen, Eleanor McLellan, Kelly Kay, and Bobby Milstein. Codebook Development for Team-Based Qualitative Analysis. *CAM Journal*, 10(2):31–36, May 1998. ISSN 1087-822X. doi:10.1177/1525822X980100020301.

[55] M.V. Mantyla and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, May 2009. ISSN 0098-5589. doi:10.1109/TSE.2008.71.

[56] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[57] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–23, November 2019. ISSN 2573-0142. doi:10.1145/3359174.

[58] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21:2146–2189, 2016.

[59] Atif Muhammed Memon. *A comprehensive framework for testing graphical user interfaces*. University of Pittsburgh, 2001.

[60] Sharan B. Merriam and Elizabeth J. Tisdell. *Qualitative Research: A Guide to Design and Implementation*. The Jossey-Bass Higher and Adult Education Series. John Wiley & Sons, San Francisco, CA, fourth edition edition, 2015. ISBN 978-1-119-00365-6 978-1-119-00360-1.

[61] Microsoft. Playwright enables reliable end-to-end testing for modern web apps, 2025. URL `https://playwright.dev`.

[62] Michael D. Myers and Michael Newman. The qualitative interview in IS research: Examining the craft. *Information and Organization*, 17(1):2–26, January 2007. ISSN 14717727. doi:10.1016/j.infoandorg.2006.11.001.

[63] Michel Nass, Emil Alégroth, and Robert Feldt. Why many challenges with GUI test automation (will) remain. *Information and Software Technology*, 138:106625, October 2021. ISSN 09505849. doi:10.1016/j.infsof.2021.106625.

[64] 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)*, pages 258–268, Dublin, Ireland, April 2023. IEEE. ISBN 978-1-66545-666-1.

doi:10.1109/ICST57152.2023.00032.

[65] Cliodhna O'Connor and Helene Joffe. Intercoder Reliability in Qualitative Research: Debates and Practical Guidelines. *International Journal of Qualitative Methods*, 19: 160940691989922, January 2020. ISSN 1609-4069, 1609-4069. doi:10.1177/1609406919899220.

[66] 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*, pages 1038–1041, Seattle WA USA, November 2016. ACM. ISBN 978-1-4503-4218-6. doi:10.1145/2950290.2983929.

[67] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, and Alberto Bacchelli. Information Needs in Contemporary Code Review. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–27, November 2018. ISSN 2573-0142. doi:10.1145/3274404.

[68] Oksana Petunova and Solvita Bērziša. Test Case Review Processes in Software Testing. *Information Technology and Management Science*, 20(1), January 2017. ISSN 2255-9094. doi:10.1515/itms-2017-0008.

[69] Rémi Rampin and Vicky Rampin. Taguette: Open-source qualitative data analysis. *Journal of Open Source Software*, 6(68):3522, December 2021. ISSN 2475-9066. doi:10.21105/joss.03522.

[70] 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*, page 541, Leipzig, Germany, 2008. ACM Press. ISBN 978-1-60558-079-1. doi:10.1145/1368088.1368162.

[71] Robot Framework Foundation. Robot framework, 2025. URL `https://robotframework.org`.

[72] 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*, pages 181–190, Gothenburg Sweden, May 2018. ACM. ISBN 978-1-4503-5659-6. doi:10.1145/3183519.3183525.

[73] Willy Scheibel, Jasper Blum, Franziska Lauterbach, Daniel Atzberger, and Jürgen Döllner. Integrated Visual Software Analytics on the GitHub Platform. *Computers*, 13(2):33, January 2024. ISSN 2073-431X. doi:10.3390/computers13020033.

[74] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, July-Aug./1999. ISSN 00985589. doi:10.1109/32.799955.

[75] SmartBear Software. Gherkin reference, 2024. URL `https://cucumber.io/docs/gherkin/reference/`.

[76] 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*, pages 341–342, Pittsburgh Pennsylvania, May 2022. ACM. ISBN 978-1-4503-9226-6. doi:10.1145/3510457.3513083.

[77] 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*, pages 677–687, Gothenburg Sweden, May 2018. ACM. ISBN 978-1-4503-5638-1. doi:10.1145/3180155.3180192.

[78] Maneela Tuteja, Gaurav Dubey, et al. A research study on importance of testing and quality assurance in software development life cycle (sdlc) models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3):251–257, 2012.

[79] Joost Visser. *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. O'Reilly, Beijing, java edition, first edition edition, 2016. ISBN 978-1-4919-5352-5.

[80] 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. *Journal of Systems and Software*, 180:111009, October 2021. ISSN 01641212. doi:10.1016/j.jss.2021.111009.

[81] 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*, pages 375–380, Salerno Italy, June 2024. ACM. ISBN 9798400717017. doi:10.1145/3661167.3661183.

Appendix A. 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?