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

Andreas Bauer[a], Florian Angermeir[a,b], Emil Alégroth[a], Seth Anglert

[a]*Software Engineering Research Lab (SERL), Blekinge Institute of Technology, Sweden*
[b]*fortiss, Germany*

Abstract

Context: Code review has become a core practice in collaborative software engineering, helping ensure code quality, detecting potential bugs, and supporting communication among developers. Prior research has shown that code review practices differ between production and test code, suggesting that established code review guidelines may fall short in the context of test and GUI-based test code. Particularly, GUI-based testing lacks adequate support during the code review process. To address this, we proposed a set of code review guidelines specifically designed for reviewing GUI-based test files, which, however, have not yet been empirically evaluated, limiting their practical relevance.
Objective: This study empirically assesses the extent to which code review comments on GUI-based tests align (explicitly or implicitly) with the concerns captured by the proposed guidelines, and uses the findings to refine the guideline set.
Method: To achieve this, we sampled code review comments discussing GUI-based test files across 100 open-source projects and manually analyzed 1000 pull requests to determine to what extent the reviewers' comments align with the proposed guidelines.
Results: Review comments aligned with the proposed guidelines in 808 of 1000 pull requests. We found empirical evidence for 25 of the 33 guidelines. The most frequently observed guideline concerns the correct use of testing techniques and exception handling, particularly regarding locators, explicit waits, and timeout behavior.
Conclusion: The observed alignment suggests that the proposed guidelines capture concerns articulated in practice, indicating practical relevance for GUI-based test reviews. This represents an initial step towards providing empirical validation of the proposed guidelines, highlighting their potential value in enhancing the quality of GUI-based test reviews.

Keywords: GUI-based testing, GUI testing, Test scripts, Code Review, Sample Study, Mining Software Repositories, GitHub

1. Introduction

Modern software systems tend to be large and complex, evolving rapidly to the point where a single developer cannot oversee all aspects of the software or fully understand the implications of every change. Thus, the development of modern software systems requires extensive collaboration among professionals with diverse technical skills and domain knowledge to manage the increasing complexity of developed systems [1, 2, 3]. To facilitate collaborative efforts, many industry and open-source projects have embraced code reviews, as informal and asynchronous discussions about changes and their potential impacts before they are merged into the codebase [4, 5, 6, 7, 8]. In addition to improving the quality of artifacts under review, code reviews facilitate knowledge sharing among developers and testers.

Similarly, software testing is a collaborative effort that relies on contributions from individuals with both technical and non-technical expertise [9]. Among the various testing approaches, graphical user interface (GUI)-based testing stands out as a technique for verification and validation of a system's behavior through its visual interface, simulating real-world user interactions [10, 11].

In our earlier exploratory work, we proposed guidelines for reviewing GUI-based test files to assist contributors and reviewers during the code review process Bauer et al. [12]. However, these proposed guidelines have not been empirically evaluated yet.

This previous exploratory research, viewed through the lens of an empirical research cycle (see Figure 1), is used to generate a theory that positions guidelines as the formalized version of that theory. In this theory, we propose that guidelines for code review in GUI-based test files can offer benefits similar to those of general guidelines for production code. We believe that tailored guidelines specific to the context of GUI-based testing can promote consistency, reduce ambiguity, and enhance both the effectiveness of the review process and the quality of the reviewed artifacts. For testers, they clarify expectations and explicate the information needs of reviewers. For reviewers, they serve as a checklist to systematically evaluate test artifacts and identify potential quality concerns.

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

We claim the following contribution of this study:

1. An empirical assessment of the alignment between proposed guidelines for reviewing GUI-based test files and code review comments from real-world projects.
2. Concrete examples that illustrate each guideline as observed in review comments, and resulting refinements to the guidelines.
3. A comparative analysis of our findings alongside those reported in other studies.
4. A replication package containing all scripts used for data collection and a set of intermediate artifacts, allowing the replication and extension of our study by other researchers.

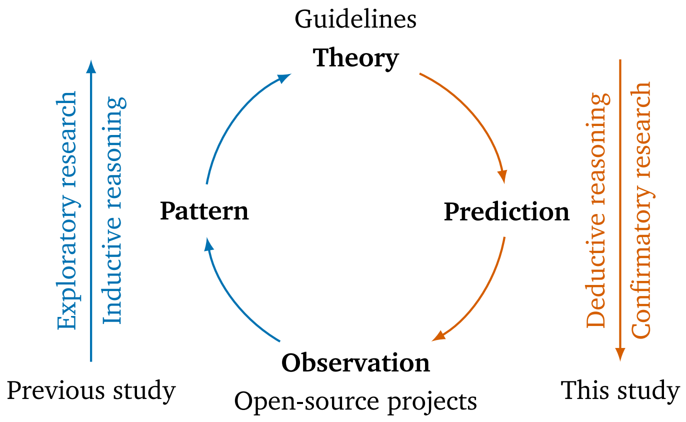

Figure 1: The empirical research cycle involves two main approaches: exploratory research, which generates theories using inductive reasoning (starting with observations), and confirmatory research, which tests theories through deductive reasoning (starting with a theory). (adopted from [13])

2. Background and Related Work

2.1. Code Review

Code review, as it is practiced nowadays, is characterized as informal, tool-supported, and lightweight processes [14]. In contrast, earlier methods of code review, often referred to as code inspections, followed a more formal, waterfall-like process [15]. This approach was associated with overhead costs, which hindered its adoption.

Code reviews have been shown to be effective at raising software quality and promoting knowledge sharing [16, 6, 17]. Defects found and resolved during code reviews often enhance the software's understandability and modifiability, rather than its visible functionality [18]. The practice is therefore widely adopted in both industry and open-source projects [4, 5, 6, 7].

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

2.2. GUI-based testing

GUI-based testing is a form of testing, performed at a higher level of abstraction, where a system's behavior is verified through its graphical user interface (GUI), mirroring user interactions [11, 10]. By interacting with the GUI, GUI-based testing enables the assessment of both the visual appearance and functional correctness of the system under test (SUT) [10]. In industry, such testing is performed manually as well as through automated tests [19, 20, 21, 22, 23, 24].

GUI-based testing can be categorized in several ways [25]. One common approach categorizes tests based on how the test cases are defined, generally falling into three categories: scripted tests, capture and replay, and model-based tests. Scripted GUI-based testing involves the development of test scripts or code using specialized automation APIs, tools, and frameworks, such as Selenium, Playwright, and Cypress. Capture and Replay testing relies on tools that record the actions performed by a tester or user on the GUI of the system. These tools create re-executable test sequences from the recorded operations. Model-based testing involves the generation of models that represent the SUT's intended behavior from a GUI perspective. These models are used to generate test sequences that ensure coverage of various states of the GUI. The models created can be visual (defined with nodes and vertices), textual (as used in behavior-driven development), or formal (for instance, in mathematical modeling).

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

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

the fact that code-based test scripts closely resemble production code, which is the typical subject of code review practices.

Additionally, GUI-based testing can be applied to the GUI of various types of applications, including web applications, native desktop applications [26], and mobile applications [27, 28]. Web applications are a common focus for GUI-based tests, and there is a wide range of testing tools available for this purpose. Therefore, this study will concentrate exclusively on web applications and exclude other types. However, our findings could be relevant, to some extent, to both mobile and desktop applications that are built on top of web technologies.

A GUI test case is composed of two components: (a) the initial GUI state in which the test case is executed, and (b) a series of events that act as the test input for the GUI [29].

2.3. Related Work

In our previous study, we conducted a multivocal literature review of both academic and gray literature to identify code review guidelines for GUI-based testing files [12]. Software engineering guidelines outline best practices for producing high-quality software. However, most existing code review practices and guidelines aim at enhancing the effectiveness and efficiency of code reviews, and focus primarily on source code and lower-level test code [30, 12].

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

The most relevant work related to this topic is by Spadini et al. [31]. In this study, the authors investigated whether and how developers review test files during code reviews. Through interviews with 12 experienced practitioners, the study revealed that reviewing test files differs from reviewing production code. Specifically, when reviewing test files, reviewers tend to focus on test-specific concerns, such as adherence to testing best practices, coverage of expected and edge-case behaviors, identification of tested and untested paths, and the clarity and correctness of assertions. While the study highlights the difference between production and test code review, the study does not distinguish among different types of tests, i.e., unit, integration, or system tests. Therefore, it remains unclear whether review practices differ even more based on the type of tests. For example, unit tests typically emphasize correctness and isolation of individual components, while system tests focus on the end-to-end behavior of a system and integration across modules. The cognitive load required to review a system's behavior may be greater than that of isolated components, potentially requiring tailored review guidelines or tooling support. Given that cognition plays an important role in software engineering activities [32, 33, 34], research is investigating its impact on, e.g., interpreting different levels of abstraction [35].

In a study conducted by Gonçalves et al. [36] (based on the registered report [37]), the authors investigate the impact of a guidance approach via a checklist on the effectiveness and efficiency of code reviews through experimentation. The research suggests that providing a guidance approach for code reviews is expected to reduce the cognitive load associated with performing these reviews. The experimental setup included three types of conditions: ad hoc review, a checklist, and a guided checklist.

For effectiveness, the researchers measured the number of functional defects identified, while for efficiency, they assessed the number of defects found in relation to the review time taken. The checklist comprised 17 binary checks (indicating either that everything is okay or that defects have been found) and is organized into topics such as general, classes, methods, arguments, variables, if-then statements, loops, recursion, errors, and a final check. The guided checklist presented the same items as the regular checklist, but did not show all items simultaneously. Instead, it displayed only those relevant to the selected code segment.

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

In contrast to our guidelines, their checklist was designed for general production code reviews. The review of test code and GUI-based test code occurs less frequently [31, 38], which limits opportunities to gain expertise through reviews. Thus, we assume that the potential benefits of guidelines supporting code reviews may be greater.

2.4. Previous evaluation attempts

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

A brief summary of the pilot study is as follows:

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

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

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

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

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

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

3. Methodology

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

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

According to the ABC framework proposed by Stol and Fitzgerald [39], this study is classified as a sample study, as we aim to draw conclusions applicable to a larger population of open-source software projects. Sample studies are commonly employed to analyze extensive collections of software development artifacts or projects, especially within the realm of open-source software.

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

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

3.1. Mining Software Repositories (MSR)

To collect relevant evidence for the proposed GUI-based testing guidelines, we employ software repository mining as a research method [40], analyzing code review comments from open-source GitHub projects.

In this section, we outline all the steps of the data-gathering process. Figure 2 illustrates a visual summary of this process. For transparency and replicability, all scripts used to perform the data gathering are available in the study's replication package [41].

Inclusion of testing tools. Before we begin identifying open-source repositories, we clarify our choice of Cypress and Playwright as testing tools for this search. In academic studies, Selenium [42] is frequently referenced as the reference browser automation tool for facilitating automated

[4]The analysis was conducted to determine if findings from interviews could be complemented by findings from open-source repositories in our previous study [38].

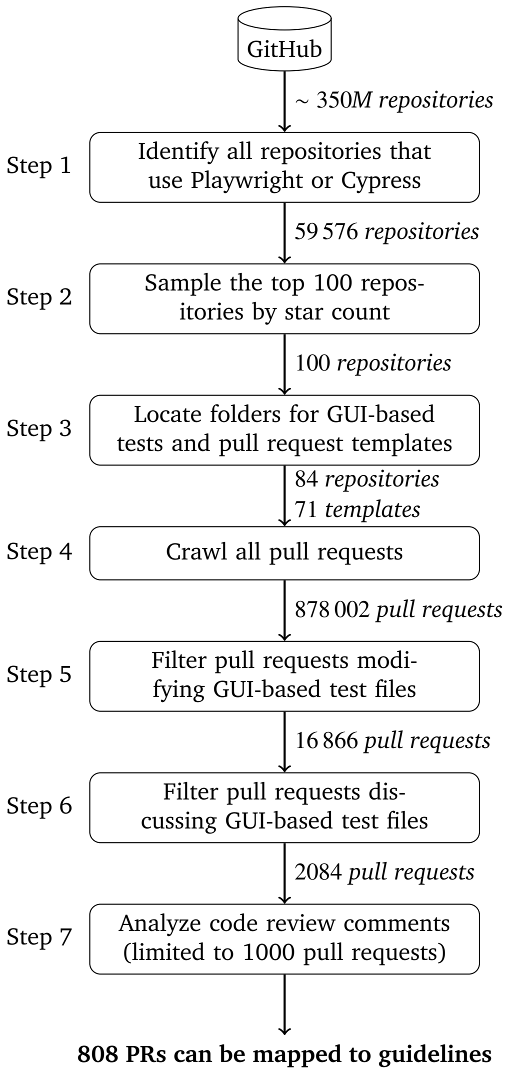
GitHub
~ 350M repositories
Step 1
Identify all repositories that use Playwright or Cypress
59 576 repositories
Step 2
Sample the top 100 repositories by star count
100 repositories
Step 3
Locate folders for GUI-based tests and pull request templates
84 repositories
71 templates
Step 4
Crawl all pull requests
878 002 pull requests
Step 5
Filter pull requests modifying GUI-based test files
16 866 pull requests
Step 6
Filter pull requests discussing GUI-based test files
2084 pull requests
Step 7
Analyze code review comments (limited to 1000 pull requests)
808 PRs can be mapped to guidelines

Figure 2: Summary GitHub review analysis

tests such as GUI-based testing [43, 44]. However, more recent tools have surpassed Selenium in usage and popularity, such as Cypress, Playwright, or Puppeteer [45]. According to the State of JS 2024 survey [46], which gathered responses from 14 015 participants, Cypress ranked third in usage among the JavaScript community, while Selenium has seen a continuous decline, now ranking ninth. Selenium is a general-purpose browser automation library used for testing web-based projects, rather than a pure testing framework, which results in multiple challenges when used for end-to-end testing scenarios, such as flakiness [47]. During the pilot study, our participating company reports that clients hiring test automation consulting staff are increasingly choosing Cypress to fulfill their testing needs, reflecting the growing popularity of this tool in industrial practice. As a result, we decided to focus on repositories that utilize either Cypress or Playwright as their testing tools.

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

Table 1: Search query to find repositories that include configuration files for Playwright or Cypress. This query locates files including the specified `filename` as part of their name and with the extensions `.js` or `.ts`. Example of files that will be found: `playwright.config.ts`, `playwright.config.js`, and `example.cypress.config.ts`.

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

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

Step 3: Locate GUI-based test directories and pull request templates. The first author manually identified directories containing GUI-based test files. These directory paths serve as filters for relevant pull requests in subsequent analysis. To make it easier to look up test files later, we also noted the test file extension (e.g., `*.test.ts`,

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

`*.spec.ts`) for each repository to account for cases where test and production code are organized in the same directory. Repositories that use Playwright or Cypress solely for non-GUI tests were excluded from further analysis. This includes scenarios where tests are executed against a web API or where the integration of single components in an isolated context is assessed. Out of the 100 repositories, 84 remain for further analysis.

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

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

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

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

On GitHub, there are two ways to provide comments: threads and general comments. Thread comments consist of one comment or a series of comments related to a specific file being modified, making it easy to identify what is under discussion. In contrast, general comments do not reference any specific file and typically address the entire pull request.

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

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

Step 7: Aligning code review comments with guidelines. We mapped comments on GUI-based testing files to the 33 guidelines that were organized in nine categories from our prior study [12]. For the mapping, we use the existing explanation for each guideline about its purpose, a description, and examples of its application to assess whether a comment aligns with a specific guideline.

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

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

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

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

We limited the analysis to 1000 pull requests due to the substantial manual effort required. In this final sample, we ensured that all repositories containing comments about GUI-based test files were represented. The mapping of each pull request to the guidelines is made transparent in the replication package. It includes a spreadsheet that lists all pull requests related to GUI-based test files, along with the guideline mappings, additional notes, and URLs to access the pull requests on GitHub.

4. Results

4.0.1. Description of Collected Data

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

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

Pull requests related to GUI-based test files. In total, we gathered data from 878 002 PRs. As shown in Table 2, modifications to GUI-based test files are rare, and code review discussions of these files are rarer still. Only 1.9% (16 866 PRs) of all PRs modify GUI-based test files, and just 0.24% (2084 PRs) contained comments or discussions.

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

4.1. Findings from pull request templates

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

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

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

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

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

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

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

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

Checklist items. Contributors shall check or uncheck items that are presented in Table 3 if applicable.

4.2. Mapping Code Review Comments to GUI-based Testing Guidelines

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

In the following analysis, we detail the level of support observed for each guideline, organized by category. Table 4 summarizes the found evidence to support our guidelines based on the gathered data. As evidence, we consider the analyzed code review comments at the PR granularity level, as well as insights from the PR templates.

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

Table 2: Overview of top 100 repositories and the number of total pull requests, pull requests changing GUI-based test files (w/ tests), and pull requests containing code review comments addressing GUI-based test files (w/ review)

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

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

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

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

G2: Use checklists. Our use checklist guideline refers to aspects like GUI element localization, test oracles, or test synchronization. While there were no code review comments related to the checklist, 43 of the 71 pull request templates included predefined checks, as detailed in Section 4.1. The items covered by these checklists were generally applicable, but only one was specifically related to

Table 3: Checklist items in pull request templates and the number of templates (#) they appear in

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

testing: it involved verifying whether a new or modified test was successfully executed in a local environment.

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

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

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

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

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

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

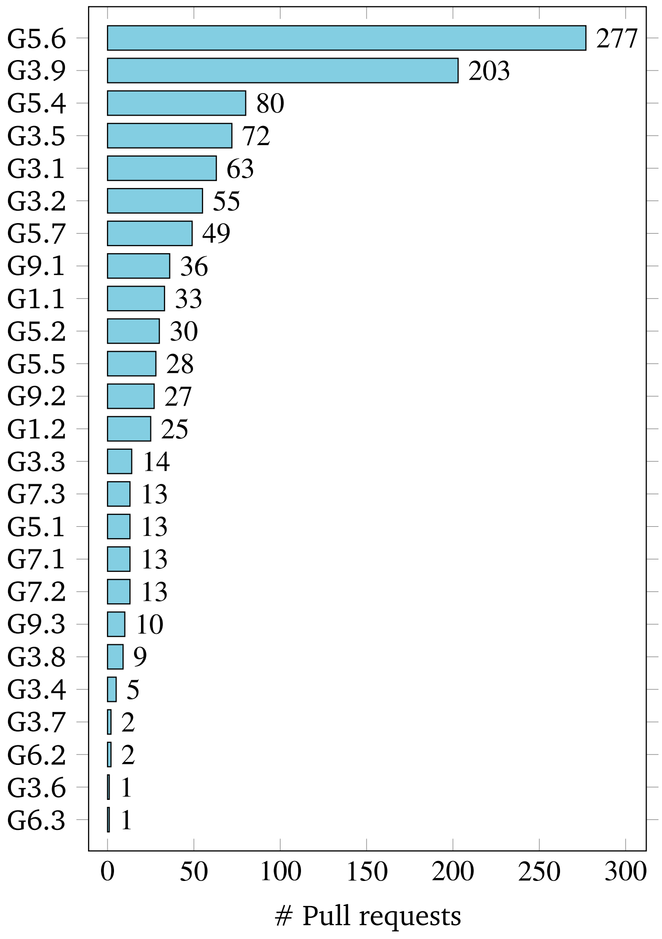

Figure 3: Guidelines observed in code review comments found in open-source repositories

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

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

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

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

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

Table 4: Overview of guidelines and the supporting evidence found in open-source repositories

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

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

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

Adding code comments (G5.2) was requested in 30 PRs to clarify ambiguous code. Examples of the requested comments include explanations for why a variable is set in the test, descriptions of complex locators, reasons for potential errors and how they are handled, as well as deviations from project-specific practices.

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

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

The proper usage of testing techniques and exception handling (G5.6) was observed most frequently with its appearance in 277 PRs. Beyond readability, G5.6 also relates to maintainability. The use of familiar testing patterns improves reviewer comprehension and supports long-term code quality. Reviewers frequently highlighted problems such as incorrect use of mocks, testing at inappropriate levels of abstraction, or improper variable initialization, factors known to contribute to test unreliability [31].

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

Modern testing frameworks provide various locator methods to identify GUI elements based on their properties. Reviewers often recommended replacing less precise locators with more robust alternatives. For example, in one repository, there was a strong preference for using the `findByRole` locator, which targets elements by semantic role (e.g., buttons) over the more generic `findByText` locators. An example of locator methods is shown in Listing 1.

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

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

Listing 1: Example two locators in Cypress targeting the same element on the GUI

Explicit wait statements (e.g., `cy.wait(2000)`), to wait for results/responses of the system under test (SUT), are often not necessary and should be avoided. Reviewers have requested the removal of these explicit wait statements. When timeouts need to be defined, reviewers recommend consolidating the timeout settings either globally or for specific test cases, rather than setting them for individual test steps (see Listing 2).

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

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

Listing 2: Example of an explicit timeout to wait for a transition to another page.

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

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

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

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

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

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

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

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

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

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

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

G9: Follow design principles and patterns. This guideline category promotes the following of design principles and patterns. Specific to GUI-based testing is the page-object model [48], a common abstraction in GUI testing.

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

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

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

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

5. Discussion

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

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

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

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

Third, our analysis of review comments shows frequent attention to test readability (G5), particularly in relation to the appropriate use of testing techniques (G5.6) and correct assertions (G5.7). These aspects are often difficult to automate and remain dependent on reviewer expertise. For example, choosing a meaningful GUI locator strategy, avoiding explicit wait/timeout statements [49], or asserting behavior based on GUI state rather than raw HTML can enhance maintainability and clarity [50]. Additionally, the most suitable testing technique will vary depending on the technology stack and the aim of the testing efforts for a specific project.

Visualizing changes (G6) only appeared in two code review comments. Our previous study found that reviewers often run test cases locally to verify correct interactions with the SUT [38]. Providing visualizations of interactions with the SUT could reduce this need by making execution behavior more transparent. However, support for such features depends heavily on the chosen testing framework. For example, Playwright offers the Trace Viewer[13], a GUI tool that enables detailed inspection of recorded traces after test execution, facilitating debugging of failures in CI. Playwright thus illustrates how visualization can address the need to provide visualizations manually, and we assume similar capabilities will be adopted by other tools.

[11]https://github.com/Kong/insomnia/pull/5076

[12]https://github.com/appsmithorg/appsmith/pull/34925

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

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

Can we explain the absence of observations for certain guidelines? Dependencies between test code and the production code it tests (G3.6) did not appear in review comments, and no clear reason could be identified for this absence. One possible explanation is that when production code and the corresponding test code are part of the same change, the overall summary of changes may be sufficient for reviewers. Further, a reviewer's familiarity with the codebase can have an impact on this guideline, but we cannot assess this based on the data we have gathered. This omission is notable, as reviewers frequently reference related tests or documentation (G3.5). Prior work, including our study [38] and that of Spadini et al. [31], identifies such dependencies as a recurring information need for reviewers.

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

One possible explanation for the lack of comments regarding adherence to requirements (G8.1) is that new contributions to the repository, such as new features and their corresponding tests, should first be discussed in GitHub issues or other platforms before being submitted as pull requests (PRs), as outlined in Section 4.1. Thus, potential discussions around requirements are resolved in other ways.

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

Should the proposed guidelines be refined in light of our empirical observations? Although observed in 203 PRs, comments on file prioritization (G3.9) exhibit a cohesive scope, suggesting no refinement is needed. In contrast, the guideline on testing techniques and exception handling (G5.6) appeared most frequently, with 277 PRs. Given that this guideline is specific to GUI-based testing without a counterpart in production code review guidelines and the large number of diverse examples support refining G5.6, we propose the following fine-grained sub-guidelines, as shown in Table 5.

Table 5: Sub-guidelines for G5.6 and the number of pull requests (PRs) where they were observed

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

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

Avoid explicit waits (G5.6.2): Unnecessary explicit wait statements with arbitrary durations constitute an anti-pattern or test smell [49]. Modern frameworks such as Playwright and Cypress automatically handle waiting for events and interactions, eliminating the need for manual waits, which slow down the execution time. For instance, they resolve pages or locate dynamically loaded elements without additional delay instructions. When specific timeout behavior is required, it should be defined at the test-case level rather than for individual statements. However, synchronization of tests and the SUT remains a significant challenge in GUI-based testing [51].

Avoid test annotations `.skip` and `.only` (G5.6.2): Annotations such as `.skip` and `.only` allow testers to focus the test execution on specific tests or temporarily skip failing tests without needing to delete or modify the test code.

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

However, reviewers have noted the presence of these annotations and have requested their removal. Contributors can use these annotations during test development, but sometimes forget to remove them when submitting a PR for code review.

What is the potential practical value of the guidelines for developers? Test files and GUI-based test files are modified and reviewed less frequently than production code, limiting reviewers' opportunities to develop comparable expertise [38, 31]. A lack of experience in code reviews has a negative long-term effect on software quality [6]. Consequently, the guidelines can support less experienced contributors and reviewers to improve the quality of code reviews.

During the review process, these guidelines can help reduce the cognitive load on reviewers by providing clear criteria to follow. However, as demonstrated by Alegroth et al. [52], programming guidelines can sometimes increase cognitive load for testers instead. Using automation to prepare and assist in the code review process, as observed in our sampled repositories, could reduce the required cognitive load, although it will not eliminate it. Continuous integration pipelines should enforce syntactic and structural checks, such as linting and test coverage thresholds. Additionally, pull request templates should encourage authors to provide context, rationale, and validation steps for GUI-based tests. In general code reviews, automation, if applied in a sensible manner, has in many cases a positive impact on the review process [53].

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

Comparison of results to other studies. Ricca and Stocco [54] surveyed the grey literature to identify best practices in end-to-end web test automation. Several of the practices they identified align closely with our findings from the analysis of code review comments and project guidelines. Table 6 shows this alignment.

Table 6: Overlap between best practices in end-to-end web test automation (as identified by Ricca and Stocco [54] and our guidelines

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

Among the technical best practices discussed, the proper handling of synchronization between the web application and the test code was highlighted most frequently. This practice aligns with our guideline G5.6, which advises against using explicit waits to wait for events from the system under test. Improper handling of synchronization via waits can not only cause an unnecessary increase in test execution time, but also may introduce flaky behavior due to the waits being non-deterministic [54].

Fulcini et al. [49] presented guidelines to avoid test smells specific to GUI-based testing. Test smells, which build upon the concept of code smells [55], refer to patterns in test code that hinder readability, maintainability, and overall quality. Based on their guidelines, they developed a linter to identify test smells. Table 7 illustrates how our guidelines overlap with theirs. Interestingly, their guidelines, R3 to R7, provide a more detailed set of recommendations for using locators. However, they do not mention the role-based locator, which has often been requested in the code review comments we observed. They found that the most frequently violated guidelines in their sampled GitHub repositories were the use of global variables in test cases, followed by the discouraged use of XPath [56] locators. While issues related to XPath locators have been noted in other studies [57], we only encountered them in one pull request comment. Similarly, discussions about global variables were observed in a few pull requests. A potential explanation for this discrepancy is the selection of repositories using Selenium as the testing tool, rather than our choice of Playwright and Cypress. Nevertheless, highlighting test smells automatically would help identify potential maintainability issues in test code during code reviews.

Table 7: Overlap between guidelines for GUI-based testing maintenance by Fulcini et al. [49] and our guidelines

Guideline by [49]	**Guideline**
R2: Keep test cases as simple and short as possible	G7.1
R3: Prefer to use locator by Id, CSS locators and Xpath when not available, in that order	G5.6
R4: Use relative XPath in place of absolute XPath	G5.6
R5: Do not use link locators	G5.6
R6: Use tag locators only for multiple elements	G5.6
R7: Use relative URLs locator instead of absolute ones	G5.6
R8: For elements and variables use name that mirrors functionalities	G5.4
R9: Keep names of variables clear to everyone	G5.4
R19: Run linters to detect anti-pattern	G1.1
R20: Adopt Page Object Pattern	G9.1
R21: Avoid Thread.sleep statements by turning fixed-time waits into condition-based ones	G5.6

6. Threats to Validity

We discuss the threats to the validity of this study in terms of construct validity, internal validity, external validity, and reproducibility. Further, we discuss the three types of threats that must be addressed in software repository mining studies according to Vidoni [58]: repository bias, data bias, and information bias.

6.1. Construct Validity

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

6.2. Internal Validity

A limitation of this study is the qualitative nature of the code review comment analysis. To mitigate researcher bias, the two main authors jointly mapped comments to guidelines in multiple sessions. Each session was limited to a maximum of 90 minutes to avoid researcher fatigue from analyzing a set of 1000 pull requests. Because this process was performed collaboratively, inconsistencies in mapping did not emerge. We consider this manual mapping a strength relative to automated methods. For example, we evaluated an LLM-based classification approach but rejected it due to its low agreement with the manual results, which would have introduced information bias [58].

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

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

6.3. External Validity

Repository Bias. This study collects data exclusively from publicly available open-source repositories hosted on GitHub. Alternative platforms such as GitLab or GitBucket are excluded, which introduces a repository bias [58]. Given GitHub's dominance, with approximately 350 million repositories, this exclusion is unlikely to limit the relevance of our findings when sampling 100 popular repositories for the analysis. GitHub is also the most common source of repositories for MSR studies [58]. It's important to note that these results may vary for less popular repositories. Additionally, this approach indirectly excludes very new repositories, as they have not yet had the opportunity to gain popularity.

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

Open-source software projects differ significantly from closed-source, commercial systems, making it inappropriate to directly transfer insights from open-source to closed-source environments. Open-source projects are typically developed by distributed teams that rely on asynchronous, text-based communication rather than face-to-face interactions. Contributors to open-source projects may participate infrequently or irregularly since they are not contractually tied to the project through employment agreements [59, 60, 61].

Studies by Rigby and Bird [62] and Bosu et al. [63] compare aspects of code reviews in open-source and closed-source systems. Generally, both studies found a substantial overlap between the code review processes in these two types of software development. However, a notable difference lies in the relationships between code review participants, which are more important in open-source projects, while knowledge transfer is a key focus in closed-source projects [63]. We also view knowledge transfer as an important function of code reviews for GUI-based testing. As a result, findings derived from analyzing open-source repositories may not be applicable to proprietary, closed-source environments, where team organization, development workflows, and quality assurance mechanisms can vary considerably.

Data bias. A data bias occurs when there are threats to the validity of the data extracted from the repositories [58]. In our study, this threat emerges from the selection of testing tools. We focus on Playwright and Cypress, excluding Selenium. Although Selenium remains prevalent in industry and academic literature, its omission reflects our emphasis on modern frameworks. Tool-specific features may influence the mapping to our proposed guidelines.

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

6.4. *Reproducability*

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

Data volatility. Temporal factors introduce data volatility, which may threaten both the validity and reproducibility of our results. First, the GitHub API evolves over time, altering query structures, rate limits, and the availability of metadata. Such changes can render past data collection procedures irreproducible or introduce inconsistencies across studies [64]. Second, popular repositories are in continuous development, as contributors frequently merge pull requests, refactor code, and expand test suites. Consequently, metrics such as the number of open pull requests or the size of the test base may differ substantially between two collection points, even if separated by only a few days.

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

7. Conclusion

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

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

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

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

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

CRediT authorship contribution statement

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

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.17104677 or [41].

References

[1] A. E. Randel and K. S. Jaussi. FUNCTIONAL BACKGROUND IDENTITY, DIVERSITY, AND INDIVIDUAL PERFORMANCE IN CROSS-FUNCTIONAL TEAMS. *Academy of Management Journal*, 46 (6):763–774, December 2003. ISSN 0001-4273, 1948-0989. doi:10.2307/30040667.

[2] P. Layzell, O.P. Brereton, and A. French. Supporting collaboration in distributed software engineering teams. In *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pages 38–45, Singapore, 2000. IEEE Comput. Soc. ISBN 978-0-7695-0915-0. doi:10.1109/APSEC.2000.896681.

[3] Kwan-Sik Na, Xiaotong Li, James T. Simpson, and Ki-Yoon Kim. Uncertainty profile and software project performance: A cross-national comparison. *Journal of Systems and Software*, 70(1-2): 155–163, February 2004. ISSN 01641212. doi:10.1016/S0164-1212(03)00014-1.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355, Hyderabad India, May 2014. ACM. ISBN 978-1-4503-2756-5. doi:10.1145/2568225.2568260.

[9] Mary-Luz Sánchez-Gordón and Ricardo Colomo-Palacios. From certifications to international standards in software testing: mapping from isqtb to iso/iec/ieee 29119-2. In *European Conference on Software Process Improvement*, pages 43–55. Springer, 2018.

[10] 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.

[11] 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.

[12] 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.

[13] Michael Dorner, Daniel Mendez, Ehsan Zabardast, Nicole Valdez, and Marcin Floryan. Measuring information diffusion in code review at spotify, 2024. URL `https://arxiv.org/abs/2406.12553`.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Vahid Garousi and Mika V. Mäntylä. A systematic literature review of literature reviews in software testing. *Information and Software Technology*, 80:195–216, December 2016. ISSN 09505849. doi:10.1016/j.infsof.2016.09.002.

[20] 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.

[21] Juha Itkonen, Mika V. Mantyla, and Casper Lassenius. How do testers do it? An exploratory study on manual testing practices. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 494–497, Lake Buena Vista, FL, USA, October 2009. IEEE. ISBN 978-1-4244-4842-5. doi:10.1109/ESEM.2009.5314240.

[22] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1):14–30, January 2010. ISSN 09505849. doi:10.1016/j.infsof.2009.07.001.

[23] Roman Haas, Daniel Elsner, Elmar Juergens, Alexander Pretschner, and Sven Apel. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1281–1291, Athens Greece, August 2021. ACM. ISBN 978-1-4503-8562-6. doi:10.1145/3468264.3473922.

[24] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling Manual and Automated Testing: The AutoTest Experience. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*, pages 261a–261a, Waikoloa, HI, January 2007. IEEE. ISBN 978-0-7695-2755-0. doi:10.1109/HICSS.2007.462.

[25] Emil Alégroth, Robert Feldt, and Lisa Ryrholm. Visual GUI testing in practice: Challenges, problemsand limitations. *Empirical Software Engineering*, 20(3):694–744, June 2015. ISSN 1382-3256, 1573-7616. doi:10.1007/s10664-013-9293-5.

[26] Marko Savic, Mika Mantyla, and Maelick Claes. Win GUI Crawler:

A tool prototype for desktop GUI image and metadata collection. In *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 223–228, Valencia, Spain, April 2022. IEEE. ISBN 978-1-6654-9628-5. doi:10.1109/ICSTW55395.2022.00046.

[27] Riccardo Coppola, Luca Ardito, Maurizio Morisio, and Marco Torchiano. Mobile Testing: New Challenges and Perceived Difficulties From Developers of the Italian Industry. *IT Professional*, 22(5):32–39, September 2020. ISSN 1520-9202, 1941-045X. doi:10.1109/MITP.2019.2942810.

[28] 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.

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

[30] 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.

[31] 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.

[32] Fabian Fagerholm, Michael Felderer, Davide Fucci, Michael Unterkalmsteiner, Bogdan Marculescu, Markus Martini, Lars Göran Wallgren Tengberg, Robert Feldt, Bettina Lehtelä, Balázs Nagyváradi, and Jehan Khattak. Cognition in Software Engineering: A Taxonomy and Survey of a Half-Century of Research. *ACM Computing Surveys*, 54(11s):1–36, January 2022. ISSN 0360-0300, 1557-7341. doi:10.1145/3508359.

[33] Lucian José Gonçales, Kleinner Farias, and Bruno C. Da Silva. Measuring the cognitive load of software developers: An extended Systematic Mapping Study. *Information and Software Technology*, 136:106563, August 2021. ISSN 09505849. doi:10.1016/j.infsof.2021.106563.

[34] Lucian Gonçales, Kleinner Farias, Bruno Da Silva, and Jonathan Fessler. Measuring the Cognitive Load of Software Developers: A Systematic Mapping Study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 42–52, Montreal, QC, Canada, May 2019. IEEE. ISBN 978-1-7281-1519-1. doi:10.1109/ICPC.2019.00018.

[35] Randall K. Minas, Rick Kazman, and Ewan Tempero. Neurophysiological Impact of Software Design Processes on Software Developers. In Dylan D. Schmorrow and Cali M. Fidopiastis, editors, *Augmented Cognition. Enhancing Cognition and Behavior in Complex Human Environments*, volume 10285, pages 56–64. Springer International Publishing, Cham, 2017. ISBN 978-3-319-58624-3 978-3-319-58625-0. doi:10.1007/978-3-319-58625-0_4.

[36] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do explicit review strategies improve code review performance? Towards understanding the role of cognitive load. *Empirical Software Engineering*, 27(4):99, July 2022. ISSN 1382-3256, 1573-7616. doi:10.1007/s10664-022-10123-8.

[37] Pavlína Wurzel Gonçalves, Enrico Fregnan, Tobias Baum, Kurt Schneider, and Alberto Bacchelli. Do Explicit Review Strategies Improve Code Review Performance? In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 606–610, Seoul Republic of Korea, June 2020. ACM. ISBN 978-1-4503-7517-7. doi:10.1145/3379597.3387509.

[38] Andreas Bauer, Tomas Helmfridsson, Emil Alégroth, and Georg-Daniel Schwarz. When gui-based testing meets code reviews. Manuscript submittet to *Software Testing, Verification and Reliability*, 2025.

[39] Klaas-Jan Stol and Brian Fitzgerald. The ABC of Software Engineering Research. *ACM Transactions on Software Engineering and Methodology*, 27(3):1–51, July 2018. ISSN 1049-331X, 1557-7392. doi:10.1145/3241743.

[40] Ahmed Hassan. Mining Software Repositories to Assist Developers and Support Managers. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 339–342, Philadelphia, PA, USA, September 2006. IEEE. ISBN 978-0-7695-2354-5. doi:10.1109/ICSM.2006.38.

[41] Andreas Bauer and Florian Angermeir. Replication package: The prevalence of code review guidelines for gui-based testing in open-source, September 2025. URL `https://doi.org/10.5281/zenodo.17104677`.

[42] 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.

[43] M Niranjanamurthy, Sushanth Navale, S Jagannatha, and Sudesha Chakraborty. Functional Software Testing for Web Applications in the Context of Industry. *Journal of Computational and Theoretical Nanoscience*, 15(11):3398–3404, November 2018. ISSN 1546-1955. doi:10.1166/jctn.2018.7632.

[44] Boni García, Jose M. Del Alamo, Maurizio Leotta, and Filippo Ricca. Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. In Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, editors, *Quality of Information and Communications Technology*, volume 2178, pages 142–149. Springer Nature Switzerland, Cham, 2024. ISBN 978-3-031-70244-0 978-3-031-70245-7. doi:10.1007/978-3-031-70245-7_10.

[45] Boni García, Jose M. Del Alamo, Maurizio Leotta, and Filippo Ricca. Exploring Browser Automation: A Comparative Study of Selenium, Cypress, Puppeteer, and Playwright. In Antonia Bertolino, João Pascoal Faria, Patricia Lago, and Laura Semini, editors, *Quality of Information and Communications Technology*, volume 2178, pages 142–149. Springer Nature Switzerland, Cham, 2024. ISBN 978-3-031-70244-0 978-3-031-70245-7. doi:10.1007/978-3-031-70245-7_10.

[46] Sacha Greif. State of JS 2024 survey, 2024. URL `https://2024.stateofjs.com/en-US/libraries/testing/`.

[47] Maurizio Leotta, Boni García, Filippo Ricca, and Jim Whitehead. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 339–350, Dublin, Ireland, April 2023. IEEE. ISBN 978-1-66545-666-1. doi:10.1109/ICST57152.2023.00039.

[48] 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.

[49] 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.

[50] Marco De Luca, Anna Rita Fasolino, and Porfirio Tramontana. Investigating the robustness of locators in template-based Web application testing using a GUI change classification model. *Journal of Systems and Software*, 210:111932, April 2024. ISSN 01641212. doi:10.1016/j.jss.2023.111932.

[51] 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.

[52] Emil Alegroth, Elin Petersen, and John Tinnerholm. A Failed attempt at creating Guidelines for Visual GUI Testing: An industrial case study. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 340–350, Porto de

Galinhas, Brazil, April 2021. IEEE. ISBN 978-1-72816-836-4. doi:10.1109/ICST49551.2021.00046.

[53] 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.

[54] Filippo Ricca and Andrea Stocco. Web Test Automation: Insights from the Grey Literature. In Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science*, volume 12607, pages 472–485. Springer International Publishing, Cham, 2021. ISBN 978-3-030-67730-5 978-3-030-67731-2. doi:10.1007/978-3-030-67731-2_35.

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

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

[57] 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.

[58] M. Vidoni. A systematic process for Mining Software Repositories: Results from a systematic literature review. *Information and Software Technology*, 144:106791, April 2022. ISSN 09505849. doi:10.1016/j.infsof.2021.106791.

[59] Ann Barcomb, Klaas-Jan Stol, Brian Fitzgerald, and Dirk Riehle. Managing Episodic Volunteers in Free/Libre/Open Source Software Communities. *IEEE Transactions on Software Engineering*, 48 (1):260–277, January 2022. ISSN 0098-5589, 1939-3520, 2326-3881. doi:10.1109/TSE.2020.2985093.

[60] Ann Barcomb, Klaas-Jan Stol, Dirk Riehle, and Brian Fitzgerald. Why Do Episodic Volunteers Stay in FLOSS Communities? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 948–959, Montreal, QC, Canada, May 2019. IEEE. ISBN 978-1-7281-0869-8. doi:10.1109/ICSE.2019.00100.

[61] Amanda Lee and Jeffrey C. Carver. Are One-Time Contributors Different? A Comparison to Core and Periphery Developers in FLOSS Repositories. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, Toronto, ON, November 2017. IEEE. ISBN 978-1-5090-4039-1. doi:10.1109/ESEM.2017.7.

[62] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212, Saint Petersburg Russia, August 2013. ACM. ISBN 978-1-4503-2237-9. doi:10.1145/2491411.2491444.

[63] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, and Christopher Chockley. Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft. *IEEE Transactions on Software Engineering*, 43(1):56–75, January 2017. ISSN 0098-5589, 1939-3520. doi:10.1109/TSE.2016.2576451.

[64] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071, October 2016. ISSN 1382-3256, 1573-7616. doi:10.1007/s10664-015-9393-5.