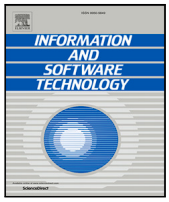
Information and Software Technology 163 (2023) 107299

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Code review guidelines for GUI-based testing artifacts

Andreas Bauer [a,*], Riccardo Coppola [b], Emil Alégroth [a], Tony Gorschek [a,c]

[a] Software Engineering Research Lab (SERL), Blekinge Institute of Technology, Karlskrona, 37179, Sweden
[b] Department of Control and Computer Engineering, Politecnico di Torino, Turin, Italy
[c] Fortiss, Germany

ARTICLE INFO

Dataset link: https://zenodo.org/record/7248201

Keywords:
GUI testing
GUI-based testing
Software testing
Code review
Modern code review
Guidelines
Practices

ABSTRACT

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

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

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

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

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

1. Introduction

The development and evolution of software is a complex undertaking dependent on a high degree of collaboration between many professionals with varying skill sets and expertise ranging from different technical skills to domain knowledge [1]. This has led to a rise and reliance on team-based activities to enable this type of collaboration. One commonly used practice of team-enabling activities is code reviews [2,3]. Code review is a software engineering practice where peers review a code contribution before additions or changes are integrated into the code base [4,5]. As a practice, code reviews help to catch errors and improve sub-optimal solutions on different types of artifacts, but also enable knowledge and experience sharing and, maybe most importantly, allow active collaboration in the team [5,6].

Another practice that has become ubiquitous in modern software development is automated testing [7]. Automated testing is used to verify the conformance of the developed software to its requirements in an automated fashion and is run frequently to provide continuous feedback to developers. Test automation can be performed on multiple levels of system abstraction, from low-level unit tests to high-level system tests and Graphical User Interface (GUI) tests. Notably, GUI-based tests can verify the behavior of a system through interactions with its GUI in a similar way a user would [8,9]. Thus, GUI-based testing constitutes a powerful tool to find regression defects that user would otherwise encounter [8].

Guidelines within Software Engineering aim to provide actors in the software development process with best practices for high-quality software development. But most existing guidelines that seek to improve the effectiveness and efficiency of code reviews focus on source code and lower-level test code [10]. In fact, to the best of our knowledge, no review guidelines have been proposed for GUI-based tests, even though similar quality benefits can be assumed to source code due to the GUI test code's similar characteristics. This observation is supported by results from interviews with GUI-based testing experts from the

* Corresponding author.
E-mail addresses: andreas.bauer@bth.se (A. Bauer), riccardo.coppola@polito.it (R. Coppola), emil.alegroth@bth.se (E. Alégroth), tony.gorschek@bth.se (T. Gorschek).

https://doi.org/10.1016/j.infsof.2023.107299
Received 7 November 2022; Received in revised form 16 April 2023; Accepted 29 June 2023
Available online 5 July 2023

industry, who state that they use ad hoc guidelines due to the lack of general guidelines in the literature. Coupled with reports that GUI-based tests often suffer quality issues, we see a concrete need for research in this area to both aid industrial practitioners and bridge a gap in knowledge in academia.

In this study, we investigate existing literature on guidelines for code reviews of software development artifacts for the purpose of mapping such guidelines for use on GUI-based testing artifacts. For this investigation, we performed a Systematic Literature Review (SLR) complemented by a gray literature review to extract review guidelines for source and test code. These results are then synthesized into a set of nine code review guideline categories. Using expert judgement and experience with GUI-based test techniques, these categories are then mapped, through examples of application, to GUI-based testing to create review guidelines for GUI-based testing artifacts. We stress that the study is delimited to guidelines explicitly related to review of artifacts, e.g., code files. Thus, omitting guidelines associated with less tangible aspects of the review process or review environment, such as reviewer selection or effort allocation [11–13].

Based on our results, we claim the following contributions:

- A macro-analysis of review guidelines of source and test code in existing white and gray literature;
- A set of nine code review guideline categories that are mapped as applicable to GUI-based test artifacts;
- Examples of how to apply the identified code review guideline categories to GUI-based test artifacts.

The remainder of the paper is organized as follows: in Section 2 we provide background information about GUI-based testing and Modern Code Reviews; in Section 3 we illustrate the methodology that we employed to conduct our literature review; in Section 4 we report the results of the review; in Section 5 we discuss the results and the threats to validity of the present study; in Section 6 we conclude the paper by summarizing the main findings and providing future research directions.

2. Background and related work

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

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

2.1. GUI-based testing

Graphical user interface (GUI) testing can be utilized for different test purposes, categorized on a general level of abstraction into two types; (1) GUI tests and (2) GUI-based tests [8]. GUI testing is defined as the practice of verifying the correctness of the GUI's visual appearance according to the system under tests (SUT) requirements, e.g., that components have the right shape, color and positioning. GUI-based tests, on the other hand, are defined as the practice of testing a SUT's conformance to its functional requirements through its GUI [8,9]. These tests are thereby system-level tests that are performed through End-to-End (E2E) scenarios—Sequences of events performed against the GUI [14]—that are executed with test drivers that can interact with the SUT's visual GUI, the GUI model or other GUI-related interfaces. These tests can be manual or automated and are commonly used for regression testing [7]. However, for this work, we primarily discuss automated GUI testing with tests written as testware (e.g., test scripts or test code) or model-based tests (e.g., visual, textual or formal models) [15].

GUI-based testing can be categorized in different ways. One categorization is based on the way the test cases are defined where three categories of tests are usually discussed, i.e., scripted tests, capture and replay and model-based tests. *Scripted* GUI-based testing is based on the development of test scripts, or test code, with the usage of dedicated automation APIs, tools and frameworks (e.g., Selenium in the web application domain [16]). *Capture & Replay* testing resorts on providing instruments, usually in the form of tools, to record the operations performed by a tester, or user, on the GUI of the system, to generate re-executable test sequences [17]. *Model-based* testing is based on the, either manual or automated, generation of models of the SUT's intended behavior from a GUI level of abstraction. These models are then used to generate test sequences that provide coverage of the different states of the GUI [18]. These models may be visual (i.e., defined with nodes and vertexes) but can also be textual (e.g., utilized in behavioral-driven development) or formal (e.g., mathematical models).

Another categorization of GUI-based test techniques, provided by Alégroth et al. [19], is based on the GUI-based tests means of interacting with the SUT, i.e. their test drivers. In this categorization, three different *generations* of GUI testing approaches are identified. *Coordinate-based*, or *first generation* GUI testing, is an approach driven by tools that identify the elements of the GUI through their coordinates on the screen. These tools generally utilize the capture and replay paradigm, where coordinates to GUI elements are provided by a human user during the recording of test sequences. *Component-based*, or *second generation* testing, is an approach driven by tools that perform interactions through the GUI's layout model or properties, e.g., through access to an application's document object model (DOM) on web applications or other GUI interfaces. Lastly, *Image recognition-based*, or *third generation GUI* testing, is an approach driven by tools that utilize computer vision to identify elements on the screen based on their visual appearance [20].

Whilst these classifications differ and outline a plethora of different types of automated GUI-based testing approaches, they are all used for the same test purpose, i.e., system-level testing. However, whilst research into the technology used by the approaches is ubiquitous, research into guidelines and ways of using the approaches is less common, leaving a gap in knowledge. Development guidelines for GUI-based tests have been proposed [21], but guidelines for reviews of GUI-based tests are, to the best of our knowledge, not covered in the body of knowledge. Such guidelines are perceived important to aid improve the effectiveness and efficiency of both individual and team-based GUI testing.

2.2. Modern code review

Code reviews started as a waterfall-like procedure back in the 1980s as software inspections, a practice where other developers manually inspect artifacts to improve an artifact's quality by verifying their correctness [22,23]. Fagan [24] described this early form of software inspection as a formal and highly structured process. As such, this practice has been shown to be effective in finding both errors and improvement potentials, applied throughout the development process of any product. Nevertheless, the formal nature of this approach and the resulting overhead hindered the adoption of software inspections in the past [25].

Nowadays, code reviews have transformed towards informal, tool-supported, lightweight processes that build a communication platform for developers [5,26–28]. This current form of code reviews is a common practice in most agile software development companies and open source software projects, also known as Modern Code Review (MCR) [5,29]. Dedicated code review tools such as Gerrit [30], Phabricator [31], or CodeFlow [32] support the code review process by providing context-specific information. Modern software forges such

as GitHub [33], GitLab [34], or Bitbucket [35] even integrate the functionalities of these dedicated tools as a part of their collaborative software development workflows.

In this work, we use the term *code change* to refer to added, deleted or changed code that is inspected through a code review. This practice is instigated by the change provider, e.g., a developer, who submits an artifact for review by a reviewer. Other terms, used interchangeably in literature for this practice, are *changeset* or *patch*. The latter is a common term in the Open Source Software community to describe artifact contributions [28].

Davila and Nunes [29] performed an SLR on the topic of modern code reviews and proposed a taxonomy for the approach. Through this work, the authors identified studies that propose strategies for MCR, which are categorized into phases of the MCR process that are; (1) review planning and setup, (2) code review, and (3) process management and support. In the planning category, most studies focus on reviewer recommendation and automated selection of reviewers. Only one study in this category is related to the code change itself, proposing a tool to add a narrative to the code and multimedia resources to support code change documentation. Reviewers can then reproduce the change and provide feedback based on a replay of these multimedia comments. In the code review category, the main identified strategies to support the code review process were to provide visualizations of code changes, present properties associated with the changes, and support for the analysis of change impact. Of the 22 included studies, 15 studies propose code-checking tasks for the purpose of understanding code changes. In 10 of these 15 studies, approaches to code change visualization are proposed.

In a study by Dong et al. [10], 57 practices and 19 code review pains from Open Source Software and industrial communities are summarized. From these results, best practices are derived and organized in 5 steps based on the lifecycle of the MCR process. One conclusion of their study is that the code context is one of the most difficult things for reviewers to understand.

Analysis of the included papers in the aforementioned work indicates that MCR is mostly studied in the context of software development. However, in a study by Spadini et al. [36], test code reviews were also investigated, in particular how developers discuss such code. Results show that reviewers mainly discuss code improvements, suggesting better testing practices and some generic code quality practices to ensure the maintainability and readability of test code. These discussions also served to provide code improvement to understand whether a test covers all paths of the production code, i.e. provide coverage. This is an interesting outcome of the study since coverage is a relevant attribute for test code reviewers that is not relevant for source code.

Another practice that was observed was that test code reviewers requested clarification of the implementation intention by asking for the rationale of the change. For knowledge transfer, reviewers link to external resources containing documentation or example to solve a problem that is part of the code change. In other cases, reviewers directly provide an example of how to tackle an issue that is discussed during a code review. The outcomes from the study of reviews of test cases overlap with the findings of Bacchelli and Bird [5]. For instance, the challenges that reviewers of both source and test code face are due to a lack of context and reasonable navigation possibilities within code review tools. Such functionality is valuable since, when reviewing test code, reviewers must often switch between the test and production code back and forth to understand the impact of a change. The paper concludes with features for future code review tools, such as ways of providing context information to more easily understand and inspect the classes that are under test and their dependencies, enable easy navigation between test and production code, and provide detailed code coverage information for tests.

3. Study design

The objective of this work is to identify guidelines for the review of GUI-based testing artifacts by mapping guidelines from source and test code review to GUI-based testing. These guidelines, from existing literature, are acquired through a systematic literature review complemented with a gray literature review. Before that, we conducted six interviews with industrial GUI testing experts to ensure that there is an industry need for guidelines. The mapping is performed by looking at characteristics that are common to GUI test artifacts compared to production and non-GUI test artifacts and what review guidelines target each characteristic. Logical inference is then used, together with the author's expert judgment and experience with GUI testing (over 30 years of combined knowledge from industry and academia), to map each guideline as applicable to GUI testing. Using logical inference, a person (or machine) goes beyond available evidence to form a conclusion Johnson-Laird [37]. Thus, a conclusion is valid if the premises (a guideline can be applied to GUI testing) is true but do not follow any specific logic. In detail, this mapping is performed in a systematic top-down approach, where we (1) evaluate the purpose of each guideline, (2) group guidelines in categories and (3) provide examples of how these guidelines apply to GUI test artifact review. Thereby providing a chain of logic to tailor the original guideline for source or test code to GUI test artifacts.

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

3.1. Research questions

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

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

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

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

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

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

3.2. Methodology

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

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

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

The collaborative sessions were conducted according to the mini-Delphi method for face-to-face meetings, where meeting sessions were closed once a consensus was reached between the authors [38]. These sessions were utilized for (1) the design of the literature review (2) the literature search (3) the analysis of acquired literature sources and (4) the mapping of results to GUI-based test artifacts.

We used an online spreadsheet (Google Sheet) to organize the data through all the phases of the methodology. The spreadsheet also enabled collaborative, asynchronous, work whilst retaining traceability of guidelines back to their original sources.[1]

In the continuation of this section, the detailed activities of each phase of the study will be presented. Wherever suitable, practices that affect the research validity have been added. For a longer discussion about the study's threats to validity, we refer the reader to Section 5.1.

3.2.1. Interviews with experts

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

The interviewees were sampled through convenience sampling—Convenience sampling is a non-random sampling where members meet certain practical criteria [39]—from our industrial network. To fulfill the practical criteria, an interviewee must (1) be a professional engineer working in the industry, (2) have more than 5 years of industrial experience with GUI testing, (3) speak English or Swedish, (4) be available when we conduct the study. As a tool-related question, we asked the interviewees *"What GUI-based testing approaches do you have experience with?"* with the follow-up question *"What were the tools you used for it?"*. We used these mentioned tools as additional gray literature sources to extract guidelines from a tool's website, manuals, and wikis. The interviewees were also asked if they adopted code reviews of GUI test for quality assurance and if they had used or knew of any general guidelines for GUI-based testing or reviews. However, although the interviewees use code reviews, they knew of no general guidelines, stating instead that the guidelines they had used were developed ad hoc to fit specific contextual needs. The interviewees also stated that general guidelines would, based on their own experiences, be of both industrial interest and benefit.

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

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

3.2.2. White literature collection

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

The SLR was conducted following a subset of Kitchenham's guidelines to conduct Systematic Literature Reviews [40]. Specifically, we applied all the steps of the Planning and Conducting phase in Kitchenham's guidelines, with the exception of the prescribed forward and backward snowballing steps after the first phase of collecting literature sources. We did not deem it necessary to perform snowballing because of the size of the set of gathered literature and because of the theoretical saturation of concepts elicited from the first round. Although this represents a threat to the validity of the result, due to the reasons stated, we find this threat to be minor.

The white literature collection, as stated in Fig. 1, involved the following steps:

Selection of Digital Libraries: Five digital libraries were used for our search; IEEE Xplore, ACM Digital Library, Science Direct, Springer Link, and Google Scholar. These were chosen because of their common use in software engineering literature reviews and their perceived complementary overlap of the literature [41].

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

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

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

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

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

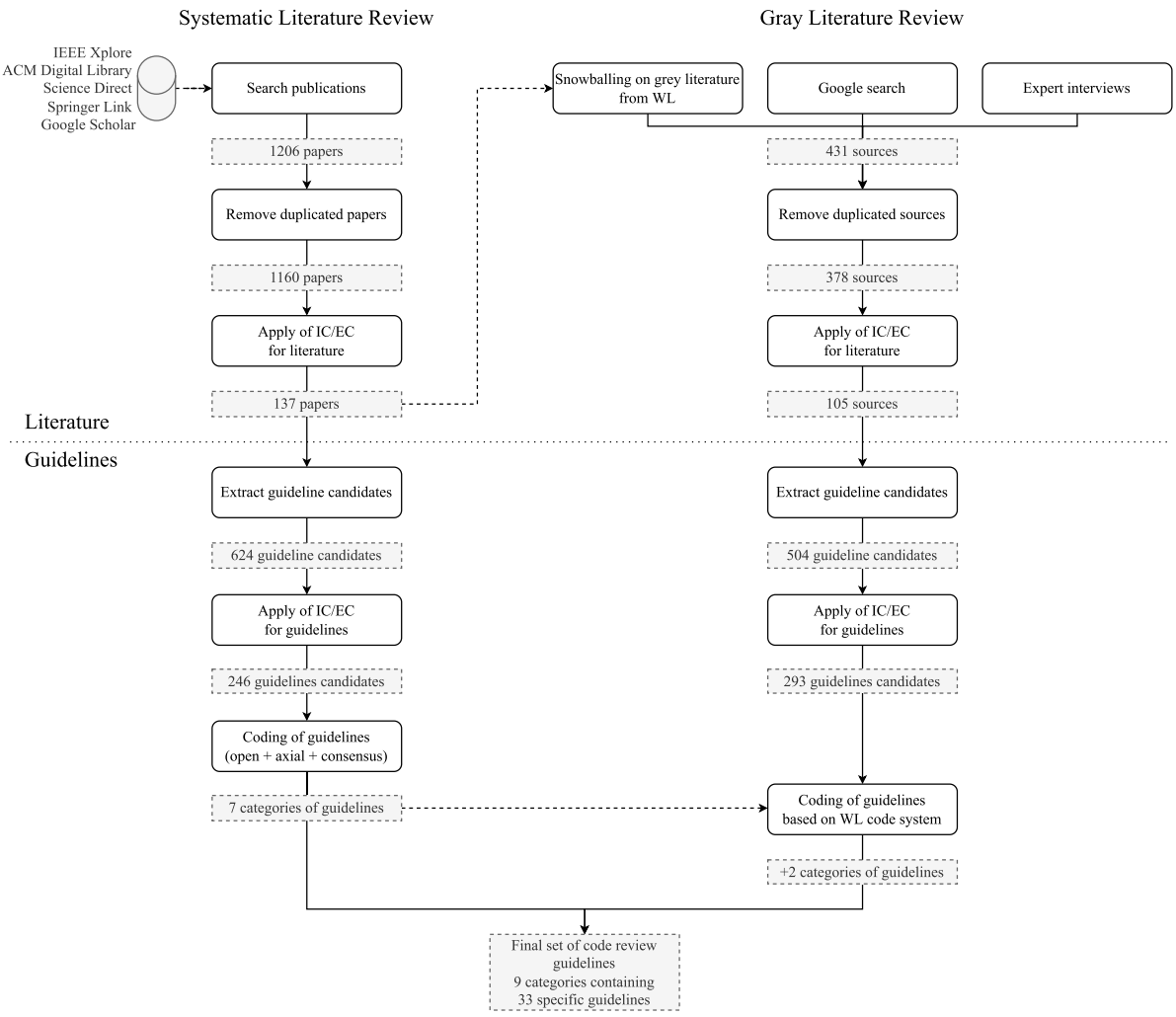


Fig. 1. Summary of the research approach.

Because of the high number of returned results, we limited the search on Science Direct to title, abstract and keywords, on ACM Digital Library to abstract and title, and on Springer Link and Scholar to titles. To enable reproducibility of the study, we limited our search results to the end of June 2022. Table 1 lists the number of sources extracted from each library by applying the search string.

To facilitate the continued data extraction and analysis phases, we organized all found sources in a spreadsheet (data extraction form [40]) and marked each source with the following attributes: *unique identifier*, *library*, *title*, *authors*, *year of publication*, *article URL*, *venue*, *exclusion reason*, and *comments*. Before data extraction, the form was discussed by all authors.

Duplicate Removal: After the sources had been extracted, we applied an automated text analysis script on the elicited CSV files to remove all the duplicated papers from our initial set (i.e., papers with the same name and same set of authors). If a source was listed in multiple digital libraries, we kept only a single instance of the paper in our final paper pool, keeping the source from the library where the source was first encountered (according to the order in Table 1).

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

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

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

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

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

3.2.3. Gray literature collection

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

Table 1
Search results by library.

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

In an attempt to address this lack of information, we extended the literature review to include also gray literature sources. As such, the study can be classified as a Multivocal Literature Review (MLR) [42].

Gray literature (GL) is defined as *what is produced on all levels of government, in academia, business or industry in print and electronic formats, but which is not controlled by commercial publishers, i.e., where publishing is not the primary activity of the producing body* [43].

Adams et al. classify gray literature into three different categories: 1st tier (or high credibility), which includes books, magazines, government reports, and white papers; 2nd tier (or moderate credibility), which includes annual reports, news articles, presentations, videos, question and answers on websites; 3rd tier (or low credibility), which includes blogs, evidence from e-mails, posts on social networks [44]. We used this classification of GL as a template for our quality assessment questionnaire (Table 2) with the following adjustments to aid our research goals: 1st tier, Books, magazines, reports or documentation from companies; 2nd tier, News articles, presentations, videos, Wiki articles; 3rd tier, Blogs, emails, tweets, Q/A sites (such as StackOverflow).

The use of gray literature served two distinct goals: (1) to attempt to complement the missing information from the academic literature and (2) to provide guidelines based on an industrial perspective [45].

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

Snowballing from White Literature: We applied *Backward Snowballing*, defined by C. Wohlin as *using the reference list of a paper to identify additional papers* [46]. Backward Snowballing was applied only to the final set of collected white literature sources, i.e., after the application of inclusion criteria. However, the snowballing was limited to only gray literature sources. This is perceived as a possible threat to the search, i.e., that relevant white literature sources may have been missed. However, due to the comprehensive set of found sources, this threat is considered minor.

As with white literature, we organized the gray literature sources in a form that consists of *unique identifier*, *title*, *article URL*, *source (Google, interviews, snowballing)*, *white literature reference if from snowballing*, *publication year*, *quality attributes (see Table 2)*.

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

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

Quality Assessment: To evaluate the quality of the collected gray literature sources, we applied a quality assessment questionnaire, which is presented in Table 2. We used a 3-point Likert scale (yes = 1, partly = 0.5, and no = 0) for the authority and outlet criteria. For the backlinks, we used a 4-point scale based on the percentile of the number of backlinks in our sample (≥75th percentile = 1, ≥50th percentile = 0.75, ≥25th percentile = 0.5, <25th percentile = 0).

The questionnaire was adapted from the quality assessment questionnaire proposed in [42], where all the sources that had a score higher than 2.5, in a range between 0 and 5, were kept in the final set of sources.

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

The described steps conducted for gray literature search allowed us to obtain 378 gray literature sources (from snowballing, search on the Google search engine, and tool mentions in expert interviews) without duplicates. After applying the inclusion criteria and the final quality assessment we were left with 105 gray literature sources that are used to extract candidate guidelines in the next step.

3.2.4. Data extraction

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

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

For each candidate guideline we collected the following information:

- The full text in the literature item describing the guideline;
- The nature of the guideline, i.e., if it is defined for source or test code;
- If it is an explicit guideline, i.e., an actionable practice or interpreted as such;

Inclusion/Exclusion Criteria for guidelines: We defined a set of inclusion and exclusion criteria for the guidelines to be kept in our final set of sources. These criteria were used to filter the candidate guidelines and ensure consistent inclusion by all authors.

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

Table 2
Quality assessment questionnaire for gray literature. We use a 3-point Likert scale (yes = 1, partly = 0.5, and no = 0) for the authority and outlet criteria.

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

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

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

3.2.5. Data analysis and coding

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

The procedure used in this phase was inspired by the Straussian Grounded Theory approach [47], more specifically the *coding* procedure they propose. Grounded Theory is a data-driven approach to construct a theory from raw data (i.e., data mined from the sources in our final set). The Straussian definition of Grounded Theory differs from the original definition of the technique (i.e., *Glaserian* Grounded Theory [48]) since it allows the definition of research questions up-front, before starting analysis of the raw data.

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

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

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

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

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

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

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

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

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

4. Results

In this section, we will present our results on used guidelines for source and test artifacts to answer our research questions. At first, we provide a macro analysis of the gathered white and gray literature sources, which includes the distribution of literature per year and a description of how guidelines from white and gray literature overlap. These results, combined with test-specific guidelines that are marked in Table 3, provide an answer to RQ1.

Next, we present the categories of used guidelines in Table 4, followed by a structured textual format with descriptions and examples of the application to GUI-based test artifacts, summarized in Table 5, to answer RQ2.

Table 3
Overview of code review guidelines used for source and test artifact review (WL = white literature, GL = gray literature, TS = test specific).

ID	Guideline	#WL	#GL	TS	WL sources	GL sources
G1	**Perform automated checks**					
G1.1	Perform automated checks on the change	21	21	No	S1, S2, S3, S9, S15, S18, S19, S20, S25, S26, S27, S28, S30, S36, S39, S42, S46, S60, S65, S75, S79	GS1, GS4, GS6, GS7, GS8, GS9, GS11, GS13, GS17, GS18, GS21, GS26, GS30, GS36, GS37, GS47, GS49, GS50, GS55, GS61, GS62
G1.2	Perform automated checks for code style	6	11	No	S36, S47, S56, S73, S76, S77	GS5, GS11, GS14, GS15, GS17, GS21, GS45, GS50, GS52, GS53, GS54
G2	**Use checklists**					
G2.1	Provide/use a checklist	8	19	No	S2, S6, S16, S19, S34, S42, S61, S76	GS2, GS4, GS8, GS9, GS11, GS13, GS14, GS15, GS20, GS22, GS26, GS30, GS37, GS39, GS42, GS47, GS54, GS56, GS62
G3	**Provide context information**					
G3.1	Provide additional context information	17	19	No	S12, S14, S17, S22, S35, S42, S46, S48, S50, S54, S62, S68, S71; S72, S73, S74, S77	GS2, GS8, GS9, GS13, GS16, GS24, GS27, GS30, GS35, GS36, GS37, GS39, GS42, GS46, GS51, GS54, GS58, GS59, GS63
G3.2	Provide rationale for the change	10	14	No	S3, S19, S21, S22, S35, S36, S75, S77, S78, S79	GS5, GS12, GS13, GS14, GS24, GS26, GS28, GS36, GS40, GS42, GS47, GS50, GS57, GS59
G3.3	Provide context information about the impact of the change	1	1	No	S22	GS3
G3.4	Provide information about the design and architecture of code affected by the change	2	4	No	S58, S63	GS5, GS21, GS54, GS59
G3.5	Provide links to related resources and documentation	6	13	No	S7, S10, S22, S29, S69, S79	GS3, GS5, GS13, GS17, GS19, GS21, GS35, GS42, GS43, GS44, GS50, GS51, GS59
G3.6	Provide information about the dependencies between test and production code	1	0	Yes	S77	
G3.7	Provide context information about the history of changes	4	0	No	S18, S35, S62, S75	
G3.8	Provide information about test edge cases	1	2	Yes	S77	GS43, GS59
G3.9	Provide a prioritization of the files/classes of the change	2	1	No	S53, S55	GS36
G4	**Utilize metrics**					
G4.1	Measure and monitor code metrics	7	7	No	S35, S36, S37, S40, S45, S51, S62	GS2, GS4, GS26, GS31, GS39, GS37, GS48
G4.2	Provide metrics about execution time (for efficiency)	1	2	No	S47	GS21, GS63
G4.3	Provide test coverage metrics (for effectiveness)	3	2	Yes	S55, S57, S77	GS31, GS34
G5	**Ensure readability**					
G5.1	Ensure readability of the change	5	9	No	S1, S39, S42, S47, S76	GS5, GS13, GS21, GS23, GS31, GS41, GS42, GS50, GS57
G5.2	Provide comments	2	7	No	S47, S78	GS3, GS5, GS6, GS22, GS26, GS29, GS47
G5.3	Follow coding style and naming conventions	2	23	No	S38, S77	GS3, GS5, GS8, GS10, GS16, GS17, GS19, GS20, GS21, GS23, GS29, GS31, GS32, GS33, GS34, GS35, GS36, GS40, GS41, GS42, GS43, GS51, GS63
G5.4	Follow coding style and naming practices in test writing	2	0	Yes	S67, S77	
G5.5	Avoid code comments if they are not clear and useful	0	5	No		GS8, GS17, GS19, GS34, GS35
G5.6	Ensure proper usage of techniques for testing and exception handling	2	5	Yes	S47, S77	GS21, GS23, GS31, GS35, GS51
G5.7	Ensure correctness of assertions in test cases	1	2	Yes	S77	GS23, GS31
G6	**Visualize changes**					
G6.1	Provide a visualization of the change	9	1	No	S11, S13, S31, S42, S49, S52, S55, S66, S67	GS57
G6.2	Provide a visualization of the change regarding its impact on the code base	3	0	No	S5, S13, S40	
G6.3	Allow tracability and easy navigation between artifacts	2	0	No	S42, S80	

(*continued on next page*)

Table 3 (*continued*).

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

Table 4
Categories of code review guidelines used for source and test artifact review (WL = white literature, GL = gray literature).

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

4.1. Macro analysis

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

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

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

Fig. 2 presents the distribution of the included white and gray literature publications per year. As can be seen, the first source that was sampled was published in 1995, implying that this has been an area of research for more than 20 years. We also note that this type of research saw an increase in interest around 2013. We also note an increasing trend in the number of published papers every year, indicating that it is still an area of interest. Notably is that the number of gray literature references is also increasing over time, implying that there is also an industrial interest in guidelines of this type. This conclusion is based on the fact that many of the gray literature sources were industrial blogs and from tool vendors' own documentation rather than academic sources.

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

Furthermore, it must be considered that IC1 for literature required the explicit mention of code review for a source to be included in our final pool. This means that sources about the traditional practice of code inspection, that did not mention (modern) code review, are not considered in Fig. 2.

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

The literature review resulted in 33 guidelines, categorized into nine guideline categories applicable to GUI-based testing artifacts. Out of the nine categories, only seven were supported by white literature, implying that there are practices in the industry not conceived by

Table 5
Overview of code review guidelines and their mapping to GUI testing.

ID	Guideline	Mapping to GUI-based test artifacts
G1	**Perform automated checks**	
G1.1	Perform automated checks on the change	Apply static code analysis tools to identify unsuitable or incorrect patterns in the code
G1.2	Perform automated checks for code style	Apply static code analysis tools to review and format code according to a style guide
G2	**Use checklists**	
G2.1	Provide/use a checklist	Provide/use a checklist that is tailored to the company or team that cover aspects related to GUI element localization, test oracles or synchronization
G3	**Provide context information**	
G3.1	Provide additional context information	Provide contextual information about the code change as a comment during code review or as a comment or annotation to the artifact
G3.2	Provide rationale for the change	Provide contextual information about the motivation why the changes are valid for the purpose of the test
G3.3	Provide context information about the impact of the change	Provide contextual information as a comment during the code review about how the change will impact other reusable test cases or components that are dependant on the change, e.g., how many test cases depend on a changed functionality
G3.4	Provide information about the design and architecture of code affected by the change	Provide contextual information about the logical and chronological behavior of the GUI tests
G3.5	Provide links to related resources and documentation	Provide list of references (links) as comment during code review, e.g., about a new introduced testing technique
G3.6	Provide information about the dependencies between test and production code	Provide a reference between test cases and the corresponding production code
G3.7	Provide context information about the history of changes	Provide a list of references (links) as a comment during code review to previous code reviews or merge requests if an artifact has been changed many times
G3.8	Provide information about test edge cases	Provide contextual information about edge cases as comment during the code review to inform the reviewer about critical paths of the test case
G3.9	Provide a prioritization of the files/classes of the change	Provide prioritization of test cases as a comment during code review if multiple test cases are part of the code review and review time is limited
G4	**Utilize metrics**	
G4.1	Measure and monitor code metrics	Measure and monitor code metrics on test cases like the complexity, coverage, or run-time
G4.2	Provide metrics about execution time (for efficiency)	Measure execution time of test cases
G4.3	Provide test coverage metrics (for effectiveness)	Measure feature-, scenario- or GUI-element coverage
G5	**Ensure readability**	
G5.1	Ensure readability of the change	Ensure readability by following code styles, norms, and conventions
G5.2	Provide comments	Comment and annotate GUI-based test artifacts to explain difficult aspect and give the reviewer insights into the test creator's reasoning behind the tests
G5.3	Follow coding style and naming conventions	Script-based GUI tests should follow a set coding style, norms, and naming convention
G5.4	Follow coding style and naming practices in test writing	Script-based GUI tests should follow a set coding style, norms, and naming convention
G5.5	Avoid code comments if they are not clear and useful	Ensure comments and annotations are helpful to the reviewer and provide additional information that cannot be easily derived from the test artifact itself
G5.6	Ensure proper usage of techniques for testing and exception handling	Check for improper use of testing techniques, such as incorrect usage of mocks, testing on the wrong level of abstraction or wrong variable initialization
G5.7	Ensure correctness of assertions in test cases	Check that assertions are following established test patterns
G6	**Visualize changes**	
G6.1	Provide a visualization of the change	Provide a graph representation of visited GUI states of the SUT
G6.2	Provide a visualization of the change regarding its impact on the code base	Presenting screenshots, or screen recordings, of the GUI states that are covered by the test
G6.3	Allow tracability and easy navigation between artifacts	Provide references to dependent tests and libraries and allow following the references as links if possible
G7	**Reduce complexity**	
G7.1	Keep size of a change as low as possible	Apply patterns and to minimize the use of branching scenarios to keep scripts as short and focused as possible
G7.2	Keep complexity of a change as low as possible	Apply patterns and to minimize the use of branching scenarios to keep scripts as short and focused as possible
G7.3	Avoid unrelated and unstructured changes	Avoid mixing code changes with unrelated changes that do not fit into the scope of the artifact change for review, e.g., code style changes
G8	**Check conformity with the requirements**	
G8.1	Ensure conformance with the requirements	Review the changed test scenario in terms of conformance with the requirements

(*continued on next page*)

Table 5 (*continued*).

ID	Guideline	Mapping to GUI-based test artifacts
G9	**Follow design principles and patterns**	
G9.1	Apply established design principles and patterns	Apply established design principles and patterns for source code and explicit ones for GUI testing
G9.2	Don't repeat yourself (DRY)	Extract common functionality into reusable components and reuse them throughout the test. Apply design patterns such as page-object pattern
G9.3	Avoid hardcoded values	Avoid hardcoded values, such as hardcoded identifiers
G9.4	SOLID principle	Avoid unrelated changes to confirm with the single responsibility principle

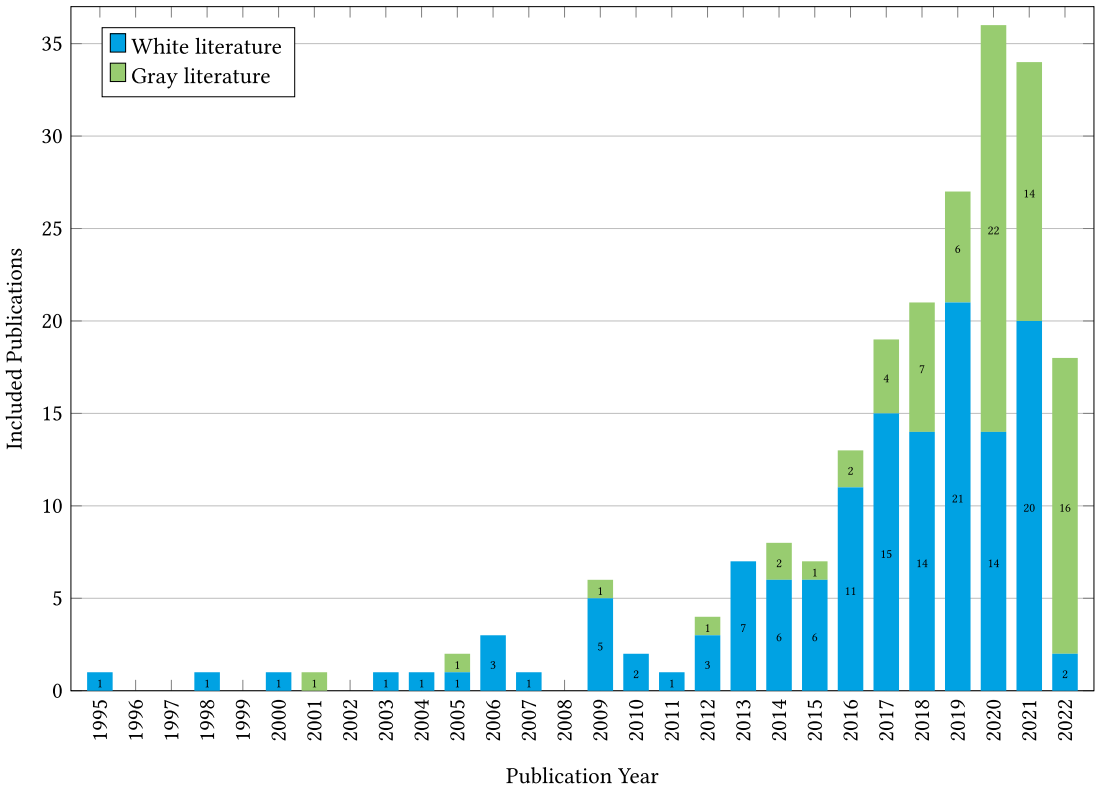


Fig. 2. White and gray literature publications per year.

the academia. The two categories of guidelines that were introduced through gray literature are *check conformity with the requirements* and *follow design principles and patterns*. Furthermore, the gray literature represents a super-set of the sample, including all guidelines from white literature.

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

References for all sources where guidelines were taken from are documented in Table 6 for white literature and Table 7 for gray literature.

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

Synthesis of the two literature sets, i.e., white and gray literature, acquired from our SLR, resulted in 33 guidelines in nine categories for review of artifacts. These guidelines are presented in Table 3, and a categories-only summary is shown in Table 4. In Table 3, we report which guideline was defined as specific to test artifacts. As a notable result, we find that only 10 sources of those we analyzed specifically described guidelines for the review of test artifacts. Further, the table presents how many occurrences of each guideline appeared in white and gray literature with references to the sources.

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

Notably, we see that the most mentioned guideline category (Table 4) from both literature sets is to provide context information to a review request. This result can be explained by the cost-value of adding such information, where cheap practices, e.g., linking a review request to the implemented requirements, can significantly improve the reviewer's understanding of the review. A similar explanation of cost-value can be given for the second most mentioned item, reducing complexity. Whilst this is not necessarily a cheap practice, since

it may be complicated to achieve, its impact in terms of value is high, since complexity affects the readability, understandability and maintainability of the artifact. Notably, the guideline category with the most significant variance in reference support between white and gray literature is the one to ensure readability. This result is surprising since this guideline is an important aspect of dealing with complexity. From the gray literature only guideline categories, the one regarding using established software engineering design principles has a high reference support. As described earlier, white literature often provides guidelines as a byproduct of other research and which could explain why established design principles or patterns are mentioned as a contribution.

Answer to RQ1: All identified code review guidelines used for source and test artifact review are presented in Table 3, and a categories-only summary is shown in Table 4. We consolidated our findings of 33 guidelines in nine categories, where seven categories emerged from white literature and two from gray literature. Nonetheless, the main purpose of these guidelines is to guide the contributor of code changes and the reviewer to improve the effectiveness and efficiency of the code review process.

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

In the continuation of this section, we present a detailed description of each guideline. Table 5 presents the mapping of the identified guidelines to GUI-based test artifacts. We structure this section according to the categories presented in Table 4 to answer RQ2. For generalizabililty, the level of abstraction of each guideline has been kept on a higher level to make them applicable to different GUI testing techniques and tools. The presented guidelines are inherently generalizable to all GUI-based testing artifacts, since they are provided as a generalization of general-purpose code review guidelines originally specified for source or generic test code. Code reviews for specific aspects of GUI-based testing are therefore not to be found in the results discussed in this section. Each guideline category is presented according to a structured textual format that explains (1) the purpose of the guideline category, (2) guidelines of the category and (3) examples of application for GUI-based tests. In the explanation of each guideline, a reference to the guidelines in Table 5 is provided by adding the corresponding ID in parentheses. Each guideline has also been ranked in terms of if it is a guideline that is *suggested*, *recommended*, or *strongly recommended* to be followed, based on reference materials. *Suggested* guidelines are in this context viewed as beneficial but not necessary, *recommended* guidelines are rated as beneficial by multiple sources and could have adverse effects if not applied and *strongly recommended* are suggested by multiple sources to have significant benefits or tangible adverse effects if not applied.

G1 Perform automated checks

Purpose and description: The purpose of performing automated checks is to improve the efficiency of the review process by reducing manual effort. This allows the reviewer to focus on the understandability and maintainability of the code changes instead of getting distracted by revising low-level code issues [2]. These automated checks are either done by the submitter before the code review or handled by a Continuous Integration (CI) infrastructure when code changes are submitted for a code review.

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

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

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

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

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

G2 Use checklists

Purpose and description: The purpose of a checklist is to guide the reviewer during the code review, outlining either specific activities to be carried out during the review or artifacts to be reviewed. Guidelines thereby provide a powerful, context-dependent, tool for reviews and are therefore *strongly recommended* for most types of artifact reviews, including GUI-based tests. The exception to this rule is when the reviewing activity has been so thoroughly embedded in the organization that its need becomes superfluous. Especially junior developers, without code review experience, benefit from checklists [49]. However, the guidelines shall take the domain, company, team culture and other environmental aspects into account [50].

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

However, based on our included literature, we could not identify any specific items that a checklist should always contain for source- and test code artifacts. Neither could we identify any example checklists for non-GUI or GUI-based testing, although checklists for source code are available. One such example is proposed by GitLab, which encourages checking for performance, security, reliability, observability, and maintainability risks [51].

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

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

G3 Provide context information

Purpose and description: The purpose of contextual information is to aid the reviewer in understanding the changes made to the project, be it requirements, code or tests. This type of information is argued to reduce the reviewer's cognitive load and thereby improve the effectiveness—Identification of errors or improvements—of the code review [52]. Additionally, providing such information has positive effects on the efficiency of the review, since it mitigates the need for the reviewer to spend time gathering this information for themselves [52].

Guidelines: The test developer is responsible for providing sufficient contextual information for the review to understand the change G3.1. This information belongs to one of two categories, either technical knowledge—Information about the test code that has been developed or its artifacts—or domain knowledge—Information about requirements, the application or its usage.

From the literature, the main type of contextual information that shall be provided is a rationale for the change (G3.2)—A motivation for the change, its cause and effect. This information aims to address the knowledge gap that is presented when a change occurs, since *what* has changed is easily seen, but *why* the change occurred may be less evident. Hence, the rationale serves to provide an answer to *why* a change has been made, which for GUI-level tests generally relates to domain aspects, such as changed requirements. These requirements may however be of both a functional, e.g., features or functions of the SUT, or non-functional nature, e.g., quality attributes of the SUT such as its performance or security or, for GUI-level tests, also the SUT's visual appearance or behavior—Changed bitmaps or modes of interaction.

The test developer also must provide information about the impact of the changes (G3.3). For instance, if the changes have impacted the test coverage, test design or the architectural design of the testware (G3.4). The latter aspect is seldom discussed for GUI-level tests, but, as discussed by Sutherland et al. in the context of software, a lack of such discussion in reviews can lead to technical or architectural debt in the codebase [53].

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

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

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

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

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

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

G4 Utilize metrics

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

Guidelines: Measuring and monitoring different types code metrics can be performed during the development or review process to give inputs to the artifact reviewer (G4.1). These metric types can be roughly clustered in terms of change impact on artifact complexity, efficiency, and effectiveness. Metrics regarding the impact of a change on the complexity of the reviewed artifact can be acquired from static code analysis tools—Static code analysis is performed as automated checks—and are valuable to provide insights regarding the readability, understandability and maintainability of the modified artifact. Examples of possible complexity metrics include cyclomatic complexity, the number of files in a solution, the size of single files or functions, sum of added and removed lines of code (i.e., code churn) and dispersion of modified lines across files. In the sampled literature, these metrics were also used as predictors of introducing defect-prone code to the code base [54].

One way to monitor the efficiency of changes, be it source or test artifacts, is to monitor the execution time (G4.2) of artifacts—By monitoring we include execution time before and after change. For test artifacts, the execution time can be measured on different levels of granularity. On a high level, the execution time of the overall test suite provides an overview, whilst lower-level, more detailed measurements of time spent in functions, test cases, or single GUI screens, can help identify bottlenecks. For source code, poor performance can be an indicator of poor code quality [26]. In the same way, poorly performing test cases should be investigated by the reviewer for test quality issues.

Measuring coverage metrics such as code, path, or GUI coverage allows insights into the effectiveness of test artifacts (G4.3). Increased coverage is correlated with increased fault-finding behavior and is therefore one of the main attributes the reviewer of a test artifact must investigate. This entails acquiring an understanding of whether the test artifacts cover all, or at least all necessary, paths of the production code [36].

Example(s) of application: The complexity of GUI-based tests shall also monitored (G4.1). Depending on the frameworks used to develop the tests, this monitoring can be done with the same tools as for source code. For instance, tools like Selenium allow the test developer to write test cases in source cod, or use test libraries such as XUnit [16,55].

Such tests can be analyzed using the same static-code analyzers as the source code and provide similarly effective results. For tools with custom scripting languages or IDEs, such as Graphwalker [56] or Scout [57], conventional solutions to measure complexity are not applicable. Instead, the reviewer needs to manually evaluate the perceived complexity, in particular taking into account the perceived readability and understandability of the tests. These attributes are important since they affect the maintainability of the tests, which, in turn, affects the tests' longevity.

The execution costs of automated GUI-based tests are more expensive compared to low-level unit tests since the tests are more computationally heavy. In fact, the execution time of a GUI-based test is generally several orders of magnitude more costly than, for instance, a unit test [58]. Therefore, information about the execution time and the required resources of these test cases are important to estimate the scaleability of the test approach (G4.2). In particular, when these tests are executed as part of a continuous integration environment, where allotted time for test execution is limited and test prioritization or selection is therefore required.

Furthermore, in the same way as for poorly performing source code, a GUI test case with a long execution time, compared to other similar GUI test cases, could indicate inefficiencies in its design. Inefficiency could include an unnecessarily long or complex test scenario, unsuitably long synchronization checkpoints, or inefficient GUI element identification. Hence, quality attributes of the test artifact that the reviewer must investigate.

Measuring test effectiveness in terms of coverage is also an important aspect of GUI-based tests (G4.3). However, due to the level of abstraction these tests operate, low-level coverage metrics, like code coverage, can be difficult to calculate. Solutions include instrumentation of the SUT or using third-party measurement tools, e.g., JaCoCo [59] or Cobertura [60]. More commonly used coverage metrics are instead feature, scenario, or GUI element coverage. Whilst these metrics do not provide insights of the same granularity, they are useful to determine the effects of a change to the tests.

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

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

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

G5 Ensure readability

Purpose and description: The purpose of ensuring readability is to improve the ease of understanding, and thereby maintaining, an artifact over time. More formally, Buse and Weimer [61] defines readability "as a human judgment of how easy a text is to understand". As such, the readability of an artifact has a direct impact on the reviewing process itself, where an artifact with low readability is perceived as more difficult to review than an artifact with high readability. Code reviews shall strive to identify the level of readability of an artifact and suggest improvements. For example, at Google, code reviews were introduced to ensure code readability and maintainability, where readability is supported by the use of a consistent code style within the code base Sadowski et al. [2]. Therefore, we rank this guideline as *strongly recommended*.

Guidelines: One way to improve the readability of code is to follow common coding styles, norms and naming conventions (G5.1). A team should enforce a coding style within the team to prevent low-level discussions about individual styling preferences during a code review (G5.2 and G5.3). A style guide is a set of conventions for how code shall be structured in a project, including how indentations shall be done, variable naming convention (i.e., use camel-case), restrictions on line length, or the ban of specific operators. For instance, Google has style guides for all major programming languages that developers must follow for all application developments [62]. Code consistency also lowers cognitive complexity for the reviewer, making it easier to focus on the artifacts conformance to the intended behavior. Consistent code can also more easily be automatically checked through, for instance, static code analysis or other automated checks.

Another way to improve readability is to annotate important or more difficult parts of the artifact. Annotations can take the form of comments or other supplementary materials (G5.4). An example of the value of annotation is provided by An et al. [63], where crash-prone code with a low ratio of code comments was found more difficult to review. However, annotations shall only be added when they are useful (G5.5), i.e., they must be helpful to the reviewer by providing additional information that is difficult to derive from the code itself. Mindless addition of annotations can otherwise lead to "clutter" in the artifact, which decreases its readability. The annotations must also be fit-for-purpose, since they can otherwise add ambiguities that confuse the reviewer, e.g., if supplementary materials are added that are not consistent with the reviewed artifact itself.

For test artifacts it is also important to consider the fit-for-purpose of the artifact, and to use the most suitable technique or pattern for the test purpose (G5.6). Similarly to coding conventions, using familiar techniques or patterns makes it easier for the reviewer to comprehend the changes. Furthermore, improper use of testing techniques, such as incorrect usage of mocks, testing on the wrong level of abstraction or wrong variable initialization, may lead to faulty test results [36]. For instance, if a technique is on the wrong level of abstraction for the test purpose, it may not be able to identify a fault. In addition, test assertions should also follow suitable patterns for the test purpose (G5.7). It is therefore important that the reviewer also evaluates the adequacy of the artifact for the purpose, and provide recommendations if this is not aligned.

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

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

A unique aspect of test code that reviewers must investigate is the test assertions (G5.6 and G5.7). These need to be clear, from a readability perspective, so that the reviewer can determine their correctness and fit-for-purpose. The reviewer must also evaluate that the assertions are placed in suitable sections of the test scenario [36]. Additionally, the reviewer must evaluate the assertions are placed following established test patterns.

G6 Visualize changes

Purpose and description: The purpose of visualizing code changes is to ease the understanding of the change and its impact. Visualization also allows for easier navigation between related artifacts and their dependencies. A study by Baum and Schneider [64] identified that reviewers often miss features in IDEs (Integrated Development Environments) for code review of non-trivial changes. They conclude that tool support would mitigate this challenge and improve both the effectiveness and efficiency of code reviews. Furthermore, for review of test artifacts, many code review tools lack test-specific information and only present changes as line-by-line comparisons. This forces the reviewer to open the artifacts in their local environments, e.g., IDEs, to be able to navigate through the dependencies and acquire a full picture of the change [36]. Using tools that have stronger visualization solutions helps to mitigate these challenges.

Guidelines: First, a visualization shall be provided of the files affected by the change and the dependencies between these files and their changes (G6.1). These dependencies between files help a reviewer to understand which code change(s) may have the biggest side effects due to its depending components. Thus, providing insights into what files shall be prioritized in the review and the order they should be reviewed in. Common to both source and test code, changes in a file whose functionality is reused in multiple instances can potentially cause more side effects. This visualization should also indicate the impact of the change on the rest of the system, e.g., a Change Impact Analysis [65]. Hence, although visualization of changed technical artifacts is a minimum, additionally affected artifacts, e.g. source code, test code, requirements and libraries, shall also be noted.

For more assistance, semantic changes and relationships shall also be presented in addition to the textual line-by-line changes [65] to determine the impact of a change (G6.2). Line-by-line differences are usually based on the underlying GIT or Unix tooling, and although they allow the reviewer to pinpoint the exact changes, they do not provide any insight into the behavior change caused by such changes. For instance, renaming a variable does not affect the behavior of a program, and therefore it does not change the semantic relationship. However, changing the algorithm that uses said variable may result in completely different software behaviors. Behaviors that may be non-compliant and not visible in the code.

Visualization helps the reviewer to more easily navigate between different artifacts, but it still requires the reviewer to open these artifacts, often in different tools. Using review tools that more easily help the reviewer navigate, and help ensure traceability, between different artifacts are therefore helpful (G6.3). For instance, mitigating the need for a reviewer to switch between source and test code in different windows during a review. As pointed out by Spadini et al. [36], artifacts are usually coupled to multiple other artifacts and contextual switches between sources thereby creating additional cognitive load.

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

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

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

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

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

G7 Reduce complexity

Purpose and description: The purpose of reducing artifact complexity is multi fold. First, reduced complexity is associated with increased understandability of the artifact. Second, complexity has a well-known impact on the defect rate of a software, where more complex components are more likely to have defects [66].

Guidelines: The main approach to reduce complexity is to keep code changes for a review as small as possible (G7.1). Small code changes are preferred by reviewer and they can provide more useful feedback compared to big code changes [67]. In addition, complexity should be avoided (G7.2) by having a narrow and well defined scope [67]. Although a development practice, these practices reduces the effort during code review, improving its efficiency and effectiveness.

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

Furthermore, included non-code artifacts like configuration files, test or build results in less useful feedback [3]. Pure style changes—Style changes include changes to naming convention or arrangement of test cases—can be integrated into the code base with a less exhaustive review, since these changes should not introduce any changes in behavior of the code. Mixing such style changes with code changes makes it harder for the reviewer to identify the actual change in behavior.

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

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

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

G8 Check conformity with the requirements

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

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

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

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

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

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

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

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

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

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

G9 Follow design principles and patterns

Purpose and description: The purpose of design principles and design patterns are to provide consistency among various types of artifacts. Another purpose is to prevent the degradation of the artifacts. For instance, software architectures and libraries may degrade over time as the code is rewritten, maintained or otherwise updated. Principles for software design can prevent these issues and thereby prevent source code fragility and improve reusability, maintainability and scalability of the software [68]. Design patterns are thereby blueprints for building reusable solutions for common problems. These patterns are well proven and have evolved over time [69].

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

Guidelines: Common design principles, including Don't Repeat Yourself (DRY), SOLID for object-oriented design, and avoiding hardcoded values, should be followed if the context allows it (G9.1). Applying the DRY principle favors the (re-)use of components instead of a duplication of code and effort (G9.2). Hardcoded values, such as hardcoded identifiers, can hinder the deployment of the application in different environments [70] and thus should be avoided (G9.3). The SOLID principle consists of single-responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion. Applying SOLID principles to object-oriented design helps to improve the maintainability of the codebase [68]. Although these are patterns that the developer must consider, it is part of the reviewers' responsibility to verify that they are followed. These mentioned principles and patterns are only a subset of what is available, and we restrict ourselves to these examples since the patterns themselves are out of the scope of this work. Additionally, their application are context-dependent, where some principles may be used in some companies but not in others. Hence, it is important that both developers and reviewers are aware of context-applicable patterns and if/how they change over time.

Example(s) of application: For GUI-based testing, principles like the usage of dynamic (i.e., wait for a specific visual state) synchronization checks to avoid unnecessary test code maintenance or automatically capture screenshots of failed actions or state transitions when a test fails for failure replication can be applied (G9.1) [21]. If tests are script-based, principles for source code like DRY and avoiding hard-coded identifiers can be applied. For example, common functionality such as logging into an application is common and should be extracted and reused throughout the test suite rather than to be repeated (G9.2). Further, hardcoded values, such as widget identifier, should also be avoided (G9.3). The reason is that if this functionality or values changes, it will have to be updated in multiple scripts. Reviewers can aid the script authors in identifying reusable functionality that can be extracted, or point the authors to reusable components when such are missed during development.

There are also specific patterns for testing. One such pattern, explicit for GUI testing, is the page-object pattern (G9.2). The pattern ensures that functionality is tested in isolation, considering one page view at the time. This practice helps to reduce the coupling between test cases and the SUT [71]. But the identified design patterns are not applicable to GUI-based tests that are on a higher level of abstraction, like image recognition-based GUI tests.

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

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

Answer to RQ2: We present a mapping of source and test artifact review guidelines to GUI-based test artifacts by providing examples on its application. A summary of this mapping is shown in Table 5. In addition, we explained the purpose of each guideline category and ranked each guideline as *suggested*, *recommended*, or *strongly recommended* based on reference materials.

5. Discussion

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

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

Thus, although the list of nine categories including 33 guidelines we present is a good starting point, we do not perceive this list to be comprehensive. This assumption is motivated by the study's focus on artifacts, not covering additional practices of the review process associated with, for instance, reviewer selection or effort allocation. The assumption is also motivated by the analysis of gray literature, where we found multiple practices that have not been covered in white literature. Thus, implying that there may be practices used in industry that academia is not aware of. In addition, we have only mentioned guidelines that could be data triangulated—Data triangulation is the practice of using different sources of information to increase the validity of a study's results [72]—with at least three sources. Hence, additional guidelines were identified, but due to a lack of support for their validity, they were not incorporated into our results. An example of such a guideline was to "provide information to reproduce identified faults" [73]. This guideline could possibly belong to a category of guidelines regarding how to provide reviewer feedback. However, due to the lack of additional such guidelines in our sample, no such category was added.

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

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

Empirical evaluation of the guidelines is important since, in a related study, guidelines for the development of GUI-based tests were suggested but when evaluated in practice did not provide a successful result [21]. Although the study is limited in scope, it suggests that best development practices for source code are not necessarily transferable to GUI test code. The reasons stated in the related work were increased cognitive load as well as lack of applicability of some practices. Due to the level of abstraction of our suggested guidelines, focusing on *what* to look at in GUI-based test reviews and not *how* to do so, we do not perceive the same concerns. However, there is still a possibility that this unknown factor that set GUI-based tests apart from source code could play a role in the applicability of the proposed guidelines. Thus, once more, stressing the need for future empirical evaluation of the guidelines. Despite this potential issue, we do not perceive that it takes away anything from the contribution of this work. The reason is, as stated, the current omission of any general guidelines for reviews of GUI-based tests in the academic body of knowledge.

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

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

Table 6
White literature sources where guidelines were taken.

ID	Title	Authors	Year
S1	A community of practice around peer review for long-term research software sustainability	Kalyan, Akshay, et al.	2019
S2	A reflective practice of automated and manual code reviews for a studio project	Oh, Jun-Suk, and Ho-Jin Choi	2005
S3	A secure code review retrospective	Buttner, Andrew, et al.	2020
S4	Accept or not? An empirical study on analyzing the factors that affect the outcomes of modern code review?	Wang, Dandan, et al.	2021
S5	Aiding code change understanding with semantic change impact analysis	Hanam, Quinn, et al.	2019
S6	An effective source code review process for embedded software	Hirayama, Masayuki, et al.	2006
S7	An empirical study of link sharing in review comments	Jiang, Jing, Jin Cao, and Li Zhang	2019
S8	Analyzing involvements of reviewers through mining a code review repository	Liang, Junwei, and Osamu Mizuno	2011
S9	Assisting the code review process using simple pattern recognition	Farchi, Eitan, and Bradley R. Harrington	2006
S10	Automatic patch linkage detection in code review using textual content and file location features	Wang, Dong, et al.	2021
S11	Can formal methods improve the efficiency of code reviews?	Hentschel, Martin, et al.	2016
S12	Can peer code reviews be exploited for later information needs?	Sutherland, Andrew, and Gina Venolia	2009
S13	ChangeViz: Enhancing the GitHub Pull request interface with method call information	Gasparini, Lorenzo, et al.	2021
S14	Characteristics of useful code reviews: An empirical study at microsoft	Bosu, Amiangshu, et al.	2015
S15	Code review analysis of software system using machine learning techniques	Lal, Harsh, and Gaurav Pahwa	2017
S16	Code review and cooperative pair programming best practice	Fu, Qiang, et al.	2017
S17	Code review knowledge perception: Fusing multi-features for salient-class location	Huang, Yuan, et al.	2020
S18	Code review quality: How developers see it	Kononenko, Oleksii, et al.	2016
S19	Code reviewing in the trenches: Challenges and Best practices	MacLeod, Laura, et al.	2018
S20	Code reviews with divergent review scores: An empirical study of the openStack and Qt communities	Hirao, Toshiki, et al.	2022
S21	Communicative intention in Code review questions	Ebert, Felipe, et al.	2018
S22	Confusion in code reviews: Reasons, Impacts, and Coping strategies	Ebert, Felipe, et al.	2019
S23	CoRA: Decomposing and describing tangled code changes for reviewer	Wang, Min, et al.	2019
S24	Decomposing composite changes for code review and regression test selection in evolving software	Guo, Bo, et al.	2019
S25	Effects of adopting code review Bots on Pull Requests to OSS projects	Wessel, Mairieli, et al.	2020
S26	Evaluating how static analysis tools can reduce code review effort	Singh, Devarshi, et al.	2017
S27	Expectations, outcomes, and challenges of modern code review	Bacchelli, Alberto, and Christian Bird	2013
S28	Fix-it: An extensible code auto-fix component in review Bot	Balachandran, Vipin	2013
S29	Generating Code review documentation for auto-generated mission-critical software	Denney, Ewen, and Bernd Fischer	2009
S30	Impact of coding style checker on code review - A case study on the openstack projects	Ueda, Yuki, et al.	2018
S31	Interactive code review for systematic changes	Zhang, Tianyi, et al.	2015
S32	Investigating code reading techniques for novice inspectors: an industrial case study	Rong, Guoping, et al.	2014
S33	Investigating code review quality: Do people and participation matter?	Kononenko, Oleksii, et al.	2015
S34	Java code reviewer for verifying object-oriented design in class diagrams	Jinto, Kanit, and Yachai Limpiyakorn	2010
S35	Lessons Learned from building and deploying a code review analytics platform	Bird, Christian, et al.	2015
S36	LightSys: lightweight and efficient ci system for improving integration speed of software	Lim, Geunsik, et al.	2021
S37	Mining peer code review system for computing effort and contribution metrics for patch reviewers	Mishra, Rahul, and Ashish Sureka	2014
S38	Mining source code improvement patterns from similar code review works	Ueda, Yuki, et al.	2019
S39	Modern code review: A case study at Google	Sadowski, Caitlin, et al.	2018
S40	Multi-Perspective visualization to assist code change review	Wang, Chen, et al.	2017
S41	Natural language insights from code reviews that missed a vulnerability	Munaiah, Nuthan, et al.	2017
S42	On the need for a new generation of code review tools	Baum, Tobias, and Kurt Schneider	2016
S43	On the understanding of programs with continuous code reviews	Bernhart, Mario, and Thomas Grechenig	2013
S44	Patch review processes in open source software development communities: A comparative case study	Asundi, Jai, and Rajiv Jayant	2007
S45	Guiding Developers to Make Informative Commenting Decisions in Source Code	Huang, Yuan, et al.	2018
S46	Practical aspects on the assessment of a review process	Kiiskila, Janne	1998
S47	Process Aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft	Bosu, Amiangshu, et al.	2017
S48	RAID: Tool support for refactoring-aware code reviews	Brito, Rodrigo, and Marco Tulio Valente	2021

(*continued on next page*)

Table 6 (*continued*).

ID	Title	Authors	Year
S49	Rebasing in code review considered harmful: A large-scale empirical investigation	Paixao, Matheus, and Paulo Henrique Maia	2019
S50	Refactoring practices in the context of modern code review: An industrial case study at xerox	AlOmar, Eman Abdullah, et al.	2021
S51	Review dynamics and their impact on software quality	Thongtanunam, Patanamon, and Ahmed E. Hassan	2021
S52	RSTrace+: Reviewer suggestion using software artifact traceability graphs	Sülün, Emre, et al.	2020
S53	Salient-class location: help developers understand code change in code review	Huang, Yuan, et al.	2018
S54	SCRUB: a tool for code reviews	Holzmann, Gerald	2010
S55	Semantics-assisted code review: An efficient tool chain and a user study	Menarini, Massimiliano, et al.	2017
S56	Share, But be aware: Security smells in Python Gists	Rahman, Md Rayhanur, et al.	2019
S57	Static security analysis based on input-related software faults	Nagy, Csaba, and Spiros Mancoridis	2009
S58	Supporting automatic code review via design	He, Jiantao, et al.	2013
S59	Test-driven code review: An empirical study	Spadini, Davide, et al.	2019
S60	Testing web enabled simulation at scale using metamorphic testing	Ahlgren, John, et al.	2021
S61	The effect of checklist in code review for inexperienced students: An empirical study	Rong, Guoping, et al.	2012
S62	The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects	McIntosh, Shane, et al.	2014
S63	The Impact of code review on architectural changes	Paixao, Matheus, et al.	2021
S64	The influence of non-technical factors on code review	Baysal, Olga, et al.	2013
S65	The Symbolic Execution Debugger (SED): a platform for interactive symbolic execution, debugging, verification and more	Hentschel, Martin, et al.	2019
S66	Tool support for code change inspection with deep learning in evolving software	Ayinala, Krishna Teja, et al.	2020
S67	Tool support for managing clone refactorings to facilitate code review in evolving software	Chen, Zhiyuan, et al.	2017
S68	Towards a taxonomy of code review smells	Dogan, Emre, and Eray Tüzün	2021
S69	Understanding shared links and their intentions to meet information needs in modern code review	Wang, Dong, et al.	2021
S70	Using metrics to track code review performance	Izquierdo-Cortazar, Daniel, et al.	2017
S71	Using paragraph vectors to improve our existing code review assisting tool-CRUSO	Kapur, Ritu, et al.	2021
S72	VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits	Perl, Henning, et al.	2015
S73	What are they talking about? analyzing code reviews in Pull-based development model	Li, Zhi-Xing, et al.	2017
S74	What design topics do developers discuss?	Viviani, Giovanni, et al.	2018
S75	What makes a code change easier to review: an empirical investigation on code change reviewability	Ram, Achyudh, et al.	2018
S76	What types of defects are really discovered in code reviews?	Mäntylä, Mika V., and Casper Lassenius	2009
S77	When testing meets code review: Why and how developers review tests	Spadini, Davide, et al.	2018
S78	Why did this reviewed code crash? An empirical study of Mozilla Firefox	An, Le, et al.	2018
S79	Why do developers reject refactorings in open-source projects?	Pantiuchina, Jevgenija, et al.	2021
S80	Why security defects Go unnoticed during code reviews? A case-control study of the Chromium OS project	Paul, Rajshakhar, et al.	2021

5.1. Threats to validity

The threats to the validity of this work have been divided into four parts; internal validity, external validity, construct validity and reliability, following the guidelines of Runeson and Höst [74].

Internal validity: Internal validity concerns the ability of the study design to conclude a correct relationship between factors, where multiple factors may have confounding, but unknown, effects on the investigated factor. For this study, one internal validity threat is that it is possible that literature sources with additional guidelines may have been overlooked. Although this could affect the guideline categories presented in this work, it is unlikely due to the high level of abstraction they are presented at. Another threat is that the mapping between the source- and test code guidelines are invalid because we have failed to take factors of the original guideline into account. Once more this is unlikely, since the mapping was done through examples that show the guidelines use case. This mapping approach has been used previously for GUI-based testing research, for instance by Alégroth and Gonzalez-Huerta [75].

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

The objective of RQ2 was to provide a mapping from unspecialized code review guidelines to guidelines for review of GUI-based test artifacts. In addition of the possibility of overlooking sources discussed previously, a threat to the validity of the result is the possible existence of facets of GUI-based test artifacts that should be code-reviewed, but that have no correspondence with normal production code. GUI-based testing artifacts include in fact several elements that strongly differ from regular code artifacts, e.g. models or screen captures of the

Table 7
Gray literature sources where guidelines were taken.

ID	Title	URL	Year
GS1	9 Best practices for code review you really need	perforce.com	2019
GS2	Best practices for peer code review - SmartBear	smartbear.com	2022
GS3	How to do a code review | eng-practices - Google	google.github.io	2020
GS4	5 code review best practices - Work Life by Atlassian	atlassian.com	2022
GS5	Code review best practices - Palantir Blog	blog.palantir.com	2018
GS6	8 Proven code review best practices for developers - Snyk	snyk.io	2022
GS7	How to make good code reviews better - Stack Overflow Blog	stackoverflow.blog	2019
GS8	16 Tech leaders share smart best practices for reviewing code	forbes.com	2020
GS9	Reviewing code - Best practices and techniques for code review	codegrip.tech	2021
GS10	How to review someone Else's code: Tips and best practices	codecademy.com	2021
GS11	Code review good practices: guide for beginners	medium.com	2021
GS12	5 code review best practices. Make others like your code review	tsh.io	2020
GS13	Code review guidelines - GitLab documentation	docs.gitlab.com	2022
GS14	Proven code review best practices from microsoft	michaelagreiler.com	2019
GS15	Code review best practices - DeepSource	deepsource.io	2019
GS16	13 Code review standards inspired by Google	betterprogramming.pub	2020
GS17	What is code review? - Guidelines and best practices	blog.ndepend.com	2021
GS18	Secure code review: How secure is your code? - DataPrivia	dataprivia.com	2022
GS19	Code review - Open practice library	openpracticelibrary.com	2020
GS20	6 code review best practices for a happier codebase and team	educative.io	2022
GS21	Code review best practices - Trisha Gee	trishagee.com	2018
GS22	Best practices for code review and Pull requests - Updivision	updivision.com	2021
GS23	5 Best practices for code review - GeeksforGeeks	geeksforgeeks.org	2022
GS24	How to improve your code review: tips and best practices	belvo.com	2021
GS25	Investigating the effectiveness of peer code review in …	jserd.springeropen.com	2018
GS26	Better code, better applications - every time	walkingtree.tech	2022
GS27	How we do it: peer code review - DataMiner Dojo	community.dataminer.services	2022
GS28	Code review guidelines for data science teams	tdhopper.com	2021
GS29	Gerrit code review product overview	gerrit-review.googlesource.com	2022
GS30	Code review: Best practices - Waverley software	waverleysoftware.com	2019
GS31	Code review checklist - Apex hours	apexhours.com	2021
GS32	The art of code review - Towards Data Sciencehttps	towardsdatascience.com	2021
GS33	Code review best practices - Level up coding	levelup.gitconnected.com	2021
GS34	Best practices code review test automation by Anton Smirnov	itnext.io	2021
GS35	Best practices for code review	dzone.com	2020
GS36	Code review best practices - GitKraken	gitkraken.com	2021
GS37	Best practices for effective code review	leobit.com	2020
GS38	Best practices for effective and efficient Agile code reviews	queue-it.com	2022
GS39	Best practices for peer code review	kessler.de	2009
GS40	Creating simple and effective guidelines for code reviews	newrelic.com	2018
GS41	Best practices for code review - C# Corner	c-sharpcorner.com	2016
GS42	Code review best practices - Programmer Friend	programmerfriend.com	2018
GS43	How rOpenSci uses code review to promote reproducible science	ropensci.org	2017
GS44	Where is the value in package peer review?	ropensci.org	2018
GS45	Recommended C style and coding standards. Pocket reference guide	gnu.org	2005
GS46	ChangeViz materials	doi.org/10.5281/zenodo.5175927	2021
GS47	Ilya Sabanin contributor to Beanstalk guides	guides.beanstalkapp.com	2019
GS48	Atlassian Crucible features	atlassian.com	2022
GS49	CodeFlow	getcodeflow.com	2014
GS50	Intel Open Source Technology Center – Patch Review	blog.ffwll.ch	2020
GS51	Apache Spark	spark.apache.org	2020
GS52	Chromium coding style	dev.chromium.org	2014
GS53	Pep 8: style guide for python code	peps.python.org/pep-0008	2001
GS54	10 faulty behaviors of code review	speakerdeck.com	2020
GS55	How we do code review — app center blog	devblogs.microsoft.com	2020
GS56	Code review best practices by Palantir	medium.com	2020
GS57	Pull request best practices - the pragmatic engineer	blog.pragmaticengineer.com	2020
GS58	The wireshark wiki - Development/SubmittingPatches	wiki.wireshark.org	2020
GS59	Sharma S. How to write a good pull request description – and why it is important URL	freecodecamp.org	2020
GS60	What is code review?	smartbear.com	2020
GS61	Code reviews at google are lightweight and fast	michaelagreiler.com	2020
GS62	The 2019 state of code review: Trends and insights into collaborative software development	static1.smartbear.co	2019
GS63	Firefox code review	wiki.mozilla.org	2016

visual content of the SUT. Guidelines to address these specific objects cannot be deduced by mapping code review guidelines for traditional code review, thereby limiting the comprehensiveness of the answer provided to RQ2 of the present manuscript. A final construct validity threat is related to the selection of a broad search string, including all code and software related guidelines. The selection of such string can have an impact on the final count of the guideline occurrence that was measured as an answer to RQ2, providing measures differing significantly than the figures that would have been obtained including only test-related sources in the final pool.

External validity: External validity concerns the generalizability of the results to other areas or domains. The study's scope is on GUI-based testing in general, regardless of test driver, GUI element localization technique or test case representation. Whilst there is a threat that some of the guidelines are less applicable in some of these permutations of

approaches, due to the high level of the guidelines, such threats are perceived as low.

Construct validity: Construct validity concerns if the studied phenomenon is the right phenomenon to meet the research objective. In this study, the objective was to map guidelines from source- and test code to guidelines for GUI-based tests. The assumption behind this objective is that the guidelines are transferable due to the common attributes between source and test code [75]. However, as discussed by Alegroth et al. [21], not all practices seem to be transferable. Hence, there is a threat to the study's results that the characteristics used for the mapping are not representative, which may lower the applicability of some of the guidelines. Due to the example-based approach to the mapping, we consider this threat to be lower, but we cannot conclude that all guidelines are applicable without empirical validation. Regardless, this threat does not affect the contribution of this work since there is currently a complete lack of guidelines for GUI-based testing reviews in the academic body of knowledge.

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

6. Conclusion

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

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

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

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

In conclusion, this work provides an important stepping stone for review guidelines for GUI-based testing. However, more work is required in the future to address the current needs from the industry, and challenges in this area.

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 dataset is available at the following URL: https://zenodo.org/record/7248201.

Acknowledgments

We would like to acknowledge that this work was supported by the KKS foundation through the S.E.R.T. Research Profile project (research profile grant 2018/010) at Blekinge Institute of Technology. Further, we thank Michael Dorner for valuable discussions.

References

[1] Amy E. Randel, Kimberly S. Jaussi, Functional background identity, diversity, and individual performance in cross-functional teams, Acad. Manag. J. 46 (6) (2003) 763–774.

[2] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, Alberto Bacchelli, Modern code review: A case study at google, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ACM, Gothenburg Sweden, 2018, pp. 181–190, http://dx.doi.org/10.1145/3183519.3183525.

[3] Amiangshu Bosu, Michaela Greiler, Christian Bird, Characteristics of useful code reviews: An empirical study at microsoft, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, Florence, Italy, 2015, pp. 146–156, http://dx.doi.org/10.1109/MSR.2015.21.

[4] Jason Cohen, Modern code review, in: Making Software: What Really Works, and Why We Believe It, 2010, pp. 329–336.

[5] Alberto Bacchelli, Christian Bird, Expectations, outcomes, and challenges of modern code review, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, San Francisco, CA, USA, 2013, pp. 712–721, http://dx.doi.org/10.1109/ICSE.2013.6606617.

[6] Nargis Fatima, Sumaira Nazir, Suriayati Chuprat, Knowledge sharing, a key sustainable practice is on risk: An insight from Modern Code Review, in: 2019 IEEE 6th International Conference on Engineering Technologies and Applied Sciences (ICETAS), IEEE, Kuala Lumpur, Malaysia, 2019, pp. 1–6, http://dx.doi.org/10.1109/ICETAS48360.2019.9117444.

[7] Stefan Berner, Roland Weber, Rudolf K. Keller, Observations and lessons learned from automated testing, in: Proceedings of the 27th International Conference on Software Engineering, ACM, 2005, pp. 571–579.

[8] Emil Alégroth, Robert Feldt, On the long-term use of visual gui testing in industrial practice: a case study, Empir. Softw. Eng. 22 (6) (2017) 2937–2971.

[9] Riccardo Coppola, Emil Alégroth, A taxonomy of metrics for GUI-based testing research: A systematic literature review, Inf. Softw. Technol. (2022) 107062.

[10] Liming Dong, He Zhang, Lanxin Yang, Zhiluo Weng, Xin Yang, Xin Zhou, Zifan Pan, Survey on pains and best practices of code review, in: 2021 28th Asia-Pacific Software Engineering Conference (APSEC), IEEE, Taipei, Taiwan, 2021-12, pp. 482–491, http://dx.doi.org/10.1109/APSEC53868.2021.00055.

[11] Motahareh Bahrami Zanjani, Huzefa Kagdi, Christian Bird, Automatically recommending peer reviewers in modern code review, IEEE Trans. Softw. Eng. 42 (6) (2016) 530–543, http://dx.doi.org/10.1109/TSE.2015.2500238.

[12] Aleksandr Chueshev, Julia Lawall, Reda Bendraou, Tewfik Ziadi, Expanding the number of reviewers in open-source projects by recommending appropriate developers, in: 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME, 2020, pp. 499–510, http://dx.doi.org/10.1109/ICSME46990.2020.00054.

[13] Adam Alami, Marisa Leavitt Cohn, Andrzej Wąsowski, Why does code review work for open source software communities? in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 1073–1083, http://dx.doi.org/10.1109/ICSE.2019.00111.

[14] Ishan Banerjee, Bao Nguyen, Vahid Garousi, Atif Memon, Graphical user interface (GUI) testing: Systematic mapping and repository, Inf. Softw. Technol. 55 (10) (2013) 1679–1694, http://dx.doi.org/10.1016/j.infsof.2013.03.004.

[15] Emil Alégroth, Kristian Karl, Helena Rosshagen, Tomas Helmfridsson, Nils Olsson, Practitioners' best practices to adopt, use or abandon model-based testing with graphical models for software-intensive systems, Empir. Softw. Eng. 27 (5) (2022) 1–42.

[16] Andreas Bruns, Andreas Kornstadt, Dennis Wichmann, Web application tests with selenium, IEEE Softw. 26 (5) (2009) 88–91.

[17] Stanislava Nedyalkova, Jorge Bernardino, Open source capture and replay tools comparison, in: Proceedings of the International C* Conference on Computer Science and Software Engineering, 2013, pp. 117–119.

[18] Alper Silistre, Onur Kilincceker, Fevzi Belli, Moharram Challenger, Geylani Kardas, Models in graphical user interface testing: Study design, in: 2020 Turkish National Software Engineering Symposium, UYMS, IEEE, 2020, pp. 1–6.

[19] Emil Alégroth, Robert Feldt, Lisa Ryrholm, Visual GUI testing in practice: Challenges, problemsand limitations, Empir. Softw. Eng. 20 (3) (2015) 694–744, http://dx.doi.org/10.1007/s10664-013-9293-5.

[20] Tsung-Hsiang Chang, Tom Yeh, Robert C. Miller, GUI testing using computer vision, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2010, pp. 1535–1544.

[21] Emil Alegroth, Elin Petersen, 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), IEEE, Porto de Galinhas, Brazil, 2021, pp. 340–350, http://dx.doi.org/10.1109/ICST49551.2021.00046.

[22] A. Frank Ackerman, Priscilla J. Fowler, Robert G. Ebenau, Software inspections and the industrial production of software, in: Proc. of a Symposium on Software Validation: Inspection-Testing-Verification-Alternatives, Elsevier North-Holland, Inc., Darmstadt, Germany, USA, 1984, pp. 13–40.

[23] A.F. Ackerman, L.S. Buchwald, F.H. Lewski, Software inspections: an effective verification process, IEEE Softw. 6 (3) (1989) 31–36, http://dx.doi.org/10.1109/52.28121.

[24] M.E. Fagan, Design and code inspections to reduce errors in program development, IBM Syst. J. 15 (3) (1976) 182–211, http://dx.doi.org/10.1147/sj.153.0182.

[25] Forrest Shull, Carolyn Seaman, Inspecting the history of inspections: An example of evidence-based technology diffusion, IEEE Softw. 25 (1) (2008) 88–90, http://dx.doi.org/10.1109/MS.2008.7.

[26] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan Orbeck, Christopher Chockley, Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft, IEEE Trans. Softw. Eng. 43 (1) (2017) 56–75, http://dx.doi.org/10.1109/TSE.2016.2576451.

[27] Peter C. Rigby, Christian Bird, Convergent contemporary software peer review practices, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, in: ESEC/FSE 2013, Association for Computing Machinery, Saint Petersburg, Russia, New York, NY, USA, 2013, pp. 202–212, http://dx.doi.org/10.1145/2491411.2491444.

[28] Tobias Baum, Olga Liskin, Kai Niklas, Kurt Schneider, A faceted classification scheme for change-based industrial code review processes, in: 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS, IEEE, 2016, pp. 74–85.

[29] Nicole Davila, Ingrid Nunes, A systematic literature review and taxonomy of modern code review, J. Syst. Softw. 177 (2021) 110951, http://dx.doi.org/10.1016/j.jss.2021.110951.

[30] Gerrit, Gerrit code review, 2022, https://www.gerritcodereview.com.

[31] Phabricator Inc., Phabricator: Discuss. Plan. Code. Review. Test, 2022, https://www.phacility.com/phabricator.

[32] CACM Staff, CodeFlow: Improving the code review process at microsoft, Commun. ACM 62 (2) (2019) 36–44, http://dx.doi.org/10.1145/3287289.

[33] GitHub Inc., GitHub: Where the world builds software, 2022, https://github.com.

[34] GitLab B.V., GitLab, 2022, https://about.gitlab.com/.

[35] Atlassian, Bitbucket | Git solution for teams using Jira, 2022, https://bitbucket.org.

[36] Davide Spadini, Maurício Aniche, Margaret-Anne Storey, Magiel Bruntink, Alberto Bacchelli, When testing meets code review: Why and how developers review tests, in: Proceedings of the 40th International Conference on Software Engineering, ACM, Gothenburg Sweden, 2018, pp. 677–687, http://dx.doi.org/10.1145/3180155.3180192.

[37] Philip N. Johnson-Laird, Human and Machine Thinking, Psychology Press, 2013.

[38] Shi Qing Pan, Maria Vega, Alan J. Vella, Brian H. Archer, G.R. Parlett, A mini-Delphi approach: An improvement on single round techniques, Prog. Tour. Hosp. Res. 2 (1) (1996) 27–39.

[39] Ilker Etikan, Sulaiman Abubakar Musa, Rukayya Sunusi Alkassim, et al., Comparison of convenience sampling and purposive sampling, Am. J. Theor. Appl. Stat. 5 (1) (2016) 1–4.

[40] B. Kitchenham, S. Charters, Guidelines for Performing Systematic Literature Reviews in Software Engineering, Vol. 2, 2007.

[41] Vahid Garousi, Mika V. Mäntylä, Citations, research topics and active countries in software engineering: A bibliometrics study, Comp. Sci. Rev. 19 (2016) 56–77.

[42] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Inf. Softw. Technol. 106 (2019) 101–121, http://dx.doi.org/10.1016/j.infsof.2018.09.006.

[43] Dominic J. Farace, Joachim Schöpfel, Grey Literature in Library and Information Studies, 2010, http://dx.doi.org/10.1515/9783598441493, Cited by: 67; All Open Access, Green Open Access.

[44] Richard J. Adams, Palie Smart, Anne Sigismund Huff, Shades of grey: guidelines for working with the grey literature in systematic reviews for management and organizational studies, Int. J. Manag. Rev. 19 (4) (2017) 432–454.

[45] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–6.

[46] Claes Wohlin, Guidelines for snowballing in systematic literature studies and a replication in software engineering, in: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, 2014, pp. 1–10.

[47] Mai Thi Thanh Thai, Li Choy Chong, Narendra M. Agrawal, Straussian grounded theory method: An illustration, Qual. Rep. 17 (5) (2012).

[48] Barney G. Glaser, Anselm L. Strauss, Elizabeth Strutzel, The discovery of grounded theory; strategies for qualitative research, Nurs. Res. 17 (4) (1968) 364.

[49] Guoping Rong, Jingyi Li, Mingjuan Xie, Tao Zheng, The effect of checklist in code review for inexperienced students: An empirical study, in: 2012 IEEE 25th Conference on Software Engineering Education and Training, IEEE, Nanjing, China, 2012, pp. 120–124, http://dx.doi.org/10.1109/CSEET.2012.22.

[50] Qiang Fu, Francis Grady, Bjoern Flemming Broberg, Andrew Roberts, Geir Gil Martens, Kjetil Vatland Johansen, Pieyre Le Loher, Code review and cooperative pair programming best practice, 2017, http://dx.doi.org/10.48550/ARXIV.1706.02062.

[51] GitLab B.V., Code review guidelines, 2022, https://docs.gitlab.com/ee/development/code_review.html.

[52] Luca Pascarella, Davide Spadini, Fabio Palomba, Magiel Bruntink, Alberto Bacchelli, Information needs in contemporary code review, Proc. ACM Hum.-Comput. Interact. 2 (CSCW) (2018) 1–27, http://dx.doi.org/10.1145/3274404.

[53] Andrew Sutherland, Gina Venolia, Can peer code reviews be exploited for later information needs? in: 2009 31st International Conference on Software Engineering-Companion Volume, IEEE, 2009, pp. 259–262, http://dx.doi.org/10.1109/ICSE-COMPANION.2009.5070996.

[54] Shane McIntosh, Yasutaka Kamei, Bram Adams, Ahmed E. Hassan, The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 192–201.

[55] .NET Foundation, xUnit.net, 2022, https://xunit.net.

[56] GraphWalker, GraphWalker, an open-source model-based testing tool, 2022, https://graphwalker.github.io.

[57] Michel Nass, Emil Alegroth, SCOUT: A revised approach to GUI-test automation, 2020, p. 2.

[58] Emil Alegroth, Arvid Karlsson, Alexander Radway, Continuous integration and visual GUI testing: Benefits and drawbacks in industrial practice, in: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), IEEE, Vasteras, 2018, pp. 172–181, http://dx.doi.org/10.1109/ICST.2018.00026.

[59] Mountainminds GmbH & Co. KG, Java code coverage for eclipse, 2022, https://www.jacoco.org.

[60] Cobertura, A code coverage utility for Java, 2022, https://cobertura.github.io/cobertura/.

[61] Raymond P.L. Buse, Westley R. Weimer, Learning a metric for code readability, IEEE Trans. Softw. Eng. 36 (4) (2009) 546–558.

[62] Google, Google engineering practices documentation, 2022, https://google.github.io/eng-practices/.

[63] Le An, Foutse Khomh, Shane Mcintosh, Marco Castelluccio, Why did this reviewed code crash? An empirical study of mozilla firefox, in: 2018 25th Asia-Pacific Software Engineering Conference, APSEC, IEEE, 2018, pp. 396–405, http://dx.doi.org/10.1109/APSEC.2018.00054.

[64] Tobias Baum, Kurt Schneider, On the need for a new generation of code review tools, in: Pekka Abrahamsson, Andreas Jedlitschka, Anh Nguyen Duc, Michael Felderer, Sousuke Amasaki, Tommi Mikkonen (Eds.), Product-Focused Software Process Improvement, Springer International Publishing, Cham, 2016, pp. 301–308.

[65] Quinn Hanam, Ali Mesbah, Reid Holmes, Aiding code change understanding with semantic change impact analysis, in: 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2019, pp. 202–212.

[66] Tim Menzies, Justin S Di Stefano, Mike Chapman, Ken McGill, Metrics that matter, in: 27th Annual NASA Goddard/IEEE Software Engineering Workshop, 2002. Proceedings., IEEE, 2002, pp. 51–57.

[67] Emre Doğan, Eray Tüzün, Towards a taxonomy of code review smells, Inf. Softw. Technol. 142 (2022) 106737.

[68] Robert C. Martin, Design principles and design patterns, Object Mentor 1 (34) (2000) 597.

[69] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E. Johnson, John Vlissides, et al., Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Deutschland GmbH, 1995.

[70] Anton Smirnov, Best practices code review test automation, 2022, https://itnext.io/best-practices-code-review-test-automation-fb970feeca4c.

[71] Maurizio Leotta, Diego Clerissi, Filippo Ricca, 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, 2013, pp. 108–113, http://dx.doi.org/10.1109/ICSTW.2013.19.

[72] Lisa A. Guion, David C. Diehl, Debra McDonald, Triangulation: establishing the validity of qualitative studies, Edis 2011 (8) (2011) 3.

[73] Felipe Ebert, Fernando Castor, Nicole Novielli, Alexander Serebrenik, Confusion in code reviews: Reasons, impacts, and coping strategies, in: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2019, pp. 49–60.

[74] Per Runeson, Martin Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131–164.

[75] Emil Alégroth, Javier Gonzalez-Huerta, Towards a mapping of software technical debt onto testware, in: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, Vienna, Austria, 2017, pp. 404–411, http://dx.doi.org/10.1109/SEAA.2017.65.