<http://www.diva-portal.org>

Postprint

This is the accepted version of a paper presented at *14th IEEE International Conference on Software Testing, Verification and Validation, ICST 2021, 12 April 2021 through 16 April 2021*.

Citation for the original published paper:

Alégroth, E., Petersén, E., Tinnerholm, J. (2021)

A Failed attempt at creating Guidelines for Visual GUI Testing: An industrial case study

In: *Proceedings - 2021 IEEE 14th International Conference on Software Testing,*

Verification and Validation, ICST 2021, 9438551 (pp. 340-350). Institute of Electrical and Electronics Engineers Inc.

<https://doi.org/10.1109/ICST49551.2021.00046>

N.B. When citing this work, cite the original published paper.

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Permanent link to this version:

<http://urn.kb.se/resolve?urn=urn:nbn:se:bth-21820>

A Failed attempt at creating Guidelines for Visual GUI Testing: An industrial case study

Emil Alégroth
SERL Sweden
Blekinge Institute of Technology
Karlskrona, Sweden
emil.alegroth@bth.se

Elin Petersén
Insitution of Computer Science
Linköping University
Linköping, Sweden
elipe344@student.liu.se

John Tinnerholm
Insitution of Computer Science
Linköping University
Linköping, Sweden
john.tinnerholm@liu.se

Abstract—Software development is governed by guidelines that aim to improve the code’s qualities, such as maintainability. However, whilst coding guidelines are commonplace for software, guidelines for testware are much less common. In particular, for GUI-based tests driven with image recognition, also referred to as Visual GUI Testing (VGT), explicit coding guidelines are missing.

In this industrial case study, performed at the Swedish defence contractor Saab AB, we propose a set of coding guidelines for VGT and evaluate their impact on test scripts for an industrial, safety-critical system. To study the guidelines’ effect on maintenance costs, five representative manual test cases are each translated with and without the proposed guidelines in the two VGT tools SikuliX and EyeAutomate. As such, 20 test scripts were developed, with a combined development cost of more than 100 man-hours. Three of the tests are then maintained by one researcher and two practitioners for another version of the system and costs measured to evaluate return on investment. This analysis is complemented with observations and interviews to elicit practitioners’ perceptions and experiences with VGT.

Results show that scripts developed with the guidelines had higher maintenance costs than scripts developed without guidelines. This is supported by qualitative results that many of the guidelines are considered inappropriate, superfluous or unnecessary due to the inherent properties of the scripts, e.g. their natural small size, linear flows, natural separation of concerns, and more. We conclude that there are differences between VGT scripts and software that prohibit direct translation of guidelines between the two. As such, we consider our study as a failure but argue that several lessons can be drawn from our results to guide future research into guidelines for VGT and GUI-based test automation.

Index Terms—Automated Testing, Visual GUI testing, Guidelines for testing, Case study, Industrial study.

I. INTRODUCTION

Modern software development is team-based, governed by processes and supported by guidelines [1]. These guidelines aim to make the codebase consistent and thereby improve its qualities such as modifiability, reusability, analyzability, modularity and testability and thereby the *maintainability* of the software (ISO 25010) [1], [2].

However, whilst coding guidelines for software are common, testware is not treated with the same rigor despite being subject to common challenges, such as technical debt [3]. For lower level tests, such as unit tests, coding guidelines have been proposed [4], but for higher level tests such as GUI tests that use image recognition, i.e. Visual GUI Testing (VGT) [5],

[6], only high level adoption and usage guidelines have been proposed [7]. This is recognized as a challenge, since research has shown that poorly designed tests are associated with maintenance problems and higher costs [8], [9], [10].

VGT is a tool-driven technique, with tools such as SikuliX [11] and EyeAutomate [12], [13], which use image recognition for interaction with the SUT. The technique has several benefits compared to other GUI-test automation techniques [14], [15] and has been shown to be both applicable [16] and feasible [6], [7] in practice. However, the technique is associated with challenges [5], [17], such as high maintenance costs.

In this paper, we propose guidelines, derived from literature on guidelines for software, testware, and VGT expert knowledge that aim to improve maintainability of VGT scripts. The guidelines are evaluated in an industrial case study at the Swedish defence contractor Saab AB. In the study, manual tests are translated into VGT scripts, designed with and without the guidelines, in two VGT tools. The guidelines’ effectiveness is then measured based on how they affect the maintenance costs of the developed scripts. These measurements are triangulated by interviews and observations from practitioners from Saab who participated in the study.

Results show that VGT would not provide positive return on investment (ROI) compared to manual testing at Saab. More importantly, and contradictory to expected results, most of the tests developed with the proposed guidelines required more effort to maintain. Interview results highlight that many of the guidelines seem inappropriate, superfluous, or unnecessary for VGT scripts due to their inherent characteristics, which was an unexpected finding. From these observations, we discuss important lessons learned and propose directions for future research. Hence, despite the unconventional nature of this paper within software engineering, which has a tradition of publishing positive results, we claim the following contributions from this work:

- C1:** Contradictory results regarding the feasibility of the development costs, maintenance costs and ROI of VGT from an industrial context.
- C2:** Lessons learned from a failed attempt at creating coding guidelines for VGT and proposals for future research on automated GUI-testing guidelines.

II. RELATED WORK

Guidelines for automated testing have been proposed for tests on various levels of abstraction, with early examples of guidelines being presented already in the mid 80s [18]. Another example is Schneider et al. who propose guidelines for JUnit tests [4]. Guidelines for GUI-level tests have also been proposed, for instance by Hellman et al. [19] who found that test cases that contain complete series of GUI interactions (single-layer architectures) are more difficult to understand, especially if the business logic is not clear. Thus, concluding that test scripts at the GUI-level, similar to software, should be modularised into areas of concern. In addition, due to high level tests' fragility to change on multiple levels of SUT abstraction, care should be taken to minimize code duplication, which is also a cause of test technical debt [20], [3]. Instead, tests should be split into reusable units, which is once more a common software practice [21]. However, as presented by Qusef et al. the resulting quality of tests, developed with guidelines, is dependent on the competence and knowledge of the engineer [22].

Many guideline studies highlight the similarities between code for test scripts (testware) and software, a connection further explored by Alégroth et al. in two studies where they map software code smells to testware [20], [3]. In fact, in one of these studies, 21 of the most common code smells for software are successfully mapped to testware, including smells such as long methods, divergent change, shotgun surgery, etc. As such, the need for testware coding guidelines is evident. However, for VGT, comprehensive analysis of such guidelines is lacking, although an attempt at some best practices has been presented [7]. These guidelines do, however, not cover development best practices exclusively, rather, they provide more general guidance for the adoption, organizational use and long-term use of VGT.

VGT has been extensively studied, primarily through empirical case studies where test cases have been developed to showcase the technique's applicability and feasibility in industrial practice [16], [23], [5], [7]. The technique has also been studied from a technical perspective and how it could be integrated into automated model-based approaches [17] and what benefits it provides in a continuous integration context when compared to lower level test techniques [12]. Many of these studies, similar to the research in this paper, compare various tools (VGT tools and/or other GUI testing tools) in industrial environments. Another commonality in these studies is that the number of developed VGT test scripts are generally few, often only ranging between 5 to 15 tests. This is because VGT test development is time-consuming and considering the resource constraints, often associated with industrial research, this is the number of tests to be expected. As such, the 20 test scripts produced for this study are on par with previously published works in this area, which should not be compared with the size of test suites reported in other automated testing research where hundreds or even thousands of test cases are often included due to industrial norms [24].

In summary, VGT is well studied in industrial practice but not with the explicit focus of identifying coding guidelines aimed at reducing VGT script maintenance costs. This study aims to fill this knowledge gap, drawing inspiration from previous research, using similar methods and procedures for data collection, sampling and analysis, to achieve results of rigor on par with previous works.

III. METHODOLOGY

This study's objective was to identify and evaluate coding guidelines for VGT scripts and measure their impact on maintenance costs in industrial practice. These guidelines should serve to improve the quality of the automated tests in terms of *modifiability*, *reusability*, *analyzability*, *modularity* and *testability*, which all have a positive impact on *maintainability* [1], [21]. This objective was chosen since maintainability of GUI test cases is a commonly reoccurring challenge in literature and perceived as one of the key grounds for VGT abandonment in practice [5]. To improve the granularity, the research objective was broken down into two research questions:

RQ1: What guidelines for software and testware can be used for developing Visual GUI testing scripts?

RQ2: What is the return of investment, in terms of applicability and cost, of using coding guidelines for the development of Visual GUI testing scripts?

To answer research question 1 (RQ1), academic literature and industrial best practices are elicited to extrapolate a set of guidelines that we propose as suitable for VGT test development to improve test quality. These guidelines also serve as input to answer research question 2 (RQ2), where the answer to RQ2 serves to verify the applicability of the guidelines.

The research questions were answered through an industrial case study performed at the Swedish military defence and civil security developer Saab AB. In particular, the study was performed in the business area of Aeronautics, i.e. software for aircraft. This software is GUI intensive, in the order of millions of lines of code, incorporating different technologies and third party components for a 3D computer-aided design (CAD) modeling tool. Due to the tool's reliance on third party components, some components have restricted code access, which limits automated testing of these components, and the software at large, to GUI-based testing. This limitation is common in practice and presents challenges for test automation that VGT can mitigate due to the technique's limited requirements on SUT access, i.e. it only needs access to the SUT's GUI [6], [5], [7], [12]. These SUT characteristics are common to SUTs studied in previous VGT research [16], [5], [6] and make Saab's product an ideal candidate to evaluate the applicability (usefulness and development costs) and feasibility (maintenance costs) of the technique.

A. Visual GUI testing, Sikuli and EyeAutomate

Visual GUI testing, also referred to as 3rd generation GUI-based testing is a tool-driven technique with tool's that use image recognition [7], [14]. Image recognition allows the tools to

interact with any system with a GUI, sending inputs by using mouse and keyboard commands and reading outputs from the pictorial GUI as shown to the user. This means of interaction enables the tools to disregard how the SUT is implemented, allowing SUTs composed of several technologies, or that lack other interfaces for testing, to be automated. Additionally, since the technique utilizes the same interfaces as the human user, the test cases become more similar to how the end-user will use the system. This characteristic has benefits when compared to other GUI testing techniques that rely on APIs, technical interfaces or document object models (DOM). For instance, Alégroth et al. demonstrated these differences in a study with the tool GUITAR where 3rd generation GUI based testing was compared to 2nd generation testing [17]. Their study found significant differences in how the two approaches report false positive and false negative test results.

Image recognition-based testing is, however, associated with many challenges [5]. Some are VGT specific, such as image recognition fragility when the SUT changes, non-deterministic widget identification and high test execution costs. Others are general for automated GUI testing, such as synchronization problems since the tests run asynchronously against the SUT. Automated GUI testing, including VGT, is also associated with high maintenance costs, since the tests are affected by not only changes to the SUT's GUI but also all changes that affect the SUT's behavior.

Despite these challenges, VGT is both applied and has been used long-term in industrial practice [7]. However, it has been observed that many companies that adopt the technique, quickly abandon it due to maintenance costs. One observation that explains this abandonment is that VGT tests are intuitive to develop and are therefore seldom properly designed, modularized, and broken into reusable items to make them maintainable over time.

The tools used in the study are SikuliX [11] and EyaAutomate [14] (Formerly known as JAutomate [13]). SikuliX is an open source tool, written in Java, with an Integrated Development Environment (IDE) that allows users to write test scripts in Python. The IDE visualizes target images in the scripts, making them intuitive (an example is shown in Figure 1 A).

EyeAutomate has a similar looking IDE to SikuliX but adopts a custom scripting language for test scripts. The tool is written in Java, enabling its use on most platforms and environments. The tool is freeware, designed for commercial use and as such includes features such as continuous integration support, test log generation, test recording, and more. An example EyeAutomate script is shown in Figure 1 B.

B. Research procedure

The study followed the seven steps presented in Figure 2, which can be split into three phases:

- 1) **Pre-study:** This phase sought to answer RQ1 by collecting coding guidelines from literature, VGT experts and the studied company, to compile applicable coding guidelines for VGT.

Fig. 1. Example snippets of SikuliX (A) and EyeAutomate (B) scripts for a web application.

Fig. 2. Visualization of the methodology used during the case study.

- 2) **At the company:** This phase served to collect input for RQ2 through the development of VGT scripts that were later maintained by practitioners from Saab. These practitioners were also observed and interviewed.
- 3) **Analysis:** This phase served to analyze the collected data and answer RQ2.

In the continuation of this section, a break-down of the methodology's steps will be presented, including the outputs from each step.

Step 1: Literature and company analysis - To formulate a basis for the VGT guidelines, we performed an unstructured literature review, looking particularly at the guidelines proposed by Visser et al. [2] and the guidelines proposed

by the Software Improvement Group (SIG). These guidelines were of interest since they focus on building maintainable software, which includes several sub-characteristics such as modifiability and reusability [1], [21], as defined in ISO 25010. However, additional inspiration was taken from literature on automated GUI testing [19], guidelines for VGT [7] and research on code- and test-smells [3], [20]. We are aware that there is a myriad of standards and code smells to consider, and although we took many of these into consideration, we delimited us to a subset of more general guidelines explicitly for maintainability due to the study’s research objective.

To complement the literature, Saab’s coding standards and naming conventions (e.g. for classes, methods, and variables) were studied. These guidelines serve to improve understandability of the produced software and have been designed by developers at Saab with at least 5 years of software development experience. It was perceived that utilizing these domain-specific guidelines would be beneficial for the acceptance of the VGT guidelines since they would be used by practitioners at the company. Note that these context-specific guidelines have purposefully been omitted from the paper to improve its generalizability. Instead, we simply stress the need of using common naming conventions (See Section III-C).

Step 2: VGT Expert interview - To complement the results of the literature review and gain understanding about how to adopt the coding guidelines, an interview was held with a VGT expert. We label this individual as an expert since he has created several VGT tools, worked with the technique in practice for more than 15 years and published his work [13].

The expert interview was open-ended, was 60 minutes in length, and focused on VGT best practices. Five best practices were identified that were incorporated in the proposed VGT guidelines (See Section III-C). Three of these guidelines are VGT specific whilst two are more generic for automated testing or GUI testing in general.

Step 3: Case company interviews - In the first step of the second phase (Step 3), interviews were conducted to elicit background information about the company and its GUI testing practices. These interviews provided an overview of how testing is conducted as well as insights into how tests are structured, what test cases could be suitable for the study and other contextual information. In addition, cost information about the current practices was elicited, in particular how much time is spent on manual GUI-based testing.

Eight semi-structured interviews were conducted, ranging from 15 to 30 minutes in length with different roles, including five system/software developers, one tester, one system architect and one test lead. The participants were involved in two different projects at Saab and they had between one to five (average 2.75) years of industrial experience.

Step 4: VGT test development - In the second step of the second phase (Step 4), manual test cases for one of the company’s products were chosen through a process based on three exclusion criteria. These criteria aimed to exclude:

- Tests that were only in one release of the SUT.
- Tests that did not require maintenance in any release.

- Tests that could not be performed manually without additional training due to the SUT’s requirements for domain-specific knowledge.

The entire test suite consists of 60 test cases, but after exclusion, 13 test cases were identified as candidates. However, due to the resource constraints (i.e. amount of given access to the SUT, personnel, and time constraints), it was not possible to automate all 13 tests. The test cases were therefore prioritized based on size, complexity, and number of updates/maintenance instances correlated to SUT updates to identify a minimal, but representative, sample. The size was quantified based on the number of test steps where a small test case had less than 15 steps, a medium test had between 14 to 30 steps and a large had more than 30 steps. Complexity was evaluated based on the amount of required domain-specific knowledge to execute, give correct inputs, and interpret outputs of the test. Additionally, the number of subsystems, six in total (denoted Subsystem A to F) involved in the tests were also considered. These subsystems differed in functionality and size but, since they were blackbox tested during the study, their size was not accounted for in the test case selection process. The number of updates in each release was also, as previously stated, considered to ensure that the selected tests required maintenance.

TC ID	Size	Complexity	Updates	Subsystems
TC1	Medium	Low	>1	A and B
TC2	Small	High	>1	A, B, and C
TC4	Large	High	>1	A, B, C, D, and E
TC10	Large	Low	1	A, B, D, and F
TC11	Small	Low	1	A and B

TABLE I
SUMMARY OF THE TEST CASES CHOSEN FOR DEVELOPMENT IN STEP 4 OF THE STUDY.

Using these metrics, five test cases were chosen for automation, labeled TC1, TC2, TC4, TC10, and TC11. Table I presents an overview of these tests.

Those five were then developed for version V0 of the SUT. Five more versions were made available, denoted V1-V5, to measure the maintenance costs (presented further in Step 5). The test cases were automated with and without the proposed guidelines using EyeAutomate [13] and Sikuli [11]. This resulted in 4 test suites of five test scripts each (20 scripts in total). The two tools were chosen due to their use in industrial practice and academic research, where they have been previously compared based on features, cost, and effectiveness [12], [6], [14].

To mitigate learning bias during automation, the order of how test cases were implemented was randomized. For example, the order TC1 was automated was first Sikuli without guidelines, followed by Sikuli with guidelines, followed by EyeAutomate without guidelines, and finally EyeAutomate with guidelines. The order the test cases were implemented was also randomized, giving the order TC4, TC2, TC1, TC10, and finally TC11. The tests were implemented by one of the authors of the paper who spearheaded the study.

Time was measured by tracking time from start of development until the test script executed successfully. Thus, development time includes test execution time for one or several test runs but excludes breaks or other interruptions.

Step 5: VGT test maintenance study - In the third step of the second phase (Step 5), three developed test cases from Step 4 (TC1, TC2, and TC4) were chosen to study the maintenance costs of VGT scripts developed with and without guidelines. The reason why not all five tests developed in Step 4 were maintained was due to time and resource constraints.

As stated in Step 4, all test cases were developed for version V0 of the SUT. To emulate an actual maintenance scenario, the SUT was changed to version V2, which included changes to the SUT that required the three test cases to be maintained. Four other versions of the SUT had been identified that required tests to be maintained, but due to the resource constraints of the study, it was determined that having to switch versions several times would not be possible.

Maintenance was conducted by three individuals; the driving researcher who maintained all three test cases (TC1, TC2, and TC4) and two employees from Saab. Employee 1, who had 1.5 years of industrial experience at the company and who was familiar with the test cases, maintained TC4. Employee 2, who had 6 years of industrial experience and 1.5 years of experience at the company but was not familiar with the test cases, maintained TC1. The allocated time, by the company, to use the employees for the study was 20 hours, which was the limiting factor why each employee only maintained one test case each. However, by having an overlap between which test cases the researcher and employees maintained, comparisons could be made to get more insight into the maintenance costs.

At the start of the maintenance procedure, the company's employees were given a tutorial of the tools and the VGT development guidelines. The employees were then instructed to maintain their allotted tests with and without the guidelines and asked to randomize the order of what tool and test (with/without guidelines) to start with. Employee 1 first maintained TC4 developed in Sikuli without guidelines, followed by TC4 developed in EyeAutomate with guidelines, followed by TC4 developed in Sikuli with guidelines, and finally TC4 developed in EyeAutomate without guidelines. Employee 2 first maintained TC1 developed in EyeAutomate without guidelines, followed by TC1 developed in Sikuli with guidelines, followed by TC1 developed in Sikuli without guidelines, and finally TC1 developed in EyeAutomate with guidelines. This procedure aimed to reduce learning bias, but since each test is semantically equivalent regardless of implementation, such bias cannot be ruled out.

Maintenance included updates to script logic, imagery and synchronization and was observed, with the talk-/think-aloud protocol [25], to acquire insights about the Saab employees' perceptions and experience with VGT and the guidelines. The observations were unstructured, documented using note-taking, and carried out during the entire duration the employees spent on maintaining the scripts.

Step 6: VGT guideline interviews - In the final step of the

second phase (Step 6), semi-structured interviews were carried out with the employees who performed the maintenance of the tests. These interviews served to triangulate the results from the observations, in particular focusing on the employees' perceptions of the different tests and tools as well as their feedback on the guidelines. In particular, each suggested guideline was discussed to gain insights if it provided value, was cumbersome or not considered applicable.

Step 7: Analysis - In the third phase of the study, the collected results were synthesised to evaluate the effects of the guidelines on the VGT script development and maintenance. Qualitative data was analyzed using simple semantic coding and clustering of interview statements whilst quantitative data was analyzed using descriptive statistics, plotting the development, maintenance costs, and approximating ROI, similar to previous works by Alégroth et al. [6]. As such, the qualitative data was used to explain the quantitative findings. In particular, to identify which guidelines were perceived to have a positive effect, no effect, or a negative effect on the maintenance effort. It also served as input to answer the research questions and draw the study's conclusions regarding the effectiveness of the guidelines, presented in Section IV, and implications of the results discussed further in Section V.

C. VGT guidelines

Table II summarizes the 12 VGT coding guidelines that were identified during the first phase of the study and used in the second phase to evaluate their effect on the maintainability of VGT scripts. The guidelines are derived from commonly used coding standards, combined with input from best practices obtained from VGT research and one VGT expert. In the continuation of this section, we will present the guidelines in more detail.

G1: Write short units of code - Short units of code are associated with easier analysis, reuse, and quality assurance compared to longer units. This is achieved by ensuring that each unit has a single responsibility, which in the case of VGT implies that scripts should focus on specific test actions, with few steps and single assertions that verify well defined SUT functionality. No unit should be larger than 15 lines of code [21]. However, we stress that this quantity is a guideline as the level of programming language must also be accounted for, i.e. 15 lines of Java code do not necessarily have the same semantic meaning as, for instance, Python code.

G2: Write simple units of code - By minimizing the number of branch conditions, cyclomatic/McCabe complexity, units become easier to analyze, understand, and modify. In the case of VGT, this implies keeping smaller units clear of decision points, isolating test functionality, and ensuring that the test suites are well designed, utilizing reusable artifacts. According to guidelines for software development, the number of branch points in a unit should not exceed four [21].

G3: Write code once - Avoiding duplication is paramount for VGT tests since script maintenance is often required and should there be minimized to as few scripts as possible. Requiring several tests to be maintained in the same way

#	Description	Rationale
G1	Write short units of code (less than 15 lines [2])	Avoid units with mixed responsibilities to maximize code reuse and understandability of the code.
G2	Write simple units of code	Avoid branch conditions (mitigate cyclomatic complexity [2]) to maximize test code readability.
G3	Write code once	Avoid code duplication to minimize need for repeated bug fixes (Avoid identical snippets longer than 6 lines of code [2]).
G4	Keep unit interfaces small	Avoid more than 4 input parameters to maximize readability and reuse (Many parameters indicate multiple responsibilities [2]).
G5	Separate test cases according to manual test steps	Avoid tight coupling (Separation of concerns [19]) of units to maximize modularity, modifiability of individual units and unit reuse.
G6	Refer to images once	Avoid multiple instances of the same image to minimize repeated maintenance [7] (Derived from G3).
G7	Keep the codebase as small as possible	Avoid unnecessary code base growth to minimize need for code analysis [7].
G8	Use descriptive and uniform naming conventions	Avoid inconsistent naming of variables and methods to maximize readability and understandability of code.
G9	Use synchronization checks	Avoid static synchronization (Halt execution of X seconds) to minimize the need for performance related maintenance.
G10	Take a screenshot when tests fail	Avoid unnecessary fault analysis by providing a screenshot of system state at failure [23] (Only necessary for Sikuli in this case).
G11	Take care to choose suitable imagery	Avoid development cost associated with image refinement by taking unique, information rich, screenshots of suitable size.
G12	Write clean code	Avoid decreased productivity of team members by removing code smells, dead code, bad comments, commented code, long identifier names, magic constants and badly handled exceptions.

TABLE II

SUMMARY OF GUIDELINES FOR VGT THAT WERE EXTRACTED AND USED DURING THE STUDY.

increases the risk of human error and that some tests are maintained differently. Additionally, copy and paste of test script functionality should be avoided.

Keep unit interfaces small - Units with many parameters become cumbersome to reuse and is often a sign that the unit has multiple responsibilities. Hence, developers should strive to minimize the number of used parameters, which should not exceed more than four per unit.

G5: Separate test cases according to manual steps - Similar to separation of unit concerns [19], separating test scripts into units with well-defined test focus helps with test suite design. Designing tests with loose coupling helps improve modularity and eases maintenance by improving analyzability and readability of the scripts. In particular, VGT scripts that have several test focuses, e.g. test several features, should be split into multiple scripts.

G6: Refer to images once - This guideline is derived from G3, and aims to reduce maintenance effort, and human error, by storing each image used in the test suite only once and referring to it through variables. This enables maintenance of all broken test cases that use the affected image at once. It also ensures consistency of used imagery throughout the test suite.

G7: Keep the codebase as small as possible - By avoiding unnecessary growth of the codebase, the test suite becomes easier to analyze and maintain. For VGT this means to actively consider the design and architecture of the tests as well as refactoring the scripts.

G8: Use descriptive and uniform naming conventions - By using consistent and descriptive naming conventions, the scripts should become easier to analyze and understand. This also makes the code-base easier to search and thus easier to reuse. In the case of VGT scripts that follow G5, using descriptive variable names for images is also essential to make

the tests understandable without having to lookup the actual code or image being used.

G9: Use synchronization checks - To ensure synchronous execution, all actions that trigger state transitions in the SUT with possible loading times must be followed by synchronization checks. These checks should be dynamic, i.e. wait for a specific visual state, rather than static, e.g. wait for a set amount of time. Dynamic checks have the benefit of adapting to varying SUT delays as well as minimizing the amount of time the script has to wait before the next action is taken. Thus, minimizing overall test execution time.

G10: Take a screenshot when tests fail - To ease SUT maintenance and test failure replication, it is good practice to automatically capture screenshots of failed actions or state transitions when a test fails [23]. This functionality is inherent to some VGT tools but for some, e.g. SikuliX, it must be added by the test developer.

G11: Take care to choose suitable imagery - The image recognition is highly dependent on images being taken of suitable size. Using larger images provide the algorithms with more information but increases the risk of changes within the image that may cause failure. Using images that are too small may not contain enough information for the algorithm to find a unique match. Unfortunately, no general guideline can be given of what is the best image to capture, since it is context-dependent. Instead, developers are encouraged to actively consider what works in their environment and adapt through gained experiences.

G12: Write clean code - Keeping TODOs, comments, and dead code in the code-base makes the scripts harder to read and analyze. Additionally, code should be devoid of code smells or other bad patterns that should be removed if identified. An example of such a smell is “shotgun surgery”, which implies that to maintain a test case, changes are required to many

units spread across a single or several test scripts [3]. Other examples include removal of bad comments, long identifier names and magic constants.

IV. RESULTS

In the first part of this section, we will present the quantitative results on development and maintenance costs that were collected in the study. These results are also used to approximate the return on investment (ROI) of automating manual test cases with VGT with and without the proposed VGT guidelines. In the second part, we present the qualitative results.

A. Development/maintenance costs and ROI

Fig. 3. Time spent to **develop** VGT tests, written in Sikuli and EyeAutomate, with and without the proposed coding guidelines.

Development costs: Figure 3 presents the time that each test case took to develop without guidelines (denoted Development Version 1 (DV1) in the graph) and with guidelines (denoted DV2 in the graph) in the two VGT tools SikuliX and EyeAutomate. In the graph, the developed test cases are presented in the order they were randomly selected to be developed in. The order of individual bars also varies based on what tool and scripting approach (DV1 or DV2) that was randomly selected to be used first.

As shown, the tests developed without guidelines (DV1) were consistently quicker to develop. This could be explained by the additional cognitive load that the guidelines placed on the developer as well as the additional work required to follow the guidelines, e.g. split tests into modules and add suitable synchronization checks. Worth noting is that the development time does not reflect the complexity of the manual tests, i.e. more complex manual test cases do not consistently take more time to implement (Complexity of tests shown in Table I). However, the size of the manual tests correlates with the

development time as TC1, TC4, and TC10 were of medium to large size (more than 15 steps) whilst TC2 and TC11 were considered small (less than 15 steps). Also worth noting is that the development time of each set of test cases was on average 21 hours (standard deviation 8) and over 100 hours in total for all 20 test cases. Additionally, we observe that the development time per individual test case is in the same order of magnitude (on average 5.7 hours) as reported in previous research (average 3 hours per test) [6].

Fig. 4. Time spent to **maintain** VGT tests, written in Sikuli and EyeAutomate, which follow/do not follow best practice guidelines for script development.

Maintenance costs: Figure 4 presents the time each test case took to maintain without guidelines (denoted DV1 in the graph) and with guidelines (denoted DV2 in the graph). The graphs are presented chronically in the order they were maintained and by which developer, where P1 was the driving researcher, whilst P2 and P3 were employees at Saab. These measurements were acquired by applying the test cases developed for version V0 of the SUT on version V2 of the SUT, as presented in Section III-B.

Note that TC4 was maintained by both P1 and P2 whilst TC 1 was maintained by both P1 and P3 to make comparison of the costs possible.

As can be seen, the maintenance costs of TC1 and TC4 do vary between developers, but not necessarily due to VGT knowledge since it is observed that P2 (with no previous VGT experience) was able to maintain the test cases quicker in most cases than P1 (the driving researcher who developed all the tests). More importantly, we see that the guidelines do not have a positive effect on reducing maintenance costs. Exceptions to these observations are for TC4, maintained by P2 with EyeAutomate, where a marginal decrease is observed. Similarly, the maintenance cost of TC1, with both tools, is lower when developed/maintained with the guidelines by P1 and for TC1 written in EyeAutomate, maintained by P3.

Consequently, we observe that the proposed guidelines do not have conclusive impact on the maintenance costs of VGT scripts. Qualitative observations to support this conclusion are presented in Section IV-B.

Furthermore, comparing the maintenance costs of the scripts, we can once more see that they are in the same order of magnitude (on average 2.8 hours per test script) as the maintenance costs presented in previous research (on average 2 hours per test) [6].

Fig. 5. ROI approximation of VGT automation compared to manual testing when using, or not using, best practice design guidelines.

Return on Investment (ROI): Figure 5 presents an approximation of the time it would take for the developed test suites to provide positive ROI compared to manual testing, inspired by [6]. Positive ROI would be achieved when the total cost for VGT (colored lines) intersect the linear cost of manual testing (solid black line).

In the figure, short-dashed lines show an approximation of the maintenance costs based on the costs measured in the study (denoted worst case), i.e. assuming that the maintenance cost for each test run would be equal to the costs measured during the study. These lines indicate that the measured costs would never provide positive ROI compared to manual testing. In contrast, the long-dashed lines indicate the best case, where no maintenance is required. These lines show that positive ROI could be achieved in 9-12 executions of the scripts developed without guidelines and 16-17 executions for scripts developed with guidelines. Finally, the solid colored lines are approximations of the maximum allowed maintenance cost that would achieve positive ROI given the measured development costs. These approximations show that the maintenance costs had to be lower than 23 percent of the measured maintenance costs for scripts developed without guidelines, and less than 17 percent of the measured costs for scripts developed with the guidelines. However, whilst this is a substantial reduction in cost, it is not improbable to achieve, as indicated in previous

research [6]. We base this conclusion on the observation that test scripts should not require maintenance for each test execution and will elaborate further on this conclusion in Section V.

B. Practitioners' perceptions and observations about the guidelines

The following section provides a synthesis of the observations and interview results acquired about each proposed guideline.

G1: Write short units of code - This guideline was identified as inappropriate for VGT scripting since it lead to unnatural division of test functionality. VGT scripts have natural units of functionality, connected to the test steps, and forcing further division, although generating unique units, was not perceived to provide any advantages, e.g. improved reusability. It was therefore concluded that longer units provided more value but what the cut-off number of lines of code per script should be was not comprehensibly determined. Instead it is assumed that lines of code per unit should be determined by the unit's intended functionality.

G2: Write simple units of code - This guideline was implicitly ignored by the participants since it was perceived as redundant. This perception was determined to be due to VGT scripts' inherent scenario-based, often linear design (i.e. lack of branch conditions), and high level, intuitive, syntax.

G3: Write code once - This guideline was applied and was perceived appropriate because it eased reuse of units when applicable. However, the guidelines specified that duplicate code of six or more lines of code should be refactored into a reusable unit. This limit was perceived by the practitioners as high, but was considered a good starting point. The reason for this perception is that common actions, e.g. to login or toggling between windows, can generally be performed with VGT tools in fewer than 6 lines of code.

G4: Keep unit interfaces small - This guideline was ignored by the practitioners because it did not feel applicable. Both VGT tools support parameter-based scripting, SikuliX through definition of Python methods and EyeAutomate through data-driven test scripts with input parameters. The practitioners instead opted for the use of clear and well-structured interfaces.

G5: Separate test cases according to manual steps - This guideline was applied but the practitioners had different perceptions about its value and application. On the positive side, the employees agreed on the guideline to be appropriate and that it provided clear alignment between the manual test case steps and the units of the test script. However, this separation of concerns and units was observed as excessive and clumsy in some cases where there were better, more cross-cutting, concerns that could improve the scripts' qualities. This way of scripting also required excessive modularization and therefore descriptive names to be generated for all the units, which was considered time-consuming. In addition, maintenance of multiple units was considered more difficult since it lead to jumping around between units of the scripts.

Instead, it was observed that script code for manual test steps of similar functionality should be kept in the same unit based on semantic purpose of the steps.

G6: Refer to images once - This guideline was valuable and perceived appropriate for the VGT scripts. However, it was also considered time-consuming, similar to G5, to come up with suitable names for the images and even find what images that are available for reuse. In some cases, it was perceived quicker to simply take a new image than to go through the list of available images.

G7: Keep the codebase as small as possible - This guideline seemed superfluous since unnecessary code is seldom added to VGT scripts. The reason is because of the scripts' high level of abstraction that aims to replicate human user inputs, which are generally the minimum number of actions required to correctly stimulate the SUT's behavior. Hence, for more advanced applications of VGT, this guideline could be appropriate, but for translated, scenario-based, manual tests it is not.

G8: Use descriptive and uniform naming conventions - This guideline was valuable and was considered a good practice to summarize the functionality of test units. However, as pointed out by the employees, what is considered descriptive may be subjective for different developers. Implying the need for additional guidelines on how to construct suitable variable names. For the study, Saab's existing naming conventions were used, but as they are context-dependent, other conventions are required in other contexts.

G9: Use synchronization checks - This guideline was considered fundamental to make the tests work and to avoid unnecessary maintenance. The use of dynamic synchronization checks was found to be appropriate but was not employed in all scripts. Examples where static delays were used was when no suitable target was available for a dynamic wait, or when smaller delays were required, e.g. when providing textual inputs.

G10: Take a screenshot when tests fail - This guideline was perceived as appropriate for VGT but was used sparingly by the participants since they were actively observing the test executions during development and maintenance. However, this guideline should be used if the tests are run unsupervised, for instance in a CI environment.

G11: Take care to choose suitable imagery - This guideline was used and perceived as appropriate, but also challenging. The practitioners opted for keeping images as small as possible since it was observed that images with unnecessary pixel information were more prone to failure, which caused unnecessary maintenance. The trade-off between suitable image size and uniqueness is however still considered a fundamental challenge and is left out as future work.

G12: Write clean code - This guideline was ignored for the same reasons as G7 but was perceived by the study participants as a valuable guideline. Hence, minimizing out-commented code, dead code, etc. to enhance readability of the scripts.

V. DISCUSSION AND FUTURE WORK

Previous research into VGT has shown that the technique can provide positive ROI over time compared to manual testing [6]. However, our results go against this result since approximations based on the measured maintenance costs never provide ROI. In fact, the maximum maintenance cost to make VGT at all viable is substantially lower, i.e. 17-23 percent of the measured costs. The ratio between development and maintenance cost was in this study 0.493, comparable to the worst case maintenance cost ratio of 0.44 observed in previous work [6]. In contrast, the assumed best case scenario, i.e. no maintenance, would provide positive ROI between 9 to 12 test runs for scripts written without the guidelines and about 16 to 17 runs with the guidelines. However, six versions of Saab's SUT were identified as potential targets for the study, each with varying degrees of changes that would require test maintenance. As such it is unlikely that neither the best or worst case maintenance scenarios will be achieved and we can therefore not conclude that the approximations of unattainable ROI are true. In previous work, the best case maintenance (frequent maintenance) still had a ratio of 0.09 compared to development cost [13], implying that some maintenance is always required. However, the choice of which tests to automate must also be considered. New tests, associated with immature SUT functionality will be subject to more maintenance as the functionality matures. Instead, scripts should be developed primarily for stable functionality that is less likely to change.

The most important, yet surprising, finding was that the proposed guidelines had a negative effect on the maintenance costs. A couple of reasons can be identified to explain this result. First, the guidelines were developed assuming that there are inherent similarities between VGT scripts and traditional software. It seems that this assumption was partially false and can be attributed to the VGT scripts inherent properties of being short, linear and have a natural separation of concerns. This conclusion is also supported by the finding that the guidelines perceived most effective were more VGT specific, i.e. guidelines G6, G9 and G11.

Second, the number of guidelines were extensive (12 in total), implying that they add considerable cognitive load to the developer. This load consists in part of remembering what the guidelines are and secondly how to apply them correctly. As a result, the guidelines cause overhead that outweigh their intended benefits. For instance, it was observed that high modularization made it difficult to navigate the scripts and identify where/how to maintain them. This observation can be explained by the VGT scripts' logical (i.e. test sequences) and chronological (i.e. synchronization) components. Splitting the sequences into different units require the developer/tester to keep more information in their head, adding more cognitive load. This observation implies that there must be a balance between the amount of modularization that is added to the scripts contra the readability of said scripts.

Consequently, even though the practitioners that took part

in the study highlight both the need and potential value of guidelines for VGT, they also believe that these particular guidelines were not generally suitable. This conclusion implies a continued need for research in this area, building upon our mistakes. Primarily, that VGT scripts, due to their inherent properties, cannot be dealt with in the same way as software, despite previous research that presents results to the contrary [3]. Future research should investigate these differences, which could also provide a more general contribution to automated GUI-based testing.

Second, the results indicate that modularization has a direct impact on the readability of the test scripts, supported by related work [1], [21], and is fundamental in many coding guidelines. However, it seems that the trade-off between these two qualities is more sensitive for VGT scripts than for software. This observation explains why software guidelines like keeping modules to a maximum of 15 lines of code less applicable. One possible reason is that the scripts are highly dependent on the aforementioned chronological component, i.e. need for synchronization. Future research should therefore study how to modularize VGT scripts to instead achieve a positive effect.

Finally, coding guidelines add cognitive load, which can be controlled with the type and number of guidelines used. If the cognitive load is not controlled for, it could add overhead that nullifies potential benefits. The results of this study provide some support for this conclusion. While not strictly related to the examined guidelines, similar observations have been made in the context of design patterns where structure imposed by patterns might lead to systems becoming more difficult to maintain [26].

Future research should therefore not only look at what guidelines to use, but also how many and which are more suitable in different contexts. For such work, researchers should be inspired by the results presented in this paper, especially the practitioners' perceptions of each suggested guideline.

A. Threats to validity

In this section we summarize threats to validity based on guidelines by Runeson and Höst. [27].

Construct validity: The study was conducted in situ at Saab AB, with test cases for a live SUT, representative of a safety-critical GUI-based system. Whilst this could limit the results to safety-critical software, most GUI-driven systems share common types of GUI components, limiting the threat. Additionally, results were collected from practitioners working in the environment with both industrial experience and domain knowledge. However, the scripts were not integrated into day-to-day operations and the industrial participants of the study had limited knowledge of VGT prior to the study. We see these as minor threats, however, since the development costs are on par with previous industrial research in similar domains [6].

Internal validity: The study was performed using two commonly used VGT tools, in a setting similar to previous research [16], [5], [6], [7]. The research methods and the

collected data were similar to previous studies. The results provide a basis for our stated conclusions and show that the tests developed with guidelines are more costly than the tests developed without them. Qualitative observations support these quantitative results.

External validity: The study was performed at one company with a limited number of practitioners. However, the case company is very similar to other case companies in previous studies and the acquired results are in line with previously reported results [6]. Regardless, it is unknown if the same results would be observed in other contexts with more developers or other software. As such, researchers should learn from our results, tailor them, and re-evaluate further,

Reliability: Empirical, industrial, case studies are never fully replicable since they, per definition, change the environment where they are performed. Instead, we've dedicated a considerable portion of the paper to detail our research procedure and the steps taken to reach the results. Raw quantitative data and qualitative data summaries have also been made available for replication purposes at: http://mimicservice.com/resources/Replication_package.zip.

VI. CONCLUSIONS

Programming guidelines are commonplace for software, utilized to improve its qualities, such as maintainability. However, for testware such guidelines are far less common, despite many perceived similarities between software and testware. For GUI-based tests driven by image recognition, VGT, guidelines have not been previously suggested to our knowledge.

In this study, we proposed a set of guidelines for VGT and evaluated them in an industrial case study. This study shows that the guidelines were ineffective and had detrimental effects on the developed scripts' maintenance costs. The lessons learned from this observation, coupled with qualitative data from observations and interviews, are:

- 1) VGT scripts have characteristics that set them apart from software, prohibiting software guidelines to be directly reused.
- 2) Programming guidelines introduce cognitive load on the tester that needs to be managed.
- 3) Guidelines focused on modular design must trade-off modularity versus readability of VGT scripts due to their logical and chronological components.

From these conclusions, we outline a need for further research into specific guidelines adapted to VGT. This need is warranted by the observed lack of ROI concerning the translation of manual tests into VGT scripts, practitioners' statements, and the value of guidelines reported in related work. As such, future work in this area is perceived to not only provide contributions to VGT, but also to other GUI-based testing techniques.

VII. ACKNOWLEDGMENTS

This work was supported by the KKS foundation through the S.E.R.T. Research Profile project and the M.E.T.A. project at Blekinge Institute of Technology.

REFERENCES

- [1] B. Luijten, J. Visser, and A. Zaidman, "Faster defect resolution with higher technical quality of software," in *4th international workshop on software quality and maintainability (SQM 2010)*, 2010.
- [2] J. Visser, S. Rigal, G. Wijnholds, P. van Eck, and R. van der Leek, *Building Maintainable Software, C# Edition: Ten Guidelines for Future-Proof Code*. "O'Reilly Media, Inc.", 2016.
- [3] E. Alégroth and J. Gonzalez-Huerta, "Towards a mapping of software technical debt onto testware," in *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2017, pp. 404–411.
- [4] A. Schneider, "JUnit best practices," *Java World*, vol. 12, p. 181, 2000.
- [5] E. Alégroth, R. Feldt, and L. Ryrholm, "Visual gui testing in practice: challenges, problems and limitations," *Empirical Software Engineering*, vol. 20, no. 3, pp. 694–744, 2015.
- [6] E. Alégroth, R. Feldt, and P. Kolström, "Maintenance of automated test suites in industry: An empirical study on visual gui testing," *Information and Software Technology*, vol. 73, pp. 66–80, 2016.
- [7] E. Alégroth and R. Feldt, "On the long-term use of visual gui testing in industrial practice: a case study," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2937–2971, 2017.
- [8] S. Berner, R. Weber, and R. K. Keller, "Observations and lessons learned from automated testing," in *Proceedings of the 27th international conference on Software engineering*, 2005, pp. 571–579.
- [9] V. Garousi and M. Felderer, "Developing, verifying, and maintaining high-quality automated test scripts," *IEEE Software*, vol. 33, no. 3, pp. 68–75, 2016.
- [10] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *Ieee software*, vol. 29, no. 6, pp. 18–21, 2012.
- [11] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using gui screenshots for search and automation," in *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, 2009, pp. 183–192.
- [12] E. Alégroth, A. Karlsson, and A. 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, 2018, pp. 172–181.
- [13] E. Alégroth, M. Nass, and H. H. Olsson, "Jautomate: A tool for system- and acceptance-test automation," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 439–446.
- [14] V. Garousi, W. Afzal, A. Çağlar, İ. B. Işık, B. Baydan, S. Çaylak, A. Z. Boyraz, B. Yolaçan, and K. Herkiloğlu, "Visual gui testing in practice: An extended industrial case study," *arXiv preprint arXiv:2005.09303*, 2020.
- [15] F. Dobslaw, R. Feldt, D. Michaëlsson, P. Haar, F. G. de Oliveira Neto, and R. Torkar, "Estimating return on investment for gui test automation frameworks," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 271–282.
- [16] E. Borjesson and R. Feldt, "Automated system testing using visual gui testing tools: A comparative study in industry," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE, 2012, pp. 350–359.
- [17] E. Alégroth, Z. Gao, R. Oliveira, and A. Memon, "Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.
- [18] R. Molari and M. Conway, "Guidelines for testing and release procedures," *NASA technical report service*, 1984.
- [19] T. Hellmann, E. Moazzen, A. Sharma, M. Akbar, J. Sillito, F. Maurer *et al.*, "An exploratory study of automated gui testing: Goals, issues, and best practices," University of Calgary, Tech. Rep., 2014.
- [20] E. Alégroth, M. Steiner, and A. Martini, "Exploring the presence of technical debt in industrial gui-based testware: A case study," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 257–262.
- [21] J. Visser, "Sig/tüvit evaluation criteria trusted product maintainability: Guidance for producers," *Software Improvement Group, Tech. Rep.*, p. 7, 2015.
- [22] A. Qusef, G. Bavota, R. Oliveto, A. De Lucia, and D. Binkley, "Scotch: Test-to-code traceability using slicing and conceptual coupling," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 63–72.
- [23] E. Alégroth, R. Feldt, and H. H. Olsson, "Transitioning manual system test suites to automated testing: An industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 2013, pp. 56–65.
- [24] P. S. Kochhar, T. F. Bissyandé, D. Lo, and L. Jiang, "An empirical study of adoption of software testing in open source projects," in *2013 13th International Conference on Quality Software*. IEEE, 2013, pp. 103–112.
- [25] S. Owen, P. Brereton, and D. Budgen, "Protocol analysis: a neglected practice," *Communications of the ACM*, vol. 49, no. 2, pp. 117–122, 2006.
- [26] F. Khomh and Y.-G. Guéhéneuc, "Do design patterns impact software quality positively?" in *2008 12th European Conference on Software Maintenance and Reengineering*. IEEE, 2008, pp. 274–278.
- [27] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, p. 131, 2009.