Empir Software Eng (2017) 22:2937–2971
DOI 10.1007/s10664-016-9497-6

On the long-term use of visual gui testing in industrial practice: a case study

Emil Alégroth[1] · Robert Feldt[2]

Published online: 24 January 2017

Abstract Visual GUI Testing (VGT) is a tool-driven technique for automated GUI-based testing that uses image recognition to interact with and assert the correctness of the behavior of a system through its GUI as it is shown to the user. The technique's applicability, e.g. defect-finding ability, and feasibility, e.g. time to positive return on investment, have been shown through empirical studies in industrial practice. However, there is a lack of studies that evaluate the usefulness and challenges associated with VGT when used long-term (years) in industrial practice. This paper evaluates how VGT was adopted, applied and why it was abandoned at the music streaming application development company, Spotify, after several years of use. A qualitative study with two workshops and five well chosen employees is performed at the company, supported by a survey, which is analyzed with a grounded theory approach to answer the study's three research questions. The interviews provide insights into the challenges, problems and limitations, but also benefits, that Spotify experienced during the adoption and use of VGT. However, due to the technique's drawbacks, VGT has been abandoned for a new technique/framework, simply called the Test interface. The Test interface is considered more robust and flexible for Spotify's needs but has several drawbacks, including that it does not test the actual GUI as shown to the user like VGT does. From the study's results it is concluded that VGT can be used long-term in industrial practice but it requires organizational change as well as engineering best practices to be beneficial. Through synthesis of the study's results, and results from previous work, a set of guidelines are presented that aim to aid practitioners to adopt and use VGT in industrial practice. However, due to the abandonment of the technique, future research is required

Communicated by: Saurabh Sinha

✉ Emil Alégroth
emil.alegroth@bth.se

[1] Department of Computer Science and Engineering, Blekinge Institute of Technology, SE-371 79, Karlskrona, Sweden

[2] Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96, Gothenburg, Sweden

to analyze in what types of projects the technique is, and is not, long-term viable. To this end, we also present Spotify's Test interface solution for automated GUI-based testing and conclude that it has its own benefits and drawbacks.

Keywords Automated testing · Visual GUI testing · Guidelines · Industrial · Case study

1 Introduction

Automated testing has become a de facto standard in software engineering practice, most commonly performed with automated unit tests (Olan 2003). Unit testing is performed on a low level of system abstraction to verify that software components adhere to the system under test's (SUT) low level requirements. But unit testing is rarely enough on its own for automated testing in industrial practice; companies also need to continuously verify high-level system properties. The reason is because market advantage is determined by a product's time-to-market in many software engineering domains, which has resulted in a trend that software needs to be developed, tested and delivered as often and quickly as possible. Companies therefore want to get human testers out of the loop and automate testing on many levels of system abstraction to reduce costs and increase test frequency. However, the support for automated testing on the GUI level, i.e. the highest level of abstraction, is limited and companies are therefore forced to complement their low-level, automated test activities with costly manual GUI-based testing (Grechanik et al. 2009a, b; Finsterwalder 2001).

Previous work has classified the currently available GUI-level test automation techniques into three chronologically delineated generations (Alégroth et al. 2015; Leotta et al. 2016). These generations are defined based on the approaches they use to stimulate and assert the SUT, which also gives the techniques different benefits and drawbacks. The 1^{st} generation relies on exact screen coordinates, the 2^{nd} on access to the SUT's GUI library or hooks into the SUT and the 3^{rd} on image recognition. Whilst the 1^{st} generation lacks robustness and is rarely used anymore, the 2^{nd} generation is commonly used in industrial practice with tool's such as Selenium (Holmes and Kellogg 2006) and QTP (Lalwani et al. 2013). However, the 3^{rd} generation, also referred to as Visual GUI testing (VGT) (Alégroth 2013), is currently emerging in industrial practice with tools such as Sikuli (Yeh et al. 2009), JAutomate (Alégroth et al. 2013b) and EggPlant (TestPlant 2013). Academic research has shown VGT's applicability and provided support for its feasibility in practice (Alégroth 2013). However, knowledge from the perspective of long-term use, i.e. use over several years, and what challenges that are associated with the long-term use of the technique, are still missing. The key reason for this lack of knowledge is because VGT has only recently gained a foothold in industrial practice and the number of early adopters that have used the technique for a longer period of time, which can be studied, are therefore few.

In fact, studies on the long-term usage of state-of-art or research-oriented ideas/solutions in industrial practice are generally missing in software engineering (Höfer and Tichy 2007). This lack of studies can be contributed to several factors. For instance, such research requires years of investment by a case company, which they are reluctant to invest in areas that have not already been proven to provide positive return on investment. Additional factors include that processes and organizations change over time which often causes resources for research to be diverted to development. Further, key individuals, e.g. champions, sometimes leave the case companies that cause the research to be abandoned or replaced before the long-term perspective can be analyzed. Consequently making research studies on the long-term effects of a new technique, or solution, difficult to achieve. Still, these studies are important to identify impediments with the research to improve its efficacy, efficiency and longevity in practice.

In this paper we help to bridge the gap in knowledge about the long-term use of VGT in practice through a single, embedded, case study (Runeson et al. 2012) at the Swedish music application development company Spotify. Spotify is a good candidate for our research goal since they have used VGT for several years. As such, they are also one of few companies that can provide insights into the entire VGT life-cycle; from adoption to use to long-term use, actually even until abandonment, in industrial practice. Here, long-term use also includes the feasibility of maintenance and challenges associated with the long-term use of VGT and VGT scripts.

Spotify develops music streaming applications for a large number of different platforms and operating systems. These applications are continuously updated with new features and functionality that requires frequent regression testing of the application. This regression testing is facilitated by a mature test process which includes automated testing from low-level unit testing to integration testing to GUI-level system testing performed with multiple approaches one of which is VGT. This company thereby gives us the opportunity to study many aspects of VGT, its alternatives as well as the development and test context in which it is used.

The study was performed with four (4) interviews with five (5) employees at the company that were carefully chosen to provide a representative view of how automated testing and VGT is used at Spotify. These interviews were chosen through snowball sampling (Kendall et al. 2008) and analyzed with Grounded Theory analysis (Glaser and Strauss 2009; Carver 2007). The interview results were complemented by results from two workshops, a survey and finally a review of the manuscript, performed by Spotify employees. Together, the results of the study give a rich overview of Spotify's context and answers *(1) how Spotify adopted VGT, (2) what benefits and (3) challenges the company has experienced with the technique, (4) and finally what alternative techniques the company use for automated GUI-based testing.*

Results of the study show that VGT can be used long-term with several benefits over other test techniques. But there are also many challenges that require both organizational, architectural and process considerations for the technique to be feasible. Because of these challenges, VGT has now been abandoned in most projects at Spotify in favor of a 2^{nd} generation technique, referred to as the Test interface, tailored to Spotify's needs. The paper reports the benefits and challenges of the Test interface and also compares its approach to VGT based on a set of different properties, including robustness, maintenance costs and flexibility.

The acquired results, together with previous work (Alégroth et al. 2013a; Alégroth et al. 2014), were then synthesized to create a set of practitioner guidelines for the adoption and use of VGT in practice. These guidelines serve to provide practitioners with decision support for VGT adoption and to prevent practitioners from falling into the pitfalls associated with VGT.

Consequently, this study contributes to the limited body of knowledge on VGT with evidence regarding the technique's long-term applicability as well as much needed practitioner guidelines for adoption, use and long-term use of the technique. In addition, the study provides a concrete, yet limited, contribution to the body of knowledge on software engineering regarding the long-term use of an academically defined test technique.

The continuation of this paper is structured as follows. Section 2 presents related work followed in Section 3 of a description of the research process. Section 4 then presents the research result, followed by practitioner guidelines for VGT adoption and use in Section 5. The results are then discussed in Section 6 followed by the paper's conclusion in Section 7.

2 Related Work

VGT is a tool driven automated test technique where image recognition is used to interact with, and assert, a system's behavior through its pictorial GUI as it is shown to the user in user-emulated system or acceptance tests (Alégroth 2013). Previous empirical work has shown the technique's applicability in practice with improved test frequency compared to manual tests, equal or even greater defect finding ability compared to manual tests, ability to identify infrequent and non-deterministic defects, etc (Borjesson and Feldt 2012; Alégroth et al. 2013a; Alégroth et al. 2015). A more recently published study has also provided support regarding the feasibility and maintenance costs associated with the technique. For instance, the study shows that maintenance costs of frequent maintenance is lower than infrequent maintenance, positive return of investment can be achieved in reasonable time, the technique can feasibly be used over months at a time, etc (Alégroth et al. 2016a). In this previous study, quantitative results from maintenance of 22 industrial grade VGT test cases are acquired, from an industrial case study, which provide detailed insights into the unique challenges associated with maintenance of VGT script logic and images over time. The study concludes that images are easier to maintain than logic and that required maintenance of logic and images in the same script is the most expensive. Modularized script design, which supports simultaneous maintenance of several scripts at once, is therefore presented as beneficial, a conclusion also supported by other more recent work (Alégroth et al. 2016b). However, the study is partially driven by academic experts and therefore does not provide results from long-term use of VGT in actual industrial practice.

The body of knowledge on VGT also presents several challenges, problems and limitations with the use of the technique in practice (Alégroth et al. 2014), e.g. lack of script robustness, maintenance related problems, immature tooling and adverse effects caused by the test environment. In particular, tool and script robustness has been discussed as primary concerns. Robustness is therefore discussed also in this paper, where robustness is defined as a tool's or script's ability to execute successfully when it is expected to succeed.

Previous work has determined that there are many commonalities between different VGT tools due to their common core functionality that relies on image recognition (Borjesson and Feldt 2012; Alégroth et al. 2013b). This commonality implies that results captured in one VGT tool can mostly be generalized to other VGT tools, similar to how different unit test frameworks can be generalized between each other. However, a distinction is still made between comparison of the capabilities of these tools and the tools' individual qualities. For instance, Sikuli (Yeh et al. 2009) is an open-source tool that has been reported in previous work to suffer from software defects whilst, for instance, EggPlant (TestPlant 2013) is a commercial, and stable, product. As such, due to VGT's flexibility, different VGT tools' capabilities can be compared indiscriminately but how different VGT tools perform them will differ. Thus, a leading hypothesis, based on previous works (Alégroth et al. 2013a; Alégroth et al. 2016a, b) is that most VGT tool's fair at least as well in practice as Sikuli. However, this is still a subject of required future research.

The body of knowledge on VGT covers many perspectives of its use in industrial practice but it still lacks results regarding the technique's long-term use, benefits, drawbacks and challenges, i.e. results that are important to determine the technique's efficacy (Höfer and Tichy 2007). However, results on long-term use are difficult to acquire due to VGT's immaturity and limited use in practice that also limits the number of case companies that have used the technique for sufficient time, i.e. several years, for this perspective to be analyzed. Thus presenting the need, and importance, of the study reported in this paper.

Related work has evaluated the industrial applicability and feasibility of other GUI-based testing approaches (Holmes and Kellogg 2006; Vizulis and Diebelis 2012). However, the body of knowledge on automated GUI-based testing is lacking empirical results regarding the long-term use of these techniques in practice since most studies only focus on maintenance costs. Maintenance costs that are discussed theoretically and presented through qualitative observations from industrial projects (Karhu et al. 2009; Berner et al. 2005). Some empirical research on maintenance costs have been reported but for open source software (Leotta et al. 2013, 2014; Nguyen et al. 2014) whilst the number of papers that include industrial systems are limited (Sjösten-Andersson and Pareto 2006).

Further, the long-term, empirical, perspective is missing in general in the software engineering body of knowledge, as shown by Höfer and Tichy that conducted a survey of all refereed papers in the Empirical Software Engineering Journal between 1996 and 2006 (Höfer and Tichy 2007). Whilst this study is almost a decade old, there is, to the authors' best knowledge, little support to the contrary of this situation today. In their survey, Höfer and Tichy found that empirical research with practitioners is common in software engineering research but that studies that focus on the long-term perspective were missing. A similar observation was made by Marchenko et al. during a three year study with 9 interviews at Nokia Siemens regarding the long-term use of test-driven development (TDD) (Marchenko et al. 2009).

A more recent systematic literature review by Rafi et al. also identified that there is a lack of research that focuses on the challenges and solutions of automated software testing (Rafi et al. 2012). In the review, 24,706 papers were surveyed but only 25 reports were found with empirical evidence about the benefits and drawbacks of automated testing. As such, more research is required on the challenges, and further the solutions, with automated testing to help advance its adoption and use in practice. (Hellmann et al. 2014; Berner et al. 2005). Further research is also required to explore the challenges and solutions from several perspectives that include process and organizational aspects, e.g. how maintenance is performed by testers in practice, and architecture, e.g. how test cases and test architectures are designed (Berner et al. 2005; Hellmann et al. 2014; Alégroth et al. 2016b).

Consequently, this paper addresses several important gaps in the software engineering body of knowledge by supplying empirical evidence of the long-term use of automated testing as well as what challenges, problems and limitations that are associated with the adoption, use and long-term use of automated testing in practice.

3 Methodology

The methodology used in this study is based on the guidelines for performing empirical case studies in software engineering defined by Runeson and Höst (2009). These guidelines were used to perform a single, embedded, industrial case study (Runeson et al. 2012) at the software application development company Spotify.

Objective: The study had two primary objectives. First, to evaluate the long term use and feasibility of VGT at Spotify, including what benefits and challenges the company had experienced during the adoption and use of the technique. Second, to evaluate the alternative automated techniques used at Spotify for GUI-based testing.

Unit of Analysis: As such, the unit of analysis in the study was Spotify's test process with focus on their test automation.

Research questions: The research objectives were broken down into the following research questions:

RQ1: What factors should be considered when adopting VGT in industrial practice?

RQ2: What are the benefits associated with the short and long-term use of VGT for automated GUI-based testing in practice?

RQ3: What are the challenges associated with short and long-term use of VGT for automated GUI-based testing in practice?

In addition to the above stated research questions, the study also investigated some other solutions to automated testing used by Spotify. The results of these investigations will be presented separately and aims to provide a comparison to the benefits and challenges of the company's VGT solution.

3.1 Case Company: Spotify

Spotify is an international software application development company that develops and maintains a product line of applications on desktop and mobile devices that stream music. The company's organization consists of 80 loosely coupled development teams located in Gothenburg, Stockholm and New York. Each team is called an autonomous Squad that consists of maximum ten (10) cross-functional team members (Kniberg and Ivarsson 2012). Cross-functionality is needed at Spotify because each Squad is responsible for the complete development of a feature that is either suggested from management or by the Squads themselves. Each Squad is therefore referred to as its own start-up company that gets to choose what practices, processes and tools they use. A set of standard processes, e.g. Scrum, practices, e.g. Daily Stand-up meeting, and tools, e.g. Sikuli (Yeh et al. 2009), are proposed but teams choose if they wish to use them. The Squads are also grouped into Tribes in related competence areas, for instance the music player, backend functionality, etc. In a Tribe, a squad can still operate with a high degree of independence but is supposed to collaborate with other Squads in the Tribe to implement a certain function or feature, supported by a Tribe lead who is responsible for ensuring a good environment within the Tribe.

Finally, Spotify has Chapters and Guilds which are familiarities (Kniberg and Ivarsson 2012) or interest groups with developers and testers across the different squads. Each developer or tester is associated with a Chapter dependent on his/her skills and competence area within a Tribe. In turn, Guilds are interest groups with specific topics that anyone can create, for instance on the topic of automated GUI-based testing, and anyone interested in the topic can join. A visualization of the organizational structure is shown in Fig. 1.

The company's core application is divided into a front-end and a back-end where the front-end refers to the GUI and features the user interacts with. Back-end development instead refers to server development and maintenance, i.e. how to stream audio in a scalable manner to the application's millions of users in real-time.

Spotify's test process includes several automated test activities that are well integrated into the company's organization. One reason for the company's successes with automation is the individual team structure that makes each team responsible for ensuring that each new feature is covered by tests, which endorses collaboration between testers and developers. This collaboration is also required for the company's model-based testing with the open

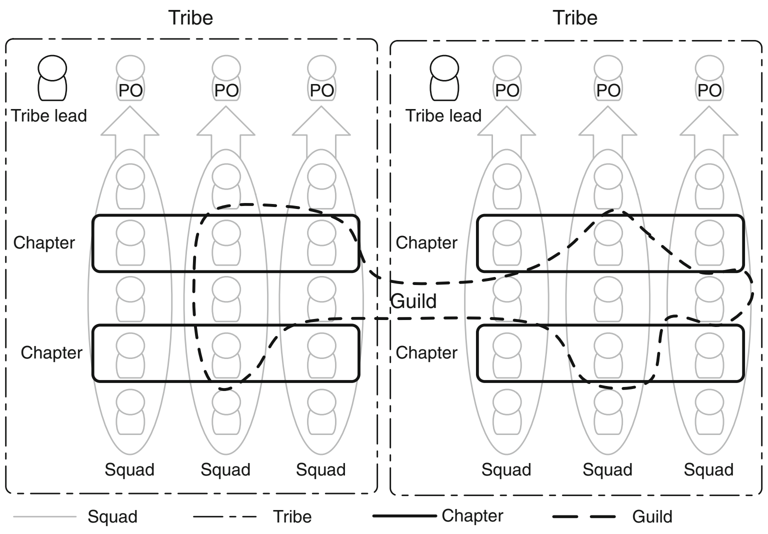

Fig. 1 Visualization of the organizational structure used at Spotify(Kniberg and Ivarsson 2012)

source tool Graphwalker (Olsson and Karl 2015) that in some projects requires test code to be embedded in the application's source code.

The automated tests are used for continuous integration but they are not executed on commit, instead the tests are executed automatically from a build server according to a predefined schedule. However, no code is allowed to be committed before it has been tested with, for instance, unit, integration or GUI-based tests. These tests are executed on virtual machines to improve test execution time and are designed to be mutually exclusive to enable them to be executed out of order and independently of other tests. This practice has been identified as a best practice for VGT in previous work (Alégroth et al. 2013a) and also allows Spotify to run subsets of tests to test a specific part of the application.

However, the main purpose of the automated testing is to mitigate repetitive manual testing to allow the manual testers to focus on exploratory testing. Spotify's test process thereby relies on tight collaboration between developers and different types of testers, supported by a plethora of well integrated test techniques. This process allows the company to release a high-quality application every three weeks on the company's website, Android market or AppStore. However, as stated, but worth noting again, is that due to the company's organizational structure, the test practices and test tools differ between different development teams.

Consequently, Spotify is a highly flexible company with a mixed organization where a multitude of processes and practices are used. The company is therefore representative of both small and medium sized software development companies that use agile processes and practices to develop applications for desktop or mobile devices. They therefore offer a unique opportunity to study the many factors that lead to the successful long-term use of a test automation technique such as VGT.

3.2 Research Design

The case study was performed in four (4) steps as shown in Fig. 2.

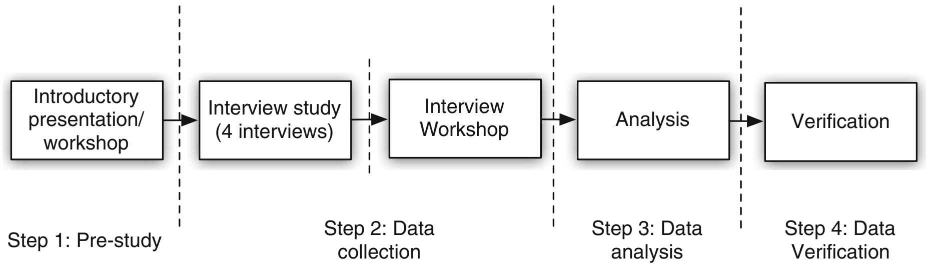

Fig. 2 Visualization of the research design

Step 1: Pre-study: The first step was a pre-study where an introductory workshop was held at Spotify to elicit information about the company's use of VGT, its context, organization, and willingness to participate in the study. The workshop was held at Spotify's Gothenburg office but was observed over video-link by a group of developers and testers at the office in Stockholm, roughly 50 individuals in total. This workshop began with a 40 minute presentation on state-of-art in testing and VGT, followed by a group discussion (semi-structured interview) with testers, developers and managers at the Gothenburg office. The workshop served to transfer some of the nomenclature that would later be used during interviews and to explain the study's research objectives. Thereby, aligning the wording for different types of testing used by Spotify's personnel and the researchers to mitigate conceptual misunderstandings during the study. Additionally, the workshop participants were used as a seed for snowball sampling (Kendall et al. 2008) to identify suitable interviewees for the study. Snowball sampling was chosen because it is an efficient way to find a specific subset of a population by having members of the population recruit their peers, in this case developers and testers that within Spotify have worked with VGT. The workshop was selected for the seeding since it included a large portion of the development staff at Spotify which enabled a widespread identification of interviewees already to start with. Hence, interviewees that could provide representative answers regarding how Spotify works with automated testing, and VGT, to answer the study's research questions. Further snowball sampling was later used during each interview to identify the most suitable individuals to interview given the study's limited time and budget constraints. The samplings resulted in six (6) individuals, all proposed by Spotify, out of which five (5) were interviewed. Triangulation of the sampled individuals names showed that most of the individuals were recommended at all sampling instances, from the initial workshop and after each interview. This indicates, despite these individuals being at different locations in Sweden, that they were somehow champions, or "goto" people, for VGT. Thus making these individuals part of a core set of people most knowledgable about VGT at Spotify and therefore the most suitable people to interview to answer the study's research questions. Further, the sampled individuals had different roles, e.g. testers and test managers,[1] and worked in different projects with different platforms, e.g. desktop and iOS. As such, they could provide a representative view of how VGT and GUI test automation is performed at

[1]The ratios between roles, gender, or experience of the employees cannot be disclosed without breaking anonymity agreements with the interviewees.

Spotify. Worth noting is that sampling was done only based on VGT knowledge, the fact that we acquired a broad spread of roles, and individuals from different projects, was a happy coincidence since all other properties of the final sample were decided by convenience sampling.

Step 2: *Data collection*: The second step of the study served to collect the study's results and was divided into two parts. Part one consisted of semi-structured interviews with the sampled individuals. These interviews gave insight to the success and failure of VGT's use in different contexts, e.g. for different variants of the application developed with different practices and processes, and gave a broad view of what factors that affect the long-term use of VGT in industrial practice.

Four (4) interviews were conducted, where the first interview was held in person with two individuals in a 90 minute session followed by three (3) interviews that were performed over video-link in 60 minute sessions. All interviews followed an interview protocol with 20 questions (See Appendix A) divided into four categories related to the adoption (RQ1-2), use (RQ2-3), maintenance (RQ2-3) and abandonment of VGT in some projects at Spotify (RQ2). The abandonment was studied to identify its cause and to acquire information about the alternative GUI-test approach, i.e. "the Test interface", which replaced VGT. All interviews were recorded and then transcribed prior to analysis.

The interview questions were selected from an initial interview protocol of 36 interview questions that was developed prior to the study. However, due to the interviews' time constraints, the interview protocol was scaled down to 20 questions in a review after the pre-study had been performed.

The second part of step 2 was a 180 minute workshop with one of the lead testers at Spotify responsible for much of the adoption of current test automation tools and practices at the company. This workshop served to verify previously gathered results and to acquire information about the current and future use of VGT at the company, in particular how to combine the Test interface with VGT for future use. In total, the results presented in this paper were extracted from 8 and a half hours (510 minutes) of interview recordings.

Step 3: *Data analysis*: The analysis was performed with a Grounded Theory approach (Glaser and Strauss 2009; Carver 2007) where the qualitative interview results were quantified through open coding (Saldaña 2012). A visualization of the presented coding procedure can be found in Fig. 3. The coding was performed in the TAMSAnalyzer tool (Weinstein 2002), which is an open source research tool where the user can define codes and relations to sub-codes. A total of 40 codes were used in the analysis, nine (9) primary codes, presented in Table 1, and 31 additional and secondary codes. These codes were defined either before or during the coding procedure to tag specific statements that could support the study's research questions. For instance, to capture statements about Spotify's test tools, the codes Graphwalker, Sikuli, TestAutomation and TestInterface were defined. The large set of codes enabled deeper analysis if required, e.g. by combining codes to search for specific statements in the TAMSAnalyzer tool, and was required to saturate the interview transcripts with codes (Wohlin and Aurum 2014).

Coding was performed by going through the interviews and assigning codes to individual statements (Step 1 of Fig. 3). A statement could be given more than one code if it was assumed important for several concepts. For instance, the statement; *"It (test automation) can of course be more integrated (in the process)... Test automation is still quite new"*,

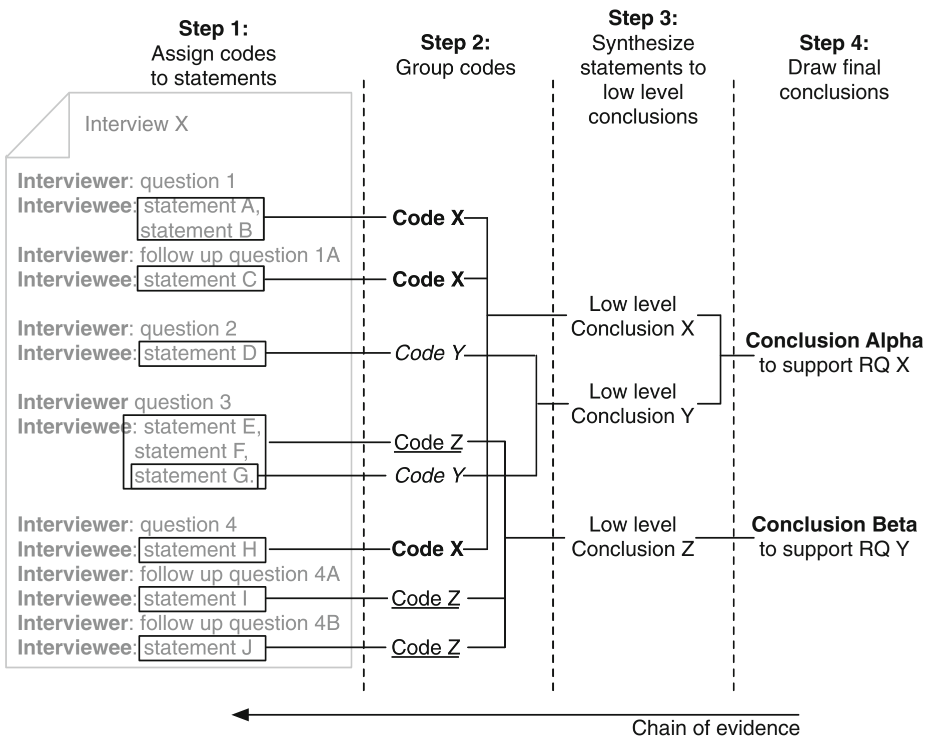

Fig. 3 Visualization of the coding procedure used during the analysis, where (1) codes were assigned to statements, (2) codes were grouped, (3) groups were synthesized to draw low level conclusions from which (4) final conclusions were drawn. Thus, ensuring a clear chain of evidence from final conclusion to statements given by interviewees. **RQ** - Research question

was tagged with both the "Process" and "TestAutomation" codes. In cases where a larger statement was tagged with an instance of a code, sub-statements in said statement could still be tagged with more specific code instances. Using this approach, 475 instances of the 40 codes were administered in the four transcribed interviews. However, these 475 instances do not equal 475 unique statements since some statements were assigned with more than one code.

After tagging the interviews, each code instance was analyzed to synthesize groups of statements connected to individual code tags (Step 2 of Fig. 3) that were then analyzed further to draw low level conclusions (Step 3 of Fig. 3). Nine (9) codes, the primary codes, were analyzed more rigorously. These codes were related to use of VGT with Sikuli, manual and automated testing, the organization for the test automation and observed benefits and drawbacks of the different test approaches. Table 2 summarizes the primary codes and also shows how many times each code tag was instanced and associated with a statement in each interview, and in total in all interviews, during the analysis. Remaining code instances, e.g. "DevelopmentProcess" were analyzed less rigorously since they did not provide direct support for the study's research questions. However, statements associated with these remaining codes were used to put the main conclusions into context and to define Spotify's processes, organization, etc.

Table 1 Summary of the nine primary codes and what types of statements were associated with each code during the interview analysis

#	Code	Description
1	DevelopmentProcess	Statements related to Spotify's development process that influenced the testing
2	Graphwalker	Statements about the model-based testing tool Graphwalker that was combined with Sikuli
3	Organization	Statements relating to Spotify's organization and how it supports their test process
4	Problems	Statements about challenges, problems and limitations with Spotify's automated GUI-based testing
5	Process	Statements about Spotify's overall process from requirements engineering to testing
6	Sikuli	Statements about the Sikuli tool, its use, benefits and drawbacks
7	TestAutomation	Statements about the automated testing performed at Spotify, including processes, tools, etc.
8	TestInterface	Statements about Spotify's 2^{nd} Generation test tool the "Test interface"
9	TestProcess	Statements about Spotify's general test process, including manual and automated practices

The nine (9) primary codes were connected to 204 statements from the five interviews. Rigorous analysis of these 204 statements resulted in 93 low level conclusions (Step 3 of Fig. 3), which were in turn grouped to formulate the study's main conclusions (Step 4 of Fig. 3). Consequently, since there were only 93 low level conclusions, each conclusion was supported by at least one (1) up to eight (8) unique statements, allowing for proper triangulation of the conclusions. This analysis was stored in an excel file to preserve the traceability and the main conclusions chain of evidence (Runeson and Höst 2009). Once the main conclusions were drawn, they were connected to answer the study's research questions based on relevance for each question. Relevance was identified through a combination of logical reasoning, qualitative analysis and analysis of which codes/concepts each conclusion was

Table 2 Summary of the main tags used during synthesis and the quantities of each tag. **Int**. - Interviewee

	Code	Interviewee 1	Interviewee 2	Interviewee 3	Interviewee 4	Sum:
1	Sikuli/Drawbacks	15	10	13	1	39
2	Test_interface/Drawbacks	4	4	8	13	29
3	Organization	8	2	3	11	24
4	Test_interface/Benefits	6	3	4	7	20
5	Sikuli/Benefits	4	6	8	1	19
6	Sikuli/Adoption	2	5	6	0	13
6	Graphwalker	7	4	1	0	12
7	Sikuli/Abandonment	2	2	3	0	7
8	Testprocess/Manual_testing	2	2	2	0	6
	Sum:	50	38	48	33	169

connected to. Additionally, several hundred more statements were used to define contextual information to the study's main results.

This analysis procedure was inspired by previous Grounded Theory research based on interviews, for instance Marchenko et al. (Marchenko et al. 2009) that used a similar analysis to evaluate the long-term use of TDD at Nokia Siemens.

Step 4: *Result verification*: Once the analysis had been completed, a draft paper was composed and sent to Spotify for review. This review was posed originally as a requirement from Spotify to allow their legal department to verify that no classified information was being published. The draft paper was also sent to the interviewed employees who helped proof-read the paper and sort out misunderstandings. As such, the review helped improve the internal validity and consistency of the results as well as the generalizability of the results for Spotify's organization. The latter facilitated by the legal department's independent analysis of the presented results and conclusions.

In summary This study consists of an embedded, qualitative, case study (Runeson et al. 2012) performed, primarily, through interviews with highly knowledgable, well chosen, interview subjects from one of very few companies that can provide actual support for the long-term, industrial, usability of VGT. The study focuses on acquiring results that showcases how Spotify adopted and used the technique but also what benefits and drawbacks they observed. Some quantitative results are presented but the study suffers from a lack of quantitative data from objective analysis of the test suites, models and tools used by Spotify. Such analysis was out of scope for this study due to resource constraints and could be considered negative towards the validity of the results, further discussed in Section 6.1.

4 Results and Analysis

This section will present the results of the synthesis divided according to the study's research questions. Quotes from the interviews have been added in the text to enrich the results where all quotes are written as: *"Italic text surrounded by quotation marks"*.

4.1 Results for RQ1: VGT Adoption

VGT, with the open-source tool Sikuli, was adopted at Spotify in 2011 because of a need for more automated testing of the company's developed applications. Initially the plan had been to add interfaces in the applications to support a myriad of different test frameworks. However, due to cost constraints these interfaces could not be achieved. *"We had to create a Test interface and knew from the beginning what it should look like. What we did not have, to solve the problem, was resources and possibility to dedicate time to create requirements for the development team (to implement the interfaces)"*. Thus, VGT became the only option for Spotify since their application lacked the prerequisites required by most other GUI-based test frameworks. The reason why Sikuli was chosen over other available VGT tools was because one of the company's developers had tried it previously and could recommended it. *"We were looking for ways to solve the problem (with automated testing), and it was a developer here at Spotify...that had previously tested Sikuli... that was why it (Sikuli) came to be"*. Additionally, Sikuli has a Java API that conveniently fulfilled the requirements that Spotify had on the VGT solution to work within the test environment.

The adoption process began with the development of a proof of concept where Sikuli was combined with the model-based testing (MBT) tool Graphwalker (Olsson and Karl 2015). Graphwalker had previously been used at Spotify for MBT because one of its developers worked for Spotify and it was therefore available for their unlimited use. The tool allows the user to create state-based models that can be exported as executable Java programs. Initially all states in the models are empty and therefore require the user to write code that allows the model code to interact with the SUT, e.g. through technical interfaces or by using image recognition technology. These interactions drive transitions between different states in the model that, in this context, represent the GUI state in the SUT. Further, because Graphwalker models are Java, it became natural for Spotify to adopt Sikuli's Java API in favor of its Python API. *"No, it (the Python API) did not map at all against what we wanted to do, we wanted it in Java"*. Hence, whilst the Python API could have been used, the Java API was simply a better fit. Consequently, the technical design solution was, for the most part, based on software that was conveniently available to the company at the time.

However, the first proof of concept solution was poorly implemented, created by a single developer, as one big script. It was only later that VGT became useful after the script had been broken down into small, reusable, modules through the use of engineering best practices. The use of engineering best practices was one reason for VGT's successful adoption at Spotify but the main reason was because the adoption was performed by a tight team of engineers that were dedicated to making VGT work. *"A lot of the success relied on engineering solutions and communication because we were developers from both Gothenburg and Stockholm who also worked in different teams with different features. However, we were probably the tightest group at Spotify because we were adement to develop it (VGT) and continuously make it better"*. Communication during the adoption process served to spread knowledge of best practices and to share reusable components among the adoption team. Additionally, test scripts were shared and reviewed to ensure their quality. As such, the team incrementally built up a working, context dependent, test environment at Spotify that interweaved their own development practices, test processes and usage of reusable test artifacts. Similar results, and needs for incremental adoption, have been reported in previous work on VGT (Alégroth et al. 2013a; Alégroth et al. 2014), implying that there are contextual factors that place a need for incremental, flexible, adoption of the technique.

No explicit changes were made to the tools or the Sikuli API during the adoption but additional help methods were developed and most of the scripts followed the page view pattern (Isaacson 2004). Thus, scripts were designed such that individual scripts or parts of scripts dealt only with a single view of the application. *"That we had (Additional methods). We had our own classes with help methods,...I think they were developed straight from the user API"*. The use of these patterns, and a structured approach to the script development, is supported by previous work (Alégroth et al. 2013a; Berner et al. 2005) but also more recent work that has found that automated GUI-based testing suffers from technical debt (Alégroth et al. 2016b). In this recent work, performed at a avionics company, it is concluded that GUI-based testware shares the same requirements as software when it comes to architectural design, which includes development patterns. However, since VGT test cases operate on a system- or acceptance test level of abstraction they are tied to the feature requirements of the SUT. This dependence on the requirements implies that many design choices become contextual and patterns must therefore be tested prior to adoption, preferably in an incremental manner. Regardless, this study is the first, to the authors' best knowledge, that names a specific pattern suitable for VGT scripts.

A core challenge during the adoption was how to set up VGT in the test environment to automatically support continuous testing. *"In the beginning you had to install Sikuli to run it...That was a problem in the beginning when we wanted to set it up on machines for nightly test runs"*. This is a challenge since the test execution time for VGT scripts, regardless of tool, is high, implying a need for parallelized execution. However, Sikuli takes over the mouse cursor and keyboard during test execution which means that only one test script can be executed on one machine at the time, a challenge we'll return to in Section 4.3. To get around this challenge, in a cost-effective manner without buying a lot of hardware, Spotify started running Sikuli on virtual machines (VMs), a solution reported also in previous work (Alégroth et al. 2013a). This solved the execution problem but it was still problematic to install everything, Java, Sikuli, etc., on each VM. *"If you have Java on the machine the tests should run. Then it is just important to package our test-jars, the jar that contained the tests, and that it includes all resources that were required (to run the tests)"*. These resources included the Graphwalker Java models, the Sikuli Java API and Sikuli scripts used to interact with the application. By packaging these resources into VM images they could more easily be installed on the VMs and thereby solve two core, and dependent, challenges with Sikuli-based VGT, i.e. long execution times and lack of support for parallelized test execution. Additionally, this allowed Spotify to reuse the images for different versions and variants of the application.

In summary we conclude that for VGT adoption to be successful it may require:

1. An incremental adoption process,
2. Good engineering practices, e.g. patterns, modularization and help classes,
3. A dedicated adoption team with good communication, and
4. Virtual environments to run the tests on.

4.2 Results for RQ2: VGT Benefits

The first observed benefit was the robustness of the Sikuli tool. *"We have actually not had any stability problems with Sikuli as such, it is actually really good"*. Stability in this case refers to the tool's robustness, as defined in Section 2, which in previous work has been reported to be a challenge with the Sikuli tool (Alégroth et al. 2013a) manifesting through unpredictable behavior and unexpected failures. *"Over a whole day, 24 hours, maybe 10.000 pass (Image recognition sweeps) and maybe 8 that fail"*. However, despite the high level of robustness the tool was still reported in two interviews not to be robust enough. *"It depends on the purpose of the tests, if you want to run tests that always go green and pass, then it can be a problem, even if it (failures) happens only then and again"*. *"If you look at it from a positive point of view there was a lot that worked but what failed was in a way annoying enough."*.

The interviewees' perceptions of robustness could however have been affected by the application's high frequency of change, which also required frequent maintenance of the test scripts. For the desktop application this maintenance was considered feasible but for the mobile platforms it was a large issue. *"In our case (Mobile applications) it was not feasible. At one point all I did was to update the images for the Facebook scenario. The desktop application worked better because it was more stable (to change) so there it (the scripts) worked over a longer period of time"*. Stable to change, in the stated context, refers to the frequency with which the user functionality or GUI changed over time that also required maintenance of the VGT scripts. Hence, these robustness issues limited the amount the VGT scripts were used. *"At most we seem to have had 20 Graphwalker models that used (for the* *most part) Sikuli. We did not use Sikuli for that many views, so I would say that we covered <10 percent of the functionality."*. A statement given after one of Spotify's developers did a brief analysis of the VGT test suites.

However, very few false results had been observed during the use of Sikuli, neither false positives or negatives. False positives were primarily caused by changes to the SUT or due to synchronization problems. *"The main false-positives that we had, they were more in the tests, caused by us not having enough time-outs."*. Further, false negatives were determined to be caused by incomplete scripts rather than challenges with the tool. *"It is possible (That a False-negative was reported), but... no, maybe not. Not because of Sikuli, rather because we didn't test it (the defective state)"*.

Additionally, Sikuli test cases were reported to be reusable between different variants of the application as long as the images used for interaction were updated accordingly. This result implies that maintenance of images can be separated from maintenance of script logic. *"Our fonts are rendered differently between OSX and Windows, we can reuse the tests but we need to change the images"*.

The primary reported benefit was however Sikuli's flexibility of use on any platform regardless of implementation which also made it applicable on production ready software. In addition it allowed the testers to incorporate external applications, e.g. Facebook, into the test scenarios which was required since Spotify supports user login through the user's Facebook account. *"The benefit is that we can use the SUT as a black box. We can use a production grade client, which we have not instrumented or added (test) functionality to. That is... (only) if you can see it you can automate it"*. *"If you want to test things in Facebook, for instance, or kill the app and restart it, ..., then you need to do something outside the app. Then we use Sikuli."*. GUI interaction also made it possible to test the appearance of the GUI, not only its functionality. *"(Sikuli ensures) that you didn't (just) test a button which then turned out to have the wrong color or something like that"*.

Sikuli's Java API was also reported as a benefit since it made it possible to, in a flexible way, code additional functionality into the test cases or the test framework when required. As such, workarounds could be created when conventional use of the API was not enough. *"What (test functionality) is missing we can simply code. You can find workarounds for most things in Sikuli, but it (the test) becomes more complex"*.

Another benefit with the Java API was that it integrated well with Graphwalker (Olsson and Karl 2015) for MBT based VGT. *"Then (for Graphwalker) Sikuli fit well since it provides a Java API...It fit like a hand in a glove so there were absolutely no problems"*. Graphwalker's will be described in more detail in Section 4.4.

Sikuli was also reported as a useful and valuable tool for finding system regression defects, especially during periods when Spotify's client has lacked robustness or has been unstable. Robustness in this case refers to the behavioral correctness of Spotify, as defied in Section 2, whilst stability in this case refers to the stability of the applications requirements, features or graphical appearance, all of which have an impact on the need for VGT script maintenance. However, the robustness problems also resulted in additional maintenance of the test scripts that was not considered feasible at that time. *"Yes, we did (find defects), primarily because our client broke continuously. So the defects we found were often that the client crashed when you entered the artist-view, or similar. In that way it was a positive thing, even if it felt as, or actually was, unfeasible to maintain, it still contributed to the defects in the system being found"*. Many of these defects could have been identified with other automated test techniques but the Sikuli solution, as presented above, also tested the GUI's appearance like manual regression tests. The VGT tests were therefore primarily used for regression testing, using scenario-based test cases defined in models that can either be executed linearly or randomly using one out of several graph traversal algorithms.

In summary we conclude that the benefits with Sikuli, and VGT, are that:

1. Test scripts are robust both in terms of execution and number of false test results,
2. Test script maintenance is considered feasible for desktop applications,
3. Test script logic can be reused between different variants of an application,
4. Test scripts are flexible and can be used to test the actual product as well as incorporate external applications with limited access to the test cases,
5. Sikuli integrates well with Graphwalker for MBT based VGT, and
6. Test scripts find regression defects that otherwise requires manual regression testing.

4.3 Results for RQ3: VGT Challenges

The main drawback with Sikuli reported by Spotify is its limited use for GUIs that present dynamic data, i.e. non-deterministic data from, for instance, a database. Whilst all VGT tools can verify that non-deterministic data is rendered by checking that a GUI transition has occurred, the tools require a specific expected output image to assert if what is being rendered is correct according to an expected result. In Spotify's case this presented a problem since much of the application consists of dynamically rendered lists of songs, artists and albums. Whilst tests could be performed on a stable database, with specific search terms, the company wanted to run the tests in the real production environment where a search for a set of songs does not always return the same list. Thereby impeding Sikuli's usefulness. *"The test data we have includes a lot of songs, albums and artists and such. They have different names, cover arts... it is very hard to verify that it is the correct image for each artists name"*. *"It is difficult to work with them (tests) in Sikuli, they need to scroll (in lists) and it is difficult to distinguish different rows, they look the same. Big buttons are very easy."*. Whilst this problem is reported for Sikuli, it has been recognized as a more general problem with VGT since the image recognition requires a reference image to operate.

Attempts to solve this problem included using Sikuli's optical character recognition algorithm (OCR) and to copy the entire list of songs or artists to the clipboard and then importing the clipboard to the scripts for analysis. However, both solutions were found unreliable/unstable. Copying text often failed because the key commands did not work properly in the application and Sikuli's OCR algorithm was in all interviews were it was mentioned (by three out of five interviewees from three different projects) stated to be unreliable. *"We have verified which songs are in a playlist. We selected all songs, copied them to the clipboard but sometimes this process failed...ctrl A and ctrl C did not work. This is a fault that is not relevant for Spotify"*. *"Then I need to extract that information (song list) dynamically somehow. You could use the OCR functionality in Sikuli, but it is way too unstable"*. Unstable in this context refers to the OCR algorithm's ability to return a correct string. The reason for this result was because of the, at the time, implementation of the Tesseract OCR algorithm (Patel et al. 2012) that, for instance, did not support algorithm training. Algorithm training and a improvements to the OCR algorithm were firstly introduced in 2013 with the release of Sikuli X-1.0rc3 when Tesseract 3 was integrated in the tool. This challenge is also considered general since OCR is a complex problem for which there is currently no completely reliably/robust algorithm available for commercial use.

Another large drawback was the amount of image maintenance that was required. Spotify's application is developed for a myriad of different platforms that all render the GUI slightly differently and have different, operating system specific, images. Thus, each time the GUI was changed, the images for that change had to be maintained in each test suite for each variant of the application. Further, since Spotify interacts with external applications, like Facebook, the test suites also had to be maintained for changes made to software outside Spotify's control, i.e. maintenance that could not be foreseen. *"It could be both (images and test logic), but it was probably most often the images."*. *"Even if we remove everything that has to do with Facebook and what is outside our control, even if we would use Sikuli entirely for our own app, we would have problems (with image maintenace)"*. *"But the problem with Facebook was that they change as much as we do. The difference is that we have no idea when and what they change."*. As such, much of the image maintenance problem came from automation of external software applications, which was also where Sikuli was the most beneficial since there was no other way to access and/or stimulate those applications automatically. This maintenance is primarily required when graphics are changed or removed on the SUT's GUI, minor changes in color or shape are often still acceptable. However, since the image recognition can identify a component regardless of its position on the GUI, addition of new components does not affect the image recognition's success-rate. Hence, care should be taken during the SUT's evolution that the VGT test suites evolve simultaneously. However, due to VGT's flexibility it can be used on almost all types of GUI's, which makes it difficult to generalize what types of GUI changes that are acceptable and which are not. More research is therefore required that aims at identifying patterns that results in script maintenance.

Yet another drawback was that Sikuli was experienced to have limited applicability to test applications on mobile devices. However, several of the interviewees stated that they did not know the current status of the mobile VGT, which, for instance, is support by the VGT tool EggPlant (TestPlant 2013). *"On mobile phones, if you want to run on an actual device, it doesn't work. I havn't check the last year, maybe they have worked on adding such support?"*.

Further, when Sikuli scripts execute, they take over the mouse and keyboard from the user. This implies that the user cannot use the computer at the same time as a script is running. *"It is according to me a problem that has to be solved. It is way to ineffective otherwise. If you write a script that takes fifteen minutes to run, then you don't want to lock up the computer for fifteen minutes, you want to debug other tests at the same time"*. Additionally, Sikuli scripts were perceived to execute slowly, especially for larger test scenarios. *"Yes, I guess it is a problem (Slow test execution). On the other hand the tests are system tests, so the actual problem is maybe not Sikuli's fault"*. This problem was assumed to be solvable by distributing the test execution over several virtual machines (VMs) and/or reference systems with physical devices, i.e. parallel test execution. However, Spotify had experienced problems with running the test cases this way in some cases because Sikuli requires a physical screen to be connected to the computer. This problem originates in Sikuli's use of the AWT Robot framework that only initiates if a physical screen is connected. *"In our test lab we don't have monitors for all machines that we have and there is something strange with that. The desktop tests worked fine but for the mobile application tests something strange happened if it (Sikuli) did not detect a screen, then it was not possible to run Sikuli-stuff."*. Hence, challenges that are specific to Sikuli and not other VGT tools, e.g. EggPlant (TestPlant 2013), but that could be mitigated by using either VM's or remote test execution as presented in previous work (Alégroth et al. 2013a; Alégroth et al. 2016a).

In summary we conclude that the challenges with Sikuli, and VGT, are that:

1. Test scripts have limited use for applications with dynamic/non-determi-nistic output,
2. Test scripts require significant amounts of image maintenance,

3. Sikuli scripts have limited applicability for mobile applications at Spotify, and
4. Sikuli locks up the user's computer during test execution.

These challenges have resulted in Spotify's abandonment of Sikuli in many projects in favor of a 2^{nd} generation test approach that will be discussed in Section 4.4. The main reasons for the abandonment were the high costs associated with maintaining images and the tool's lacking applicability for mobile applications. These challenges were as such the main long-term, continuous, challenges that Spotify experienced with VGT, answering research question 3 (RQ3). Thus, the tool did not fully support Spotify's needs but was still good enough to support the company's needs for automated GUI-based testing for several years. However, it should be noted that, at the time of writing this manuscript, Sikuli is still used at Spotify in less than a handful of projects for the desktop application in combination with other automated and manual test approaches. The number is however dwindling, with more projects transitioning over to the Test interface approach.

Results from the the workshop in phase 3 also indicate that the adoption of VGT at Spotify was instrumental for the current test automation culture at Spotify. Hence, when developers saw the benefits of test automation with VGT, several developers took it upon themselves to do the necessary refactoring required to make additional automation techniques applicable.

4.4 Other Test Automation at Spotify

This section provides an overview of two of the main tools that Spotify use for automated GUI-based testing.

Graphwalker As mentioned, Spotify's automated GUI-based testing is based on a model-based testing framework called Graphwalker. Graphwalker models are created graphically in a tool called yEd (yworks 2016) and then exported to a Graph ML format. These models can then be used to create executable Java class stubs, i.e. empty methods, in which the user defines the interactions with the SUT.

Each model defines a linear test scenario but several models can be linked together through a model on a higher level of abstraction, i.e. a model of models, to create more complex test scenarios. This ability also allows the user to reuse scenarios, e.g. a login scenario, to lower development costs. *"If you have general scenarios (that run) in several views, you can make that scenario into a model and simply switch to it every time... you get an overview model and you can reuse the scenario in other scenarios... It (Graphwalker) also has support for conditions (for branching scenarios), which are usually states. For instance if we have a login scenario, then we set that if Login=True then it knows that state in the model and may not go to another state before the condition is True (in the application)"*. Thus providing some parameter based programming support in the models themselves.

In addition, Graphwalker supports random, real-time, traversal of models for automated random testing, with traversal algorithms such as A* (Nilsson 1980) and random. *"Yes, you get many permutations of the model and a lot of interesting things happen when you run the tests this way (automated traversal). However, we traverse our models with so called on-line generation, which means that we don't generate a path from the model that we then use. Instead we always ask Graphwalker to generate the path in run-time. Mostly we used the random generator, which gives us different permutations. What we want to ensure is that we have full coverage (node coverage) of the model. So, the generators are what makes it* *possible for us to traverse the model in different ways. Stop conditions make it possible to express when we are done. We don't have any stop nodes in the model".*

Test interface As mentioned in Section 4.1, Spotify's intention was originally to add Test interfaces to the application for automated testing. However, the Test interfaces had not been achievable due to resource constrains. When Sikuli was abandoned, the need for automated GUI-based testing once again presented itself but due to the cultural changes in the company, with a greater focus on automated testing, Spotify's initial plan could now be realized.

The solution, simply named the *Test interface* by Spotify's developers, is a 2^{nd} generation GUI based testing approach where hooks (test methods), are embedded into the source code. These methods are designed ad hoc to provide the tester with the state information (s)he needs to write a test to test a certain function or feature of the SUT. Each Test interface is then accessed/used by test case scenarios that are defined in Graphwalker, some reused from previous Sikuli testing. Since the Test interfaces can be embedded on any level of system abstraction below the bitmap GUI it provides the approach with a great deal of flexibility but disallows the developers to use this approach to verify the SUT's end user behavior, discussed further in this section.

Benefits Several benefits, but also drawbacks, were reported with the Test interface solution. The primary benefit is the flexibility the Test interface provides Spotify to perform, for instance, tests with dynamic data, e.g. playlists, and other test objectives not fully supported by Sikuli. *"There is support for anything really, but it requires that you write it (the test support) yourself. Hence, if you, for instance, want swipes then you have to add a method in the interface that actually does that...if you have a tabel, a playlist, where there are songs, then there is for each table, playlist, methods to scroll to an item at an index... you don't do that on the real UI layer".*

Further, the execution speed and robustness of the Test interface is perceived higher than for the Sikuli tests. *"(The benefits are) Time, it is faster with the new (Test interface). Stability (Robustness)".* In addition, the execution time can be improved since the Test interface can modify how the application behaves, e.g. it can remove animations between state transitions. *"The difference (to Sikuli) is that with the Test interface we can remove animations, so when you open "Playing view", the player, then we set the Test interface not to animate but instead render (the output) instantly".* Further, the Test interface runs in the background without locking up the computer, unlike Sikuli, which allows the user to continue work whilst the tests are running. This approach is also more stable since it allows extraction of different types of data directly from the application, which make interactions and assertions more exact. *"The benefit is that you can read the unique identifiers that each song has...I can go to an album and read which songs are there and save them. Then I can go to "Add to your music". Then I can go to "Your music" and assert if they are there (The unique identifiers)".* Thus solving the dynamic data assertion problem that Spotify experienced with Sikuli, presented in Section 4.3 whilst also improving the tests' robustness. *"I would blindly ship this (The application) to employees being sure that 90 percent of the application would work".*

However, the Test interface flexibility and robustness comes at the expense of opening the application up to explicit internal access to its functions and features, which also presents a threat of user misuse. *"We have full insight into the client code...this Test interface is very open, you open up the client to do what you want".* This threat is removed by a test architecture, abstract model shown in Fig. 4, where each loosely coupled Test interface method is

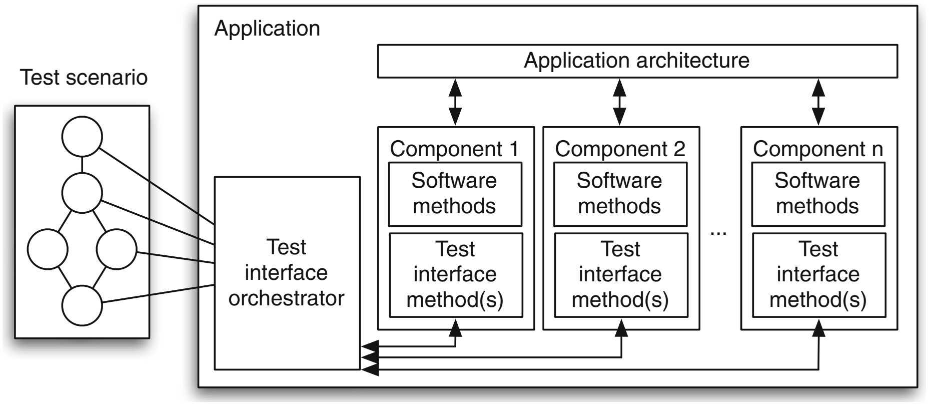

Fig. 4 Visualization of the Test interface architecture within Spotify. All Test interface methods are accessed through a test orchestrator, which connects test case nodes to specific Test interface methods

coordinated by an orchestrator class within the application. Before the product is released, the orchestrator class is removed from the application which turns all the Test interface methods into "dead code", i.e. code that is unreachable in the application. A tool called ProGuard is then applied which removes all dead code, effectively removing all the Test interfaces. *"ProGuard is a tool in which... will remove those endpoints (methods) because they are not in use anymore."*.

Another benefit of this architecture is that if changes are made to the application's source code that breaks the coupling to a Test interface method(s) the developer will receive a compilation error. As such, the developer gets an instant notice when, and what, Test interface methods requires maintenance. *"The plan is that the Test interface is part of the code such that if you change a feature you should get a compilation error... and you never get unstable tests."*.

The transition to the Test interface has required a huge investment and organizational change but is assumed to have lowered test maintenance costs compared to Sikuli. *"Yes, I would claim that (Perceived lower maintenance costs). However, it is difficult to say because we have also improved our process. We have hired people that are dedicated to each platform, before we (Small test team) had to do these parts (Test maintenance)"*. Thus, the lowered maintenance costs are caused by a combination of factors but the previous image maintenance costs have been removed since images are no longer used for interaction and assertion of the application's correctness.

Drawbacks However, this leads to the Test interface first drawback, it does not verify that the rendered, pictorial, GUI is correct. *"We can miss bugs now, for instance... we do not notice if the client is upside down (GUI rendered incorrectly)"*. *"What we miss now when we run the Test interface is the UI part. We see if the functionality works but we don't know if it (the GUI) looks right. We could see that with Sikuli, at least partially"*.

Further, interactions with the application during testing is not performed in the same way as a user interacts with the software. Hence, instead of clicking on components, these interactions are invoked from layers underneath the pictorial GUI. *"But when we build our own interfaces, then we are clicking, in a way, from beneath"*. *"We create events that essentially do the same thing but without the physical click"*. Further, because the Test interface code is removed in the release ready version of Spotify, the tested version of the application is not the same as what is delivered to customers. *"The main drawback, which we knew from the beginning, is that we're not testing the real products, we're testing something else, more or less"*.

Synchronization between the test cases and the SUT was presented as a challenge with Sikuli. However, the same challenge has been observed with the Test interface. *"In that regard it (Synchronization) is the same. It looks reasonably the same independent of if it is Sikuli or the Test interface. You have to solve it in different ways, but the core problem is the same. One part of the challenge with test automation is how to deal with asynchronous test execution"*.

Another common problem for both Sikuli and the Test interface is that none of them actually verifies that Spotify plays music, i.e. auditorial output. *"We have manual testers that go through stuff (e.g. that music is playing), so we capture those things. It is not a big risk that a version of the application reaches the customer without sound"*. In one of the interviews, a solution was discussed to integrate functionality, similar to the popular application (Shazam 2016), which is a mobile application that allows the user to automatically detect the name, band and other information of a track by playing the track to the application. However, at the time of the study no such feature was planned.

In summary we conclude that Spotify use Test interfaces embedded in the source code, driven by a model based testing solution, i.e. Graphwalker, for GUI-based testing with the benefits that:

1. The Test interface provides more flexibility of use to test specific parts of the Spotify application, e.g. lists, than Sikuli,
2. Test interface tests execute quicker and more robustly than Sikuli tests, and
3. Broken test cases are instantly identified by coupling to the software components that generate a compilation error if a Test interface method requires maintenance, ensuring that they are continuously up to date.

However, the Test interface still has drawbacks, for instance that:

1. Test interface test cases do not verify that the pictorial GUI is correct, only the functionality,
2. Test interface interaction with the application differs from human interaction, i.e. interactions are invoked rather than performed through the user's means of interaction,
3. Synchronization between Test interface test cases and the application is still a challenge, similar to Sikuli, and
4. Neither Sikuli or the Test interface are able to verify that the application actually plays music.

4.5 Quantification of the Qualitative Results

The study reported many benefits and challenges with both Sikuli and the Test interface (TI). To get an overview of the qualitative results presented in this work, the properties of the two techniques were quantified. The quantification was done on a five (5) point scale from low (1) to high (5) as shown in Table 3 with the researchers' hypothesized results in column *R*. As input to the quantification, the amount of support for and against each property for both techniques were analyzed from the interview and workshop results. However, we stress that this quantification is only based on data from Spotify's context, which adversely affects the results external validity.

Table 3 Interviewees' perceptions of the VGT solution compared to the Test interface. Each property is ranked on a scale from 1 to 5 where 1 is low and 5 is high. **R** - Researchers, **IPX** - Industrial practitioner X, **Corr. Med.** - Corrected Median (Rounded to Integer)

	R		IP1		IP2		IP3		Corr. Med.	
Property	VGT	TI	VGT	TI	VGT	TI	VGT	TI	VGT	TI
Ease of Graphwalker integration	5	5	5	5	5	5	5	5	5	5
Robustness	3	5	3	5	2	5	3	4	3	5
Frequency of correct test results (No false positives or negatives)	4	4	–	–	4	3	4	3	4	3
Defect finding ability	4	3	5	5	4	5	4	5	4	5
Migratability of tests between SUT variants	3	3	3	4	2	3	3	3	3	3
Maintenance costs of VGT scripts	2	5	4	2	5	3	4	3	4	3
Support for Parallel test exectuion	3	4	1	5	1	5	2	5	2	5
Ease of Synchronization between tests and SUT	3	4	4	2	4	3	4	3	4	3
Speed	2	5	3	4	2	4	3	4	3	4
Flexibility of integration with different plattforms	4	4	3	5	3	5	3	5	3	5
Desktop	5	5	4	5	4	5	4	5	4	5
Android	3	4	1	5	1	5	1	5	1	5
iOS	3	3	1	5	1	5	1	5	1	5
Web	5	3	4	5	4	5	4	5	4	5
Flexility of use for different testing	3	4	2	5	1	5	1	5	2	5
Dynamic data	2	5	2	5	1	5	1	5	2	5
Work in parallell to test execution	2	5	1	5	1	5	1	5	1	5
Test of external software	5	1	5	1	4	1	4	1	5	1
Test of auditory output	1	1	2	2	1	2	1	1	1	2
MBT support	5	5	5	5	5	5	5	5	5	5
Support for different types of tests	5	3	4	4	4	4	4	4	4	4
Acceptance test	5	3	4	4	4	4	4	4	4	4
System test	4	5	3	4	3	5	3	5	3	5
GUI-based testing	5	1	4	3	5	2	5	2	5	2

To verify the correctness of the quantification, a questionnaire survey was then sent to the interviewees, of which three (3) responded, results shown in columns marked *IPX* in Table 3. Additionally, a corrected median was evaluated (Medians rounded to integer values), see column *Corr. Med.* in the table, based on the researchers and the industrial practitioners quantified results. As can be seen, there are some discrepancies between the researchers and industrial practitioners results that may be caused by several factors. For instance, the industrial practitioners had deeper knowledge about the techniques' use in their context than the researchers and it is possible that ambiguities in the qualitative results may have swayed the researchers' perception. Regardless, the estimates are overall close and by using the median overall we are therefore confident that these results reflect a general overview of the differences between the two techniques in contexts similar to Spotify.

The corrected median results were therefore analyzed statistically with the non-parametric Wilcoxon signed-rank test to test if there was any statistical significant difference between the two techniques. This test was used since the data sets were paired on the measured variables (properties) and because normality analysis, with the Shapiro-Wilks normality test, showed that the samples were not normally distributed, with p-values 0.0122 and 3.689e^{-05} for the VGT sample and Test interface sample respectively. The result of the comparison showed that we must reject the null hypothesis, p-value of 0.0419, and therefore we conclude that there is statistical significant difference between the two techniques in terms of their properties. This result was verified with the true median (p-value: 0.006) to ensure that the corrected median did not add any significant bias. Analysis of the statistical power of the two samples also showed that it was 0.924, i.e. of suitable statistical strength, and as such reliable. A deeper discussion about the implication of this result will be presented in Section 6.

5 Guidelines for Adoption and use of VGT in Industrial Practice

In this section we present a set of practitioner oriented guidelines for the adoption, use and long-term use of VGT. These guidelines, summarized in Table 4, were created through synthesis and triangulation of solutions, guidelines, factors, etc., for best practice VGT and automated GUI-based testing presented in previous (Alégroth et al. 2013a; Alégroth et al. 2014) and related work (Hellmann et al. 2014). As such, some of the guidelines are common practice in other disciplines but are here, based on the study's results, confirmed as suitable practices for VGT adoption, use and long-term use as well. The purpose of the guidelines is to provide practitioners with decision making support as well as guidance to avoid common pitfalls with VGT. However, it should be noted that future work is required to expand and fine-tune the guidelines and verify their generalizability, comprehensability and impact in other companies and domains.

5.1 Adoption of VGT in Practice

Manage expectations VGT is associated with high learnability and ease-of-use that makes it tempting to use it for automation of all types of test cases. However, VGT is primarily a regression test technique for system and acceptance tests and is therefore not suitable for testing of immature or frequently changing functionality in the SUT since the maintenance costs of such scripts will be high. Test cases that are developed in early stages of VGT adoption should also be removed after exploring what types of SUT functionality they can test. This latter recommendation being a product of the results acquired in this study.

Another common expectation is that VGT can completely replace manual testing in an organization but this is not the case since VGT scripts can only find defects in system states that are explicitly asserted. In contrast, a human can observe faulty SUT behavior regardless of where or how it manifests on the SUT's GUI and VGT scripts therefore need to be complemented with manual test practices, e.g. exploratory testing (Itkonen and Rautiainen 2005).

Use incremental adoption Large scale adoption is seldom recommended for any new technique or practice, and the same applies to VGT. Instead, VGT should be adopted in a staged/incremental adoption process with one or several pilot projects to evaluate the

Table 4 Summary of guidelines to consider during the adoption, use or long-term use of VGT in industrial practice. Column "Support" shows if this study (A) or which previous work, B (Alégroth et al. 2013a) and C (Alégroth et al. 2014) and related work D (Hellmann et al. 2014) that supports the presented guideline

Phase	#	Guideline	Description	Support
Adoption	1	Manage expectations	It is not suitable/possible to automate anything and everything with VGT, consider what is automated and why?	A, B, C, D
	2	Incremental adoption	A staged adoption process that incrementally evaluates the value of VGT is suitable to minimize cost if the technique is found unsuitable.	A, B, C
	3	Dedicated team	Dedicated teams do not give up easily and identify how/when to use VGT.	A, B, C
	4	Good engineering	VGT costs depend on the architecture of tests/test suites and engineering best practices should therefore be used, e.g. modularization.	A, B, C, D
	5	Software	Different software solutions, e.g. VGT tools and third party software, should be evaluated to find the best solution for the company's needs.	B, C, D
Use	6	Roles	VGT requires training of new roles, which is associated with additional cost.	A, B
	7	Development process	VGT should be integrated into the development process, e.g. definition of done, and the SUT's build process, i.e. automatic execution.	A
	8	Organization	Organizational change disrupts development until new ways of working settle.	B
	9	Code conventions	Code conventions improve script readability and maintainability.	B
	10	Remote test execution	For distributed systems, VGT scripts should be run locally or use VGT tools with built in remote test execution support	B, C
Long-term	11	Frequent maintenance	The test process needs to prevent test cases degradation to keep VGT maintenance costs feasible long-term.	D
	12	Measure	The costs and value of VGT should be measured to identify improvement possibilities, e.g. new ways of writing scripts.	D
	13	Version control	When the number of SUT variants grow, so do the test suites and they should therefore be version controlled to ensure SUT compatibility.	D
	14	Life-cycle	Positive return on investment of VGT adoption occurs after at least one iteration, so how long will the SUT live?	B, C

technique with several different VGT tools. The reason is because these tools have different capabilities that make them more or less suitable based on contextual factors such as the test automation culture of the company, if the system is distributed, if the testers have programming knowledge, etc. We discuss some of these different properties in previous work (Alégroth et al. 2013b). Additionally, the pilot projects should strive to find suitable test cases to automate and take maintenance costs into consideration. These projects therefore need to span over a longer period of time, at least a few months, to be able to evaluate the long-term aspects of the development and maintenance of scripts. Thereby allowing disruptions caused by technical learning or use to reside properly before a decision is taken on the technique's adoption or abandonment.

Use a dedicated team VGT is easy to use but it is associated with many challenges that can stifle a team's progress and lead to developer frustration. A team of dedicated individuals should therefore drive the adoption process such that the technique is not abandoned prematurely, i.e. before all aspects of the technique have been evaluated. This recommendation is once more based on the findings from this work and have previously not been discussed in previous work but is supported by success stories presented in previous work (Alégroth et al. 2013a).

Use good engineering VGT scripts, especially in the open source tool Sikuli (Yeh et al. 2009), require a deal of engineering to be as usable and maintainable as possible. For instance, VGT scripts should not be adopted as 1-to-1 mappings of manual test cases if these test cases include loops or branches since this will make the scripts more difficult to read and maintain. Instead, test cases of this type should be broken down into as short and linear test scripts as possible. These scripts should also be written in a modular way where script logic is separated from images to make the logic and images reusable between all test scripts in the test suite (Hellmann et al. 2014). This modularization practice supports maintenance since changes to the SUT only require script logic, or images, to be maintained in one place for the entire suite. Further, VGT scripts must be synchronized with the SUT's execution, synchronization that should be added systematically to the scripts, preferably in a way that makes it possible for the user to change script timing globally for different scripts, or parts of scripts, in the entire test suite simultaneously. Additionally, it is a good practice to add failure mitigating code in the scripts, for instance by having assertions rerun if they fail, to ensure script robustness. However, care should be taken with this practice since emphasis on robustness has negative effects on the scripts' readability and execution time. Finally, VGT scripts should be documented to improve readability and to ensure that reusable script modules are easily accessible.

Consider used software An automated test environment often consists of more than the tool and the SUT, it also contains simulators, third party software, build systems, etc. Different VGT tools have different built in capabilities to integrate with this environment that further stresses the need to evaluate different VGT tools during a pilot project. Additionally, if environmental software components are interchangeable, e.g. remote desktop or VM clients, several options should be tested to find the best possible solution for the company's context. Whilst this guideline is primarily based on previous work (Alégroth et al. 2014), it is supported by the results from Spotify that reported the evaluation of different VM's and even test frameworks for different tasks.

5.2 Use of VGT in Practice

Change roles Adoption of a new technique can require new roles to be formulated, e.g. a role dedicate to the development and maintenance of scripts. However, an individual placed in such a role needs training and/or time to familiarize themselves with the new technique which also adds to the adoption costs of the technique. Additionally, the new role's responsibilities will change that must be accounted for when planning the individual's workload.

Consider the development process VGT should be integrated into the development process to be effective. This implies adding the technique to the normal routines at the company for instance by adding VGT to the definition of done (If such is available) of a feature or task. Additionally, the scripts should be added to the company's build and test process to allow them to be executed automatically and frequently, e.g. every night. These nightly runs should support randomization of test script executions since it has been found that changing the order of the test scripts between test executions can have a positive effect on the scripts' failure-finding ability (Alégroth et al. 2013a).

In addition, the company needs to consider for what purpose VGT is used and cover other test related needs with other techniques. VGT is primarily a regression test technique but it can also be used to provide stimuli to a SUT during long-term tests to make these tests more representative of use of the SUT in practice. Hence, it is perceived that VGT can be used for more than regression testing, but, as stated, it cannot replace manual testing. Therefore, the adopting company needs to redefine their test process to make use of all the company's test techniques' benefits in the best way possible. Especially since VGT scripts find SUT failures that must then be analyzed manually, with root-cause analysis, to identify the associated defects in the code.

Whilst these guidelines have been hinted at in previous work on VGT, the results from this study gave confirmatory support of its importance. In summary, tool adoption requires substantial change to the context in which the tool is being adopted.

Change the organization With changes to roles and the development process comes also changes to the company's organization, e.g. diversion of responsibilities between individuals, new co-workers, etc. These changes can disrupt development for a time, which will have monetary impact before the new processes and organization settles. The impact of this organizational change can vary dependent on how VGT is adopted but it is suggested that VGT knowledge is spread across the organization but primarily handled by dedicated individuals, as reported in this study and previous work (Alégroth et al. 2013a).

Define code conventions Code conventions keep scripts consistent that make them easier to read and maintain. Additionally, these conventions can be used to convey how specific VGT related practices, e.g. systematic synchronization, should be performed by the developers or how the code should be structured to promote modularization, reuse and maintainability.

Minimize remote test execution VGT tools can be executed on top of remote desktop or VNC clients to facilitate testing of distributed systems. However, results from previous work (Alégroth et al. 2013a) indicate that this practice has adverse effects on some VGT tools' image recognition success-rate. Therefore, for distributed systems, it is recommended that the company uses a VGT tool with built in VNC functionality, e.g. EggPlant (TestPlant 2013), to mitigate these adverse effects. Another practice, recommended in previous work, is to minimize the use of VNC and run the scripts locally to the greatest extent possible. This practice implies that certain test cases become out of scope for VGT, the impact of which should be evaluated during the pilot study.

5.3 Long-term use of VGT in Practice

Adopt frequent maintenance To avoid test script degradation it is important to frequently maintain and improve the test scripts (Hellmann et al. 2014). Frequent maintenance also helps lower test maintenance costs since it avoids, to a larger extent, simultaneous maintenance of both logic and images that is more complex than logic or images separately (Alégroth et al. 2016a). As such, a maintenance process should be integrated into the company's overall development process to ensure that changes to the SUT have not caused scripts to break. Dependent on how VGT has been adopted, this maintenance process either requires common knowledge among all developers of how and when to update the scripts or clear communication channels to the individual(s) responsible for test script maintenance. Regardless, frequent maintenance is an important activity to ensure the long-term feasibility of VGT scripts in practice.

Measure for improvement Measuring the status of the VGT process is important to gauge its value contra the costs of performing it, for two main reasons. First, to evaluate if VGT is beneficial for the evolving SUT, i.e. is the technique equally suited to test new features of the SUT as it was when the technique was adopted? For instance, is it suitable to test the new features through the pictorial GUI or is a lower level automated test technique more suitable? Second, VGT scripts are large and slow in comparison to many other lower level test techniques. This implies that a VGT test suite becomes saturated quickly if a dedicated time slot is allocated for the test suite's execution. Especially since VGT scripts execute in the order of minutes and test suites in the order of hours. Hence, if VGT is used for continuous integration it may quickly become necessary to do test prioritization and test suite pruning (Hellmann et al. 2014), which is non-trivial without proper measures to identify what test cases to change, remove or execute for a specific test objective. Such a measurement scheme should therefore be put in place as soon as possible after the technique's adoption.

Version control scripts As reported from Spotify, it is possible to reuse test script logic between variants of a SUT but not the images. However, the variants of Spotify's applications share a lot of functionality which is not generally the case in practice. Further, the features and functionalities of variants of a SUT can diverge over time, which implies that the script logic cannot be reused. However, in some cases it may be required to migrate or reset old test cases to an old variant of the SUT, which implies that VGT scripts should be version controlled together with the SUT's source code to ensure script compatibility with different variants and versions of the SUT.

Consider the SUT's life-cycle VGT scripts are associated with a development cost that requires the scripts to be executed several times before they provide positive return on investment. Further, the test scripts are used for system and acceptance testing that implies that they can not be created before the SUT has reached a certain level of maturity. As such, they need to be formulated later in the development cycle and are therefore better suited for SUTs that will go through more than one development iteration or be maintained for a longer time period. Hence, for small projects where the product will be discontinued after the project, e.g. development of a one-off or a prototype, it may not be feasible, even suitable, to adopt VGT. Instead, manual regression and exploratory testing should be used.

6 Discussion

The main implication of the results presented in this work is that VGT can be used long-term in industrial practice. However, care must be taken how the technique is adopted and used for it to be feasible and to mitigate its challenges. This implication stresses the need for best practice guidelines regarding the adoption and use of VGT in practice, i.e. guidelines such as those presented in Section 5. The guidelines presented in this work are however not considered comprehensive and further work is therefore required to expand this set and evaluate their usefulness and impact in different companies and domains. The reason being the numerous contextual factors that must be considered when adopting a new tool or technique in a practical context. As input to such research, best practices for traditional software development could be analyzed and migrated for use in VGT scripting.

Another finding was that despite following best practices, the challenges associated with VGT proved too much for the technique's continued use in several projects at Spotify. Primarily this was due to high maintenance costs and because of Sikuli's limited ability to test mobile applications, i.e. run tests in the mobile device. It is possible that these challenges could have been mitigated by, for instance, pruning the test suites to focus only on stable SUT functionality and GUI elements, a statement supported by the result that the technique was feasibly used for Spotify's desktop and web applications. Additional mitigation could have been achieved by adopting another VGT tool, e.g. Eggplant (TestPlant 2013), which has better support for mobile testing. However these challenges can not be ignored and they, image maintenance in particular, should therefore be studied further to find generally applicable means of mitigation, both through process improvements and technical support. The reported VGT challenges finally resulted in the implementation of the Test interface at Spotify, a framework that has become the replacement for VGT in all but a handful of projects. This result implies that for a majority of projects there may exist better automated test solutions and future research should therefore be dedicated to identifying the properties that make VGT successful in some cases and not in others.

Further, Section 4.5 presented a comparison between Sikuli and Spotify's Test interface solution that showed statistically significant difference between the techniques. A key difference was the Test interface inability to emulate user behavior, i.e. stimulation and assertion of the SUT through the same interfaces as the human user. Instead, the Test interface, being a 2^{nd} generation GUI-based test solution, invokes interactions from beneath the GUI. This approach is suitable to test the application's functionality but does not verify if a user can access this functionality or that the SUT's GUI appearance is correct. In turn this implies that the Test interface requires more complementary manual testing than VGT, which also limits the technique's use for continuous delivery where a new feature should be built, verified and validated, shipped and installed to the customer automatically on each commit (Olsson et al. 2012). However in terms of flexibility of use, in Spotify's context, the Test interface could be tailored to fulfill test objectives not supported by Sikuli, for instance testing of non-deterministic outputs. On the other hand, Sikuli could test external software, e.g. Facebook, which is not supported by the Test interface. Meanwhile, both techniques were found equally easy to integrate with the Graphwalker MBT framework that improved both techniques' applicability and feasibility, e.g. by supporting migration of scripts between applications, improved maintainability of scripts, etc. Still, the presented analysis did not cover the costs of the techniques' adoption, where the Test interface approach was considered more costly than VGT, which is also the reason why Spotify adopted Sikuli in the first place rather than the more rigorous Test interface solution. The adoption costs were not compared in the analysis because it focused on the technique's properties in use in industrial practice.

As such, there are tradeoffs between the properties of the two techniques in terms of speed, robustness, flexibility, cost, etc. However, many of these divergent properties are complementary, which implies that a combination of both techniques could provide additional benefits, a sentiment also shared by Spotify. *"That would get rid of those parts (test tasks cumbersome in Sikuli)... a combination with Sikuli and this (the Test interface) would then be a solution"*. This sentiment also supports the conclusion drawn in previous work that, in an experimental setting, showed the value of combining VGT testing with 2^{nd} generation GUI-based testing to mitigate false test results (Alégroth et al. 2015). This previous work also evaluated automated model-based GUI-testing for test case generation and execution, work that is also supported by this study since Spotify's Graphwalker solution supports automated random testing of the GUI. Hence, technology that theoretically could be advanced to support automated exploratory GUI-based testing to further mitigate the need of manual testing of software applications. Thus, another topic that warrants future research.

Another interesting observation from this study was that the Test interface fruition was caused by the adoption of VGT that changed the automation culture at Spotify. The implication of this observation is that VGT could be a suitable first step for a company to improve their test automation practices. Especially in contexts where software legacy or lack of Test interfaces prevent the use of other test automation frameworks.

Consequently, this study provides valuable insights regarding the long-term perspective of using automated GUI-based testing in industrial practice, including challenges for the long-term use of these techniques. Based on these results it was possible to synthesize practitioner oriented guidelines, which shows the value and need for this type of research. This study thereby provides a concrete contribution to the limited body of knowledge on VGT but also a general contribution to the body of knowledge on software engineering that currently includes very few studies that focus on the long-term perspective of industrially adopted research solutions (Höfer and Tichy 2007; Marchenko et al. 2009). Hence, studies that are required to draw conclusions regarding the impediments of such research solutions and improve their efficacy, efficiency and longevity in industrial practice. In summary, this study thereby provides the following contributions:

C1: Industrial experiences and best-practices synthesized into a set of guidelines for avoidance of pitfalls and challenges with Visual GUI Testing (VGT) during adoption, use and long-term use of the technique(**RQ1, RQ2 and RQ3**),

C2: Industrial, comparative, results on the benefits between VGT and a 2^{nd} generation GUI-based test framework; the Test interface (**RQ2 and RQ3**),

C3: Evidence towards the previously unsupported conclusion that VGT can be used long-term (Over several years) in industrial practice,

C4: Support for the use of MBT-based VGT and the need for hybrid 2^{nd} and 3^{rd} generation testing, and

C6: Several potential, important, areas of future industrial research to solidify VGT's applicability and feasibility in long-term use in practice.

6.1 Threats to Validity

Internal validity Several measures were taken to ensure the internal validity of the study's results. First, the interviewees were carefully chosen through snowball sampling based on expert knowledge from within the case company's organization to ensure that the sample could provide representative results to answer the study's research questions. The study's time and budget constraints also made it possible to interview all but one of the interview candidates, which were spread across the organization and different projects. Additionally, the interviews were triangulated with results from two workshops that were attended by experts and individuals from all over Spotify's organization and provided results that also supported the interviews.

Second, to ensure a clear chain of evidence, the interviews were fully transcribed, coded and analyzed with a Grounded Theory approach (Glaser and Strauss 2009) that was modeled on previous research on the long-term use of TDD (Marchenko et al. 2009). This allowed low level conclusions to be drawn that were supported by one (1) to eight (8) statements each. These low level conclusions were then grouped further to draw the study's final conclusions, thereby ensuring that each conclusion was well grounded in the representative interview sample's statements.

Third, the guidelines presented in this work were triangulated with third degree data from both previous and related work. These guidelines were thereby supported both by the rigorous analysis in this study but also triangulated with external sources of evidence. The use of related work also helps mitigate bias in the guidelines fruition.

However, the study, as stated in Section 3.2, lacks objective, quantitative, results from analysis of Spotify's test suites, tools and models. This presents a threat to the validity of the study since the interviewees may have added biased opinions to the results. We do however estimate effects of such bias to be low due to the use of the aforementioned, rigorous, grounded theory analysis as well as triangulation of results from the diverse set of interview subjects. Future research should however be devoted to filling the gap for quantitative results left by this study.

External validity Only one case company was used for this study, chosen through convenience sampling, since there are few companies available that have used VGT for a longer period of time, i.e. companies that meet the prerequisites of this type of study. However, Spotify's organization is based on self organizing teams, called Squads, which are treated as their own startup companies that choose their own processes, practices and tools. Squads, which consist of maximum 10 individuals, are also grouped into feature teams, called Tribes, which consist of several Squads. The interviewees chosen in the study came from different Squads and Tribes that raises the external validity of study's results to both small and medium sized companies where a medium sized company would be in excess of 50 employees but less than 100 employees.

Further, the VGT guidelines presented in this work were triangulated with third degree data that was acquired in other companies and domains, including larger safety-critical software development companies. Thereby ensuring that the guidelines are generalizable for both small agile companies as well as large safety-critical software developers. Additionally, the related work was based on research with other types of automated GUI-based testing, which implies that the reported guidelines can be generalized beyond VGT.

However, future research is required to build on this work and expand it to other companies and domains to verify its results. This study thereby represents a foundation for such research since the reported results can help more companies avoid the pitfalls associated with automated GUI-based testing and only then reach a state of maturity where this type of research can be performed.

In addition, future work should also aim to evaluate the long-term use of other VGT tools since this study is limited to the use of Sikuli. Whilst we perceive, based on previous research, that other VGT tools such as EggPlant or JAutomate, which are more mature, would fair at least as well as Sikuli, more research is required to verify said fact. However, this research still provides evidence that VGT, i.e. image recognition driven test automation, is applicable long-term in industrial practice.

Construct validity The study was performed with interviews and workshops with industrial experts with years of experience and knowledge about VGT and automated testing in industrial practice. One interviewee was also part of the team that adopted VGT at Spotify. As such, these subjects could provide in depth answers regarding VGT's life-cycle in their context and valid support to answer the study's research questions.

Further, the guidelines presented in this work were primarily triangulated with empirical research on VGT performed in industrial practice, thereby ensuring the guidelines validity to help practitioners.

Conclusion validity/Reliability To improve the reliability of this study, as much detail as possible has been presented regarding the case company, the research process as well as how analysis was performed. These measures should make it possible to judge the validity of this work as well as to replicate the study in similar contexts to the one described. Additionally, references have been added to clarify the methods used in this work, e.g. case study design (Runeson and Höst 2009; Runeson et al. 2012), Grounded theory (Glaser and Strauss 2009), open coding (Saldaña 2012), snowball sampling (Kendall et al. 2008), etc., to endorse the study's replication.

7 Conclusions

This paper presents the results of a single, embedded, case study focused on the long-term use of Visual GUI Testing and automated GUI-based testing at the software application development company Spotify. The case study's results were acquired from two workshops and four (4) interviews with five (5) carefully chosen individuals from different parts of Spotify's organization to provide a representative view of the company's use of automated GUI-based testing. These results were then analyzed using Grounded theory to acquire the study's main conclusions.

Based on the study's results it was concluded that VGT, with Sikuli, can be used long-term in industrial practice but that there are many challenges associated with the technique, e.g. high maintenance costs of images, that some VGT tools have limited applicability for mobile application testing, etc. Because of these challenges Spotify have now abandoned VGT in most projects in favor of a 2^{nd} generation approach referred to as the Test interface. The Test interface has several beneficial traits, including higher flexibility in Spotify's context than Sikuli as well as lower maintenance costs. However, the Test interface approach does not verify that the pictorial GUI conforms to the system's requirements and still suffers from the same synchronization problems as other automated test techniques, including VGT.

Further, it was determined that Spotify's test process is well integrated into the company's organization, which, together with engineering best practices, were instrumental to VGT's successful adoption at the company. Based on a synthesis of these results, combined with results from previous and related work, 14 practitioner oriented guidelines could be defined for the adoption, use and long-term use of VGT in industrial practice.

This study thereby provides an explicit contribution to the body of knowledge of VGT about the long-term industrial use of the technique. Additionally, the study provides a general contribution to the body of knowledge of software engineering that is currently missing studies that focus on the long-term perspective of research solution's use and challenges in industrial practice.

Appendix A: Interview Questions

Table 5 presents the interview questions used during the four interviews at Spotify.

Table 5 Interview protocol used during the interviews at Spotify. **Maint.** - Maintenance, **Aband.** - Abandonment

Phase	#	Interview question
Adoption	1	What was the reason why Spotify choose to adopt Sikuli?
	2	What did the adoption process look like for the adoption of Sikuli?
	3	What barriers/challenges were observed during the adoption of Sikuli?
	4	How was Sikuli's Java API adopted to fit Spotify's test process?
	5	Why was Graphwalker chosen as suitable for the test architecture?
	6	What types of tests were performed with Sikuli?
Usage	7	What benefits were observed with Sikuli compared to other types of testing?
	8	What drawbacks were observed with Sikuli compared to other types of testing?
	9	How often were/could the Sikuli scripts executed?
	10	What was used as specifications for the Sikuli scripts?
	11	What types of defects could be identified with Sikuli?
Maint.	12	What was the need for maintenance of the Sikuli scripts?
	13	What was the need to maintain logic contra images in the scripts?
	14	How much tim was required to maintain the test scripts?
	15	What did the maintenance process for the Sikuli scripts look like?
	16	What challenges were identified with maintaining the Sikuli scripts?
	17	What was the main cause why the Sikuli scripts required maintenance?
Aband.	18	What caused the abandonment of Sikuli for the Test interface?
	19	What was the timespan from adoption to abandonment of Sikuli?
	20	Has the abandonment of Sikuli caused any new challenges for Spotify?

References

Alégroth E (2013) On the industrial applicability of visual gui testing. Tech. rep., Department of Computer Science and Engineering Software Engineering (Chalmers). Chalmers University of Technology, Goteborg

Alégroth E, Feldt R, Olsson H (2013a) Transitioning manual system test suites to automated testing: An industrial case study. In: Proceedings of the 6^{th} IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg, pp 56–65

Alégroth E, Nass M, Olsson H (2013b) JAutomate: A Tool for System-and Acceptance-test Automation. In: Verification and Validation (ICST), 2013 IEEE Sixth International Conference on Software Testing. IEEE, pp 439–446

Alégroth E, Feldt R, Ryrholm L (2014) Visual gui testing in practice: challenges, problems and limitations. Empir Softw Eng:1–51

Alégroth E, Gao Z, Oliveira R, Memon A (2015) Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study. In: Proceedings of the 8^{th} IEEE International Conference on Software Testing, Verification and Validation (ICST 2015), Graz

Alégroth E, Feldt R, Kolström P (2016a) Maintenance of automated test suites in industry: An empirical study on visual gui testing. Inf Softw Technol 73:66–80

Alégroth E, Steiner M, Martini A (2016b) 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, pp 257–262

Berner S, Weber R, Keller R (2005) Observations and lessons learned from automated testing. In: Proceedings of the 27th international conference on Software engineering, ACM, pp 571–579

Borjesson E, Feldt R (2012) Automated system testing using visual gui testing tools: A comparative study in industry. In: Verification and Validation (ICST), 2012 IEEE Fifth International Conference on Software Testing. IEEE, pp 350–359

Carver J (2007) The use of grounded theory in empirical software engineering. In: Empirical Software Engineering Issues Critical Assessment and Future Directions. Springer, pp 42–42

Finsterwalder M (2001) Automating acceptance tests for GUI applications in an extreme programming environment. In: Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, pp 114–117

Glaser BG, Strauss AL (2009) The discovery of grounded theory: Strategies for qualitative research. Transaction Publishers

Grechanik M, Xie Q, Fu C (2009a) Creating GUI testing tools using accessibility technologies. In: Verification and Validation Workshops, 2009. ICSTW '09 International Conference on Software Testing. IEEE, pp 243–250

Grechanik M, Xie Q, Fu C (2009b) Maintaining and evolving GUI-directed test scripts. In: 2009. ICSE 2009. IEEE 31st International Conference on Software Engineering. IEEE, pp 408–418

Hellmann T, Moazzen E, Sharma A, Akbar MZ, Sillito J, Maurer F et al. (2014) An exploratory study of automated gui testing: Goals, issues, and best practices

Höfer A, Tichy WF (2007) Status of empirical research in software engineering. In: Empirical Software Engineering Issues. Springer, Critical Assessment and Future Directions, pp 10–19

Holmes A, Kellogg M (2006) Automating functional tests using selenium. — pp 270–275

Isaacson C (2004) Web site development software. US Patent App. 10/844,095

Itkonen J, Rautiainen K (2005) Exploratory testing: a multiple case study. In: 2005 International Symposium on Empirical Software Engineering 2005, vol 10.

Karhu K, Repo T, Taipale O, Smolander K (2009) Empirical observations on software testing automation. In: ICST'09 International Conference on Software Testing Verification and Validation, 2009. IEEE, pp 201–209

Kendall C, Kerr LR, Gondim RC, Werneck GL, Macena RHM, Pontes MK, Johnston LG, Sabin K, McFarland W (2008) An empirical comparison of respondent-driven sampling, time location sampling, and snowball sampling for behavioral surveillance in men who have sex with men, fortaleza, Brazil. AIDS Behav 12(1):97–104

Kniberg H, Ivarsson A (2012) Scaling agile @ spotify. online], UCVOF, ucvox files wordpress com/2012/11/113617905-scaling-Agile-spotify-11 pdf

Lalwani T, Garg M, Burmaan C, Arora A (2013) UFT/QTP Interview Unplugged: And I Thought I Knew UFT!, 2nd edn. KnowledgeInbox

Leotta M, Clerissi D, Ricca F, Tonella P (2013) Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: 2013 20th Working Conference on Reverse Engineering (WCRE). IEEE, pp 272–281

Leotta M, Clerissi D, Ricca F, Tonella P (2014) Visual vs. dom-based web locators: An empirical study. In: Web Engineering, Lecture Notes in Computer Science, vol 8541. Springer, pp 322–340

Leotta M, Clerissi D, Ricca F, Tonella P (2016) Chapter five-approaches and tools for automated end-to-end web testing. Adv Comput 101:193–237

Marchenko A, Abrahamsson P, Ihme T (2009) Long-term effects of test-driven development a case study. In: Agile Processes in Software Engineering and Extreme Programming. Springer, pp 13–22

Nguyen BN, Robbins B, Banerjee I, Memon A (2014) Guitar: an innovative tool for automated testing of gui-driven software. Autom Softw Eng 21(1):65–105

Nilsson NJ (1980) Principles of artificial intelligence. Tioga Publishing

Olan M (2003) Unit testing: test early, test often. J Comput Sci Coll 19(2):319–328

Olsson HH, Alahyari H, Bosch J (2012) Climbing the stairway to heaven–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In: 2012 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA). IEEE, pp 392–399

Olsson N, Karl K (2015) Graphwalker: The open source model-based testing tool. http://graphwalker.org/index

Patel C, Patel A, Patel D (2012) Optical character recognition by open source ocr tool tesseract: A case study. Int J Comput Appl 55(10)

Rafi D, Moses K, Petersen K, Mantyla M (2012) Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test (AST), pp 36 –42. doi:10.1109/IWAST.2012.6228988

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Softw Eng 14(2):131–164

Runeson P, Höst M, Rainer A, Regnell B (2012) Case study research in software engineering: Guidelines and examples. John Wiley & Sons

Saldaña J (2012) The coding manual for qualitative researchers. 14, Sage

Shazam (2016) Shazam. http://www.shazam.com/

Sjösten-Andersson E, Pareto L (2006) Costs and Benefits of Structure-aware Capture/Replay toolss. SERPS'06 p 3

TestPlant (2013) eggPlant. http://www.testplant.com/

Vizulis V, Diebelis E (2012) Self-Testing Approach and Testing Tools. Datorzinātne un informācijas tehnoloģijas p 27

Weinstein M (2002) Tams analyzer for macintosh os x: The native open source, macintosh qualitative research tool. http://tamsys.sourceforge.net/

Wohlin C, Aurum A (2014) Towards a decision-making structure for selecting a research design in empirical software engineering. Empir Softw Eng:1–29

Yeh T, Chang T, Miller R (2009) Sikuli: using GUI screenshots for search and automation. In: Proceedings of the 22nd annual ACM symposium on User interface software and technology, ACM, pp 183–192

yworks (2016) yEd Graph Editor: High-quality diagrams made easy. https://www.yworks.com/products/yed

Emil Alégroth is a PhD in Software Engineering at Blekinge Institute of Technology, Sweden, as well as Chalmers University of Technology, Sweden. His main interests are automated testing with emphasis on automated GUI-based testing or Visual GUI Testing. He is also interested in decision-making in Software Engineering, behavioral software engineering and technical debt in software and testing. Emil received a PhD in Software Engineering from Chalmers University of Technology in 2015.

Robert Feldt is a professor of software engineering at Chalmers University of Technology, Sweden and at Blekinge Institute of Technology, Sweden. He has also worked as an IT and software consultant for more than 20 years. His research interests include human-centered software engineering, software testing and verification and validation, automated software engineering, requirements engineering and user experience. Most of the research is of empirical nature and conducted in close collaboration with industry partners. He received a Ph.D. (Techn. Dr.) in computer engineering from Chalmers University of Technology in 2002.