An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices

Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{tdhellma, emoazzen, absharma, mzakbar, sillito, frank.maurer}@ucalgary.ca

***Abstract*—Manually testing GUIs can be expensive and complex, so the creation of automated GUI test suites has been an area of significant interest. However, to our knowledge, the motivations of testers and the problems they encounter when attempting to create and use automated GUI tests have not been explored. We used Grounded Theory to investigate the goals motivating automated GUI testing, the issues testers encounter, and the best practices applied to overcome these issues. Through this study, we demonstrate that automated GUI test suite evolution and architecture are extremely important to the success of automated GUI testing and describe techniques that can be of use to practitioners. In addition to these best practices, this study identifies additional areas in which future research should be concentrated.**

Keywords- automated GUI test; test suite architecture; test suite evolution; best practices;empirical study

I. INTRODUCTION

Graphical User Interfaces (GUIs) display information and possible actions to users through graphical elements, or *widgets*. These elements allow an application to be controlled using a mouse, stylus, or, for touch-screen devices, a finger. The freedom of interaction that GUIs offer greatly increases the ease of use of software applications. As with other parts of an application, it's crucial that GUIs be covered with automated tests – by which we mean automated, end-to-end testing of an application through its GUI. However, automated GUI testing (AGT) is a notoriously difficult task.

The central difficulties with AGT – complexity of GUIs, verification of results from widgets, and the rapid rate of change of GUIs – are well-understood in existing literature, but previous work focuses on technical solutions: better tools, different test paradigms, etc. This paper seeks to investigate what makes automated GUI testing difficult from the perspective of practitioners. Specifically, we investigate three primary research questions: what goals do people expect to be able to achieve using AGTs; what sort of issues do they encounter; and what techniques have they developed to overcome these issues and achieve their goals?

In order to investigate these questions, we performed a series of semi-structured interviews with people who have had experience with the creation and use of AGTs. We analyzed the transcripts of these interviews using Grounded Theory [1]. In addition to providing insight into each of these research questions and empirically confirming the expected difficulties with AGTs, our study found two central problems that are not discussed in AGT literature: *AGT suite architecture* and *AGT co-evolution with the GUI and underlying system*. These issues represent significant challenges to the effectiveness of AGTs according to our participants.

This paper is structured as follows. Section II describes our research methodology. Results of the study are presented in Sections III, IV, and V. Section VI explains promising directions for future work. Threats to the validity are covered in Section VII. Finally, Section VIII concludes this paper.

II. METHODOLOGY

We conducted semi-structured interviews with eighteen participants with varying experience in automated GUI testing. We analyzed the resulting transcripts using Grounded Theory [1]. Our two-stage analysis involved open coding [2], sorting of codes, and identification of cross-cutting categories.

A. Participant Demographics

Eighteen participants with experience in the creation, maintenance, or use of AGTs were recruited for this study. During the first phase of our data analysis, however, we noticed that participants with less than one year of practical experience tended to focus on tool-specific rather than general issues with AGTs – issues such as learnability, usability, and reliability of the tools they had used. Because we wanted to focus on general issues with AGTs, we excluded these interviews from the second phase of our data analysis.

This left us with a set of interviews of eight participants that we used for phase two of our analysis. Information about these participants' amount of experience with AGTs, area of employment, and primary role is provided in Table I. In this table, participants with less than two years of experience with AGT were categorized as "junior;" participants with less than five but more than two years of experience with AGT were categorized as "intermediate;" and participants

with more than five years of experience with AGT were categorized as "senior."

B. Interview Design

We used semi-structured interviews because of the exploratory nature of this study. Therefore, the initial questions we asked were very high-level: "the last time you were using AGTs, what were you trying to accomplish"; "what functionality were you targeting with these tests"; etc. We were then able to explore issues that participants identified in their responses in greater detail. These interviews ranged in length from 12 to 60 minutes with an average duration of 32 minutes.

C. Analysis

Our analysis was split into two phases. Phase one operated on the full set of eighteen interviews and was used to discover general topics of importance to our participants. Phase two sought to identify themes important to the restricted set of eight participants focusing only on the topics discovered in phase one.

First, a round of open coding [2] was performed. Four of the authors participated in this coding process to limit the bias of individual researchers. The three general topics we discovered through this process were: *goals; issues;* and *best practices*.

We used these results to inform phase two of our analysis. The first step in phase two was to perform open coding again on this set of eight interviews. In this round of coding, we only coded topics directly related to one of the three topics identified in phase one. This selectiveness helped us to focus and constrain our analysis and was done by having a team of two authors examine each interview initially, then having at least one other author re-examine their work; again, this was done to reduce individual bias.

Through this realization, we discovered two important themes: AGT suite architecture and AGT co-evolution with the GUI and underlying system. We further subdivided each theme into issues that our participants overcame, the best practices they used to overcome them, and issues that our participants had not been able to solve. The following three sections explain the goals our participants expressed, along with the themes of test suite evolution and test suite architecture.

III. Goals of Automated GUI Testing

Tools should support specific uses, so we used our interviews to investigate what the goals of automated GUI testing actually are from the perspective of practitioners.

TABLE I. Participant demographics

ID	Experience	Employment Sector	Role
1	Junior	Academia	Developer
2	Senior	Academia	Professor
3	Senior	Industry	Tester
4	Junior	Industry	Developer
5	Intermediate	Industry	Developer
6	Intermediate	Industry	Developer
7	Senior	Industry	Tester
8	Senior	Industry	Tester

A. Automated Acceptance Testing

Automated acceptance tests (AATs) are traditionally created in collaboration with the customer as an encapsulation of expectations about how a feature should work. AATs take the form of an automated test that operate at the system level to demonstrate that a feature is working. Participant 4, for example, uses AGTs as AATs so that he has "a certainty that we're doing things right."

Six of our participants (2, 3, 4, 5, 7, 8) use AGTs as AATs to verify that "this software meets the user's requirements" (Participant 3). Not all of these participants agreed on where the customer's expectations should come from. Participants 2, 3, 7, and 8 felt that expectations should come directly from the customer, but Participants 2 and 7 felt that AATs can derive from their own expectations of how the system should behave. Participant 2, for example, created AATs so that he could "be sure that what I have is correct according to my expectations." Participant 5 also came into conflict with the traditional understanding of AATs in that he derived customer's expectations from design artifacts like written specifications and user interface prototypes as opposed to from customers/users.

B. Automated Regression Testing

Automated regression tests (ARTs) are used to alert developers that a regression error has occurred. Five of our participants (2, 4, 6, 7, 8) used AGTs as ARTs. This is important to "make sure that things are not breaking as we move to a new version" of the system (Participant 4). The ARTs they create are able to catch errors in "the wiring of the application itself and… the wiring that is done in views through configuration" – the linking between elements of the GUI and methods in the business logic or other GUI elements (Participant 8). This is important to our participants because "unit tests won't catch those" (Participant 8).

Participants 6 and 8 add new ARTs when regression errors are caught by human testers. Participant 8 relied on GUI-level ARTs because "if you want to change a part of the system, you write a test, and if you break another test, you can see [the change's] impact on another scenario." This rapid feedback was also important to Participant 2, who used a suite of ARTs to make it safe for him to experiment with "several variations… and still see that effectively those different approaches could have the same result."

C. Other Goals of Automated GUI Testing

Our participants also expressed several other motivations behind the creation of AGTs. Participant 8, for example, uses AGTs because "you have to make sure everything gets tied together and works properly." Participant 2 uses AGTs to "fine-tune what is the problem that I am facing." Participants 3 and 7 use AGTs "to make sure that we don't have to physically do those tests every day" (Participant 3). Participant 3 also uses AGT to make sure that "anything like a show-stopper, anything that would make [the application]

not useable, is not there" so that the software that "is being developed and tested… [can be put] into use right away."

D. Recommendations for Tool Developers

Tools for creating AGTs need to directly support automated acceptance testing and automated regression testing as primary use cases. Additionally, our participants note two major difficulties. First, participants sometimes ended up automating GUI tests with lower defect detection potential than the test they originally envisioned. Second, when a test fails, participants wonder first if the test is broken – demonstrating that the participants don't consider their AGTs to be reliable to the same degree that unit tests are considered reliable. Addressing these issues should be a high priority for developers of tools that support the creation of AGTs.

E. Related Work

Meszaros in [3] investigates the motivations behind creating automated unit tests, some of which are similar to the goals our participants had for AGTs. Meszaros lists eight goals of test automation, four of which line up with our findings: tests as specification; tests as safety net; risk reduction; and bug repellent. However, the other four goals that Meszaros lists do not match up with our findings: tests as documentation; defect localization; ease of creation/maintenance; and improved quality. In future work, it would be useful to look into this matter further to determine if these latter four goals really are not present as goals for AGT – and, if so, why not?

IV. AGT Suite Evolution

AGTs, on a very basic level, reflect aspects of a system under test; as the system changes, AGTs that refer to it will also need to change. Test suite evolution was a major theme in many of our interviews.

A. Challenges Posed by Evolving GUIs and System

Of our eight participants, five (3, 5, 6, 7, 8) encountered issues related to test suite evolution. The basic problem is that many changes to a GUI will require reciprocal changes to corresponding AGTs. Participant 3 discovered that "because you have existing automation, and those tests are rigged by you according to the system… if there is a change in [a] feature you have to go back to your automation and reflect that change." So, when the GUI under test – or a feature accessible through the GUI – changes, AGTs can report failures while the system under test is actually functioning as expected.

This is complicated by the fact that it is likely that a GUI will continue to change over the course of development, meaning that suites of AGTs will require ongoing maintenance. Participant 6 found that an AGT is not "something you can just set up once and then you're done. You have to consistently maintain it the same way you consistently maintain any piece of software." This sort of ongoing effort is expensive – in Participant 3's experience, "continuous improvement… can kill you." Participant 7 encountered the situation where "someone starts with great intentions, goes and creates all these tests, then something changes [in the GUI]… and [test maintenance] becomes a full-time job."

The effort of updating a suite of AGTs is more complicated in light of the fact that our participants also had trouble figuring out what to do about a failing test. Both participants 5 and 6 both expressed difficulty in pinpointing the cause of a test failure. In Participant 5's experience, the "GUI test could not show us what is the root of the problem, whether there is some problem in the business logic of the application or something wrong with the user interface." To deal with this, Participant 6's process for investigating an AGT failure reflected this difficulty with debugging: "my first instinct is to look at the errors and figure out 'is this something that I've seen before' … then it's go through the test, figure out where things stop, figure out what the test was designed to do… at the step that it failed at." His process deals in large part with debugging the test itself to figure out if the system is actually broken or if the test needs to be updated. We noticed from this that the problem of understanding an AGT is twofold: figure out if the system is at fault or if the test is; if the system is at fault, figure out what part of the system is at fault.

B. Best Practice: Comprehensive Test Maintenance

Out of the five participants that encountered issues with AGT suite evolution, three (3, 7, 8) had developed ways of mitigating the impact of these issues. The first recommendation our participants had for ways to decrease the burden of AGT suite evolution was to remember the essentially reflective nature of AGTs. Since AGTs should reflect the way in which a feature works, modification of AGTs should be considered an essential step in modifying how a feature works. Participant 8 found that each "test has to be enhanced as you're developing more of the system to reflect how we use the system." As a result of this positive mindset, within his team, "most of the time when the regression test goes red, it's because a feature is broken." Contrast this with, for example, Participant 6's experience with AGTs and his default assumption being that a failing test is simply broken and needs to be fixed. In Participant 6's context, when a test breaks, there is a problem with that test which needs to be resolved. The focus of this mindset is on getting the AGT suite passing, and while it will quickly get the system back into a green state, it fails to acknowledge that a failing AGT should indicate a problem with the system. Participant 8's mindset acknowledges that AGTs and system (not the GUI alone) co-evolve and continue to reflect the end-user's expectations. In both contexts AGTs break, but Participant 8 avoids developing an antagonistic relationship with his test suite by acknowledging and embracing this relationship.

A key point about comprehensive maintenance is the need to continually improve AGTs so that they continue to provide value as the system changes. Participant 3 understands a broken test as an indication that his team needs to "continue to improve our automated tests to make sure they're giving us the best results… results that uncover other issues that might exist in the software." In this mindset, AGTs can't simply be created at some point in time and left

in that initial state for the duration of the project. In that state, the AGT won't have any realistic prospect of detecting errors. The system will quickly evolve past the point where its AGTs are relevant. Participant 3 improves his AGTs continually both to ensure that the feedback they provide is good and to continually increase their defect detection potential. From this viewpoint, AGT evolution provides an opportunity to improve the value an AGT can provide that should be viewed as a way to offset the cost of maintenance.

However, it's also quite possible that the overhead of maintenance for specific AGTs will become too high to justify the value they provide. Participant 7 is adamant about making sure that each test has a *raison d'être*: "Automation for automation's sake is not worthwhile… you have to be able to look at any given test and be able to say 'this is why this test is here, this is why I can't just have an intern sit there and do this.'" Participant 3 found that when "those changes are getting in the way of the project… it's becoming cost-ineffective to put that test into automation. … The advantage of automation is to help you do some things without being too involved, and if I get too involved then there's no point in automation." This sort of situation can occur when "the test never should have been there to begin with, it was written poorly, or the underlying reason for the test is no longer valid" (Participant 7). In the event that a test begins to require too much maintenance effort, it's important to realize this situation early on and perform the test manually or abandon it entirely if it is no longer offering any value to offset its upkeep.

C. Open Issues

Participants 3 and 7 raised several additional issues regarding test suite evolution. First, there is a likelihood that AGTs will go stale over the course of a project – for example, Participant 7 wonders "at what point do you say 'I need to rewrite this test because it's never given me any sort of value.'" This issue is present in other forms of testing, but the tendency of AGTs to require more maintenance and to take longer to run than other AGTs means that the cost of putting up with stale AGTs is higher. Another consequence of the fact that AGTs run at a very high level is that it's difficult to determine the point at which the number and quality of GUI tests is reasonable for testing a given system. Even though it's possible to "have tests on 100% of things, you have not covered 100% of the scenarios. It's not even theoretically possible" (Participant 7). With respect to AGTs, the difficulty is that it is easy to create tests that generate very high coverage metrics, but it is difficult to determine if they are actually testing the system in a relevant manner. Participant 7 expects that "a year from now, the software's changed… so the tests that you wrote a year ago may not be relevant."

This brings up the question: how long can we reasonably expect an AGT to last? Both Participants 5 and 7 found that their AGT suites only tend to last about two years. After that amount of time, their companies tend to decide to make use of a new technology for their user interfaces. This sort of re-architecting poses a distinct threat to a suite of AGTs.

We would like to take this opportunity to raise a question related to AGT suite evolution: what is the fundamental difference between this form of software evolution and any other form of software evolution? Why haven't solutions found in the general field of software evolution been applied to automated GUI testing? One of the more obvious differences that we note is that there is only a small amount of experimental support for technologies like syntax highlighters within the GUI domain [4] [5]. Within a typical IDE, for example, changes to a method signature cause compile errors that are immediately apparent. Further research into other fundamental differences between AGT evolution and evolution of other types of systems would identify practical issues that GUI testing tools could address.

D. Related Work

The ways in which AGTs evolve over the course of a software development project have been explored previously in [6]. This study researched insights into the co-evolution of a GUI-based application and its AGT suite and found that updated AGTs were able to detect flaws in future versions of the system.

The necessity of continuously asking whether or not a given AGT should be automated echoes Marick's early work "When Should a Test Be Automated?" The findings we report here are validated by the three questions Marick encourages us to ask before we automate any test [7]: is automating this test going to save effort; how long is this test going to last; and how likely is this test to find bugs? While the issues and best practices our participants raise are not

TABLE II. Issues and corresponding best practices.

Issue	Best Practice
Test Suite Evolution	**Comprehensive Test Maintenance**
Frequency of Maintenance	Continuous Improvement
Difficulty of Debugging	Prune Test Suite
Focus on Passing Tests	Focus on Working System
Detecting Stale Tests	(None)
Determining Test Lifetime	(None)
Test Suite Architecture	**Three-Layer Architecture**
Understandability	Separation of Concerns
Code Duplication	Increase Modularity
Low Reusability	Data-Driven Testing
Up-Front Investment	(None)

unique to AGTs, they do seem to carry a lot more weight than with other forms of testing. It would be useful to investigate this relationship in future work to determine what makes AGTs special in this way.

Related work has also been done into the basic questions: "do tests actually evolve alongside code?" and "how can we tell?" In [8], a qualitative approach is taken in which visualizations of software repositories were made to determine if it was possible to understand the co-evolution of test and production code. While this research was successful, understanding the visualizations was difficult. The authors extended this work in [9] to automatically determine whether test code is co-evolving with production code. This work could be used to address our findings in that it can make it possible to determine whether AGTs are evolving suitably to continue to add value to the software development effort.

An approach specific to evolving suites of AGTs has been proposed in which a "change guide" is created by automatically noting when widgets used in test code have changed and notifying human testers that a test failure could occur at a specific location within a test script [4]. This approach has also been used to explicitly type the widgets used in test scripts to assist in test maintenance [5]. This work doesn't attempt to remove humans from the process of test suite maintenance, but instead provides tool support for these activities. Approaches have also been proposed using genetic algorithms [10], heuristic approaches [11], and compiler-based approaches [12] in order to attempt to automatically repair broken GUI test cases.

V. Test Suite Architecture

Many of the difficulties our participants experienced had to do primarily with how the AGTs themselves were structured. We observed that the tight coupling between test code and GUI implementation and the low reusability of AGT code was largely due to the fact that AGTs tend to be created using a single-layer architecture where testing goals, business logic, and widget interaction are lumped into the same entity.

A. Problems Caused by Single-Layer Architectures

Four participants (1, 4, 7, 8) encountered issues related to the way their AGTs were structured. First, their tests were difficult to understand. Participant 6's process for figuring out which action caused a test failure only narrowed the cause of a test failure down to a certain position within a test. From there, it is necessary to actually understand what the test code at that point was attempting to do, which widget it was attempting to interact with, and, eventually, what caused the test failure. This is apparently more difficult than with most other kinds of tests. Participant 8 found that "it was hard to read the tests and figure out what the application was doing," and he realized that this was "because of the level of detail in the tests – moving the mouse, clicking buttons, filling that text box with that name, and then you try to look at the application and figure out what the 'user' is trying to do." His AGTs, which should have been an expression of user objectives, were implemented as a series of very direct, low-level interactions. Understanding the purpose of these atomic actions complicated test maintenance. Participant 8 found that using a testing framework where "the language of the tests was very low-level and very business-oriented" made matters even worse.

Next, our participants found that, when using this sort of test architecture, a relatively unimportant change to the GUI or the underlying system could cause many failures. Often "we have a huge test suite. We make a small change somewhere. Then we have a huge number of [tests] failing" (Participant 4). The root of this issue is the duplication inherent in creating AGTs that interact with the GUI at a low level. For example, Participant 4 was using "hard coding to find the UI elements" in each test. This means that, whenever the GUI was updated, a large swath of the test suite would need to be updated to reflect what was essentially a single change, and Participant 4 found that "it's a lot of work to go and fix those tests."

We notice that AGTs written using a single-layer architecture and a high level of detail pose a challenge to reusability. Instead of creating tests in such a way that parts can be easily extracted and used elsewhere, participants tended to create AGTs as "custom test code with very little generic application" (Participant 1). We notice that this focus on creating highly-detailed, highly-customized test code gave some of our participants a tendency to create a new test from scratch rather than making use of existing test code.

We note that one reason for this could be the popularity of capture/replay tools (CRTs) among our participants. CRTs support the creation of AGTs by observing and recording interactions with a GUI. CRTs make it easier to create AGTs, but they usually record test scripts in a domain-specific language. These AGTs tend to exist independent of other tests and be structured according to a single-layer architecture. "So," Participant 7 explained, "what you're left with if you're using a CRT… are say 20 tests that you run. They're probably fairly easy to put together that first time… but then say something changes on the screen, and you have to go to 20 different places… You have to rebuild that test again – over and over and over again." In this respect, CRTs exacerbate the problems that we have already identified by making it easy to create a large amount of duplicate code.

This duplication can immediately become an issue when testing a web interface on multiple browsers. "You have to look at every browser," Participant 6 found, "and you have to add special rules or special cases for every browser." In this situation, a single change can necessitate updates in many different affected AGTs and in the many different versions of each of those tests.

B. Best Practice: Multi-Layer Architecture

Out of the four participants who encountered issues with test suite architecture, three (1, 7, 8) provide techniques for dealing with them. The suggested test architecture is summarized in Fig. 1.

The first point participants raised was the need for some amount of modularity in the AGTs they were building. For example, Participant 1 likes a feature of Selenium[1] that

[1] A major web testing tool. See: www.seleniumhq.org

allowed him to "create modules; for instance, a login method," which he can reuse "rather than having to redo an entire segment of the script." This enables him to encapsulate sets of actions that have the potential to be used outside the context of a single test and avoid creating redundant code. This also creates a single point of failure. If one of the widgets involved in Participant 1's login module were to change, this change would initially cause every test using that module to fail; however, a fix will need to be implemented in only a single section of the test code rather than being propagated to a large number of AGTs.

Participant 8 felt that better AGTs resulted from including user goals in a separate layer of the test suite that "take[s] care of navigation, flow." This makes it possible to change details about how to interact with a GUI without losing sight of a high-level user story. Participants 7 and 8 also found it useful for a test suite architecture to contain an intermediate layer. Participant 8 explained that the middle layer he uses is "a level of detail where we express the business goals" and "is about separating the navigation flow from the user objectives." We believe that this method of abstraction should allow the actions required to trigger business logic to change without impacting user goals. This is important because it means we don't need to figure out what a test is supposed to be testing as a first step to modifying that test; it should be obvious from the top layer's description of the test. Further, if at some point in the future we need to radically re-architect the GUI or underlying software, we won't need to entirely recreate each AGT. This means that, to an extent, a suite of AGTs structured in this manner should be safeguarded against software re-architectings.

Further, this multi-layer architecture provides our participants with potential for reuse. Participant 7 complimented the three layers proposed so far with data-driven testing so that "you can have a database or whatever with the data that you're going to test against. You can have all your test cases in there." With both test data and information required to locate widgets stored in a database and a multi-layer test suite architecture, it is possible to test a large number of test cases using a single, small AGT combined with a relatively small set of data defining different test cases and success criteria. This architecture should be robust against changes in that it should be possible to make changes to the various portions of a test – user story, middle layer interactions, discrete interactions with the GUI, data used to define test cases – without breaking AGTs in a way that will require overwhelming effort to repair.

C. Open Issue

Setting up the various layers of the architecture is an investment. Participant 8 found that "it took some time… to build the infrastructure… you go pretty slowly at the beginning because you have to build everything in the test infrastructure." However, "as you put more of the infrastructure in place, you can reuse that in different scenarios, so it starts to pay off pretty quickly." The process of moving from a single-layer architecture to a multi-layer architecture took several two-week iterations. Future work should aim to reduce the time between when we start investing in a multi-layer AGT architecture and when the return starts to outweigh the investment.

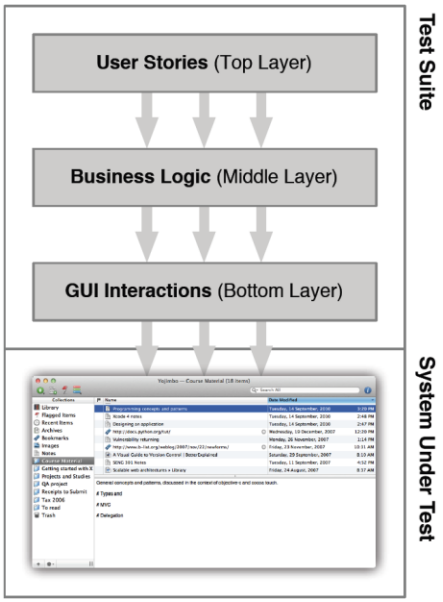

Fig.1. Multi-layer AGT suite architecture showing calls between layers.

D. Related Work

While work exists on the architecture of software systems, little work exists on the architecture of AGT suites specifically. Multi-layer test suite architectures have been briefly proposed in the field of hardware testing [13], but within software engineering specifically this topic has not been discussed.

VI. Future Work

This investigation was able to identify a host of areas – based on the concerns of people actively engaged in the use of AGTs – for future work.

A. Additional Topics

In future publications, we intend to explore the concerns we could not address in the current publication, including:

1) Widget Identification

Five of our participants (1, 3, 4, 6, 7) have trouble getting their AGTs to reliably find the relevant set of widgets for a test. The basic issue is that changes to a GUI can break AGTs that rely on the structure of the GUI itself or on information about particular widgets. Fixing these broken tests can waste a lot of time, especially in a situation where a failing test does not represent a broken feature. An additional complication exists in the case where information about a widget changes over the course of a test. This makes it difficult for a human to make the semantic connection between the identifying characteristics of widgets present in a test and widgets in a GUI and can complicate debugging.

Our participants had experimented with three solutions: use keyword-based testing (1, 7); use a system to

heuristically identify widgets (6); use a screenshot-based tool (3). However, keyword-based testing can complicate the process of using automated tools to extend existing AGTs (as in [14]). In the case of heuristic identification, we caution that the cases where identification fails could be much more complicated to debug. Further research is needed on this topic specifically. Screenshot-based testing, on the other hand, is very brittle against a variety of trivial changes. Each of these may represent worse solutions to a bad problem, so future work needs to be done not only to identify better solutions, but to explain why this problem is so much more complicated for AGTs than for other forms of testing.

2) Targets for Automation

Participants 3, 4, and 7 felt that a primary candidate for a test that should be automated was something that was easy to automate. When pressed on this issue, all three also felt that the task should also be repetitive. Participant 7 also found that these tasks should be things that customers will care about, have high visibility, and be important if they were to actually fail. Guidelines for which GUI tests to automate and which to perform manually would be useful for practitioners and this topic would be an excellent source of future work.

Participants 4 and 6 found that AGTs involving the look-and-feel and presentation/layout of GUIs are hard to automate. It would seem that there are characteristics of GUIs that are difficult to test, and it would help in the design of future tools to look into what makes a given manual test difficult to successfully turn into an AGT.

VII. Limitations

Several limitations are present in this study. First, our results are based on interviews. In order to determine whether these findings are valid in general, it would be necessary to perform more detailed, longitudinal observational investigations into the effects of test suite co-evolution with GUI/system code and test suite architecture on the effectiveness of AGTs.

Second, due to the filtering of interviews at our second phase of data analysis, our study looked only at the experiences of experienced automated GUI testers. The issues that less-experienced testers had were clustered around tool-specific problems, but not in themselves invalid. Future work should study these testers to determine why, for example, people tend to give up on AGTs.

VIII. Conclusions

This paper presents insights into the goals, issues, and best practices of automated GUI testing. We were able to discover that the major goals AGTs were used to achieve were acceptance testing and regression testing. Further, we found that test suite evolution and test suite architecture were significant issues for our participants. In terms of test suite evolution, participants that had dealt with this issue had discovered that AGTs need to co-evolve with the system they operate on, AGTs need to be upgraded as the system or GUI changes, and frequently-breaking AGTs may need to be removed from automation. In terms of test suite architecture, we found that a three-layer test suite architecture had made it easier to maintain, understand, and reuse their AGTs. We believe that multi-layered architectures may provide protection against major changes to the system under test.

This paper serves as a first attempt to explore the large and largely unexplored area of automated GUI testing. Our results suggest that further research is needed in the following areas: identifying widgets from test code; what makes a good target for test automation; ways to lower the initial investment in multi-layered test suite architectures; and the evolution of AGTs. It will remain difficult to provide effective support for automated GUI testing until we understand more about what makes it so difficult in the first place.

References

[1] B. G. Glaser and A. L. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research, Chicago: Aldine, 1967.

[2] J. Saldaña, The Coding Manual for Qualitative Researchers, London: Sage, 2009.

[3] G. Meszaros, xUnit Test Patterns: Refactoring Test Code, 2nd ed., Upper Saddle River, NJ: Addison-Wesley, 2007.

[4] M. Grechanik, Q. Xie and C. Fu, "Maintaining and Evolving GUI-Directed Test Scripts," in *31st International Conference on Software Engineering*, Washington DC, USA, 2009.

[5] C. Fu, M. Grechanik and Q. Xie, "Inferring Types of Reference to GUI Objects in Test Scripts," in *International Conference on Software Testing Verification and Validation*, Denver, Colorado, USA, 2009.

[6] Y. Shewchuk and V. Garousi, "Experience with Maintenance of a Functional GUI Test Suite Using IBM Rational Funcitonal Tester," in *International Conference on Software Engineering and Knowledge Engineering*, Boston, USA, 2010.

[7] B. Marick, "When Should a Test Be Automated?," in *11th International Software Quality Week (QW'98)*, San Francisco, 1998.

[8] A. Zaidman, B. van Rompaey, S. Demeyer and A. van Deursen, "Mining Software Repositories to Study Co-Evolution of Production and Test Code," in *1st International Conference on Software Testing, Verification, and Validation*, Lillehammer, Norway, 2008.

[9] Z. Lubsen, A. Zaidman and M. Pinzger, "Using Association Rules to Study the Co-Evolution of Production and Test Code," in *6th IEEE International Working Conference on Mining Software Repositories*, Vancouver, Canada, 2009.

[10] S. Huang, M. Cohen and A. M. Memon, "Repairing GUI Test Suites Using a Genetic Algorithm," in *3rd IEEE International Conference on Software Testing, Verification, and Validation*, Washington DC, USA, 2010.

[11] S. McMaster and A. M. Memon, "An Extensible Heuristic-Based Framework for GUI Test Case Maintenance," in *International Conference on Software Testing, Verification, and Validation*, Denver, Colorado, USA, 2009.

[12] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," in *9th European Software Engineering Conference / 11th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, Helsinki, Finland, 2003.

[13] P. Bernardi, A. Bertuzzi, M. Grosso, V. Tancorre and S. Tritto, "Testing Parametric Cores: A Multi-Layer Test Program to Improve and Automate the EDA-ATE Link," in *7th European Manufacturing Test Conference*, Munich, Germany, 2005.

[14] T. D. Hellmann and F. Maurer, "Rule-Based Exploratory Testing of Graphical User Interfaces," in *International Conference on Agile Methods in Software Development*, Salt Lake, UT, 2011.