

UNIVERSITY OF CALGARY

Automated GUI Testing for Agile Development Environments

by

Theodore D. Hellmann

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

APRIL, 2015

© Theodore D. Hellmann 2015

Abstract

For as long as graphical user interfaces (GUIs) have existed, it has been difficult to test them. This difficulty has become more pressing with the advent of Agile development methodologies, which stress that software should be fully tested to demonstrate that it works from developers' and users' points of view. In this thesis, I explored the fundamental question of whether automated GUI testing and Agile development environments are, or can be made, compatible. I conducted systematic mapping studies to determine how GUI testing and agile are currently compatible in existing literature, and performed an interview study with practitioners in order to determine how well these results matched up with what is done in industry. Based on the results of these studies, I confirmed that test-driven development (TDD) was a major area where automated GUI tests could fit into an agile process. I proposed a novel approach to TDD of GUIs that leverages Agile User Interaction Design techniques: by using an interactive low-fidelity prototype – already part of many Agile development processes – as the basis for automated GUI tests, TDD can be performed on a GUI with minimal added effort. A controlled experiment was used to validate this approach with practitioners, and the results indicated that this approach would be useful in practice.

Acknowledgements

This work was funded in part through the NSERC SurfNet Strategic Network Grant.

This work was funded in part through an NSERC IPS grant in collaboration with TRTech.

This work was funded in part through the TRLabs Telecommunications Research Scholarship.

This work would not have been possible without the many co-authors I have worked with over the years. I would like to thank, in no particular order: Frank Maurer, Ali Hosseini-Khayat, Chris Burns, Teddy Seyed, Jennifer Ferreira, Abhishek Sharma, Tiago Silva da Silva, Apoorve Chokshi, Sydney Pratte, Zahra Shakeri Hossein Abad, Gabriela Jurca, Elham Moazzen, Md. Zabedul Akbar, and Jonathan Sillito.

I would also like to thank the many people who have worked with and supported me over the years I have spent in the Agile Software Engineering lab, including (in no particular order): Ali Hosseini-Khayat, Chris Burns, Teddy Seyed, Abraam Mikhail, Patrick King, Rob Post, Nadder Makhlouf, Keynan Pratt, Eric Kryskie, Julia Paredes, Tulio Alcantara, Pierre Bastianelli, Abhishek Sharma, Yaser Ghanam, Tiago Silva da Silva, Josy Oliviera, Maha Shouman, Shelley Park, Apoorve Chokshi, Marinho Moreira, Arlo Louis O'Keefe, Julia Zochodne, Kevin Bradley, Gellert Kispal, Gabriela Jurca, Fadi Botros, Mandy Wong, Sydney Pratte, Edwin Chan, Alaa Azazi, Yuxi Wang, Nicole Loison, Angela Solano, Rahul Bir, Daniel Sabourin, Rahul Bhaskar, Mahshid Marbouti, Mehrdad Neurolazde, Seyed Medhi Nashi, Shahed Khandkar, SM Sohan, Felix Reiger, Xin Wang, Christian Schlueter, Darren Andreychuk, Ealaf Selim, Elham Moazzen, Shahbano Farooq, David Snell, David Cooper, Barry Goerzen, Eric Lowther, Hanif Mohamed, Jim Jeffries, Rebecca Gianuzzi, Lance Doell, Neil Tallim, Rob Tasker, Jennifer van Zelm, Wali Rahman, Simon Engler, Amani Alali, Ayman Issa, Chris Luce, Jamie Starke, Brian Pokojoy, James King, Craig Anslow, Jeff LaFrenz, Robin Arsenault, Grace Whitehead, Mary Lim, Susan Lucas, Katie Cummings, Bev Shevchenko, Erin Moloney, Craig Ireland, Lorraine Storey, Carey Williamson, Arthur W. Frank, Grigori Melnik, Mikko Korkola, Jonathan Sillito, Mark Stadel, Tim Bliek, Darcy Grant and the rest of CPSC IT, all of the NSERC and Tri-Council Ethics staff, Angela Martin, Frank Maurer, Joerg Denzinger, Roberto Flores, Anton Riedl, and Matthew Dunlap.

Dedication

For Mom and Dad.

Table of Contents

Approval Page.....	ii
Abstract	ii
Acknowledgements.....	iii
Dedication.....	iv
Table of Contents	v
List of Tables	viii
List of Figures and Illustrations	ix
List of Symbols, Abbreviations and Nomenclature.....	xii
Epigraph.....	xii
CHAPTER ONE: INTRODUCTION.....	1
CHAPTER TWO: BACKGROUND.....	7
2.1 Agile Software Engineering.....	7
2.2 Software Testing.....	10
2.3 User Interfaces	15
2.4 Testing Graphical User Interfaces	17
CHAPTER THREE: RELATED WORK.....	22
3.1 Capture-Replay Tools (CRT).....	22
3.2 Maintenance.....	24
3.3 Architecture	27
3.4 UITDD	29
CHAPTER FOUR: PRACTITIONER MOTIVATION FOR AUTOMATED GUI TESTING	32
4.1 Methodology	32
4.1.1 Participant Demographics	33
4.1.2 Interview Design	34
4.1.3 Analysis	35
4.2 Goals of Automated GUI Testing	36
4.2.1 Automated Acceptance Testing.....	36
4.2.2 Automated Regression Testing.....	37
4.2.3 Other Goals of Automated GUI Testing	38
4.2.4 Other Issues	38
4.2.5 Discussion.....	39
4.3 Test Suite Evolution.....	39
4.3.1 Challenges Posed by Evolving GUIs and System	40
4.3.2 Best Practice: Comprehensive Test Maintenance	41
4.3.3 Open Issues.....	43
4.4 Test Suite Architecture	44
4.4.1 Problems Caused by Single-Layer Architectures	45
4.4.2 Best Practice: Multi-Layer Architecture	47
4.4.3 Open Issue	49
4.4.4 Discussion.....	49

4.5 Conclusions.....	49
CHAPTER FIVE: MAPPING ACADEMIC WORK ON AUTOMATED GUI TESTING	
5.1 Agile Testing.....	51
5.1.1 Methodology.....	52
5.1.1.1 First Study.....	53
5.1.1.2 Second Study	54
5.1.1.3 Processing Papers	55
5.1.2 Detailed Results.....	56
5.1.2.1 What has Agile testing been used for?.....	56
5.1.2.2 What types of research have been published?	58
5.1.2.3 Do academics or practitioners drive the field of agile testing?.....	61
5.1.2.4 What benefits do publications claim to offer?	65
5.1.2.5 What problems are associated with Agile testing?	67
5.2 GUI Testing	68
5.2.1 Methodology.....	69
5.2.1.1 Searching for Papers	69
5.2.1.2 Screening Initial Papers	71
5.2.1.3 Keywording using Abstracts.....	72
5.2.2 Detailed Results.....	72
5.2.2.1 What approaches to GUI testing have been researched?	72
5.2.2.2 What problems with GUI testing have been identified through this research?.....	78
5.2.2.3 What benefits do the publications claim to offer?	84
5.2.2.4 Which research methodologies have been used to study GUI testing?	90
5.3 Conclusions.....	93
CHAPTER SIX: TEST-DRIVEN DEVELOPMENT	95
6.1 Solution Description	97
6.1.1 Tools	97
6.1.2 Solution Concept	98
6.1.3 Demonstration	99
6.2 Validation Study	102
6.2.1 ExpenseManager	103
6.2.1.1 Prototyping ExpenseManager.....	104
6.2.1.2 Creating Tests for ExpenseManager.....	106
6.2.2 Participants	107
6.3 Results.....	108
6.3.1 During the Study.....	108
6.3.2 After the Study	110
6.4 Concluding Remarks on UITDD	113
CHAPTER SEVEN: LIMITATIONS	114
7.1 Weaknesses of the Interview Study	114
7.2 Weaknesses of Systematic Mapping Studies.....	116

7.2.1 Weaknesses of Systematic Mapping Studies Generally	116
7.2.2 Limitations of the Systematic Mapping Studies in Chapter Five	120
7.3 Weaknesses with the UITDD Studies.....	122
7.3.1 Limitations of the Approach.....	122
7.3.2 Limitations of the Evaluation	123
 CHAPTER EIGHT: FUTURE WORK	126
8.1 Future Work from Interviews	126
8.2 Future Work on Literature Mappings	128
8.3 Future Work on UITDD	129
 CHAPTER NINE: CONCLUSIONS.....	131
 WORKS CITED	135
 APPENDIX I: PAPERS FROM FIRST AGILE TESTING SYSTEMATIC MAPPING STUDY [54].....	144
 APPENDIX II: PAPERS FROM SECOND AGILE TESTING SYSTEMATIC MAPPING STUDY [55].....	153
 APPENDIX III: PAPERS FROM GUI TESTING SYSTEMATIC MAPPING STUDY	161
 APPENDIX IV: PUBLICATIONS DURING THIS DEGREE	169
 APPENDIX V: COPYRIGHT CONSENT	171

List of Tables

Table 1: Participant demographics.....	33
Table 2: Issues and corresponding best practices.	50
Table 3. Preliminary, nonunique search results by target, search string, and database... ...	70
Table 4: Frequency of keywords used to describe approaches to GUI testing. The top 10 of these, bolded, are discussed in this chapter.	73
Table 5. Frequency of keywords used to describe problems encountered during GUI testing. Those in bold are discussed in this chapter.	78
Table 6. Frequency of keywords relating to benefits of tools/techniques. Bolded: major keywords discussed in this section.	84
Table 7: Quantitative responses from survey.....	107
Table 8. Passing Tests by Participant	110
Table 9. Perception of Usefulness of TDD of GUIs.....	111
Table 10. Ranking of Available Resources.....	112

List of Figures and Illustrations

Figure 1: Vertical slices through system from iterative and incremental development (originally from [9]).	9
Figure 2. Glenn Vanderburg's 13 XP practices, from [2].	11
Figure 3: Common CLI demonstrating an issue with discoverability.	16
Figure 4: Several overlapping GUIs, demonstrating windowing and widget grouping.	17
Figure 5: Multi-layer AGT suite architecture showing calls between layers.	47
Figure 6: Keywords applied to papers used in [54].	56
Figure 7: Keywords applied to papers used in [55].	56
Figure 8: Frequency of types of paper found in both mapping studies [54] [55].	59
Figure 9: Number of publications per author in both studies (percentage of the total and absolute number per study). Bar heights are based on percentage.	60
Figure 10: Types of paper published in each year, from both [54] and [55] combined.	61
Figure 11: Publications over time, divided into separate categories based on where the authors worked at the time of publication.	63
Figure 12: Types of publication by source of publication.	64
Figure 13: Breakdown of benefits offered by publications by conference.	65
Figure 14: Breakdown of problems encountered with testing by conference.	67
Figure 15: Distribution of papers at the conferences analyzed between industry, academic, and collaborative sources.	67
Figure 16: Number of papers excluded by reason for exclusion. (Note: "Not Right Type" refers to non-research papers included in conference publications, for example reports on the results of workshops).	71
Figure 17: Approach to GUI testing by year of publication for each paper considered.	76
Figure 18: Absolute number of keywords by source.	77
Figure 19: Frequency of problems encountered in papers by year of publication.	82
Figure 20: Absolute number of keywords for each paper source by problem.	83
Figure 21: Frequency of benefit keywords in papers by year of publication.	87

Figure 22: Absolute representation of benefits offered by publications grouped by source.....	89
Figure 23: Bubble chart showing frequency of research methodologies used per year ..	90
Figure 24: Absolute numbers of research methodologies used by source.....	93
Figure 25: Storyboard of a test sequence. Highlighted areas represent mouse clicks in the first four states and the field to be verified in the last state.....	100
Figure 26: Test for the calculator's simple addition feature.....	101
Figure 27: Failing test - application logic still missing.....	101
Figure 28: A complete interface. The original test still passes.....	102
Figure 29: One page from the SketchFlow prototype of ExpenseManager.....	105
Figure 30: State map of ExpenseManager prototype.....	106
Figure 31: Workflow for development of GUI-based application observed during studies.....	108

List of Symbols, Abbreviations and Nomenclature

<u>Symbol</u>	<u>Definition</u>
AAT	Automated Acceptance Test
AGT	Automated GUI Test
ART	Automated Regression Test
CLI	Command Line Interface (Console)
CRT	Capture/Replay Tool
GUI	Graphical User Interface
SUT	System Under Test
TDD	Test-Driven Development
UI	User Interface
UITDD	User Interface Test-Driven Development

Epigraph

Uproot your questions from their ground and the dangling roots will be seen. More
questions!

Frank Herbert, *Chapterhouse Dune*

Chapter One: Introduction

The purpose of software development is to solve a problem or take advantage of an opportunity. The solution that software provides is an attempt to meet the needs of its users. It is generally impossible to use formal methods to prove that a system meets its users' needs or is free of errors^{1,2}, so the extent to which software works is generally gauged using software tests. Software tests encapsulate user expectations about a system, and can be automated so that they are executed against a system frequently during its development. This allows the system's developers to understand what parts of the system are or are not meeting expectations.

Agile software engineering is a collection of software development methodologies that emphasize this user-centric, test-informed approach to software development. For more than 13 years, Agile has been gaining strong support – to the point that Agile development methodologies can now be considered a mainstream [1]. Agile techniques place specific emphasis on frequent delivery of working software to customers in order to embrace and respond to changes. Within teams using Agile development methodologies, the main measure of progress is working software.

Automated software tests are generally used, especially by teams using Agile development methodologies, as a proxy for customer expectations and can indicate when functionality is incorrect. However, this approach is much more difficult to perform on systems with graphical user interfaces (GUIs). GUIs are part of most modern systems for the simple reason that they make it easier to interact with a system, but testing a system through its GUI is much more difficult than writing more traditional tests that interact with a system's code directly.

¹ Donald Kunth's famous quote on the matter is: "Beware of bugs in the above code; I have only proved it correct, not tried it." [77]

² Edsger W. Dijkstra's equally-famous quote is: "Program testing can be used to show the presence of bugs, but never to show their absence!" [10]

This is because tests that interact with GUIs function in a fundamentally different way from traditional tests. Whereas other types of tests interact directly with the code of an application, GUI tests work by interacting with the system in a way similar to how users would interact with the system. Rather than calling a method directly, as in a unit test, a GUI test would instead have to locate a specific part of the application on-screen, interact with it by triggering actions that a user could perform – like mouse clicks and keyboard input – and then evaluate correctness based on what the GUI displays onscreen as a result of these interactions. However, because GUI tests interact with software in a similar way to the system’s intended users, they can be very useful for evaluating whether a system is correct from the point of view of its users. If an application is not tested in the same way in which it will be used, errors that users are able to trigger – or functionality that users should be able to trigger, but can’t – may go unnoticed. Additionally, because GUI tests are cross-process, it can be difficult to determine whether a test failure indicates an actual violation of a users’ expectations of system behaviour. Many researchers are working to solve these problems directly (see Chapters 3 and 4), so in this thesis I will instead focus on GUI testing at the process level – on identifying and evaluating ways in which automated GUI testing can be merged with an Agile development process.

These difficulties are especially pronounced in teams using certain Agile development methodologies due to the frequency of change and centrality of testing they encourage. *Test-driven development* (TDD) specifically is a core Agile development methodology in which tests are written before the code they verify, enough code is written to cause the tests to pass, and then the code is refactored into a better solution while still causing the tests to pass. However, TDD is exceedingly difficult to perform at the GUI level due to many factors, including the likelihood of the GUI to change and the likelihood that these changes can cause test failures. *Refactoring*

itself, the process of improving the implementation of a system or its components without changing how it works functionally, may become problematic, as many GUI tests are tightly bound to elements of a GUI like the DOM tree's structure or the placement or appearance of individual widgets. Even when striving for *simple design*, the simplest way of implementing a GUI may not be compatible with the most testable way to structure a GUI – for instance, it would make it easier to test a GUI if unique identifiers were added for each widget, but keeping these identifiers up to date and unique can make developing the GUI take more effort.

Specifically, this thesis has one main goal: to determine if any methodologies used for testing in Agile development environments and any approaches to automated GUI testing are compatible. Where these two fields seem to be incompatible, the subgoal is to point out the differences between the two areas and identify a future work for bridging the fields; where techniques in both areas are complimentary, the goal is to propose a novel technique for a unified Agile GUI testing process leveraging the strengths of both fields.

GUI testing and Agile development methodologies are already the subject of large bodies of existing academic literature, but they are also practices that are used in industry. Because of this, it's necessary to approach this field from a variety of different perspectives. In order to properly evaluate the industry perspective, I use semi-structured interviews and experiments with practitioners. Semi-structured interviews allowed me to assess how practitioners currently work with GUI tests and whether these integrate with any Agile development methodologies they may use while experiments allow me to observe practitioners using techniques I propose. On the other hand, it is necessary to use some form of literature study in order to properly evaluate the academic perspective on GUI testing and Agile development methodologies. I chose to use systematic literature reviews as a way of investigating the academic perspective because this

allowed me to survey at a high level the approaches that have already been tried in academic circles. By looking at the crossover between these two groups of studies, I was able to focus in on techniques with a higher relevance in industry and which had not already been proposed in existing academic literature, yet which could benefit from the results of previously published work.

The first question that needs to be answered to meet this goal is to understand what techniques and processes are associated with testing in Agile development environments – what approaches are used, what issues are encountered and benefits are proposed, etc. – including, of course, an understanding of whether GUI testing is already an essential part of testing in teams using Agile development methodologies. The corollary to this question is to understand what GUI testing currently is, in the same terms – including, of course, an understanding of whether GUI testing already incorporates Agile development methodologies. Based on the answers to these questions, I identify areas in which Agile development methodologies and GUI testing could complement each other, and evaluate the integration of the two, User Interface Test-Driven Development (UITDD), in a pilot experiment using practitioners to evaluate its usefulness.

In order to contextualize the topics investigated in this dissertation from the point of view of people actually performing GUI testing as part of their work activities, I performed an interview study (Chapter Four). Semi-structured interviews were used to determine what goals, issues, and best practices people had when performing automated GUI testing. In this study, the participants also did not use a TDD for development of their applications' GUIs, and also encountered significant issues with test maintenance – possibly caused by the architecture of their test suites (these issues are also explored in Chapter Three). Overall, the participants focused on using GUI tests for acceptance and regression testing rather than any of the advanced

techniques found through the systematic mapping of GUI testing publications (presented in Chapter Five). As with agile testing, adopting GUI testing techniques was a significant issue.

I then approached this topic from an academic perspective by conducting a series of systematic mapping studies. The first two of these studies – one using an automated search for papers, one using a thorough search of conferences specific to Agile development methodologies – were done in order to gain an understanding of what testing approaches are used in environments where Agile development methodologies are used. The (brief) result of these mappings was that Agile testing sometimes makes use of GUI tests, but it is far from the focus of the field. Frequently, when GUI testing is done in Agile development environments, it is generally done using capture/replay tools (CRTs) – tools that record a tester's interactions with the system and can replay these interactions as an automated test. Agile testing also places heavy emphasis on test-driven development, but that adopting agile testing techniques was a notable issue. The results of these mappings are presented in Chapter 5.1.

I followed this up with a third systematic mapping (Chapter 5.2) to gain an understanding of what GUI testing is. The (brief) result of the mapping is that Agile development methodologies are not part of GUI testing – only a single paper (out of 166) mentioned any Agile development methodologies. However, as with Agile testing, capture/replay tools were a significant topic.

Based on these results, test-driven development of GUIs was investigated further. In two pilot studies, I first proposed a model for using tools and techniques from both GUI testing and user interaction design – a field of increasing interest to Agile groups – in order to create tests for a GUI using capture/replay tools before the application itself is created. I then extended this study – using a different set of tools to demonstrate that the technique was not tool-specific – by

performing an experiment in which practitioners used a test-first approach to GUI development. This study showed that, as in the interview study, participants focused on using GUI tests as acceptance tests for the system to be developed. Participants reported both that they found the approach useful and would be interested in using it in their own work. These results are presented in Chapter Six.

As mentioned earlier, I also wanted to provide a clear set of recommendations for future work in this area. By looking at all of the studies conducted as part of this dissertation, I identified maintenance, architecture, and adoption as issues that could hamper the integration of GUI testing into Agile development processes. In Chapter Eight, I lay out a series of follow-up studies that could be performed to directly contribute to this goal.

Chapter Two: Background

Again, the goal of this thesis is to determine if and how automated Agile development methodologies and GUI testing are, or could be made to be, compatible. In order to better inform this discussion, this chapter provides an overview of the core concepts of Agile software engineering, software testing generally, user interfaces generally, and testing of graphical user interfaces.

2.1 Agile Software Engineering

Agile software engineering encompasses a set of development and project management methodologies that focus on incremental, iterative development of working software. The goal of both the development and management practices associated with Agile software engineering is to put solutions in front of customers/users as soon as possible in order to get feedback from actual utilization of the software by the people it's being built for and address concerns that arise from it quickly – a form of user acceptance testing that helps ensure the project is meeting its customer's goals. Teams using Agile development or management approaches tend to be self-organizing and cross-functional with emphasis on communication between project members, and, above all else, heavily reliant on strong, automated software testing [2]. In this thesis, however, I focus on methodologies related to the development aspects of Agile software engineering – practices directly related to the implementation of software systems, rather than on process management methodologies.

Agile development methodologies need to be selected for and tailored to each project. Practices that work with one development team or with one project manager or with one customer may not work with others, which can make the term “Agile” quite an ambiguous one in practice. Several terms exist to describe sets of methodologies that work well together, such as Scrum (a set of methodologies for Agile project management) and Extreme Programming (a set

of methodologies for Agile software development). Because of this, it's common to see companies using more than one type of Agile (for example, Scrum with XP) or simply select their own sets of Agile methodologies. Portmanteaus are also commonly used to describe combinations of methods (f.ex.: ScrumBan and Scrum-but) make distinctions between methodologies even harder to find. However, some of the more well-known and well-defined Agile methodologies are:

- ❖ eXtreme Programming (XP) [3]
- ❖ Scrum [4]
- ❖ Crystal Clear [5]
- ❖ Lean [6]
- ❖ Kanban [7]

The principles underlying all of these methodologies were first laid out explicitly in the Agile Manifesto, published online in 1998 [8]. The Manifesto lays out four principles of Agile, with the items on the left providing more value than the items on the right (emphasis original):

- ❖ **Individuals and interactions** over processes and tools
- ❖ **Working software** over comprehensive documentation
- ❖ **Customer collaboration** over contract negotiation
- ❖ **Responding to change** over following a plan

Note that these principles imply that working, vertical slices of functionality should be presented to the customer on a regular basis. “Working” is generally defined as “passes all tests” (including acceptance tests – tests performed to demonstrate that functionality meets a user’s expectations of how a feature was expected to work), and a “vertical slice” of functionality is a feature that extends from the GUI of the application through the application logic and into any back-end data sources in order to demonstrate a feature working from end to end. By developing vertical slices instead of horizontal slices – for example, developing the entire database layer, then the entire business layer, then the entire user interface – it’s possible to have customers evaluate the feature as it is completed and get their feedback as quickly as possible. An example

of this can be seen in Figure 1. This approach allows software developers to interact with customers from a very early stage and embrace their changes throughout development, which helps reduce the cost of such changes as well as increase customer satisfaction [9].

Whereas the development of horizontal slices of functionality would allow developers to completely finish one layer of functionality and assume that it will not change throughout development of further features, this vertical style of development means that each part of the system can be expected to change throughout development as various features fuse together into a cohesive whole. The constant state of change that this creates enforces that developers have a strategy in place for quickly understanding when one part of the system changes in such a way that the parts of the system that rely upon it no longer function correctly. This is where automated software testing comes into play: automated tests can be used to encapsulate developer and/or customer expectations so that these can be verified against the system during its development and run just like code so that repeated test execution is virtually effortless. Agile puts emphasis on various types of automated tests for a variety of purposes, including verifying that a feature is implemented to do what the customer needs it to do (acceptance testing) and

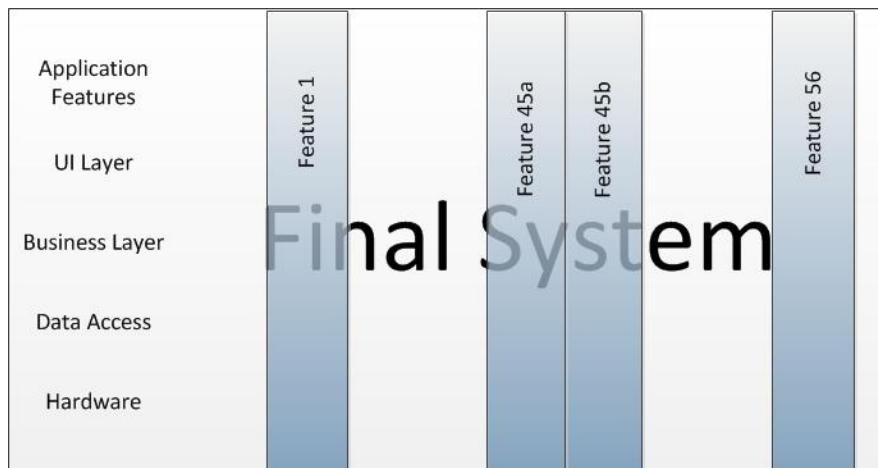


Figure 1: Vertical slices through system from iterative and incremental development (originally from [9]).

verifying that features still work after additional development work or refactoring has been done (regression testing). These two approaches to testing are fundamental to Agile software engineering (see, for example, Chapter 4.2). Without automated testing, it becomes difficult to continue any style of Agile software engineering as, among other things, the amount of effort required to repeatedly perform testing manually takes up too much time.

To illustrate the centrality of testing in Agile software engineering, consider the interrelations between Agile development methodologies commonly associated with XP as discussed in work by Vanderburg [2]. Figure 2, reproduced from that paper, makes it clear to see just how central software testing is to XP. Out of the 13 practices depicted in this diagram, nearly every non-testing practice relies on testing in some way (the exceptions in his model being “coding standards” and “planning game”). In fact, more practices depend upon testing than on any other practice. Based on this I would posit that, as GUIs simultaneously become both more complex (compare, for example, Windows 7 to Windows 8) and more central to applications, the continued success of Agile will be in large part dependent upon our ability to understand and create automated software tests that test the system end-to-end through its GUI – just like a user would.

2.2 Software Testing

The purpose of software testing is to find *bugs* so that they can be fixed. The term “bug” refers to any sort of problem with the system that could lead to a failure that could impact a user’s experience. It’s important to note, as Dijkstra quite famously pointed out, that “testing shows the presence, not the absence, of bugs” [10]. That is to say, if a suite of tests is written and run and discovers several bugs, the tests have proven that those specific bugs exist and effort should now be expended to figure out how to fix them. This doesn’t prove that those were the only bugs in the system – in fact, it is impossible to write and/or run enough tests to prove that

even simple functionality is absolutely correct [10]. Testing should be seen as an attempt to gain confidence that a system meets the expectations of its developers and users.

A series of events must happen in order for a bug to occur. First, some *error* must be made. An error could be as simple as a developer making a typo in the code or as complex as the users not clearly expressing the requirements that are documented and used to drive development. Errors can result in the creation of *defects*, or the possibility that the system can deviate from its expected behaviour. When a defect does lead to a system entering a state that it should not have been able to enter, this is known as a *fault*. Once a fault has occurred, this can lead to a *failure* – behaviour that contradicts the user’s expectations, or a discrepancy between what a feature is expected to do and its actual behaviour or a degree of loss of functionality for the user. The term bug is slang, and may be used informally to refer to any of the above; however, I will always use “bug” to refer to a violation of users’ expectations. There is not necessarily a one-to-one relationship between defects, faults, and failures. The same fault can be

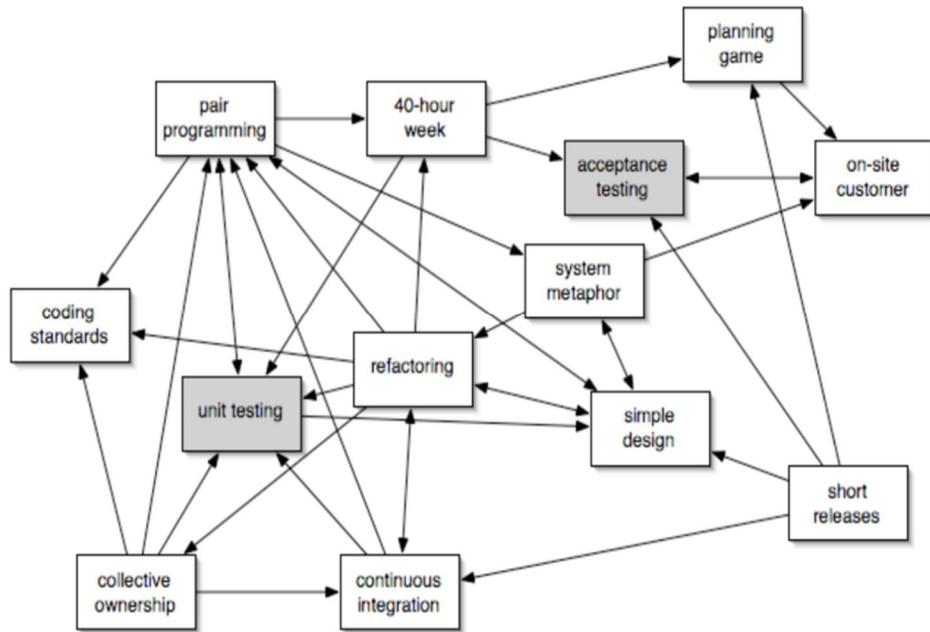


Figure 2. Glenn Vanderburg's 13 XP practices, from [2].

expressed by multiple defects in the code of an application, and a single failure can be caused by different faults or require more than one fault in order to occur.

Finally, an important aspect of software testing is the reality that bugs aren't bugs if no one cares about them. There are far too many bugs in any system to fix them all, so developers must always focus on the bugs that are most likely to impact the user in a significant way. Software testing is of necessity a triage process wherein bugs are identified, prioritized for handling based on the importance of the features affected, the likelihood that the bug will be triggered, and the severity of the bug, and then localized to a specific part of the codebase. This is important given that there is always a time limit for software development and so the most dangerous bugs – those most likely to be noticed by and interfere with users or those that occur most regularly – are handled first.

A test fundamentally consists of two parts: a *test procedure* and a *test oracle*. A test procedure consists of a series of interactions with a system under test (SUT) that are required to bring it to a state where verifications can be performed. It should be noted that the basic ability to carry out the steps of a test is, in a way, a form of verification. Tests that lack a test oracle are sometimes referred to as *smoke tests* under the understanding that, if it's not even possible to carry out the test procedure, then there are definitely bugs in that part of the system. However, realistically, a test oracle – a set of expectations about how the system should function – will be necessary to catch bugs. In the process of testing a system, bugs are exposed via the execution of a test procedure, but will generally only be noticed if a test oracle manages to notice the discrepancy between expected and actual states of the SUT. It is entirely possible for bugs to be triggered during the course of a test run, but not actually cause a failing test because the oracle was not designed to detect them.

Test procedures and oracles are both defined either in a test script (for a human user to use when manually running a test to ensure consistency between test runs) or as an automated test (a test that can be run like any other executable program). There are of course distinctions within these categories. In terms of automated tests, randomized tests look for methods in code or widgets in an interface and send random input; much like with smoke testing, if the system responds to random input by crashing or violating generic expectations, there is a bug. In terms of manual tests, exploratory testing is a process of continuous learning, test design, and test execution in which a person uses his/her expectations about how a system should behave to attempt to force the system into an unacceptable state.

One of the main reasons that Agile development methodologies rely heavily on test automation is that automation can reduce the overall effort involved in running tests, especially when tests are run repeatedly. This is because, ideally, an automated test should be written once, maintained infrequently, and run automatically very frequently. This compares favourably to manual testing, in which the same process is followed but running a test takes a significant amount of a tester's time. Where tests may not be run frequently, teams may decide that it's less effort to simply run them manually.

However, there's more to automated testing than the potential to save effort – an automated test can have a variety of purposes which are unrelated to finding bugs. For example, tests can help clarify system requirements in a test-first environment (*tests as specification*). It's not uncommon for acceptance tests to be automated and used by a team so that developers know when a feature is complete: when its tests pass, a feature is done. Tests can also be used to help understand what the system actually does (*tests as documentation*). An automated test both tells and shows us what the system is expected to do. Automated tests can also help us track down the

location of a defect when a bug is detected (*defect localization*). It's rarely obvious, in a complex system, what piece of code leads to a given bug, and tests can help reproduce and find the origin of bugs quickly. These uses for tests demonstrate that automated tests can be used to improve communication between developers and the system, between developers and other developers, and between the entire development team and their customers [11].

Tests can be designed to interact with a system from the perspective of customers or of developers. Developer-facing tests often deal with the correctness of the system's code. These tests are often used at the unit or integration level of the system to validate developers' expectations about what the code they produce is supposed to do. Specifically, unit tests demonstrate that individual methods or classes within the code work as expected in isolation, while integration tests demonstrate that methods of classes can work together to accomplish a goal. While these tests are good for increasing confidence in the correctness of the system's code, they do not demonstrate that the system meets its users' needs. For example, it's entirely possible that a developer would be able to trigger a feature by calling methods in a system's code, but this does not mean that a user would be able to do this through the system's GUI. Customer-facing tests are needed for this. Tests that interact with a system's GUI are good examples of customer-facing tests in that they are designed to interact with a system in the same way that customers would, can be shown to customers to validate that they meet the customer's expectations, and can be used as a stand-in for customer expectations.

Test-driven development (TDD) takes testing one step further. Rather than defining tests after the code they relate to is complete – the traditional approach to testing – developers and testers can create tests based on their expectation of what code should do *before* writing any of

that code. This ensures that developers have considered potential defects that could be coded into the system before implementation has even begun, thus avoiding these issues in the first place.

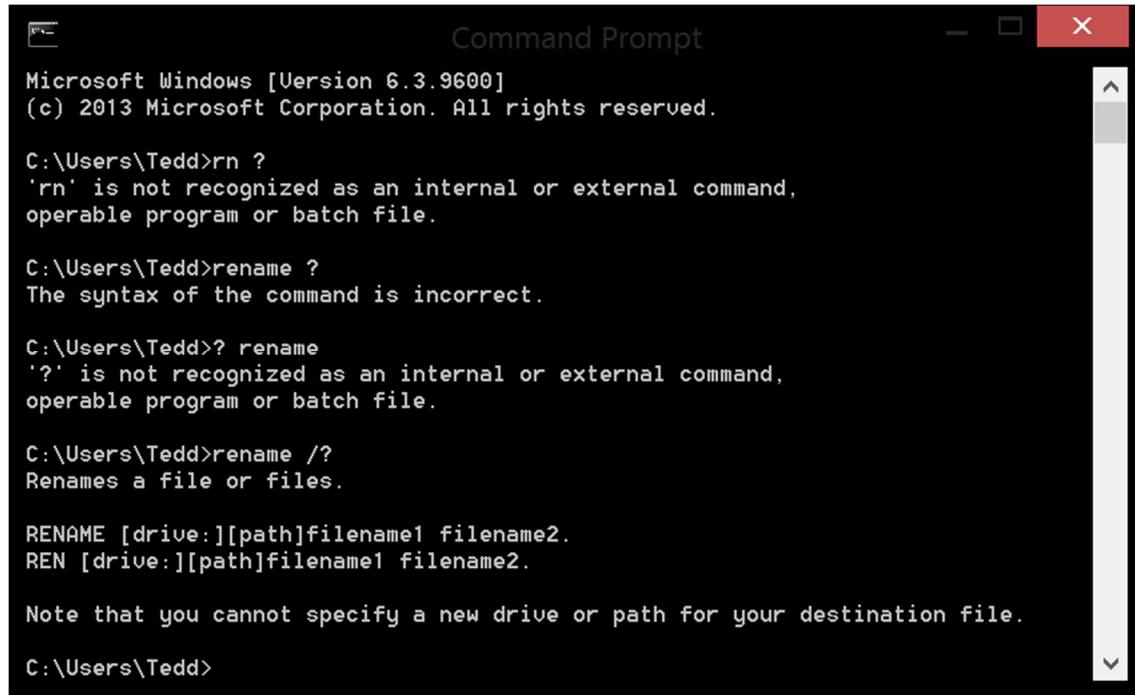
2.3 User Interfaces

User interfaces (UIs) are a crucial part of every modern application. From a user's point of view, the UI *is* the application. If features don't work or aren't accessible from the user interface, they may as well not exist.

A UI is, simply put, the part of an application that supports interaction between the application and its user. It is the part of the system through which users send data and commands to the application, and it also the part of the system through which the application sends results and data back to the user.

The first major paradigm shift in UI technology was the advent of *command-line interfaces* (CLIs), commonly known as *consoles*. CLIs take user input through a keyboard and display results through monitors – usually in the form of simple text. The result of this is that only very discrete inputs are acceptable. This means that CLIs allow very little freedom to users – there are a limited number of commands allowed to users – and users must learn these commands before they can begin to interact with each application. This means that users have to be trained to use an application before they can begin working on tasks.

Graphical user interfaces (GUIs), on the other hand, allow users a large amount of freedom of interaction, including the ability to discover how to use an application through the use of that application. This concept is a key part of what makes GUIs so important: GUIs can make applications more accessible to untrained and/or non-technical users. GUIs are important to applications, then, in that they make applications more accessible than they would otherwise be to a wider range of users.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\Tedd>rn ?
'rn' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Tedd>rename ?
The syntax of the command is incorrect.

C:\Users\Tedd>? rename
'?' is not recognized as an internal or external command,
operable program or batch file.

C:\Users\Tedd>rename /?
Renames a file or files.

RENAME [drive:][path]filename1 filename2.
REN [drive:][path]filename1 filename2.

Note that you cannot specify a new drive or path for your destination file.

C:\Users\Tedd>
```

Figure 3: Common CLI demonstrating an issue with discoverability.

GUIs improve upon CLIs by taking user input using a pointing device (traditionally, a mouse) in addition to a keyboard. The basis of a GUI is a system of graphical elements, or *widgets*, that receive user input or display responses from the application. GUIs are usually grouped together into containers like windows or frames in order to make information easier for users to understand and simplify interaction, and operating systems typically allow multiple applications to display GUIs simultaneously (the exceptions being some tablets computers and cell phones that were not designed to focus on a single application at a time).

With GUIs and CLIs, there is typically a distinction between input devices and output devices, but recent advances in touch-based technologies are removing this distinction. Mobile phones (f.ex. iPhone, Android, and Windows Phone devices), touchscreen computers (f.ex. the iPad, Asus EEE Slate, and Microsoft Surface, or really through any computer hooked up to a touch-enabled monitor), and digital tabletops (f.ex. the Samsung SUR-40 PixelSense table, SMART Board, and Evoluce One) all make use of this technology. Cutting-edge touchless

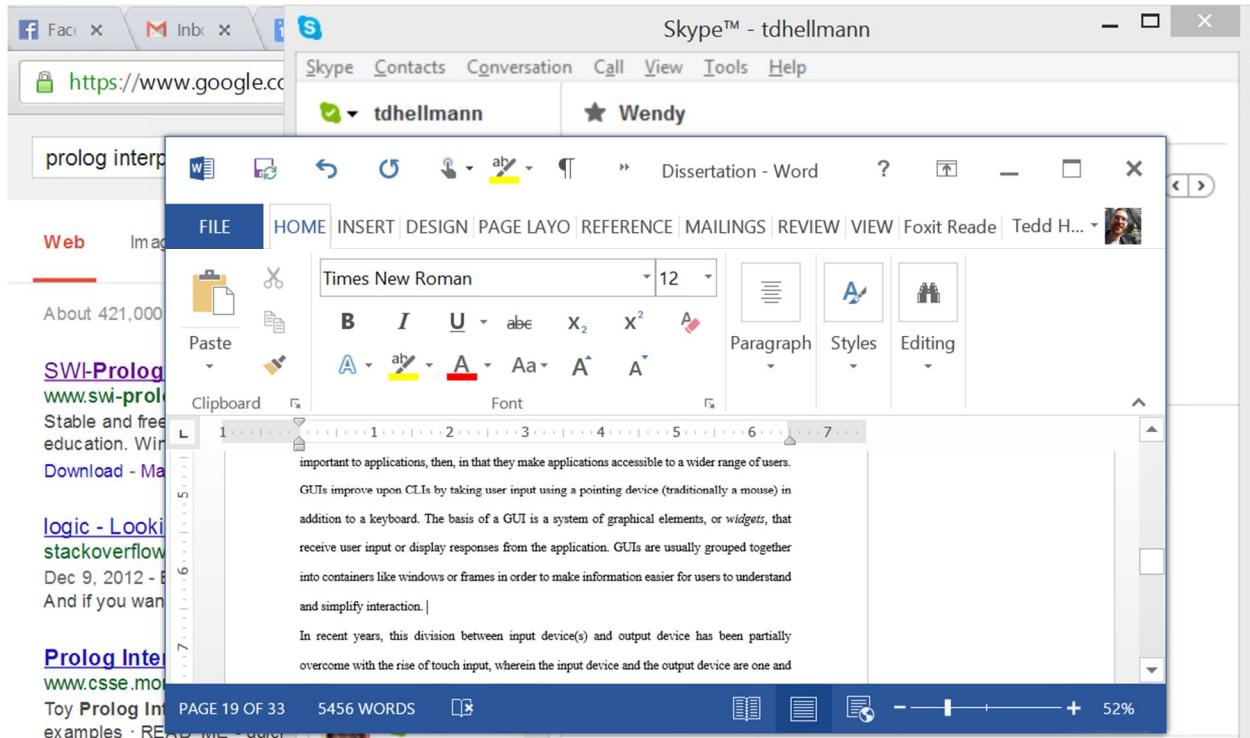


Figure 4: Several overlapping GUIs, demonstrating windowing and widget grouping.

interaction technologies (like the Microsoft Kinect, Leap Motion controller, and PlayStation Eye) are making it possible to interact with a computer without even touching it, but at the cost of restoring the division between input and output devices.

2.4 Testing Graphical User Interfaces

GUIs are well-understood and testing is generally well-understood, so it would be easy to assume that useful processes for testing GUIs exist. GUI testing is in reality quite a complicated, difficult, repetitive, and expensive process.

A very common argument regarding GUI testing is that GUI testing actually doesn't need to be done at all. This argument is based on the premise that it is possible (and advisable) to remove almost all application logic from GUIs. While powerful patterns for doing this do exist (f.ex. the Model-View-Controller pattern), there is still a significant amount of code present in a GUI – in fact, research on business software in actual use in industry has found that 45-60% of

an application's code is dedicated to its GUI [12]. Research has also found that GUIs are indeed a significant source of bugs. One industrial case study found that 9% of all bugs in the systems that were analyzed were errors in the code for the systems' GUIs [13]. In fact, GUI bugs were the 5th most common class of bugs in these systems [13]. Another industrial case study found that 60% of software defects found by users after release relate to GUI code, and of these defects, 65% result in a loss of functionality [14] – which, again, reinforces the importance of GUI tests that interact with the system in the same way as the users who are finding all of these bugs will. Despite advances in the way GUIs are architected, it remains a fact that GUIs contain a significant amount of important code and that this code is a dangerous source of important bugs.

While GUI testing is definitely necessary, it is definitely not easy. The simplest way to perform GUI testing is with manual testing, wherein a human tester interacts with an application to verify that its responses are correct. A human tester can easily interact with an application and has some advantages over automated tests in recognizing when bugs occur, but manual testing is slow, error-prone, and boring. If testing is done frequently – as is expected in an Agile development environment – then manually testing a GUI quickly becomes unfeasible.

Automated GUI tests are, if anything, more difficult to use than manual GUI testing. GUI testing is almost always cross-process testing – testing in which the test code runs in one process while the application code runs in a separate process. This is due to the way in which GUIs are structured and is significantly different from the bulk of standard developer-facing testing techniques, in which test code and application code usually run in the same thread or in threads within the same process. Cross-process testing is frustrating in that code running in one process is unaware of and does not have direct access to code in another process. Whereas in a unit test it

is simple to create an instance of an object and send input to a method to trigger specific functionality, GUI testing is performed through the GUI – an interface which only makes certain information available. It's not possible to just create a widget in isolation, or to invoke a single method, or even to directly access all information about the GUI. A test must first find the widget that can trigger functionality, interact with it in the way in which a user would, and then verify that the result is as expected in the same way. This adds a significant amount of complexity to the testing process.

For example: consider a test which checks to see if logging in to a web-based application using a username/password combination is possible. First, an automated GUI test must identify which widgets to enter username and password strings into. This could be done by scanning the DOM tree of the application for widgets for information that would indicate that a given widget is the username or password field (as is done in tools like Selenium²) or it could be done by performing pattern-matching of a previously-recorded image on the screen itself (as is done in tools like Sikuli³). In the former case, the tester needs to know extremely specific information about the widget in question – which may change either as the software is changed or over the runtime of the application, or may be automatically-generated and meaningless to humans. In the latter case, the widget must be visibly unique in order to be correctly identified, and cosmetic changes to the interface can interfere with correct identification. After identifying the correct widgets to interact with, the test must then send data to the widgets as a user would. This could involve sending mouse and keyboard events directly to the operating system in order to indirectly cause events to be triggered in the GUI. This is error-prone as some of the easiest ways of sending input (f.ex. instantaneously setting the mouse coordinates to the center of a textbox)

² <http://docs.seleniumhq.org/>

³ <http://www.sikuli.org/>

may not trigger the user interface correctly (f.ex. the username textbox may not accept input from the keyboard unless it has received focus from the mouse moving into its screen space, which may not happen if the mouse cursor is simply moved directly to these coordinates). Further, verifying that the input sent to the textboxes was received correctly may be non-trivial – what if, as in many cases, the textbox for the password doesn't show the exact characters entered and functions for getting the text contents are disabled (as would be prudent from a security standpoint)? All of these issues must be resolved before a test can even verify whether the system's response to input was as expected.

Tools do exist to simplify this process, but they carry their own dangers. Most GUI testing tools work on the capture/replay paradigm. In capture/replay, testing tools monitor the set of interactions between a human tester and the system and record these steps so that they can be replayed later as automated tests. However, capture/replay tools (CRTs) do not tend to record tests in a human-readable manner, meaning that it is much more difficult to modify an existing test than to record a new one. Still, CRTs remain quite easy to use in the creation of tests. This frequently leads to the situation where testers create a large number of automated GUI tests (AGTs) which cannot easily be modified and may fail confusingly. Not only is it difficult to debug an AGT to see where exactly the test detected a failure, but it is often the case that AGTs report that the system under test is broken when in reality the test itself has not been updated in response to changes to the system – leading to a situation in which the test itself looks to be the cause of the failure. This can lead to a large amount of “broken” tests that either waste time – by requiring effort to be spent on repeatedly updating/fixing the test – or are re-recorded instead of repaired. The latter is a problem because, in the case that the test was indicating the presence of a real bug, rerecording a test on the system without adequately understanding the cause of the

failure in the first place runs the risk of cementing that bug in place – since fixing the bug in the future would cause the newly-recorded test to fail, which would at best be confusing.

Further to this, Atif Memon has published a series of papers on various properties of test procedures and test oracles on bug detection specifically for GUI-based systems; the general result is that long test procedures [15] and complex test oracles [16] produce the best results, but long and complex tests are generally more expensive to create and maintain. To complicate matters further, an automated GUI test will be stronger if it triggers each possible event from a large number of states [17]. In essence, the larger and more complicated a GUI test suite is the more likely it will be to detect bugs.

As stated previously, there are two main ways to address these difficulties: better tool support or better *process* support. In previous work, I investigated one technique for using artificial intelligence to provide better tool support for automated GUI testing [18] [19]. To build upon this work with my dissertation, I investigate issues with the process of automated GUI testing and provide recommendations that improve it by making it more suitable to an Agile development context.

Chapter Three: Related Work

There is a large body of literature on both testing in Agile development environments and automated GUI testing individually, but there are relatively few papers that investigate a combination of the two. While the studies presented in Chapter Five focus on investigating the set of papers on Agile and automated GUI testing, respectively, this chapter focuses on related work that could be used to integrate the two practices.

3.1 Capture-Replay Tools (CRT)

Many approaches to the use of automated GUI tests in Agile development environments focus on the use of CRTs for the production of tests. However, a recurring theme in these papers is that the authors believe there is a fundamental contradiction between a capture-based approach and a test-first approach to testing [20], [21], [22], [23]. In order to capture a series of interactions with a user interface so that they can be replayed as a test, that interface has to exist, which implies that development work needs to be done before tests can be created:

“Many agilists (especially advocates of eXtreme Programming) would argue that [CRT] test automation completely undermines the notion of automating acceptance tests before the functionality is built because it requires the SUT to exist before the tests can be recorded” [20].

Some authors posit CRT-based testing as an option when working with a legacy system, noting that “since we already had an existing version of the system to use as a ‘gold standard’, we could leave the effort of defining the expected outcomes to the record and playback framework” [20]. In this way, it is possible to use tests defined through interaction with the previous version of the system to make sure that the updated system is functionally-equivalent.

One group of authors has taken a strongly divergent approach to integrating automated GUI tests into an agile process by using very non-agile methodologies. Similar to the approach described in Chapter Six, they use user interface prototypes as a basis for test definitions.

However, rather than instrumenting prototypes with information that will allow CRTs to work with them, the authors propose manually generating formal specifications of the prototypes [22]. This involves the creation of a description of the user interface's expected behavior in Z-spec, which allows a tool to automatically generate expectations of the interface's behaviour. In order for this to work, intermediate layers between the Z-spec definition and the actual user interface must be created, leading to a large amount of overhead – all of which will need to be updated if the GUI is changed. Overall, this approach represents a huge increase in complexity (which we take in this thesis as a measure of the possible states that a system can enter or the number of test cases that would be necessary to determine whether it is functioning correctly) – even the tests for the simple calendar application they present are very complex – and a large step backwards in terms of agility since it would make it extremely difficult to embrace any sort of change to the system after the tests were defined.

One publication demonstrates how simple it can be to record tests using a CRT before a GUI exists. Sikuli is a CRT that works by locating and interacting with widgets using image capture when a test is recorded and matching recorded images with sections of the screen when a test is run [24]. This means, quite simply, that it is possible to record a test based on a high-fidelity prototype created in Photoshop or a similar image editing tool and then run this test against the actual system as it is developed. A small downside to this approach is that it is geared towards a different type of prototype than are discussed in Chapter Six, meaning that tests recorded in this way can only be run against the actual implementation of the GUI. Further, the authors make assumptions about the uniqueness of widgets, stating for example that a script including “click(X) clicks on the close button”, which is incorrect – the script will cause the tool to click on *a* close button, which, when there are multiple applications open at once, will not

guarantee that the *desired* window was actually closed. However, the basic approach itself is very easy to work with and is one of the few truly new approaches to GUI testing to be published in recent years.

3.2 Maintenance

Some publications seek to make automated GUI tests more compatible with Agile by reducing the amount of time spent maintaining test code. As Meszaros has stated:

“If the behavior of the system is changed (e.g. the requirements are changed and the system is modified to meet the new requirements), any tests that exercise the modified functionality will most likely fail when replayed. This is a basic reality of testing regardless of the test automation approach used” [20].

This is in fact expected behaviour – if code is functionally altered and its corresponding tests do *not* fail, then they were never capable of detecting bugs in the first place and therefore useless⁴. However, nearly all authors who publish on this subject note that a major cause of wasted time on GUI testing is related to “fixing” tests that are broken when the GUI is updated but system functionality is not changed [25], [26], [20], [21], [27], [24], [28], [9], [29], [30], [9]. Further, some authors note that, when using CRTs, it will be faster to re-record a broken test entirely rather than to attempt to understand and repair a test [27]. This is based on the assumption that CRTs record tests in human-unreadable scripts. This can be a huge amount of effort – it’s possible for up to 74% of test cases to be “broken” after modifications to an application’s GUI [31] [32]. This means that most of the tests in a suite must be either fixed or recreated from scratch, either of which represents a significant amount of rework.

In many cases, “broken” tests aren’t actually broken, but some other aspect of the test environment has interfered with a test run to cause a false positive – as Meszaros put it, a “loss of synchronization between the test tool process and the SUT process” [20]. To avoid wasting time

⁴ This is the premise behind mutation testing – changing code to make sure corresponding tests fail.

on these instances, some authors suggest creating a simple test harness that re-runs all failing tests, reporting only tests that fail twice in a row as legitimate failures – which prevents failures caused by, for example, system events like popup notifications that may interfere with a GUI test run from being reported as failures of the system [28]. This does, of course, create the possibility that features containing bugs that only intermittently cause failures from being brought to attention. Authors have also studied the feasibility of maintaining automated GUI tests as a system evolves, and have found that in large part maintenance of tests is possible and can extend the lifespan of tests [33], [24], [28], though at a cost: [28] specifically points out that a significant investment of time (one hour per test to be repaired) was required every time a GUI test was broken by changes to the GUI.

In certain cases, it may be possible to perform an automated repair of a broken test without human intervention. Work out of Atif Memon’s group has focused on attempting to largely automate the process of repairing tests after the code they reference has been changed. His group has proposed using genetic algorithms [34], heuristic approaches [35], and a compiler-inspired techniques [31] in attempts to repair broken tests without the need for much (if any) human interaction. However, one needs to question whether these approaches makes sense: if a test failure is automatically assumed to be erroneous – and automatically repaired without human intervention – what was the point of the test in the first place? This sort of approach could be valuable in cases where developers communicate to testers that no functionality has changed between one commit and the next, or when developers are able to select which test cases a system should attempt to automatically repair, could be counterproductive if it became the default response for every test failure. Tests need to be able to fail and allowed to communicate a

violation of expectations to developers and testers when they do; otherwise there really isn't a point to testing.

One approach to the issue with maintenance effort has been to provide tool assistance for determining the root cause of a failure. One set of authors proposes that the most likely cause of failures would be changes to the types of widgets in a GUI, and to deal with this they propose tools for automatically inferring the types of objects in a test script [25], [36]. The intent of this is to provide testers with more contextual information about the widgets under test in order to help them determine the root cause of a failure. If, for example, a tester understood that a test was failing because it was attempting to click an element that was of the wrong type to respond to click events, it could help to make the cause of the test failure more obvious.

One advantage of the way Sikuli has been designed is that the development environment for tests allows the images that will be used to replay a test to be embedded in a test itself – meaning that testers can directly see what a test is trying to locate and interact with, making it very easy to debug and update a failing test [24].

Tests can also be structured intelligently in order to reduce the impact of test failures. For example, any test suite that uses helper methods for interacting with widgets improves the maintainability of tests – if a widget is removed or its behaviour is altered, a fix might only need to be made in the helper method to repair any test that made use of that widget. This is one of the proposed benefits to using model-based testing for GUIs – in fact, it's posited as one of the major benefits of Actionable Knowledge Models [37]. Some tools take this one step further and provide functionality for compositing recurring sequences into single events for the purpose of testing – for example, taking all the low-level events involved in clicking through menus to save a document and creating a single “save_file” event for the purpose of testing [23].

3.3 Architecture

Research has also looked into how to structure test suites and GUIs to make automated GUI testing more applicable in Agile development environments. Unfortunately, publications on this subject tend to recommend refactoring the GUI so that it can be bypassed during testing. This doesn't meet the definition of GUI testing used in this paper, but I will discuss it briefly here given that these authors all assert they are performing GUI testing and also because it is a major theme of the interview study presented in Chapter Four.

Early publications involving architectures for GUI testing rehash design recommendations originally presented in the original paper on the Model-View-Controller pattern [38]. The basic idea behind Model-View-Controller is to separate the data that is represented by the GUI, the logic that controls the GUI, and the display of the GUI into different components in order to garner a number of benefits, including the ability to test all of the logic of the GUI without ever interacting with the GUI itself. Instead, methods in the Controller are triggered and data in the Model are altered in order to assess whether the logic functions correctly and the appropriate event handlers are called. In testing setups that leverage *mocking*, expectations about which methods in the View should be called and/or which methods should receive calls from the View are also used to determine correctness. Mocking is the practice of replacing one unit of code with a non-functional (but compiling) unit that is easier to test and can make testing code with inconvenient dependencies less difficult. Work presented in [39], [40], [41], and [29] suggests methods for separating out or hooking into the methods related to the actual logic represented by interactions with widgets in the GUI and testing only those. This may be easier to accomplish, but it also leaves the parts of the GUI that the user actually interacts with untested. Perhaps tellingly, the authors of these papers often assert that changes to the view

of a GUI do not imply changes to the logic of the application itself (see for example [29]), which is certainly not a valid assumption when thinking from a user’s point of view.

A number of GUI testing tools allow the steps for creating a reasonable architecture for tests to be skipped, but achieve the same result of allowing GUI logic to be accessed programmatically. For example, [42] automatically creates a set of objects that “reveal the internal state of a particular source class” so that they can be accessed from test code operating in another process. The xUseCase frameworks [43], [27] work in a similar way by inserting an API between the GUI and the rest of the application that can be used to detect or fire events in either direction for the purpose of testing. This approach is interesting in that it can also be set up such that the GUI can inform the test, via an event, when it is ready to proceed with the next step of a test. This means that a test will never proceed with a step before the GUI is ready, which could alleviate some of the frustrating issues relating to test code and application code losing synchronization due to test events executing before the GUI can properly receive and react to them.

To a large extent, this is complementary to how the approach in [30] functions – an application-specific API is inserted into the testing tool. When a test needs to interact with the GUI, it calls the application-specific API in order to perform those interactions.

Researchers seem split on whether tests should be structured such that they interact with each other. One group argues that tests should be as independent and self-contained as possible [21] while still making use of helper methods. These researchers argue that keeping tests isolated from each other allowed tests to be rearranged and refactored without worrying about the impact on related tests. Other authors argued that a large number of tests will repeat the same set of steps to guide the system to a state where the logic of the system is available for testing, and that

this represents a sort of branching structure [28]. From this point of view, a high degree of coupling in tests makes sense, with short tests branching off from a main sequence of interactions. However, the side effect of this is that any failure along the main branch of a test could cause a large number of tests to fail, and, if the fix is not simple and/or fast, this could be a significant setback. Interestingly, my previous work on combining rule-based artificial intelligence with a CRT, described in [19], can be viewed as an improvement of this approach – individual rules branch out from each test case to expand test coverage, but none of these branches will be run if the base test case fails, meaning that the tester would be presented with a single basic test failure rather than one failure for each rule that would have been run.

A single paper has got close to the version of a 3-layer architecture discussed initially in Chapter Four. In [44], the authors propose separating a test into a “test control tier” (which controls test objectives), an “action tier” (which contains sequences of events), and a “keyword tier” (which contains low-level interactions with a GUI). However, this publication focuses on splitting an event model of the system into different domains such that it is easier for an automated tool to automatically generate tests from it rather than discussing the use of a multi-tier architecture for increasing test maintainability.

3.4 UITDD

Most of the CRTs mentioned earlier won’t work if a GUI doesn’t exist, which essentially rules out straightforward TDD (though Chapter Six discusses how TDD can be made to work with these tools). This leaves two alternatives for UITD: either writing tests by hand, or finding a way to make a CRT work without a GUI. The former is nothing new – all that’s required is a human-readable test language – but the latter is rather counter-intuitive. In fact, many papers comment on how this approach isn’t possible (see, for example, [20]).

Any CRT that records its tests in a human-readable language can be used test-first. In this situation, testers simply write tests manually in the language that a CRT would use so that they can be run against the GUI once it has been created. Tools like Selenium make this highly possible, and in fact this avenue has been explored by researchers [21]. When writing tests test-first using CRTs, these authors recommend a process along the lines of (emphasis mine):

1. Write tests before development
2. *After* development, get them to green; then
3. Refactor as mercilessly as possible [21].

Notice that the essential difference here is that tests pass *after* development – when, normally, we would consider development to be complete only after the tests for the new work are passing. This inversion is necessary because, according to the authors of that publication, the GUI that is developed seldom matches the manually-created tests. This implies that the authors see the tests generated during TDD of GUIs to be more of use as specifications the final system should meet rather than real tests that are intended to be run against the system and detect regressions as development advances. However, one wonders why such specifications would be automated, in this case. However, in light of this specification-first approach, some tools also provide support for using CRTs to modify parts of test scripts that need to be updated after development of the actual GUI has been completed [23], [27].

Of course, it's possible to write GUI tests by hand directly in existing testing frameworks. Several frameworks exist that can simplify this process of manually writing GUI tests [45] [46].

One team has also taken a rather counterintuitive approach to test-first development of automated GUI tests – which may stem from a rather unique interpretation of the purpose of TDD. The authors propose that testers (manually) create a formal model of the GUI in Z notation. This model would be based on UI prototypes that would be produced as part of an Agile-UX process (similar to the prerequisites of my approach in [47] and [48]). Their tool then

automatically generates GUI tests from the pre- and post-conditions expressed in the formal model [49], [22]. However, this approach is not practical for use in an Agile development environment given the complexity of creating and maintaining formal models of a system – especially one that can be expected to change as frequently as a GUI. Such a process would prevent changes from being fully embraced, as each change would require a large expenditure of effort to update the formal model.

The approach presented in [24], which leverages image recognition to identify portions of a GUI that are important to a test, is quite brilliant – because Sikuli relies on computer vision to find and interact with widgets in the GUI, it's possible to simply record tests from high-fidelity prototypes of the GUI. These tests can then be run against the SUT as soon as it is a reasonably close match to the prototypes. It's also trivial in Sikuli to replace the image used to identify a widget, so it would be completely appropriate to use a create-and-modify approach similar to [21] with Sikuli.

Chapter Four: Practitioner Motivation for Automated GUI Testing⁵

In order to investigate GUI testing from the viewpoint of practitioners, and also to make sure that the results of this dissertation research are useful to people who actually perform automated GUI testing as part of their normal work process, the first study I present is one that investigates how these tests are used. To accomplish this, a series of semi-structured interviews were conducted with people who had used or were currently using automated GUI tests. These interviews were used to investigate three core topics relating to automated GUI testing:

- What do practitioners use AGTs to accomplish?
- What issues do practitioners encounter that frustrate them?
- What techniques have practitioners found useful for overcoming these issues?

The transcripts of these interviews underwent qualitative data analysis and open and closed coding in order to extract results. In addition to providing insight into each of these research questions, our study found two central problems that are not discussed in existing literature: test suite architecture and test suite co-evolution with the GUI and underlying system. According to the interviewees, these issues represent significant challenges to the effectiveness of automated GUI tests.

4.1 Methodology

Semi-structured interviews were conducted with 18 participants with varying experience in automated GUI testing. All interviews were transcribed in full and a two-stage analysis was performed on the transcriptions: first, on the full set of 18, and second, on the set of 8 participants with significant experience with GUI testing only. These analyses involved open coding [50], sorting of codes, and identification of cross-cutting categories.

⁵ This chapter is based on work published in [51].

4.1.1 Participant Demographics

18 participants with experience in the creation, maintenance, or use of automated GUI tests were recruited for this study. All participants had some amount of experience with GUI testing, but some participants had only short-term experience. It became clear during analysis that participants with only short-term experience with automated GUI tests tended to have superficial issues specific to the tools they were using (e.g., figuring out how to use a specific tool) rather than issues with the use of automated GUI tests generally (e.g., issues with test suite maintenance). Because we wanted to focus on general issues with automated GUI tests, we excluded interviews with participants with limited experience from the second phase of our data analysis, leaving 8 participants with more than a year of experience for the second stage of analysis.

Information about these more experienced participants' level of experience, area of employment, and primary role is provided in Table 1. In this table, participants with more than one but less than two years of experience with automated GUI tests were categorized as "junior;" participants with more than two but less than five years of experience were categorized as "intermediate;" and participants with more than five years of experience were categorized as

Table 1: Participant demographics.

ID	Experience	Employment Sector	Primary Role	Interview Length (minutes)
1	Junior	Academia	Developer	12
2	Senior	Academia	Professor	20
3	Senior	Industry	Tester	60
4	Junior	Industry	Developer	27
5	Intermediate	Industry	Developer	33
6	Intermediate	Industry	Developer	60
7	Senior	Industry	Tester	40
8	Senior	Industry	Tester	60

“senior.”

All but one participant (Participant 2) had industry experience, even though two (Participant 1 and Participant 2) were currently in academia.

4.1.2 Interview Design

Semi-structured interviews were used because of the exploratory nature of this study on the one hand but the need for consistency between participants on the other. We felt that structured interviews – interviews in which a set of pre-defined questions are asked to all participants with no deviations – were too rigid to allow us to gather useful results, while unstructured interviews – interviews with only a general theme and no pre-defined questions or structure – would have been too inconsistent. A standardized set of high-level initial questions were used to start every interview:

“The last time you were using AGTs...”

- “what were you trying to accomplish”
- “what functionality were you targeting with these tests”
- “what GUI testing tool were you using, and do you still use it”
- “what issues did you encounter during the testing process”
- “what was your process when a test failure was reported”

Issues that participants brought up in their responses were then followed up on in greater detail. These interviews ranged in length from 12 to 60 minutes with an mean duration of 39 minutes. Because a standardized set of high-level questions were used with all interview participants, even the shorter interviews were useful in obtaining results about the core questions investigated in this study. This starting set of questions also means that this study could in part be replicated by other researchers. However, follow-up questions about participants’ responses sometimes evoked quite detailed responses, which was the reason behind the longer interviews. The semi-structured nature of the interviews both means that they study is less repeatable than it would have been

with a fully-structured, invariant set of interview questions. However, by asking follow-up questions about the specific techniques participants used to overcome issues, we are able to present a much more interesting set of recommendations in this study.

4.1.3 Analysis

The analysis was split into two phases. Phase One operated on the full set of 18 interviews and was used to discover general topics of importance to participants. Phase Two sought to identify themes important to the restricted set of 8 participants with significant GUI testing experience and focused only on the topics discovered in Phase One. This allowed focus to be placed on the issues most pertinent to these more experienced practitioners.

First, a round of open coding [50] was performed in order to find the general topics of highest importance to interviewees. This was done as a collaborative effort between the myself, Elham Moazzen, Abhishek Sharma, and Md. Zabedul Akbar, the first 4 authors of the resulting paper [51], which is important because it allowed a more thorough analysis to be done than would have been possible by a single researcher working alone. The three general topics discovered through this process were: goals; issues; and best practices.

These results were used to inform Phase Two of the analysis. The first step in Phase Two was to perform selective coding again on this set of eight interviews. In this round of coding, only topics directly related to one of the three topics identified in Phase One were coded. This round of selective coding allowed us to focus in on specific areas of interest while being more flexible than closed coding – we still extracted keywords from the interviews themselves instead of seeking to apply a preselected list of terms to them. This selectiveness helped to focus and constrain the analysis to a small set of very important topics and was done by having a team of two researchers examine each interview initially, then having at least one other researcher re-

examine their work; this was done to reduce individual bias and increase the consistency of the results.

Two important themes were discovered through the second analysis: test suite architecture and test suite co-evolution with the GUI and underlying system. Each theme was further subdivided into issues that participants overcame, best practices they used to overcome these issues, and issues that participants had not been able to solve. The following subsections explain the goals that participants expressed, along with the themes of test suite evolution and test suite architecture.

4.2 Goals of Automated GUI Testing

The first goal of the interviews was to investigate what the goals of automated GUI testing actually are from the perspective of practitioners. The analysis of the interview transcripts uncovered two main uses for automated GUI tests: acceptance testing and regression. This subsection covers these topics as well as some additional interesting issues that participants brought up less frequently.

4.2.1 Automated Acceptance Testing

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

Six participants (2, 3, 4, 5, 7, 8) used automated GUI tests to verify that "this software meets the user's requirements" (Participant 3). Not all of these participants agreed on where these 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 acceptance tests can derive

from their own expectations of how the system should behave. Participant 2, for example, created automated GUI tests 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 acceptance tests in that he derived customer’s expectations from design artifacts like written specifications and user interface prototypes – there was no direct communication or collaboration with the system’s intended customers/users. In both cases, participants are using automated GUI tests to demonstrate that a feature is working from the point of view of an end-user.

4.2.2 Automated Regression Testing

Automated regression tests are used to alert developers that a regression – functionality that was working in a previous version of the system but stops working after additional development work – has occurred. Five of our participants (2, 4, 6, 7, 8) used automated GUI tests to detect regressions. This is important to “make sure that things are not breaking as we move to a new version” of the system (Participant 4). The regression tests 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 interesting in that interviewees noted that there was a gap in the current recommended approach to GUI development and testing – namely, to use Model-View-Controller or a similar pattern, assume the GUI code is too simple to possibly fail, and then write unit tests to cover the business logic of the application. However, our participants noted that bugs could be present in the linking between the GUI and the business logic in practice, and that “unit tests won’t catch those” (Participant 8). These participants felt that automated GUI tests were fundamentally necessary for detecting a class of regression bugs that other approaches to testing were not able to catch.

Participants 6 and 8 added new automated GUI tests when regression errors were caught by human testers. Participant 8 relied on GUI-level regression tests 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 automated GUI tests to make it safe for him to experiment with “several variations... and still see that effectively those different approaches could have the same result.”

4.2.3 Other Goals of Automated GUI Testing

Our participants also expressed several other motivations behind the creation of automated GUI tests. Participant 8, for example, emphasizes that “you have to make sure everything gets tied together and works properly.” Participant 2 uses automated GUI testing to “fine-tune what is the problem that I am facing.” Participants 3 and 7 additionally want to reduce the amount of effort required to test their systems, “to make sure that we don’t have to physically do those tests every day” (Participant 3). Participant 3 also uses automated GUI tests 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.”

4.2.4 Other Issues

From these results, automated acceptance testing and automated regression testing are the primary use cases for automated GUI tests. The participants noted two additional major difficulties. First, participants sometimes ended up automating GUI tests with lower defect detection potential than the test they originally envisioned. At present, they lack guidance that would make it clear to them what sort of tests they should be automating – even given that there have been several academic publications on this subject (for example, [16]). 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. Not only does this end up wasting time, but it also undermines the trust that developers should be able to place in their test suites.

4.2.5 Discussion

Meszaros in [11] 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 the findings of this interview study:

- Tests as specification
- Tests as safety net
- Risk reduction
- Bug repellent

However, the other four goals that Meszaros lists do not match up with the results in this section:

- Tests as documentation
- Defect localization
- Ease of creation/maintenance
- Improved quality

In follow-up studies, it would be useful to look into this matter further to determine if these latter four goals really are not present as goals for automated GUI testing – and, if so, why not?

4.3 Test Suite Evolution

Automated GUI tests, on a very basic level, reflect aspects of a system under test; as the system changes, tests that refer to it will also need to change. Test suite evolution was a major theme in many of the interviews and remains a major issue for participants despite the body of work published in academic circles (see Chapter 3.2).

4.3.1 Challenges Posed by Evolving GUIs and System

Of the 8 participants, 5 (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 automated GUI tests. 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, automated GUI tests can report failures while the system under test is actually functioning as expected. Notice that this is how tests work generally – if features change, but tests stay the same, then of course the tests will fail. If they don’t fail, then they were not actually capable of detecting a bug in the first place, but participants consistently stated that tests failing when code was changed was a legitimate issue. With GUI tests specifically, the participants felt as though tests failing in this situation was a problem.

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 automated GUI tests will require ongoing maintenance. Participant 6 found that an automated GUI test 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 automated GUI tests is more complicated in light of the fact that 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 (defect

localization, from [11]). 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 a test 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. This implies that the problem of understanding an automated GUI test is twofold: first, figure out if the system is at fault or if the test is; if the system is at fault, next, figure out what part of the system is at fault.

These results are reminiscent of [28], an experience report in which the authors report a similar situation: a new intern joins the team specifically to automate manual GUI tests using Selenium; the intern creates a large suite of automated GUI tests; something changes in a part of the application relied on by most of these tests, causing a large number of tests to fail; testers were forced to sift through the failing tests to determine which were simply broken and which were indicating bugs.

4.3.2 Best Practice: Comprehensive Test Maintenance

Out of the five participants that encountered issues with automated GUI test suite evolution, three (3, 7, 8) had developed ways of mitigating the impact of these issues. The first recommendation participants had for ways to decrease the burden of test suite evolution was to remember the essentially reflective nature of GUI tests. Since automated GUI tests should reflect the way in which a feature works, updating tests 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 automated GUI tests 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 test suite passing. While this will quickly get the system back into a green state, it fails to acknowledge that a failing automated GUI test should indicate a problem with the system (as was noted earlier in Chapter 3.2). Participant 8’s mindset acknowledges that automated GUI tests must co-evolve with the system to continue to reflect the end-user’s expectations. In both contexts tests 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 test suites 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, automated GUI tests 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 tests won’t have any realistic prospect of detecting errors. The system will quickly evolve past the point where its GUI tests are relevant. Participant 3 improves his automated GUI tests continually both to ensure that the feedback they provide is good and to continually increase their defect detection potential. From this viewpoint, test suite evolution provides an opportunity to improve the value an automated GUI test can provide and should be viewed as a way to offset the cost of maintenance.

However, it's also quite possible that the overhead of maintenance for some automated GUI tests will become too high to justify the value they provide. Participant 7 is adamant about making sure that each test has a purpose: "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.

4.3.3 Open Issues

Participants 3 and 7 raised several additional issues regarding test suite evolution. First, there is a likelihood that automated GUI tests 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 automated GUI tests to require more maintenance and to take longer to run than other types of tests means that the cost of putting up with stale automated GUI tests is higher. Another consequence of the fact that GUI tests run at a very high level is that it's difficult to determine the point at which the number and quality of 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 automated GUI tests, 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 automated GUI test to last? Both Participants 5 and 7 found that their automated GUI test 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, which may mean that a significant amount of the test suite would need to be redone.

It’s worth noting that most of the issues raised in this subsection exist in other forms of testing, so it’s interesting that professional testers would list them as issues specific to GUI testing. It may be the case that this is because tools for creating automated GUI tests provide less support than those that exist for unit testing (despite the academic work in this area as presented in Chapter Three). For example, there is only a small amount of experimental support for technologies like syntax highlighters within the GUI domain [26] [52]. Within a typical IDE, for example, changes to a method signature cause compile errors that are immediately apparent, whereas this is not generally the case with automated GUI tests.

4.4 Test Suite Architecture

Many of the difficulties participants experienced had to do primarily with how they structured their test suites. The tight coupling between test code and GUI implementation and the low reusability of automated GUI test code was largely due to the fact that these tests tend to be created using a single-layer architecture where testing goals, business logic, and widget interaction are lumped into the same entity.

4.4.1 Problems Caused by Single-Layer Architectures

4 participants (1, 4, 7, 8) encountered issues related to the way their automated GUI tests 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. 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 automated GUI tests, 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, 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 automated GUI tests 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. (For an example of why this is problematic, imagine the DOM tree of a UI wherein elements are accessed using their *XPath*, an attribute that uses a widget’s parents to identify its exact location within the DOM tree. Assume that a widget is hard-coded as “Parent.Element” and that, when an element like a frame is added to surround all of this, the path would correctly look like “Frame.Parent.Element”. For every

instance of “Parent.Element” in the code, this element would need to be updated to read “Frame.Parent.Element” – which is suddenly and dramatically nontrivial as soon as there is a possibility of more than one “Parent.Element” existing anywhere in any test, which can be common.) 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.”

Automated GUI tests written using a single-layer architecture and a high level of detail also 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 automated GUI tests as “custom test code with very little generic application” (Participant 1). This focus on creating highly-detailed, highly-customized test code gave some participants a tendency to create a new test from scratch rather than making use of existing test code.

One reason for this could be the popularity of CRTs among our participants. CRTs make it easier to create tests, but they usually record test scripts in a domain-specific language. These automated GUI tests 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.

While this thesis does not generally consider testing of web-based systems, the problems with code duplication become extremely obvious when trying to test a web application on

multiple browsers using automated GUI tests. “You have to look at every browser,” Participant 6 found, “and you have to add special rules or special cases for every browser.” While CRTs like Selenium provide support for running tests against multiple browsers⁶, the problem is that different browsers may respond differently to actions or display elements slightly differently, meaning that custom code may have to be built into tests specifically for different browsers. In this situation, a single change can necessitate updates in many different affected automated GUI tests and in the many different versions of each of those tests.

4.4.2 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 Figure 5.

The first point participants raised was the need for some amount of modularity in the automated GUI tests they were building. For example, Participant 1 liked a feature of Selenium that allowed him to “create modules; for instance, a login method,” which he can reuse “rather than having to redo an entire segment

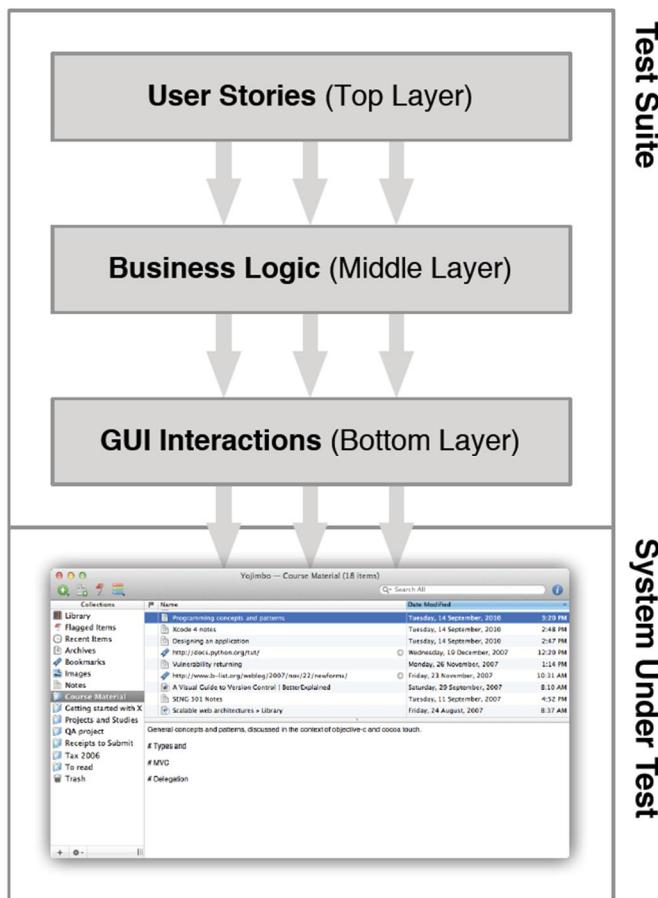


Figure 5: Multi-layer AGT suite architecture showing calls between layers.

ers (docs.seleniumhq.org/about/platforms.jsp), or the title of the page than other browsers.

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 automated GUI tests.

Participant 8 felt that better automated GUI tests 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 automated GUI tests. This means that, to an extent, a suite of automated GUI tests structured in this manner should be safeguarded against software re-architecting. An investigation of the effect of the structure of GUI tests suites is one of the directions for future work listed in Chapter Eight.

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 automated GUI test 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 automated GUI tests in a way that will require overwhelming effort to repair.

4.4.3 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.

4.4.4 Discussion

Unfortunately, not much work has been done in the area of GUI test suite architecture. Most papers that discuss architecture and GUI tests describe ways of restructuring GUIs to make a system easier to test, and usually end up describing approaches that aren’t quite GUI testing – they suggest accessing the system beneath the GUI through tests rather than testing the system as a whole (see Chapter 3.3).

4.5 Conclusions

This chapter presents the results of an interview study focusing on 8 experienced developers/testers, leading to insights into goals, issues, and best practices of automated GUI

Table 2: Issues and corresponding best practices.

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

testing. The major goals that participants had for automated GUI tests were acceptance testing and regression testing. Further, test suite evolution and test suite architecture were significant issues for participants. In terms of test suite evolution, participants that had dealt with this issue had discovered that automated GUI tests need to co-evolve with the system they operate on, need to be upgraded as the system or GUI changes, and some tests may not be appropriate for automation. In terms of test suite architecture, participants found that a three-layer test suite architecture made it easier to maintain, understand, and reuse their automated GUI tests. The common issues and recommended practices for dealing with them are summarized in Table 2.

Chapter Five: Mapping Academic Work on Automated GUI Testing

In the previous chapter, I presented the results of an investigation into how automated GUI tests are used by testers and developers. However, this investigation does not cover the body of work produced by academics and published in peer-reviewed conferences, workshops, and journals. In fact, not one participant in the previous chapter's study made reference to academic literature during their interview, even in cases where a solution to issues they had raised had been proposed in academic literature, which may indicate that the practitioner view of automated GUI testing is somewhat different from the academic view. In order to further investigate what automated GUI tests can be used for and where future work on the topic can be most effective, I additionally conducted three systematic mapping studies on Agile testing and on GUI testing. In this chapter, I summarize the results of these explorations.

Systematic mapping studies take a large number of papers as input and categorize them based on their titles and abstracts. The result of such a study is a framework for understanding a field of research at a high level – in other words, to find out what topics a given field encompasses. These studies are gaining popularity within software engineering research, but must be approached with caution [53] due to concerns about their repeatability. Specifically, there are concerns with respect to the repeatability of systematic mapping studies stemming from the search criteria used to find papers initially and the inclusion/exclusion criteria used to narrow down search results into a final paper set for the study. In order to deal with some of these issues, I took two precautions with the following studies on agile testing:

1. The initial search and inclusion/exclusion steps were done collaboratively.
2. The investigation into Agile testing was done once with an automated paper search [54] and once with a manual paper search [55] in order to provide a basis for comparison.

However, the systematic mapping of GUI testing itself was done solely by me.

Because this is, to my knowledge, the first time a set of comparative systematic mapping studies has been done with an explicit goal of comparing and combining the results of different search methodologies, the comparison of an automated paper search against a manual paper search for papers on the same topic may be of interest to researchers outside the scope of this dissertation.

5.1 Agile Testing⁷

In order to understand how Agile and automated GUI tests can best be combined, it is important to understand what Agile testing is already associated with in existing work. In my investigation of this topic, I conducted two systematic mapping studies: one using an automated search and including 166 papers on the subject [54]; the other using a manual search and including 110 papers on the subject [55]. Significantly, the overlap of the papers in the second study with the papers found for the first study was only 17 papers. On the positive side, this means that, while the two studies asked many of the same questions, since the papersets were largely distinct, we can compare the two sets of results against each other to increase our confidence in them. On the negative side, this raises some questions about whether the many existing systematic mapping and literature review studies already in the literature did in fact manage to collect enough relevant papers to produce generalizable results – a question that is, unfortunately, outside the scope of this dissertation and will need to be revisited in the future. However, given that I do have two sets of results to compare against each other, this section focuses on comparing the results of these two studies in order to get a rounder view of the field of Agile testing.

In this section, the field of Agile testing will be investigated through the following research questions:

⁷ This work was published at Agile 2012 [54] and Agile 2013 [55].

- RQ1. What has Agile testing been used for?
- RQ2. What types of research have been published?
- RQ3. Do academics or practitioners drive the field?

The second publication additionally looks into:

- RQ4. What problems are associated with agile testing?
- RQ5. What benefits do publications claim to offer?

5.1.1 Methodology

5.1.1.1 First Study

For the first study, the methodology followed the guidelines provided by Petersen et al. [56]. Each step in this investigation involved at least two, and usually three, researchers. This was so that disagreements could be resolved through discussion in order to provide higher reliability. RQs 1-3 were used in order to design our search string:

- “Agile” and “test” in Title-Abstract-Keywords

This search was intentionally quite imprecise in order to attempt to catch a large number of papers from the field, but it’s worth noting that using specific subterms within each keyword (for example, searching for “XP”, “eXtreme Programming”, “Kanban”, “Lean” etc. in addition to “Agile” may have turned up more papers). It’s also worth noting that publications at Agile conferences did not actually use “Agile” as a keyword, probably due to redundancy; this is the reason these papers were not identified until the manual search in the second study.

The search was run against the SciVerse Scopus⁸ and IEEE Xplore⁹ databases, resulting in an initial paperset of 894 papers. No inclusion criteria were used, but papers were excluded if they were outside the field of software engineering or not discussing software testing. Two rounds of exclusion were carried out: one on titles only; one on abstracts, only continuing on to read the body of papers where the abstract and title were not sufficient to make a decision. The

⁸ <http://www.scopus.com/>

⁹ <http://ieeexplore.ieee.org/>

paper set was narrowed down to 283 papers after the first pass and down to the final set of 166 papers after the second pass.

5.1.1.2 Second Study

As with the previous study, at least two researchers participated in each step of the paper collection and analysis process and followed the Petersen guidelines insofar as possible. As opposed to the automated paper search used in the first study, the second study used a manual search. Rather than defining a search string to use to query database engines, the search instead started with the set of all papers published at the Agile, XP, and XP/Agile Universe conferences and excluded papers not related to testing from this initial set.

There were 993 papers in the initial set for this study, all dating between 2002 (the first year one of these conferences was established) and 2012 (the most recent year available at the time of the study) – 99 more than were found by the automated query used in the first study. The first elimination step was to remove papers that did not explicitly discuss testing or a testing-related practice (like continuous integration, which relies on automated testing). This was done on the basis of paper titles with researchers only continuing to read the abstracts of papers where the title was not clear enough to make a decision. In this first pass, the paperset was reduced to 139 papers. A second elimination step was done on the basis of paper type: we wished to focus on research, so workshop proposals/summaries, tutorials, and demos were excluded. This left a set of 110 papers, of which 17 also appeared in the paperset from the previous study. This allows us an opportunity to perform a sort of triangulation of what topics comprise Agile testing from two largely orthogonal sources: those that explicitly describe their techniques as Agile but do not publish at Agile conferences and those that discuss testing topics at Agile venues.

5.1.1.3 Processing Papers

For both studies, the papersets underwent qualitative data analysis and open and closed coding in order to extract results.

For RQs 1, 4, and 5 (“What has Agile testing been used for?”, What problems are associated with agile testing?”, and “What benefits do publications claim to offer?”), open coding was used. In open coding, the keywords used to describe papers are drawn out of the papers themselves. Papers were assigned low-level keywords based on their titles and abstracts. These keywords were then grouped together into higher-level categories. The codes generated across the two papers were largely consistent given that I was heavily involved in the coding process for both papers, which makes a great deal of sense given my deep involvement in this field.

For RQs 2 and 3 (“What types of research have been published?” and “Do academics or practitioners drive the field?”), closed coding was used. In closed coding, a predefined set of keywords is simply applied to papers, again based on their titles and abstracts. For RQ3, the keywords used were simply “industry”, “academic”, or “both”, depending on where the authors worked at the time of publication. For RQ3, the keywords used were taken from [57], a paper that provides a framework for grouping papers into different categories based on the type of methodology used. The categories, explained below in Section 5.2.1.2 in greater detail, were: Solution, Validation, Evaluation, Philosophical, Experience, and Opinion. We used this existing classification scheme without modification in both studies.

The keywords for papers were stored using EndNote¹⁰, but all visualizations were created using Excel¹¹.

¹⁰ <http://endnote.com/>

¹¹ <http://office.microsoft.com/en-ca/microsoft-excel-spreadsheet-software-FX010048762.aspx>

5.1.2 Detailed Results

5.1.2.1 What has Agile testing been used for?

The first step in the analysis was to look at the keywords used in papers in order to determine what agile testing was used for, generally, in these publications. Figure 6 and Figure 7 provide a visual comparison of the frequency of the top 10 keywords in each paper. Most of the top keywords in both studies line up – TDD, using tests as specification, unit testing, acceptance testing, acceptance test-driven development, testing web systems, and GUI testing are all in the top 10 topics for both studies. Some topics even appear at the same position within both studies: TDD is the most frequently-applied keyword in both, while GUI testing is the 10th in both studies. Several terms appear in the top results of the first study that are ranked lower in the second study and vice versa, however. Formal specification (mathematical attempts to prove a system's correctness – including, for example, the Z-Specification language), continuous integration (running all tests whenever any code is committed to source control), functional testing (testing focusing on entire features rather than individual modules or methods in the code), and performance testing (testing to measure non-functional performance aspects of the

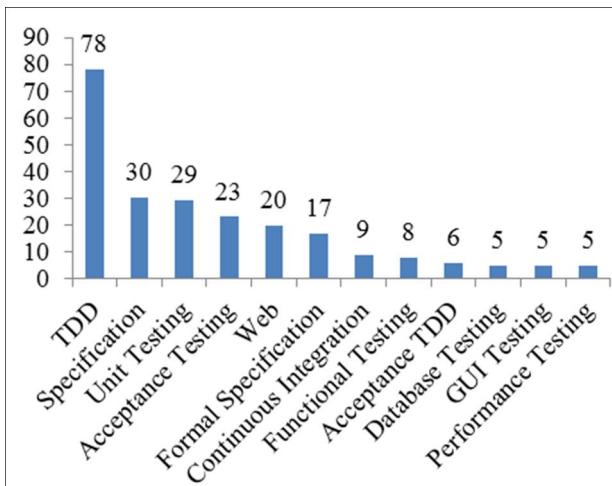


Figure 6: Keywords applied to papers used in [54].

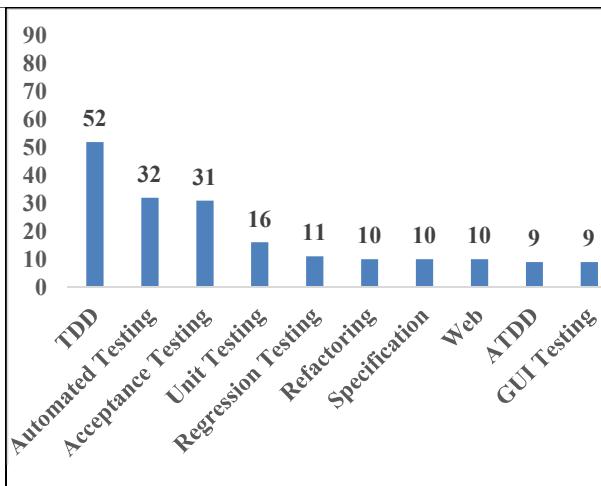


Figure 7: Keywords applied to papers used in [55].

system such as response time or ability to handle large numbers of simultaneous requests) were common topics in the first study but not the second, while automated testing, regression testing, and refactoring occurred in the second study but not the first. Notice that these topics are somewhat distinct – for instance, in this document, many of the top ranked terms from the first study had not been defined previously in this document while the topics from the second study, which gathered papers targeted at audiences interested in Agile development methodologies, have largely been discussed previously. It's possible that this is due to a differing understanding of the term “Agile” between papers on agile testing published at non-Agile conferences (predominant in the first study) and papers on testing published at Agile conferences (consistent throughout the second study).

Specifically of interest in this study is the intersection of TDD as the most common topic and GUI testing as 10th most common topic. In the first study, the intersection of TDD and GUI testing is only four papers [58] [40] [47] [22], with the second-most recent [47] being written by the present author. In the second study, the intersection is again four papers [29] [21] [59] [48], with the most recent of which [48] written by the present author. Note that there is no overlap between the papers that discuss both Agile and GUI testing picked up by each study. This was, again, due to the information used to search for these papers. For example, the two papers I wrote were not picked up by both studies for very straightforward reasons: [47] was not picked up by the second study because it was not published at an Agile-focused conference and [48] was not picked up by the first study because it did not use the word “Agile” to describe itself. These studies are discussed in more detail in Chapter Three, but at present the main observation is that there were only 6 papers not by the present author uncovered in these two studies that discuss UITDD, yet independently TDD and GUI testing are important topics. Based on this, it's

possible to assert that further research on UITDD specifically is necessary, especially research which takes care to ensure that the results will be applicable within organizations using agile development approaches. In light of this, I conducted further research on this topic, the results of which are presented in Chapter Six.

5.1.2.2 What types of research have been published?

In order to ensure that my research was done in a way that strongly contributed to the field, it also made sense to find out what type of papers on Agile testing tended to get published. In light of this, papers in both studies were categorized based on the type of research they presented in accordance with Wieringa et al.'s categorization scheme [57]. Even though the classification scheme proposed in [57] was originally developed for papers discussing requirements engineering, it also works quite well for other subfields within computer science and has, at this time, been referenced in at least 93 Computer Science and Software Engineering publications, mostly systematic mapping studies¹². The scheme categorizes papers into the following types:

- **Solution** – a novel solution is proposed, but only a proof of concept (if any) is offered
- **Validation** – further investigates a solution, but is not evaluated in practice or rigorously
- **Philosophical** – provides a new framework for understanding a field
- **Opinion** – presents the author's personal opinions without evidence to back up claims
- **Experience** – describes the author's experience on a project in industry
- **Evaluation** – investigation of a problem in practice with a large, rigorous study

Overall, nearly 28% of all papers between the two studies were categorized as experience reports, with 22% and 20% being categorized as solution or evaluation papers, respectively¹³. Opinion papers made up the smallest percentage at only 9. At only slightly more than this, 10%, philosophical papers make up the smallest research-based contribution to the field of agile

¹² Based on a search for citations of that publication on Scopus conducted on 2015-04-17.

¹³ Percentages add up to more than 100 because one paper may be categorized as more than one paper type.

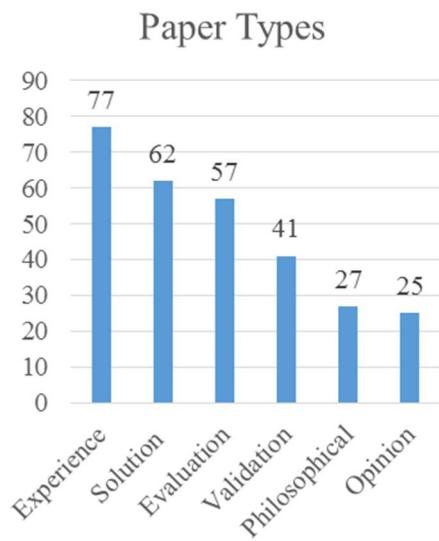


Figure 8: Frequency of types of paper found in both mapping studies [54] [55].

testing. There is also a lower amount of validation papers than might be expected, at only about 15% of the whole.

Superficially these numbers are a bit surprising. Intuitively, one might expect to see a kind of funnelling effect acting on the types of papers that get published: a high number of solution papers suggesting new directions for research, followed by a slightly lower number of validation papers following up on these suggestions with small validation experiments in a lab

setting, followed by a significantly lower number of evaluation papers done rigorously in partnership with practitioners in industry, followed by experience papers from practitioners trying techniques on their own, followed by a small number of philosophical papers organizing the field as a whole. However, this relies on a research model in which new ideas originate in academia, where they are evaluated iteratively in increasingly realistic studies, before being adopted by practitioners. This model for research progression, again, might make intuitive sense given that one would expect less-promising techniques to be weeded out before making it into practice or to more intense types of study.

However, it does not appear that research in this field is structured in this manner. Instead, it would appear that there is little evolution of approaches to testing in Agile development environments contained in these papersets: the vast majority of authors who publish on this topic publish precisely once (~89% [54], ~86% [55]). This data is visualized in Figure 9. Rather than thinking of papers as representing a kind of continuum projecting towards refined

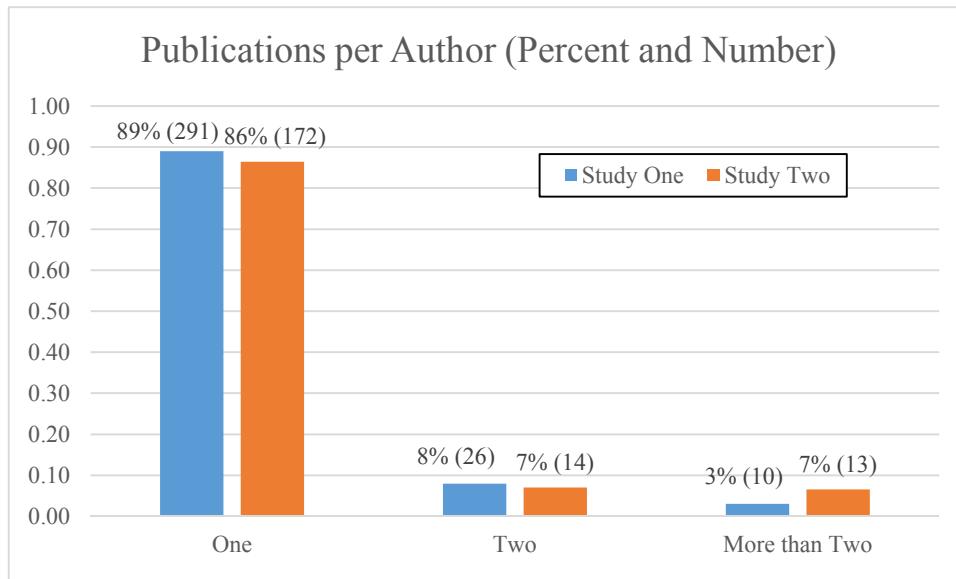


Figure 9: Number of publications per author in both studies (percentage of the total and absolute number per study). Bar heights are based on percentage.

results over time, it is likely more accurate to think of publications as a set of distinct points which must be viewed in aggregate to make sense. Hence the strong need for systematic literature reviews and systematic mapping studies, like the three presented in this dissertation.

Again, one might also expect to be able to see a progression over time, but this expectation was also not borne out by the data from these studies. Figure 10 shows the type of paper published over time using the results from both systematic mapping studies. From this figure, it's possible to see certain trends in the types of papers that are published over time. For example, philosophical papers tend to be infrequent, with most years seeing only between 1 and 3 new publications. The exception to this would be in 2010, when a spike of 8 new papers occurred. In terms of understanding this field as a whole, it would be interesting to see what causes these kinds of spikes and dips (like, for example, the number of solution papers published in 2008).

Paper Types by Year of Publication

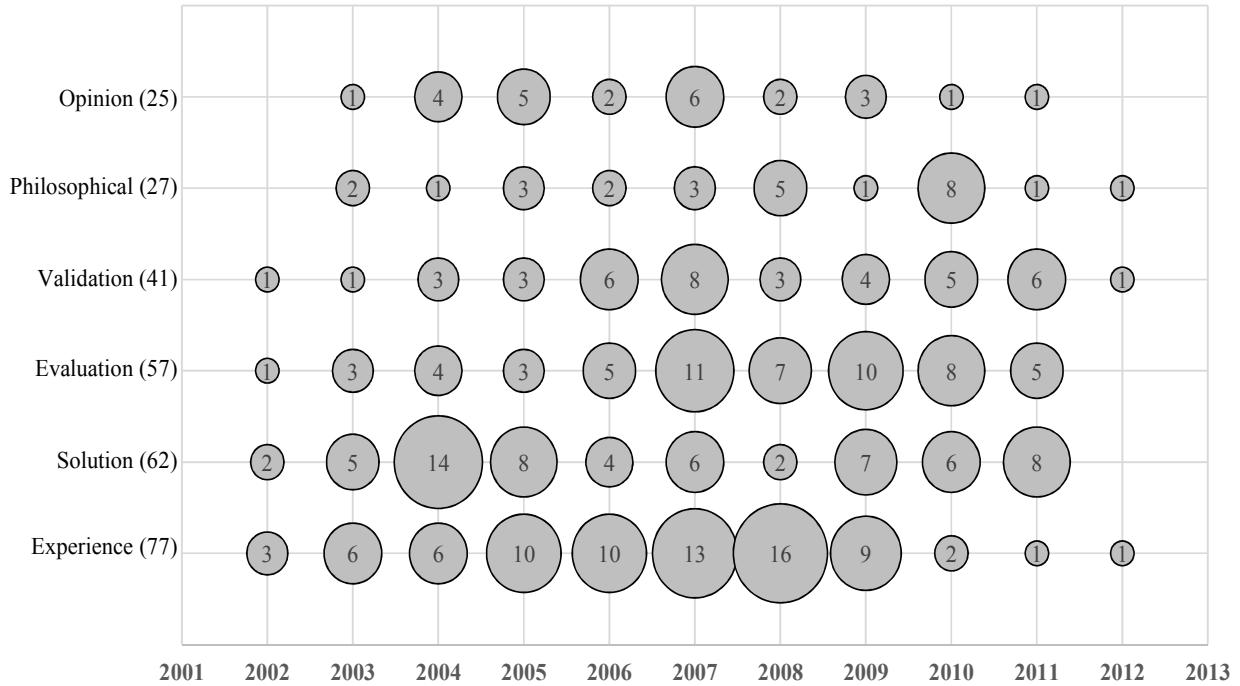


Figure 10: Types of paper published in each year, from both [54] and [55] combined.

There is a strong showing of experience reports in most years, though there is a strong decline after 2009. This is important in that it demonstrates that testing in Agile development environments is not just an academic topic, but is a topic of interest to practitioners in their work as well. But what are these practitioners using testing for? And what issues are they encountering? The following section looks into the differences between academics and practitioners in the field of testing in Agile development environments in more detail, while the two sections after that investigate these questions in more detail.

5.1.2.3 Do academics or practitioners drive the field of agile testing?

In order to understand if academics and practitioners publishing papers on testing in Agile development environments are approaching the topic in the same way, one of these mapping studies [54] also coded papers based on where the authors worked at the time of publication. If a paper only had authors from universities or from companies, that paper was

coded as “academic” or “industry” respectively. A paper would be coded as “both” if either it had a mixture of authors from universities and companies or if research was conducted by a body bridging the gap between academia and industry (such as, for example, SINTEF¹⁴).

Based on this analysis, the publications in the field were majority-driven by academics – about 56%, came from groups of only academic authors. Collaborations between academics and practitioners were rare, at about 14% of the total, with industry-only papers making up the balance of about 30%. While this is unsurprising – practitioners have essentially no motivation to publish research papers while academics must “publish or perish” – it is somewhat concerning in that it raises the possibility that, if academics and practitioners are looking at the field from different perspectives, academics could be working on problems that either aren’t apparent in industry or aren’t relevant to industry.

A breakdown of the number of publications from each group over time can be seen in Figure 11. Of the 166 total publications, 94 publications (~57%) were contributed by academics, 49 (~30%) were contributed by practitioners, and 23 (~14%) were contributed by collaborations between academics and practitioners. While in absolute terms practitioners have roughly half the publications of academics, these publications aren’t evenly distributed over time – that practitioner contribution of publications in this field nearly disappear after 2009. This is an odd development in that agile testing is a very practical problem – the disappearance of practitioners from research publications is striking because, based on Chapter Four and the present chapter, the sort of problems that practitioners express are not the sort of problems that academics are working on solving (as will be demonstrated in the following two chapters).

¹⁴ SINTEF is an industry-oriented research center funded by the government of Norway. <http://www.sintef.no/home/>

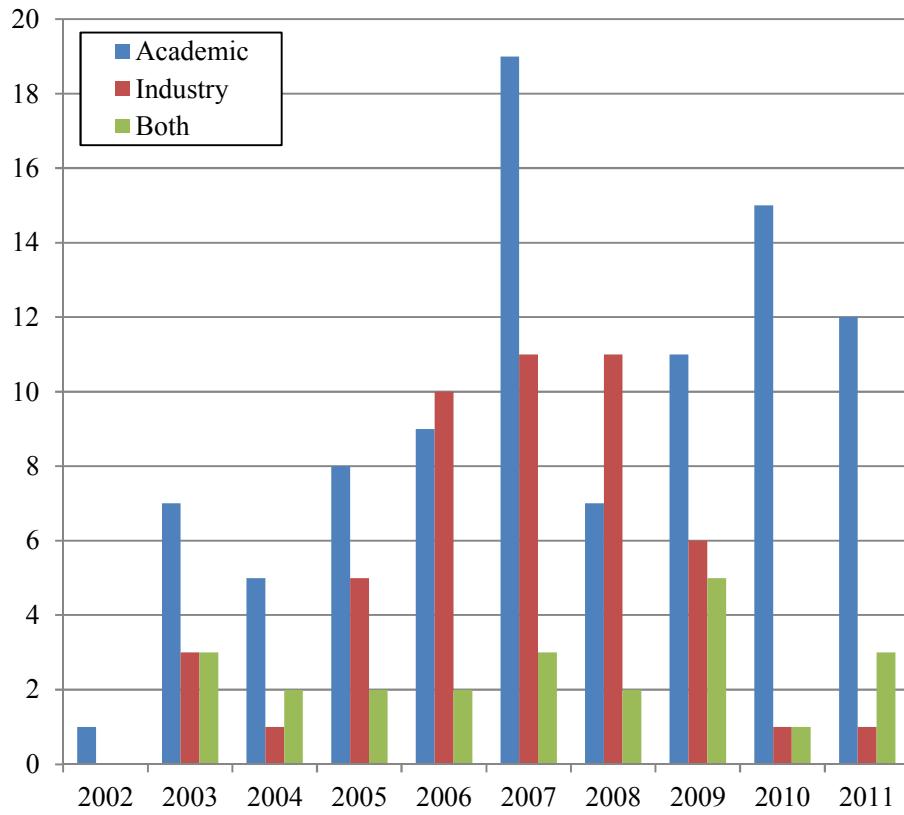


Figure 11: Publications over time, divided into separate categories based on where the authors worked at the time of publication.

The types of papers published by academics, by practitioners, and by collaborations of the two was also investigated – the results of this can be seen in Figure 12. Unsurprisingly, most publications by practitioners were either experience reports or opinion papers – 37 out of 49 papers fell into these two categories. Very few industry-only publications fall into any other category. One solution to this would be for practitioners to devote effort to formalizing their hypotheses before making changes to their testing processes – the key difference between simply reporting experiences and conducting research. Still, while it would be better for industry publications to be more rigorous, this isn't to say that these publications aren't useful. By reporting on what techniques for testing in Agile development environments did or did not work in practice, experience reports can serve to draw attention to areas where new research would be of benefit.

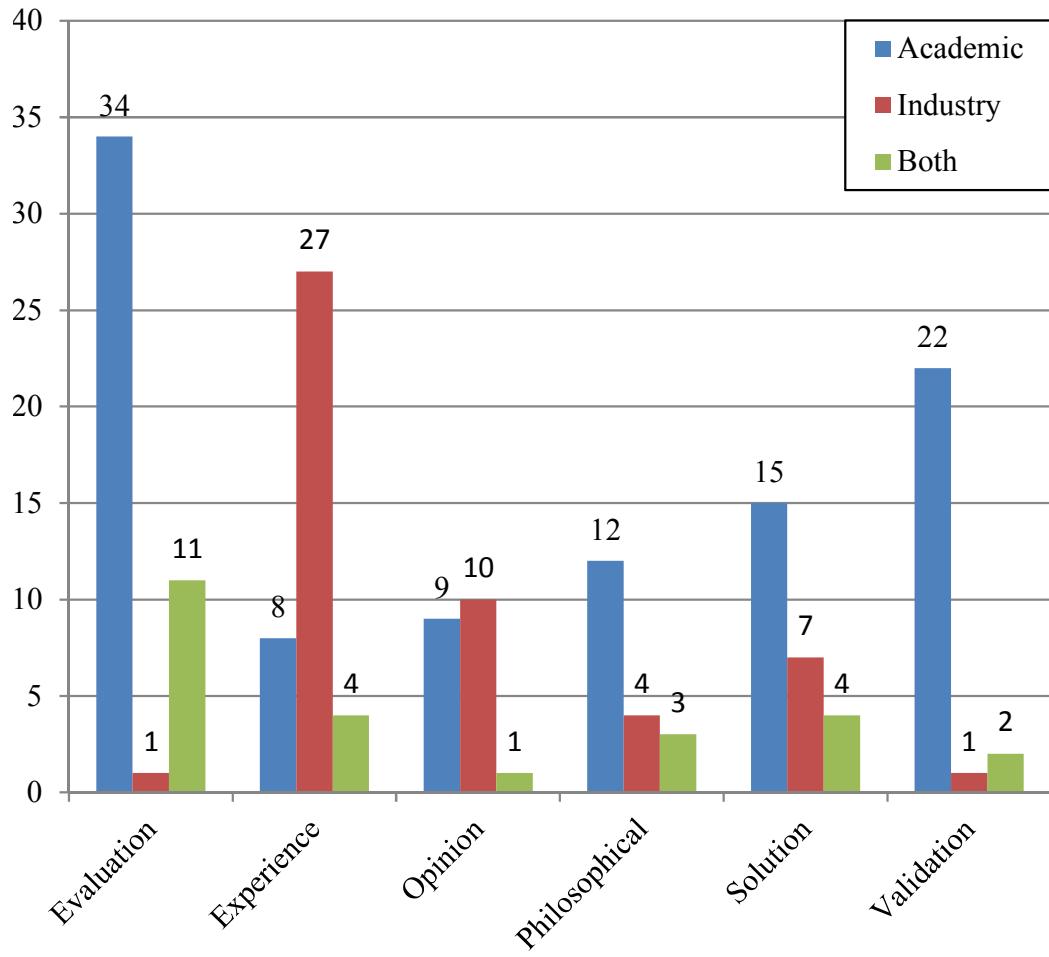


Figure 12: Types of publication by source of publication.

A majority of academic publications, on the other hand, fall into the evaluation and validation categories – 66 out of 94 within the two together. Again, this is as expected given that academic work should be held to a more rigorous standard and both of these types of papers will involve some sort of controlled experiment or action research study. Evaluation papers were also the highest type of study from collaborations between academics and practitioners, with 11 out of 23 publications of this type.

Overall, academics form the bulk of publications on testing in Agile development environments. This is somewhat dangerous given that academics approach the field in a significantly different way from practitioners, as is shown in the following two sections. Opinion

papers and experience reports tend to be grounded in practical experience, while evaluation, validation, solution, and especially philosophical papers tend to be at least several years away from what could be directly used in industry. This difference in focus could lead to academic work on Agile testing being inapplicable to practice or, worse, addressing different problems than those faced in industry. Based on this, two of the three studies presented in this thesis (Chapter Four and Chapter Six) were done in close collaboration with practitioners in order to ensure their needs were accurately being addressed.

5.1.2.4 What benefits do publications claim to offer?

This section focuses on the study performed in [55], in which publications on testing from the XP, Agile, and XP/Agile Universe conferences were investigated. The benefits of Agile testing were investigated by looking at the abstracts, titles, and keywords of papers to determine what the research discussed in these papers were supposed to help with. This investigation was actually quite difficult, at least through the lens of a systematic mapping study, given that most authors neglected to motivate their research. Out of the 110 papers included in the paperset for [55], only 28 benefits were described. The 7 keywords relating to benefits are broken down by

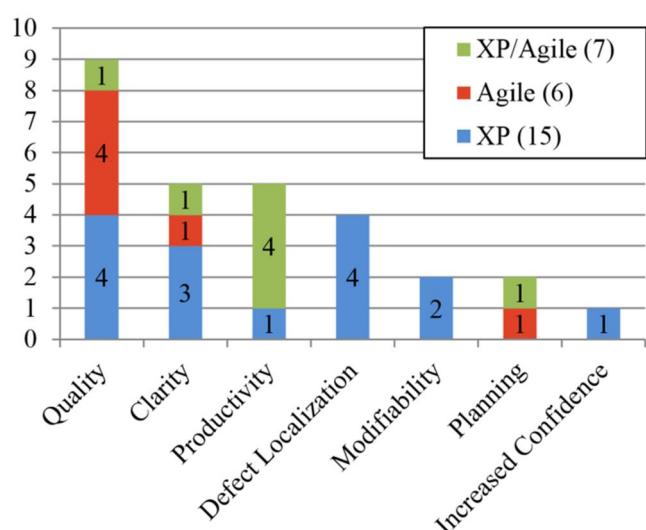


Figure 13: Breakdown of benefits offered by publications by conference.

publication venue in Figure 13.

Even given the low number of publications which actually give a reason in the abstract why anyone would want to use their research, the results here indicate that the benefits that these publications offer do not line up with the thinking of the participants in the interview study presented

in Chapter Four. In that study, only one participant indicated that they were using tests specifically for defect localization and one other participant indicated that automated testing increased his confidence levels. However, most of the participants indicated that they had issues with the maintainability and structuring of tests – a benefit that was only obliquely addressed in two of the publications investigated in this systematic mapping. This indicates that there is a need to perform further research in the direction of these topics, specifically in view of how to integrate them into an agile environment.

Based on these results, it would seem that the sort of research being published on testing in Agile development conferences would not fully address the needs participants expressed in Chapter Four. In the following section, I investigate whether authors assumed that practitioners were having the same sort of problems as those expressed by the participants in Chapter Four.

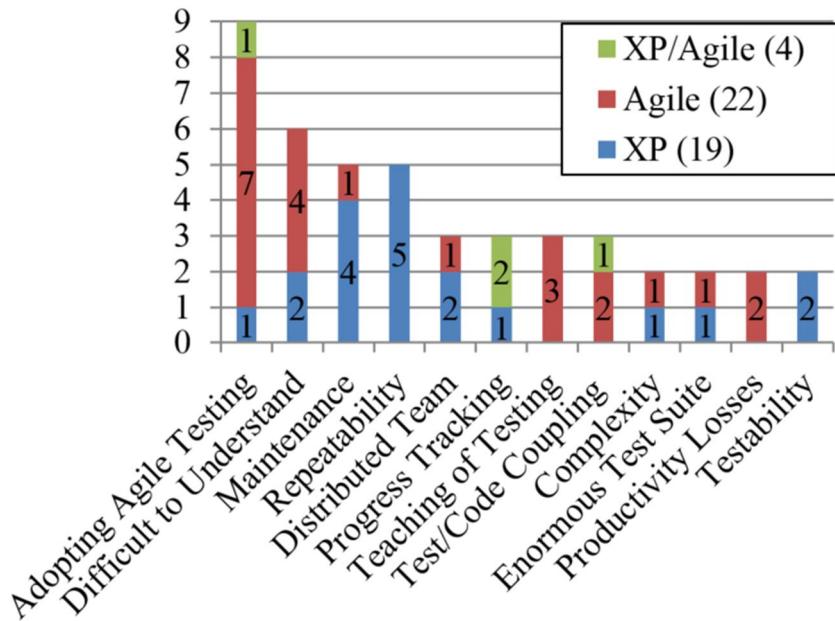


Figure 14: Breakdown of problems encountered with testing by conference.

5.1.2.5 What problems are associated with Agile testing?

In order to investigate the motivation behind research into agile testing, the problems identified by authors in the abstracts, titles, and keywords of papers were also investigated in [55]. One interesting result of this investigation was that not many publications actually identify what difficulties they are providing solutions for. In the 110 publications considered in this study, only 45 problems with testing were identified – even given that it was possible for a single paper to list more than one problem.

All of the keywords identified for this question are shown broken down by conference in Figure 14. From this figure, the most commonly-discussed problems vary by publication venue, with the Agile conference being strongly focused on issues of adoption, teaching, and understandability, and the XP conference being focused on the repeatability and maintenance of tests. In the study presented in Chapter Four, the issues brought up by the participants primarily

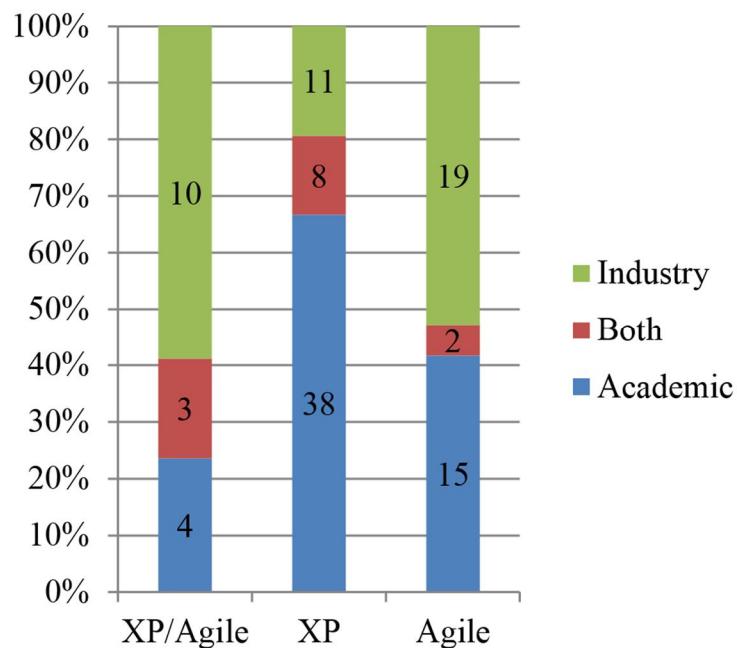


Figure 16: Distribution of papers at the conferences analyzed between industry, academic, and collaborative sources.

focused around issues relating to test maintenance – how to evolve test suites over time as the system changes and how to structure tests so as to reduce maintenance costs. It's interesting to note that, in this respect, the XP conference appears to be more relevant to the problems faced by Chapter Four's participants than the Agile or XP/Agile Universe conferences – even though XP had the lowest percentage of practitioner-generated papers, as shown in Figure 15.

5.2 GUI Testing

In order to investigate what topics existing academic publications in GUI testing have explored, another systematic mapping study was conducted. This study was much along the lines of the two studies described in the previous section. The research questions investigated in this study were:

- RQ1. What approaches to GUI testing have been researched?
- RQ2. What problems with GUI testing have been identified through this research?
- RQ3. What benefits do the publications claim to offer?
- RQ4. Which research methodologies have been used to study GUI testing?

These questions were additionally investigated in terms of:

1. How has research on these topics changed over time?
2. How do publications by practitioners, by academics, and by collaborations of the two differ?

As with the previous studies, a paper was only considered an industry-academia collaboration if at least one person from a university or research institution at time of publication and at least one author from a company at time of publication are listed. Government institutions, think-tanks, and research consortiums were evaluated on a case-by-case basis but were generally coded as “both.”

Because I wished to approach this topic from an academic perspective, I searched for peer-reviewed publications from both academic and industrial authors that deal with the general topic of GUI testing.

5.2.1 Methodology

5.2.1.1 Searching for Papers

After defining research questions, the next step in this mapping was to search through research databases for an initial set of papers. The research databases I chose to use were:

1. ACM Digital Library
2. IEEE Xplore
3. SciVerse Scopus

Since the ACM and IEEE host a large number of the conferences in this field, using their databases was a natural decision. Scopus was also used because it contains papers from a wider variety of sources than the other two databases. For example, at the time this study was run, the ACM Digital Library is composed of 1.8 million entries and IEEE Xplore is composed of 3.1 million entries while SciVerse Scopus is composed of 46 million entries.¹⁵

Two sets of search strings were defined. The first was used to search the titles and keywords of papers while the second was used to search the abstracts of papers. This was done because, during an early exploratory run of the search strings under consideration, it was found that searching for “user interface” and “test” separately in an abstract would generate too many irrelevant results, yet searching for “GUI test” in keywords or titles would generate only a few relevant results. The exact search strings used as well as the number of results for each search can be seen in Table 3. Note that the creation of the search strings used in this study are much more rigorously defined than in the previous studies. This was necessary because the field of GUI testing uses many synonymous terms, and so I attempted to include as many of them as possible in the search string used to find papers.

¹⁵ All numbers rounded to two significant figures; all numbers recorded as they were posted on the websites of each database on 30 Dec. 2011.

Based on these searches, 470 papers were identified; however, only 188 of these were unique by title and authors. This was due to the fact that a single paper could be indexed by multiple databases, but also because some authors tend to publish updates to previous publications under the same name as the original. An effort was made to reduce the number of updated versions of papers published by the same author set. For example, when posters were revised into workshop proposals, workshop proposals revised into full papers, full papers revised into journal articles – within the same year and with minimal changes to the abstract – the shorter publication was removed from this study. However, it is difficult to catch this sort of duplication based on titles and abstracts alone – especially for more popular topics – so there may still be some duplication in the set of papers that was used for this analysis. 9 papers were eliminated as duplicates during this process, leaving a set of 179.

Table 3. Preliminary, nonunique search results by target, search string, and database.

Search Target	Search String	Database	Results
Keyword	GUI AND Test	IEEE Xplore	3
Keyword	“User Interface” AND Test	IEEE Xplore	3
Title	GUI AND Test	IEEE Xplore	45
Title	“User Interface” AND Test	IEEE Xplore	9
Abstract	“GUI Test”	IEEE Xplore	28
Abstract	“User Interface Test”	IEEE Xplore	8
Keyword	GUI AND Test	ACM Digital Library	41
Keyword	“User Interface” AND Test	ACM Digital Library	12
Title	GUI AND Test	ACM Digital Library	12
Title	“User Interface” AND Test	ACM Digital Library	44
Abstract	“GUI Test”	ACM Digital Library	41
Abstract	“User Interface Test”	ACM Digital Library	5
Keyword	GUI AND Test	Scopus	50
Keyword	“User Interface” AND Test	Scopus	21
Title	GUI AND Test	Scopus	59
Title	“User Interface” AND Test	Scopus	17
Abstract	“GUI Test”	Scopus	61
Abstract	“User Interface Test”	Scopus	11

5.2.1.2 Screening Initial Papers

The third step was to do an initial reading of the titles and abstracts of each paper in order to weed out irrelevant papers. Results reporting on the results of workshops, results that did not seem to be discussing GUI testing, results outside the bounds of computer science and software engineering, and results for which an abstract was not available or was not available in English were not included. Finally, research focused on usability evaluation or web application testing exclusively was excluded, as these represent significantly different topics. Usability evaluation is more a design methodology than a testing methodology even though some papers refer to it as “usability testing”, and web testing is commonly done by bypassing the GUI entirely, so papers on these topics alone were excluded in this study.

The results of this screening process can be seen in Figure 16. Out of the initial 179 non-duplicate papers, a total of 62 papers were excluded as irrelevant. One additional paper was excluded because it was the sole paper in the set for 2012, which strongly implies that this survey was conducted too early for papers accepted for publication in 2012 to be available at the time

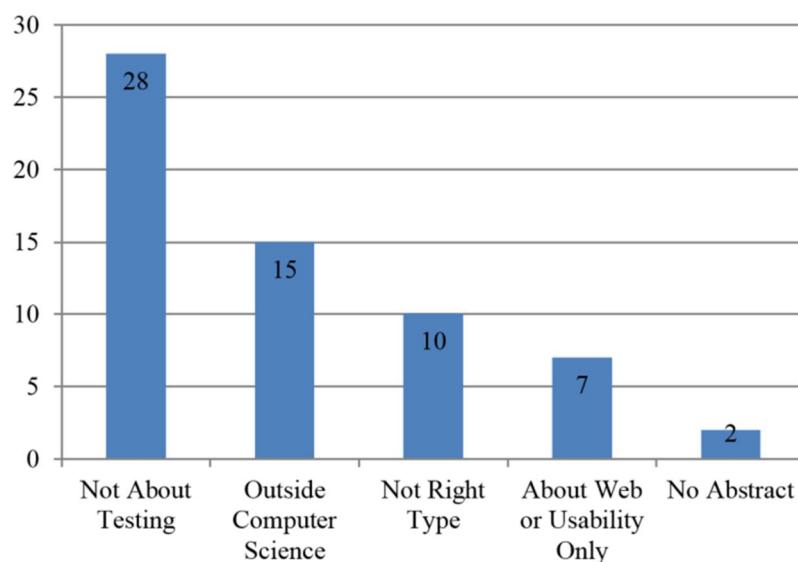


Figure 17: Number of papers excluded by reason for exclusion. (Note: “Not Right Type” refers to non-research papers included in conference publications, for example reports on the results of workshops).

the study was conducted. This left a set of 116 papers from 2011 and before for use in the evaluation.

5.2.1.3 Keywording using Abstracts

The fourth step in the analysis was to perform coding on each paper in order to develop a framework for investigating each research question could be answered. Coding was performed in this study by reading the abstract and title of each paper to identify keywords that are descriptive of the paper with respect to each research question.

To answer RQ1, RQ2, and RQ3, I used open coding. In this situation, whenever a new keyword was identified, I reconsidered previously-keyworded papers to ensure that the keywording was consistent across all papers.

To answer RQ4, I used closed coding to keyword the data. While there is not a paper defining the types of papers and evaluation types for software testing publications, these categories are well-defined in requirements engineering. While the subjects are different, the paper types should be analogous, so I used the paper types defined in [57] for closed coding for RQ4 – the same definitions as were used as the basis for this portion of the other two systematic mapping studies.

5.2.2 **Detailed Results**

5.2.2.1 What approaches to GUI testing have been researched?

To answer this research question, 47 different keywords were applied a total of 259 times to the 116 papers that were collected. Table 4 summarizes the results for keywords that appeared more than 10 times during this process.

The first thing to note is that “Agile” is not one of the top keywords – in fact, that keyword was only applied to three papers (two of which were my own publications). While GUI testing was one of the major keywords in the Agile testing mappings, from the perspective of

GUI testing research, Agile testing is not a major topic at all, which underscores the lack of work in this direction.

Model-Based Testing was the most explored topic with 47 occurrences. Model-based testing involves the creation of a representation of a GUI as a series of nodes representing states connected by edges that represent transitions. These models are used to automatically generate sequences of interactions and expectations about the model that are translated into GUI tests and run against the actual system under development. The types of models discussed in these publications included event-interaction sequences, event flow graphs, action sequences, state and behavior models, and hierarchical models. It can be advantageous to use model-based testing given that 1) GUI tests can be slow or expensive to create or run and using a model can be faster for this process; and 2) the interactions between GUI components can be very complicated and a

Table 4: Frequency of keywords used to describe approaches to GUI testing. The top 10 of these, bolded, are discussed in this chapter.

Frequency	Keywords
47	Model-Based Testing
40	Automatic Test Generation
19	Capture/Replay Tool
16	Fault Detection
15	Coverage Criteria
14	Test Suite Reduction
12	Test Suite Meta-Information
9	Manual Test Generation
8	Test Architecture
6	Automatic Test Repair, Genetic Algorithm
5	Planning, Prioritization, Test-Driven Development, Test Suite Improvement
4	Infeasible Test Cases
3	Ant Colony Optimization, Formal Languages
2	Assisted Test Generation, Assisted Test Repair, Automated Smoke Test, Ontology, Performance Testing, Random Test, Runtime Test Generation, Symbolic Execution
1	Agile, Assisted Test Execution, Complexity Estimation, Component Abstraction, Contract-Based Test, Education, Guessing Missing Test Data, Machine Learning, Manual Test Execution, Manual Test Repair, Multi-Agent System, Neural Network, Particle Swarm Optimization, Properties of GUI Bugs, Proprietary vs. Open-Source, Recommendations, Screenshot-Based, Task Model, Test-Code Co-Evolution, Use Case Model, User-Assisted Validation

model can be useful in understanding and simplifying these interactions.

Automatic Test Generation was the topic of 40 papers. This keyword was applied to papers where authors sought to create a system that could be given a GUI and, without human intervention, create a suite of test cases. Interest in Automatic Test Generation could be driven by the complexity of GUIs, the expense of creating tests manually, or the belief that GUIs are too complicated for human testers to comprehensively test such systems.

Capture/Replay Tools (CRTs) came in third at 19 occurrences. As mentioned previously, CRTs work by recording a human tester's interactions with a GUI in a format that can be replayed later as a regression test. These tools are especially useful in that they allow testers without programming skills to create automated tests. However, they can also be a liability in that creating a test is much easier than maintaining one. It's important to note that automated approaches were over twice as frequent in this paperset as CRTs, which require manual work in order to generate a test. It would seem that published research on GUI testing is aimed less towards human-centered processes like CRTs than it is towards automated approaches, which is dangerous given that no publication in this paperset discusses a fully-automated approached being used in an actual industry setting. While this topic is popular with academic researchers, they do not provide evidence that this practice is useful to or would be used by practitioners, whereas practitioner publications frequently mention using CRTs.

Fault Detection was the fourth most common keyword and appeared in the paperset 16 times. As it turns out, a significant part of the difficulty of GUI testing comes not from causing an error, but from noticing that an error has occurred. Papers dealing with Fault Detection are investigating methods not only for detecting potentially erroneous states, but also for deciding – automatically – if a given unexpected state constitutes a bug.

Coverage Criteria appeared 15 times in the paperset. Code coverage is a measure of how much of a system has been exercised by a given set of tests, and Coverage Criteria are the criteria used for this assessment. This is important for understanding how much of a system is tested, how thoroughly a given part of the system is tested, and which parts of a system are untested, all of which are important for estimating the reliability of the system. When tests run within the same process as the code they measure, it is simple to get this information. However, GUI tests are usually run from a separate process from the GUI with which they are interacting, which makes the acquisition of this data problematic and makes understanding the strength and usefulness of a suite of GUI tests difficult. Papers focusing on coverage criteria dealt not only with methods for collecting this data, but also with ways of making sense of it.

Test Suite Reduction addresses the issue that GUI tests can be slow and expensive to run and occurred 14 times in these publications. These papers suggested methods for finding out how much of a given test suite is redundant to reduce the amount of time wasted running tests that either test the same parts of the system as other tests or suggest methods for identifying tests that have no reasonable prospect of detecting bugs in the first place.

Test Suite Meta-Information came in seventh place after being applied 12 times. Meta-information – such as the length of a test case – can be used to predict the potential of a given test suite to trigger a bug. In general, longer test cases have stronger defect detection potential. Papers investigating this topic looked into questions like: can the probability of triggering a bug be increased by simply adding events to an existing test case?

The prevalence of these keywords as they occurred over time can also be mapped. Figure 17 shows the frequency of these keywords between 1995 (the first paper included in this set) and 2011. Interest in some topics is increasing and maintaining its gains – for example, interest in

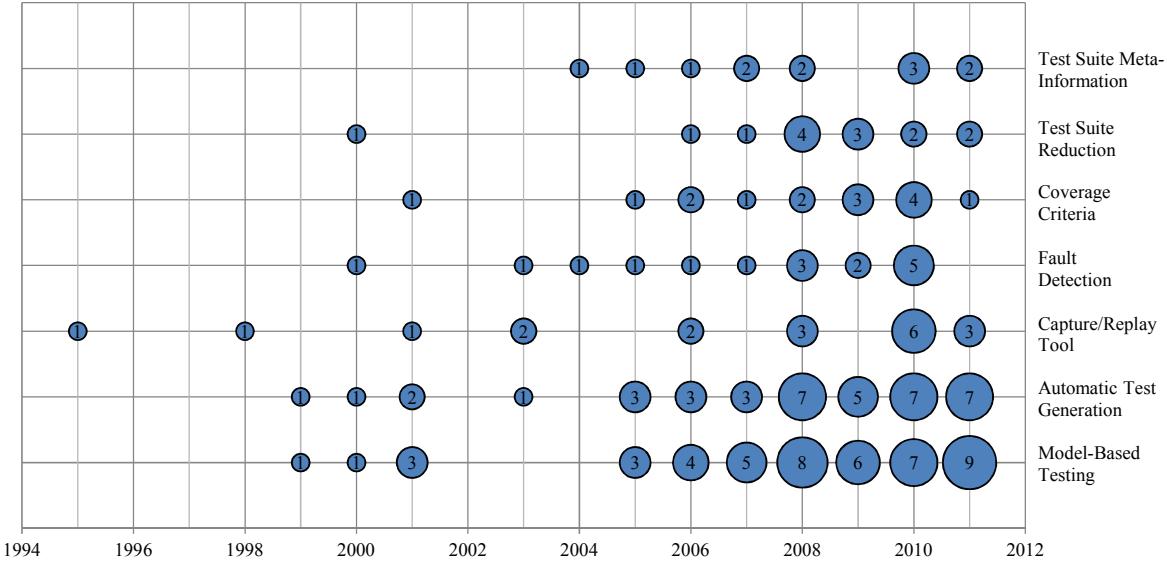


Figure 18: Approach to GUI testing by year of publication for each paper considered.

Model-Based Testing and Automatic Test Generation increased since their first publications – in 1999 – and, with a roughly three-year gap for each, largely continued to gain interest through to 2011. On the other hand, the number of publications discussing some approaches to GUI testing – such as Capture/Replay Tools – varies widely from year to year.

The number of papers on GUI testing published in a given year also varies wildly. Compare 2001, for example, in which more papers than any previous year were published, with 2002, in which not a single publication on GUI testing made it into the paperset used for this analysis (two papers were included in the set of initial papers, but these were both screened out for not actually being about GUI testing). This on its own is interesting given that the topics discussed in the field of GUI testing are relatively constant – publications dealing with CRTs today discuss much the same issues as they did two decades ago, for instance.

To determine if this insularity could be blamed, in part, on an “ivory tower” effect, I also investigated the source of each publication, and categorized papers as “academic”, “industry”, or “both” based on where the authors worked at the time of publication (as in the previous two

studies discussed in this chapter). Overall, out of the 116 papers identified in my search, the overwhelming majority – 96, or nearly 83% – were produced by entirely by academic authors. Only 11 papers – ~9.5% – were produced by industry practitioners alone, while 9 – ~7.5% – were produced by combinations of academics and practitioners or from authors with affiliations with institutions connected to both industry and academia.

The top 7 keywords from this section were also investigated with respect to their source. Figure 18 shows the absolute number of keywords by source, respectively. This chart clearly shows both that academics dominate the field and, on their own, industry authors will focus on different topics from academic authors. Practitioners focused primarily on CRTs, with a secondary emphasis on automatic test generation. There were no practitioner publications on any of the other top keywords. Academics, on the other hand, focused strongly on model-based testing and, while there were a significant number of academic publications on CRTs, there were fewer than half the number of publications on that topic than on model-based testing or

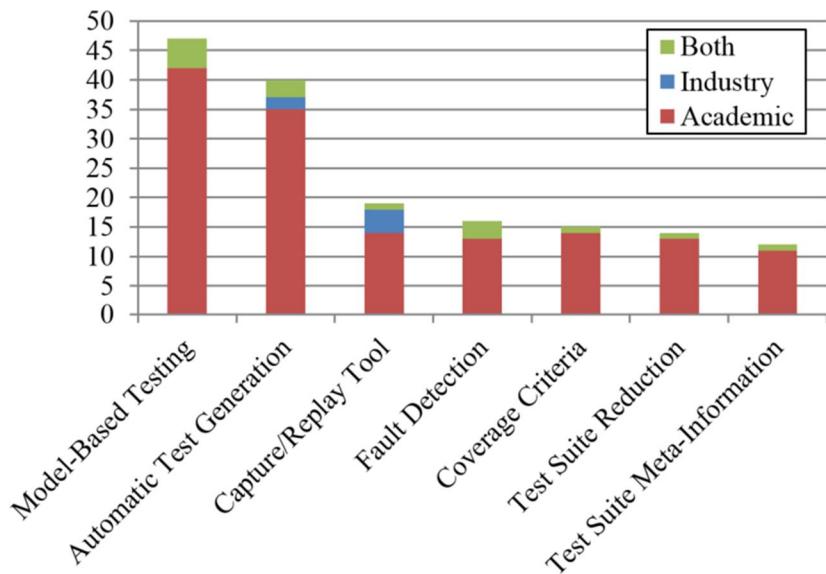


Figure 19: Absolute number of keywords by source.

automatic test generation. This could be taken to indicate a mismatch between the sort of topics practitioners are interested in, or the sort of techniques that make sense for them to utilize, and the sort of topics academics are pursuing.

5.2.2.2 What problems with GUI testing have been identified through this research?

As with the previous section, open coding was used to find keywords in and apply them to papers in order to understand what issues authors use as motivation for their research into GUI testing. For this research question, 26 keywords were applied a total of 218 times to the 116 papers. Table 5 shows each keyword that appeared 10 or more times. This amounts to around 76% of all keywords identified during this stage of the investigation.

Occurring 59 times – more than once per four papers – the “Complexity” keyword was applied to papers in which the overwhelming complexity of GUIs was listed as a motivation for conducting a study. This keyword was applied whenever the authors pointed to aspects of GUI-based applications that lead to a very large number of possible states – such as comments about the number of possible events, different possible inputs, interactions between events, and general

Table 5. Frequency of keywords used to describe problems encountered during GUI testing. Those in bold are discussed in this chapter.

Frequency	Keywords
59	Complexity
20	Lacking Techniques
19	Test Generation is Difficult
18	Maintenance
17	NONE
11	Cost, Many Test Cases
10	Resource-Intensive
9	Manual
8	Defect Detection
7	Lacking Tool Support
6	Technical Issues
3	Missing Information, Organizational Issues, Time-Consuming
2	Difficult to Use Tools, Element Identification, Environmental Side-Effects, Source Code Unavailable, Test-First Development
1	False Positives, Multi-Language GUIs, Non-Functional Requirements, User Involvement

complexity in GUIs. Based on this widespread concern with Complexity, the emphasis on Model-Based Testing and Automated Test Generation in the previous section make sense. Given the vast number of possible actions that can be taken on a GUI and the number of ways in which these actions can affect what further actions are possible, systems for modeling and understanding a GUI and methods for automatically creating tests are very important. In short, from these papers, GUIs are too complex to be reasonably understood by single humans, so the natural next step would be to find automated ways of understanding and testing GUIs. One possible reason for the complexity of modern GUI- and Web-based systems is that popular APIs encourage it. As Janet Gregory noted in a presentation [13], the complexity encouraged by languages like JavaScript is in direct opposition to the recommendations for the use of software design patterns like Model-View-Controller for simplifying GUI design and testing – recommendations dating back to 1979 [60].

Authors remarked in 20 publications that traditional testing techniques are inapplicable to or inadequate for GUI testing – for example, because the cross-process nature of GUI testing makes it difficult to apply approaches designed for unit testing, wherein it's possible to directly call methods. The “Lacking Techniques” keyword was applied to these instances. Authors noted that advice on how to test specific types of GUIs, information on how to understand test coverage, or best practices for GUI test design simply didn't exist. This is quite remarkable given that GUIs have existed since at least the 1960s and have been widespread since the early 1980s. A potential explanation for this is the fact that modern GUI toolkits seem to be developed for functionality first and testability second; as such, testing methodologies are always playing catch-up [13].

The third most commonly-applied keyword was “Test Generation is Difficult”. This keyword was applied – 19 times – whenever authors addressed the problem of generating automated GUI tests, selecting good GUI tests, or the specialized skills required of testers to manually create GUI tests. These are distinct from the other problems raised in this section in that these papers focus on the difficulty of applying existing techniques and seek to make this process easier. As with the Complexity keyword, the prominence of this keyword makes sense given the focus on Automated Test Generation and Capture/Replay Tools as keywords in the previous section.

The “Maintenance” keyword was also applied quite frequently – 18 times in the 116 papers of the paper set. GUI tests are especially likely to need maintenance whenever changes are made that alter the events that are triggerable from a given state of the GUI. Additionally, a change to the way a single event works can make maintenance required in a large number of test cases, which can be a tedious, time-intensive process. It is interesting that this topic was listed as a problem 18 times, yet only 9 papers proposed solutions relating to the Automated, Assisted, or Manual Test Repair keywords identified above. This would suggest that further research into the impact of and potential solutions to maintenance-related issues should be encouraged in the future.

One difficulty encountered in performing the investigation in this section was that problems with GUI testing were often not specified or were not specified in enough detail to be useful. In fact, this happened often enough to justify creating a separate keyword for these papers – and this “NONE” keyword, at 17 occurrences, is the 5th most common. I would argue that this may make it less likely for practitioners – people in search of solutions to specific problems – to actually read these papers. This would mean that these papers would be less likely to have an

impact on practitioners, and could serve to deepen the disconnect between practitioners and academics in the field of GUI testing.

“Cost” was the sixth most common keyword at 11 occurrences. This keyword was applied to cases in which authors were concerned about the human cost of GUI testing in terms of time taken to create tests, time taken to maintain tests, training and experience required to create good GUI testers, and cost of employing competent testers to perform GUI testing. Given that the other problems with GUI testing would be negligible if it was not expensive to perform GUI testing, it’s surprising that this issue was not more common. The frequency of this keyword still underscores the impact of maintenance of GUI tests on software development efforts.

“Many Test Cases” tied with Cost for sixth place. This keyword was used to denote papers in which the authors noted that the large number of test cases that were likely to be necessary to test a GUI are a central part of the difficulty with GUI testing. Because of the large number of unique states that even a simple GUI can enter, it’s common for GUI test suites to be quite large. This contributes significantly to other difficulties – especially test maintenance. Given that a single change to a GUI can break many tests, these two factors together pose a significant challenge. Given this, it’s not surprising that Coverage Criteria – which relates to the ability to tell if a given set of tests is adequate – and Test Suite Reduction were the fifth and sixth most common benefit keywords in the previous section. It’s worth noting that a common mistake made by new GUI testers is to create an unmaintainably large suite of GUI tests early on. Based on this, research into how to teach GUI testing would be a profitable direction for future.

“Resource-Intensive” was the eighth most common keyword at 10 occurrences. This keyword applied to papers which noted that GUI tests with a good chance of catching bugs required a lot of space to store and a lot of time to run. It is interesting to note that this has been

listed as a problem as recently as last year even though one would have expected the rise of parallelization to have dealt with this issue. In fact, parallelization of GUI tests is advisable for a variety of reasons – including the fact that Environmental Side-Effects (the sixteenth most common keyword in this section), such as those caused by other GUI tests running on the same system, can cause errors which are very difficult to debug.

Figure 19 presents a bubble chart of the frequency of each problem over time from which we can see trends in the problems addressed by publications in this paperset.

The first interesting trend we can see from this set is that Complexity tends to spike in importance – some years, nearly all publications will discuss complexity (for example, 2000), which in other years Complexity will not be discussed at all. After dominating the field between 2008 and 2010, papers addressing Complexity dropped very significantly in 2011. Also, despite having a strong showing in 2000 and 2001, Complexity was not the subject of publications for the following three years.

Next, it is disturbing to note that, in 2010 and 2011, papers with the NONE keyword rose sharply – to the point where it is the second most common keyword for both of those years. This

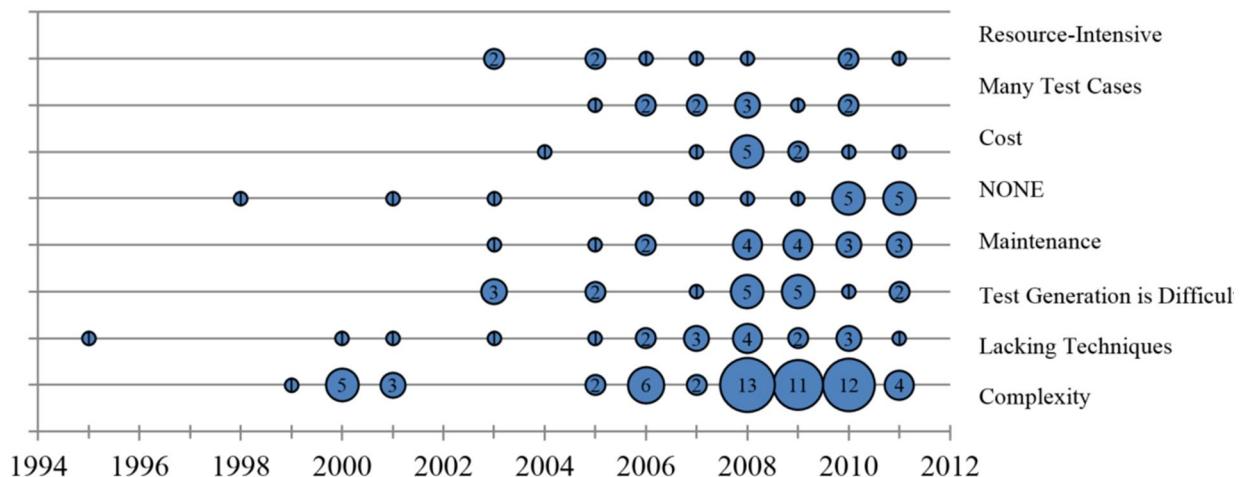


Figure 20: Frequency of problems encountered in papers by year of publication.

means that, the abstracts of these papers do not identify the research problem to be addressed in them. This should be viewed as a decrease in the quality of these abstracts, and it is troubling that abstracts that do not have a stated research problem are on the rise in this specific field.

Maintenance seems to have become a significant issue starting in 2008. The difficulty of GUI test maintenance is related to the number of tests there are to maintain and the way in which these tests are structured, which could additionally serve as an explanation as to why this topic only emerged recently. In order to deal with this issue, future work needs to investigate techniques for increasing the maintainability of GUI tests suites – for example, structuring GUI tests such that they are easier to maintain.

In order to investigate differences between the problems encountered by academics, practitioners, and collaborations between the two, Figure 20 was created. Based on Figure 20, it would at least seem that academic and practitioner understanding of the problems with GUI testing line up better than the benefits explored in the previous section. Of all the top keywords, only Resource-Intensive was not investigated by a single industry publication. This isn't

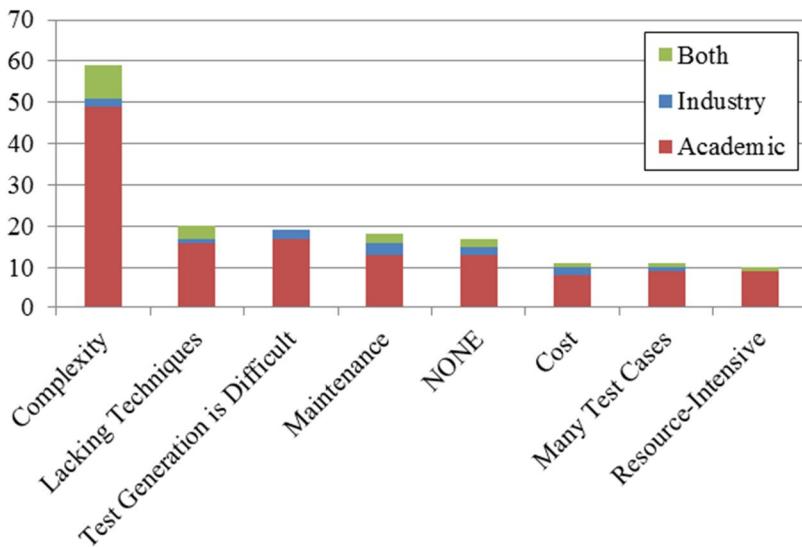


Figure 21: Absolute number of keywords for each paper source by problem.

surprising given that hardware costs tend to be the cheapest issue that most companies experience – it tends to be comparatively cheap to just allocate additional resources to a problem if that will provide an adequate solution than it would be to pay developers and testers to implement a more efficient solution. Overall, though, it would appear that academics and practitioners at have both at least published once on most of the major issues with GUI testing.

5.2.2.3 What benefits do the publications claim to offer?

Open coding was used to identify keywords relating to the advantages of using a given approach to GUI testing as explained in the abstracts of publications. These benefit keywords were restricted to benefits of the method proposed in the abstract – not benefits of GUI testing in general. This is an important distinction, as many abstracts begin with a description of the general benefits of testing or the assumed benefits of GUI testing. In the process of coding papers to answer this question, 14 keywords were applied a total of 175 times to the 116 papers in the paperset. Table 6 lists each of these along with their frequency. The top 8 of these keywords are discussed in detail in this section.

The most frequently-occurring benefit of specific approaches to GUI testing was that the method was found to “Increase Test Effectiveness”. This keyword was applied any time a tool or

Table 6. Frequency of keywords relating to benefits of tools/techniques. Bolded: major keywords discussed in this section.

Frequency	Keyword
32	Increase Test Effectiveness
28	None
23	Easier
20	Guidelines
14	Cost, Fewer Tests
12	Adapts to Change
11	Indicate Test Quality
6	Early Feedback
5	Test-First
3	Frequent Feedback, Understand Use Cases
2	Reusable, Stability

technique was found to increase the likelihood that a given test would detect a defect, increase the code coverage of a test suite, or otherwise improve the usefulness of tests. This emphasis is interesting given that the Defect Detection keyword used in the previous section was only the tenth most commonly-occurring problem listed in abstracts with 8 occurrences (3.7% of total). Increase Test Effectiveness, on the other hand, occurs 32 times – 18% of the all benefit keywords. This is another indication of a mismatch between research problems and research goals. A large number of papers promise benefits in terms of effectiveness, but this is only indicated as an issue in a small number of abstracts.

Again, the “None” keyword was second most common; at 28 occurrences, it accounts for 16% of the keywords applied. This keyword does not necessarily denote research with no benefits, but rather research with no stated benefits in the abstract besides, in certain cases, a statement that a given methodology worked or “is practical.”

13% of papers stated that their research would make GUI testing easier, leading to the creation of the “Easier” keyword. These papers found that their results would make it easier to understand or create tests, increase developer productivity, or make it easier to understand the GUI under test. This relates to the Complexity, Test Generation is Difficult, and Maintenance major problem keywords from the previous section as well as several of the less-common keywords (“Manual”, “Time-Consuming”, “Difficult to Use Tools”, etc.). In all, the Easier keyword accounts for only 13% of keywords while around 50% of problem keywords would need to be addressed with this benefit, if one is expecting research to provide solutions to the problems it identifies. It’s worth noting that approaches to GUI testing like Model-Based Testing, Automated Test Generation, and other techniques that could be used to make GUI

testing easier account for about 46% of all approaches, and yet only 13% of papers report that their methods did in fact make GUI testing easier.

The “Guidelines” keyword was applied to papers that explicitly stated that their research had resulted in the creation of a set of steps or in specific advice for performing GUI testing that would avoid difficulties encountered by the authors. This keyword occurred 20 times, which makes up about 11% of all benefit keywords. This keyword relates strongly to the Lacking Techniques keyword from the problems section, above. Lacking Techniques occurred 20 times in that section, accounting for about 9% of the keywords in that section. Based on these numbers, this is an instance where a problem and solution are well-matched in the research.

The “Cost” keyword was applied in this section to papers proposing ways to reduce the time required to run a suite of GUI tests, the space required to store these tests, or other requirements for running GUI tests. Cost was applied to 14 papers, or 8% of all benefits listed. This benefit most strongly relates to the Resource-Intensive keyword from the previous section – which, at 10 occurrences, accounted for about 4.5% of problems listed. Abstracts promising a reduction in the number of tests or the size of individual tests were tagged with the Fewer Tests keyword, which tied with Cost for 5th place. This benefit most strongly relates to the Many Test Cases problem from the previous section, which, at 11 occurrences, made up 5% of all keywords. While the percentages differ, the absolute numbers of keywords for both terms show that there is a reasonable match between problems identified in the abstracts of the paperset and the benefits of research with respect to this issue.

12 papers suggested that they were able to increase the extent to which a test “Adapts to Change”, so this keyword related to 7% of all benefit keywords. The Maintenance problem specifically relates most strongly to this keyword, and the percentage of research devoted to this

topic (8%) lines up well in this case. However, based on the results from Chapter Four, this problem either hasn't been sufficiently well-addressed or the results published in academic venues are not making the transition from academic research to use in industry.

The last keyword occurring more than 10 times was applied to research designed to "Indicate Test Quality". This keyword was applied 11 times, accounting for 6% of keywords in this section of the analysis. Methods for indicating test quality that were identified in the paperset included creating code coverage tools and determining the characteristics of GUI tests with a strong probability of detecting bugs or being highly maintainable. While the quality of tests was not directly identified as a problem in the previous section, this benefit does somewhat relate to several problems, including Test Generation is Difficult and Defect Detection.

So far, we have discussed how the relative occurrences of benefits in the paperset relate to the relative occurrences of problems identified in the previous section. What remains to be shown is the way in which these problems vary over time. To this end, Figure 21 shows a bubble chart of the frequency of the keywords identified in this section against the year in which those keywords appeared.

The first trend of note is that papers listing no specific benefits have increased over time.

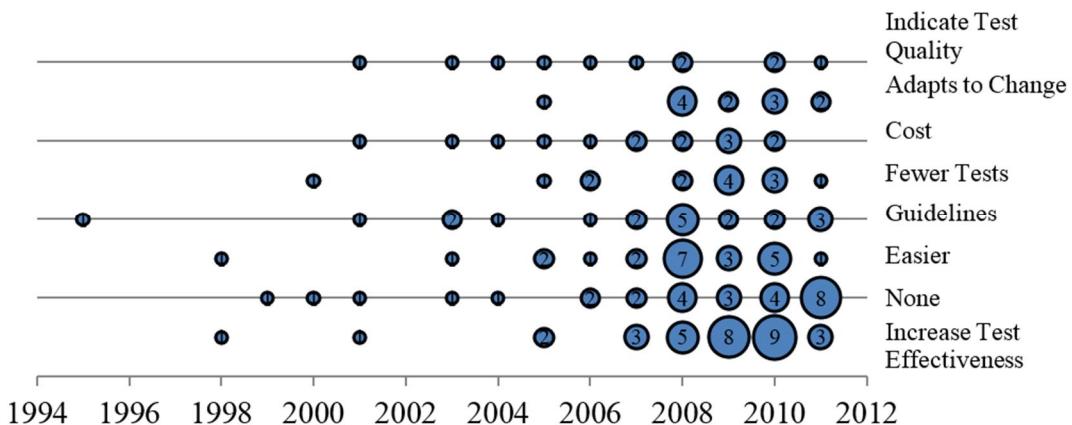


Figure 22: Frequency of benefit keywords in papers by year of publication.

Alarmingly, as many papers were tagged with the None keyword in 2011 as the three next most important keywords for that year combined. From this study, it is not possible to make judgements about the quality of papers themselves, only their abstracts; however, future work should determine if this is the case and, if so, conference organizers will need to find ways of encouraging higher standards for publication.

Another interesting trend is the significantly reduced frequency of the Increase Test Effectiveness and Easier keywords and, to a lesser extent, the Fewer Tests and Cost keywords. In fact, none of the major keywords discussed in this section with the exception of None actually increased in frequency for 2011. It's worth noting that this decrease isn't due to a decrease in the raw number of papers published that year, since 19 papers from that year made it into the paperset compared to 23 from 2010.

It's interesting to note that Adapts to Change didn't become important until 2008 – very recently given the size of the timeslice captured by this study. This is the same year that Maintenance rose to prominence as a problem keyword. From this correlation, it looks as if maintenance wasn't seen as a problem until papers identifying it as a problem and providing a solution began to be published in considerable numbers in 2008. This is counter-intuitive given that most manual or assisted test generation methods – common in industry – tend to create large test suites which immediately become a maintenance nightmare given their affinity for breaking when the GUI is changed. This problem was even more acute with early GUI testing tools and methodologies. It makes little sense, then, for this topic to have only become important recently. It's possible that issues with maintaining a suite of GUI tests were accepted as an unavoidable part of GUI testing, but, if this is the case, it's troubling that these issues weren't mentioned earlier on in the history of the academic field. It would seem that researchers in the field of GUI

testing need to produce outlines for future research – overviews of existing problems that need to be solved in the future – in order to get this sort of research in motion earlier.

This sort of issue again raises the question: are the benefits offered in practitioner research the same as those offered by academic or collaborative work? In order to answer this question, Figure 22 was created to visualize the benefits proposed by different groups of authors.

There are gaps between benefits proposed by these groups. Only Academic papers have proposed benefits for Indicate Test Quality – neither practitioners nor collaborations have found ways of improving this. Again, since this benefit wasn't indicated by a specific problem in the previous section, it's hard to know what has been made of these results in practice.

There were two other keywords from which one group was missing: Fewer Tests and Cost. No Industry papers addressed Fewer Tests, which is a significant issue (as raised in Chapter Four). Having a smaller test suite could drastically reduce the effort required to maintain the test suites used by many companies. Further, no joint studies showing benefits in terms of cost exist.

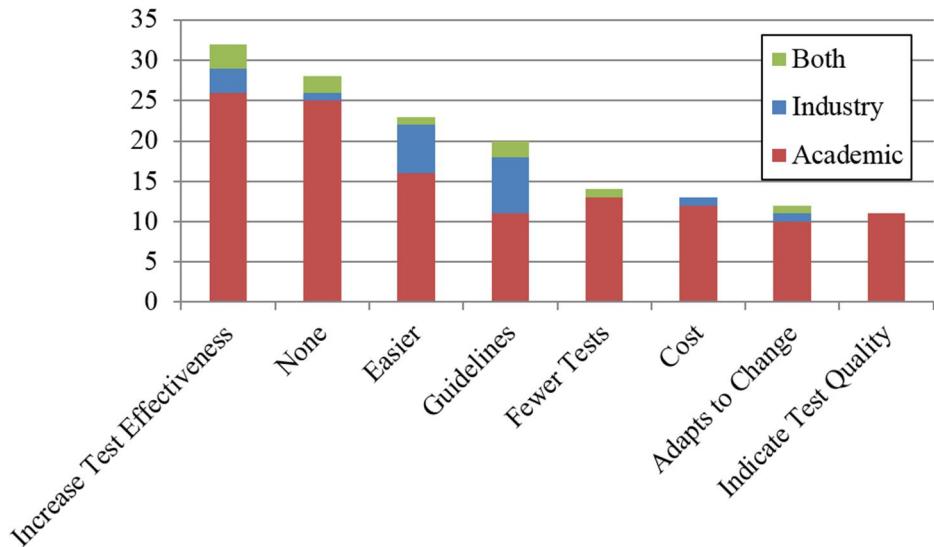


Figure 23: Absolute representation of benefits offered by publications grouped by source.

5.2.2.4 Which research methodologies have been used to study GUI testing?

In order to answer this research question, I used closed coding. As in the studies described in the previous section, the paper types described in Weiginra et al. [57] were used as the descriptions of possible paper types. In order to gain a better understanding of the research methodologies used in the field of GUI testing, Figure 23 was created.

In this bubble chart, we can see that by far the most common research methodologies used are Solution proposals and Validation studies. In fact, out of the 127 keywords applied to papers to describe research methodologies used, 80% fell into these categories (54 Validation and 48 Solution). The number of both types of papers has increased dramatically starting in 2008, though there is more variation in the number of Solution papers appearing each year. This strongly indicates that GUI testing is not yet a mature field, as many novel techniques are being tentatively explored each year.

Another indication of the immaturity of the field is the low number of Evaluation and Philosophical papers. Evaluation studies show how techniques or tools perform in practice, so

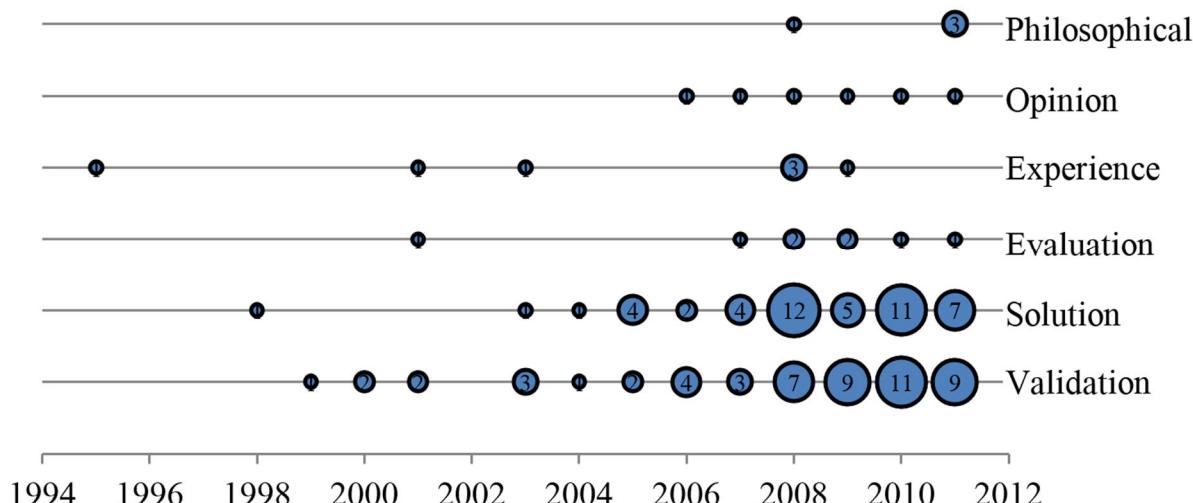


Figure 24: Bubble chart showing frequency of research methodologies used per year.

they require the buy-in of an industry partner willing to take a risk by using a technique that has not yet proven itself in practice. This may also explain why, with one exception, Evaluation studies only became a regular part of the field in 2007. Additionally, it's not surprising that only one Evaluation study has been done by an Industry-only group of authors, since the amount of scientific rigor required for this sort of research is not common outside of academia. Despite these difficulties, it's troubling that so few of these studies have been published because, until more Evaluations are done, it will remain difficult to tell if the research being done in this field has a reasonable chance of making a difference in practice.

Similarly, only four Philosophical papers were identified in this paperset. Philosophical papers are specifically important to research done by academics in that they help to structure a field to better inform future research. Because Philosophical research is usually done by looking back at previous publications, though, it's difficult to perform this sort of study on an immature field. Interestingly, out of the 4 Philosophical studies included in this paperset: the first provides a classification of common issues encountered in GUI testing; the next provided a classification of GUI bugs; the third provides a classification of general testing activities; and the last provides a classification of GUI validation techniques. It's interesting to note that, from these studies, there are as many publications dealing with the task of understanding techniques for GUI testing as there are publications seeking to understand what exactly the difficulties involved with GUI testing are. It's worth noting that it will be exceedingly difficult to research general-purpose GUI testing tools and techniques until we have a good understanding of what the problems we are intending to solve actually are, so it would be worthwhile to encourage additional research in this direction in the future.

Experience and Opinion papers are useful in that they point out directions towards which future research should focus. While a single Opinion paper has been published every year since 2006, Experience reports only appear sporadically. The distinction between Experience reports and Evaluation research is that Experience reports tend to be descriptions of an attempt to use a tool or technique in practice rather than to report on a rigorous experiment. Because of this, practitioners are more likely to write Experience reports. Given that only a few of these papers made it into the paperset used in this paper, it's likely that these reports are being published in other manners – perhaps through blogs, social networks, and practitioner-oriented books. Given that peer-reviewed publications are generally discouraged from including non-peer-reviewed publications like these in their “Related Work” sections, this again raises the possibility of a disconnect between the issues being researched in peer-reviewed literature and the issues encountered in industry.

Given that we have already seen issues with the research methodologies that have been published by year, it's natural to ask if there are also issues in the distribution of research methods between authors from industry, academic, or collaborations of the two. Figure 24 presents a breakdown of absolute number of keywords colored by source.

There are two main points of interest in these graphs. First, 5 studies from Both (out of 10 total) were Validation studies. This is odd considering that collaborations of practitioners and academics should gravitate towards Evaluation studies. In these circumstances, it should be possible to have academic rigor within a practitioner environment – the definition of an Evaluation paper. The fact that these papers were classified as Validation means that a study took place, but implies that the research was not sufficiently rigorous to count as an Evaluation. Future collaborations need to ensure that these opportunities are used to create solid, rigorous

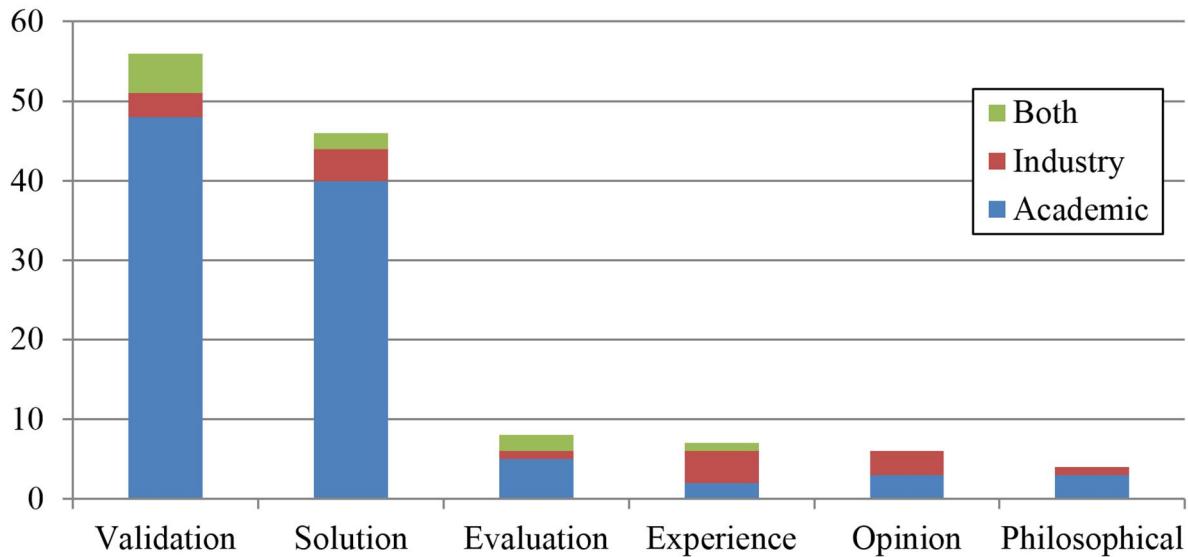


Figure 25: Absolute numbers of research methodologies used by source.

Evaluation studies in order to better extend our knowledge of how techniques and tools work in practice.

Second, no papers have been published by collaborations between practitioners and academics that present Opinion or Philosophical research. This is dangerous in that it is much more likely for either academics or practitioners to present well-considered opinions or frameworks for understanding the field of GUI testing without considering taking advantage of insights from the other side of the desk – insights that may end up being the difference between proposing something reasonable and something untenable.

5.3 Conclusions

Through mapping peer-reviewed publications on Agile testing and on GUI testing, several important points have become clear. The first is that the research questions involving understanding what GUI testing and Agile testing, as two mostly independent fields, are focused on was answered. In Agile testing, heavy emphasis was placed on TDD. GUI testing is not something that has been well-explored within the domain of testing in Agile development

contexts, although several publications in that space do exist. GUI testing, on the other hand, is focused around model-based testing and automatic test generation – topics that do not occur at all within the set of papers on testing in Agile development environments. Additionally, only a single paper within the GUI testing paperset discusses Agile development methodologies. From these results, I have demonstrated that GUI testing and Agile testing are not already integrated and that further work would need to be done in order for this to be possible. Additionally, the results indicate that TDD is a large focus of research on testing in Agile development environments and is not a focus of GUI testing research. Based on this, enabling TDD of GUI-based applications is a promising avenue for further research, and is pursued further in the following chapter.

Chapter Six: Test-Driven Development¹⁶

TDD encourages communication between customers and developers, increases programmer confidence, increases software quality, and decreases bug density without decreasing productivity [61], [62]. Recent work has shown that the effects of TDD are overwhelmingly positive as reported in rigorous, industry-relevant research papers [63] and because of these benefits, TDD has always been a core part of Agile. However, published approaches to TDD test the software layer just underneath the GUI rather than testing the software through its GUI – as a user would access it. This does not meet the definition of GUI testing as used in this thesis.

In addition, it's not possible to make use of CRTs for test driven development of user interfaces because this usually requires an existing GUI in order to create a test (see Chapter Three). In this chapter, I explain a novel approach to TDD of GUI-based applications inclusive of the GUI itself (termed User Interface Test-Driven Development, or UITDD) that is suitable for use in Agile development environments where the team is already using prototyping techniques from the domain of User eXperience design (UX).

The difficulty of creating a suite of GUI tests before the GUI exists – as well as maintaining the suite after development of the GUI has started – is a major barrier to the uptake of UITDD (see Chapter Three). For the most part, developers wishing to perform test-driven GUI development must write complex test code manually [46] [45], or write system tests just below the level of the GUI, though two methods similar to the one described in this section have been described since the original publication of the papers on which this section is based in academic (but not practitioner) publications [24] [22]. UITDD is especially difficult given the tight coupling between GUI tests and GUI code. In order to test a widget, automated tests need to

¹⁶ This chapter is based on work published in [47] and [48].

be able to locate it reliably at runtime and interact with it reasonably in order to create a useful test. To do this, a test must make use of specific information about each widget involved in a test – such as the widget’s position, type, name, or identifier. Unfortunately, this makes it difficult to write a GUI test before the GUI itself, as a good deal of information about the implementation of the GUI may be necessary in order to write a reliable GUI test.

In order to simplify the process of UITDD, I developed an approach that leverages usability evaluation and user interface prototyping. By first creating a digital prototype of the GUI, using tools like Expression Blend¹⁷ or ActiveStory Enhanced, it is possible to perform iterative usability evaluations to identify and fix flaws in the GUI early in the development process, before effort has been expended in development of the actual GUI. Then, the prototype can be augmented with additional information about the expected behavior of the GUI. This allows for complex acceptance tests to be recorded using a CRT – overcoming one of the main difficulties with UITDD identified in Chapter 3.4. CRTs can then be used to record test scripts. These scripts can then be run against the actual GUI as it is developed.

In short, using a sufficiently detailed prototype for Agile interaction design will garner two benefits. First, usability concerns will be discovered early in development, meaning the final GUI will be less likely to require changes. Second, if the prototype is decorated with automation information – information that can be used to identify and make assertions about widgets – and if this information carries over to the actual implementation, then tests can be recorded from the prototype and replayed on the actual GUI as it is implemented.

¹⁷ <http://www.microsoft.com/expression>

6.1 Solution Description

6.1.1 Tools

The tool used for usability evaluation in the proof-of-concept investigation of the approach to UITDD described in this chapter, ActiveStory Enhanced, was developed mainly by another member of the Agile Software Engineering Lab at the University of Calgary [64] [65]. ActiveStory Enhanced allows usability engineers to create low-fidelity prototypes that study participants can interact with as though they were working applications. These prototypes are arranged into a “storyboard” – an ordered set of pictures of different possible states of an application with labeled transitions between states. These transitions represent interactions, usually triggered by the application’s user, that will cause it to transition between different states. In ActiveStory Enhanced, transitions are implemented using “hot zones” – clear transparent buttons covering specific regions of a prototype. This means that a usability evaluation can be performed on an ActiveStory Enhanced storyboard by having a participant interact with the system as though it were functional – each button click causing a transition to a new state such that it looks like the application is actually working. Unlike a working application, creating a storyboard using a set of quick sketches of various states of an application is a cheap, quick process and can be easily altered for iterative usability testing. Low-fidelity prototypes are specifically interesting in that they can be distributed online and used for usability testing with a large number of users over the internet in a cheap, distributed fashion.

LEET (a recursive acronym for LEET Enhances Exploratory Testing) is a CRT that provides rule-based AI support for automated GUI testing. LEET’s rule engine works by allowing a human tester to create rules defining things the system should or should not be able to do, runs these rules whenever specified preconditions are met, and then restarts the test run, thus increasing code coverage [19]. For example, it’s possible to create a rule in LEET with a

precondition specifying “a hyperlink titled ‘Logout’ is visible” and a rule stating “when ‘Logout’ is clicked, the text ‘You have been logged out of the system’ should be displayed onscreen”. This rule would then be run at every state where a user is logged in and ensure that the logout functionality works as specified – of course, restarting the test after every time the rule is executed and preventing the same rule from executing more than once from a given state. I developed LEET around the Windows Automation API, which allows for easy keyword-based testing of user interfaces. The ability to use keywords to identify widgets of interest in a test script is crucial for my approach to UITDD. Since the only property of a widget that is used to identify it uniquely in the approach LEET uses to keyword-based testing is its AutomationID, it is possible to record a test from an ActiveStory Enhanced prototype that has had AutomationIDs added to the widgets that represent hot zones. As long as the actual GUI for the system uses the same keywords to identify the widgets represented by hot zones, a test recorded against the ActiveStory Enhanced prototype can be replayed against the actual GUI.

6.1.2 Solution Concept

First, user stories are used to develop a low-fidelity prototype of the GUI using ActiveStory Enhanced. Iterative usability evaluations should be performed on these prototypes in order to decrease the likelihood that changes will need to be made to the final GUI, since they’ll be caught before implementation is actually done. Since low-fidelity prototypes can be created quickly at little cost, they are ideal for making iterative usability evaluations part of the Agile development process.

Once the prototype has become sufficiently stable through usability evaluation, it can be decorated with additional automation information to allow complex verifications to be made. Using LEET, a set of acceptance tests can be recorded from interactions with the decorated prototype. These tests can then be run on the GUI-based application as development progresses.

The first benefit of this approach is that UITDD can be performed without additional limitations. The simplest tools for creating GUI tests, CRTs, can be used, meaning tests do not have to be written by hand, as is the case with existing tools used for UITDD with the exception of [24]. However, unlike [24], the approach described in this chapter does not suffer from the issues related to mis-identifying widgets in different applications as the ones needed to run a test due to their visual similarity (as described in Chapter Three).

Second, it is expected that test maintenance costs will be lower due to the usability testing that is performed prior to implementation. While this will require more design work up front than is normally expected in an Agile process, but this issue is also present in the field of Agile UX, where most publications acknowledge that some advance design work will be necessary up-front [66].

Finally, while tools exist to facilitate repair of broken tests (see Chapter 3.2), the best solution would be to decrease the instances of tests breaking in the first place.

6.1.3 Demonstration¹⁸

For an example of UITDD, let us consider the design of a calculator like that provided with Windows. It will contain keys representing numbers, keys representing operators, and a display at the top that shows either the number being entered or the result of the previous operation. For now, we'll consider the addition feature only. In this story, the five button, plus button, nine button, and equals button are clicked in that order, and we expect that the display should read “14” at the end.

A storyboard of this test sequence is shown in Figure 25 on a prototype created in ActiveStory Enhanced. For this test, “5” will be clicked, then “+,” then “9,” then “=,” and for

¹⁸ From work published in [47].

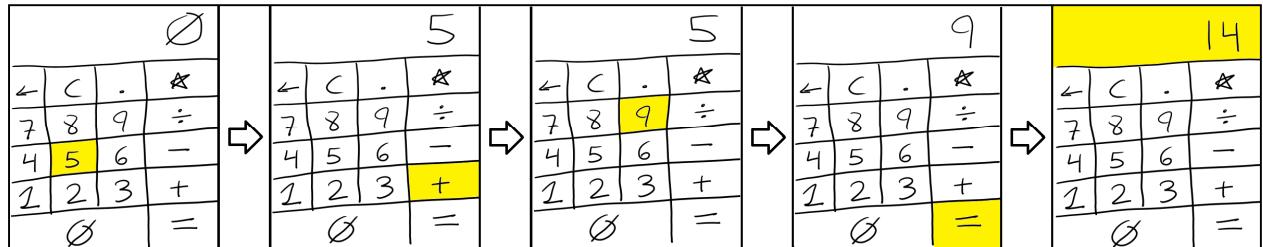


Figure 26: Storyboard of a test sequence. Highlighted areas represent mouse clicks in the first four states and the field to be verified in the last state.

“14” to be displayed as the result. Now, LEET can be used to record a sequence of interactions with the prototype to use as a test script. The result is the storyboard of a test shown in Figure 25.

Automation information added to this prototype through ActiveStory Enhanced makes it possible to find widgets and verify information about them. For example, the hot zone (indicated by yellow highlighting in the first state of the storyboard in Figure 25) above the “5” button has “Five” set as its AutomationID. When the actual GUI is created, if the actual “5” button is given the same ID, the test will find and click it just as it would the button in the prototype. Similarly, the Content property of a hot zone above the display region on the prototype has been set to 14 in the goal state of the prototype, and its ID is set to Display. In the Automation API, widgets that display text by default tend to set their Name property to that text. Thus, it is possible to verify that a widget with AutomationID “Display” exists, and the name property of this widget is “14.” This will work on the actual GUI for most widgets that display text.

The test we’ve just recorded can run successfully on the prototype just like it could against a real system – the first step of the test would be to start the prototype, which can be done through ActiveStory Enhanced by having LEET execute a “START” command targeting the executable file, generated by ActiveStory Enhanced, associated with the storyboard. The next step is for a developer to create the actual GUI. For this example, Windows Presentation Framework (WPF) is used because it will automatically add much of the necessary automation information to widgets from fields that are commonly used. For example, after adding the five,

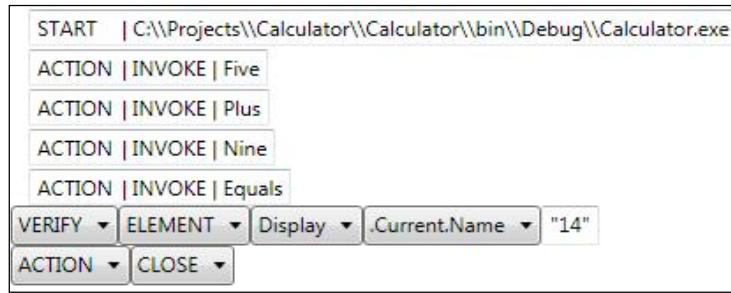


Figure 27: Test for the calculator's simple addition feature.

nine, plus, and equals buttons and the display field to the main window, we need only change the name property of each widget so that it matches the corresponding widget in the prototype – WPF ensure that these are automatically mapped to AutomationIDs that can be used by LEET to identify which widgets should be interacted with at each step of a test.

Now, in order to run this test on the actual GUI instead of the prototype, the START action in the test need only be changed to target the executable file for our GUI. In Figure 26, START has been changed to start the actual GUI instead of the prototype, and will do start the application as its first step. The test will fail, as shown in Figure 27, because none of the application logic for these buttons exists at this point.

Note that the test fails on the second to last line of Figure 26 – verifying the content of the display field – when running against the actual GUI. It is able to locate each widget and perform actions, and it fails because the content of the display is “0” instead of “14.” This is



Figure 28: Failing test - application logic still missing.

because keyword-based testing will tolerate cosmetic changes to widgets – changes to the look and feel of widgets that don't impact their functionality. As can be seen by the difference between Figure 25 and Figure 28, these changes can look quite dramatic. However, from the perspective of keyword-based testing, the interfaces are nearly identical – they are still composed of buttons with the same names. Widgets can be resized, moved, even switched between analogous types without causing LEET to indicate a test failure – which makes sense because functionally the application is still behaving the same in the actual implementation as it was in the prototype.

After adding in the event-handling logic for each button, which includes updating the display, the original test now passes. The interface can be completed and this test will still function appropriately, as seen in Figure 28.

6.2 Validation Study¹⁹

The feasibility of using UITDD was evaluated through a pilot experiment with front-end designers and developers using a controlled experiment that aimed at determining if UI tests based on a UI prototype can serve as a basis for developing the UI code. In this pilot experiment, 3 participants with backgrounds in design and/or development were tasked with implementing event handlers in a GUI-based application, ExpenseManager. This application as provided to test subjects only contained GUI code, meaning there were no event handlers to actually implement its functionality. In order to achieve this task, participants were provided with:



Figure 29: A complete interface. The original test still passes.

¹⁹ From work published in [48].

- 1) An interactive prototype of ExpenseManager demonstrating its functionality
- 2) Automated GUI tests that executed against the prototype of ExpenseManager
- 3) Automated GUI tests that executed against the non-functional GUI of ExpenseManager

Participants were observed during the experiment to record which tools they used to determine the correctness of their implementation, asked to complete a post-study questionnaire, and were briefly interviewed about their experience.

6.2.1 ExpenseManager

The application to be developed by study applicants is a GUI-based application for the creation, saving, modification, and totaling of expense reports, similar to those required by many organizations before they will reimburse employees for travel expenses. ExpenseManager's GUI contains three tabs: View Totals; New Report; and Modify Report. The View Totals tab allows users to view the total expense incurred by, reimbursed to, and outstanding from employees either for the entire company or for a specific individual. The New Report tab is used for creating and saving new expense reports. These expense reports consist of a name and trip number, as well as a dollar amount for transportation, lodging, meals, and conference fees. These amounts are summarized into a subtotal at the bottom of each report, and, if any reimbursement has been made, this is displayed along with the amount that remains to be reimbursed to the employee. The Modify Report tab allows users to update the amount reimbursed for specific saved reports, and changes made to reports will be reflected in the View Totals tab.

In order to create the version of ExpenseManager used for the study, a self-evaluation was performed in order to make sure that the process for UITDD can work in principle to demonstrate both that the process can work in principle and that the study would be possible for participants to complete. The functionality described above was first determined, a prototype of the system was created using SketchFlow, tests were created for the SketchFlow prototype using

LEET, a new WPF application was created, the tests were modified to target the WPF application, and development of the WPF application was continued until all tests passed – essentially, it was created using the same process that could be used in industry. The final version included a complete GUI, event handlers, and a very simple data structure for storing reports. After I had completed this activity successfully and development of a working implementation of all of the features in the WPF version of ExpenseManager was finished, all of the code in the event handlers – the code that actually makes the GUI do anything – was deleted. Participants in the study sessions would only need to focus on adding code to these empty event handlers in order to make the application function as specified in its tests.

This entire process, from prototypes through to passing tests, took me five hours to complete. For the study, participants were given an hour to implement the missing event handlers for ExpenseManager so that the automated GUI tests would pass again. This simplification decreased the amount of time that would be required for participants to finish the study as well as focused the evaluation on the usefulness of the GUI tests to participants' GUI event development.

6.2.1.1 Prototyping ExpenseManager

SketchFlow was used for prototyping in this study in place of ActiveStory Enhanced in order to demonstrate that the approach is not tool-dependent. In SketchFlow, prototypes can be created by dragging-and-dropping widgets onto a canvas or they can be drawn and linked to other states in the prototype using hot zones, as in ActiveStory Enhanced. Each canvas represents a different state in the storyboard in the same way that each picture represents a state in ActiveStory Enhanced. Widgets can be assigned behaviors, one of which is of course transitioning to a different state – just like a hot zone in ActiveStory Enhanced. The resulting prototypes are executable, allowing test users to interact with them as though they were

functional applications. The fact that prototypes are executable means that it can be targeted using CRTs to create tests, thus making UITDD possible. However, the prototypes are intentionally kept rough-looking – as can be seen in Figure 29, they could have conceivably been created by scanning pictures of an interface drawn by someone with very good handwriting and a straightedge. In essence, SketchFlow and ActiveStory Enhanced, while differing greatly from a designer’s point of view, are two different ways of creating rough-looking, arguably low-fidelity prototypes that will achieve the same goal from the perspective of UITDD.

This type of prototype is convenient for use in usability evaluation [67] [68], specifically Wizard of Oz-like tests [69] [70] [68], in which a user is presented with a mock system and instructed to perform a task. This form of evaluation allows designers to identify potential usability flaws and refine the design of the GUI before expending effort on its implementation. Typically, this is an iterative process, involving several evaluations and revisions, until the

Expense Report	
View Totals	New Report
Modify Report	
Entity:	John Smith/I
Name:	John Smith
ID:	I
Transport:	750
Lodging:	500
Meals:	150
Conference Fees:	100
Subtotal:	1500
Paid:	0
Owing:	1500
Update Report	

Figure 30: One page from the SketchFlow prototype of ExpenseManager.

majority of usability flaws are resolved and user satisfaction reaches an acceptable level.

One of the states of the SketchFlow prototype of ExpenseManager can be seen in Figure 30, while the storyboard of states and possible transitions involved in the prototype, represented as a map in SketchFlow, can be seen in Figure 30.

6.2.1.2 Creating Tests for ExpenseManager

SketchFlow prototypes are essentially WPF user interfaces using styles that make them look like hand-drawn prototypes. One side-effect of this is that the widgets of which they are composed are automatically annotated with information that will make it possible to use keyword-based testing through the Windows Automation API. As with ActiveStory Enhanced, it is possible to go through the normal TDD process of writing tests, running them to watch them fail, writing code until they pass, and refactoring the codebase.

For the development of ExpenseManager, four tests were recorded from the prototype verifying the following functionality:

1. Reports do not get totaled under the View Totals tab until they are saved.
2. Saved reports are still totaled under the View Totals tab even when they have been cleared from the New Report tab.
3. Saved reports modified from the Modify Report tab are also updated on the View Totals tab.

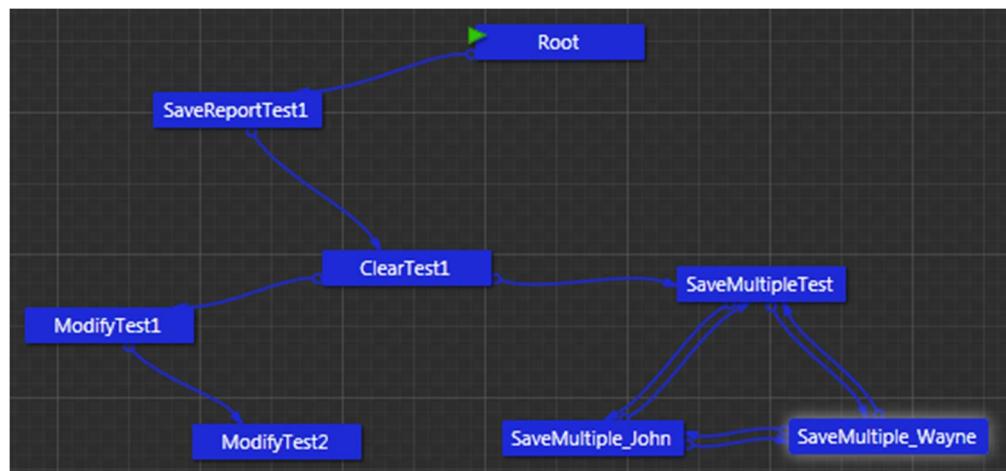


Figure 31: State map of ExpenseManager prototype.

- The View Totals tab is able to show both the totals for all saved reports and the totals for all reports corresponding to a specific user.

It is important to stress that these tests verify that the implementation of ExpenseManager is functionally equivalent to the functionality expressed in the prototype.

6.2.2 Participants

Three participants were recruited from the Calgary Agile Methods User Group (CAMUG)²⁰. Table 7 presents some demographic information about the participants. While all participants had substantial experience with GUI development, none of the participants had previously used prototypes as part of this process. Additionally, none of the participants had experience with automated GUI testing lasting more than two years, despite having been developing GUIs for more than two years. Further, only one participant used TDD for applications he was currently developing. These points imply that the integration of the approach to UITDD described in this paper might be difficult to achieve, since the participants I was able to recruit, if representative of the general population of front-end developers did not have significant experience with the prerequisite techniques of test driven development of GUIs, user interface prototyping, and TDD. Despite this, participants were able to adapt to the process of UITDD, which could point towards its adoptability in practice.

Table 7: Quantitative responses from survey.

	Participant 1	Participant 2	Participant 3
Experience with Testing	Over 2 Years	None	0-2 Years
Experience with GUI Testing	0-2 Years	None	0-2 Years
Experience with GUI Development	Over 2 Years	Over 2 Years	Over 2 Years
Experience with UI Prototyping	None	None	None
Uses TDD on Own Projects	No	No	Yes

²⁰ <http://calgaryagile.com/>

6.3 Results

The results of this pilot evaluation indicate that this approach to UITDD could be of use to our participants in their normal jobs. These results come from two sources: observations of participants collected during the study and results collected from post-study questionnaires and interviews.

6.3.1 During the Study

All three participants entered into the same development cycle during their development of ExpenseManager. First, participants would run a single test against the prototype in order to determine which story to begin implementing next. Once they had picked a test, participants would use the GUI builder to identify the widgets involved in the story. From the GUI builder, they would then navigate to a related event handler and begin implementing discrete parts of a story. After changes had been made to an event handler, participants would run their implementation of ExpenseManager and manually interact with it in order to determine if their implementation met their expectations. If it did, they would then run tests against the prototype again in order to determine if their implementation also met the expectations defined in the

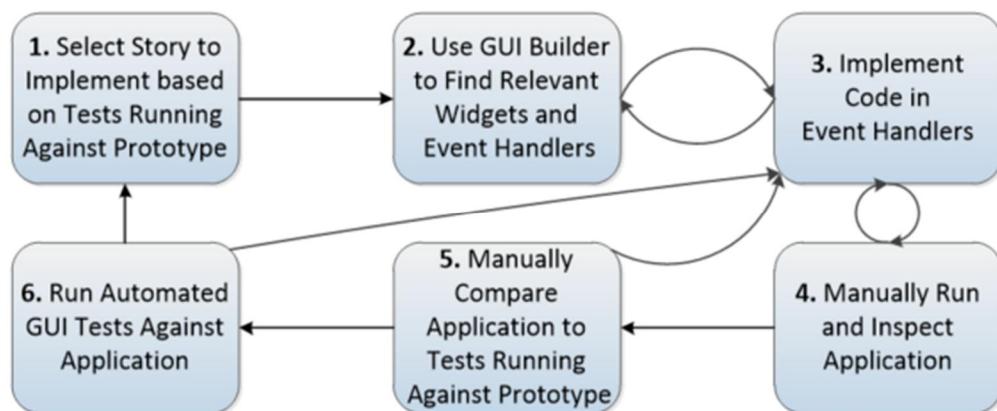


Figure 32: Workflow for development of GUI-based application observed during studies.

prototype. Only after a feature was completely implemented, in their eyes, would participants then run the test against the actual GUI. A diagram of this workflow can be seen in Figure 31.

Participants used the provided automated GUI tests as the sole indication of completeness of functionality – as soon as a test passed, participants moved on to the next feature. This could be a liability since, if the test suite for a feature is incomplete, the resulting implementation will also be incomplete. This might imply that it may be beneficial to either treat GUI tests as unit tests by testing a number of different test procedures and error conditions or ensure that the developer who is responsible for coding a given feature in a GUI is involved in creating its tests. The former would help to provide developers with more examples of how a feature is expected to function so that developers would be encouraged to implement complete solutions. The latter would encourage developers to culture an understanding of how a feature is intended to work, which could lead to stronger implementations. However, it is possible that this could also be due to a lack of investment in the system being developed, as ExpenseManager is only a simple application.

Before code has been added to the event handlers, tests run against the application will immediately fail, without a corresponding visual indication as to which widget caused the test failure. The error message provided for this failure by LEET is not inherently understandable, and, once participants noticed this, they would tend to hold off on running tests against the application until it was mostly complete. For example, before code is added to update the subtotal of an expense report when values are added, a test might fail with the message “Assert.AreEqual failed. Expected:<750>, Actual:<0>.” Notably missing from this message is the element that was expected to have a value of 750, for example. Further, it was observed that participants were unable to understand the form in which tests are recorded. However, this is

more an issue with the specific testing tool used than it is with UITDD itself. This was addressed in later development work on LEET.

By the end of each study session, each participant had at least some of the tests passing when run against the application. A summary of which tests were passing by test participant can be seen in Table 8.

It is interesting to note that Participant 1 had no previous experience with Visual Studio, WPF, or C#. Participant 1 did have significant previous experience with Java, and was able to use this, in conjunction with the resources provided during the study, to complete more features than the other participants.

When given a choice of several resources to aid in the development of ExpenseManager, developers used GUI tests to determine when stories were complete, and would not move on until the test related to a feature accessible through the GUI of the application was passing. The observations described in this subsection seem to support the idea that GUI tests used in a test-first setup serve an important role in the development process.

6.3.2 After the Study

After each study session was completed, the participant was asked to fill out a survey which assessed the participant's background in GUI development and testing, as well as the participant's perception of the usefulness and usability of the UITDD approach.

Table 8. Passing Tests by Participant

	Clear Report	Modify Report	Save Report	Save Multiple
Participant 1	✓	✗	✓	✓
Participant 2	✓	✗	✓	✗
Participant 3	✓	✗	✓	✗

Several survey questions were also included to gauge participants' perception of the usefulness of UITDD in light of their experiences in developing ExpenseManager. The answers to these questions are recorded in Table 9. The first and second questions were ranked from A (Very Useful) to E (Useless). Despite the similarity of the first and second questions, both were asked in order to gauge the usefulness of this approach to TDD both in the current and in a broader context. These responses imply that the participants saw the potential of this approach to be of some benefit in their own work.

The first question also included a follow-up question asking participants to explain why they chose that answer. Participant 1 noted that the tests that were provided "checks for interactions/updates I may have missed during coding," but notes that the technique is "only as useful as the assertions!" This second point relates to an instance where he noticed a bug in a feature, despite the fact that the test relating to it was passing. Participant 2 responded that TDD of GUIs lets you "see functionality straight away," while participant 3 noted that running the GUI tests was "faster than running code" to verify it manually. It is interesting to note that each participant found something different of value in the approach.

Participants were also asked to rank the resources they were provided with in terms of decreasing usefulness for their development task. Their responses can be seen in Table 10,

Table 9. Perception of Usefulness of TDD of GUIs

	Participant 1	Participant 2	Participant 3
Found TDD of ExpenseManager To Be	B. Somewhat Useful	A. Very Useful	C. Useful
Would Expect TDD of Own Work To Be	B. Somewhat Useful	A. Very Useful	B. Somewhat Useful
Would Consider Using This Approach to TDD on Own Projects	Yes	Yes	Yes

below. It is of note that participants' perception of the importance of various resources does not line up with observations recorded by the researcher during the course of study sessions as represented in Figure 31, above. Instead of ranking features by frequency of use, participants seem to have ranked resources based on the value they provided to the development effort. Participants 1 and 3 noted that the UI prototype was a "standard to compare against when coding" and "captured the intent of user functionality," whereas participant 2 noted that the GUI builder "gave me names of things I needed to use." This implies that one way to immediately make this approach to UITDD provide higher value to users would be to improve the interface through which tests are run so that participants can understand the features represented by the UI tests, or to make technical details of widgets being tested more visible so that users can understand the expected details of the system they are building.

Participants pointed out various usability issues stemming from tools used in this study. First, participants remarked on the unreadability of the error messages produced by tests generated by LEET. Participants requested that, when tests are run through Visual Studio, the error messages that result from failing tests need to be much clearer, and include contextual information about the widgets the test was interacting with when the test failed. They also suggested that, when tests are run through LEET itself, tests should also be able to pause after

Table 10. Ranking of Available Resources

Usefulness	Participant 1	Participant 2	Participant 3
Highest	UI Prototype	GUI Builder	UI Prototype
-	Communication with Researcher	Debugger	Communication with Researcher
-	UI Tests	UI Prototype	UI Tests
-	GUI Builder	Communication with Researcher	GUI Builder
Lowest	Debugger	UI Tests	Debugger

the last action is taken in a test script so that users can see the final state of the application. Similarly, users expressed a desire to be able to pause and step through each line of a test – similar to the way a debugger functions – rather than the current behavior, in which each step of a test executes after a set delay. Finally, users were unsure of when widgets were being interacted with by test code.

6.4 Concluding Remarks on UITDD

This pilot evaluation suggests that the approach to UITDD outlined in this chapter is useful for development of GUI-based applications. The study's participants used GUI tests primarily for determining when a feature was done and as a roadmap for the completion of the application. As a result of observation of participants during the study, I developed a workflow showing the way in which study participants utilized UI tests in order to determine when they were done working on a specific feature of the sample application. Participants also reported through the post-experiment survey that they felt GUI tests were useful for test-driving the development of the test application. Through these surveys, participants also expressed the view that UITDD had benefits and would be a useful practice.

Chapter Seven: Limitations

There are of course a number of weaknesses with any body of work that need to be acknowledged so that the assumptions behind results can be objectively critiqued, and this thesis is no exception. In this chapter, I describe some of the limitations of the work I presented in this thesis both in terms of essential weaknesses of the research methodologies themselves and weaknesses of the way in which studies were conducted and analyses were carried out.

7.1 Weaknesses of the Interview Study

The interview study presented in Chapter Four was carried out in order to get a basis for understanding how practitioners are currently using GUI testing as part of their development process. The weakness with using an interview study to collect this data is that, on a fundamental level, what is being collected is practitioners' perception of how they use GUI testing. Interviewees are likely to focus on occurrences that were surprising, or that were very positive or very negative, but not on topics that have become mundane. For instance, in the study presented here, interviewees spoke at length about GUI tests "failing", but did not mention instances where more-familiar unit tests failed, or draw comparisons between their suites of GUI tests and their suites of unit tests. Ideally, it would have been useful to observe the interviewees in their normal workplace environment to conduct an observational study to provide additional verification for the topics the interviewees mentioned. Since this was not possible, I have instead made an effort to cross-reference topics brought up in the interview study with instances where the same topic occurred in the related work section or in the systematic mapping studies (Chapter Five) in order to provide support for the interviewees' reports.

Next, interviews rely strongly on the expertise of the individual conducting the interview. It is easy, especially for novice interviewers, to skew an interview towards personal biases by, for example, asking follow-up questions on only topics relating to their personal interests. In

order to minimize the odds of this happening, and to increase consistency across all interviews conducted for the study, interviews were conducted from a script outlining the basic topics to be covered in each interview and the same script was used for all interviews. However, of the 8 interviews used for the final analysis in Chapter Four, 1 was conducted by another researcher, which raises the possibility of inconsistencies between the ways in which interviews were conducted. Again, to guard against the possibility of differences in interviewing style leading to biases in the analysis, both researchers participated in the analysis of this data.

Further, as with any study based on a researcher's interpretations of data, there is a possibility that an individual researcher's bias might skew results in a certain direction, or that one researcher working alone would miss results that others might notice. In order to guard against this possibility, all interview data was analyzed by at least two researchers, with any disagreements resolved through discussion.

Attempts were made to control for differences in the interviewees themselves. For example, rather than performing an analysis of all interviews – including those done with interviewees with little experience in GUI testing – only those interviewees with significant, longer-term experience were included. This was done because I noticed early in the analysis that interviewees with limited experience with GUI testing – specifically, with less than one year of experience – focused almost exclusively on very tool-specific issues (like an inability to figure out how to use a specific tool) rather than on essential issues with the nature of GUI testing itself. By excluding these participants, it was possible to focus the study around ways to improve the process of GUI testing, rather than ways to improve specific GUI testing *tools*. However, it would be interesting in the future to go back and look at what issues less-experienced participants encounter, or what issues are more pressing early in the process of integrating GUI

testing into a development process, in order to investigate the topic of adoption of GUI testing. The analysis in this dissertation focused only on people who had persevered to overcome issues with specific tools to make GUI testing part of their development process.

However, the result of this was that there were only 8 participants used in the study. This low number of participants could have skewed results towards issues only encountered by specific individuals. Again, this was why an effort was made to cross-reference points made in this study to papers presented in other sections of this thesis. Additionally, by focusing on participants with higher experience – in most cases, more than two years of experience with GUI testing – the study may have gravitated towards issues that had not had time to impact participants with only short-term experience. The most obvious examples here would be issues relating to test maintenance and test evolution – two very central themes for 8 participants that were focused on. Of these, it's worth noting that no entirely new topics were raised by the final two participants interviewed (7 and 8).

7.2 Weaknesses of Systematic Mapping Studies

The limitations of the systematic mapping studies presented in Chapter Five fall into two categories: those that are related to the nature of systematic mapping studies, and those that are related to the way in which these specific systematic mapping studies were carried out.

7.2.1 Weaknesses of Systematic Mapping Studies Generally

The first weakness with systematic mapping studies stems from the data on which they are based. A systematic mapping study is intended to provide an overview of a field of study – the general trends, but not the fine-grained details. Data is collected by focusing on the key sections of publications – mainly abstracts and titles, but additionally the introduction and concluding sections where necessary. Because of this, the accuracy of a systematic mapping study is highly dependent on the ability of the authors of the source papers to accurately

summarize the key results of their work. As was noted several times in Chapter Five, many authors in the fields of GUI testing and Agile testing did *not* describe all of the key aspects of their work – they neglected to provide motivation for why the work is necessary, to describe the problem they are attempting to solve, or to describe the approach that was used – in the titles, abstracts, introductions, or conclusions of their work. This is the origin of the “None” keyword used in these chapters. It might be possible to gain more insight into these papers by performing a full systematic review, but this was not done in this dissertation. In a systematic review, ideally, the ways studies are conducted and reported are analyzed in order to find ways of making results comparable across studies so that each study can be treated as numeric data point as part of a whole. However, this methodology requires some amount of consistency in the way in which studies are conducted and results are reported in order to be truly effective. Before this could really be feasible in the fields of testing in Agile development environments or GUI testing, the way results are reported would need to be made significantly more detailed and consistent across conferences. Attempting to perform a systematic review of papers in the existing paperset would require the exclusion of nearly all of the practitioner publications and many of the academic publications because the experimental designs and the results are simply not described in enough detail to be directly comparable to each other.

As a first step in this direction, it would be useful for conferences dealing with testing in Agile development environments and GUI testing to begin using a standard format for abstracts (and, ideally, for descriptions of studies and results) – perhaps one based on those used in other fields²¹. Not only would this make it a requirement for authors to include all crucial information related to their publication right in the abstract, but it would also make it possible to directly

²¹ See for example: http://www.nlm.nih.gov/bsd/policy/structured_abstracts.html

compare the results of different research efforts to each other so that systematic studies could really determine what the effect of different techniques is.

The second major weakness with systematic mapping studies is that, paradoxically, they require the researcher to either be or to become an expert in all of the terms used in the field. I say paradoxically because, in many fields, systematic mapping studies are used as initial research projects through which students who are new to the subject area can orient themselves. However, this might be easier to do in fields with well-structured terminology. At least in publications on testing in Agile development environments and GUI testing, there are a large number of synonyms for identical or nearly identical techniques. For example, within the field of automated GUI testing, several highly-similar terms are routinely used: event-sequence graph; event-flow graph; directed interaction graph; complete interaction sequence; event-interaction graph; and actionable knowledge graph. While it is of course permissible to define terms for clarity within a given publication, the terms in the previous sentence refer to nearly identical models which are generated and used in nearly identical ways for the purpose of representing and generating tests from GUIs. If a researcher were not familiar with the way in which these models were applied, might be mistaken for distinctly different approaches to GUI testing.

Also, as I mention in Chapter Five, the search for papers can only find papers that self-identify with the keywords the researcher uses. Specifically in the first mapping on agile testing, I note that authors may not think to include search terms that could be assumed when considering the venue at which they are published – papers published at the Agile and XP conferences, for instance, might not bother to include “Agile” as a keyword. While this makes sense in context, it also makes it difficult to search for these papers automatically, unless, again, the researcher conducting the mapping is already familiar with the conferences that should be investigated and

has the resources to manually go through proceedings. If a systematic mapping study is done as a researcher's introduction to a field – a common method for finding out which areas have been investigated and which have not – this is problematic as it would require some level of familiarity with the field.

The generalization of the several preceding paragraphs is that it's possible that relevant papers were not found during the literature search phases of the systematic mappings. Part of the reason that the second systematic mapping study [55] was run was to compare a manual search approach with the automated search used in [54]. As a result, a significant number of additional papers were found. While this was good in the current context in that it allowed me to cross-verify the frequency of topics between the two studies and see that they did largely match up, it does imply that, in future work, combination literature search methods should be applied. For example, I would recommend that automated literature searches be combined with manual searches through conferences that either come up frequently in the papers identified during the automated search or that the researchers are aware may be related. For any papers identified during the manual search that were missed by the automated search, the terms used in the automated search should be refined until they additionally discover these new papers. Researchers should also look through the works cited in all included papers and determine whether these papers should be included in order to increase the number of relevant papers that are used in the analysis. It is worth noting that this process was used to find additional papers for the related work section, Chapter Three, of this thesis.

Another major consideration with meta-analyses is the issue of repeatability. Work by Kitchenham et al. [71] has demonstrated that, given the same research question, different researchers will select different studies for inclusion and draw different conclusions based on

these sources. This means that, for higher repeatability, researchers need to present as much detail as possible about the methodology used for this type of study and to follow generally-accepted guidelines where possible. Based on this, not only was the approach for each study described in detail in Chapter Five, but the studies themselves were based on existing systematic studies done by Petersen et al. [72] and Dybå and Dingsøyr [73].

Finally, one problem that is well-known in other fields that in all likelihood also applies in the fields of GUI testing and testing in Agile development environments is the *file drawer problem* (for a detailed treatment of this topic, see: [74]). This term refers to the inevitability that only a minority of studies that are conducted are actually published²² and therefore available for inclusion in a meta-analysis such as a systematic mapping. This could be because the papers report negative results or because a similar study has already been published. The danger is that, if these unpublished studies are presenting valid results, a systematic study conducted without them will come to different conclusions than if they would have been included.

7.2.2 Limitations of the Systematic Mapping Studies in Chapter Five

One of the limitations of doing systematic studies on practitioner-oriented topics is that practitioners do not tend to communicate with each other in the same way as academics do. Where it's reasonable to assume that academics publish all their relevant work in peer-reviewed venues, and thus the body of these publications is a good representation of academic work, practitioners tend to publish in less-formal venues. Many ideas are shared through blog postings, forums, and books. These formats tend to be faster than peer-reviewed venues, meaning that practitioner thinking on topics can change very rapidly, even though they obviously do not represent rigorous work. The mappings presented in this thesis do not take this perspective into

²² For a collection of acceptance rates at some software engineering conferences, see:
<http://web.engr.illinois.edu/~taoxie/seconferences.htm>

account – in large part because of the difficulty involved in determining whether a given blog, for example, is worth including in the results. In order to get a really thorough understanding of practitioner thinking about Agile GUI testing, these sources would need to be incorporated somehow. In this thesis, I instead chose to validate the results of my mapping studies using an interview study in order to address this issue, but it would be very useful in future work to find some way of including results from these non-standard venues.

Another issue with the mapping studies was the low number of researchers involved. In the mapping of GUI testing work done in Chapter 5.2, only a single researcher was involved in the search for and analysis of papers. In the other papers, at least two authors were involved in the coding of each paper, but, again, I performed the majority of the analysis of results alone for all of these studies. This could of course have led to researcher bias. Additionally, the other researchers involved in coding were not experts in GUI testing and were trained in this topic by me, so it is possible that my personal biases on this subject were also transmitted to the other researchers.

One weakness specific to the second systematic mapping of testing in Agile development environments [55] was not all authors from the first systematic mapping on this topic also participated in the second. This is significant because the results of each study are compared significantly in and, in some instances, even combined in this thesis. In order to minimize the risk of keywords being used inconsistently across studies, I personally participated in the coding of every paper in both studies. However, again, this engenders a certain amount of risk of researcher bias. A different way around this issue could have been to measure inter-rater reliability statistically so that a numerical representation of the likelihood that, presented with the same abstract, two researchers would tag it with the same set of keywords. This method is useful

in that it can be used to provide a confidence value for how likely the keywords associated with papers are to be correct, but was not done in any of these mappings.

A final issue with the systematic mappings of testing in Agile development environments is that none of the publications included for analysis really describe what they mean by “Agile”. This is important from my personal experience in that, in my six years researching Agile, only one or two companies that say they are practicing Agile methodologies are actually practicing Agile methodologies. It would be extremely helpful if authors would provide more details in publications about what their working definition of “Agile” is so that researchers performing meta-analyses would be better able to determine the context in which different approaches are purported to work.

7.3 Weaknesses with the UITDD Studies

The weaknesses with this investigation into UITDD can be classed into two categories: limitations with the approach to UITDD itself and limitations with the evaluation.

7.3.1 Limitations of the Approach

The approach to UITDD presented in Chapter Six relies on a software development environment in which a specific type of user experience design technique is being used maturely. In order to adopt this approach to UITDD, teams are required to opt to use interactive prototypes of a GUI so that tests can be recorded from this prototype. This means that, in order to apply this new testing technique, teams may additionally be required to adopt a new UX technique. However, I feel this is justified in that, if a team is trying to perform automated GUI testing without performing usability evaluations to determine if customers find the GUI acceptable, the amount of change related to the GUI would likely be high enough that it would make automated GUI testing on the project very difficult. By requiring this UX step be done, not only is UITDD possible, but it’s likely creation of automated GUI tests after the GUI’s design has stabilized

would be more likely to be successful. This is due to a fundamental assumption that UX can reduce the number of changes required of a user interface.

However, a weakness of this approach is that it suffers from the limitations of both CRTs and low-fidelity prototypes. If a CRT is not able to record a given interaction – like, perhaps, replaying a tap-and-hold gesture on a touch-enabled device – then it would not be possible to record a test of this interaction even though the prototype is capable of expressing it. The same is true of prototypes. If a low-fidelity prototype is not capable of demonstrating the way a feature should work – for example, menus fading out after a user has made a selection – then it's not possible to record this interaction even if the CRT was capable of it. Developers would not be able to create tests for these sorts of interactions without either manually editing test scripts (or extending the CRTs they use to create tests) or waiting until after the actual GUI was coded. However, it's worth noting that CRTs are becoming increasingly capable of recording complex interactions – for example, even touch interactions on touch-enabled tables can be recorded and replayed as tests.^{23/24}

7.3.2 Limitations of the Evaluation

The first and most severe problem with this pilot study is that the evaluation was only done on a set of three participants. While care was taken to use only participants with relevant backgrounds, a larger pool of participants would have been beneficial. However, it was significantly difficult to gather even this many participants with a reasonable level of experience for use in this study given that the study was conducted at the University of Calgary and required participants to invest time in traveling to the University and participating in the study outside of

²³ Recording and replaying interactions with the Microsoft Surface Simulator: [http://msdn.microsoft.com/en-us/library/ee804844\(v=surface.10\).aspx](http://msdn.microsoft.com/en-us/library/ee804844(v=surface.10).aspx)

²⁴ LEET support some of this functionality, as well:
http://nsercsurfnet.ca/system/newsletters/newsletters/000/000/012/original/07.SurfNet_Newsletter_May-Jun_2011.pdf?1395099249

normal business hours. As it is, there is a risk that the pool of participants may not accurately represent the pool of potential users at large. However, care was taken to cross-reference the approaches used by the participants to other studies presented in this thesis – to show that other studies, or studies run by completely independent groups of researchers, also came to similar conclusions.

Second, the study was only a controlled experiment. As such, there was a time limit for participants to learn the approach to UITDD, to orient themselves to the codebase of ExpenseManager, to read and understand the tasks that were asked of them, and to implement the code to complete those tasks. It would be useful in future research to either conduct longer study sessions, multiple study sessions with each participant, or longer-term case studies. This would allow more information to be collected about the ways in which the system can be used, and improvements that can be made that only occur to participants when exposed to the system for longer periods of use.

Ultimately, it would be valuable to conduct a longer-term study that would more accurately represent how this approach to UITDD could be used in practice. A controlled experiment can only provide so much data in that the study environment is necessarily limited so that researchers can focus on evaluating only a few aspects of an approach. The entire point of controlling an experiment is to reduce the number of variables that one has to consider. However, this of course makes it more difficult to determine if the results of a controlled experiment will transfer over to real-world settings. While it would of course be more difficult, it would be more realistic to collect results from the long-term use of the technique in a real-life setting.

Third, the application used for this study may have been too complicated for a one-hour experiment. Participants were unable to complete the implementation of the system in the limited time they were given. For future controlled experiments, it could make sense to provide participants with a simpler test system and set of tasks to work with. However, ultimately, this approach to UITDD must be evaluated on the development of complicated, real-world systems in order to decisively determine its real-world value.

Finally, participants had significant difficulty using LEET as the testing tool in this study. This is likely a confounding issue that makes it difficult to extricate issues with the testing tool used to create and run tests from issues with the approach to UITDD itself. If LEET continues to be the testing tool of choice in future experiments, a technique needs to be used to make it visually explicit that LEET is interacting with a specific widget as a test is running. The level of detail provided by LEET upon test failure should also be enhanced in order to make it easier to interact with. Some work in this direction has been completed – for example, LEET now places a red rectangle around the widget with which it's trying to work while a test is running, and saves a screenshot of the state of the GUI to a results directory when a test fails. This was done in order to make it easier to work with LEET generally. However, in future evaluations, it still might make sense to make use of a more widely-used GUI testing tool in order to demonstrate that this approach to UITDD can work with more common – and less experimental – CRTs.

Chapter Eight: Future Work

Throughout the process of doing the research behind this thesis, I have had to prioritize between a tantalizing array of possible research directions. It was simply not possible to follow up on all possible research topics, and each study I performed raised a number of new issues to investigate. These topics for future work are broken down in this section by the chapter of this thesis to which they related.

8.1 Future Work from Interviews

The first thing that needs to be done in future work is to complete the analysis of the interviews with less-experienced participants in order to gain additional insight into the issues people encounter when they are just beginning to use GUI testing. 10 interviews that were not included in the study have already been conducted and transcribed, and the first stage of coding the interviews has already been completed, but they were not included in the analysis presented in this thesis because the participants tended to focus on issues with specific tools. Additionally, many of these participants reported that they only put a low amount of effort into learning how to use specific GUI testing tools and had largely given up the practice – which makes the interviews less useful from the perspective of how people actually use GUI tests, but interesting in their own right in terms of why people give up on GUI testing. Further analysis of these interviews should be done in order to see if a set of key issues are likely to cause developers and testers to give up on GUI testing entirely, as this result would be useful in the creation of future generation of testing tools. Additionally, further interviews could be conducted to focus on these issues specifically, which could conceivably branch out into a large research area in its own right. From the mappings presented in Chapter Five, this topic has not been previously explored and could be useful.

The second thing to come out of the interview studies was the centrality of maintenance to GUI testing efforts. However, few studies have attempted to quantify just how much time is consumed by test maintenance. In [28], it was briefly mentioned that the team determined that about one hour of maintenance per test case was required, but this was not measured in a controlled fashion. In [24] and [33], authors determine that it is feasible to repair broken GUI test cases, but do not provide recommendations for performing test maintenance or for reducing the likelihood or cost of maintenance. Future work should be done in order to follow up on the results of [24] in terms of determining the root cause of GUI test case failures, including determining whether the root causes are the same between desktop, web, and touch –based GUI test failures. Additionally, confounding factors that can impact the difficulty of repairing a suite of broken GUI tests should also be explored. This could be done either using controlled regression studies, as in [24] and [33], or by actually performing an evaluation with a real team in industry as development of a product progresses.

Relatedly, the impact of the way a suite of GUI tests is designed and structured should also be explored in future work – again, perhaps through a case study done with an industry partner. Many researchers have previously proposed ways of structuring GUIs to improve testability, but, so far as I am aware, no prior work has looked into ways of structuring GUI test suites to improve maintainability. In [44], a three-tier structure for GUI test suites was proposed, but this approach is only superficially related to the one presented in Chapter Four – the authors there are separating concerns in a test script in order to make it easier for goal-directed AI systems to perform automatic testing of a GUI. In Chapter Four, I propose structuring a GUI in order to make it easier for human testers to understand and maintain a test suite, and this is the aspect of architecture that I believe needs further study.

8.2 Future Work on Literature Mappings

The first way the systematic mapping studies could be expanded upon would be to extend them into full systematic literature reviews. This would be done by reading and summarizing the entirety of papers so that the actual results and studies within papers could be compared against each other. The purpose of this sort of study is to be able to make recommendations about how automated GUI testing should be performed in Agile development environments in order to achieve specific benefits – something that can't be done based on the results of a systematic mapping study alone due the limited amount of information provided in the paper abstracts.

Further, as mentioned earlier, it would make sense to find ways of including more practitioner-oriented sources in future meta-analyses. The obvious way to achieve this would be to find ways of including blogs, technical books, and other non-peer-reviewed sources in a meta-analysis. However, this raises issues with comparisons between these sources. Obviously, the level of rigor in a blog post is not comparable to the level of rigor in a peer-reviewed academic publication, so it would be difficult to figure out ways of legitimately comparing results across the two domains. Further, work would need to be done in order to establish inclusion criteria for non-peer-reviewed sources based on at least the level of detail with which they describe the problem they are facing, the way in which the solutions they propose are described or evaluated, or perhaps even the way the online community responds to their ideas via upvotes, reblogging, online review articles for published books, or other sharing and rating mechanisms. So, while it would be helpful to find ways of including more practitioner-oriented sources in future mappings and reviews, quite a bit of work might need to go into figuring out how to actually construct inclusion and exclusion criteria for collecting sources, or how to actually analyze the resulting set of articles.

One other way of getting a more complete view of the state of practice in automated GUI testing would be to perform a survey of all tools that can be used for this purpose. Comparing the features available in these tools would be useful in that it would allow inferences to be made about whether techniques proposed in academic publications are actually making it into use in industry. If, for example, commercial GUI testing tools support model-based testing, or specific types of models, this would be encouraging, as model-based testing makes up a very large fraction of research into GUI testing. If, on the other hand, very few commercial tools offer this functionality, which has been undergoing active research for many years, this might imply that this research is not aligning with the needs of practitioners in an immediate sense. This could also be done from the other direction: academic teams that built GUI testing tools for their research could be contacted and asked for information about their tools – specifically, how many times the tools have been downloaded, how many times they have been contacted about the tools by people using them in industry, if they are aware of how many users their tools have, etc. Both approaches would allow a fuller understanding of the flow of ideas between academic and practitioner domains in the field of automated GUI testing.

8.3 Future Work on UITDD

The first loose end in the UITDD studies that needs to be tied up is including more participants and conducting a longer-duration experiment session. Ideally, a long-term case study should be used to evaluate the way in which UITDD is used during the course of an actual software development project in industry. This would primarily verify that the approach works outside a laboratory setting, but would also allow a greater number of participants to be observed over a longer term, thereby allowing more thorough collection of results.

The approach to UITDD proposed in this dissertation also focuses on low-fidelity prototypes and high-fidelity prototypes. These prototypes are discussed in terms of throwaway

prototyping – an approach where every prototype is discarded after it is used for usability evaluations and potentially to create GUI tests. Mixed-fidelity prototypes – combinations of hand-drawn placeholders and actual working widgets – weren't investigated. This type of prototype is specifically quite interesting in that it could be used to gradually evolve a rough prototype into the finished, working GUI instead of throwing out the prototype and essentially reimplementing the GUI based on the prototype. This would overcome one difficulty I foresee with the approach to UITDD: that there is a leap between the tests running against the prototype and then running on a completely separate GUI. By slowly evolving the prototype from a rough drawing into a polished UI, the amount of effort involved in maintaining the tests created for UITDD might be reduced.

Further, this approach to UITDD should be evaluated on applications for GUI paradigms besides traditional desktop and web applications – for touch-based applications, including those for phones and tablets, and for touchless interfaces, including those implemented for the Kinect sensor. This is possible given the creation of prototyping tools geared towards these domains, including the successor to ActiveStory: Enhanced, ProtoActive [75].

Chapter Nine: Conclusions

The main goal of this thesis was to determine if Agile methodologies and automated GUI testing are compatible. I investigated this topic from a number of perspectives, starting with an interview study of people who regularly use automated GUI tests as part of their development process. I then performed systematic mapping studies of GUI testing and testing in Agile development environments which were published in peer-reviewed venues to see how well the results of the interviews I conducted correlated with established literature. By looking at areas where GUI testing and testing in Agile development environments were not well-aligned, I decided to determine if the two domains could be unified by designing and evaluating an approach to TDD of GUI-based applications.

In the interview study conducted in Chapter Four, I conducted semi-structured interviews to determine how automated GUI tests are currently used in practice, what issues its users encounter, and what best practices they have discovered in order to overcome these issues. The main uses I found were unsurprising from an Agile perspective: focus was placed on using automated GUI tests for regression testing and for acceptance testing. Study participants did not use a test-first approach to GUI testing or even mention it as a goal. Rather, participants were concerned mainly with issues of test evolution and test suite architecture. Evolution of the system under test was a major concern in that it would necessitate test suite maintenance, which was seen as a major, unwelcome cost. In order to overcome issues with maintenance, the participants mostly recommended nurturing the understanding that test maintenance is a part of ongoing development, aggressively refactoring test code, and understanding that the maintenance overhead of some tests may not be worthwhile. Architecture of the suite of GUI tests was also a major concern in that popular GUI testing tools will encourage GUI tests to be structured in a very naïve format, making it possible and likely for relatively innocuous changes to the GUI

under test to cause failures in a large number of tests, necessitating a large maintenance overhead. In order to prevent this cost, interview participants recommended that test suites be refactored towards a multi-tier architecture separating the user stories demonstrated by tests from an intermediate layer containing the logic of interactions from a layer of basic interaction with widgets in the GUI. Additionally, due to the fact that 10 of our original participants gave up on GUI testing within a short period of time, it's clear that the basic adoption of automated GUI testing as part of a software engineering effort needs additional focus. Many themes uncovered in these interviews resonated with the results of other studies – both in this thesis and in related work.

In the first two systematic mapping studies I describe in Chapter Five, testing in Agile development environments was investigated from the perspective of peer-reviewed publications. From looking at the topics discussed in these publications, TDD was the main focus of this subject area. However, the participants in the interview study did not indicate that TDD was something they practiced, nor that they had considered it, leading me to conclude that TDD is an area in which further research is necessary for automated GUI testing to be compatible with testing in Agile development environments. The mappings did indicate that test maintenance was an area of difficulty, but papers did not explicitly discuss techniques for facilitating test maintenance. Instead, publications focused on improving issues like test quality or adoption of Agile testing, demonstrating that there may be a disconnect with the topics related to Agile testing discussed in this published literature and the way practitioners actually use GUI tests. Architecture was not discussed in the set of papers used for the systematic mappings of testing in Agile development environments, however. Finally, adoption of Agile testing practices occurred

repeatedly as an issue of concern in these studies, which correlates with adoption being a problem in the interview study.

In the third systematic mapping study described in Chapter Five, GUI testing was investigated in terms of topics discussed in peer-reviewed publications, just as Agile testing was investigated. Based on the results of this study, GUI testing research is mostly focused around use of model-based testing and automatic test suite generation – neither of which was discussed in any of the previous studies. In the studies discussed previously, testing focused mainly around regression testing, acceptance testing, and TDD – model-based testing and automatic test generation were not discussed within the publications on testing in Agile development environments. Again, this shows that each field is centered around different topics at present. I chose to join these fields of research through, again, investigating how automated GUI tests can be created as part of a TDD workflow. The GUI testing study did, however, show that maintenance was a concern in about 10% of the GUI testing papers surveyed, indicating that this might be another potential vehicle for the integration of automated GUI testing and testing in Agile development environments. As with testing in Agile development environments, test suite architecture also did not occur in the GUI testing study, indicating, based on the interview study, that it may be an area in which both fields can be improved.

Based on the findings of the previous 4 studies, two further studies were conducted on a novel approach to TDD of GUI-based applications and presented in Chapter Six. In the first study, a basic approach to the use of low-fidelity prototypes as a basis for creating tests for use in subsequent development of a GUI was proposed and a proof-of-concept that the technique was plausible was presented. Following this, a pilot evaluation was conducted in which participants were provided with low-fidelity prototypes and GUI tests and asked to complete demonstration

of 4 features in a sample application. Based on observations conducted during this study, a workflow for how GUI tests were incorporated into a development process was presented. This workflow correlates with existing literature, demonstrating that this novel approach to UITDD is consistent with the needs of practitioners. Further, participants indicated through written surveys that they found UITDD useful and would be interested in applying it to their own projects, demonstrating that the approach has merit in practice.

Overall, this dissertation presents the results of 6 novel studies that demonstrate areas in which automated GUI testing and testing in Agile development environments are compatible – something that has not previously been reported. Further, this study provides a pilot evaluation of a technique that could feasibly integrate GUI testing into an Agile development process, which fulfills the primary goal listed at the beginning of this dissertation. Further, these studies fulfill the secondary goal of this thesis by identifying a plan for future work: additional research is needed in the areas of test suite maintenance and test suite architecture in order to better integrate automated GUI testing and testing in Agile development environments, and further research is needed for how to get people to adopt these testing practices in the first place.

Works Cited

- [1] D. West, T. Grant, M. Gerush and D. D'Silva, "Agile Development: Mainstream Adoption Has Changed Agility," Forrester Research, 2010.
- [2] G. Vanderburg, "A Simple Model of Agile Software Practices - or - Extreme Programming Annealed," in *Object-Oriented Programming, Systems, Languages, and Applications*, New York, 2005.
- [3] K. Beck, Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.
- [4] K. Schwaber, Agile Project Management with Scrum, Microsoft Press, 2004.
- [5] A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, First ed., Addison-Wesley Professional, 2004.
- [6] M. Poppendieck and T. Poppendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley, 2003.
- [7] D. J. Anderson, Kanban, Blue Hole Press, 2010.
- [8] K. Beck, J. Grenning, R. C. Martin, M. Beedle, J. Highsmith, S. Mellor, A. von Bennekum, A. Hunt, K. Schwaber, A. Cockburn, R. Jeffries, J. Sutherland, W. Cunningham, J. Kern, D. Thomas, M. Fowler and B. Marick, "Manifesto for Agile Software Development," February 2001. [Online]. Available: <http://agilemanifesto.org/>.
- [9] T. Hellmann, A. Hosseini-Khayat and F. Maurer, "Agile Interaction Design and Test-Driven Development of User Interfaces - A Literature Review," in *Agile Software Development: Current Research and Future Directions*, T. Dingsøyr, T. Dybå and N. Brede Moe, Eds., Trondheim, Springer, 2010.
- [10] E. W. Dijkstra, "On The Reliability of Mechanisms," April 1970. [Online]. Available:

<http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>. [Accessed 03 March 2014].

- [11] G. Meszaros, xUnit Test Patterns: Refactoring Test Code, 2nd ed., Upper Saddle River, NJ: Addison-Wesley, 2007.
- [12] A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces," University of Pittsburgh, 2001.
- [13] P. Brooks, B. Robinson and A. M. Memon, "An Initial Characterization of Industrial Graphical User Interface Systems," in *International Conference on Software Testing, Verification, and Validation*, Denver, 2009.
- [14] B. Robinson and P. Brooks, "An Initial Study of Customer-Reported GUI Defects," in *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.
- [15] Q. Xie and A. M. Memon, "Designing and Comparing Automated Test Oracles for GUI-Based Software Applications," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, February 2007.
- [16] A. M. Memon, I. Banerjee and A. Nagarajan, "What Test Oracle Should I Use for Effective GUI Testing?," in *International Conference on Automated Software Engineering*, Montreal, 2003.
- [17] Q. Xie and A. M. Memon, "Studying the Characteristics of a "Good" GUI Test Suite," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*, Raleigh, NC, 2006.
- [18] T. D. Hellmann, "Enhancing Exploratory Testing with Rule-Based Verification," 2010.
- [19] T. D. Hellmann and F. Maurer, "Rule-Based Exploratory Testing of Graphical User

- Interfaces," in *Agile Methods in Software Development*, Salt Lake City, Utah, 2011.
- [20] G. Meszaros, "Agile Regression Testing Using Record & Playback," in *Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, California, 2003.
- [21] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in *AGILE 2006*, 2006.
- [22] J. Bowen and S. Reeves, "UI-Driven Test-First Development of Interactive Systems," in *Engineering Interactive Computing Systems*, Pisa, Italy, 2011.
- [23] W. Chen, T. Tsai and H. Chao, "Integration of Specification-Based and CR-Based Approaches for GUI Testing," in *19th International Conference on Advanced Information Networking and Applications*, 2005.
- [24] T.-H. Chang, T. Yeh and R. C. Miller, "GUI Testing Using Computer Vision," in *Human Factors in Computing Systems*, Atlanta, Georgia, 2010.
- [25] C. Fu, M. Grechanik and Q. Xie, "Inferring Types of References to GUI Objects in Test Scripts," in *International Conference on Software Testing, Verification, and Validation*, 2009.
- [26] 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.
- [27] J. Andersson and B. Geoff, "The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing," in *Extreme Programming and Agile Processes in Software Engineering*, Garmisch-Partenkirchen, Germany, 2004.
- [28] C. McMahon, "History of a Large Test Automation Project using Selenium," in *Agile*, Chicago, 2009.

- [29] M. Alles, D. Crosby, B. Harleton, G. Pattison, C. Erickson, M. Marsiglia and C. Stienstra, "Presenter First: Organizing Complex GUI Applications for Test-Driven Development," in *Agile Conference*, Minneapolis, Minnesota, 2006.
- [30] M. Finsterwalder, "Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment," in *Extreme Programming and Flexible Processes*, Villasimius, Italy, 2001.
- [31] 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.
- [32] A. M. Memon, "Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 2, pp. 1-36, October 2008.
- [33] Y. Shewchuk and V. Garousi, "Experience with Maintenance of a Functional GUI Test Suite Using IBM Rational Funcitonl Tester," in *International Conference on Software Engineering and Knowledge Engineering*, Boston, USA, 2010.
- [34] 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.
- [35] 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.
- [36] M. Grechanik, Q. Xie and F. Chen, "Maintaining and Evolving GUI-Directed Test Scripts,"

- in *International Conference on Software Engineering*, Vancouver, Canada, 2009.
- [37] Z. Yin, C. Miao, Z. Shen and Y. Miao, "Actionable Knowledge Model for GUI Regression Testing," in *IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2005.
- [38] S. Burbeck, "Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)," 1987, 1992. [Online]. Available: <http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>.
- [39] B. Marick, "Bypassing the GUI," *Software Testing and Quality Engineering Magazine*, pp. 41-47, September/October 2002.
- [40] R. Martin, "The Test Bus Imperative: Architectures That Support Automated Acceptance Testing," *IEEE Software*, vol. 22, no. 4, pp. 65-67, 2005.
- [41] M. Feathers, "The Humble Dialog Box," 2002. [Online]. Available: <http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf>. [Accessed 13 08 2014].
- [42] C. Erickson, R. Palmer, D. Crosby, M. Marsiglia and M. Alles, "Make Haste, Not Waste: Automated System Testing," in *Extreme Programming and Agile Methods*, New Orleans, Louisiana, 2003.
- [43] J. Andersson, G. Bache and C. Verdoes, "Multithreading and Web Applications: Further Advances in Acceptance Testing," in *Extreme Programming and Agile Processes in Software Engineering*, Sheffield, UK, 2005.
- [44] A. Kervinen, M. Maunumaa and M. Katara, "Controlling Testing Using Three-Tier Model Architecture," in *Workshop on Model Based Testing*, Vienna, Austria, 2006.

- [45] A. Ruiz and P. Y. W., "Test-Driven GUI Development with TestNG and Abbot," in *IEEE Software*, 2007.
- [46] A. Ruiz and Y. W. Price, "GUI Testing Made Easy," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2008.
- [47] T. D. Hellmann, A. Hosseini-Khayat and F. Maurer, "Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design," in *International Conference on Software Testing, Verification, and Validation Workshops*, Paris, 2010.
- [48] T. D. Hellmann, A. Hosseini-Khayat and F. Maurer, "Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation," in *International Conference on Agile Processes in Software Engineering and Extreme Programming*, Madrid, 2011.
- [49] J. Bowen and S. Reeves, "UI-Design Driven Model-Based Testing," in *International Workshop on Formal Methods for Interactive Systems*, London, UK, 2009.
- [50] J. Saldaña, The Coding Manual for Qualitative Researchers, London: Sage, 2009.
- [51] T. Hellmann, E. Moazzen, A. Sharma, M. Akbar, J. Sillito and F. Maurer, "An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices," University of Calgary, Calgary, 2014.
- [52] 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.
- [53] S. MacDonnel, M. Shepperd, B. Kitchenham and E. Mendes, "How Reliable Are Systematic Reviews in Empirical Software Engineering?," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 676-687, Sep. - Oct. 2010.

- [54] T. Hellmann, A. Sharma, J. Ferreira and F. Maurer, "Agile Testing: Past, Present, and Future," in *Agile*, Salt Lake City, UT, 2012.
- [55] T. Hellmann, A. Chokshi, Z. Abad, S. Pratte and F. Maurer, "Agile Testing: A Systematic Mapping Across Three Conferences," in *Agile*, Nashville, TN, 2013.
- [56] K. Petersen, R. Feldt, S. Mujtaba and M. Mattsson, "Systematic Mapping Studies in Software Engineering," in *12th International Conference on Evaluation and Assessment in Software Engineering*, Bari, Italy, 2008.
- [57] R. Wieringa, N. Maiden, N. Mead and C. Rolland, "Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion," *Journal of Requirements Engineering*, vol. 1, no. 11, pp. 102-107, 2005.
- [58] S. Fraser, D. Astels, K. Beck, B. Boehm, J. McGregor, J. Newkirk and C. Poole, "Discipline and Practices of TDD (Test Driven Development)," in *Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, California, 2003.
- [59] O. Mishali, Y. Dubinsky and S. Katz, "The TDD-Guide Training and Guidance Tool for Test-Driven Development," in *Agile Processes in Software Engineering and Extreme Programming (XP)*, Limerick, Ireland, 2008.
- [60] T. Reenskaug, "The Original MVC Reports," February 2007. [Online]. Available: http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf.
- [61] R. Jeffries and G. Melnik, "Guest Editors' Introduction: TDD - The Art of Fearless Programming," *IEEE Software*, vol. 24, no. 3, pp. 24-30, May-June 2007.
- [62] N. Nagappan, E. M. Maximilien, T. Bhat and L. Williams, "Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams," in

Empirical Software Engineering, 2008.

- [63] H. Munir, M. Moayyed and K. Petersen, "Considering Rigor and Relevance When Evaluating Test Driven Development: A Systematic Review," *Information Software and Technology*, vol. 56, no. 4, pp. 375-394, 2014.
- [64] A. Hosseini-Khayat, Y. Ghanam, S. Park and F. Maurer, "ActiveStory Enhanced: Low Fidelity Prototyping and Wizard of Oz Usability Testing Tool," in *International Conference on Agile Processes and eXtreme Programming*, Pula, Italy, 2009.
- [65] A. Hosseini-Khayat, Y. Ghanam, S. Park and F. Maurer, "ActiveStory Enhanced: Low-Fidelity Prototyping and Wizard of Oz Usability Tool," in *Agile Processes in Software Engineering and Extreme Programming*, 2009.
- [66] G. Jurca, T. D. Hellmann and F. Maurer, "Integrating Agile and User-Centered Design: A Systematic Mapping and Review of Evaluation and Validation Studies of Agile-UX," in *Agile Methods in Software Development*, Orlando, FL, 2014.
- [67] B. Buxton, *Sketching User Experiences*, Morgan Kaufmann, 2007.
- [68] C. Barnum, *Usability Testing and Research*, New York: Pearson Education, 2002.
- [69] A. Hosseini-Khayat, "Distributed Wizard of Oz Usability Testing for Agile Teams," Calgary, 2010.
- [70] P. Wilson, "Active Story: A Low Fidelity Prototyping and Distributed Usability Testing Tool for Agile Teams," August 2008.
- [71] B. Kitchenham, P. Brereton, L. Zhi, D. Budgen and A. Burn, "Repeatability of Systematic Literature Reviews," in *Evaluation & Assessment in Software Engineering*, Durham, UK, 2011.

- [72] K. Petersen, R. Feldt, S. Mujtaba and M. Mattsson, "Systematic Mapping Studies in Software Engineering," in *Evaluation & Assessment in Software Engineering*, Bari, Italy, 2008.
- [73] T. Dybå and T. Dingsøyr, "Empirical Studies of Agile Software Development: A Systematic Review," *Information and Software Technology*, vol. 50, no. 9-10, pp. 55-63, 2008.
- [74] J. D. Scargle, "Publication Bias: The "File Drawer" Problem in Scientific Inference," *Journal of Scientific Exploration*, vol. 14, no. 1, pp. 91-106, 2000.
- [75] T. de Souza Alcantara, J. Ferreira and F. Maurer, "Interactive Prototyping of Tabletop and Surface Applications," in *Engineering Interactive Computing Systems*, London, UK, 2013.
- [76] B. Marick, "Classic Testing Mistakes," in *Software Testing, Analysis, and Review*, San Diego, 1997.
- [77] D. E. Knuth, "Knuth: Frequently Asked Questions," [Online]. Available: <http://cs.stanford.edu/~uno/faq.html>. [Accessed 14 February 2014].
- [78] J. Itkonon, M. V. Mäntylä and C. Lassenius, "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing," in *First International Symposium on Empirical Software Engineering and Measurement*, Madrid, Spain, 2007.
- [79] R. Rosenthal, "The File Drawer Problem and Tolerance for Null Results," *Psychological Bulletin*, vol. 86, no. 3, pp. 638-641, May 1979.

Appendix I: Papers from First Agile Testing Systematic Mapping Study [54]

1. Knublauch, H. Extreme Programming of multi-agent systems. 2002.
2. Edwards, S.H., Improving student performance by evaluating how well students test their own programs. ACM Journal on Educational Resources in Computing, 2003. 3(3).
3. Fraser, S., et al. Discipline and practices of TDD (Test Driven Development). 2003.
4. George, B. and L. Williams. An initial investigation of test driven development in industry. 2003.
5. Kaufmann, R. and D. Janzen. Implications of test-driven development a pilot study. 2003.
6. Meszaros, G., R. Bohnet, and J. Andrea, Agile regression testing using record and playback. 2003. p. 111-119.
7. Nelson, R., A testing checklist for database programs: Managing risk in an agile environment. 2003. p. 91-95.
8. Pancur, M., et al. Comparison of frameworks and tools for test-driven development. 2003.
9. Pancur, M., et al. Towards empirical evaluation of test-driven development in a university environment. in EUROCON 2003. Computer as a Tool. The IEEE Region 8. 2003.
10. Pinna, S., et al., Developing a tool supporting XP process. 2003. p. 151-160.
11. Pinna, S., et al., XPSwiki: An agile tool supporting the planning game. 2003. p. 104-113.
12. Pipka, J.U., Test-driven web application development in Java. 2003. p. 378-393.
13. Reichlmayr, T. The agile approach in an undergraduate software engineering course project. 2003.
14. Vanhanen, J., J. Jartti, and T. Kähkönen, Practical experiences of agility in the Telecom industry. 2003. p. 279-287.
15. Baumeister, H., A. Knapp, and M. Wirsing. Property-driven development. 2004.
16. Baumeister, J., D. Seipel, and F. Puppe. Using automated tests and restructuring methods for an agile development of diagnostic knowledge systems. 2004.
17. George, B. and L. Williams, A structured experiment of test-driven development. Information and Software Technology, 2004. 46(5 SPEC. ISS.): p. 337-342.
18. Geras, A., M. Smith, and J. Miller, A survey of software testing practices in Alberta. Canadian Journal of Electrical and Computer Engineering, 2004. 29(3): p. 183-191.
19. Hayashi, S., et al., Test driven development of UML models with SMART modeling system. 2004. p. 395-409.
20. Manhart, P. and K. Schneider. Breaking the ice for agile development of embedded software: An industry experience report. 2004.
21. Melnik, G. and F. Maurer. Introducing agile methods: Three years of experience. 2004.

22. Van Schooenderwoert, N. and R. Morsicato. Taming the embedded tiger - Agile test techniques for embedded software. 2004.
23. Andersson, J., G. Bache, and C. Verdoes. Multithreading and web applications: Further adventures in acceptance testing. 2005.
24. Bache, G., R. Mugridge, and B. Swan. Hands-on domain-driven acceptance testing. 2005.
25. Geras, A., et al. A survey of test notations and tools for customer testing. 2005.
26. Janzen, D. and H. Saiedian, Test-driven development: Concepts, taxonomy, and future direction. Computer, 2005. 38(9): p. 43-50.
27. Louridas, P., JUnit: Unit testing and coding in tandem. IEEE Software, 2005. 22(4): p. 12-15.
28. Martin, R.C., The test bus imperative: architectures that support automated acceptance testing. Software, IEEE, 2005. 22(4): p. 65-67.
29. Mugridge, R. and W. Cunningham. Agile test composition. 2005.
30. Nair, S. and P. Ramnath. Teaching a goliath to fly. 2005.
31. Pautasso, C. JOpera: An agile environment for Web service composition with visual unit testing and refactoring. 2005.
32. Read, K., G. Melnik, and F. Maurer. Student experiences with executable acceptance testing. 2005.
33. Ryu, H.Y., B.K. Sohn, and J.H. Park. Mock objects framework for TDD in the network environment. 2005.
34. Smith, M., et al. E-TDD - Embedded test driven development a tool for hardware-software co-design projects. 2005.
35. Wellington, C.A. Managing a project course using extreme programming. 2005.
36. Yagüe, A. and J. Garbajosa. The role of testing in agile and conventional methodologies. 2005.
37. Yu, J., et al. Agile testing of location based services. 2005.
38. Ambler, S.W., Survey says: Agile works in practice. Dr. Dobb's Journal, 2006. 31(9): p. 62-64.
39. Arakawa, S. and S. Yukita. An effective agile teaching environment for java programming courses. 2006.
40. Berg, C. and S.W. Ambler, Assurance & agile processes. Dr. Dobb's Journal, 2006. 31(7): p. 42-45.
41. Bowyer, J. and J. Hughes. Assessing undergraduate experience of continuous integration and test-driven development. 2006.
42. Brolund, D. and J. Ohlrogge, Streamlining the agile documentation process test-case driven documentation demonstration for the XP2006 conference. 2006. p. 215-216.

43. Carlsson, R. and M. Rémond. EUnit - A lightweight unit testing framework for erlang. 2006.
44. Crispin, L., Driving software quality: How test-driven development impacts software quality. IEEE Software, 2006. 23(6): p. 70-71.
45. Degerstedt, L. and A. Jönsson. LINTest, A development tool for testing dialogue systems. 2006.
46. Duarte, A., et al. GridUnit: Software testing on the grid. 2006.
47. Elssamadisy, A. and J. Whitmore. Functional testing: A pattern to follow and the smells to avoid. 2006.
48. Hedberg, H. and J. Iisakka. Technical Reviews in Agile Development: Case Mobile-D™. in Quality Software, 2006. QSIC 2006. Sixth International Conference on. 2006.
49. Ho, C.W., et al. On Agile performance requirements specification and testing. 2006.
50. Holmes, A. and M. Kellogg. Automating functional tests using selenium. 2006.
51. Janzen, D. and H. Saiedian. On the Influence of Test-Driven Development on Software Design. in Software Engineering Education and Training, 2006. Proceedings. 19th Conference on. 2006.
52. Karamat, T. and A.N. Jamil. Reducing test cost and improving documentation in TDD (Test Driven Development). 2006.
53. Kongsli, V. Towards agile security in web applications. 2006.
54. Kum, W. and A. Law. Learning effective test driven development: Software development projects in an energy company. 2006.
55. Puleio, M. How not to do agile testing. 2006.
56. Sangwan, R.S. and P.A. Laplante, Test-driven development in large projects. IT Professional, 2006. 8(5): p. 25-29.
57. Talby, D., et al., Agile software testing in a large-scale project. Software, IEEE, 2006. 23(4): p. 30-37.
58. Turnu, I., et al., Modeling and simulation of open source development using an agile practice. Journal of Systems Architecture, 2006. 52(11): p. 610-618.
59. Andrea, J., Envisioning the Next-Generation of Functional Testing Tools. Software, IEEE, 2007. 24(3): p. 58-66.
60. Araújo, B.C., et al. Web-based tool for automatic acceptance test execution and scripting for programmers and customers. 2007.
61. Bouillon, P., et al., EZUNIT: A framework for associating failed unit tests with potential programming errors. 2007. p. 101-104.
62. Chubov, I. and D. Droujkov, User stories and acceptance tests as negotiation tools in offshore software development. 2007. p. 167-168.
63. de Vries, S., Software Testing for security. Network Security, 2007. 2007(3): p. 11-15.

64. Dobson, J. Performance testing on an agile project. 2007.
65. Domino, M.A., R.W. Collins, and A.R. Hevner, Controlled experimentation on adaptations of pair programming. *Information Technology and Management*, 2007. 8(4): p. 297-312.
66. Far, B. Software reliability engineering for agile software development. 2007.
67. Fletcher, M., et al. Evolving into embedded development. 2007.
68. Frezza, S. and M.-H. Tang. Testing as a Mental Discipline: Practical Methods for Affecting Developer Behavior. in *Software Engineering Education & Training*, 2007. CSEET '07. 20th Conference on. 2007.
69. Gagliardi, F., Epistemological justification of test driven development in agile processes. 2007. p. 253-256.
70. Gutierrez, R.L.Z., B. Mendoza, and M.N. Huhns. Behavioral queries for service selection: An agile approach to SOC. 2007.
71. Hummel, O. and C. Atkinson, Supporting agile reuse through extreme harvesting. 2007. p. 28-37.
72. Irvine, S.A., et al. Jumble java byte code to measure the effectiveness of unit tests. 2007.
73. Johnson, P.M. and H. Kou. Automated recognition of test-driven development with Zorro. 2007.
74. Katara, M. and A. Kervinen, Making model-based testing more agile: A use case driven approach. 2007. p. 219-234.
75. Kongslie, V. Security testing with selenium. 2007.
76. Landre, E., H. Wesenberg, and J. Ølmheim. Agile enterprise software development using domain-driven design and test first. 2007.
77. Lanubile, F. and T. Mallardo, Inspecting automated test code: A preliminary study. 2007. p. 115-122.
78. Lindegaard, K.P. and D. Kristiansen. Signal based functionality testing of control systems. 2007.
79. Martin, R.C., Professionalism and test-driven development. *IEEE Software*, 2007. 24(3): p. 32-36.
80. McKinney, D. and L.F. Denton. Developing collaborative skills early in the CS curriculum in a laboratory environment. 2007.
81. Rahman, S.M. Applying the TBC method in introductory programming courses. 2007.
82. Reichhart, S., T. Gîrba, and S. Ducasse, Rule-based assessment of test quality. *Journal of Object Technology*, 2007. 6(9): p. 231-251.
83. Sanchez, J.C., L. Williams, and E.M. Maximilien. On the sustained use of a test-driven development practice at IBM. 2007.
84. Savoia, A., Are you immune to test infection? *Dr. Dobb's Journal*, 2007. 32(7): p. 10.

85. Simons, A.J.H., JWalk: A tool for lazy, systematic testing of java classes by design introspection and user interaction. *Automated Software Engineering*, 2007. 14(4): p. 369-418.
86. Simons, A.J.H. and C.D. Thomson. Lazy systematic unit testing: JWalk versus JUnit. 2007.
87. Siniaalto, M. and P. Abrahamsson. A comparative case study on the impact of test-driven development on program design and test coverage. 2007.
88. Vodde, B. and L. Koskela, Learning test-driven development by counting lines. *IEEE Software*, 2007. 24(3): p. 74-79.
89. Wang, Y., et al. Ontology-based test case generation for testing web services. 2007.
90. Weiss, D. and M. Zduniak, Automated integration tests for mobile applications in Java 2 micro edition. 2007. p. 478-487.
91. Xueling, S. and F. Maurer. A tool for automated performance testing of Java3D applications in agile environments. 2007.
92. Ambler, S.W., Scaling test-driven development. *Dr. Dobb's Journal*, 2008. 33(2): p. 71-73.
93. Brooks, G. Team Pace Keeping Build Times Down. in Agile, 2008. AGILE '08. Conference. 2008.
94. Cannizzo, F., R. Clutton, and R. Ramesh. Pushing the boundaries of testing and continuous integration. 2008.
95. Carlson, B. An Agile classroom experience: Teaching TDD and refactoring. 2008.
96. Ferreira, C. and J. Cohen. Agile systems development and stakeholder satisfaction: A South African empirical study. 2008.
97. Gehringer, D., DFT benefits for developers - And entire organizations. *Electronic Products* (Garden City, New York), 2008. 50(7).
98. Geras, A. Leading manual test efforts with agile methods. 2008.
99. Kakarontzas, G., I. Stamelos, and P. Katsaros. Product line variability with elastic components and test-driven development. 2008.
100. Lily, H., et al. E-RACE, A Hardware-Assisted Approach to Lockset-Based Data Race Detection for Embedded Products. in *Software Reliability Engineering*, 2008. ISSRE 2008. 19th International Symposium on. 2008.
101. Londesbrough, I. A Test Process for all Lifecycles. 2008.
102. Mostefaoui, G.K. and A. Simpson. On quality assurance of web services in agile projects: An experience report. 2008.
103. Mugridge, R., Managing agile project requirements with storytest-driven development. *IEEE Software*, 2008. 25(1): p. 68-75.
104. Nagappan, N., et al., Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering*, 2008. 13(3): p. 289-302.

105. Park, S. and F. Maurer. The benefits and challenges of executable acceptance testing. 2008.
106. Puolitaival, O.P., Adapting model-based testing to agile context. 2008. p. 1-80.
107. Rendell, A. Effective and pragmatic test driven development. 2008.
108. Shah, V. and A. Nies. Agile with fragile large legacy applications. 2008.
109. Shaye, S.D. Transitioning a team to Agile test methods. 2008.
110. Siniaalto, M. and P. Abrahamsson, Does test-driven development improve the program code? Alarming results from a comparative case study. 2008. p. 143-156.
111. Zawadzki, M., Extending continuous integration into ALM. Dr. Dobb's Journal, 2008. 33(10): p. 56-63.
112. Adams, J. Test-driven data structures: Revitalizing CS2. 2009.
113. Bondi, A.B. and J.P. Ros. Experience with training a remotely located performance test team in a quasi-agile global environment. 2009.
114. Connolly, D., F. Keenan, and F. McCaffery. Developing acceptance tests from existing documentation using annotations: An experiment. 2009.
115. Gallardo-Valencia, R.E. and S.E. Sim. Continuous and collaborative validation: A field study of requirements knowledge in agile. 2009.
116. Gautham, R.N. Testing Processes in Business-Critical Chain Software Lifecycle. in Software Engineering, 2009. WCSE '09. WRI World Congress on. 2009.
117. Ghanam, Y. and F. Maurer. Extreme Product Line Engineering: Managing Variability and Traceability via Executable Specifications. in Agile Conference, 2009. AGILE '09. 2009.
118. Hanssen, G.K. and B. Haugset. Automated acceptance testing using fit. 2009.
119. Hill, J.H., et al. Unit testing non-functional concerns of component-based distributed systems. 2009.
120. Ieshin, A., M. Gerenko, and V. Dmitriev. Test automation: Flexible way. 2009.
121. Kessler, D. and T.J. Andersen. Herding cats: Managing large test suites. 2009.
122. Knauth, T., C. Fetzer, and P. Felber. Assertion-driven development: Assessing the quality of contracts using meta-mutations. 2009.
123. Lacoste, F.J. Killing the gatekeeper: Introducing a continuous integration system. 2009.
124. McMahon, C. History of a large test automation project using selenium. 2009.
125. Nan, L., U. Praphamontipong, and J. Offutt. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. in Software Testing, Verification and Validation Workshops, 2009. ICSTW '09. International Conference on. 2009.
126. Onions, P. and C. Patel. Enterprise SoBA: Large-scale implementation of acceptance test driven story cards. 2009.

127. Pasternak, B., S. Tyszberowicz, and A. Yehudai, GenUTest: A unit test and mock aspect generation tool. *International Journal on Software Tools for Technology Transfer*, 2009. 11(4): p. 273-290.
128. Robles Luna, E., J. Grigera, and G. Rossi, Bridging test and model-driven approaches in web engineering. 2009. p. 136-150.
129. Smith, M., J. Miller, and S. Daeninck, A test-oriented embedded system production methodology. *Journal of Signal Processing Systems*, 2009. 56(1): p. 69-89.
130. Smith, M., et al., A more agile approach to embedded system development. *IEEE Software*, 2009. 26(3): p. 50-57.
131. Stolberg, S. Enabling Agile Testing through Continuous Integration. in *Agile Conference*, 2009. AGILE '09. 2009.
132. Vejandla, P.K. and L.B. Sherrell. Why an AI research team adopted XP practices. 2009.
133. Vu, J.H., et al. Evaluating test-driven development in an industry-sponsored capstone project. 2009.
134. Atkinson, C., F. Barth, and D. Brenner. Software testing using test sheets. 2010.
135. Bâillon, C. and S. Bouchez-Mongardé, Executable requirements in a safety-critical context with Ada. *Ada User Journal*, 2010. 31(2): p. 131-135.
136. Bartsch, S. Supporting authorization policy modification in agile development of web applications. 2010.
137. de Matos, E.C.B. and T.C. Sousa, From formal requirements to automated web testing and prototyping. *Innovations in Systems and Software Engineering*, 2010. 6(1): p. 163-169.
138. El-Attar, M. and J. Miller, Developing comprehensive acceptance tests from use cases and robustness diagrams. *Requirements Engineering*, 2010. 15(3): p. 285-306.
139. França, A.C.C., F.Q.B. Da Silva, and L.M.R. De Sousa Mariz. An empirical study on the relationship between the use of agile practices and the success of Scrum projects. 2010.
140. Ghanam, Y. and F. Maurer, Linking feature models to code artifacts using executable acceptance tests. 2010. p. 211-225.
141. Hametner, R., et al. The adaptation of test-driven software processes to industrial automation engineering. 2010.
142. Hellmann, T.D., A. Hosseini-Khayat, and F. Maurer. Supporting test-driven development of graphical user interfaces using agile interaction design. 2010.
143. Jureczko, M. and M. Mlynarski, Automated acceptance testing tools for web applications using Test-Driven Development. *Przeglad Elektrotechnicznny*, 2010. 86(9): p. 198-202.
144. Kettunen, V., et al. A study on agility and testing processes in software organizations. 2010.
145. Liu, S., An approach to applying SOFL for agile process and its application in developing a test support tool. *Innovations in Systems and Software Engineering*, 2010. 6(1): p. 137-143.

146. Park, S. and F. Maurer. A network analysis of stakeholders in tool visioning process for story test driven development. 2010.
147. Petersen, K. and C. Wohlin, The effect of moving from a plan-driven to an incremental software development approach with agile practices: An industrial case study. *Empirical Software Engineering*, 2010. 15(6): p. 654-693.
148. Sfetsos, P. and I. Stamelos. Empirical studies on quality in agile practices: A systematic literature review. 2010.
149. Shrivastava, D.P., R.V. Lakshminarayan, and S.V.S.L. Sujatha. New engineering technique of software development. 2010.
150. Yongxiang, H. The Application and Research of Software Testing on Agile Software Development. in E-Business and E-Government (ICEE), 2010 International Conference on. 2010.
151. Besson, F.M., et al. Towards automated testing of web service choreographies. 2011.
152. Borg, R. and M. Kropp. Automated acceptance test refactoring. 2011.
153. Bowen, J. and S. Reeves. UI-driven test-first development of interactive systems. 2011.
154. Causevic, A., D. Sundmark, and S. Punnekkat. Factors limiting industrial adoption of test driven development: A systematic review. 2011.
155. Di Bernardo, R., F. Castor, and S. Soares. Towards agile testing of exceptional behavior. 2011.
156. Di Bernardo, R., et al. Agile Testing of Exceptional Behavior. in Software Engineering (SBES), 2011 25th Brazilian Symposium on. 2011.
157. Dos Santos, A.M., et al. Testing in an agile product development environment: An industry experience report. 2011.
158. Dulz, W. A comfortable TestPlayer for analyzing statistical usage testing strategies. 2011.
159. Ghanam, Y. and F. Maurer. Using acceptance tests for incremental elicitation of variability in requirements: An observational study. 2011.
160. Haugset, B. and G.K. Hanssen. The home ground of automated acceptance testing: Mature use of fit ness. 2011.
161. Keznikl, J., et al. Extensible polyglot programming support in existing component frameworks. 2011.
162. Luna Robles, E., G. Rossi, and I. Garrigós, WebSpec: A visual language for specifying interaction and navigation requirements in web applications. *Requirements Engineering*, 2011. 16(4): p. 297-321.
163. Razak, R.A. and F.R. Fahrurazi. Agile testing with Selenium. in Software Engineering (MySEC), 2011 5th Malaysian Conference in. 2011.
164. Wilkerson, J., J.J. Nunamaker, and R. Mercer, Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development. *Software Engineering, IEEE Transactions on*, 2011. PP(99): p. 1-1.

165. Woo, J., N. Ivezic, and H. Cho, Agile test framework for business-to-business interoperability. *Information Systems Frontiers*, 2011: p. 1-20.

166. Zhang, X. and Q. Wang. The agile software development method in the domain of digital standard test. in E -Business and E -Government (ICEE), 2011 International Conference on. 2011.

Appendix II: Papers from Second Agile Testing Systematic Mapping Study [55]

1. Jalis, A., Probe Tests: A Strategy for Growing Automated Tests around Legacy Code, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 122-130.
2. Johansen, K. and A. Perkins, Establishing an Agile Testing Team: Our Four Favorite “Mistakes”, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 52-59.
3. Kitiyakara, N., Acceptance Testing HTML, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 112-121.
4. Newkirk, J., A Light in a Dark Place: Test-Driven Development with 3rd Party Packages, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 144-152.
5. Rostaher, M. and M. Hericko, Tracking Test First Pair Programming — An Experiment, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 174-184.
6. Stotts, D., M. Lindsey, and A. Antley, An Informal Formal Method for Systematic JUnit Test Case Generation, in Extreme Programming and Agile Methods — XP/Agile Universe 2002, D. Wells and L. Williams, Editors. 2002, Springer Berlin Heidelberg. p. 131-143.
7. Erickson, C., et al., Make Haste, Not Waste: Automated System Testing, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 120-128.
8. Meszaros, G., R. Bohnet, and J. Andrea, Agile Regression Testing Using Record and Playback, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 111-119.
9. Meszaros, G., S. Smith, and J. Andrea, The Test Automation Manifesto, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 73-81.
10. Mugridge, R. Test driven development and the scientific method. in Agile Development Conference, 2003. ADC 2003. Proceedings of the. 2003.
11. Mugridge, R. and E. Tempero. Retrofitting an acceptance test framework for clarity. in Agile Development Conference, 2003. ADC 2003. Proceedings of the. 2003.
12. Nelson, R., A Testing Checklist for Database Programs: Managing Risk in an Agile Environment, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 91-95.
13. Ou, R., Test-Driven Database Development: A Practical Guide, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 82-90.
14. Wenner, R., JNI Testing, in Extreme Programming and Agile Methods - XP/Agile Universe 2003, F. Maurer and D. Wells, Editors. 2003, Springer Berlin Heidelberg. p. 96-110.

15. Andersson, J. and G. Bache, The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 1-10.
16. Andrea, J., Putting a Motor on the Canoo WebTest Acceptance Testing Framework, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 20-28.
17. Andrea, J., Generative Acceptance Testing for Difficult-to-Test Software, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 29-37.
18. Baumeister, H., Combining Formal Specifications with Test Driven Development, in Extreme Programming and Agile Methods - XP/Agile Universe 2004, C. Zannier, H. Erdoganmus, and L. Lindstrom, Editors. 2004, Springer Berlin Heidelberg. p. 1-12.
19. Holcombe, M. and F. Ipaté, Complete Test Generation for Extreme Programming, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 274-277.
20. Ibba, A., Mockrunner – Unit Testing of J2EE Applications –, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 254-257.
21. Kölking, M. and A. Patterson, Going Interactive: Combining Ad-Hoc and Regression Testing, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 270-273.
22. Lokpo, I., M. Babri, and G. Padiou, Assistance for Supporting XP Test Practices in a Distributed CSCW Environment, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 262-265.
23. Lui, K. and K.C. Chan, Test Driven Development and Software Process Improvement in China, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 219-222.
24. Melnik, G., K. Read, and F. Maurer, Suitability of FIT User Acceptance Tests for Specifying Functional Requirements: Developer Perspective, in Extreme Programming and Agile Methods - XP/Agile Universe 2004, C. Zannier, H. Erdoganmus, and L. Lindstrom, Editors. 2004, Springer Berlin Heidelberg. p. 60-72.
25. Mugridge, R., Test Driving Custom \mathcal{F}it Fixtures, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 11-19.
26. Ostroff, J., D. Makalsky, and R. Paige, Agile Specification-Driven Development, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 104-112.
27. Rogers, R.O., Acceptance Testing vs. Unit Testing: A Developer's Perspective, in Extreme Programming and Agile Methods - XP/Agile Universe 2004, C. Zannier, H. Erdoganmus, and L. Lindstrom, Editors. 2004, Springer Berlin Heidelberg. p. 22-31.

28. Van Schooenderwoert, N. and R. Morsicato. Taming the embedded tiger - agile test techniques for embedded software. in Agile Development Conference, 2004. 2004.
29. Walter, B. and B. Pietrzak, Automated Generation of Unit Tests for Refactoring, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 211-214.
30. Wang, Y. and H. Erdogmus, The Role of Process Measurement in Test-Driven Development, in Extreme Programming and Agile Methods - XP/Agile Universe 2004, C. Zannier, H. Erdogmus, and L. Lindstrom, Editors. 2004, Springer Berlin Heidelberg. p. 32-42.
31. Washizaki, H., Y. Sakai, and Y. Fukazawa, Conditional Test for JavaBeans Components, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 282-283.
32. Watt, R. and D. Leigh-Fellows, Acceptance Test Driven Planning, in Extreme Programming and Agile Methods - XP/Agile Universe 2004, C. Zannier, H. Erdogmus, and L. Lindstrom, Editors. 2004, Springer Berlin Heidelberg. p. 43-49.
33. Wenner, R., Abstract Test Aspect: Testing with AOP, in Extreme Programming and Agile Processes in Software Engineering, J. Eckstein and H. Baumeister, Editors. 2004, Springer Berlin Heidelberg. p. 237-241.
34. Andersson, J., G. Bache, and C. Verdoes, Multithreading and Web Applications: Further Adventures in Acceptance Testing, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 210-213.
35. Bohnet, R. and G. Meszaros. Test-driven porting. in Agile Conference, 2005. Proceedings. 2005.
36. Geras, A., et al., A Survey of Test Notations and Tools for Customer Testing, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 109-117.
37. Holcombe, M. and B. Kalra, Agile Development Environment for Programming and Testing (ADEPT) – Eclipse Makes Project Management eXtreme, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 255-258.
38. Istrate, F. and M. Holcombe, Using State Diagrams to Generate Unit Tests for Object-Oriented Systems, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 214-217.
39. Mugridge, R. and W. Cunningham, Agile Test Composition, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 137-144.
40. Prashant, G., et al. Creating a living specification using FIT documents. in Agile Conference, 2005. Proceedings. 2005.
41. Read, K., G. Melnik, and F. Maurer. Student experiences with executable acceptance testing. in Agile Conference, 2005. Proceedings. 2005.

42. Read, K., G. Melnik, and F. Maurer, Examining Usage Patterns of the FIT Acceptance Testing Framework, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 127-136.
43. Simons, A.H., Testing with Guarantees and the Failure of Regression Testing in eXtreme Programming, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 118-126.
44. Smith, M., et al., E-TDD – Embedded Test Driven Development a Tool for Hardware-Software Co-design Projects, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 145-153.
45. Tappenden, A., et al. Agile security testing of Web-based systems via HTTPUnit. in Agile Conference, 2005. Proceedings. 2005.
46. Tinkham, A. and C. Kaner. Experiences teaching a course in programmer testing. in Agile Conference, 2005. Proceedings. 2005.
47. Yu, J., et al., Agile Testing of Location Based Services, in Extreme Programming and Agile Processes in Software Engineering, H. Baumeister, M. Marchesi, and M. Holcombe, Editors. 2005, Springer Berlin Heidelberg. p. 239-242.
48. Alles, M., et al. Presenter First: organizing complex GUI applications for test-driven development. in Agile Conference, 2006. 2006.
49. Chen, J., et al., Making Fit / FitNesse Appropriate for Biomedical Engineering Research, in Extreme Programming and Agile Processes in Software Engineering, P. Abrahamsson, M. Marchesi, and G. Succi, Editors. 2006, Springer Berlin Heidelberg. p. 186-190.
50. Chih-Wei, H., et al. On agile performance requirements specification and testing. in Agile Conference, 2006. 2006.
51. Holmes, A. and M. Kellogg. Automating functional tests using Selenium. in Agile Conference, 2006. 2006.
52. Melnik, G., F. Maurer, and M. Chiasson. Executable acceptance tests for communicating business requirements: customer perspective. in Agile Conference, 2006. 2006.
53. Müller, M., The Effect of Test-Driven Development on Program Code, in Extreme Programming and Agile Processes in Software Engineering, P. Abrahamsson, M. Marchesi, and G. Succi, Editors. 2006, Springer Berlin Heidelberg. p. 94-103.
54. Puleio, M. How not to do agile testing. in Agile Conference, 2006. 2006.
55. Rust, A., B. Bishop, and K. McDaid, Test-Driven Development: Can It Work for Spreadsheet Engineering?, in Extreme Programming and Agile Processes in Software Engineering, P. Abrahamsson, M. Marchesi, and G. Succi, Editors. 2006, Springer Berlin Heidelberg. p. 209-210.

56. Bouillon, P., et al., EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 101-104.
57. Chubov, I. and D. Droujkov, User Stories and Acceptance Tests as Negotiation Tools in Offshore Software Development, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 167-168.
58. Deng, C., P. Wilson, and F. Maurer, FitClipse: A Fit-Based Eclipse Plug-In for Executable Acceptance Test Driven Development, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 93-100.
59. Dijk, I. and R. Wijnands, Test Driving the Wrong Car, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 250-252.
60. Dobson, J. Performance Testing on an Agile Project. in Agile Conference (AGILE), 2007. 2007.
61. Gagliardi, F., Epistemological Justification of Test Driven Development in Agile Processes, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 253-256.
62. Johnson, P.M. and K. Hongbing. Automated Recognition of Test-Driven Development with Zorro. in Agile Conference (AGILE), 2007. 2007.
63. Lanobile, F. and T. Mallardo, Inspecting Automated Test Code: A Preliminary Study, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 115-122.
64. Melnik, G. and F. Maurer, Multiple Perspectives on Executable Acceptance Test-Driven Development, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 245-249.
65. Mishra, D. and A. Mishra, Adapting Test-Driven Development for Innovative Software Development Project, in Agile Processes in Software Engineering and Extreme Programming, G. Concas, et al., Editors. 2007, Springer Berlin Heidelberg. p. 171-172.
66. Sanchez, J.C., L. Williams, and E.M. Maximilien. On the Sustained Use of a Test-Driven Development Practice at IBM. in Agile Conference (AGILE), 2007. 2007.
67. Sumrell, M. From Waterfall to Agile - How does a QA Team Transition? in Agile Conference (AGILE), 2007. 2007.
68. Wellington, C.A., T.H. Briggs, and C.D. Girard. Experiences Using Automated Tests and Test Driven Development in Computer Science. in Agile Conference (AGILE), 2007. 2007.
69. Cannizzo, F., R. Clutton, and R. Ramesh. Pushing the Boundaries of Testing and Continuous Integration. in Agile, 2008. AGILE '08. Conference. 2008.
70. Carlson, B. An Agile Classroom Experience: Teaching TDD and Refactoring. in Agile, 2008. AGILE '08. Conference. 2008.

71. Geras, A. Leading Manual Test Efforts with Agile Methods. in Agile, 2008. AGILE '08. Conference. 2008.
72. Haugset, B. and G.K. Hanssen. Automated Acceptance Testing: A Literature Review and an Industrial Case Study. in Agile, 2008. AGILE '08. Conference. 2008.
73. Kuhn, A., et al., JExample: Exploiting Dependencies between Tests to Improve Defect Localization, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, et al., Editors. 2008, Springer Berlin Heidelberg. p. 73-82.
74. Marsh, S. and S. Pantazopoulos. Automated Functional Testing on the TransCanada Alberta Gas Accounting Replacement Project. in Agile, 2008. AGILE '08. Conference. 2008.
75. Miller, A. A Hundred Days of Continuous Integration. in Agile, 2008. AGILE '08. Conference. 2008.
76. Mishali, O., Y. Dubinsky, and S. Katz, The TDD-Guide Training and Guidance Tool for Test-Driven Development, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, et al., Editors. 2008, Springer Berlin Heidelberg. p. 63-72.
77. Opelt, K. and T. Beeson. Agile Teams Require Agile QA: How to Make it Work, An Experience Report. in Agile, 2008. AGILE '08. Conference. 2008.
78. Park, S. and F. Maurer, Multi-modal Functional Test Execution, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, et al., Editors. 2008, Springer Berlin Heidelberg. p. 218-219.
79. Rendell, A. Effective and Pragmatic Test Driven Development. in Agile, 2008. AGILE '08. Conference. 2008.
80. Shaye, S.D. Transitioning a Team to Agile Test Methods. in Agile, 2008. AGILE '08. Conference. 2008.
81. Abbattista, F., A. Bianchi, and F. Lanubile, A Storytest-Driven Approach to the Migration of Legacy Systems, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 149-154.
82. Höfer, A. and M. Philipp, An Empirical Study on the TDD Conformance of Novice and Expert Pair Programmers, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 33-42.
83. Kessler, D. and T.J. Andersen. Herding Cats: Managing Large Test Suites. in Agile Conference, 2009. AGILE '09. 2009.
84. Khandkar, S., et al., FitClipse: A Tool for Executable Acceptance Test Driven Development, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 259-260.
85. Kim, E., J. Na, and S. Ryoo, Developing a Test Automation Framework for Agile Development and Testing, in Agile Processes in Software Engineering and Extreme

Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 8-12.

86. Lacoste, F.J. Killing the Gatekeeper: Introducing a Continuous Integration System. in Agile Conference, 2009. AGILE '09. 2009.

87. Marchenko, A., P. Abrahamsson, and T. Ihme, Long-Term Effects of Test-Driven Development A Case Study, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 13-22.

88. McMahon, C. History of a Large Test Automation Project Using Selenium. in Agile Conference, 2009. AGILE '09. 2009.

89. Park, S. and F. Maurer, Communicating Domain Knowledge in Executable Acceptance Test Driven Development, in Agile Processes in Software Engineering and Extreme Programming, P. Abrahamsson, M. Marchesi, and F. Maurer, Editors. 2009, Springer Berlin Heidelberg. p. 23-32.

90. Stolberg, S. Enabling Agile Testing through Continuous Integration. in Agile Conference, 2009. AGILE '09. 2009.

91. Basit, W., F. Lodhi, and U. Bhatti, Extending Refactoring Guidelines to Perform Client and Test Code Adaptation, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 1-13.

92. Besson, F., D. Beder, and M. Chaim, An Automated Approach for Acceptance Web Test Case Modeling and Executing, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 160-165.

93. Brennan, P., Transitioning a Large Organisation: Adopting TDD, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 261-268.

94. Burella, J., et al., Dealing with Navigation and Interaction Requirements Changes in a TDD-Based Web Engineering Approach, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 220-225.

95. Connolly, D., F. Keenan, and F. Caffery, AnnoTestWeb/Run: Annotations Based Acceptance Testing, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 381-382.

96. Erdogan, G., P. Meland, and D. Mathieson, Security Testing in Agile Web Application Development - A Case Study Using the EAST Methodology, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 14-27.

97. Ghanam, Y. and F. Maurer, Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 43-57.

98. Nyman, R., I. Aro, and R. Wagner, Automated Acceptance Testing of High Capacity Network Gateway, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 307-314.

99. Park, S. and F. Maurer, A Literature Review on Story Test Driven Development, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 208-213.
100. Pedroso, B., R. Jacobi, and M. Pimenta, TDD Effects: Are We Measuring the Right Things?, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 393-394.
101. Steinert, B., et al., Continuous Selective Testing, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 132-146.
102. Turhan, B., et al., A Quantitative Comparison of Test-First and Test-Last Code in an Industrial Project, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2010, Springer Berlin Heidelberg. p. 232-237.
103. Ghanam, Y. and F. Maurer. Using Acceptance Tests for Incremental Elicitation of Variability in Requirements: An Observational Study. in Agile Conference (AGILE), 2011. 2011.
104. Haugset, B. and G.K. Hanssen. The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse. in Agile Conference (AGILE), 2011. 2011.
105. Hellmann, T., A. Hosseini-Khayat, and F. Maurer, Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2011, Springer Berlin Heidelberg. p. 223-237.
106. Hellmann, T.D. and F. Maurer. Rule-Based Exploratory Testing of Graphical User Interfaces. in Agile Conference (AGILE), 2011. 2011.
107. Selim, E., et al., A Test-Driven Approach for Extracting Libraries of Reusable Components from Existing Applications, in Agile Processes in Software Engineering and Extreme Programming, A. Sillitti, et al., Editors. 2011, Springer Berlin Heidelberg. p. 238-252.
108. Čaušević, A., D. Sundmark, and S. Punnekkat, Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment, in Agile Processes in Software Engineering and Extreme Programming, C. Wohlin, Editor. 2012, Springer Berlin Heidelberg. p. 138-152.
109. Hellmann, T.D., et al. Agile Testing: Past, Present, and Future -- Charting a Systematic Map of Testing in Agile Software Development. in Agile Conference (AGILE), 2012. 2012.
110. Little, T., et al. Leveraging Global Talent for Effective Test Agility. in Agile Conference (AGILE), 2012. 2012.

Appendix III: Papers from GUI Testing Systematic Mapping Study

1. Mosley, D.J., Client-Server User-Interface Testing. *Ieee Software*, 1995. 12(1): p. 124-126.
2. Ostrand, T., et al., A visual test development environment for GUI systems. *International Symposium on Software Testing and Analysis*, 1998: p. 82-92.
3. Memon, A.M., M.E. Pollack, and M.L. Soffa, Using a goal-driven approach to generate test cases for GUIs. *International Conference on Software Engineering*, 1999: p. 257-266.
4. Memon, A.M., M.E. Pollack, and M.L. Soffa, Automated test oracles for GUIs. *SIGSOFT Software Engineering Notes*, 2000. 25(5): p. 30-39.
5. White, L. and H. Almezen, Generating test cases for GUI responsibilities using complete interaction sequences. *11th International Symposium on Software Reliability Engineering, Proceedings*, 2000: p. 110-121.
6. Donovan, D., et al., Incorporating software reliability engineering into the test process for an extensive GUI-based network management system. *12th International Symposium on Software Reliability Engineering, Proceedings*, 2001: p. 44-53.
7. Memon, A.M., M.E. Pollack, and M.L. Soffa, Hierarchical GUI test case generation using automated planning. *Ieee Transactions on Software Engineering*, 2001. 27(2): p. 144-155.
8. Memon, A.M., M.L. Soffa, and M.E. Pollack, Coverage criteria for GUI testing. *International Symposium on Foundations of Software Engineering*, 2001: p. 256-267.
9. Daboczi, T., et al., How to test graphical user interfaces. *Ieee Instrumentation & Measurement Magazine*, 2003. 6(3): p. 27-33.
10. Memon, A., et al., DART: A framework for regression testing "nightly/daily builds" of GUI applications. *International Conference on Software Maintenance, Proceedings*, 2003: p. 410-419.
11. Memon, A.M., I. Banerjee, and A. Nagarajan, What test oracle should I use for effective GUI testing? *International Conference on Automated Software Testing*, 2003: p. 164-173.
12. Memon, A.M. and M.L. Soffa, Regression testing of GUIs. *International Symposium on Foundations of Software Engineering*, 2003: p. 118-127.
13. Meszaros, G., Agile regression testing using record & playback. *Object-Oriented Systems, Languages, and Applications*, 2003: p. 353-360.
14. Liu, T. and P. Dasieqicz, Incorporating a contract-based test facility to the GUI framework. *Canadian Conference on Electrical and Computer Engineering*, 2004: p. 405-408.
15. Memon, A.M. and Q. Xie, Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for GUI-based software. *20th Ieee International Conference on Software Maintenance, Proceedings*, 2004: p. 8-17.
16. Cai, K.Y., et al., On the test case definition for GUI testing. *QSIC 2005: Fifth International Conference on Quality Software, Proceedings*, 2005: p. 19-26.

17. Limpiyakorn, Y. and P. Kurusathian, Test case prioritization for GUI testing. SERP '05: Proceedings of the 2005 International Conference on Software Engineering Research and Practice, Vols 1 and 2, 2005: p. 338-344.
18. Limpiyakorn, Y. and P. Wongsuttipakorn, GUI test case generation from UML. SERP '05: Proceedings of the 2005 International Conference on Software Engineering Research and Practice, Vols 1 and 2, 2005: p. 323-329.
19. Memon, A., A. Nagarajan, and Q. Xie, Automating regression testing for evolving GUI software. Journal of Software Maintenance and Evolution-Research and Practice, 2005. 17(1): p. 27-64.
20. Memon, A.M. and Q. Xie, Studying the fault-detection effectiveness of GUI test cases for rapidly evolving software. Ieee Transactions on Software Engineering, 2005. 31(10): p. 884-896.
21. Yin, Z.L., et al., Actionable knowledge model for GUI regression testing. 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Proceedings, 2005: p. 165-168.
22. Cai, K.Y., L. Zhao, and F. Wang, A dynamic partitioning approach for GUI testing. 30th Annual International Computer Software and Applications Conference, Vol 2, Short Papers/Workshops/Fast Abstracts/Doctoral Symposium, Proceedings, 2006: p. 223-228.
23. Katara, M., et al., Towards deploying model-based testing with a domain-specific modeling approach. TAIC PART - Testing: Academic & Industrial Conference - Practice and Research Techniques, Proceedings, 2006: p. 81-89.
24. Kervinen, A., M. Maunumaa, and M. Katara, Controlling Testing Using Three-Tier Model Architecture. Electronic Notes in Computer Science, 2006. 164(4): p. 53-66.
25. Memon, A.M., Employing user profiles to test a new version of a GUI component in its context of use. Software Quality Journal, 2006. 14(4): p. 359-377.
26. Xie, Q., Developing cost-effective model-based techniques for GUI testing. International Conference on Software Engineering, 2006: p. 997-1000.
27. Xie, Q. and A.M. Memon, Studying the characteristics of a "good" GUI test suite. Issre 2006:17th International Symposium on Software Reliability Engineering, Proceedings, 2006: p. 159-168.
28. Ye, M., et al., Neural networks based test cases selection strategy for GUI testing. WCICA 2006: Sixth World Congress on Intelligent Control and Automation, Vols 1-12, Conference Proceedings, 2006: p. 5773-5776.
29. Alsmadi, I. and K. Magel, GUI path oriented test generation algorithms. Proceedings of the Second IASTED International Conference on Human-Computer Interaction, 2007: p. 216-219.
30. Feng, L. and S. Zhuang, Action-driven automation test framework for graphical user interface (GUI) software testing. 2007 Ieee Autotestcon, Vols 1 and 2, 2007: p. 22-27.
31. Hayat, M.U. and N. Qadeer, Intra component GUI test case generation technique. Iciet 2007: Proceedings of the International Conference on Information and Emerging Technologies, 2007: p. 154-158.

32. Magel, K. and I. Alsmadi, GUI structural metrics and testability testing. International Conference on Software Engineering and Applications, 2007: p. 91-95.
33. Memon, A.M., An event-flow model of GUI-based applications for testing. Software Testing Verification & Reliability, 2007. 17(3): p. 137-157.
34. Ruiz, A. and Y.W. Price, Test-driven GUI development with testNG and abbot. Ieee Software, 2007. 24(3): p. 51-+.
35. Xie, Q. and A.M. Memon, Designing and comparing automated test oracles for GUI-based software applications. Acm Transactions on Software Engineering and Methodology, 2007. 16(1).
36. Yuan, X. and A.M. Memon, Using GUI run-time state as feedback to generate test cases. ICSE 2007: 29th International Conference on Software Engineering, Proceedings, 2007: p. 396-405.
37. Alsmadi, I., The utilization of user sessions in testing. 7th Ieee/Acis International Conference on Computer and Information Science in Conjunction with 2nd Ieee/Acis International Workshop on E-Activity, Proceedings, 2008: p. 581-585.
38. Alsmadi, I., Building a user interface test automation framework using the data model. PhD Dissertation - North Dakota State University, 2008.
39. Chen, W.K., Z.W. Shen, and C.M. Chang, GUI Test Script Organization with Component Abstraction. Secure System Integration and Reliability Improvement, 2008: p. 128-134.
40. Chen, W.K., Z.W. Shen, and T.H. Tsai, Integration of specification-based and CR-based approaches for GUI testing. Journal of Information Science and Engineering, 2008. 24(5): p. 1293-1307.
41. Ganov, S.R., et al., Test generation for graphical user interfaces based on symbolic execution. International Workshop on Automation of Software Test, 2008: p. 33-40.
42. Hackner, D.R. and A.M. Memon, Test Case Generator for GUITAR. Icse'08 Proceedings of the Thirtieth International Conference on Software Engineering, 2008: p. 959-960.
43. Jaaskelainen, A., A. Kervinen, and M. Katara, Creating a Test Model Library for GUI Testing of Smartphone Applications. Qsic 2008: Proceedings of the Eighth International Conference on Quality Software, 2008: p. 276-282.
44. Kreeger, M.N. and N. Lativ, Abandoning Proprietary Test Tools for Graphical User Interface Verification. Taci Part 2008:Testing: Academic and Industrial Conference Practice and Research Techniques, Proceedings, 2008: p. 52-56.
45. Kwon, O.H. and S.M. Hwang, Mobile GUI testing tool based on Image Flow. 7th Ieee/Acis International Conference on Computer and Information Science in Conjunction with 2nd Ieee/Acis International Workshop on E-Activity, Proceedings, 2008: p. 508-512.
46. Lu, Y., et al., Development of an Improved GUI Automation Test System Based on Event-Flow Graph. International Conference on Computer Science and Software Engineering, 2008: p. 712-715.
47. McMaster, S. and A.M. Memon, Call-stack coverage for GUI test suite reduction. Ieee Transactions on Software Engineering, 2008. 34(1): p. 99-115.

48. Memon, A.M., Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. *Acm Transactions on Software Engineering and Methodology*, 2008. 18(2).
49. Moreira, R.M.L.M. and A.C.R. Paiva, Visual abstract notation for GUI modelling and testing - VAN4GUIM. *Icsoft 2008: Proceedings of the Third International Conference on Software and Data Technologies*, Vol Se/Gsdca/Muse, 2008: p. 104-111.
50. Pichler, J. and R. Ramler, How to Test the Intangible Properties of Graphical User Interfaces? *International Conference on Software Testing, Verification, and Validation*, 2008: p. 494-497.
51. Ruiz, A. and Y.W. Price, GUI Testing Made Easy. *Taci Part 2008:Testing: Academic and Industrial Conference Practice and Research Techniques, Proceedings*, 2008: p. 99-103.
52. Strecker, J. and A.M. Memon, Relationships between Test Suites, Faults, and Fault Detection in GUI Testing. *International Conference on Software Testing, Verification, and Validation*, 2008: p. 12-21.
53. Thornton, M., et al., Supporting Student-Written Tests of GUI Programs. *Sigcse'08: Proceedings of the 39th Acm Technical Symposium on Computer Science Education*, 2008: p. 537-541.
54. Xie, Q. and A.M. Memon, Using a Pilot Study to Derive a GUI Model for Automated Testing. *Acm Transactions on Software Engineering and Methodology*, 2008. 18(2).
55. Xuebao, Q.D., Automated GUI test case generation technology based on coloured Petri nets model. *Journal of Tsinghua University*, 2008. 48(4): p. 600-603.
56. Yuan, X., Feedback-directed model-based gui test case generation. *PhD Dissertation - University of Maryland at College Park*, 2008.
57. Yuan, X. and A.M. Memon, Alternating GUI Test Generation and Execution. *Taci Part 2008:Testing: Academic and Industrial Conference Practice and Research Techniques, Proceedings*, 2008: p. 23-32.
58. Zhu, X.C., et al., A Test Automation Solution on GUI Functional Test. *2008 6th Ieee International Conference on Industrial Informatics*, Vols 1-3, 2008: p. 1344-1349.
59. Barancev, A.V., S.G. Groshev, and V.A. Omelchenko, Generation of test scripts for application with GUI optimized for manual execution. *Central and Eastern European Conference on Software Engineering Conference*, 2009: p. 137-142.
60. Brooks, P., B. Robinson, and A.M. Memon, An Initial Characterization of Industrial Graphical User Interface Systems. *Second International Conference on Software Testing, Verification, and Validation, Proceedings*, 2009: p. 11-20.
61. Budnik, C.J., F. Belli, and A. Hollmann, Structural Feature Extraction for GUI Test Enhancement. *Icstw 2009: Ieee International Conference on Software Testing, Verification, and Validation Workshops*, 2009: p. 255-262.
62. Fu, C., M. Grechanik, and Q. Xie, Inferring Types of References to GUI Objects in Test Scripts. *Second International Conference on Software Testing, Verification, and Validation, Proceedings*, 2009: p. 1-10.

63. Ganov, S., et al., Event Listener Analysis and Symbolic Execution for Testing GUI Applications. *Formal Methods and Software Engineering*, Proceedings, 2009. 5885: p. 69-87.
64. Grechanik, M., Q. Xie, and C. Fu, Experimental Assessment of Manual Versus Tool-Based Maintenance of GUI-Directed Test Scripts. *2009 Ieee International Conference on Software Maintenance*, Conference Proceedings, 2009: p. 9-18.
65. Grechanik, M., Q. Xie, and C. Fu, Maintaining and Evolving GUI-Directed Test Scripts. *2009 31st International Conference on Software Engineering*, Proceedings, 2009: p. 408-418.
66. Jaaskelainen, A., et al., Automatic GUI Test Generation for Smartphone Applications - an Evaluation. *2009 31st International Conference on Software Engineering*, Companion Volume, 2009: p. 112-122.
67. Li, H., et al., An Ontology-based Approach for GUI Testing. *2009 Ieee 33rd International Computer Software and Applications Conference*, Vols 1 and 2, 2009: p. 626-627.
68. Li, L., et al., A method for combinatorial explosion avoidance of AI planner and the application on test case generation. *International Conference on Computational Intelligence and Software Engineering*, 2009: p. 1-4.
69. McMaster, S. and A.M. Memon, An Extensible Heuristic-Based Framework for GUI Test Case Maintenance. *Icstw 2009: Ieee International Conference on Software Testing, Verification, and Validation Workshops*, 2009: p. 251-254.
70. Qian, S.Y. and F. Jiang, An Event Interaction Structure for GUI Test Case Generation. *2009 2nd Ieee International Conference on Computer Science and Information Technology*, Vol 2, 2009: p. 619-622.
71. Tong, X., et al., Study on generation strategies of GUI automation testing cases. *Systems Engineering and Electronics*, 2009. 31(1): p. 174-177.
72. Xing, X. and F. Jiang, GUI Test Case Definition with TTCN-3. *Computational Intelligence and Engineering*, 2009: p. 1-5.
73. Yuan, X., M.B. Cohen, and A.M. Memon, Towards Dynamic Adaptive Automated Test Generation for Graphical User Interfaces. *Icstw 2009: Ieee International Conference on Software Testing, Verification, and Validation Workshops*, 2009: p. 263-266.
74. Alsmadi, I., Using Genetic Algorithms for Test Case Generation and Selection Optimization. *2010 23rd Canadian Conference on Electrical and Computer Engineering (Ccece)*, 2010.
75. Bertolini, C. and A. Mota, A Framework for GUI Testing Based on Use Case Design. *Software Testing, Verification, and Validation Workshops*, 2010: p. 252-259.
76. Chang, T.H., T. Yeh, and R.C. Miller, GUI Testing Using Computer Vision. *Chi2010: Proceedings of the 28th Annual Chi Conference on Human Factors in Computing Systems*, Vols 1-4, 2010: p. 1535-1544.
77. Chen, W.K. and Z.W. Shen, GUI test-case generation with macro-event contracts. *International Conference on Software Engineering and Data Mining*, 2010: p. 145-151.
78. Elaska, E., et al., Using Methods & Measures from Network Analysis for GUI Testing. *International Conference on Software Testing, Verification, and Validation*, 2010: p. 240-246.

79. Hellmann, T.D., A. Hosseini-Khayat, and F. Maurer, Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. Software Testing, Verification, and Validation Workshops, 2010: p. 444-447.
80. Huang, C.Y., J.R. Chang, and Y.H. Chang, Design and analysis of GUI test-case prioritization using weight-based methods. Journal of Systems and Software, 2010. 83(4): p. 646-659.
81. Huang, S., M.B. Cohen, and A.M. Memon, Repairing GUI Test Suites Using a Genetic Algorithm. International Conference on Software Testing, Verification, and Validation, 2010: p. 245-254.
82. Jovic, M., et al., Automating performance testing of interactive Java applications. Automation of Software Test, 2010: p. 8-15.
83. Li, L., et al., Improved AI planner and the application on GUI test case generation of military software. Journal of PLA University of Science and Technology, 2010. 11(3): p. 267-273.
84. Miao, Y. and X.B. Yang, An FSM based GUI Test Automation Model. 11th International Conference on Control, Automation, Robotics and Vision (Icarcv 2010), 2010: p. 120-126.
85. Navarro, P.L.M., G.M. Perez, and D.S. Ruiz, Open HMI-Tester: An open and cross-platform architecture for GUI testing and certification. Computer Systems, Science, and Engineering, 2010: p. 283-296.
86. Navarro, P.L.M., D.S. Ruiz, and G.M. Perez, A proposal for automatic testing of GUIs based on annotated use cases. Advances in Software Engineering, 2010: p. 1-13.
87. Nguyen, D.H., P. Strooper, and J.G. Suess, Model-based testing of multiple GUI variants using the GUI test generator. Workshop on Automation of Software Test, 2010: p. 24-30.
88. Rauf, A., et al., Automated GUI Test Coverage Analysis Using GA. International Conference on Information Technologies: New Generations, 2010: p. 1057-1062.
89. Rauf, A., et al., Evolutionary Based Automated Coverage Analysis for GUI Testing. Contemporary Computing, Pt 1, 2010. 94: p. 456-466.
90. Rauf, A., et al., Ontology driven semantic annotation based GUI testing. International Conference on Emerging Technologies, 2010: p. 261-264.
91. Shewchuk, Y. and V. Grarousi, Experience with maintenance of a functional GUI test suite using IBM Rational Functional Tester. International Conference on Software Engineering and Knowledge Engineering, 2010: p. 489-494.
92. Sinnig, D., F. Khendek, and P. Chalin, A Formal Model for Generating Integrated Functional and User Interface Test Cases. International Conference on Software Testing, Verification, and Validation, 2010: p. 255-264.
93. Stanbridge, C., Retrospective Requirement Analysis Using Code Coverage of GUI Driven System Tests. International Requirements Engineering Conference, 2010: p. 411-412.
94. Yuan, X. and A.M. Memon, Generating Event Sequence-Based Test Cases Using GUI Runtime State Feedback. Ieee Transactions on Software Engineering, 2010. 36(1): p. 81-95.

95. Yuan, X. and A.M. Memon, Iterative execution-feedback model-directed GUI testing. *Information and Software Technology*, 2010. 52(5): p. 559-575.
96. Zhao, L. and K.Y. Cai, On Modeling of GUI Test Profile. International Conference on Software Testing, Verification, and Validation, 2010: p. 260-264.
97. Adamoli, A., et al., Automated GUI performance testing. *Software Quality Journal*, 2011. 19(4): p. 801-839.
98. Aho, P., N. Menz, and T. Raty, Enhancing generated Java GUI models with valid test data. IEEE Conference on Open Systems, 2011: p. 310-315.
99. Alsmadi, I., Activities and Trends in Testing Graphical User Interfaces Automatically. *Journal of Software Engineering*, 2011. 5(1): p. 1-19.
100. Appsami, G., V.M. Raj, and M. Zubair, Automated testing of Silverlight and Moonlight applications. *Innovations in Emerging Technology*, 2011(49-54).
101. Arlt, S., C. Bertolini, and M. Schaf, Behind the Scenes: An Approach to Incorporate Context in GUI Test Case Generation. *Software Testing, Verification, and Validation Workshops*, 2011: p. 222-231.
102. Bauersfeld, S., S. Wappler, and J. Wegener, A Metaheuristic Approach to Test Sequence Generation for Applications with a GUI. *Search Based Software Engineering*, 2011. 6956: p. 173-187.
103. Bryce, R.C., S. Sampath, and A.M. Memon, Developing a Single Model and Test Prioritization Strategies for Event-Driven Software. *Ieee Transactions on Software Engineering*, 2011. 37(1): p. 48-64.
104. Daniel, B., et al., Automated GUI refactoring and test script repair. *Workshop on End-to-End Test Script Engineering*, 2011: p. 38-41.
105. Grove, R. and J. Faytong, Identifying Infeasible GUI Test Cases Using Support Vector Machines and Induced Grammars. *Software Testing, Verification, and Validation Workshops*, 2011: p. 202-211.
106. Hauptmann, B. and M. Junker, Utilizing user interface models for automated instantiation and execution of system tests. *Workshop on End-to-End Test Script Engineering*, 2011: p. 8-15.
107. Hellmann, T.D., A. Hosseini-Khayat, and F. Maurer, Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation. *Agile Processes in Software Engineering and Extreme Programming*, 2011. 77: p. 223-237.
108. Hu, C. and I. Neamtiu, Automating GUI testing for android applications. *International Workshop on Automation of Software Test*, 2011: p. 77-83.
109. Lai, J., H. Zhang, and B. Huang, The object-FMA based test case generation approach for GUI software exception testing. *International Conference on Reliability, Maintainability, and Safety*, 2011: p. 717-723.
110. Navarro, P.L.M., G.M. Perez, and D.S. Ruiz, Towards Software Quality and User Satisfaction through User Interfaces. *International Conference on Software Testing, Verification, and Validation*, 2011: p. 415-418.

111. Pajunen, T., T. Takala, and M. Katara, Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework. International Conference on Software Testing, Verification, and Validation, 2011: p. 242-251.
112. Rauf, A., A. Jaffar, and A.A. Shahid, Fully Automated Gui Testing and Coverage Analysis Using Genetic Algorithms. International Journal of Innovative Computing Information and Control, 2011. 7(6): p. 3281-3294.
113. Sadeh, B., et al., Towards Unit Testing of User Interface Code for Android Mobile Applications. Software Engineering and Computer Systems, Pt 3, 2011. 181: p. 163-175.
114. Takala, T., M. Katara, and J. Harty, Experiences of System-Level Model-Based GUI Testing of an Android Application. International Conference on Software Testing, Verification, and Validation, 2011: p. 377-386.
115. Usaniov, A. and K. Motiejunas, A Method for Automated Testing of Software Interface. Information Technology and Control, 2011. 40(2): p. 99-109.
116. Wu, Y.M. and Z.F. Liu, Analysis of OCR Design and Implementation for GUI Modeling. Measuring Technology and Mechatronics Automation Iv, Pts 1 and 2, 2012. 128-129: p. 1303-1307.

Appendix IV: Publications During This Degree

- Gabriela Jurca, **Theodore D. Hellmann**, Frank Maurer: Integrating Agile and User-Centered Design: A Systematic Mapping and Review of Evaluation and Validation Studies of Agile-UX. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2014), Orlando, FL, 2014.
- **Theodore D. Hellmann**, Apoorve Chokshi, Zahra Shakeri Hossein Abad, Sydney Pratte, Frank Maurer: Agile Testing: A Systematic Mapping Across Three Conferences: Understanding Agile Testing in the XP/Agile Universe, Agile, and XP Conferences. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2013), Nashville, TN, 2013.
- Chris Burns, Teddy Seyed, **Theodore D. Hellmann**, Mario Costa Sousa, Frank Maurer, "A Usable API for Multi-Surface Systems". In Proceedings of the Workshop on Envisioning Future Collaborative Interactive Spaces (BLEND'13). Paris, France, 2013.
- Frank Maurer, **Theodore D. Hellmann**, "People-Centered Software Development: An Overview of Agile Methodologies" in Software Engineering, International Summer Schools, ISSSE 2009-2011. Andrea De Lucia and Filomena Ferrucci, Eds. Salerno, Italy: Springer, 2013.
- Tiago Silva da Silva, Milene Selbach Silveira, Frank Maurer, **Theodore D. Hellmann**: User Experience Design and Agile Development: From Theory to Practice. Journal of Software Engineering and Applications, Vol.5 No.10, October 2012, 743-751.
- **Theodore D. Hellmann**, Abhishek Sharma, Jennifer Ferreira, Frank Maurer: Agile Testing: Past, Present, and Future. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2012), Dallas, TX, 2012.
- Abhishek Sharma, **Theodore D. Hellmann**, Frank Maurer: Testing of Web Services – A Systematic Mapping. Proceedings of the IEEE World Congress on Services (SERVICES 2012), Honolulu, Hawaii, 2012.
- Chris Burns, Teddy Seyed, **Theodore D. Hellmann**, Jennifer Ferreira, Frank Maurer: Towards a Usable API for Constructing Interactive Multi-Surface Systems. Proceedings of the Workshop on Infrastructure and Design Challenges of Coupled Display Visual Interfaces (PPD 2012), Capri, Italy, 2012.
- **Theodore D. Hellmann**, Frank Maurer: Rule-Based Exploratory Testing of Graphical User Interfaces. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2011), Salt Lake City, UT, 2011. (Full Paper)
- **Theodore D. Hellmann**, Ali Hosseini-Khayat, Frank Maurer, Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation. Proc. of 12th International Conference on Agile Processes and eXtreme Programming (XP 2011), Madrid, Spain, 2011.

- Ali Hosseini-Khayat, **Theodore D. Hellmann**, and Frank Maurer: Distributed and Automated Usability Testing of Low-Fidelity Prototypes. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2010), Orlando, FL, 2010.
- **Theodore D. Hellmann**, Ali Hosseini-Khayat, and Frank Maurer, Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. 1st International Workshop on Test-driven Development (TDD 2010). Paris, France, 2010.
- **Theodore D. Hellmann**, Ali Hosseini-Khayat, and Frank Maurer, "Agile Interaction Design and Test-Driven Development of User Interfaces - A Literature Review," in Agile Software Development: Current Research and Future Directions, Torgeir Dingsøyr, Tore Dybå, and Nils Brede Moe, Eds. Trondheim, Norway: Springer, 2010, ch. 9.

Appendix V: Copyright Consent



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Abhishek Sharma, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Abhishek Sharma, Jennifer Ferreira, Frank Maurer: Agile Testing: Past, Present, and Future. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2012). Dallas, TX, 2012.
- Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer: An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices. Technical Report. University of Calgary: Calgary (2014). Report Number 2014-1057-08.

Signature: _____

Date: April 14, 2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Ali Hosseini-Khayat, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Ali Hosseini-Khayat, Frank Maurer, Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation. Proc. of 12th International Conference on Agile Processes and eXtreme Programming (XP 2011), Madrid, Spain, 2011.
- Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer, Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. 1st International Workshop on Test-driven Development (TDD 2010). Paris, France, 2010.

Signature: 

Date: 4/11/2015



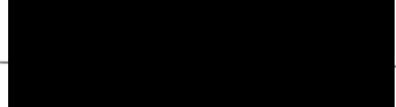
Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Apoorve Chokshi, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Apoorve Chokshi, Zahra Shakeri Hossein Abad, Sydney Pratte, Frank Maurer: *Agile Testing: A Systematic Mapping Across Three Conferences: Understanding Agile Testing in the XP/Agile Universe, Agile, and XP Conferences*. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2013), Nashville, TN, 2013.

Signature: _____



Date: April 10, 2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Elham Moazzen, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer: An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices. Technical Report. University of Calgary: Calgary (2014). Report Number 2014-1057-08.

Signature: _____

Date: 14/10/11/2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Frank Maurer, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer: An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices. Technical Report. University of Calgary: Calgary (2014). Report Number 2014-1057-08.
- Theodore D. Hellmann, Apoorve Chokshi, Zahra Shakeri Hossein Abad, Sydney Pratte, Frank Maurer: Agile Testing: A Systematic Mapping Across Three Conferences: Understanding Agile Testing in the XP/Agile Universe, Agile, and XP Conferences. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2013), Nashville, TN, 2013.
- Frank Maurer, Theodore D. Hellmann, "People-Centered Software Development: An Overview of Agile Methodologies" in Software Engineering, International Summer Schools, ISSSE 2009-2011. Andrea De Lucia and Filomena Ferrucci, Eds. Salerno, Italy: Springer, 2013.
- Theodore D. Hellmann, Abhishek Sharma, Jennifer Ferreira, Frank Maurer: Agile Testing: Past, Present, and Future. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2012), Dallas, TX, 2012.
- Theodore D. Hellmann, Ali Hosseini-Khayat, Frank Maurer, Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation. Proc. of 12th International Conference on Agile Processes and eXtreme Programming (XP 2011), Madrid, Spain, 2011.
- Theodore D. Hellmann, Ali Hosseini-Khayat, and Frank Maurer, Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. 1st International Workshop on Test-driven Development (TDD 2010). Paris, France, 2010.

Signature: _____

Date: 10-4-2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Jennifer Ferreira, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Abhishek Sharma, Jennifer Ferreira, Frank Maurer: Agile Testing: Past, Present, and Future. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2012), Dallas, TX, 2012.

Signature: _____ Date: April 13, 2015

U



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Jonathan Sillito, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer: An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices. Technical Report. University of Calgary: Calgary (2014). Report Number 2014-1057-08.

Signature: _____



Date: 20 Apr 2015



UNIVERSITY OF
CALGARY

Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Sydney Pratte, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Apoorve Chokshi, Zahra Shakeri Hossein Abad, Sydney Pratte, Frank Maurer: *Agile Testing: A Systematic Mapping Across Three Conferences: Understanding Agile Testing in the XP/Agile Universe, Agile, and XP Conferences*. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2013), Nashville, TN, 2013.

Signature: _____

Date: April 9, 2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Md. Zabedul Akbar, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Elham Moazzen, Abhishek Sharma, Md. Zabedul Akbar, Jonathan Sillito, Frank Maurer: An Exploratory Study of Automated GUI Testing: Goals, Issues, and Best Practices. Technical Report. University of Calgary: Calgary (2014). Report Number 2014-1057-08.

Signature: 

Date: April 13, 2015



Co-Author Consent Form

The University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4

I, Zahra Shakeri Hossein Abad, grant Theodore D. Hellmann permission to use the content of the following co-authored publications in his PhD dissertation and to have this work microfilmed.

- Theodore D. Hellmann, Apoorve Chokshi, Zahra Shakeri Hossein Abad, Sydney Pratte, Frank Maurer: *Agile Testing: A Systematic Mapping Across Three Conferences: Understanding Agile Testing in the XP/Agile Universe, Agile, and XP Conferences*. Proceedings of the International Conference on Agile Methods in Software Development (Agile 2013), Nashville, TN, 2013.

Signature: _____

Date: 2015 Apr 10