Test-Driven Development of Graphical User Interfaces: A Pilot Evaluation

Theodore D. Hellmann, Ali Hosseini-Khayat, Frank Maurer

The University of Calgary, Department of Computer Science,
2500 University Drive NW, Calgary, Alberta, Canada
{tdhellma, hosseisa, fmaurer}@ucalgary.ca

Abstract. This paper presents a technique for test-driven development of GUI-based applications, as well as a pilot evaluation. In our approach, user interface prototypes are created in such a way as to allow capture/replay tools to record interactions with them. These recordings can then be replayed on the actual GUI as it is being developed in a test-driven fashion. The pilot evaluation found that developers integrated GUI tests, based on user interface prototypes, into their development process and used them as a way to determine when a feature is actually complete. Study participants felt that TDD of GUI based applications is useful.

Keywords: Interaction design, test-driven development, user interface testing, graphical user interface

1 Introduction

Test-driven development (TDD) has proven benefits for the software engineering process in terms of increased developer confidence and increased software quality without a significant decrease in productivity [1] [2]. Over time, tests created through TDD form a regression suite that will quickly notify developers when changes to a part of the application are causing tests for any part of the application to fail. However, if tests in the regression suite are broken by changes to the underlying system that do not actually impact the application's functionality, this safety net can become a liability, with developers dismissing failing tests as "the test's fault" [3]. Unfortunately, automated GUI test suites are likely to fail erroneously, making TDD of GUI-based applications (UITDD) something of a double-edged sword [4] [3].

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 the main barrier to the uptake of UITDD. Currently, developers wishing to perform test-driven GUI development must write complex test code manually. This is especially difficult given the tight coupling between GUI tests and GUI code. In order to test a widget, automated tests need to 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. This makes GUI tests likely to break in response to changes in the system under test, even when these changes do not actually impact the functionality of the system. This is because changes to any of the characteristics used to identify widgets could prevent test code from finding the correct widget.

In order to simplify the process of UITDD, we have developed an approach that leverages the advantages of usability evaluation and user interface prototyping. By first creating a prototype of the GUI, it is possible to perform iterative usability evaluations to identify usability flaws early in the development process, before effort has been expended in development of the actual GUI. Capture/replay tools (CRTs) – tools that record users' interactions with an application – can then be used to record test scripts. These scripts can then be run against the actual GUI as it is developed, as described in Section 4.

To begin evaluating our approach, it is necessary to answer the following question: does having tests for a GUI written before the functionality of the GUI is implemented help developers? In order to investigate this question, we conducted a pilot evaluation in which a small number of participants were provided with a variety of support options, including a GUI test suite, and asked to develop the missing glue code in a sample application. This paper presents results from this pilot study, including both observations made by researchers during study sessions and participant responses to a post-survey questionnaire.

2 What Makes GUI Testing and UITDD Difficult?

In order understand this problem it is first necessary to understand the difficulties with GUI testing. GUI tests are unusually difficult to write and maintain for a variety of reasons. First, even simple GUIs are very complicated – both from the perspective of design and from the perspective of testing. Second, a GUI test should only fail when meaningfully problematic situations are encountered, but creating such GUI tests is difficult. Third, an application's GUI is likely to change throughout the course of development, which can result in tests failing despite the fact that the features of an application are still functioning. Each of these areas is described in more detail in the following subsections. Finally, the central issue that UITDD must overcome is addressed: how can we create GUI tests easily before the GUI itself exists?

2.1 Complexity

Modern GUIs are composed of a set of user interface elements, or *widgets*. Like most classes in object-oriented systems, widgets tend to have a long chain of inheritances. For example, in the Windows Presentation Framework (WPF), the chain of inheritance between Button and the base Object class contains 8 other classes. The Button class is composed of 136 properties and 211 methods, and can fire 111 events in response to various interactions. This means that even a simple GUI has a huge number of possible user interactions, GUI states, and events. Table 1 provides an overview of the number of widgets, properties, methods, and events involved in the simple calculator application shown in Fig. 1.

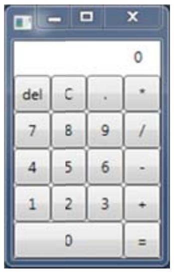


Fig. 1. A calculator application

Table 1. Properties, methods, and events of user-defined widgets in application shown in Fig. 1.

Widgets	19
Properties	2577
Methods	4007
Events	2108

This complexity means that the number of possible sequences of events grows exponentially with the number of steps in a test script [12]. The bug-finding potential of a GUI test suite is determined by the number of events that are triggered and the number of states from which each event is triggered [13]. This means that there is a large amount of testing that can be done, rendering it impractical to perform comprehensive GUI testing.

2.2 Verification

A test is separated into two parts: a *test procedure* and a *test oracle*. The test procedure interacts with part or all of an application in order to generate a state that is interesting from a testing perspective. The test oracle then verifies that the system behaves as expected in response to this interaction. The effectiveness of automated tests is directly limited by the difficulty of writing useful test oracles [14]. In GUI testing, this is compounded by the fact that the more values a GUI test oracle is verifying, the higher its chances of detecting bugs [15].

An alternative to this approach would be to make use of manual testing – either scripted manual testing, in which test cases are written down for humans to perform, or exploratory testing, where humans use their knowledge and intuition to search for bugs. Since manual testing relies on human intelligence to determine whether a GUI is functioning, it avoids some of the issues associated with automated GUI testing. However, scripted manual testing takes a significant amount of time and effort to perform and exploratory testing cannot be performed in a test-driven fashion. In our approach, described in Section 4, exploratory testing combined with a CRT is used as a means to identify important paths through ExpenseManager in order to focus automated testing effort on these “interesting” subsystems.

2.3 Finding Widgets from Test Code

In GUI testing, it is necessary to look up widgets when a test is run based on information about the widgets that is recorded when a test is created. This process of looking up widgets is in large part responsible for the fragility of GUI tests. When widgets in a GUI change in ways that lead to this search missing them or returning an incorrect widget, then not only will tests of this widget break, but the resulting error can be difficult to understand. For example, in Fig. 1, if the text displayed on the widget labeled “del” is important to its proper identification, and that text is changed to “delete,” this change could lead to a misidentification.

There are two ways of performing this search: *testing with object maps* and *keyword-based testing*. In the former, as much information as possible about a widget is included in the test so that a heuristic search for the desired element can be made when the test is run. Because of this, it is possible for information about widgets to change without breaking tests that rely on them. In the latter, a single, unique identifier is assigned to each widget, meaning that only this identifier needs to remain unchanged in order to a test to locate the same widget.

Even when these search methods are used, it’s possible for up to 74% of test cases to be broken after modifications to an application’s GUI [16] [17]. This means that most of the tests in a suite must be either fixed or recreated from scratch, which represents a significant amount of rework. The stability of a GUI is a key problem that must be addressed before UITDD becomes practical. In Section 4, we describe an approach to TDD which makes heavy use of usability evaluation in order to minimize the changes necessary to a GUI after development on the actual interface has begun.

2.4 Creating GUI Tests without a GUI

An essential requirement of TDD is that it needs to be possible to write tests before the feature being tested is implemented. In this way, developers are able to begin thinking about the design of their code earlier and avoid making mistakes in the first place. However, writing GUI tests is different from writing any other kind of test. Because of the complexity of looking up widgets, interacting with them, and verifying their behavior, GUI tests are difficult to write by hand since detailed knowledge about the implementation of the GUI is necessary. Commonly, CRTs are used to create GUI tests. But, in order to use a CRT, a GUI must already exist and be functioning. Because of this, two alternatives exist for the creation of GUI tests without a GUI: GUI tests can be written by hand; or a way can be found to make CRTs work before a GUI is available.

Several tools exist that can simplify this process of manually writing GUI tests [18] [19]. However, creating GUI tests manually remains technically challenging. In our previous work, as described in Section 4, we have devised a method of instrumenting GUI prototypes. CRTs can then be used to record interactions with these prototypes so that tests can be rerun against the actual GUI rather than its prototypes. This makes it possible to use the recorded tests to develop the GUI in a TDD fashion.

3 Related Work

Approaches to overcoming issues described in the previous section have been proposed and evaluated. This section provides an overview of publications directly related to UITDD, but not those that bypass the GUI, including those related to the use of Model-View-Controller [20] and test harnesses [21].

Despite the wide use of design patterns like Model-View-Controller [20] to make systems more testable below the level of the user interface, GUI testing remains important given the fact that bugs originating from errors in the code of user interfaces can seriously impair the functionality available to a user. Two studies carried out on systems developed by ABB Corporate Research have shown that the number of defects originating from errors in GUI code can represent 9% of the bugs found in an application through in-house testing, but represent 60% of the bugs found in an application by customers after the release of a product [22] [23]. Of these customer-reported defects, 65% were categorized as non-cosmetic, and resulted in a loss of functionality to the user, such as inability to print from a given screen or a system crash. The amount of non-trivial bugs stemming from GUI code makes sense in light of the fact that a significant portion of an application's code – 45-60% – can be required for the creation and management of its GUI [24].

Work by Ruiz and Price has resulted in brief evaluations of the use of two approaches to test-driven GUI development: using TestNG-Abbot [18] and FEST (Fixtures for Easy Software Testing) [19]. Both of these systems provide support for manually writing programmatic tests for GUI functionality before coding the corresponding GUI. This approach requires test authors to speculate about the future implementation of the GUI, which means that once it is actually implemented, tests may need to be reconciled with this implementation. Because writing tests manually represents a significant expenditure of effort even with the support of tools like FEST, the reconciliation process may be expensive. Because this approach requires test authors to understand the technical details of a system under test, it makes it difficult for customers to understand the tests they are helping to define.

Another approach uses GTT (GUI Testing Tool) to define tests through an abstract, high-level specification language [25]. GTT combines a specification editor, which can be used to write tests before a GUI is developed, with a CRT, which can be used to create or edit tests based on interactions with an existing GUI. The main benefit to this approach is that tests defined manually or through the CRT are recorded in the same high-level language, which simplifies test maintenance after a GUI has been created. However, tests created in this fashion are difficult to understand and require intimate knowledge of the system under development.

Selenium is a web testing tool rather than a general GUI testing tool, but it has been applied to the testing of web-based systems in a test-driven fashion [3]. Selenium's keyword-based system of identification makes it easier to create and understand tests. However, the authors of the study found that tests would frequently fail even after corresponding GUI functionality had been implemented due to minor issues, such as errors identifying the desired widget or timing issues. Because of this, the team had to revise the role of tests in its process. Tests were still written before a feature was implemented for specification purposes, but these tests were generally discarded after a feature was implemented. Instead, new tests were recorded using Selenium's CRT and added to the regression suite. While this helped reduce the time spent fixing broken tests, it also removed a major benefit of TDD of GUIs: identifying when a feature is complete.

4 A Prototype-Driven Approach to UITDD

One of the main barriers to UITDD identified in the previous sections is the difficulty of using CRTs. CRTs can only be used when a GUI already exists. To address this, we propose that a prototype of the GUI be created with sufficient detail for CRTs to record tests based on interactions with this prototype. Our previous work suggests that it is possible to use tests recorded from prototypes to verify the functionality of the actual GUI as it is developed [5]. In this section, an example will be presented to describe this process. This example will be used as a basis for the pilot evaluation conducted in Section 5.

4.1 ExpenseManager

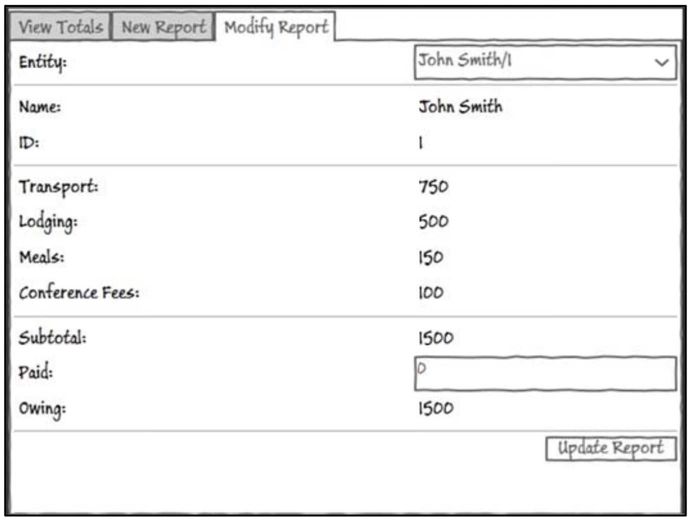
ExpenseManager is a GUI-based application for the creation, saving, modification, and totaling of expense reports, similar to those required by many organizations before they will reimburse employees for travel expenses. ExpenseManager's GUI contains three tabs: *View Totals*; *New Report*; and *Modify Report*. The View Totals tab allows users to view the total expense incurred by, paid to, and outstanding reimbursements 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.

4.2 Prototyping ExpenseManager

After defining the features of ExpenseManager, a prototype of the application was created. This prototype was created using SketchFlow [6]. Prototypes are created by dragging-and-dropping widgets onto a canvas. Each canvas can be used to represent a different state of the system. Widgets can also be assigned behaviors such as transitioning to a different state. The resulting prototypes are executable, allowing test users to interact with them as though they were functional applications.

This prototype can be utilized for the purpose of usability evaluation [7] [8], specifically Wizard of Oz-like tests [9] [10] [8], 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 majority of usability flaws are resolved and user satisfaction reaches an acceptable level.

A prototype of all of the functionality of ExpenseManager described in Section 4.1 was created using SketchFlow. One of the states of the SketchFlow prototype of



Entity:	John Smith/1
Name:	John Smith
ID:	1
Transport:	750
Lodging:	500
Meals:	150
Conference Fees:	100
Subtotal:	1500
Paid:	0
Owing:	1500

Fig. 2. One page from the SketchFlow prototype of ExpenseManager

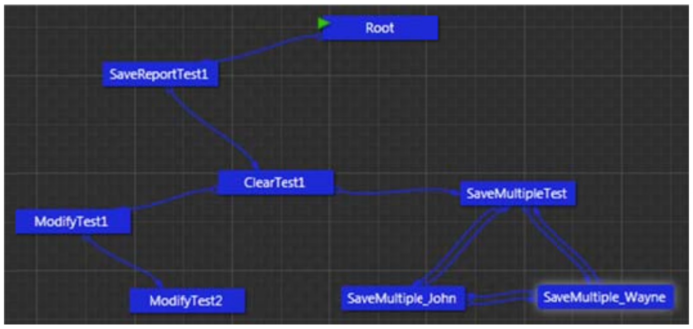


Fig. 3. State map of ExpenseManager prototype

ExpenseManager can be seen in Fig. 2, while the map of states and possible transitions involved in the prototype can be seen in Fig. 3.

4.3 Creating Tests for ExpenseManager

SketchFlow prototypes are WPF user interfaces and trigger events that can be detected through the Windows Automation API. This allows CRTs based on the Automation API to record events triggered by users' interactions with these prototypes. In this evaluation, we used LEET (LEET Enhances Exploratory Testing) to record events from interactions with prototypes of ExpenseManager [11]. LEET is based on the Automation API, and uses keyword-based identification to determine which widget to interact with for each step in the test script. This means that LEET will search for a widget with a matching keyword and then perform some action with it. Because of this, it is possible to run a test recorded from a prototype of ExpenseManager on the actual GUI as long as widgets are given the same keyword in the actual GUI as they are in the prototype. When these tests are first run against the actual GUI, they will fail – as they should in TDD – and will only pass when the actual GUI implements the same functionality expressed in the prototype. Using this approach, it is possible to go through the normal TDD cycle process of writing tests, running them, writing code, and refactoring.

For the development of ExpenseManager, four tests were recorded from the prototype verifying the following functionality: reports do not get totaled under the View Totals tab until they are saved; saved tests are still totaled under the View Totals tab even when they have been cleared from the New Report tab; saved reports modified from the Modify Report tab are also updated on the View Totals tab; and 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 – in other words, the tests will only be as complete as the prototypes.

5 Pilot Evaluation

We designed a pilot evaluation to determine whether our approach is actually useful to developers, given a variety of tools to assist them in development of functionality in a GUI-based application. Our results are based on the experiences of three participants recruited from the Calgary Agile Methods User Group (CAMUG) [26].

Table 3 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 participants did not have significant experience with the prerequisite techniques of GUI testing, user interface prototyping, and TDD. Despite this, participants were able to adapt to the process of UITDD.

Table 2. 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

5.1 Creating the Test System

First, researchers completed the development of ExpenseManager, including its GUI and event handlers, based on the tests created in Section 4. This was done to ensure that the tests provided to participants were adequate for the development of ExpenseManager. This entire process, from prototypes through to passing tests, took five hours to complete. In all, ExpenseManager's GUI is composed of 50 visible widgets, not counting those associated with the base Window object and Separators: 1 TabControl, 3 TabItems, 2 ComboBoxes, 8 TextBoxes, and 36 Labels. This version included all of the functionality described above, including a complete GUI, complete event handlers, and a complete data structure for storing reports.

Then, the researchers removed all of the code contained in ExpenseManager's 7 event handlers – the "glue code" that connects its GUI with its data structures. This means that ExpenseManager was left with a complete but nonfunctional GUI, a constructor, a data structure, and stubs for event handlers. Participants were asked to complete the implementation of this version of ExpenseManager, which requires only that they implement these missing event handlers. This simplification decreased the amount of time that would be required of participants as well as focused the pilot evaluation on the usefulness of the previously-created GUI tests to participants' development.

Participants were given one hour to complete this task and provided with several tools to aid them. First, participants were given the user interface prototype and the GUI tests created in Section 4, as well as access to Visual Studio 2010's debugger and GUI builder and the ability to ask the researcher for technical clarifications.

5.2 Observations Collected During the Study

Interestingly, 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 story, participants would use the GUI builder to identify the next widget involved in the story. From the GUI builder, they would then navigate to a related event handler in the code-behind 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 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 Fig. 4.

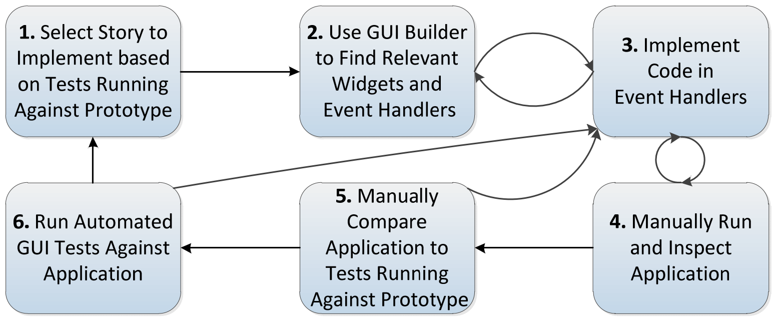

1. Select Story to Implement based on Tests Running Against Prototype
2. Use GUI Builder to Find Relevant Widgets and Event Handlers
3. Implement Code in Event Handlers
4. Manually Run and Inspect Application
5. Manually Compare Application to Tests Running Against Prototype
6. Run Automated GUI Tests Against Application

Fig. 4. Workflow for development of GUI-based application observed during studies

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 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.

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 3.

Table 3. Passing Tests by Participant

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

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.

5.4 Observations Collected from Post-Study Surveys

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 this approach.

Table 4. 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

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 4. 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 5, 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 Fig. 4. 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.

Table 5. 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

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 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. A technique needs to be used to make it visually explicit that LEET is interacting with a specific widget as a test is running. These suggestions will help steer the development of LEET towards a more usable application.

In conclusion, this pilot evaluation suggests that the approach to UITDD proposed in this paper is useful for development of GUI-based applications. GUI tests were used primarily for determining when a feature was done and participants used tests as a roadmap for the completion of the application. Participants also reported through the post-experiment survey that they felt GUI tests were useful. Other major contributions of this pilot evaluation include a model describing the way developers use GUI tests, suggestions for improvement of GUI testing tools, and suggestions for improvement of this approach to UITDD.

5.5 Study Limitations

First, only three participants took part in this study. A larger subject pool would mean that opinions could be gathered from a wider group of participants with different backgrounds, which would have allowed us to draw more interesting conclusions. As it stands, this experiment can be regarded as a successful pilot evaluation that encourages setting up a more comprehensive.

Second, the time allotted for each study session was limited to one hour in order to avoid inconveniencing participants. However, 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.

Third, the application used for this study was, based on Table 2, too complicated for a one-hour study. Participants were unable to complete the implementation of the system in just one hour. For future controlled experiments, it would be preferable to have participants develop a simpler system. 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.

6 Future Work

In addition to making the improvements to LEET identified in the previous section, there are several directions that could be taken to improve this approach to UITDD. First, this study should be extended to include additional participants or, ideally, case studies of long-term projects with industrial partners in order to increase the reliability and real-world applicability of its findings.

Second, LEET is also being used to support rule-based testing of GUI-based applications. In rule-based testing, short, reusable rules are defined that relate to behavior the system being developed should always (or never) exhibit. These rules are then executed after every step of a test script in order to insure that the system being developed conforms to each rule in every state accessed during the test. Due to the reusable nature of these rules, research should be done to determine how useful they will be to the process of TDD. Additionally, improvements to LEET will need to be made to eliminate distractions based on shortcomings of the tool from future research.

Finally, we hope to test our approach to UITDD on multi-touch and gesture-based interfaces, like those present in the iPhone and Windows Phone 7 mobile devices and in digital tabletops like those produced by Smart, Microsoft, and Evoluce. These interfaces pose interesting challenges in the testing, prototyping, and UITDD of touch-based interactions.

7 Conclusion

In this paper we present a pilot evaluation of an approach to test-driven development of GUI-based applications. This approach uses user interface prototyping and usability evaluation to develop a prototype of a user interface that is less likely to change than when a GUI is developed without these steps. These tests can then be run on the actual GUI of the application under development in a test-first fashion.

However, it was necessary to ask the question: will developers even find these tests useful? In order to investigate this, a pilot evaluation was conducted in which developers were presented with a variety of tools to assist them in the development of a small application, ExpenseManager. Observations were recorded by the researchers in order to observe how developers used these tests, and post-experiment surveys were used to assess the developers' perceptions of UI tests. As a result of the observations, we developed a workflow showing the way in which study participants would utilize UI tests in order to determine when they were done working on a specific feature of the sample application. Through the surveys, we were able to determine that participants seemed to feel that UITDD had benefits, and would be a useful practice. The encouraging results from the pilot study suggest that a more comprehensive study should be conducted to get significant results.

Bibliography

1. Nagappan, N., Maximilien, E., Bhat, T., Williams, L.: Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams. In : Empirical Software Engineering, pp.289-302 (2008)
2. Jeffries, R., Melnik, G.: Guest Editors' Introduction: TDD - The Art of Fearless Programming. IEEE Software, 24-30 (2007)
3. Holmes, A., Kellogg, M.: Automating Functional Tests Using Selenium. In : AGILE 2006, pp.270-275 (2006)
4. Itkonon, J., Mäntylä, M., Lassenius, C.: Defect Detection Efficiency: Test Case Based vs. Exploratory Testing. In : First International Symposium on Empirical Software Engineering and Measurement, Madrid, Spain, pp.61-70 (2007)
5. Hellmann, T., Hosseini-Khayat, A., Maurer, F.: Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design. In : Third International Conference on Software Testing, Verification, and Validation Workshops, Paris, pp.444-447 (2010)
6. Microsoft: SketchFlow. In: Microsoft Expression. Available at: http://www.microsoft.com/expression/products/Sketchflow_Overview.aspx
7. Buxton, B.: Sketching User Experiences. Morgan Kaufmann (2007)
8. Barnum, C.: Usability Testing and Research. Pearson Education, New York (2002)
9. Hosseini-Khayat, A.: Distributed Wizard of Oz Usability Testing for Agile Teams. Master's Thesis, University of Calgary, Calgary (2010)
10. Wilson, P.: Active Story: A Low Fidelity Prototyping and Distributed Usability Testing Tool for Agile Teams. MSc Thesis, Univerity of Calgary (August 2008)
11. Hellmann, T. In: LEET (LEET Enhances Exploratory Testing) - CodePlex. (Accessed 2010) Available at: http://leet.codeplex.com/
12. Xie, Q., Memon, A.: Using a Pilot Study to Derive a GUI Model for Automated Testing. ACM Transactions on Software Engineering and Methodology 18(2), 1-35 (October 2008)
13. Xie, Q., Memon, A.: Studying the Characteristics of a "Good" GUI Test Suite. In : Proceedings of the 17th International Symposium on Software Reliability Engineering, Raleigh, NC, pp.159-168 (2006)
14. Kaner, C., Bach, J.: Black Box Software Testing. In: Center for Software Testing Education and Research. (Accessed Fall 2005) Available at: www.testingeducation.org/k04/documents/BBSTOverviewPartC.pdf
15. Memon, A., Benerjee, I., Nagarajan, A.: What Test Oracle Should I Use for Effective GUI Testing. In : 18th IEEE International Conference on Automated Software Engineering, Montreal, pp.164-173 (2003)

16. Memon, A., Soffa, M.: Regression Testing of GUIs. In : ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.118-127 (2003)

17. Memon, A.: Automatically Repairing Event Sequence-Based GUI Test Suites for Regression Testing. ACM Transactions on Software Engineering and Methodology 18(2), 1-36 (October 2008)

18. Ruiz, A., W., P.: Test-Driven GUI Development with TestNG and Abbot. In : IEEE Software, pp.51-57 (2007)

19. Ruiz, A., Price, Y.: GUI Testing Made Easy. In : Testing: Academic and Industrial Conference - Practice and Research Techniques, pp.99-103 (2008)

20. Burbeck, S.: Appplications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC). In: How to Use Model-View-Controller (MVC). (Accessed 1987, 1992) Available at: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html

21. Marick, B.: Bypasing the GUI. Software Testing and Quality Engineering Magazine, 41-47 (September/October 2002)

22. Brooks, P., Robinson, B., Memon, A.: An Initial Characterization of Industrial Graphical User Interface Systems. In : International Conference on Software Testing, Verification, and Validation, Denver, pp.11-20 (2009)

23. Robinson, B., Brooks, P.: An Initial Study of Customer-Reported GUI Defects. In : Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, pp.267-274 (2009)

24. Memon, A.: A Comprehensive Framework for Testing Graphical User Interfaces. PhD Thesis, University of Pittsburgh (2001)

25. Chen, W., Tsai, T., Chao, H.: Integration of Specification-Based and CR-Based Approaches for GUI Testing. In : 19th International Conference on Advanced Information Networking and Applications, pp.967-972 (2005)

26. Calgary Agile Methods User Group. In: CAMUG. Available at: http://calgaryagile.com/