Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design

Theodore D. Hellmann, Ali Hosseini-Khayat, Frank Maurer
Department of Computer Science, University of Calgary
2500 University Drive NW, Calgary, AB
Canada, T2N 1N4
{tdhellma, hosseisa, fmaurer}@ucalgary.ca

Abstract— **Test-driven development of GUIs is currently very difficult. On the one hand, to avoid frequent updates of the tests, test-driven development requires a degree of stability in the application under development, whereas GUIs are very likely to change during development. On the other hand, the easiest way of creating GUI tests – using a capture/replay tool – requires the GUI to exist. This paper introduces a new approach to user-interface test-driven development, wherein a capture-replay tool is used to record test scripts from low-fidelity prototypes. This allows GUI tests to be written simply and without requiring that the GUI exist first.**

Agile User Experience Design, GUI Testing, Test-Driven Development

I. Introduction

There are many reasons to practice test-driven development (TDD). It encourages communication between customers and developers, increases programmer confidence, increases software quality, and (arguably) decreases bug density without decreasing productivity [1], [2]. As most published approaches to TDD test the software layer just underneath the GUI, test-driven development of graphical user interfaces (GUIs) remains an unsolved problem.

The reason for this is twofold. First, GUIs are very likely to change repeatedly over the course of development. This means that tests will need to be updated and repaired frequently – which is a nontrivial task. Second, the easiest method of creating GUI tests – using a capture-replay tool (CRT) – requires that a GUI exists before tests can be defined. While there are other issues that make testing GUIs difficult, these two present the greatest challenges to user interface test-driven development (UITDD).

The approach presented in this paper combines a low-fidelity prototyping tool with a CRT. First, details of user stories are collected. Second, these stories are used to create a low-fidelity prototype of the system. Third, usability evaluations of the prototype are conducted to identify and fix flaws that would require changes to the interface. These steps are repeated until the prototype is sufficiently stable. 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. The resulting tests can then be run on the GUI while it is being developed to ensure that it matches the user expectations expressed in the prototype.

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 is maintained in the actual implementation, then tests can be recorded from the prototype and replayed on the actual GUI as it is implemented.

II. Problem

In TDD, it has to be possible to write tests before the code for them exists. Once this is done, code can be written that makes the tests pass. Throughout TDD, it needs to be possible to refactor existing code without worrying about breaking tests. Two significant barriers prevent effective application of TDD to GUI-based applications. First, refactoring or changing GUIs tends to break their tests, even if none of the changes are semantic. Second, writing GUI tests by hand is difficult. It's possible to use CRTs to easily write tests, but this isn't an option until after a GUI has been coded, meaning that UITDD must be done by writing tests manually.

Since a GUI will need to change repeatedly over the course of a project, its test suite will need repeated maintenance. This can make GUI tests a double burden. First, maintenance expenses can be substantial [3]. Second, if developers start to see failures as "the test's fault" rather than as an indication of potential bugs in the application itself [4], they will be more reluctant to change code and less likely to take failing tests seriously. Since developer confidence is one of the core benefits of TDD, the second issue is likely the more serious of the two [1].

Two potential solutions to the issue of frequent changes to the GUI arise. First, the team can try to minimize the changes necessary in a GUI and its test suite. Second, methods for automatically repairing or significantly easing the repair of broken GUI tests could be explored. Existing work on the latter approach will be explained in Section III. Our idea, which uses the former approach, will be explored in Section IV.

III. Related Work

A. Capture-Replay Tools

CRTs work by recording interactions with a GUI and storing them as a sequence of actions that can be replayed on that GUI. The fundamental difficulty with this is that methods in widgets can't be accessed easily by test code.

Initially, CRTs avoided this problem entirely by simply recording keyboard input and the screen position of mouse clicks. A test script based on this would simply replay these actions. This sort of testing was useful for detecting crashes, but verifying correct system behavior was another matter. Relying on screen coordinates also has the distinct disadvantage of being very sensitive to non-semantic changes to the GUI under test [5], [3]. Rearranging widgets, for example, would cause test failures even though the application was functioning properly.

The next generation of CRTs use a method called testing with object maps, which works by storing as much information as possible about a widget so that a fuzzy match can be made when the test is run [5], [3]. This makes tests more robust against changes, and also allows widgets to be accessed by the test so that their behavior can be tested. While testing with object maps is more robust and useful than testing with direct input, it's still difficult to code due to the amount of information that must be known about a widget in order to correctly locate it.

Keyword-based testing is a GUI testing technique that has been developed relatively recently. Rather than storing much data about a widget, this system simply assigns a unique keyword to each widget [6], [7]. This means that only a keyword is required to locate and interact with a specific widget from within a test. Keyword-based testing is a robust, easy way to write GUI tests, and is now possible through most CRTs.

B. User Interface Test Driven Development

In recent years, several tools have been developed to support UITDD [7], [8], [6], [4]. These tools are used for UITDD because they simplify manual GUI test authoring by providing framework support that makes identification of and interaction with widgets simpler and more robust. Some provide added robustness by storing tests in an intermediate form along with an intermediate representation of widgets used in testing, which aids in test maintenance [6].

While these tools can reduce the effort involved in UITDD, it's important to note that tests must still be coded manually. Writing GUI test scripts by hand can be a tedious, error-prone task, and an agile team using this approach in the past found that tests written for UITDD tend to need modification before they can even pass for the first time after the corresponding GUI code is written [4]. This team found it faster to rerecord tests using a capture-replay tool than to attempt to repair the initial target GUI test.

C. Support for Test Script Maintenance

Tool support for test maintenance has also been a subject of recent research. Work by Memon and Soffa takes a compiler-inspired approach by attempting to replace events in a broken test automatically in an attempt to create a legal sequence of test steps without the need for human interaction [9]. The TIGOR system works by adding explicit type information to GUI test scripts, simplifying manual maintenance [5]. REST, on the other hand, makes a connection between widgets in an application's code and their use in tests, and is able to make suggestions as to where and why a test script is likely to fail [3]. Actionable Knowledge Models store tests in an intermediate model, which allows the root cause of a test failure to be addressed in a single location rather than propagated between individual test scripts manually [10].

IV. Our Approach

A. Tools

We developed a tool, ActiveStory Enhanced, which supports agile interaction design [11]. ActiveStory Enhanced allows usability engineers to create low-fidelity prototypes. These prototypes are composed of a set of pictures of various states of the user interface and "hot zones," implemented as clear transparent buttons covering specific regions of a prototype, which can cause transitions between states. Low-fidelity prototypes are cheap to create and alter and, through ActiveStory Enhanced, can be usability tested with a number of users in a cheap, distributed fashion.

LEET (a recursive acronym for LEET Enhances Exploratory Testing) is a capture-replay tool we developed based on Microsoft's User Interface Automation Framework (UIAF) [12]. This framework allows for keyword-based testing, which is essential to our approach. Since the only property of a widget that is used to identify it is its AutomationID, it is possible to record a test from an ActiveStory Enhanced prototype and replay it on an actual GUI. This is possible by decorating each hot zone with an AutomationID and ensuring that this same ID is also used for the corresponding widget in the GUI.

B. Process

First, user stories are used to develop a low-fidelity prototype of the GUI using ActiveStory Enhanced. Usability evaluations can be performed on these prototypes. This can 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 iterated agile usability evaluations.

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 under development.

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

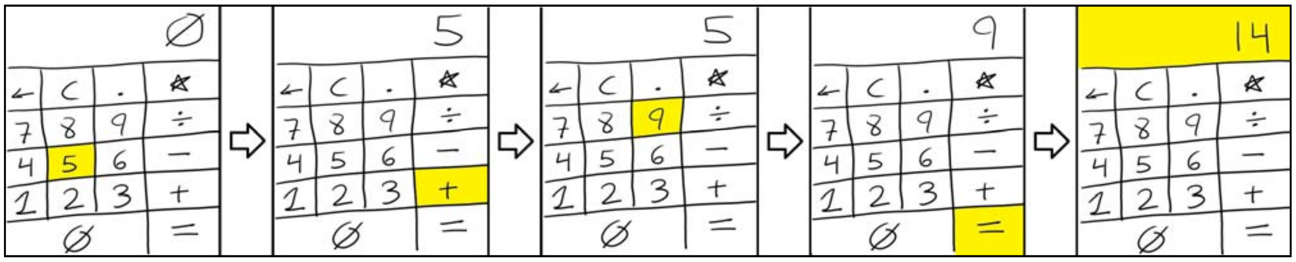

Figure 1. Test sequence for example. Highlighted areas represent mouse clicks in the first four pages and the field to be verified in the last page.

tools used for UITDD. 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, it is expected that this benefit, as well as increased user buy-in, will compensate. Finally, while tools exist to facilitate repair of broken tests, the best solution would be to decrease the instances of tests breaking in the first place. This can avoid the issues with other approaches to UITDD described above.

C. *Example*

For an example of UITDD, let us consider the design of a calculator like that provided with Windows 7. 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 Fig. 1 on a prototype created in ActiveStory Enhanced. For this test, we will expect "5" to be clicked, then "+," then "9," then "=," and for "14" to be displayed as the result. Now, we use LEET to record a sequence of interactions with the prototype to use as a test script. The result is the test shown in Fig. 2.

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 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 UIAF, widgets that display text 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. The next step is 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, 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 will automatically interpret these as AutomationIDs.

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 Fig. 2, START has been changed to start the actual GUI instead of the prototype, and will do this as its first step. The test will fail, as shown in Fig. 3, because none of the application logic for these buttons exists at this point.

Note that the test fails on the second to last line – 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 because keyword-based testing will tolerate cosmetic changes to widgets. Widgets can be resized, moved, even switched between analogous types without breaking the test script.

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 Fig. 4.

V. Limitations

The evaluation of our proposed approach is upcoming. While it is possible to use our method for UITDD, no statements can yet be made as to its practicality or usefulness. In the short term, we plan to conduct experiments in which developers will be asked to conduct UITDD of several features of a simple application. This will provide us with information as to whether our method of UITDD is practical for individuals, and how much it aids in application development.

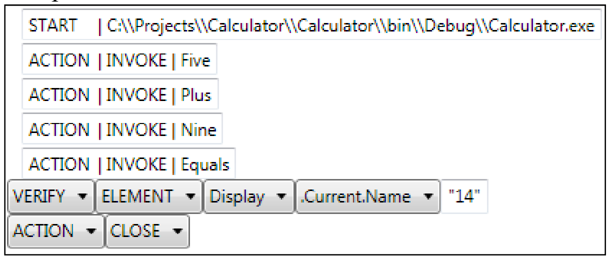
```
START    | C:\\Projects\\Calculator\\Calculator\\bin\\Debug\\Calculator.exe
ACTION | INVOKE | Five
ACTION | INVOKE | Plus
ACTION | INVOKE | Nine
ACTION | INVOKE | Equals
VERIFY | ELEMENT | Display | .Current.Name | "14"
ACTION | CLOSE
```

Figure 2. Test for the calculator's simple addition feature.

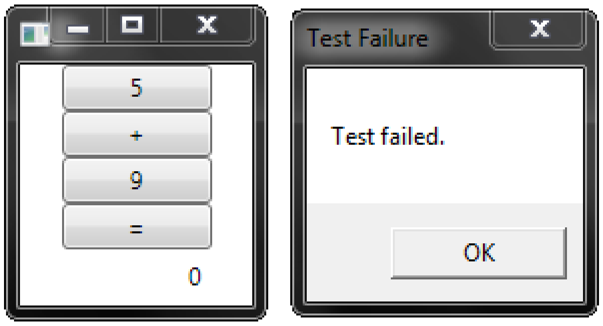

Figure 3. Failing test - application logic still missing.

In the long term, we hope to be able to conduct a case study of our method's use in an industry setting over an extended period of time to determine its actual usefulness to development teams.

It is also assumed that, by doing repeated usability evaluations of a prototype of a GUI, the number of changes developers will be required to make to the GUI later in the development process will be lower. At present, the authors are unaware of any case studies in support of this.

Finally, tests created using the method outlined in Section IV will be subject to the limitations of the low-fidelity prototype from which they are recorded. For example, current low-fidelity prototyping tools, ActiveStory Enhanced included, work well when prototyping buttons, hyperlinks, and the like, but struggle with other common features of user interfaces. For example, there is currently no tool support for low-fidelity prototyping of text boxes, draggable items, and gestures, to name a few. This means that tests for these types of elements must still be written manually.

VI. Future Work

Once the evaluation of the proposed process has been performed, the authors hope to explore mixed-fidelity UITDD. Mixed-fidelity prototypes combine hand-drawn elements from low-fidelity prototypes with actual widgets. Our plan is to create a bridge between these widgets in the prototype and actual features of the application being developed. By incrementally replacing sections of the low-fidelity prototype with functional widgets, a GUI can be incrementally developed from its prototype. This approach could avoid the gap between recording tests on a low-fidelity prototype and running them on a separate GUI.

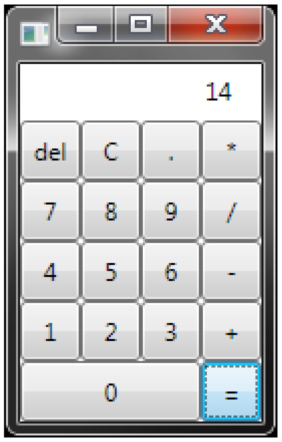

Figure 4. A complete interface. The original test still passes.

References

[1] R Jeffries and G. Melnik, "Guest Editors' Introduction: TDD - The Art of Fearless Programming," *IEEE Software*, pp. 24-30, 2007.

[2] 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, pp. 289-302.

[3] M. Grechanik, Q. Xie, and F. Chen, "Maintaining and Evolving GUI-Directed Test Scripts," in *IEEE 31st International Conference on Software Engineering*, 2009, pp. 408-418.

[4] A. Holmes and M. Kellogg, "Automating Functional Tests Using Selenium," in *AGILE 2006*, 2006, pp. 270-275.

[5] 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, pp. 1-10.

[6] 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, pp. 967-972.

[7] A. Ruiz and Y. W. Price, "GUI Testing Made Easy," in *Testing: Academic and Industrial Conference - Practice and Research Techniques*, 2008, pp. 99-103.

[8] A. Ruiz and Price Y. W., "Test-Driven GUI Development with TestNG and Abbot," in *IEEE Software*, 2007, pp. 51-57.

[9] A. M. Memon and M. L. Soffa, "Regression Testing of GUIs," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003, pp. 118-127.

[10] 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, pp. 165-168.

[11] Ali Hosseini-Khayat, Yaser Ghanam, Shelly Park, and Frank Maurer, "ActiveStory Enhanced: Low-Fidelity Prototyping and Wizard of Oz Usability Tool," in *Agile Processes in Software Engineering and Extreme Programming*, 2009, pp. 257-258.

[12] Theodore D. Hellmann. (2010) LEET (LEET Enhances Exploratory Testing) - CodePlex. [Online]. http://leet.codeplex.com/