9 Agile Interaction Design and Test-Driven Development of User Interfaces – A Literature Review

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

Abstract: This chapter describes the development of GUI-based applications, from usability engineering and prototyping to acceptance test-driven development, in an agile context. An overview of current agile interaction design practices will be presented, including a thorough analysis of the current role of prototyping and current attempts to facilitate test-driven development of GUI systems, as presented in academic and industrial literature. Traditional usability engineering approaches shows that if user input is taken into consideration early in the development process by repeatedly conducting usability tests on low-fidelity prototypes of the GUI system, the final version of the GUI will be both more usable and less likely to require revision. The major risk associated with test-driven development of GUIs is the high likelihood of change in the target GUI, which can make test development unnecessarily expensive and time consuming. A unification of these styles of development will be presented, along with a prediction of how this process can be used to simplify creating testable GUI-based applications by agile teams.

9.1 Introduction

Currently, agile quality assurance practices like test-driven development, continuous integration, and acceptance testing have been widely adopted into mainstream agile development teams. The purpose of this chapter is to introduce two additional types of testing that have yet to be widely integrated into agile software development: usability evaluation and testing of graphical user interfaces.

Traditional usability engineering approaches show that if user expectations are taken into consideration early in the development process by repeatedly conducting usability tests on low-fidelity prototypes of the graphical user interface (GUI) of a system, the final version of the GUI will be both more usable and less likely to require revision. Usability testing focuses on improving the ease of use of a GUI. On top of this, automated testing of GUIs can improve their stability and reliability. In an agile context, test-driven GUI development seems to be an obvious approach and has been discussed by the community (Poole 2005). However, a major risk associated with test-driven development of GUIs is the high likelihood of change in the target GUI and the resulting fragility of tests, which can make test-driven development unnecessarily expensive and time consuming. In this chapter, we will give an overview on the related work on agile interaction design, automated GUI testing and test-driven development of user interfaces. We will then present a unification of these two GUI-related approaches, along with a prediction of

how this process can be used to simplify the creation of tested GUI-based applications.

9.2 Agile Interaction Design

Software usability has the potential to determine the success or failure of a software system. As such, usability engineering practices such as prototyping (Buxton 2007) and usability testing (Barnum 2002) have become increasingly common in software development projects, particularly for commercial software. Adapting usability techniques for use in combination with agile methods is an ongoing process that is gaining increasing attention by usability practitioners and proponents of agile methods (agile-usability Yahoo! Group, 2009). This section presents the motivation behind interaction design in agile methods, the basic concepts of interaction design and adaptation of these concepts for use by agile teams. While some research in this area exists, the integration efforts are currently driven by industry practitioners. As a result, we are discussing both industrial as well as academic resources in our literature review.

9.2.1 Usability and Usability Evaluation

Software with poor usability can have negative effects on productivity and acceptance. If the software is a commercial product to which users have an alternative, poor usability can cause customers to reject the product and choose a competitor's product instead. However, if usage is mandated, bad usability can result in reduced productivity for the end-user and decreased sales for the developer. Bad usability in software developed for in-house use can lead to users circumventing the system and generally reduced productivity. In some cases, bad usability can lead to the failure of a project because of end-users rejecting the system and fighting against its use. It is clear that these effects are all undesirable to varying degrees for the entities involved.

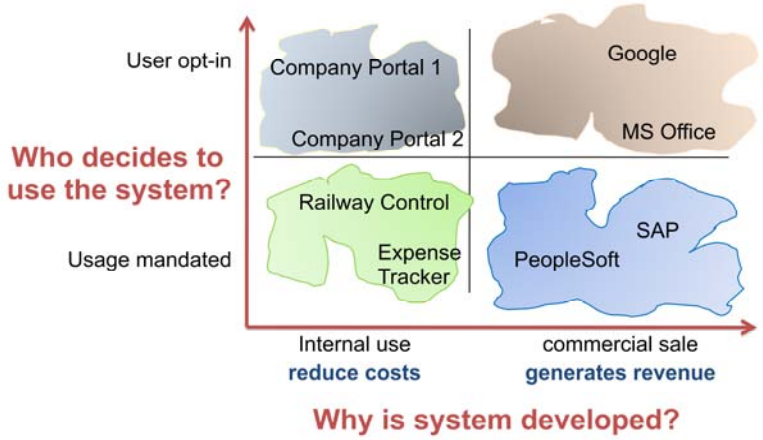

Fig. 1. Usability quadrant (Image Source (Maurer 2009))

On the other hand, software with good usability can result in increased productivity, customer satisfaction and revenue (Barnum 2002; Nielsen 1994). Additionally, a user interface that has gone through multiple usability evaluations is more likely to be relatively stable in terms of changes needed because of the discovery of usability flaws in later stages. Software usability is receiving increasing attention particularly for commercial products. The first step in improving usability is typically creating prototypes of the user interface based on an analysis of user needs and tasks. Prototypes usually start with sketches of user interfaces (so called low-fidelity prototypes) and can then be refined into medium or high fidelity prototypes. Typically, the next step involves evaluating the design of the prototype using a variety of techniques (see below). Finally, the results of the evaluation are analyzed and the design is improved to alleviate any usability issues that may have been found. The process is then started again using the updated design, forming an iterative design process in which each iteration improves upon the design until a satisfactory design is obtained.

9.2.2 Traditional Usability Evaluation and Conflicts with Agile Methods

Traditionally, large amounts of design work and usability evaluation are done prior to any implementation. Implementation of a release only starts once the entire iterative user experience design process is completed, including prototype design, evaluation, analysis and redesign. Once the design has been finalized, it is handed over to the developers for implementation of that phase. Before the next release, the designers gather feedback from the latest release and incorporate it in the design for the next release. In this way design flaws found in a release, either during development or post-release, are corrected in the next release of the software. This

results in a large gap between when a flaw is identified in a release and when it is actually fixed, if it is not found and corrected in the initial design phase.

The traditional usability evaluation process appears to have some fundamental conflicts with the principles of agile methods, particularly the need for short iterations, the preference towards minimal up-front design work and the non-functional design artifacts. Agile methods prescribe devoting minimal time to designing and creating design documents in the start of a project. Instead, design is done during each iteration for the features involved in that particular iteration. Clearly, this poses a problem when employing traditional usability techniques, where all the design work should be finished prior to any other work on the project. Short iterations are another concept found in agile methods. This practice can be problematic if a complete traditional usability testing process is attempted at the beginning of each iteration as that process is a lengthy one. Recruiting participants and collocating them with the testers makes traditional usability testing a lengthy process, as well as prototype design and analysis of evaluation results.

9.2.3 Agile Methods and Discount Usability

One possible solution to the apparent clash between agile methods and usability evaluation techniques is to utilize discount usability techniques (Nielsen 1994), proposed initially by Jakob Nielsen. The concept behind discount usability is that some amount of usability evaluation is better than none. As such, usability evaluation is done with a more limited scope and with cheaper methods. Heuristic evaluation (Nielsen 1995) and Wizard of Oz testing (Nielsen 1993) are both discount usability techniques. Heuristic evaluation involves evaluating a prototype or actual design against a set of heuristics based on what an interface with good usability is like.

Wizard of Oz usability testing is a concept derived from Frank Baum's novel, The Wonderful Wizard of Oz (Baum 1900). In the story, one of the characters, known as the Wizard of Oz, manipulates a contraption from behind a curtain, deceiving the main characters into thinking the contraption is a working one. In usability terms, the concept of Wizard of Oz testing is showing a test subject a view of a non-working prototype, observing the user's input or reaction and manipulating the view of the prototype as appropriate, thereby giving the illusion that the prototype actually works.

When used in discount usability approaches, the prototype used for Wizard of Oz testing is typically a pen and paper, low-fidelity prototype. One of the test conductors acts as the "wizard" while another observes the users' behavior and reactions. By observing the users' actions and behavior the designers can get an idea of what the user is expecting, whether or not an element is located in the right place and many other indications of usability flaws. Often the Wizard of Oz testing approach is accompanied by a think aloud protocol (Nielsen 1994), whereby participants are asked to articulate their thoughts as they interact with the prototype. Prototypes can have multiple levels of detail, although the most commonly used level

of detail for Wizard of Oz testing is a low-fidelity one. The following section will describe the different levels of detail and their benefits and disadvantages.

9.2.4 Prototype Levels of Detail

User interface prototypes can be categorized by their level of detail into one of three categories: low-fidelity, high-fidelity or mixed-fidelity. Low-fidelity prototype refers to prototypes that resemble (or are) sketches of the user interface. Wireframe designs, pen and paper sketches and whiteboard drawings are examples of low-fidelity prototypes (Nielsen 1990).

Low-fidelity prototypes are a concept adopted from similar design artifacts in other fields, such as movie production, where storyboards (Hart 1999; Bailey et al. 2001) are created prior to the bulk of the production process. Low-fidelity prototypes have the advantage of focusing the attention of test participants and reviewers on basic and essential usability questions such as the placement of buttons, navigational issues and organization of the interface. This is opposed to the feedback that is obtained based on higher level prototypes or finished systems, where users tend to focus on superficial issues such as fonts and colors. Another advantage of low-fidelity prototypes is that they are easy to create, cheap in terms of time and resources and can easily be discarded or modified. Additionally, when providing feedback on these types of prototypes users feel that input they provide that requires major modifications will have a better chance of being taken into account, since the system does not appear finalized. Due to the fact that they are cheap, easy to modify and require minimal upfront design work they are ideal for

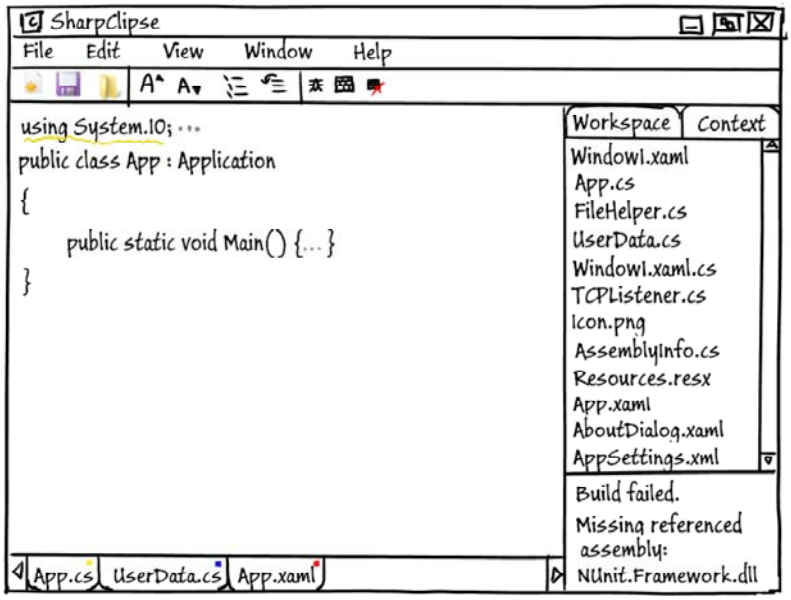

Fig. 2. Low-fidelity prototyping

integrating usability methods with agile methods. Additionally they provide good ROI, as a relatively smaller amount of time is spent designing and testing them while still addressing and discovering the most serious issues.

High-fidelity prototypes are ones that closely resemble a finished system. These prototypes are typically built with GUI-builder tools, often the same tools that are used to build the finished system. An obvious advantage to using high-fidelity prototypes is that in many cases they can be used directly in the finished system. However, this comes with the drawback that most of the feedback received from evaluating these types of prototypes will be focused on superficial issues (Nielsen 1994; Barnum 2002). While this type of feedback is useful in later stages of design, in the early stages of design it is more important to get feedback on issues such as navigation and widget (graphical object) placement.

Mixed-fidelity prototypes (Coyette et al. 2009) may consist of a low-fidelity prototype with elements of a high-fidelity prototype integrated in it, for example a wireframe with actual buttons or other controls. The opposite is also possible in which a high-fidelity prototype has portions that are sketches or wireframes; however, these types of mixed fidelity prototypes are less common. Mixed-fidelity prototypes can be useful in transitioning between a low-fidelity prototype and an actual implementation.

9.2.5 Project Vision

In the current state of agile software development, it can be difficult to obtain a perception and understanding of the broader picture of the software. Features and iterations entail a vertical separation of the system under development. In other words, each feature or iteration represents a vertical cut through the system (cf. Fig. 3), which is limited to the parts of the system relevant to that feature or iteration. As a result, maintaining a consistent and unified vision throughout the many iterations of development can be difficult.

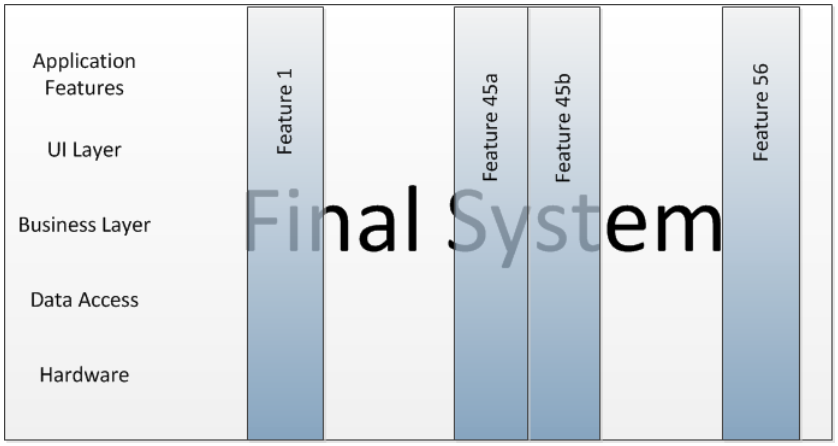

Fig. 3. Vertical slices through system from iterative and incremental development

As an aid to obtaining a broad view of the system, prototypes of varying detail and fidelity can be designed and utilized (Patton 2002; Meszaros and Aston 2006). In addition to providing the broader picture, they also serve as design artifacts necessary for performing usability evaluation. Prototypes are abstracted representations of the user interface of a software system in the sense that they generally do not have any underlying business code and in some cases do not contain any form of implementation at all. The need for a broad project vision early in the project can be considered as motivation for designing prototypes.

9.3 User Interface Test-Driven Development

There are two ways of testing a GUI-based application (GAP): with or without automation. In manual GUI testing, a tester interacts directly with the AUT. This sort of testing is time-consuming, meaning that fewer tests can be run over the course of a project, and distinctly un-agile. In automated GUI testing, tests are written to simulate user interaction with a GAP, and can be run as part of a continuous integration setup. Automated tests can be run faster and more regularly than manual GUI tests, but at a price. First, a GUI test must mimic the way in which a real user would interact with the system, which can be difficult since widgets are often hard to interact with from test programs. Next, GUI tests tend to be fragile, and maintenance can be expensive: it has been estimated that Accenture Technology Labs spends between $50 and $120 million annually to repair broken GUI test scripts (Grechanik et al. 2009). Compounding this, GUIs are very likely to change over the course of development, virtually ensuring that test maintenance will be required. Finally, GAP tests are difficult to write in the first place due to the degree of freedom GUIs allow users. This is because GUIs can enter a large

number of possible states in response to user input, and it is often difficult to determine the validity of a given state in an automated fashion. Given these difficulties, it might be tempting to declare automated GUI testing too expensive to do and to focus on testing the underlying application instead. However, any untested code should be viewed as a potential source of bugs. Even using a model-view-controller -style architecture does not obviate the need for GUI testing – especially since 45-60% of a GAP's code may lie in its GUI (Memon 2001). Despite this, automated GUI testing in practice is generally done inadequately despite the reality that bugs in the GUI really do impact end-users (Xie and Memon 2008; Robinson and Brooks 2009).

Test-driven development (TDD) is a very important tool for agile teams. TDD provides a safety net of regression tests to agile developers, allowing them to confidently change code in response to changing requirements, reduce bug density, and increase communication between team members, but TDD of GUI-based systems is difficult because current GUI testing tools focus on testing existing systems (Jeffries and Melnik 2007; Nagappan et al. 2008). This means either that authoring GUI tests in TDD must essentially be done by hand or must be delayed until after the GUI has been coded. Doing the former will violate the agile principle of simplicity because TDD of GAPs currently requires the same tests to be written and repaired multiple times. Doing the latter, however, will mean that the safety net of regression tests provided by TDD won't cover the entire system. So, at present, agile development of GAPs is an essentially contradictory process. In this section, the context of GUI testing will be explored: the impact of GUI defects on customers; the idiosyncrasies of GAPs that make them difficult to test; and finally the difficulties and advantages to attempting to apply test-driven development to GAPs. Previous attempts at test-driven development of GAPs will be explored, and conflicts between the requirements of these attempts and the tenets of agile software will be explored, followed by a discussion of possible reconciliations.

9.3.1 What Makes GUI Testing Difficult?

The difficulties encountered in GUI testing are no different from those encountered in other kinds of testing. They are simply less avoidable here than they are in other contexts. Three key encumbrances for testing in general become very difficult to address when performing GUI testing: complexity, verification, and change. Individually, each encumbrance represents a significant challenge; together, they make GUI testing nearly impossible to perform in an agile fashion.

Complexity

The more freedom is allowed a user, the larger an application's interaction space becomes. Work by Xie and Memon (Xie and Memon 2008) has shown that the number of possible event sequences increases exponentially with the length of a test case for a GAP. Compounding this, experimentation (Xie and Memon 2006) by the same group has shown that there are two important factors in determining

how effective a test suite will be: the number of events from the GUI's interaction space that have been used in the test suite; and the number of different states in which these are triggered. In other words, the interaction space in which testing can be performed on GAPs is truly huge, and testing only a portion of it is very likely to have a negative impact on the test suite's ability to detect defects.

Without an effective suite of regression tests, agile developers can't be as confident when making changes to existing code, but creating tests comprehensive enough to deal with the complexity of GUIs is a daunting task – especially when attempted in a test-driven fashion.

Verification

Any test consists of two parts: a test procedure and a test oracle. A test procedure might consist of a set of actions to take on the system, whereas a test oracle contains information that will be used to check whether the system responded appropriately to these actions. Since automated testing is constrained by the way in which test oracles are defined and used (Kaner and Bach 2005), the fact that it's difficult and/or expensive to create useful GUI test oracles makes it difficult to create good GUI tests. In work by Memon, Banerjee, and Nagarajan (Memon et al. 2003), both the information contained in an oracle ("oracle information") and the way this information is compared to the GAP at test-time ("oracle procedure") are explored. In short, GUIs are usually composed of windows, each of which contains a set of widgets. Since the state of a widget can be described as the parameters it exposes and their values, the state of a window can be described as the state of its widgets. If tests are being generated automatically, this information can be compared to the application under test (AUT) either after each step of the test or after the entire test has completed. It was found that more complex oracles have stronger fault-detection ability. For example, oracles that store more information about the state of the AUT and that compare this information to the AUT after each step. Of course, these will take more time to run and space to store. This, combined with the complexity issue, implies that GUI test suites are likely to grow very large very quickly. This makes automated GUI testing of an application very difficult in an agile environment due to the overhead involved in test suite creation and the fact that rapid feedback will be less likely as the project continues. Commonly used GUI testing tools also require the GUI to be available before they can be used.

Change

Unfortunately, the fact that GUI testing is best done with complex test oracles makes these oracles exceedingly vulnerable to changes in the GUI. GUIs are also likely to be changed on a regular basis during the course of development, which means that GUI test suites will be broken again and again. Research has shown (Memon and Soffa 2003) that it is possible for over 74% of GUI test cases for a GAP to become unusable after modifications to its GUI. When this happens, test cases have to be repaired or rerecorded.

Traditionally, tool support for repairing broken GUI tests has been lacking, but several interesting tools have been proposed recently. In work by Memon and Soffa (Memon and Soffa 2003), a compiler-inspired approach is taken. GUI test suites are scanned for sequences that are illegal in the new version of the GUI, and events are deleted or inserted until the sequence is again found to be legal. Fu, Grechanik, and Xie take a different approach, one that focuses on how objects are used in test scripts rather than their specific type at runtime (Fu et al. 2009). Their system, TIGOR, is able to determine information about the types of widgets used in test scripts and make this explicit, making manual repair of broken GUI tests less difficult. The same team has also developed another system, REST (Grechanik et al. 2009), which is able to determining which objects in the GUI have been changed between revisions and then generate suggestions based on where in a test script a failure is likely to happen, and why. Similarly, Actionable Knowledge Models (Yin et al. 2005) have been used to store test data so that revisions can be propagated between many tests quickly.

However, while these tools represent a step in the right direction, the frequent breaking and subsequent repair of GUI tests remains a significant problem. The most significant danger here is that, as has been reported on agile projects in the past (Holmes and Kellogg 2006), team members may begin to dismiss failing tests as "the test's fault," in which case the credibility of an automated test suite will be significantly reduced. While repairing broken tests is a significant advance, a better solution would be to find a way to increase the robustness of GUI tests.

It's interesting to note that change remains a significant challenge to GUI testing even within agile teams. While agile teams should be able to react to and embrace change, the level of complication involved in GUI testing means that change remains a significant risk even to agile development efforts. A solution to the problem of TDD of GAPs, then, must be able to address to the triple threat of change, validation, and complexity if it is to be of use to agile teams.

9.3.2 GUI Testing – An Overview

To test a GUI, a test script must first be defined. A test script consists of a series of actions to be performed on the AUT. Scripts can be used without a test oracle, in which case a crashing AUT signifies a failed test. This sort of testing can be executed by either a manual tester or an automated test runner. More advanced tests can be created by adding a test oracle to a script, which allows for more detailed testing. It's possible to write GUI test scripts by hand, but GUI testing is generally done using a capture-replay tool (CRT). CRTs capture user interactions and store them as a script that can be replayed later as a regression test.

While CRT-based tools are popular for writing GUI tests, invariant-based and model-based methods show promise. Invariant-based testing is done by defining a set of invariants, or things that are not allowed to change, about a system, and verifying that the GUI upholds each invariant after each step in a test script. Model-based testing is done by creating an intermediate representation of an AUT and us-

ing this to automatically create a test suite that will meet certain criteria. In this section, each of these techniques will be explained.

Capture-Replay Tools

CRTs were initially very basic systems. They would simply record mouse movements, clicks, and keyboard input as scripts, which would later be replayed. Relying on screen coordinates has the distinct disadvantage of creating fragile tests, since refactoring of the GUI, such as rearranging its widgets, will cause test failures even though the AUT itself is functioning appropriately (Fu et al. 2009; Grechanik et al. 2009). For example, even cosmetic refactoring of an application's GUI, refactoring that changed the location of widgets without changing any of the functionality of the system, would break tests. Because of this, a system called testing with object maps was developed to ease widget identification (Fu et al. 2009; Grechanik et al. 2009). In this system, detailed information about a widget is recorded so that a best-fit match can be made when the test is run. While more robust, this sort of test is difficult to write by hand, so keyword-based identification has been gaining popularity (Ruiz and Price 2008; Chen et al. 2005; Ruiz and Price 2007). In this system, developers assign a unique identifier to each widget so that it can be easily located in test fixtures.

While modern CRTs allow rapid creation of relatively stable scripts, there are two main disadvantages to their use on agile projects. First, it is often faster to simply discard and re-record a broken test than it is to attempt maintenance, leading to a gap between TDD of GUIs and maintenance of GUI tests. Since tests are likely to break frequently, the same tests are likely to be recorded multiple times in different versions of the GUI. Care must be taken to ensure that the newly-recorded tests remain identical to the now non-working tests they are replacing, lest the regression suite drift away from the functionality it was intended to test. If this were to happen, it would compromise the usefulness of the regression suite of GUI tests as a safety net for development. Second, since CRTs record a user's interactions with a working GUI, a GUI must exist in order to use a CRT, meaning that test-driven development is not currently possible from a CRT itself, which forces agile teams to choose between TDD or CRTs.

Invariant-Based Testing

In invariant-based testing, rules which define expected or prohibited system behavior are created (Mesbah and van Deursen 2009). Mesbah and Deursen have applied this technique to the testing of AJAX-based web-based applications (WBAPs) (Mesbah and van Deursen 2009). In their study, invariants were applied to characteristics like the DOM tree; for instance, the DOM should never contain error messages. As a script is executed on the AUT, invariants are checked after each step. This method ties in nicely with the issue of GUI complexity in that it is testing for a wide range of possible errors at each step of the test, and each invariant is based on some functional expectation of the AUT rather than on whether the AUT matches an ideal version of itself. The distinction is that what is being tested with invariants is closer to "is the system correct?" than it is to "does the actual

system match an ideal version?" When verifying a GUI against an ideal version, that ideal version must first be created. Also, the correctness of the actual GUI will be largely dependent on the correctness of the model. On the other hand, testing for system correctness is only limited by the correctness of the customer's specifications, so agile teams are already familiar with correcting the sort of errors that these tests can incur.

Model-Based Testing

Model-based testing requires generation of an intermediate version of the GUI, either by reverse-engineering a working GUI or generating one based on specifications. This model is then used for automatic test case generation. Model-based GUI testing has recently explored the use of Hierarchical Predicate Transition Nets (Reza et al. 2007), UML diagrams (Vieira et al. 2006), Labeled State Transition Systems (Jaaskelainen et al. 2009), and Event-Flow Models (Memon 2007; Lu et al. 2008). Advantages of these systems include ease of automation, increased code coverage, reduced coupling between the implementation of a test and the implementation of a GUI, and increased cohesion within the test suite. Disadvantages are that a model of the GUI must first be created, and this model must be maintained or regenerated when the system changes. This, again, violates the agile principle of simplicity in that additional work must be done on a regular basis in order to support testing.

9.3.3 Test-Driven Development

In test-driven development (TDD), a developer will first write tests based on the requirements for the feature s/he's about to implement, which will be capable of verifying that the new feature is working correctly. Next, s/he will write just enough code in order to make the new tests pass - without, of course, breaking any other tests. Finally, the new code should be refactored until it is not just passing the tests, but well implemented as well.

TDD is an incredibly important core process of agile software engineering. Over time, TDD forms a safety net of tests covering the entire application. This safety net means that developers can confidently change code in response to changing requirements since regression errors will be caught immediately by existing tests. TDD also encourages communication between customers and developers, which is central to the success of an agile project (Jeffries and Melnik 2007). Most importantly, there is evidence that TDD is able to and increases quality by decreasing the defect density (bugs per KLOC) of an application without significantly decreasing productivity ((Nagappan et al. 2008), for example), though that remains an open subject (Jeffries and Melnik 2007).

However, the most straightforward methods of testing GUIs require the GUI to exist before the tests can be recorded, which makes performing TDD on GUI-based applications difficult. So how has TDD of GUIs been done so far?

9.3.4 Test-Driven Development of GUI-Based Applications

Test-driven development of user interfaces has been discussed in the agile community for some time now (Poole 2005). However, no broadly used solutions have been found for agile teams. Several methods that have been proposed will be described, along with a short explanation of why these approaches aren't ideal for agile development environments.

TestNG-Abbot and FEST

Ruiz and Price (Ruiz and Price 2007) developed an integration of TestNG, a Java testing framework, and Abbot, a library for testing Swing GUIs, which facilitates TDD of GUIs. TestNG-Abbot makes writing GUI tests manually much easier in that it hides much of the complexity of coding GUI tests in two ways. First, it makes it easier to find widgets using enhanced matcher objects. Second, it makes it easier to interact with widgets by wrapping complicated Robot functionality inside more coder-friendly methods. The result is that this integration overcomes some of the dissociation between test code and widgets in that it makes it possible for tests to treat widgets in the GUI in much the same way as it would treat objects in the rest of the application. The TestNG-Abbot integration grew into FEST, which makes writing GUI tests even simpler and further enables TDD of GAPs (Ruiz and Price 2008) .

GUI Testing Tool

GTT (GUI Testing Tool) enables TDD of GAPs through specification-based testing (Chen et al. 2005). In this system, testers define sets of user interactions and the expected system response based on a list of pre-defined GUI components. Note that manual test development in GTT is done at a more abstract level than in the above, where tests are coded directly. Additionally, because a GUI is likely to change once it has been developed, breaking existing tests, GTT is integrated with a CRT to facilitate test maintenance.

Selenium and Similar Software

Testing WBAPs is similar to testing GAPs in that both can be tested through their user interfaces, and similar progress has been made in this field. TDD of WBAPs through directly coding tests is possible using Selenium (and many other mostly-analogous applications). However, a study done by Digital Focus found that using this method for TDD seldom works seamlessly (Holmes and Kellogg 2006). Tests written in a test-first fashion would rarely work after a feature is implemented. Further, this usually happened for minor reasons - for example, misidentification of a widget, or the test running faster than the WBAP could respond. Because of this, developers got into the habit of writing a test, then writing application code, and then getting the test to pass afterwards. It is interesting to note that Selenium gives multiple options for how tests can be repaired: developers can of course manually alter test code, but they also have the option of using Selenium's CRT to re-record a test on the new version of an interface.

All of these approaches suffer from basic faults that prevent them from being an ideal solution for TDD of GAPs. The most important of these faults is that no attempt is made to minimize the amount of change that GUIs will need to undergo during development. At present, GUIs are very likely to change over the course of development, which is likely to break their attendant tests, which will have to be repaired or recoded – either by hand, as in FEST, or through a CRT, as in Selenium and GTT. This is the opposite of the core agile principle of maximizing work not done in that it requires the same work to be done multiple times.

Also, these systems create a gap between TDD and test maintenance. When the GUI actually exists, a broken test can be more simply fixed by using a CRT to re-record it on the actual GUI than it is to repair the test by hand. This means that a new test, written in a different manner from the first test, is being substituted into the regression suite for the application. When tests are rewritten in this manner, extreme care needs to be taken that new tests are semantically identical to the tests they are replacing. Otherwise, there is a risk that the suite of regression tests carefully created through TDD will no longer be able to act as an adequate safety net, and developers may find themselves once again coding without a suite of tests to ward against regression errors.

9.3.5 Future Directions: Using Prototyping for TDD of GUIs

So far, we addressed both GUI testing and TDD of GUIs. Existing approaches provide frameworks that make it easier to write GUI tests by hand and for maintaining these tests once a GUI has been developed. Before these approaches can be smoothly integrated into agile development environments, several conflicts with agility must be resolved.

First, when great effort must be placed on maintenance of GUI tests, emphasis shifts towards the tool used for creating or running these tests and away from the original intent of the tests - namely, to prove to all interested parties that the software being developed meets the customer's requirements. Similarly, a large, fragile, or untrustworthy test suite makes it very difficult to welcome change, much less embrace it or use it for the customer's advantage, due to the amount of extra work that must be done to ensure the integrity of the tests.

Second, writing GUI tests according to TDD must become a simpler process. While writing GUI tests from an existing GUI is relatively simple, and while test-first development of a GUI is technically possible, it remains difficult to write tests that will actually assist in GAP development. TDD of GUIs must become an intrinsic part of the rest of the testing process, rather than representing an obstacle to it.

Third, the limitations of GUI TDD negate many of the benefits associated with TDD. Namely, TDD should increase a developer's confidence in the code base, but a fragile test suite does the opposite. In this situation, developers will be reluctant to embrace changes due to the technical limitations of GUI testing.

In order to enable GUI TDD in a straightforward, robust manner, it is necessary to integrate it into the existing agile testing framework. First, a stable basis for

GUI testing must be established. We believe this can be done through an agile usability evaluation process. This would involve repeated usability evaluation, through which an application's GUI would be prototyped, tested, and analyzed. Because of this, usability issues are more likely to be discovered before the GUI is actually implemented, making the GUI more stable during the course of development.

Additionally, if a low-fidelity prototype were decorated with realistic automation information, including widget identification information and information about how widgets will cause the GUI to transition between states, these interactions could be recorded and used for TDD of the actual GUI. We are currently conducting research to support this proposed approach.

9.4 Conclusion

In this chapter, agile usability evaluation and test-driven development of GUI-based applications have been covered. We covered the conflicts between traditional usability testing and agile methods. The current solution in the agile community – the use of discount usability testing – is also addressed. Recent advances in the testing of GUI-based applications were discussed, as well as attempts at enabling test-driven development of these systems. Issues preventing the adoption of test-driven GUI development into agile development practices were presented, and an overview on a possible solution to these obstacles was given.

References

(2009). Retrieved from Storyboard Image Source: http://www.jonstanicek.com/storyboards.htm

(2009). From Excel 2007 High-fidelity prototype image source: http://dashboardspy.com/img/excel-2007-hi-fidelity-prototype.png

(2009). Retrieved from agile-usability Yahoo! Group: http://tech.groups.yahoo.com/group/agile-usability/

Bailey, B. P., Konstan, J. A., & Carlis, J. V. (2001). DEMAIS: designing multimedia applications with interactive storyboards. Proceedings of the Ninth ACM International Conference on Multimedia (pp. 241-250). Ottawa: ACM Press.

Barnum, C. (2002). Usability Testing and Research. New York, NY: Pearson Education.

Baum, F. (1900). The Wonderful Wizard of Oz. Chicago: George M. Hill Company.

Buxton, B. (2007). Sketching User Experiences: Getting the Design Right the Right Design. Morgan Kaufmann.

Chen, W., Tsai, T., & Chao, H. (2005 March). Integration of Specification-Based and CR-Based Approaches for GUI Testing. Proceedings of the 19th International Conference on Advanced information Networking and Applications, 1, 967-972.

Coyette, A., Kieffer, S., & Vanderdonckt, J. (2009). Multi-fidelity prototyping of user interfaces. In Human-Computer Interaction - INTERACT 2007 (pp. 150-164). Berlin / Heidelberg: Springer.

Fu, C., Grechanik, M., & Xie, Q. (2009 April). Inferring Types of References to GUI Objects in Test Scripts. Proceedings of the 2009 International Conference on Software Testing Verification and Validation, 1-10.

Grechanik, M., Xie, Q., & Chen, F. (2009 May). Maintaining and evolving GUI-directed test scripts. Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 408-418.

Hart, J. (1999). The Art of the Storyboard: Storyboarding for Film, TV, and Animation. Focal Press.

Holmes, A., & Kellogg, M. (2006 July). Automating Functional Tests Using Selenium. Proceedings of the Conference on AGILE 2006, 270-275.

Jaaskelainen, A., Katara, M., Kervinen, A., Maunumaa, M., Paakkonen, T., Takala, T., et al. (2009 May). Automatic GUI test generation for smartphone applications - an evaluation. 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009, 112-122.

Jeffries, R., & Melnik, G. (2007 May/June). Guest Editors' Introduction: TDD--The Art of Fearless Programming. IEEE Software, 24(3), 24-30.

Kaner, C., & Bach, J. (2005 Fall). From Center for Software Testing Education & Research: http://www.testingeducation.org/k04/documents/BBSTOverviewPartC.pdf

Lu, Y., Yan, D., Nie, S., & Wang, C. (2008 December). Development of an Improved GUI Automation Test System Based on Event-Flow Graph. Proceedings of the 2008 International Conference on Computer Science and Software Engineering - Volume 02, 2, 712-715.

Maurer, F. (2009 July). Agile methods and interaction design: friend or foe? Proceedings of the 1st ACM SIGCHI Symposium on Engineering interactive Computing Systems, 209-210.

Memon, A. M. (2001). A Comprehensive Framework for Testing Graphical User Interfaces. Doctoral Thesis, University of Pittsburgh.

Memon, A. M. (2007 September). An event-flow model of GUI-based applications for testing. Software Testesting, Verification, and Reliability, 17(3), 137-157.

Memon, A. M., & Soffa, M. L. (2003 September). Regression testing of GUIs. Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 118-127.

Memon, A., Banerjee, I., & Nagarajan, A. (2003 October). What test oracle should I use for effective GUI testing? Proceedings. 18th IEEE International Conference on Automated Software Engineering, 164-173.

Mesbah, A., & van Deursen, A. (2009 May). Invariant-based automatic testing of AJAX user interfaces. Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, 210-220.

Meszaros, G., & Aston, J. (2006). Adding Usability Testing to an Agile Project. Proceedings of the Conference on AGILE 2006, (pp. 289-294). Washington, DC.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008 June). Realizing quality improvement through test driven development: results and experiences of four industrial teams. Empirical Software Engineering, 13(3), 289-302.

Nielsen, J. (1990). Paper versus computer implementations as mockups scenarios for heuristic evaluation. Proceedings of the IFIP Tc13 Third Internationl Coneference on Human-Computer Interaction, (pp. 315-320). Amsterdam.

Nielsen, J. (1993). Usability Engineering. Morgan Kaufmann.

Nielsen, J. (1994). Guerilla HCI: using discount usability engineering to penetrate the intimidation barrier. In J. Nielsen, R. Bias, & D. Mayhew (Eds.), Cost-Justifying Usability (pp. 245-272). Orlando, FL: Academic Press.

Nielsen, J. (1995). Usability inspection methods. Conference Companion on Human Factors in COmputing Systems, (pp. 377-378). Denver, Colorado.

Patton, J. (2002). Hitting the Target: Adding Interaction Design to Agile Software Development. OOPSLA 2002 Practitioners Reports, (pp. 1-ff). Seattle, Washington.

Poole, C. (2005). Test-Driven User Interfaces. Proceedings of Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005, 285-286.

Reza, H., Endapally, S., & Grant, E. (2007 April). A Model-Based Approach for Testing GUI Using Hierarchical Predicate Transition Nets. Proceedings of the international Conference on information Technology, 366-370.

Robinson, B., & Brooks, P. (2009 April). An Initial Study of Customer-Reported GUI Defects. Proceedings of the IEEE international Conference on Software Testing, Verification, and Validation Workshops, 0, 267-274.

Ruiz, A., & Price, Y. W. (2007 May). Test-Driven GUI Development with TestNG and Abbot. IEEE Software, 24(3), 51-57.

Ruiz, A., & Price, Y. W. (2008 August). GUI Testing Made Easy. Proceedings of the Testing: Academic & industrial Conference - Practice and Research Techniques, 99-103.

Vieira, M., Leduc, J., Hasling, B., Subramanyan, R., & Kazmeier, J. (2006 May). Automation of GUI testing using a model-driven approach. Proceedings of the 2006 international Workshop on Automation of Software Test, 9-14.

Xie, Q., & Memon, A. M. (2006 November). Studying the Characteristics of a "Good" GUI Test Suite. Proceedings of the 17th international Symposium on Software Reliability Engineering, 159-168.

Xie, Q., & Memon, A. M. (2008 Nov.). Using a pilot study to derive a GUI model for automated testing. ACM Trans. Softw. Eng. Methodol., 18(2), 1-35.

Yin, Z., Miao, C., Shen, Z., & Miao, Y. (2005 September). Actionable Knowledge Model for GUI Regression Testing. Proceedings of the IEEE/WIC/ACM international Conference on intelligent Agent Technology, 165-168.

Author Biographies

Theodore D. Hellmann is the lead developer of LEET. He has experience with user interface automation, acceptance and unit testing, and the .NET Framework. His research interests include acceptance testing, test-driven development, and GUI testing. He joined the Agile Software Engineering lab at the University of Calgary after graduating magna cum laude from Christopher Newport University with B.Sc. in Computer Science.

Ali Hosseini-Khayat is a PhD student at the University of Calgary under the supervision of Dr. Frank Maurer. He holds a Master's degree in Computer Science from the University of Calgary. He is the lead developer of ActiveStory Enhanced and has experience with user interface design, usability evaluation and the .NET Framework.

Frank Maurer, is currently a full professor at the University of Calgary. His research has included work in artificial intelligence, agile software engineering, ac-

ceptance testing, and many other areas of software development. Over the course of his career he has formed many industrial, research, and governmental partnerships. Most recently, Frank is the principle investigator of an NSERC Strategic Network grant (the largest grant available from the Canadian National Science and Engineering Research Council).