LIFT: Taking GUI Unit Testing to New Heights

Jason Snyder, Stephen H. Edwards, Manuel A. Pérez-Quinones
Department of Computer Science
Virginia Tech
Blacksburg, VA 24073
{snyder84, edwards, perez}@cs.vt.edu

ABSTRACT

The Library for Interface Testing (LIFT) supports writing unit tests for Java applications with graphical user interfaces (GUIs). Current frameworks for GUI testing provide the necessary tools, but are complicated and difficult to use for beginners, often requiring a significant amount of time to learn. LIFT takes the approach that unit testing GUIs should be no different than testing any other type of code. By providing a set of frequently used filters for identifying GUI components and a set of operations for acting on those components, LIFT lets programmers quickly and easily test their GUI applications.

Categories and Subject Descriptors
K.3.2 [Computer and Information Science Education]: Computer Science Education; D.2.5 [Software Engineering]: Testing Tools

General Terms
Verification, Design

Keywords
GUI testing, unit testing, Java, JUnit, Swing, Java Task Force, JTF, objectdraw

1. INTRODUCTION

The primary goal of LIFT is to enable programmers, and especially new or student programmers, to write GUI tests for Java applications without having to learn a new, complicated tool set. LIFT does this in several ways. The first is by treating GUI testing as much like traditional testing as possible. The majority of tests written using LIFT follow the simple pattern of retrieving a reference to an object, manipulating the object in some way (usually by calling one or more of its methods), then testing the state of the object to ensure it behaved as expected. Second, LIFT greatly improves upon the typical methods that other frameworks use to retrieve and manipulate graphical interface objects, and does so in a type safe way. Finally, LIFT transparently deals with synchronization between the GUI event thread and the main thread running the tests.

2. RELATED WORK

Testing applications has always been an integral part of the software development process. Unit testing is one type of testing where the programmer writes many tests for one unit (usually a class or module), and each one verifies that one aspect of the unit is working correctly and behaving as intended. There are many Java libraries available to do unit testing, with JUnit being the most popular \[9\]. Unit testing has become very popular, especially with the introduction of Test Driven Development (TDD). In TDD, tests are written before the corresponding application code is written, and tests serve as a specification of the intended behavior. Then the corresponding code is written and refined until all of the tests pass. This is an effective technique, but can be difficult to do with applications that have a GUI. Testing GUIs requires being able to programmatically access the on-screen components along with a way to manipulate those components.

Since version 1.3, the Java language has provided the `java.awt.Robot` class to simulate user input. It allows the user to programmatically manipulate the mouse and keyboard to interact with a running program. Beyond these basic capabilities, `java.awt.Robot` provides little support for testing. In particular, it does not provide a way to access user interface components. In addition, the mouse simulation methods take specific location parameters. If the program’s interface changes, the tests would need to be rewritten to reflect the new locations of the components. The brittleness this introduces makes raw tests using only `java.awt.Robot` difficult to write and cumbersome to maintain as the underlying program evolves.

To overcome these challenges, several GUI testing frameworks for Java have already been created. These frameworks tend to fall into one of two categories: those where tests are created by recording user actions (recorders), and those where tests are created by writing code (coders). Marathon \[4\] is an example of the recorder type. After telling Marathon to start recording, the user performs the actions necessary to test part of his or her program. Marathon converts the actions into a python or ruby script that can then be run later to test the program. While recording can make it easier to define tests, it also necessitates that the test scripts be...
re-recorded if the program interface changes, just like with java.awt.Robot. In addition, because the test scripts are in either Python or Ruby, it requires the programmer to learn at least one language in addition to Java. Finally, because recording interactions requires the program interface to be present, these types of frameworks require that at least the user interface portion of the program be written before testing can start, making TDD more difficult.

UIspec4J [6] and jfcUnit [3] are both examples of the coder type of GUI testing framework. Rather than recording program interactions, when using these frameworks the programmer writes test methods that interact with his or her program. They provide a way to retrieve references to components and a way to manipulate those references or the graphical interface. Unlike recorder frameworks, testing can begin at the start of development, as with Test-driven Development. However, both frameworks are fairly complex, especially for new or student programmers.

Some frameworks combine these two approaches. Abbot [1] is one example, and is arguably the most popular GUI testing framework currently available for Java. Users can use the Abbot libraries to programatically run their tests, or use Abbot’s sister software Costello to record Abbot-compatible tests. Like UIspec4J and JFCUHunit though, Abbot is much too complex for new programmers.

LIFT is an example of the coder type of framework. It is based on enhancements made to Abbot for use with the objectdraw [5] student graphics library. Objectdraw is a framework that allows students to create simple graphical applications with geometric shapes on a canvas. Thornton, et. al. added support for testing these graphical applications [10]. After this proved successful, the testing support was then ported to the ACM Java Task Force (JTF) library [2], another popular educational graphics framework. LIFT was created during this porting process when we realized that many of the same techniques used for testing objectdraw and JTF applications could also be used with Swing applications. This would benefit many more people, as objectdraw and JTF are only used in education, while Swing is used by a large portion of the Java community.

3. USING LIFT

In undergraduate courses, many textbooks and instructors using JUnit employ v.3.x-compatible tests, since the 3.x API is simpler for novices to understand and does not require using annotations. Typical JUnit (v.3.x) tests are written by extending the junit.framework.TestCase class. This class provides test running infrastructure and several types of assert statements for testing objects. Programmers write methods to test their code, usually along with start up and tear down methods that are run before and after each test method.

LIFT provides the class GUITestCase, an extension of junit.framework.TestCase, that adds support for GUI testing. By extending GUITestCase rather than TestCase, the programmer gains access to all of the LIFT methods for use in writing tests.

3.1 Component Retrieval

In Abbot, the standard way to retrieve a GUI component is to create an Abbot Matcher that describes the component you are looking for. Figure 1 shows an example of a Matcher that finds a JButton with the text “OK”.

```java
button = (JButton)getFinder().find(
    new Matcher()
    { public boolean matches(Component c) {
        return c instanceof JButton && ((JButton)c).getText().equals("OK");
    }});
```

Figure 1: Using an Abbot Matcher.

```java
button = getComponent(
    JButton.class, Filter.textIs("OK"));
```

Figure 2: Using the generic version of getComponent().

A Matcher like this needs to be written every time a different component needs to be retrieved. Note that defining a new matcher typically requires declaring an anonymous inner class containing an in-line method definition, as well as multiple type casts. This style of programming is common for Swing programmers, but is not covered in most introductory Java courses and can be challenging for programmers who are still learning how to write GUI programs.

Even though not all matchers are complex, writing a Matcher can be very tedious. LIFT improves upon this by providing a simpler set of Matcher-like objects that we call Filters. The Filter class contains a set of static methods that return a single component that match the Filter. For example, the Filter.*textIs(String text) method returns a Filter that will only match components whose text contains the string passed into the method.

Along with Filters, LIFT also provides several methods analogous to the Abbot find() method that take a Filter as a parameter and return one or more components that match the Filter. The main one of these is called getComponent()—a generic method that takes both a component type and a filter. The Abbot matcher code in Figure 1 can be rewritten using LIFT as shown in Figure 2.

This templated version is type-safe and guarantees that the returned object is of the right type. Not only does this make the code more robust, it also makes casting the returned object to the correct type unnecessary.

The getComponent() method is intended to find a single, unique component determined by the given filter. If no component or multiple components exist that match the Filter, an appropriate exception is thrown. LIFT also provides getFirstComponentMatching() and getAllComponentsMatching() for when there may not be a unique match. The getFirstComponentMatching() method returns one (arbitrary) match if multiple components conform to the filter, or null if none are found. The getAllComponentsMatching() method returns a (possibly empty) List of matching components. Both methods are generic on the type of component being sought.

Along with textIs(String text), LIFT provides many other filters based on commonly used component attributes. Some of these include:
Matcher

button = getComponent(
  JButton.class,
  "OKButton"));

button = getComponent(
  JButton.class,
  where.textIs("Next")
    .and.enabledIs(false)
    .and.hasFocusIs(true));

button = (JButton)getFinder().find(
  new Matcher() {
    public boolean matches(Component c) {
      return c instanceof JButton
          && ((JButton)c).getText().equals("Next")
          && c.isEnabled() == false;
    }
  });

Figure 4: Specialized getComponent() for named components.

• nameIs(String name)
• enabledIs(boolean enabled)
• widthIs(int width)
• parentIs(Component parent)
• ancestorIs(Component ancestor)
• hasFocusIs(boolean focus)

When used in conjunction with the setName(String name) method shared by all Component objects, the nameIs(String name) filter can greatly simplify testing. If all of the components that will need to be used during testing are given unique names in the GUI code, the nameIs(String name) filter can then be used to retrieve them when necessary. This turns out to be so useful in practice that an overloaded version of getComponent() is provided in LIFT that takes a String parameter and internally uses the nameIs(String name) filter to retrieve the component. Returning to our previous example, if the button had been named as shown in Figure 3 it could then be retrieved with the code in Figure 4.

3.2 A Filter Mini-language

The filter system used by LIFT allows programmers to easily create objects much like Abbot Matchers, but without all of the overhead and details usually required by that tool. However, the filters each only test for a single component attribute, and often more specificity is needed. Abbot Matchers allow you to make the matching criteria as specific as desired. For example, Figure 5 shows how to create a Matcher that returns a button with the text “Next” that is currently not enabled.

button = getComponent(
  JButton.class,
  where.textIs("Next")
    .and.enabledIs(false)
    .or(where.textIs("Previous")
         .and.hasFocusIs(true)));

Figure 5: A more complex Abbot Matcher.

Figure 7: A somewhat convoluted (though legal) LIFT filter.

To enable creating this type of filter in LIFT, a mini-language of sorts was created. Figure 5 shows how to create a Filter that acts like the Abbot Matcher in Figure 5.

All “sentences” in this language begin with the keyword where, which is a static field that refers to a filter generator object. The name of this field was chosen to provide for more readable expressions, as an alternative to explicitly using the name of the Filter class, for example. The where field is declared in the base class used for writing GUI tests, called GUITestCase (a subclass of JUnit’sTestCase class).

For this mini-language, objects like where that generate filters are instances of a special class called Operator—that is, they are filter operators used to create and combine filters. The where object provides access to the various primitive filter-creating methods, which can be combined using and, or, and not operators. Parentheses can also be used to group filters together if necessary. The example shown in Figure 7, while unlikely to be necessary, is perfectly legal. In this example, LIFT would search for a disabled JButton with the text “Next”, or a JButton that currently has the focus and has the text “Previous”.

3.3 GUI Interactions

The first step in testing GUIs is retrieving references to on-screen components. The getComponent() method along with the composable filter mini-language provides a convenient mechanism for retrieving such references. Once the components are retrieved, one must then interact with them in some way. This could mean clicking on buttons, typing in text boxes, selecting things from combo boxes, or other GUI interactions. With Abbot, this is accomplished through the use of Tester objects. There are different Testers for different types of Components. The programmer must first create a Tester by passing the Component they wish to manipulate to the getTester() method. He can then call methods like actionClick() or actionSelectItem() on that tester to manipulate the GUI. This is demonstrated in Figure 8.

LIFT uses the same mechanism conceptually, but does so internally, hiding all of the details from the programmer. Instead, the LIFT GUITestCase base class provides methods called click(), doubleClick(), mouseMove(), mouseDragFrom(), mouseDropOn(), and so on. Once a GUI
button = (JButton) getFinder().find(  
   new Matcher() {  
      public boolean matches(Component c) {  
          return c instanceof JButton  
              && ((JButton) c).getText()  
                  .equals("OK");  
      }  
   });

tester = (ButtonTester) ComponentTester  
   .getTester(button);  
tester.actionClick(button);

Figure 8: Creating and using an Abbot Tester.

button = getComponent(JButton.class, where.textIs("OK"));
click(button);

Figure 9: Figure 8 rewritten using LIFT.

3.4 Synchronization Issues

When testing a GUI-based program, there are typically (at least) two different threads running. The first is the main thread itself, which is where the test methods run. The second is a special thread called the event dispatching thread (EDT), which handles all GUI interactions. The EDT is a special background thread that is a standard part of Java AWT and Swing GUIs—it is the sole thread that can update or change the visual appearance of all GUI components. Students rarely see these two threads, since most student-written GUI programs do nothing with the main thread other than to create the application’s main window.

Once the GUI is created, all event processing is handled by the EDT. Swing methods are not thread-safe, in fact, and keeping all event processing on the same thread naturally serializes their invocations. Problems can arise if the programmer does not know how to deal with the two separate threads. For example, if one is testing a button that increments a counter, the code might look something like that in Figure 10. In this case, any listener attached to the button will be executed in the EDT, while the assert statement will be executed by the main thread as part of the test method. If the button’s listener(s) takes a long time to execute, the assert statement may be called before the button code has a chance to update the label, resulting in a failed assertion even if the label does eventually get updated correctly. In order to prevent this, all LIFT methods use Abbot methods internally so all GUI interactions are executed on the EDT. In effect, LIFT interaction methods like click() post the necessary interaction event(s) for the GUI components to receive, and then block, waiting for those events to be completely processed. Even if a listener responding to one event triggers additional events, Abbott’s internals wait until the EDT’s queue of GUI events has emptied, indicating that the GUI has once again become idle. Only then does the LIFT method return back to the test, allowing the next statement in the test method to be executed.

This “post and wait” strategy ensures that the main thread (executing the test) and the EDT (processing GUI events) are transparently synchronized in most cases. Performing this synchronization prevents race conditions between interaction events in test cases and other object manipulation or inspection actions carried out on the main thread as part of each test. In effect, students can safely use a “naive, single-threaded mental model when writing their tests, and typically do not need to take the EDT into account when writing their tests.

However, there are still some situations where the “post and wait” strategy alone is insufficient. Several Swing modal dialog components provide helper methods that block the calling thread while the dialog is presented, the user interacts with it, and a selection is made. This is a natural way to use a modal dialog, since no other part of the application should be accessible to the user while the modal dialog is visible. When modal dialogs like this are created from the EDT in response to GUI interactions, the “post and wait” strategy works perfectly. However, problems arise if such a modal dialog helper method is invoked directly on the main thread as part of a test case.

For example, suppose a student is testing a Player class with a makeAMove() method. In some situations, perhaps no legal moves are available, and the student has written makeAMove() to use a JOptionPane to present an error message in this case. One test for this method might consist of one or more method calls to alter the program state so that no moves are possible, a call to makeAMove(), a confirmation that a JOptionPane appeared, and finally a click on the “OK” button of the error message. Figure 11 shows such a test method.

```java
player.makeAMove();  // no moves possible
// confirm the JOptionPane opened
assertNotNull(
   getComponent(JOptionPane.class));
// click the OK button
click(getComponent(
   JButton.class, where.textIs("OK")));
```

Figure 11: Incorrectly testing a method producing a JOptionPane.
Figure 12: Correctly testing a method producing a JOptionPane.

Unfortunately, showing a JOptionPane inside makeAMove() is a blocking action, so that makeAMove() will not return to its caller until the dialog has been closed. Because of the way modal dialogs work in Swing, if the call to makeAMove() is placed directly in the test code (which is executed on the main thread), this will block the main thread until the modal dialog is closed. However, that means that no statements following the makeAMove() call in the test method will be able to execute, and the test method would never terminate (unless a person intervened and clicked the button on the screen manually).

For situations like this, LIFT provides a method named callGUIIOMethod(). This generic method takes an object, a method name, and a list of parameters. It then calls the named method on the object with the specified parameters using reflection, but does so from the EDT rather than the main thread, using the “post and wait” strategy to allow the GUI to process any pending events before it returns back to the test method. callGUIIOMethod() must be used for methods on student-written classes that indirectly cause GUI events. This allows modal dialogs like JOptionPane to be handled correctly, and also ensures that any other Swing actions are carried out on the EDT. Figure 12 shows the makeAMove() example rewritten using callGUIIOMethod(), which now works correctly. The EDT waits for the modal dialog to complete, while the remainder of the test method continues to run in the main thread.

5. EVALUATION

At this point, no formal evaluation of LIFT has been conducted. However, a formal study of the objectdraw-based predecessor to LIFT was performed and the results are summarized here. Thornton et al. provide further information [10].

Three semesters of data were analyzed in the study. Students in the first semester wrote and tested non-GUI programs. Students in the second semester wrote GUI programs using the objectdraw library, but did not have to test the GUI portion of their programs. Students in the third semester wrote GUI programs using objectdraw along with a complete set of tests, including tests for their GUIs. The first finding was that GUI assignments require writing slightly more code than non-GUI assignments. However, more code does not necessarily mean more work. GUI tests usually required more lines of code than their non-GUI counterparts, but are often conceptually easier to grasp. “In a GUI assignment, students have an instant visualization of what is happening, which results in faster development of the program and the test cases.” The study also revealed that students working on GUI assignments typically start and finish earlier, despite spending more time overall than on non-GUI assignments. Finally, the study found that those students who did not have to test their code (the first semester) did progressively worse as the semester went on, while the performance of those who were testing from the start (the third semester) improved throughout the semester.

Along with a quantitative analysis, the study also included student survey results about their experiences testing GUIs. The survey results indicate that students thought the graphical assignments were more interesting and fun, and that the testing helped them find more bugs. Lastly the students agreed that testing their GUIs was beneficial.

Our classroom experiences with LIFT in a CS2-level course where students learn the basics of Swing GUI programming are promising. The simplicity of the LIFT test methods does simplify the process of writing test cases and students
understand them as easily as other library methods. Assignments ranging from simple panels containing buttons all the way up to full blown 2D-shape editing/drawing programs have used LIFT successfully. When students follow model-view-controller (MVC) design principles carefully, they find that testing their model separate from the GUI is easy, and then writing unit tests for the GUI is simplified because of the separation of concerns and responsibilities the design pattern imposes.

Also, having the capability of automatically running unit tests on GUIs has been a major help in assignment grading. Course staff no longer have to manually download and run each student’s work. Further, by using Web-CAT, students can get immediate feedback on the quality of their solution and thoroughness of their testing right away, which often reveals hidden bugs. As an example, we have used a 2D-shape-based graphical editor as an assignment in our CS2 course, both with and without LIFT. Without using automated tests (or requiring students to write their own test cases), many students would fail to consider the different possibilities when using the mouse to drag a rubberbanded shape outline to define a shape’s position and size. Students nearly always handled the typical case of an initial click on the upper left corner of a shape’s final location, followed by a drag down and to the right. However, many students failed to consider an initial click followed by a drag up and to the left, and others implemented their logic incorrectly. However, without explicit tests, these bugs often went undiscovered until grading, and some no doubt slipped through then as well. However, when LIFT was used in the class, students received feedback on these failures before the assignment deadline because they had to test their own code, and all submissions were exercised to the same degree of rigor before course staff manually graded the assignment. Anecdotally, course staff felt that this reduced the number of latent defects in student solutions, because many more errors were detected and reported back to students through the use of LIFT tests.

At the same time however, there have been some negatives to using LIFT in the classroom. Three notable issues arose, all of which center around student perceptions. In our CS2 course, students were required to write their own LIFT-based tests, and their submissions were also checked against instructor-written LIFT tests. First, we found that some students reacted negatively to being required to thoroughly test their own code. Such students typically complained that they “knew” their code was correct, and viewed writing detailed tests (say, to cover all the possibilities for dragging out rubber-banded shapes) as unnecessary effort—a perception that is contrary to the empirical evidence. Other students felt that writing tests for GUI interfaces was more tedious that writing tests for text-based programs because there were more details to check, an observation reported elsewhere [10]. Finally, GUI tests run close to “human” speed because the interface and all of its changes are fully rendered on the screen and all mouse interactions are faithfully simulated. As a result, running GUI tests can take longer. For students who perceived one or more of these issues, their perception of the value of GUI testing—and, by extension, the value of the LIFT library—was somewhat reduced.

6. FUTURE WORK

So far, LIFT has been used successfully in our introductory Java courses. Though no formal evaluation has been performed on the current generation of LIFT for Swing interfaces, students found the LIFT API to be straightforward and easy to use. Future work will include a more formal evaluation to determine what impact LIFT has on student work, including final score, amount of code testing coverage, and time spent writing code. In addition, we plan to survey students to determine which features in LIFT they liked and did not like, along with possible areas for improvement.

7. ACKNOWLEDGEMENTS

This work is supported in part by the National Science Foundation under grant numbers DUE-0618663 and DUE-0635394. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES