Automated GUI Refactoring and Test Script Repair (Position Paper)

Brett Daniel[1], Qingzhou Luo[1], Mehdi Mirzaaghaei[2]
Danny Dig[1], Darko Marinov[1], Mauro Pezzè[2]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{+, qluo2, dig, marinov}@illinois.edu
[2]Faculty of Informatics, University of Lugano, Switzerland
{mehdi.mirzaaghaei,mauro.pezze}@usi.ch

ABSTRACT

To improve the overall user experience, graphical user interfaces (GUIs) of successful software systems evolve continuously. While the evolution is beneficial for end users, it creates several problems for developers and testers. Developers need to manually change the GUI code. Testers need to manually inspect and repair highly fragile test scripts. This is time-consuming and error-prone.

The state-of-the-art tools for automatic GUI test repair use a *black-box* approach: they try to *infer* the changes between two GUI versions and then apply these changes to the test scripts. However, inferring these changes is challenging.

We propose a *white-box* approach where the GUI changes are automated and knowledge about them is reused to repair the test cases appropriately. We use *GUI refactorings* as a means to encode the evolution of the GUIs. We envision a smart IDE that will record these refactorings precisely as they happen and will use them to change the GUI code and to repair test cases. We illustrate our approach through an example, discuss challenges that should be overcome to turn our vision into reality, and present a research agenda to address these challenges.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Design

Keywords: Graphical user interfaces, GUI refactoring, automated GUI testing, GUI maintenance

1. PROBLEM AND MOTIVATION

Graphical user interfaces (GUIs) are an indispensable part of today's software. Most GUIs are designed using rapid prototyping [15], which continuously evolves the GUI in a quest to improve the overall user experience and keep users engaged. The rate of change in GUIs can be even higher than in the core domain logic. For instance, popular web-based systems like email and social media, as well as desktop applications like office suites, routinely revamp their GUIs, while the core underlying system stays relatively the same.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ETSE '11, July 17, 2011, Toronto, ON, Canada
Copyright 2011 ACM 978-1-4503-0808-3/11/07 ...$10.00.

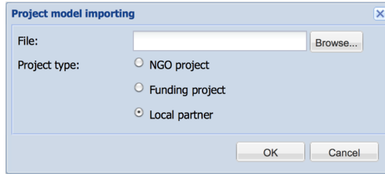
Testing GUI-based applications is usually different from conventional software because these applications require interaction between the user and the application. Many software organizations rely on manual GUI testing to provide user interactions, which is error-prone, tedious, and time-consuming. Automated GUI testing often involves testing frameworks that simulate user interaction with GUIs through test scripts [6, 10, 14, 17, 21, 22]. Because GUI-based applications are complex and have a large number of interaction sequences, writing the test scripts is non-trivial, and there is a high incentive to reuse these scripts. The fact that GUI-based applications change frequently exacerbates the problem of GUI testing. Maintenance of GUI test scripts is tedious and error-prone, and can be even more costly than manual GUI testing [8].

For an example of real GUI evolution, consider Figure 1 that shows screenshots corresponding to revisions 844 and 845 of the Sigmah project [19], described as follows [18]: "Sigmah is an open source, web-based project monitoring and management system for the UN, NGOs ..." The particular dialog box is used for loading a project document of a given type. In revision 844 these three types are presented as three radio buttons, and in revision 845 these three types are presented as a listbox with three items. As a running example to discuss our approach, we will use our own simplified code, written with the SWT toolkit, instead of the actual Sigmah code because the latter was rather repetitive (and even had a copy-paste bug in revision 844).

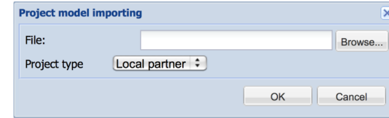
Figure 2(a) shows an automated test script, written with the SWTBot testing framework [21], for the window with the radio buttons. This script enters a document name, clicks on one of the radio buttons, and clicks on the `OK` button. Figure 2(b) shows the corresponding test script for the window with the listbox. Note the change that replaces a click on the radio button with a selection of the corresponding item from the listbox. After the developers (manually) changed the GUI, the test script also needs to be updated.

The example change is an instance of *GUI refactoring* [13, 16], which we define using Model-View-Controller [11]:

Definition: *A GUI refactoring only changes the visual aspects of a GUI (and correspondingly the view and controller parts of the code that implement the GUI) but not the way it behaves (i.e., not the underlying model).*



(a) Sigmah revision 844 (three Radio buttons)



(b) Sigmah revision 845 (one ListBox with three items)

Figure 1: Evolution of the Sigmah GUI from Radio buttons to ListBox

```
1 public void testRadioButton(Display display) throws Exception {
2     bot = new SWTBot();
3     bot.text(0).setText("local.txt");
4     bot.radio("Local partner").click();
5     bot.button("OK").click();
6     assertTrue(bot.label("Local partner").isVisible()); }
```

(a) Test local partner for Radio buttons

```
1 public void testDropdownMenu(Display display) throws Exception {
2     bot = new SWTBot();
3     bot.text(0).setText("local.txt");
4     bot.comboBox(0).setSelection("Local partner");
5     bot.button("OK").click();
6     assertTrue(bot.label("Local partner").isVisible()); }
```

(b) Test local partner for ListBox

Figure 2: An example SWTBot test, updated from Radio buttons to ListBox

We can call the example refactoring ReplaceRadioButtonsWithListBox. This refactoring changes only the view (from Radio buttons to ListBox) and the controller (the events) but does not change the underlying model: the old and the new GUI feed from the same array of three `Strings`. We elaborate on this example in Section 3.

Researchers have proposed different approaches to automate updating of GUI test scripts, e.g., [9, 10]. All these approaches infer changes to the GUI between different program versions. We call these approaches *black-box* because they focus on the GUI widgets and not on the GUI code. Moreover, even when considering the GUI code, they need to *infer* the changes. Because such inference is challenging and inaccurate, the automatic updating of test scripts still requires a lot of user involvement and confirmation.

The most widely practiced development process for GUI evolution indeed remains labor-intensive: developers manually change GUIs (both the view that displays the data and the controller that processes the events), and testers manually change test scripts to match GUI changes. This is the case even when the changes are GUI refactorings that do not modify the underlying logic. Contrast this with the high level of automation when refactoring the core domain logic code along with its unit tests: developers use automated refactoring tools provided in IDEs to carry out the changes in both the domain logic and the test code.

2. PROPOSED RESEARCH AGENDA

We hypothesize that *automated GUI refactoring* could improve software productivity in GUI evolution. Getting more information from GUI developers about GUI code changes would help in updating GUI test scripts. We propose an approach that (i) makes it easier for GUI developers to make changes, (ii) automates recording of these changes, and (iii) automates applications of corresponding updates to the GUI test scripts. We envision automated support in the IDEs and GUI design tools that would allow developers to apply GUI refactorings in the future, much like they can apply code refactorings now. We refer to our approach as *white-box* because it is aware of the changes made to the GUI code and carries them over to the GUI test scripts.

To realize our vision, we propose these research steps.

Study GUI Evolution: We will empirically study how GUIs evolve to identify the most common types of changes. Our previous study on evolution of APIs for object-oriented components found that over 80% of changes are API refactorings [4], which only change the names or locations of API elements but not the way they behave. Similarly, we expect a number of GUI changes to be GUI refactorings. Through our example, we already described the ReplaceRadioButtonsWithListBox refactoring. A related refactoring would be ReplaceTwoRadioButtonsWithCheckBox. For instance, a GUI change from Eclipse 3.5 to 3.6 replaced two radio buttons with a checkbox in the dialog for creating new Java projects. While some GUI refactorings have been briefly described [13], we are not aware of an extensive catalog of GUI refactorings. As an outcome of our work, we expect to provide such a catalog, similar to catalogs of object-oriented API refactorings [5].

Automated Support for Refactoring GUI Code: We will develop techniques and tools that GUI designers and developers can use to easier evolve their GUIs. We need to extend current IDEs to support GUI refactorings. For example, rather than manually editing the code to replace radio buttons with a listbox, a developer could select a group of radio buttons and click on an automated refactoring, ReplaceRadioButtonsWithListBox. As for all refactorings, the IDE would perform an analysis (potentially not only static but a hybrid dynamic and static analysis) to identify whether the refactoring can proceed, and if so, it would perform an appropriate transformation of the code. This transformation includes (i) changing the code related to the *view*, i.e., code that creates GUI widgets (e.g., creating `ListBox` instead of `Radio` buttons), (ii) changing the code related to the *controller* identity, i.e., types/names of events being handled (e.g., `OnSelect` instead of `OnClick`), and/or (iii) changing the code of event handlers (e.g., updating the communication with the *model* but not the model itself).

The resulting IDE extension would not only perform the refactoring changes but also record the refactorings conducted by the GUI developer by capturing the *mapping* between original and modified GUI widgets (i.e., objects and methods used in the widgets). Our previous contribution to the Eclipse IDE [3] extends its refactoring engine with an infrastructure for recording refactorings, along with their parameters. We will build on this infrastructure to record GUI refactorings as well.

Automated Repair of GUI Test Scripts: Our main motivation is to make it easier to update GUI test scripts:

making it easier for developers to change GUIs through automated refactorings is a way to "trick" the developers into providing mapping information necessary to update GUI test scripts. Rather than attempting to infer the changes that developers made, we would have a detailed recording of their changes. We will develop techniques and tools that can systematically apply corresponding changes to the existing test scripts, specifically to the references to the GUI objects affected by GUI modifications. For each refactoring (and some testing framework), we would synthesize an appropriate transformation that can automate update of test scripts. For example, replacing a set of radio buttons with a listbox in the GUI (when using SWTBot for tests) requires changing some `click` events into `setSelection` events. Note that it is non-trivial to find which events should be changed (because many `click` events can be on unchanged widgets such as the `OK` button in our running example); we anticipate building on the state-of-the-art results in static analysis for GUI test scripts [6] (to identify type of GUI objects) and likely combining them with a dynamic analysis (to identify precise object ids). The key in enabling us to perform precise, automated updates of test scripts is having precise *mapping* from GUI evolution. An additional advantage of our approach is that it need not examine all the test scripts to understand if they are broken; instead, it can refactor only the test scripts that are affected by the GUI changes.

Note that we propose to automate only (widely used) GUI refactorings, which by definition preserve behavior. There are many other GUI changes that do not preserve behavior. As a trivial example, consider adding a fourth radio button to the three existing buttons. As a more involved example, consider adding an error message when a non-existent file name is entered. It is unclear how to automatically update the GUI code for such changes in general, the same way that the existing automated code refactorings do not attempt to automate arbitrary non-behavior-preserving code changes.

3. EXAMPLE

To illustrate the challenges and potential solutions in our approach, we elaborate on the example ReplaceRadioButtonsWithListBox refactoring. Figures 3 and 4 show sample code that implements an evolution scenario similar to the one in Figure 1. Figure 3 is the common code for both versions to create the main window.

```
1  Display display = new Display(); Shell shell = new Shell(display);
2  // ... code to create a file selector
3  String[] projectType = new String[]{
4      "Local partner", "NGO project", "Funding project"};
5  createWidgets(shell, projectType);
6  shell.pack(); shell.open();
```

Figure 3: Common SWT Code for Both Versions

Figure 4(a) shows the code that creates Radio button widgets and sets up event handlers for them. (In this simple example, the handlers only record the selected item so that it can be printed later.) Figure 4(b) shows the code that creates a list box, called a `Combo` (of `DROP_DOWN` kind) in SWT, and sets up an event handler for it. The differences are highlighted, and they are of two types.

The difference in the *view* is that the radio buttons are replaced with a list box. For this example, it means that the creation of the widget objects is not done in the loop but before the loop (only one widget is created), and the type is changed from `Button` to `Combo`. Similarly, the setting of event handler (`addSelectionListener`) is not done in the loop but after the loop. Note that in general a group of radio buttons could have a different handler for each button. It is a precondition of the ReplaceRadioButtonsWithListBox refactoring that all buttons have the same handler (because that one would be reused for the list box). This is indeed the most common case, but a refactoring tool would need to check for this and warn the programmer if the precondition is violated.

The main body of the loop is setting the values for either the text of radio buttons or the items in the list. In this example, the change would be relatively easy to automate. (The implicit assumption here is that the refactoring tool is specialized for a toolkit and knows how to map appropriate methods such as `setText` to `add`.) However, note that the original code could have been written in a different way: rather than having a general loop, it could have unrolled the loop for radio buttons for the three values in `items`. It would make the automation of the change much harder (and likely infeasible) for the refactoring tool. However, we expect that the developers would realize the benefit of the GUI refactoring tool and appropriately design their code to enable easier future refactoring.

The difference in the *controller* is in the event `widgetSelected`. In this particular case, the name of the event is the same. However, in general, this name could change (e.g., it could have been the case that for list box the event is called `itemSelected`), so the refactoring tool would need to know how to map the events. Moreover, the handler body need to be updated. In this case, the body was simply getting the text from the selected item. The only change is from `Button` to `Combo`. Again, the refactoring tool would need to apply an appropriate mapping.

After updating the code, our refactoring engine would update the GUI test scripts. Figure 2 shows SWTBot scripts before and after the refactoring. The tool would need to identify where the scripts sent events to the changed objects and would need to appropriately send (different) events to the new objects. In this example, the change is in Line 4 of Figure 2(a) which sent `click` to the `Local Partner` radio button. To refactor the test case, the refactoring tool would need to call `setSelection` with two parameters: the (new) list box and the selected item. The first parameter may be acquired automatically from the mapping of old and new widgets, or could be provided by the user. The second parameter is the value of the selected item, which is `Local partner` in this case.

From this simple example we can see several challenges in refactoring GUIs and GUI test scripts. While traditional code refactorings are derived from the language semantics, GUI refactorings additionally depend on the libraries and toolkits being used. We do not expect to support all toolkits but rather to focus only on one. Still, identifying whether a GUI refactoring can apply and what changes to make is highly non-trivial. We expect that a partially automated tool that requires some help from developers would still be valuable and make it easier to change GUI code than a completely manual approach. Last but not least, identifying where and how exactly to change GUI test scripts remains an open problem. We plan to pursue this line of research by working on many examples and generalizing from the experience. The running example presented in this paper is only the starting point.

```
1  private void createWidgets(Shell shell, String[] items) {
2      createWidgetsWithEvent(shell, items,
3          new SelectionAdapter() {
4              public void widgetSelected(SelectionEvent e) {
5                  selectedItem = ((Button) e.widget).getText();
6              } }); }
7  private void createWidgetsWithEvent(final Shell shell, String[] items,
       SelectionAdapter selectionAdapter) {
8      for (String i : items) {
9          Button b = new Button(shell, SWT.RADIO);
10         b.setText(i);
11         b.addSelectionListener(selectionAdapter); }
12     // ... code to create an OK button and display a message box
13 }
```

(a) Radio Buttons

```
1  private void createWidgets(Shell shell, String[] items) {
2      createWidgetsWithEvent(shell, items,
3          new SelectionAdapter() {
4              public void widgetSelected(SelectionEvent e) {
5                  selectedItem = ((Combo) e.widget).getText();
6              } }); }
7  private void createWidgetsWithEvent(final Shell shell, String[] items,
       SelectionAdapter selectionAdapter) {
8      Combo c = new Combo(shell, SWT.DROP_DOWN);
9      for (String i : items) {
10         c.add(i); }
11     c.addSelectionListener(selectionAdapter);
12     // ... code to create an OK button and display a message box
13 }
```

(b) Dropdown List

Figure 4: Evolved GUI code in SWT

4. RELATED WORK

There is a large body of research on maintaining GUI code and tests. Li and Wohlstadter [12] used dynamic information to show generated GUI widgets in a GUI editor and to help GUI editors to map source code to GUI views when code changes. Goderis et al. [7] use a declarative meta language to increase the maintainability of GUIs by finding and modularizing crosscutting concerns. Many researchers address the reverse engineering of user interfaces to support migration to modern GUI systems [1, 2]. For example, Abrams et al. [1] create a level of abstraction for GUIs with adding a layer of XML based GUIs. Staiger et al. [20] instead use static, whole-program analysis for reverse-engineering GUIs and extracting possible interactions between widgets.

Nagarajan and Memon [16] proposed an event-based profiling for refactoring GUI-based applications via changes in the GUI layout and removal of unused event handlers. Our proposal aims to provide a set of GUI refactorings for developers and to automate the entire process, both in GUI views and their corresponding controllers. Memon et al. proposed a technique that repairs obsolete GUI tests by modeling GUIs with event-flow graphs [14]. To help in understanding GUI test scripts, Fu et al. introduced an approach [6] to infer the type of widgets and map them to code components. Grechanik et al. offer an approach to identifying changes between two GUI applications and automatically maintaining test scripts by comparing the successive GUI trees [9]. Huang et al. used genetic algorithms to repair broken GUI test cases and to generate new test cases [10]. Our test repair proposal is built upon our proposed automatic GUI refactoring, and we believe that we can achieve higher accuracy in a more automatic way.

Acknowledgments The first author passed away on December 5, 2010. Brett contributed significantly to the initial discussions on GUI refactorings. Brett will be missed dearly by his family, friends, and colleagues.

We thank anonymous reviewers for comments on a previous version of this paper. This material is based upon work partially supported by the US National Science Foundation under Grant No. CCF-0746856.

5. REFERENCES

[1] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: An appliance-independent XML user interface language. In *WWW*, 1999.

[2] A. De Lucia, R. Francese, G. Scanniello, G. Tortora, and N. Vitiello. A strategy and an Eclipse based environment for the migration of legacy systems to multi-tier web-based architectures. In *ICSM*, 2006.

[3] D. Dig. *Automated Upgrading of Component-based Applications*. PhD thesis, UIUC, 2007.

[4] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM*, 2005.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[6] C. Fu, M. Grechanik, and Q. Xie. Inferring types of references to GUI objects in test scripts. In *ICST*, 2009.

[7] S. Goderis, D. Deridder, E. V. Paesschen, and T. D'Hondt. DEUCE : A declarative framework for extricating user interface concerns. *JOT*, 2007.

[8] M. Grechanik, Q. Xie, and C. Fu. Experimental assessment of manual versus tool-based maintenance of GUI-directed test scripts. In *ICSM*, 2009.

[9] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving GUI-directed test scripts. In *ICSE*, 2009.

[10] S. Huang, M. Cohen, and A. Memon. Repairing GUI test suites using a genetic algorithm. In *ICST*, 2010.

[11] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *JOOP*, 1988.

[12] P. Li and E. Wohlstadter. View-based maintenance of graphical user interfaces. In *AOSD*, 2008.

[13] M. Marinillo. *Professional Java User Interfaces*. Wiley, 2006.

[14] A. Memon. Automatically repairing event sequence-based GUI test suites for regression testing. *ACM TOSEM*, 2008.

[15] B. A. Myers. User interface software tools. *ACM TOCHI*, 1995.

[16] A. Nagarajan and A. Memon. Refactoring using event-based profiling. In *REFACE*, 2003.

[17] Selenium home. `http://seleniumhq.org/`.

[18] Sigmah home. `http://code.google.com/p/sigma-h/`.

[19] `http://sigma-h.googlecode.com/svn/trunk/`.

[20] S. Staiger. Static analysis of programs with graphical user interface. In *CSMR*, 2007.

[21] SWTBot home. `http://www.eclipse.org/swtbot/`.

[22] Q. Xie, M. Grechanik, and C. Fu. REST: A tool for reducing effort in script-based testing. In *ICSM*, 2008.