

Test-Driven GUI Development with TestNG and Abbot

Alex Ruiz and Yvonne Wang Price, *Oracle*

Testing GUIs, although difficult, is essential to make applications safer and more robust. Several recommendations and practices can simplify test-driven GUI development for Java Swing applications.

Testing GUIs can make the entire system safer and more robust. Any GUI, even one providing only the simplest capabilities, likely encloses some level of complexity. The more user-friendly a GUI is, the more complexity it might be hiding from the user.¹ Any complexity in software must be tested because code without tests is a potential source of bugs. A well-tested application has a greater chance of success.²

GUI testing is also essential during application maintenance. During this phase, code might be refactored frequently to improve its design, and this code often encloses great portions of the user interface. Having a solid test suite that includes the GUI code can give us confidence that we're not inadvertently introducing bugs.

In spite of test-driven development's reported benefits, GUI development has been slow to include it as a core practice.¹ Although testing GUIs is difficult, we've developed some recommendations and practices that can simplify test-driven GUI development for Java Swing applications.

Testing GUIs is hard

GUIs are complex pieces of software. Testing their correctness is challenging for several reasons:

- Tests must be automated,¹ but GUIs are

designed for humans (not computer programs such as automated tests) to use.

- Conventional unit testing, involving tests of isolated classes, is unsuitable for GUI components. In GUI terms, a "unit" involves cooperation of more than one GUI component, which can itself consist of more than one class.
- GUIs respond to user-generated events. To test GUIs, we need a way to simulate such events, wait until they have been broadcast to all listeners, and then check the simulated action's result as the GUI would show it to the user.¹ Writing code that simulates user interaction with GUIs can be tedious and error prone.
- The room for potential interactions with a GUI is huge. At any time, users can click on any object in a window, moving among completely unrelated features. So many po-

```
private JButton findButtonWithText(String text) {
    for (Component c : getComponents()) {
        if (c instanceof JButton) {
            JButton button = (JButton) c;
            if (text.equals(button.getText())) return button;
        }
    }
    return null;
}
```

Figure 1. Manually finding a button.

```
Point point = button.getLocationOnScreen();
Robot robot = new Robot();
robot.setAutoWaitForIdle(true);
robot.mouseMove(point.x + 10, point.y + 10);
robot.mousePress(BUTTON1_MASK);
robot.mouseRelease(BUTTON1_MASK);
```

Figure 2. Using the java.awt.Robot to simulate a user clicking on a button.

- Changes in the GUI's layout shouldn't affect robust tests.
- Conventional test coverage criteria, such as "90 percent coverage of lines of code," might not catch all the user interaction scenarios. In GUI testing, the amount of code being tested is still important, but it's equally important to focus on the number of possible states in which you're testing each portion of the application.

GUI-testing methods

Two common methods for GUI testing are the record/playback script technique and programmatic GUI tests. In the first method, the test interacts with an existing GUI, and all the user-generated events are recorded in a script. Developers can later replay the script to recreate user interactions for a particular scenario. However, this technique contemplates testing existing GUIs, and our interest is to write tests before any GUI code is written.

The record/playback method can be useful when you need a test suite for an existing GUI in the shortest possible time. Its major weakness is its expensive maintenance. Any change in the application requires rerecording all the affected test scenarios. At the same time, recording all test scenarios can create duplicate test code when testing similar scenarios. Recorded scripts are often long and written in proprietary lan-

guages lacking object-oriented (OO) language features. Modularization of repetitive actions is labor intensive and error prone and typically requires learning a proprietary language.

On the other hand, programmatic GUI testing lets us apply the test-first practice that TDD advocates. Using an OO language, we can write tests that are both easier to maintain and more adjustable to application changes. Programmatic tests can leverage the benefits of OO languages and design patterns to create modularized, reusable, and maintainable code.³

Developers choosing a popular, mature OO language can also benefit from using a feature-rich integrated development environment. Modern IDEs such as Eclipse (www.eclipse.org) and IntelliJ IDEA (www.jetbrains.com/idea) can improve productivity and lower maintenance costs by making boring, error-prone tasks (such as refactoring) quick and trivial.

Programmatic tests are easier and cheaper to maintain than tests created with the record/playback technique. They're also harder to write. Creating a test programmatically requires a consistent way to find specific GUI components in the application being tested, simulate user interaction on such components, and verify that the final output matches the expected one.¹

The open source arena provides a variety of Java GUI testing tools. Libraries such as Jacareto (<http://jacareto.sourceforge.net>) and Marathon (www.marathontesting.com) are limited to the record/playback testing technique. Other libraries, such as Jemmy (<http://jemmy.netbeans.org>) and UISpec4J (www.uispec4j.org), support programmatic GUI testing exclusively. Only Abbot⁴ (which we'll discuss in more detail later) and jcf-Unit (<http://jfcunit.sourceforge.net>) support both GUI testing techniques.

A naive approach for programmatic Swing GUI testing

The following is a simple but inflexible approach for GUI testing. The first step is to manually find the GUI component of interest. The code fragment in figure 1 shows how to find a `JButton` containing the text "Apply."

Once the component has been found, the `java.awt.Robot` can be used to control the mouse and keyboard, simulating user interaction with such components. For example, figure 2 simulates a user clicking on the previously found button.

Although the code in figure 2 makes it possi-

ble to simulate user interaction, the solution is too low-level and isn't reliable. There's a dependency on exact component position, which can (and will) change over time. To facilitate GUI testing, we need a better, higher-level `Robot` class to drive our GUIs.

Introducing Abbot (and Costello)

Abbot (<http://abbot.sourceforge.net>) is a Java library for testing Swing GUIs that supports both the record/playback and programmatic GUI testing styles. It includes the script editor Costello for creating test suites for existing GUIs and a Java API that can help us create programmatic tests.⁴

Costello provides record/playback functionality for testing existing GUIs. User interactions are saved in XML-based scripts (which are also hand-editable). Unlike many record/playback tools, Abbot script-based tests are robust because they use numerous attributes to dynamically identify a GUI component without depending on any positional attributes.⁴ Changes in the GUI layout normally won't break Abbot script-based tests.

Abbot's real power comes from its Java API. The Abbot API consists of two building blocks, which significantly simplify creating programmatic GUI tests:

- `ComponentFinder` lets us get a reference to a GUI component based on search criteria encapsulated in a `Matcher`. Abbot provides automatic, reliable component lookup without adding clutter to the code (for example, getter methods such as `getSaveButton`) to get a handle on GUI components. We generally don't need to worry that minor component changes will break the tests.
- `Robot` is the root of all `ComponentTesters`. Abbot provides `Robot`-like objects that know how to simulate user actions on GUI components. Although the `java.awt.Robot` can reproduce user events, it's too basic and is difficult to use for GUI testing. Abbot builds a level of abstraction on top of the `java.awt.Robot` that's much easier to understand, making tests easier to maintain.

Out of the box, Abbot provides an extension for JUnit that lets developers write both programmatic and script-based GUI tests as regular JUnit tests. Because JUnit by itself

doesn't provide support for testing Swing applications, Abbot provides the missing pieces needed to create JUnit GUI tests. Because these GUI tests are also ordinary JUnit tests, we can integrate them with the rest of our application's test suite, providing benefits such as consolidated test execution reports.

Using Abbot with TestNG

JUnit introduced automated unit tests to Java developers. Although it does this successfully, JUnit aims to test classes in isolation, leaving developers without the extra features and flexibility necessary for higher levels of testing. TestNG (<http://testng.org>) picks up where JUnit leaves off. TestNG is a Java testing framework designed to cover different categories of tests, from unit to functional to integration tests. It was inspired by JUnit's simplicity and NUnit's use of metadata annotations for configuration.

TestNG has many features that JUnit lacks, making it more powerful and easy to use:

- Test methods can depend on one or more specific methods.
- TestNG supports data-driven testing and parameterized tests.
- TestNG groups let developers categorize tests, which makes test execution more flexible.

Like JUnit, TestNG doesn't support GUI testing. In addition, Abbot provides extensions for JUnit only. To combine the power of Abbot and TestNG, custom integration code is necessary. Although writing and maintaining our own Abbot extensions for TestNG isn't difficult, it's time consuming. That's why we created the open source project TestNG-Abbot (<http://code.google.com/p/testng-abbot>), which, as you might have guessed, provides an easy way to use Abbot and TestNG together.

Example

Here we show how to use TestNG-Abbot for applying TDD to GUI development. We also introduce some suggestions that can greatly simplify creating and maintaining programmatic GUI tests.

We adapted this example from Tuzi-Mail (http://javaforge.com/proj/summary.do?proj_id=1096), an open source Swing email client used as a test case for TestNG-Abbot. We're going to create a panel for the wizard that creates

JUnit aims to test classes in isolation, leaving developers without the extra features and flexibility necessary for higher levels of testing.

Figure 3. GUI design sketch of a wizard for creating a new email account.

a new email account. Figure 3 shows a sketch of the panel's user interface.

We expect the panel will behave as follows:

1. The user enters his or her name and email address (both required).
2. The email address must be valid.
3. The system displays an error message if the required input is missing or invalid, preventing the user from going to the next step in the wizard.

To make the code more modular and testable, we need to separate model and view. For successful GUI testing (and also good OO development practices), we must move as much code as possible from the GUI to the domain model. In this example, we're going to keep email address validation in the model.⁵ As good TDD practitioners, we'll start with a failing test. The code in figure 4 shows a typical TestNG test. Keep in mind that

- test classes don't need to extend any class, which is a big advantage in a language with single inheritance such as Java, and
- test methods don't need to start with the word "test" because the `@Test` annotation tells TestNG that the method `shouldReturnIsValidIfAddressIsValid` is a test.

Following the TDD process, we need to create the class `Email`, run the test, and verify

that it fails as we expect it to. Then we need to implement email address validation to pass the test, following the principle of doing "the simplest thing that could possibly work." The code in figure 5 shows the implementation of email address validation.

Now we can run the test (for example, using the TestNG plug-in for Eclipse) and verify that it passes. Even though the test passed, there's not enough evidence that the code is doing what it's supposed to (for example, we might be always returning `true`). The next step is to write one or more tests that verify that the GUI actually reports a given invalid email address as invalid.

Now that we've implemented email validation, we need to create the GUI for users to enter their names and email addresses. As usual, we'll start with a failing test. The test in figure 6 verifies that the New Email Account wizard displays an error message if the user presses the Next button without entering his or her name in the proper text field.

The test uses a `org.testng.abbot.fixture.DialogFixture` from TestNG-Abbot (line 9) to launch the GUI under test and find specific GUI components. To guarantee proper use of `DialogFixture`, we must

- use the TestNG annotation `@BeforeMethod` to instantiate an `AbbotFixture` before any test method is executed (line 11) and
- use the TestNG annotation `@AfterMethod` to free up resources used by `DialogFixture` (such as the mouse and windows) after any test method executes (line 16).

This test uses TestNG-Abbot to invoke the GUI to test and simulate user input by

- launching the GUI using the `DialogFixture` in line 13,
- making sure that the text field where the user enters his or her name is empty (line 21),

Figure 4. An initial failing test for our domain model.

```
// Omitted imports and package declaration

public class EmailTest {

    @Test public void shouldReturnIsValidIfAddressIsValid() {
        assertTrue(new Email("alruiz15@yahoo.com").valid);
    }
}
```

- simulating a user entering a valid e-mail address in the second text field of the dialog (line 22),
- finding the “Next” button and simulating a user pressing it (lines 23), and
- verifying that the error message “Please enter your name” is displayed in a JOption Pane (lines 24).

The test in figure 6 brings up a worthwhile observation. Although Abbot supports component lookup by name (using a `NameMatcher`), type (using a `ClassMatcher`), or custom criteria (by implementing `Matcher`), our example using a `DialogFixture` performs component lookup by name for these reasons:

- Finding components by type is trivial as long as the GUI under test has only one component of that type. If it has more than one component of the specified type, we must do some extra work to identify the one we’re looking for.
- We can’t rely on a component’s displayed text as a way to identify it. Displayed text tends to change, especially if the applica-

tion supports multiple languages.

- Using a unique name for GUI components guarantees that the developer can always find them, regardless of any change in the

```
// Omitted imports and package declaration
public final class Email {
    private static final Pattern EMAIL_PATTERN =
        Pattern.compile(
            "([a-zA-Z0-9\\.]+)@[a-zA-Z_]+?\\. [a-zA-Z]{2,3}$");

    private Matcher matcher;
    public final String address;
    public final boolean valid;

    public Email(String address) {
        this.address = address;
        valid = valid();
    }

    private boolean valid() {
        if (isEmpty(address)) return false;
        matcher = EMAIL_PATTERN.matcher(address);
        return matcher.matches();
    }
}
```

Figure 5. A domain model encapsulating email validation. An email address is valid only if it isn’t an empty String and matches the regular expression pattern described in `EMAIL_PATTERN`.

```
// Omitted imports and package declaration

1 public class NewEmailAccountWizardTest {
2
3     private static final String EMAIL = "alex.ruiz.05@gmail.com";
4
5     private static final String NAME_FIELD = "wizard.account.name.field";
6     private static final String EMAIL_FIELD = "wizard.account.email.field";
7     private static final String NEXT_BUTTON = "wizard.account.next.button";
8
9     private DialogFixture fixture;
10
11     @BeforeMethod public void setUp() throws Exception {
12         fixture =
13             new DialogFixture(new AbbotFixture(), new NewEmailAccountWizard());
14     }
15
16     @AfterMethod public void tearDown() throws Exception {
17         fixture.cleanUp();
18     }
19
20     @Test public void shouldDisplayErrorIfNameNotEntered() {
21         fixture.findTextComponent(NAME_FIELD).deleteText();
22         fixture.findTextComponent(EMAIL_FIELD).enterText(EMAIL);
23         fixture.findButton(NEXT_BUTTON).click();
24         fixture.findOptionPane().shouldDisplayMessage("Please enter your name");
25     }
26 }
```

Figure 6. An initial failing test for our New Email Account wizard. This test simulates a user entering an email address but not his or her name, in which case we expect the application under test to display an error message.

```
// Omitted imports and package declaration

final class NewEmailAccountWizard extends JDialog {

    private static final long serialVersionUID = 1L;

    private final JTextField nameField = new JTextField();
    private final JTextField emailField = new JTextField();
    private final JButton nextButton = new JButton("Next >");

    NewEmailAccountWizard() {
        setLayout(new GridBagLayout());
        setSize(100, 100);
        GridBagConstraints c = new GridBagConstraints();
        c.gridx = c.gridy = 0;
        c.fill = BOTH;
        add(nameField, c);
        nameField.setName("wizard.account.name.field");
        c.gridy++;
        add(emailField, c);
        emailField.setName("wizard.account.email.field");
        c.gridy++;
        add(nextButton, c);
        nextButton.setName("wizard.account.next.button");
        nextButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if (isEmpty(nameField.getText()))
                    showMessageDialog(
                        NewEmailAccountWizard.this, "Please enter your name");
                if (!new Email(emailField.getText()).valid)
                    showMessageDialog(
                        NewEmailAccountWizard.this, "Please enter a valid e-mail address");
            }
        });
    }
}
```

Figure 7. New Email Account wizard with email validation.

GUI, as long they haven't been removed from the GUI.

It's interesting that the test has only one compilation error (line 9). As TDD prescribes, we haven't created the class under test (`NewEmailAccountWizard`) yet. There are no additional compilation errors because we don't need to hard-code component hierarchies to find components. We only need to tell TestNG-Abbot what we're looking for, and the framework will take care of the rest.

Once the test is in place, we need to add some code to make it pass. Although the code doesn't look very good, it satisfies our main purpose: to make our test pass. Later on, we can refactor and polish it.

As the specification states, the GUI must display an error message if the user enters an

invalid email address. As expected, we must add a new failing test that verifies this requirement for `NewEmailAccountWizardTest` before we implement the feature in the GUI. Then it's time to implement the code to make the new test pass (see figure 7).

Now that the test is passing, we need to make the GUI user-friendly by adding some instructional text and moving the components around. The tests will make sure we don't introduce bugs when adding some "makeup" to the GUI. Figure 8 shows a screenshot of the completed New Email Account wizard panel.⁴


The code for the panel in figure 3 went through several refactoring cycles and doesn't look even close to the code in figure 7. Because the source code and tests are part of the open source Tuzi-Mail project, you can download the complete source code. And, because it's re-

leased under the Apache 2.0 license (www.apache.org/licenses/LICENSE-2.0.html), you can use it as you wish.

Testing tips

Because using a library or framework by itself doesn't guarantee good coding practices, we've compiled some tips for GUI testing:

- Practice TDD, even for GUIs.
- Separate the model and view to make code more modular and testable.
- Use a unique name for GUI components to ensure reliable, consistent component lookup.
- Don't spend time testing default component behavior (buttons responding to mouse clicks) or visual appearance (label text or color values).
- Concentrate on testing your GUI's expected behavior.

GUI testing is a necessary practice that has increased complexity in software development. TestNG-Abbot, a developer-friendly library, provides a simple API that facilitates the application of TDD to Java GUIs. TestNG-Abbot provides fixtures for the most basic Swing components. Future improvements will cover support for more complex Swing components, such as `JTrees` and `JTables`; support for third-party GUI components, such as the ones provided by SwingLabs' `SwingX` (<https://swingx.dev.java.net>); and an easy-to-use, domain-specific language for GUI testing using a dynamic programming language such as Groovy or JRuby. 

Acknowledgments

See www.computer.org/software for additional code samples related to this article.

References

1. P. Hamill, *Unit Tests Frameworks*, O'Reilly, 2004.
2. K. Beck, *Test-Driven Development by Example*, O'Reilly, 2002.
3. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

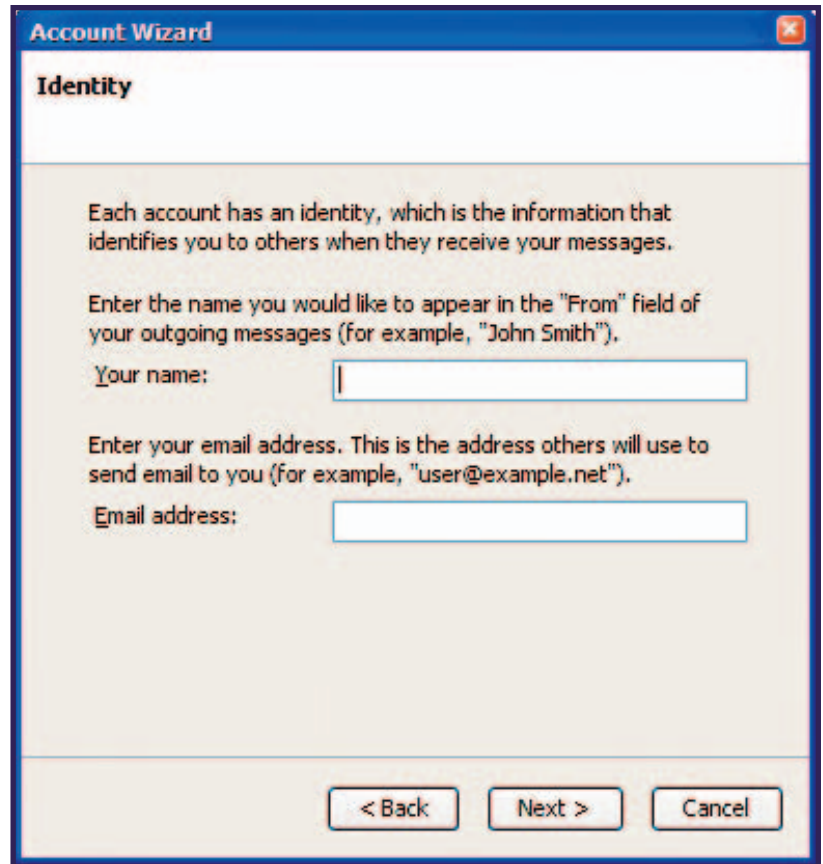


Figure 8. The completed panel for the New Email Account wizard.

About the Authors



Alex Ruiz is a principal software engineer for the JDeveloper team at Oracle. His research interests include aspect-oriented programming, Swing, concurrency, persistence, and testing. He received his bachelor's degree in computer science from Florida Atlantic University. Contact him at 500 Oracle Parkway, Redwood Shores, CA 94065; alex.ruiz@oracle.com.

Yvonne Wang Price is a developer, senior technical instructor, and Java lead at Oracle. Her research interests include software development using Java, GUI development with both Swing and JavaServer Faces, object-relational mapping, and agile practices. She received her master's degree in computer science from the University of New Brunswick. Contact her at 500 Oracle Parkway, Redwood Shores, CA 94065; yvonne.price@oracle.com.



4. A. Ruiz, "Programmatic GUI Tests with TestNG-Abbot (with Demo)," http://jroller.com/page/alexRuiz?entry=testng_abbot_supports_programmatic_tests.
5. D. Astels, *Test-Driven Development: A Practical Guide*, Prentice Hall, 2003.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.