

Getting Started

This guide explains how to get started with FEST's Swing module.

Before you start

Before starting to write any test, please do the following:

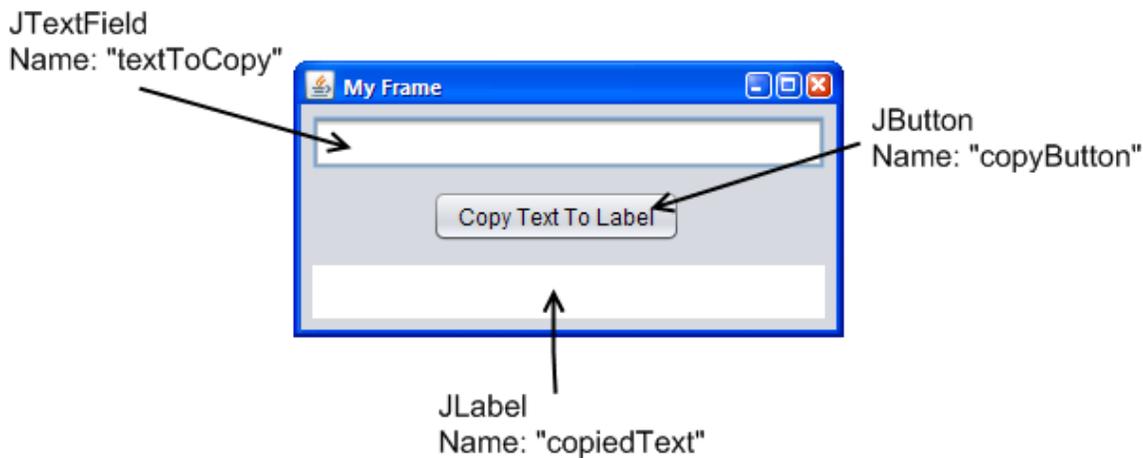
1. Read about [writing EDT-safe GUI tests](#).
2. Download the latest version of the Swing module from the [project's download page](#). The file name should be similar to fest-swing-`{VERSION}`.zip, where `{VERSION}` is the latest version of the module.
3. Include the file fest-swing-`{VERSION}`.jar and its [dependencies](#) in your classpath.

Writing your first GUI test

When writing GUI tests, use the fixtures in the package [org.fest.swing.fixture](#). These fixtures provide specific methods to simulate user interaction with a GUI component and also provide assertion methods that verify the state of such GUI component. Although you could work with the FEST [Robot](#) directly, the `Robot` is too low-level and requires considerably more code than the fixtures.

There is one fixture per Swing component. Each fixture has the same name as the Swing component they can handle ending with "Fixture." For example, a `JButtonFixture` knows how to simulate user interaction and verify state of a `JButton`.

For our first test, let's assume we have a very simple `JFrame` that contains a `JTextField`, a `JLabel` and a `JButton`:



The expected behavior of this GUI is: when user clicks on the `JButton`, the text of the `JTextField` should be copied to the `JLabel`.

The following sections describe the steps necessary to test our GUI.

Creating a test from scratch

1. Enable checks for EDT access violation

FEST provides the class `FailOnThreadViolationRepaintManager` that forces a test failure if access to GUI components is not performed on the EDT. You can find more details [here](#).

2. Create a fixture for a Frame or Dialog

Depending on the GUI to test, create a fixture to handle either a `Frame` or a `Dialog` in the "setUp" method of your test. The "setUp" method is the method that initializes the test fixture **before** running **each** test method:

- `setUp` method (JUnit 3.8.x)
- a method marked with `@Before` (JUnit 4.x)
- a method marked with `@BeforeMethod` (TestNG)

The following example uses a [FrameFixture](#):

```
private FrameFixture window;

@BeforeMethod public void setUp() {
    MyFrame frame =
    GuiActionRunner.execute(new
    GuiQuery<MyFrame>() {
        protected MyFrame executeInEDT() {
            return new MyFrame();
        }
    });
    window = new FrameFixture(frame);
    window.show(); // shows the frame to test
}
```

It may seem a little weird the way we create a new instance of `MyFrame`. Since creation of a frame triggers a "paint" action, we need to create the frame in the Event Dispatch Thread (EDT.) More details about the EDT and Swing threading can be found [here](#).

3. Clean up resources used by FEST-Swing

FEST-Swing forces sequential test execution, regardless of the testing framework (JUnit or TestNG.) To do so, it uses a semaphore to give access to the keyboard and mouse to a single test. Cleaning up resources after running each test method releases the lock on such semaphore. To clean up resources simply call the method `cleanUp` in the FEST-Swing fixture inside:

- `tearDown` method (JUnit 3.8.x)
- any method marked with `@After` (JUnit 4.x)
- any method marked with `@AfterMethod` (TestNG)

Example:

```
@AfterMethod public void tearDown() {
    window.cleanUp();
}
```

4. Write methods to test your GUI's behavior

Start using [FEST-Swing fixtures](#) to test your GUI. FEST-Swing fixtures simulate a user interacting with a GUI in order to verify that such GUI behaves as we expect. For our example, we need to verify that the text in the `JTextField` is copied to the `JLabel` when the `JButton` is clicked:

```
@Test public void
shouldCopyTextInLabelWhenClickingButton() {

    window.textBox("textToCopy").enterText("Some
    random text");
    window.button("copyButton").click();

    window.label("copiedText").requireText("Some
    random text");
}
```

As you probably guessed by now, in our example we look up UI components by their unique name.

Putting everything together

The following code listing shows the whole test that verifies the described GUI is behaving correctly:

```
import org.testng.annotations.*;
import org.fest.swing.fixture.FrameFixture;

public class FirstGUITest {

    private FrameFixture window;
```

```
@BeforeClass public void setUpOnce() {

FailOnThreadViolationRepaintManager.install(
);
}

@BeforeMethod public void setUp() {
    MyFrame frame =
GuiActionRunner.execute(new
GuiQuery<MyFrame>() {
        protected MyFrame executeInEDT() {
            return new MyFrame();
        }
    });
    window = new FrameFixture(frame);
    window.show(); // shows the frame to
test
}

@AfterMethod public void tearDown() {
    window.cleanUp();
}

@Test public void
shouldCopyTextInLabelWhenClickingButton() {

window.textBox("textToCopy").enterText("Some
random text");
    window.button("copyButton").click();
}
```

```
window.label("copiedText").requireText("Some
```

```
random text");
    }
}
```

Alternatively, extend a FEST-Swing test case

Starting with version 1.1, FEST-Swing provides a base test case class, to simplify creation of GUI tests. The following code listing provides the same functionality as the example from the previous section, with less code, thanks to [FestSwingTestngTestCase](#):

```
import org.testng.annotations.*;
import org.fest.swing.fixture.FrameFixture;
import
org.fest.swing.testng.testcase.FestSwingTest
ngTestCase;

public class FirstGUITest extends
FestSwingTestngTestCase {

    private FrameFixture window;

    protected void onSetUp() {
        MyFrame frame =
GuiActionRunner.execute(new
GuiQuery<MyFrame>() {
            protected MyFrame executeInEDT() {
                return new MyFrame();
            }
        });
        // IMPORTANT: note the call to 'robot()'
        // we must use the Robot from
FestSwingTestngTestCase
```

```
        window = new FrameFixture(robot(),
frame);
        window.show(); // shows the frame to
test
    }
```

```
    @Test public void
shouldCopyTextInLabelWhenClickingButton() {

window.textBox("textToCopy").enterText("Some
random text");
        window.button("copyButton").click();

window.label("copiedText").requireText("Some
```

```
random text");  
    }  
}
```

Warning

When using a base `TestCase`, do not create a new `Robot`. The base `TestCase` creates one for you. If there is more than one `Robot` in your test, only the first one will have access to the screen, while the rest will block till they get the 'screen lock'. A `Robot` can be created manually or indirectly using the constructors [FrameFixture\(Frame\)](#) or [DialogFixture\(Dialog\)](#). Please use the overloaded versions that take a `Robot` as parameter, passing the already created `Robot` (by calling the method `robot()`.)

See also:

- [Base TestCase \(TestNG and JUnit\)](#)
- [Jar Files Explained](#)