Testing Event-Driven Programs in Processing

Hiroshi Hosobe
Faculty of Computer and Information Sciences, Hosei University
3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
hosobe@acm.org

ABSTRACT

Event-driven programming is a paradigm that is widely used in many fields. Processing is a set of programming languages and environments specialized in event-driven programming for interactive graphical applications. It provides only low-level event-handling functions, which imposes difficulty on novice programmers in programming complex behaviors. This paper proposes a method for unit-testing event-driven Processing programs. It allows writing testable Processing programs and test programs in Java. To demonstrate how it works, this paper presents case studies on testing whether mouse and key events are correctly handled.

CCS Concepts

• Software and its engineering→Software creation and management→Software verification and validation→Software defect analysis→Software testing and debugging; • Software and its engineering→Software notations and tools→Context specific languages→Graphical user interface languages

Keywords

unit testing; event-driven programming; Processing

1. INTRODUCTION

Event-driven programming is a paradigm that is widely used in many fields including operating systems, distributed systems, and graphical user interfaces [1][2][3]. It constructs a program as a set of procedures (called event handlers) that process various events (or actions) that come from, for example, other processes, remote computers, and user interface devices.

Processing [4][5] is a set of programming languages and environments that are often used for the education of novice programmers. A characteristic of Processing is that it is specialized in event-driven programming for interactive graphical applications; in Processing, programmers construct programs by writing event handlers. Unlike ordinary graphical user interface programming environments, Processing provides only low-level functions, which imposes difficulty on novice programmers in writing programs that exhibit complex behaviors.

In this paper, we propose a method for testing event-driven programs that are written in Processing. It is based on unit testing that is widely used in software development. The method allows writing testable Processing programs in Java in an almost normal way, and enables the resulting programs to run in the same way as normal Processing programs. Also, the method allows writing test programs in Java by specifying a set of test methods that group similar test cases described in terms of assertions. To demonstrate how the proposed method works for unit-testing event-driven Processing programs, we present three case studies on testing whether mouse and key events are correctly handled. In the second and third case studies, we show, for comparison, incorrect programs that fail in unit testing.

The rest of this paper is organized as follows. Section 2 describes previous work related to the proposed method. Section 3 proposes our method, and Section 4 gives its implementation. Section 5 presents examples of the use of our method, and Section 6 discusses the method. Finally, Section 7 provides conclusions and future work.

2. RELATED WORK

Unit testing is a test of individual programs or modules in order to ensure that there are no analysis or programming errors [6]. Unit testing is widely used in software development in organizations such as companies [7]. JUnit [8][9] is a unit testing tool that is widely used for the development of Java programs.

Unit testing is used for graphical user interfaces based on event-driven programming. For example, jfcUnit [10] and Abbot [11] are tools for unit testing for graphical user interfaces constructed in Java. They allow generating events for graphical user interfaces and writing test programs that test internal states. Such methods for testing graphical user interfaces are called script-based methods [12].

In addition to script-based methods, graphical user interface testing methods such as model-based and capture/replay methods have been studied. GUITAR [12] is a model-based method for testing graphical user interfaces; it allows testing graphical user interfaces by generating test cases based on models of events described with graphs.

There is a tutorial on the use of unit testing for Processing [13]. The tutorial uses unit testing for the test-driven development [14] of functions that are defined in Processing programs. However, unlike the method that we propose in this paper, this tutorial does not treat event handlers.

3. PROPOSED METHOD

In this section, we propose a method for unit-testing event-driven programs that are written in Processing. Our method allows writing test programs in Java. As with many other unit testing tools, it allows specifying a set of test methods that group similar test cases. Also, it allows specifying a test case in terms of an assertion. Such an assertion is typically defined as follows:

ESSE 2020, November 6–8, 2020, Rome, Italy.

© 2020 ACM. ISBN. 978-1-4503-7678-5/20/11…$15.00.

(1) First, create an instance of the main class of the target Processing program, and call its `startTest` method;

(2) Call a sequence of methods that send events to the main Processing instance, recording its internal states at the same time;

(3) Finally, check whether the recorded internal states satisfy a necessary condition.

Our method allows writing testable Processing programs in Java in an almost normal way. Its difference from the normal way is that our method needs to define main programs as subclasses of class `PTestableApplet` while the normal way of writing Processing programs in Java [15] needs to define them as subclasses of class `PApplet`. `PTestableApplet` is a subclass of `PApplet` that is able to perform necessary functions for Processing programs; when executed with the static `main` method, the programs run in the same way as normal Processing programs. Figure 1 and Figure 2 show a testable Processing program and its test program respectively that we will use for a case study in Subsection 5.1.

To realize our method, `PTestableApplet` introduces two modes, normal and test. When executed with the static `main` method, it runs in the normal mode; in this mode, it simply calls `PApplet`'s methods to make it behave in the same way as normal Processing programs. By contrast, when called with the `startTest` method by a test program, it starts the test mode; in this mode, it simulates Processing's execution by generating events that are specified in the test program and then calling the corresponding event handlers that can be defined in the target Processing program.

```
 1: public class SimpleButton
        extends PTestableApplet {
 2:   boolean toggle = false;
 3:   public void settings() {
 4:     size(400, 400);
 5:   }
 6:   public void draw() {
 7:     background(toggle ? 0 : 255);
 8:     fill(0xff0000ff);
 9:     rect(175, 175, 50, 50);
10:   }
11:   public void mousePressed() {
12:     if (mouseX >= 175 && mouseY >= 175 &&
            mouseX < 225 && mouseY < 225) {
13:       toggle = !toggle;
14:     }
15:   }
16:   public static void main(String[] args) {
17:     SimpleButton.main("SimpleButton");
18:   }
19: }
```

Figure 1: Example of a testable Processing program: a correct implementation of a simple graphical button.

4. IMPLEMENTATION

By using the proposed method, we implemented a prototype system in Java with AdoptOpenJDK 1.8.0_232-b09. We used the `core.jar` library of Processing 3.5.3 to execute the normal mode of the proposed method and to obtain necessary information for the test mode. We used JUnit 5.5.2 for unit testing.

5. CASE STUDIES

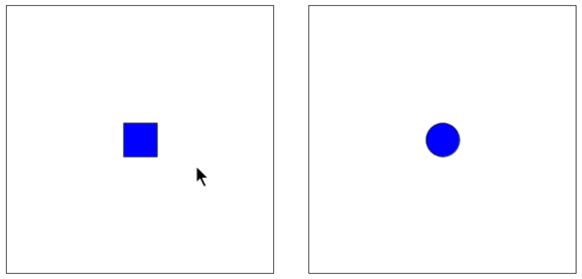
In this section, we present three case studies to demonstrate how the proposed method works.

```
import static org.junit.jupiter.api.Assertions.assertTrue;
import org.junit.jupiter.api.Test;
public class SimpleButtonTest {
  @Test
  public void testPressButton() {
    assertTrue(() -> {
      SimpleButton button = new SimpleButton();
      button.startTest();
      boolean toggle0 = button.toggle;
      button.moveMouse(200, 200, 1); // move to button
      button.pressMouse();
      button.pass(1); // do nothing
      button.releaseMouse();
      button.pass(1); // do nothing
      return !toggle0 && button.toggle;
    });
  }
  @Test
  public void testPressOutsideOfButton() {
    assertTrue(() -> {
      SimpleButton button = new SimpleButton();
      button.startTest();
      boolean toggle0 = button.toggle;
      button.moveMouse(300, 300, 1); // move to outside
      button.pressMouse();
      button.pass(1); // do nothing
      button.releaseMouse();
      button.pass(1); // do nothing
      return !toggle0 && !button.toggle;
    });
  }
}
```

Figure 2: Example of a test program for Processing: a test program for the simple graphical button.

5.1 Simple Graphical Button

The first case study treats a simple graphical button. As shown in Figure 3(a), it draws a blue square at the center of the window. When it is normally clicked with a mouse, it changes the background color of the window; specifically, the background is toggled either from white to black or from black to white.



(a) (b)

Figure 3: Event-driven programs used in the case studies: (a) a graphical button; (b) an object movable with keys.

Figure 1 shows a correct implementation of this simple button in Processing. It declares a Boolean-type instance variable `toggle`, which indicates whether the current background color is black or white. In the draw method (lines 6–10), the program first clears the window by using the current background color, and then draws and fills a square with blue. (The draw method is an event handler that is repeatedly called by Processing every 1/60 seconds to draw the screen.)

Method `mousePressed` is an event handler that we test in this case study. In Processing, `mousePressed` is called when a mouse button is pressed. In this program, `mousePressed` (lines 11–15) changes the value of `toggle` if the mouse button is released inside the square.

Figure 2 shows a test program that performs unit testing for the simple button. Methods `testPressButton` and `testPressOutsideOfButton` are test methods, where `assertTrue` is used for assertions that require their arguments to take true values. In both methods, the initial value of `toggle` is recorded in variable `toggle0`. `testPressButton` (lines 5–17) specifies the following events: first, the mouse cursor is moved to the center of the square in 1 second; then, the mouse button is pressed; after passing 1 second without any inputs (even during which the draw method is repeatedly called every 1/60 seconds), the mouse button is released; finally, another 1 second is passed again without any inputs. In this case, the button click should be accepted (and change the value of `toggle`), and therefore `testPressButton` checks that `toggle0` is false, and that the final value of `toggle` is true. By contrast, `testPressOutsideOfButton` (lines 19–31) specifies events that should not cause a button click. Therefore, `testOutsideOfButton` checks that both `toggle0` and the final value of `toggle` are false.

We executed our prototype system to apply this test program to the correct implementation of the simple button shown in Figure 1. The system reported that the two test methods passed successfully.

5.2 Cancelable Graphical Button

The second case study treats a cancelable graphical button. Similar to the previous one, it draws a blue square at the center of the window. Also, it changes the background color of the window when it is normally clicked with a mouse. In addition, it supports the “cancel” operation of the button click; if the mouse cursor goes to the outside of the square without the release of the mouse button, it ignores the button click (i.e., does not toggle the background color). It should be noted that such a cancel operation is commonly supported by buttons that appear in ordinary graphical user interfaces.

Figure 4 shows a correct implementation of the cancelable button. It declares a Boolean-type instance variable `buttonPressed` in addition to `toggle`: `buttonPressed` is introduced to present a feedback about whether the graphical button is being pressed or not. In the draw method (lines 7–11), the program first clears the window by using the current background color, and then draws and fills a square with either red (if the button is being pressed) or blue (otherwise).

This case study tests method `mouseReleased` as well as `mousePressed`; `mouseRelased` is an event handler that is called when a mouse button is released. In this program, `mousePressed` (lines 12–14) assigns true to `buttonPressed` if the mouse cursor is inside the square and false otherwise. Note that, unlike the previous case study, it does not immediately change the value of `toggle` since this operation might be canceled. Instead, `mouseReleased` (lines 15–20) changes the value of `toggle` if the mouse button is released inside the square. If the mouse button is released outside the square, it does not change the value of `toggle`, which means that the button click is canceled.

```
 1: public class CancelableButton
        extends PTestableApplet {
 2:   boolean toggle = false;
 3:   boolean buttonPressed = false;
 4:   public void settings() {
 5:     size(400, 400);
 6:   }
 7:   public void draw() {
 8:     background(toggle ? 0 : 255);
 9:     fill(buttonPressed ? 0xffff0000 :
            0xff0000ff);
10:     rect(175, 175, 50, 50);
11:   }
12:   public void mousePressed() {
13:     buttonPressed =
            mouseX >= 175 && mouseY >= 175 &&
            mouseX < 225 && mouseY < 225;
14:   }
15:   public void mouseReleased() {
16:     if (buttonPressed &&
            mouseX >= 175 && mouseY >= 175 &&
            mouseX < 225 && mouseY < 225) {
17:       toggle = !toggle;
18:     }
19:     buttonPressed = false;
20:   }
21:   public static void main(String[] args) {
22:     CancelableButton.main(
            "CancelableButton");
23:   }
24: }
```

Figure 4: Correct implementation of the cancelable graphical button.

```
 1: import static org.junit.jupiter.api.
        Assertions.assertTrue;
 2: import org.junit.jupiter.api.Test;
 3: public class CancelableButtonTest {
 4:   @Test
 5:   public void testPressButton() {
        ...
17:   }
18:   @Test
19:   public void testPressOutsideOfButton() {
        ...
31:   }
32:   @Test
33:   public void testCancelButton() {
34:     assertTrue(() -> {
35:       CancelableButton button =
              new CancelableButton();
36:       button.startTest();
37:       boolean toggle0 = button.toggle;
38:       button.moveMouse(
              200, 200, 1); // move to button
39:       button.pressMouse();
40:       button.moveMouse(
              300, 300, 1); // move to outside
41:       button.releaseMouse();
42:       button.pass(1); // do nothing
43:       return !toggle0 && !button.toggle;
44:     });
45:   }
46: }
```

Figure 5: Test program for the cancelable graphical button.

Figure 5 shows a test program that performs unit testing for the cancelable button. Methods `testPressButton` and `testPressOutsideOfButton` are the same test methods as those in the previous case study (and therefore are not shown here). The additional method `testCancelButton` (lines 33–45) specifies events that should cause the cancel of the button click by moving the mouse cursor to the outside of the square before the mouse button is released. Therefore, `testCancelButton` checks that both `toggle0` and the final value of `toggle` are false.

We executed our prototype system to apply this test program to the correct implementation of the cancelable button shown in Figure 4. The system reported that the three test methods passed successfully.

For comparison, we applied the same test program to the implementation of the simple button shown in Figure 1 (after renaming its class name), which we introduced in the previous case study. Our prototype system reported that `testPressButton` and `testPressOutsideOfButton` succeeded, but that `testCancelButton` failed, which was the expected result.

5.3 Object Movable with Keys

The third case study treats a graphical object that can be moved with arrow keys. As shown in Figure 3(b), it draws a blue circle initially at the center of the window. While arrow keys are being pressed, the circle continuously moves to the direction corresponding to the pressed arrow keys. The important point is that it supports the simultaneous press of multiple arrow keys; for example, while the right and up arrow keys are being simultaneously pressed, the circle continuously moves to the upper right direction. It should be noted that such movement of a graphical object is commonly supported by video games.

Figure 6 shows a correct implementation of the movable object in Processing. It declares four integer-type instance variables `x`, `y`, `vx`, and `vy`: `x` and `y` indicate the coordinates of the center of the circle; `vx` and `vy` indicate the changes in 1 frame (i.e., 1/60 seconds) that should be made in `x` and `y` respectively, which can be regarded as the “velocity” components of the circle. In the draw method, the program first clears the window, then updates `x` and `y`, and draws and fills a circle with blue.

Methods `keyPressed` and `keyReleased` are event handlers that we test in this case study. In Processing, `keyPressed` and `keyReleased` are called when a key is pressed and released respectively. In this program, `keyPressed` (lines 16–26) assigns −1 or 1 to `vx` or `vy` if the pressed key is an arrow key. Also, `keyReleased` (lines 27–33) assigns 0 to `vx` or `vy` if the released key is an arrow key.

Figure 7 shows a test program that performs unit testing for the movable object. Methods `testMoveRight` and `testMoveRightUp` are test methods. In these methods, the initial and intermediate values of `x` and `y` are recorded in variables such as `x0` and `y0`. `testMoveRight` (lines 5–19) specifies the following events: first, 1 second is passed without any inputs; next, after the right arrow key is pressed (but not released), 1 second is passed (during which the object should continuously move to the right); finally, after the right arrow key is released, 1 second is passed. By contrast, `testMoveRightUp` (lines 21–41) specifies the following events: first, 1 second is passed without any inputs; next, after the right arrow key is pressed, 0.25 seconds are passed (during which the object should continuously move to the right); then, after the up arrow key is pressed, 0.5 seconds are passed (during which the object should continuously move to the upper right direction due to the simultaneous press of the two keys); after the right arrow key is released, 0.25 seconds are passed (during which the object should continuously move upward); finally, after the up arrow key is released, 1 second is passed. At the end of these methods, they check that the past and final values of `x` and `y` satisfy the expected relations.

We executed our prototype system to apply this test program to the correct implementation of the movable object shown in Figure 6. The system reported that the two test methods passed successfully.

For comparison, we show an incorrect implementation of the movable object in Figure 8. It does not define `keyPressed` and `keyReleased`. Instead, in draw (lines 7–22), it uses instance variable `keyPressed`, which is presented by Processing as an alternative way of handling the press of a key. Although the use of variable `keyPressed` might seem simple, it does not handle the simultaneous press of multiple keys. By applying the same test program to this incorrect implementation, our prototype system reported that `testMoveRight` succeeded, but that `testMoveRightUp` failed, which was the expected result.

```
 1: public class Mover extends PTestableApplet {
 2:   int x = 200;
 3:   int y = 200;
 4:   int vx = 0;
 5:   int vy = 0;
 6:   public void settings() {
 7:     size(400, 400);
 8:   }
 9:   public void draw() {
10:     background(255);
11:     x += vx;
12:     y += vy;
13:     fill(0xff0000ff);
14:     ellipse(x, y, 50, 50);
15:   }
16:   public void keyPressed() {
17:     if (keyCode == LEFT) {
18:       vx = -1;
19:     } else if (keyCode == RIGHT) {
20:       vx = 1;
21:     } else if (keyCode == UP) {
22:       vy = -1;
23:     } else if (keyCode == DOWN) {
24:       vy = 1;
25:     }
26:   }
27:   public void keyReleased() {
28:     if (keyCode == LEFT ||
          keyCode == RIGHT) {
29:       vx = 0;
30:     } else if (keyCode == UP ||
          keyCode == DOWN) {
31:       vy = 0;
32:     }
33:   }
34:   public static void main(String[] args) {
35:     Mover.main("Mover");
36:   }
37: }
```

Figure 6: Correct implementation of the movable object.

```java
import static org.junit.jupiter.api.
    Assertions.assertTrue;
import org.junit.jupiter.api.Test;
public class MoverTest {
  @Test
  public void testMoveRight() {
    assertTrue(() -> {
      Mover mover = new Mover();
      mover.startTest();
      int x0 = mover.x, y0 = mover.y;
      mover.pass(1); // do nothing
      int x1 = mover.x, y1 = mover.y;
      mover.pressKey(Mover.RIGHT);
      mover.pass(1); // move to right
      int x2 = mover.x, y2 = mover.y;
      mover.releaseKey(Mover.RIGHT);
      mover.pass(1); // do nothing
      return x1 == x0 && y1 == y0 &&
          x2 > x1 && y2 == y1 &&
          mover.x == x2 && mover.y == y2;
    });
  }
  @Test
  public void testMoveRightUp() {
    assertTrue(() -> {
      Mover mover = new Mover();
      mover.startTest();
      int x0 = mover.x, y0 = mover.y;
      mover.pass(1); // do nothing
      int x1 = mover.x, y1 = mover.y;
      mover.pressKey(Mover.RIGHT);
      mover.pass(0.25f); // move to right
      int x2 = mover.x, y2 = mover.y;
      mover.pressKey(Mover.UP);
      mover.pass(
          0.5f); // move to upper right
      int x3 = mover.x, y3 = mover.y;
      mover.releaseKey(Mover.RIGHT);
      mover.pass(0.25f); // move upward
      int x4 = mover.x, y4 = mover.y;
      mover.releaseKey(Mover.UP);
      mover.pass(1); // do nothing
      return x1 == x0 && y1 == y0 &&
          x2 > x1 && y2 == y1 &&
          x3 > x2 && y3 < y2 &&
          x4 == x3 && y4 < y3 &&
          mover.x == x4 && mover.y == y4;
    });
  }
}
```

Figure 7: Test program for the movable object.

6. DISCUSSION

As shown in the previous section, the proposed method allows unit testing for event handlers in Processing programs that are written in the almost normal way. In addition, the proposed method allows writing test programs that checks whether the internal states of Processing programs appropriately change, by generating events that correspond to common Processing events such as the motion of a mouse cursor and the press or release of a mouse button. Understanding such test programs does not require difficult knowledge for novice programmers who are the main target of the programming education using Processing. Therefore, we think that the proposed method also is applicable to the education of novice programmers.

```java
public class Mover extends PTestableApplet {
  int x = 200;
  int y = 200;
  public void settings() {
    size(400, 400);
  }
  public void draw() {
    background(255);
    if (keyPressed) {
      if (keyCode == LEFT) {
        x -= 1;
      } else if (keyCode == RIGHT) {
        x += 1;
      } else if (keyCode == UP) {
        y -= 1;
      } else if (keyCode == DOWN) {
        y += 1;
      }
    }
    fill(0xff0000ff);
    ellipse(x, y, 50, 50);
  }
  public static void main(String[] args) {
    Mover.main("Mover");
  }
}
```

Figure 8: Incorrect implementation of the movable object.

When executed in the test mode, the current prototype system performs unit testing without displaying a screen. Also, it repeatedly calls the draw method in a virtual internal time, instead of calling it every 1/60 seconds in real time. In the case of usual unit testing, this is faster and more convenient. However, when a new test program is constructed, or when unit testing is used for the purpose of education as previously described, it will be better to enable the execution of a test program while displaying a screen in real time.

For our prototype system, we adopted the way of writing Processing programs in Java [15]. However, the widely used Java-based Processing system [4] uses a specialized development environment called the Processing Development Environment (PDE) that allows directly writing event handlers without explicitly using class `PApplet`, which is more convenient for novice programmers. To enable our system to adopt this way of writing Processing programs, we need to extend PDE.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a method for unit-testing event-driven programs written in Processing. It allows writing testable Processing programs and test programs in Java. We showed three case studies to demonstrate the proposed method.

Our future work includes the confirmation of the utility of the proposed method for more complex event-driven programs. Other future directions are to enable screens to be displayed during the test mode and to extend PDE by integrating the proposed method in order to enhance the usefulness of the proposed method. Also, we want to expand the proposed method by enabling the description of event models and the automatic generation of test cases.

8. ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP17H01726.

9. REFERENCES

[1] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven programming for robust software. In *Proc. ACM SIGOPS EW*, pages 186–189, 2002.

[2] J. Fischer, R. Majumdar, and T. Millstein. Tasks: Language support for event-driven programming. In *Proc. ACM PEPM*, pages 134–143, 2007.

[3] A. Milicevic, D. Jackson, M. Gligoric, and D. Marinov. Model-based, event-driven programming paradigm for interactive web applications. In *Proc. ACM Onward!*, pages 17–36, 2013.

[4] B. Fry and C. Reas. Processing. https://processing.org/

[5] C. Reas and B. Fry. Processing: Programming for the media arts. *AI Soc.*, 20(4):526–538, 2006.

[6] ISO/IEC/IEEE. Systems and software engineering—vocabulary. Intl. Std. 24765, 2017.

[7] P. Runeson. A survey of unit testing practices. *IEEE Softw.*, 23(4):22–29, 2006.

[8] K. Beck and E. Gamma. Test-infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[9] P. Louridas. JUnit: Unit testing and coding in tandem. *IEEE Softw.*, 22(4):12–15, 2005.

[10] M. Caswell, V. Aravamudhan, and K. Wilson. jfcUnit user documentation, 2004. http://jfcunit.sourceforge.net/

[11] T. Wall. Abbot framework for automated testing of Java GUI components and programs, 2011. http://abbot.sourceforge.net/

[12] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon. GUITAR: An innovative tool for automated testing of GUI-driven software. *Autom. Softw. Eng.*, 21(1):65–105, 2014.

[13] A. Timmons. Unit testing and test driven development, 2017. https://p5js.org/learn/tdd.html

[14] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Comput.*, 38(9):43–50, 2005.

[15] B. Fry. *Visualizing Data: Exploring and Explaining Data with the Processing Environment*, chapter 8, pages 220–263. O'Reilly, 2007.