Event-Driven Programming Facilitates Learning Standard Programming Concepts[1]

Kim B. Bruce Andrea Danyluk, and Thomas Murtagh
Department of Computer Science
Williams College
Williamstown, MA 01267
{kim,andrea,tom}@cs.williams.edu

Abstract

We have designed a CS 1 course that integrates event-driven programming from the very start. In [1] we argued that event-driven programming is simple enough for CS 1 when introduced with the aid of a library that we have developed. In this paper we argue that early use of event-driven programming makes many of the standard topics of CS 1 much easier for students to learn by breaking them into smaller, more understandable concepts.

1 Introduction

We recently implemented a major update of our CS 1 course, which is now based on Java (replacing Pascal). With the support of specially designed libraries, this course takes an objects-first approach, uses truly object-oriented graphics, incorporates event-driven programming techniques from the beginning, and includes concurrency quite early in the course.

Our provision of library classes that support the use of object-oriented geometric figures allows students to write quite interesting programs in an event-driven style after only a few lectures. Library features supporting concurrency are introduced in the fourth week of classes, at the same time as while loops, enabling interesting examples involving animations. As argued in our earlier papers [2, 1], the combination of object-oriented graphics, event-driven programming, and concurrency provides for a very interesting and pedagogically sound introduction to programming.

[1]Research partially supported by NSF CCLI grant DUE-0088895.

In this paper we argue that the use of an event-driven programming style from the beginning also allows instructors to provide more effective introduction to standard CS 1 material such as loops, parameters, and class definitions.

2 Introducing Event-driven Programming Early

In [2] and [1], we describe the library we created to support our approach and how it can be used to teach event-driven programming early in CS 1. There are several reasons for introducing event-driven programming early. First, modern programs in wide distribution tend to use graphic user interfaces and react to user-generated events, and students need to learn how to program in this style [3, 7, 9]. Another reason is that the use of GUI interfaces and event-driven programming is highly motivating for students, especially when compared with traditional programming involving line by line text input and output [4, 5].

While many observers agree on these benefits for event-driven programming, there are concerns that event-driven programming is too difficult for novices [8, 6]. This argument has considerable validity if students are forced to use the very general tools developed for professional programmers. For example, event-driven programming with GUI components in standard Java requires programmers to

1. Create and initialize the GUI component (e.g., create a choice button and add the choices to it).
2. Add the component to a container object.
3. Ensure that the listener class implements the appropriate listener interface.
4. Add an object from the listener class as a listener to the component.

To accomplish this clearly requires more knowledge than students can be expected to have early in an introductory course. To enable novices to program in this style,

our library contains a `WindowController` class that creates a specialized `Canvas` and inserts it into the center of an `Applet`. This `WindowController` class implements the `MouseListener` and `MouseMotionListener` interfaces and contains stub methods corresponding to all of the event-handling methods promised by those interfaces.

Our students are told that their event-handling classes should extend the `WindowController` class and are told the names and signatures of methods that should be written in order to handle the appropriate mouse events. An additional benefit gained from this is that the methods written by students are naturally short, eliminating the necessity of nagging students to break programs into smaller pieces.

The following is a simple program that draws a series of small framed squares on the canvas when the mouse is dragged.

```
public class Squares extends WindowController
{
  public void onMouseDrag(Location mouseLoc )
  {
    new FramedRect( mouseLoc, 4, 4, canvas);
  }
}
```

Readers are encouraged to compare the complexity of this program with a standard Java program producing the same effect.

3 Event-driven Programming as Facilitator

In this section we argue that the use of an event-driven approach from the beginning allows instructors to provide a more effective introduction to standard CS 1 materials. In particular, we discuss the following topics:

- classes,
- parameters, and
- loops,

and show how an event-driven approach makes these concepts easier to learn.

3.1 Classes

Many instructors would like to use an objects-first approach in CS 1, but run into several roadblocks. In order to write classes, students need to be able to write instance variable declarations and method definitions. Method definitions include the use of parameters as well as the statements inside of method bodies. This may lead one to believe that students need to spend six weeks or more learning basic programming concepts before they are ready to write their first classes.

The event-driven programming style that we introduce results in students learning how to use methods and instance variables in a very restricted context in the first week of the course. Students are given the method names and parameters for each of the event-handling methods. They need only write the appropriate method bodies, which tend to be quite simple (see the example above).

Students use instance variables to "remember" information that must be retained between method invocations. These are quite easy for students to use, and quite interesting programs can be written without any use of loops. As an example, consider the following program that allows a user to draw in a window by dragging the mouse:

```
public class Scribble extends WindowController
{
  // remembers location from which to draw
  private Location oldPoint;

  public void onMousePress(Location point) {
    oldPoint = point;
  }

  public void onMouseDrag(Location point) {
    new Line(oldPoint, point, canvas);
    oldPoint = point;
  }
}
```

Note how little students need to know to write this simple program that has fairly sophisticated behavior.

By the time our students are ready to design their own classes in the third week of our course, they are already familiar with writing classes (which extend `WindowController`), declaring instance variables, writing the bodies of simple methods, and using formal parameters. The new topics at that point involve writing constructors and determining names and parameters for the methods in the new class. Even these are natural extensions of concepts they have seen. For example, they used a `begin` method for the same purpose as a constructor in extensions of `WindowController`.

Our first examples of writing classes involve graphic images that behave similarly to the graphic objects in the library. For example, we design a T-shirt class that generates T-shirts that can be moved on the screen. Thus, even the method names, such as `move`, are similar to those they have seen before, except that rather than only using the method names in sending messages to objects from pre-defined classes, students write the method declarations and bodies.

Moreover, the methods that students write are similar to the event-driven methods they have been writing in that they are executed on demand. The Java applications written early in many CS 1 courses consist of a static `main` method which is executed to completion (perhaps invoking other methods). Thus there is a strong notion of a predetermined execution path. By contrast, the event-driven methods are called only when an action occurs. When they finish, the system waits for another event. The methods in normal classes, like the T-shirt class above, are also called by an external action, this time by another object sending a message; upon completion the object remains in existence, waiting for another message.

By starting with event-driven programming, we make it easy for students to make the transition from writing simple methods in a class with event-driven methods to the more general situation of a class with methods that may be called in an undetermined order.

3.2 Parameters

One of the most difficult aspects for students to understand, when learning to write methods or procedures, is the use of parameters and the correspondence between actual and formal parameters. In typical introductory courses students encounter this correspondence for the first time when moving from monolithic main procedures/methods to writing helper methods which abstract away some of the complexity.

With an event-driven approach, students never write monolithic procedures or methods. Instead they begin by writing small method bodies which respond to various user actions. While the first examples presented can ignore parameters, students are soon shown how to use the formal parameters in the method declarations. In the `Squares` class above, for example, our students learn that they can use the formal parameter, `mouseLoc`, to specify where the new squares should be drawn. They are simply told that when the method is invoked, the value of `mouseLoc` will be the location of the mouse. In particular, the students don't yet need to confront the notion of the correspondence between formal and actual parameters.

At the same time, our students gain experience creating and using objects generated from the graphics library. Thus they invoke both constructors and methods, some with multiple parameters. In this case, they supply the actual parameters, but, because they do not see the method bodies, they still do not have to face the issues of the correspondence between formal and actual parameters.

When students begin to define classes, this prior experience with both actual and formal parameters, though each separate from the other, provides the background to help students better understand the notions of formal-actual parameter correspondence. When defining a T-shirt class, for example, students see methods similar to those they have used with the geometric objects, and can see how the actual parameters provided with message sends end up corresponding with the formal parameters used inside the method bodies.

Most introductory courses using an object-oriented approach do have students sending messages to objects from predefined classes early on, gaining experience with using actual parameters, but experience with formal parameters is usually postponed until several weeks later when students write methods for the first time. Our event-driven approach has students writing methods and using formal parameters from the first week of classes. Yet they do not have to face issues of formal-actual correspondence until a few weeks later.

While there are other topics involving the use of parameters that must be addressed (e.g. the use of parameters versus instance variables), we can draw more heavily on the students' prior experiences to help them understand the formal-actual parameter correspondence because they have seen each separately.

3.3 Loops

One of the most interesting aspects of using event-driven programming is that one can write programs with repetitive behavior without involving loops. The class `Squares` above is a good example in that dragging the mouse around on the screen results in repetitively drawing squares.

One can take advantage of this behavior to help students learn to program loops. For example, the following program draws a new blade of grass each time a user clicks the mouse:

```
public class Grass extends WindowController
{
  // constants omitted

  // x-coord of next blade of grass
  private double bladePos;

  public void begin() {
  // draw solid ground and sun
    new FilledRect(0,GROUND_LINE,
      SCREENWIDTH,SCREENHEIGHT-GROUND_LINE,
      canvas);
    new FilledOval(SUN_INSET,SUN_INSET,
      SUN_SIZE,SUN_SIZE,
      canvas).setColor(Color.yellow);
    bladePos = 0;
  }
```

```
  public void onMouseClick(Location point) {
  // grow grass with each mouse click
    if (bladePos < SCREENWIDTH) {
      new Line(bladePos,GRASS_TOP,
        bladePos,GROUND_LINE,
        canvas).setColor(Color.green);
      bladePos = bladePos + GRASS_SPACING;
    }
  }
}
```

If the students have already seen conditional statements, this program is simple to understand. Each click of the mouse creates a new blade of grass a bit to the right of the last as long as `bladePos` is not off the right side of the screen.

While this program does the job, it is clearly a painful way to create a field of grass. This provides motivation for introducing loops to perform the repetitive activity.

More importantly, because the body of the method `onMouseClick` is designed to executed repeatedly, we have already figured out the building blocks of the while loop. We have determined the need for the variable `bladePos`, and how it is to be updated. Moreover, the `if` statement has already specified the conditions under which the body of the loop should no longer be executed. (In practice we would probably first introduce a version of the program without the `if` and only later add it. This allows us to separate concerns even more effectively.)

It is now very simple to rewrite this program with a while loop. All that is necessary is to move the body of `onMouseClick` to the end of the `begin` method and then to change the `if` to a `while`:

```
public class Grass2 extends WindowController
{
  // constants omitted

  // x-coord of next blade of grass
  private double bladePos;

  public void begin() {
  // draw solid ground and sun
    new FilledRect(0,GROUND_LINE,
      SCREENWIDTH,SCREENHEIGHT-GROUND_LINE,
      canvas);
    new FilledOval(SUN_INSET,SUN_INSET,
      SUN_SIZE,SUN_SIZE,
      canvas).setColor(Color.yellow);

    bladePos = 0;

  // grow blades of grass
    while (bladePos < SCREENWIDTH) {
      new Line(bladePos,GRASS_TOP,
        bladePos,GROUND_LINE,
        canvas).setColor(Color.green);
      bladePos = bladePos + GRASS_SPACING;
    }
  }
}
```

Thus we can introduce the idea of loops concretely through repeated executions of methods driven by separate events, and then move in a very straightforward fashion to the syntax of `while` loops. Notice as well that in the first version of the program, execution pauses after each invocation of the `onMouseClick` method. This is very similar to running a while loop with a debugger and a breakpoint at the end of each loop. It allows the programmer to examine the effect of each execution of the body rather than looking only at the result after the loop is completed.

In summary, the use of event-driven programming allows the introduction of the different components of a loop slowly via a series of examples. One can start just with executions of the body of the loop by putting it inside of an event-handling method. This can be tested with repeated events to ensure correctness of successive invocations of the loop body. Then one can add a conditional statement to skip execution when the task is completed. Finally, converting the conditional to a `while` loop (and possibly moving it to a different part of the program) completes the construction of a loop whose parts have already been tested.

4 Conclusions

In earlier papers [2, 1] we have argued that a well designed library can make possible the introduction of event-driven programming in CS 1. We also discussed there the advantages of event-driven programming and an object-oriented graphics library in enabling an objects-first approach to CS 1.

In this paper, we argue that the early introduction of event-driven programming makes many of the standard topics of CS 1 much easier for students to learn. We illustrated this argument with examples involving the introduction of classes, parameters, and loops.

For the last two years we have been teaching a course using these ideas and the library we developed for this purpose. We are currently writing a text based on these ideas. Our materials are being tested this year by faculty at other institutions in the United States. We hope to have an even larger group using the materials in 2002-2003, with the publication of our text projected for the fall of 2003.

References

[1] Bruce, K. B., Danyluk, A., and Murtagh, T. Event-driven programming can be simple enough for CS 1. In *Proceedings of the 2001 ACM ITiCSE Conference* (2001), pp. 1–4.

[2] Bruce, K. B., Danyluk, A., and Murtagh, T. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the 2001 ACM SIGCSE Symposium* (2001), pp. 6–10.

[3] Culwin, F. Object imperatives! In *Proceedings of the 1999 ACM SIGCSE Symposium* (1999), pp. 31–36.

[4] Jimenez-Peris, R., Khuri, S., and Patino-Martinez, M. Adding breadth to CS 1 and CS 2 courses through visual and interactive programming projects. In *Proc. of the 30th SIGCSE Tech. Symp. on Computer Science Education* (1999), pp. 252–256.

[5] Mutchler, D., and Laxer, C. Using multimedia and gui programming in CS 1. In *Proc. of the SIGCSE/SIGCUE Conf. on Integrating Technology in Computer Science Education* (1996), pp. 63–65.

[6] Reges, S. Conservatively radical java. In *Proc. ACM SIGCSE Symposium* (2000), pp. 85–89.

[7] Stein, L. A. What we've swept under the rug: Radically rethinking CS 1. *Computer Science Education 8*, 2 (1998), 118–129.

[8] Wolz, U., and Koffman, E. simpleIO: A Java package for novice interactive and graphics programming. In *Proceedings ITiCSE* (1999), pp. 139–142.

[9] Woodworth, P., and Dann, W. Integrating console and event-driven models in CS 1. In *Proc. of the 30th SIGCSE Technical Symp. on Computer Science Education* (1999), pp. 132–135.