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

Kim B. Bruce
kim@cs.williams.edu

Andrea Danyluk
andrea@cs.williams.edu

Thomas Murtagh
tom@cs.williams.edu

Department of Computer Science
Williams College
Williamstown, MA 01267

ABSTRACT

We have designed a CS 1 course that integrates event-driven programming from the very start. In [2] 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.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms

languages

Keywords

CS 1, event-driven programming, Java

1. INTRODUCTION

A few years ago we implemented a major update of our CS 1 course, which is now based on Java. With the support of the specially designed objectdraw library, 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.

The objectdraw library eliminates much of the syntactic overhead of writing programs in an event-driven style, yet provides an easy transition to the standard Java event-handling style of programming. The library also supports the use of object-oriented geometric figures, which allows students to write quite interesting programs in an event-driven style after only a few lectures. For example there are classes that represent framed and filled rectangles, ovals, and arcs, as well as as lines, text, and images, all of which can be created on a canvas. They all support methods to change their dimensions, location, and color, as well as to determine whether a point is contained in the graphic object. Moreover, changes to objects show up immediately on the canvas without requiring calls of `repaint`, as is required with standard Java.

Concurrency is 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 [3, 2], the combination of object-oriented graphics, event-driven programming, and concurrency provides for a very interesting and pedagogically sound introduction to programming.

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 [3] and [2], 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 [6, 11, 13]. 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 [7, 9]. Finally, line by line text input in Java is quite complicated, requiring students to understand and catch exceptions.

While many observers agree on these benefits for event-driven programming, there are concerns that event-driven programming is too difficult for novices [12, 10]. 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

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

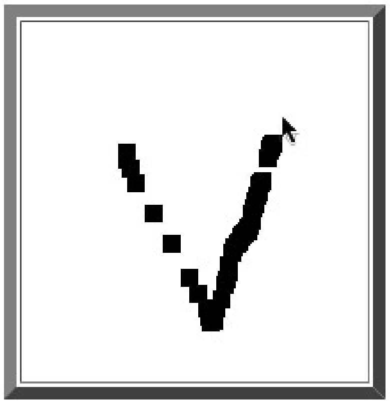

Figure 1: "V" formed by dragging the mouse while running class `Squares`

components in standard Java requires programmers to

1. Create and initialize the GUI component (e.g., create a a pop-up menu using `JComboBox` 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 by writing the method to be executed when an event is generated by the component.
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 a `JApplet`. 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. Students are told that their event-handling classes should extend the `WindowController` class and are provided with the names and signatures of methods that should be written in order to handle the appropriate mouse events.

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

```
public class Squares extends WindowController {

  // draw square at mouse location after each drag
  public void onMouseDrag(Location mouseLoc ) {
    new FilledRect( mouseLoc, 4, 4, canvas);
  }
}
```

A picture showing what would be produced if the user dragged the mouse in the shape of a "V" is shown in Figure 1.

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.

Given this, instructors typically look for good examples of pre-defined classes for students to work with so that they can at least get some experience with sending messages to objects before introducing the notion of writing classes. However, even if the instructor succeeds in finding such classes, students will likely start out writing a static `main` method that looks entirely different from the methods that appear in most classes students will design.

We use a truly object-oriented library of graphics objects (see [3]) and event-driven programming to solve these problems. Because the graphics objects are stateful and changes (e.g., a response to a `move` message) appear instantly on the screen, students get good feedback on the results of sending messages.

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 fairly interesting programs can be written without any use of loops. As an example, consider the following program that prompts 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;

  // display message to user at start-up
  public void begin() {
    new Text("Drag the mouse to draw",40,20,canvas);
  }

  // remember where mouse went down
  public void onMousePress(Location point) {
    oldPoint = point;
  }

  // draw line from last point to new mouse location
  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.

An important benefit gained from using event-driven programming is that the methods written by students are naturally short, eliminating the necessity of nagging students to break programs into smaller pieces.

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, students use a class constructor for the same purpose as the `begin` method used in extensions of `WindowController`, such as that given above.

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 predefined 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). With such examples 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 mentioned 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.

Thus 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.

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, consider the following program. It initially draws the ground and sun on the canvas. It then draws a new blade of grass each time a user clicks the mouse. The resulting picture is shown in Figure 2.

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

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

  // draw solid ground and sun
  public void begin() {
    new FilledRect(0,GROUND_LINE, SCREENWIDTH,
      SCREENHEIGHT-GROUND_LINE, canvas);
```

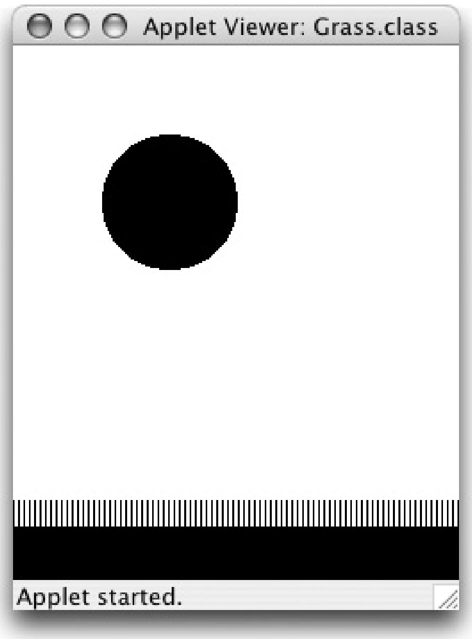

Figure 2: Sun, earth, and grass

```
    new FilledOval(SUN_INSET,SUN_INSET,
      SUN_SIZE,SUN_SIZE, canvas);
    bladePos = 0;
  }

  // grow new blade of grass with each mouse click
  public void onMouseClick(Location point) {
    if (bladePos < SCREENWIDTH) {
      new Line(bladePos,GRASS_TOP,
        bladePos,GROUND_LINE, canvas);
      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 `onMouseClick` method is designed to be 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 continue to 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 change the `if` to a `while`:

```
public class Grass2 extends WindowController {

  // constants omitted

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

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

    bladePos = 0;

  // grow blades of grass when user clicks
  public void onMouseClick(Location point) {
    while (bladePos < SCREENWIDTH) {
      new Line(bladePos,GRASS_TOP,
        bladePos,GROUND_LINE, canvas);
      bladePos = bladePos + GRASS_SPACING;
    }
  }
}
```

If, as is likely, it is desired to draw the grass on start-up, then the while loop can be moved to the end of the begin method.

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. TRANSITIONING TO STANDARD JAVA

We had several goals in the creation of the objectdraw library. For example, we wished to provide true object-oriented graphics that could be used as examples of objects early in the course, yet were of value throughout the course. More relevant to the focus of this paper, we wished to reduce the syntactic overhead of event-driven programming involving mouse actions on a canvas.

One possible disadvantage of using a library is that at some point students need to be taught the "standard" way of accomplishing the same results. Because we wished to make the transition to standard Java event-driven programming for our students as simple as possible, we made sure that the event-driven programming style supported by the library is consistent with Java's listener-based style.

While we relieved the programmer from declaring that their class extending `JApplet` implemented the appropriate listener interface and actually adding the listener to the canvas, in other respects our library encouraged students to write code that emulates the listener approach of standard Java. Mouse events trigger execution of a standard method (e.g., `onMouseClick(...)`) in their applet class. The methods are only slightly simplified from standard Java in that the formal parameter is associated with the place that the user clicks on the canvas, rather than an event that must be queried for the x and y coordinates of the click.

Once students learn this style of programming, it is easy for them to transition to using standard Java for event handling. Half-way through our course, we introduce students to Java SWING GUI components. Because they already understand the event-driven style of programming, they need only learn the syntax necessary to associate listeners with the components (and learn a bit about layout). They already understand how the methods of listeners are called by the run-time system when an event is generated. They simply have to follow the rules to create the components and associate listeners with them.

Others designing libraries for introductory courses have taken somewhat different approaches. For example, the NGP library developed at Brown [5, 1] also supports both graphics and event-driven programming. However, it supports event-driven programming using the old Java 1.0 model that was discarded with Java 1.1. In this older model one associates actions with components (e.g., buttons) by subclassing the component and overriding its action method. If one learns event handling in this way, students will have to learn an entirely new way of handling event-driven programming when transitioning to standard Java.

On the other hand, the BreezySwing library of Lambert and Osborne [8] goes farther than we have by providing predefined event-handling methods for virtually all GUI components, including buttons, menus, text fields, etc. By restricting our support for event-handling methods, we provide an opportunity for students to learn how to do standard Java event-handling once they have developed enough comfort with the language that the syntactic overhead does not get in the way.

5. CONCLUSIONS

In earlier papers [3, 2] 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 five years we have been teaching a course using these ideas and the library we developed for this purpose. We have written a text [4] based on these ideas, which will be published this winter by Prentice-Hall. Our materials have been successfully tested for the last several years by faculty and students at colleges, universities, and high schools throughout the United States. They have discovered that the use of event-driven programming, when supported by a library like our objectdraw library, has strong pedagogical and motivating factors that improve students' learning experiences.

6. REFERENCES

[1] C. Alphonce and P. Ventura. Using graphics to support the teaching of fundamental object-oriented principles in CS 1. In *OOPSLA Educators' Symposium*, 2003.

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

[3] K. B. Bruce, A. Danyluk, and T. Murtagh. A library to support a graphics-based object-first approach to CS 1. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer science education*, pages 6–10, 2001.

[4] K. B. Bruce, A. P. Danyluk, and T. P. Murtagh. *Java: An eventful approach*. Prentice Hall, 2004.

[5] D. B. Conner, D. Niguidula, and A. van Dam. Object-oriented programming: Getting it right at the start. In *OOPSLA Educators' Symposium, Portland, OR*, 1994.

[6] F. Culwin. Object imperatives! In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, pages 31–36, 1999.

[7] R. Jimenez-Peris, S. Khuri, and M. Patino-Martinez. 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*, pages 252–256, 1999.

[8] K. A. Lambert and M. Osborne. *Java: A Framework for Programming and Problem Solving*. Brooks Cole, 2nd edition, 2001.

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

[10] S. Reges. Conservatively radical java. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 85–89, 2000.

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

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

[13] P. Woodworth and W. Dann. Integrating console and event-driven models in CS 1. In *Proc. of the thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 132–135, 1999.