A Library to Support a Graphics-Based Object-First Approach to CS 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

In this paper we describe a library we have developed that supports an "OO-from-the-beginning" approach to CS 1. The design of interactive graphical programs helps students to both use objects and write methods early while designing and implementing interesting programs. The use of real graphics "objects" and event-driven programming are important components of this approach.

1 Introduction

In the fall of 1999, the authors developed a new version of the introductory Computer Science course (CS 1) at Williams College. Our goals were as follows:

- Use an object-first approach, requiring students to think from the start about the programming process with a focus on methods and objects.
- Use graphics and animation extensively. Experience in an earlier version of this course convinced us that graphics can be an important tool both because students are able to create more interesting programs, and because graphic displays provide students with visual feedback when they make programming errors.
- Introduce event-driven programming early. Most programs students use today are highly interactive. Writing programs that are similar to those they use is both more interesting and more "real" to the students. This goal was motivated, in part, by Stein [11].

As we developed the course, we found that these seemingly disparate goals complemented each other.

In this paper we discuss our goals and methods for achieving them, focusing on how our locally-designed library, ObjectDraw, allowed us to overcome potentially large hurdles to the use of Java with novice programmers.

2 Motivation

Two of our fundamental goals, the desire to use graphics and the desire to teach using an objects-first approach, provided the motivation to construct our graphics library and influenced its organization.

As we began the design of our course, the goal of emphasizing the use of an object-oriented style from the first lecture presented the most obvious challenges. Before a student can begin to learn mechanisms for defining and using classes, the student must master a significant amount of material. The student must learn to define methods. If the examples used are to adequately motivate the use of objects, the student must know how to declare and use parameter names. For all but the most trivial examples, at least conditional control structures are required. Initially, it seemed that several weeks would be required to present these prerequisites before object-oriented features could be introduced.

We felt that the most promising approach to this problem was to integrate the use of pre-defined classes of objects into our introduction to programming basics. In this way, we could familiarize the students with the object-oriented notion that work is accomplished by sending messages to objects. In addition, because students use these objects without knowing how they are implemented, they develop a sense of the appropriate level of encapsulation/abstraction to be used before learning to define their own classes.

Others who have employed this approach have often used collections of classes implementing "micro-worlds". The object-oriented version of Karel [2] provides an en-

vironment in which students write short programs to manipulate robot objects through methods. Faculty at Wellesley College have created several micro-worlds to illustrate aspects of object-oriented programming.

Our desire to use graphics extensively in the course made it clear that we would need to develop a set of classes to simplify the use of Java's graphics facilities. The Java AWT does not provide an interface that is appealing for an introductory course. The need to do all actual drawing in a single `paint` method implies that a program must maintain enough state to communicate the desired appearance of the display to the `paint` method. For all but the simplest examples, this requires complex data structures that are far out of reach during the first weeks of an introductory course.

While we initially viewed the need for a graphics library and the need for an appropriate "micro-world" as independent problems, we quickly realized that a single library could fulfill both needs. We saw that if we could design a graphics library simple enough to provide the means to introduce students to the use of objects and methods, it would actually be better than using a micro-world because it would be integrated into the entire course. Accordingly, we set out to construct a graphics library whose design would emphasize the object-oriented approach.

3 An Object-oriented Graphics Library

Others have recognized that the graphics facilities provided by Java's AWT were unsuitable for an introductory course and created libraries to simplify graphics programming. The methods defined by such libraries perform the requested drawing in an off-screen buffer. The libraries also include a `paint` (or `draw`) method that copies this off-screen buffer to the screen.(See [5] and [1], for example.)

The resulting libraries generally fail to exhibit object-oriented concepts. Their designs typically parallel the underlying Java AWT Graphics class. There is a single object with a name like "pen" (or "turtle" in the case of Slack's turtle graphics [9]) which accepts a long list of drawing methods. In many such libraries, there are no run-time objects corresponding to the geometric shapes displayed on the screen. In some, one can create "rectangles" and/or "lines" as objects, but the degree to which these correspond to the objects on the screen is limited. One may be allowed to provide a "rectangle" object as a parameter to a "pen" drawing method, but if there are any methods available to modify the rectangle object itself, applying them will have no immediate effect on the display.

In designing our library we took a different approach which we felt would enable us to make the behavior of objects very concrete for our students. We provide classes for a series of objects that can be produced on the display: `Lines`, `Rectangles`, `Ovals`, `Text`, etc. When one of these objects is constructed using the "new" operator, it actually appears on the screen. In addition, there is a list of methods that can be used to modify each of these objects: `move`, `setColor`, `setWidth`, etc. Again, if any of these methods are invoked, the screen is updated (essentially immediately) to reflect the requested change.

Internally, the implementation of our library is slightly different from the off-screen buffer approach described above. We define a class called `DrawingCanvas` that represents a drawing area on the screen. When a graphical object is created, the programmer must specify the canvas on which it should appear. Each canvas is implemented as a list of graphical objects rather than as an off-screen bit map. When either the system requests a repaint of the screen or the user's code modifies any object, our library's version of `paint` erases the entire screen and redraws all the objects in each canvas. Fortunately, we found this simple technique performed efficiently enough that screen updates appear to occur instantaneously.

This collection of graphical objects and the methods associated with them have provided an excellent framework for introducing the notion of objects to our students. The fact that the objects produced by constructors are not abstract, but are concrete and visible on the screen, makes it very easy for students to appreciate the connection between their code and its behavior.

Introducing the library takes very little time. We constructed an application that provides a virtual sandbox through which students can experiment with the effects of invoking the constructors and methods associated with these classes by clicking on buttons in a display. In our first lab, after only one session of class time devoted to Java and the graphics library, students are able to familiarize themselves with the library by completing a set of guided exercises in this environment. By the end of the lab period, they abandon the "virtual sandbox" and write a Java applet that uses our library to produce a diagram of a simple road sign that varies in reaction to mouse actions.

4 Advantages of Event-Driven Programming

In addition to the graphics features described in the preceding section, our library provides support for the use of an event-driven style of programming. Introducing event-driven programming was one of our fundamental goals. Surprisingly, we have found that using this approach simplifies the process of preparing students to use object-oriented techniques. In particular, the use of

the event-driven style helps us familiarize our students with the definition of methods and the overall structure of classes.

The first programs written by students in our course are defined as classes that extend a `WindowController` class from our library. `WindowController` is itself a slight extension of the Java Applet class. It creates a `DrawingCanvas` in which the students can place graphical objects, and it allows them to handle mouse events by defining event-handling methods with names like `onMouseMove` and `onMouseDown`. These methods are very similar to the mouse event-handling methods of the Java AWT. The main difference is that they expect a simple parameter describing the coordinates where the mouse event occurred, rather than a more complex "Event" object.

From the start, our students' programs are class definitions that consist of lists of short event-handling method definitions. Such a class definition is much more typical of other classes than a class definition containing a single, large method that functions as a "main" program. While our students are not at first conscious of the possibility of generalizing the use of the class construct to define new objects, they learn to view methods as a means of describing how a single object (their program) should respond to outside stimuli.

The definition of event-handling methods also has the advantage that the introduction of formal parameters is separated from the introduction of actual parameters. Students use formal parameters when defining event-handling methods, but do not have to worry about where the actual values come from. At the same time, they are actively using actual parameters when invoking the constructors and methods of our graphics library. By the time we introduce the definition of new classes of objects, students are comfortable with both notions and are well prepared to deal with the idea that a formal name written in one part of their code can refer to an actual parameter from another part.

In conjunction with the use of objects through our graphics library, the event-handling style of programming provides an excellent way to prepare students for the introduction of classes. By the time we begin asking students to define their own classes, they have already used all of the required language mechanisms. Instead of explaining parameter passing or class syntax, we can focus on the role of objects.

To provide a concrete sense of the features of both our graphics library and the use of event-driven techniques in our course, in Figure 1 we show a simple program that uses our library. When this program is run, the `begin` method, which is similar to the `init` method for Applets, is executed. It constructs a red ball, a box, and a text item that all show up immediately on the screen. When the mouse button is pressed, the program remembers the coordinates, `point`, where it was pressed, and whether `point` is actually within the ball. When the user drags the mouse, the ball is moved if the ball was originally grabbed. When the mouse is released, the ball is moved back to its starting position if the ball was being dragged and the release point is inside the box.

```
public class DragBall extends WindowController
{
  private FilledOval ball; // ball
  private FilledRect box;  // box
  private Coords lastPos;  // last mouse posn

  // whether the ball has been grabbed
  private boolean ballGrabbed = false;

  // make the box
  protected void begin()
  {
    ball = new FilledOval(95,50,10,10, canvas);
    ball.setColor(Color.red);
    box = new FilledRect(90,150,20,20, canvas);
    new Text("Drag the ball!", 75,20, canvas);
  }

  // Save starting point and if point in box
  protected void onMousePress(Coords point)
  {
    lastPos = point;
    ballGrabbed = ball.contains(point);
  }

  // if mouse is in box, then drag the box
  protected void onMouseDrag(Coords point)
  {
    if ( ballGrabbed ) {
      ball.move( point.getX()-lastPos.getX(),
                 point.getY()-lastPos.getY());
      lastPos = point;
    }
  }

  // if dragged into box, move back to start
  protected void onMouseRelease(Coords point)
  {
    if (ballGrabbed && box.contains(point))
      ball.moveTo(200,100);
  }
}
```

Figure 1. Simple objectdraw program

5 Lab Assignments using the ObjectDraw Library

To clarify the sequence in which we introduce topics to our students and to demonstrate the flexibility our library provides, we provide a sketch of the programming assignments we have used during the first few weeks of our course. Students are assigned one program per week through most of the semester. As mentioned earlier, students begin in the first week (after only one real lecture session) with a tutorial-format introductory lab which introduces them to the use of our graphics classes.

Lab 1. The week after the tutorial lab, students write a "laundry sorter" program. When completed, the program allows its user to drag colored rectangles representing items of clothing to one of three bins designated for whites, darks and colors. The code for this program is structurally similar to that for the example program shown in Figure 1. It is more complicated in that the students must include code to create several bins rather than one, code to generate items of random colors and an if statement to see if the correct bin was selected.

Lab 2. The next lab tests students' abilities to design and implement classes. The assignment is to construct a magnet class, where a magnet is represented as a rectangle with the poles located near the ends. Two magnets are drawn on the screen, and the program enables users to drag them about. When one magnet is moved close to the other, it attracts or repels the other. The instructors provide the class definition for the poles. Again no loops are needed.

Lab 3. In the third week of class, we introduce students to the idea of simple animation. More fundamentally, however, students are introduced to the notions of loops and a simple form of concurrency. The lab involves implementing a simple game in which a user tries to drop a ball into a target box. Students define a `Ball` class that extends `ActiveObject`, a class provided in our ObjectDraw library to handle the management of threads. This lab also serves as an exercise in thinking carefully about parameters.

Lab 4. The following week's lab is to program the *Frogger* game. Images of cars move across the screen to form four lanes of traffic. The user controls a frog's attempts to hop across the road without being run over by clicking on the mouse button in the direction the frog is to hop. Each car is controlled by a separate thread, encapsulated as an `ActiveObject`.

Later labs, emphasizing topics such as arrays, Strings, files, and recursion, were easy to design and implement with our ObjectDraw library, including a final program in which students implemented a simplified version of *PacMan* or *Space Invaders*.

6 Related Work

Many have discussed the tension between the desire to limit the complexity faced by beginning programmers and the desire to present the object-oriented approach from the very start of an introductory course.

Reges [6] proposes providing students with applications which include the GUI components, but omit one or more classes, which are then assigned to students to implement. This approach has the advantage of making it easy for students to use GUI components without the overhead of having to learn AWT or Swing.

Buck and Stucki [3] propose a technique similar to Reges. They provide code to handle user interface methods and then ask the student to add code to a class to complete the implementation. They limit the degree to which students address design issues even further than Reges by specifying the methods the student should implement and their parametrizations.

We fear that these approaches will leave students feeling that they have no understanding of how to write complete programs. They are essentially stuck with being dependent on faculty-written frameworks for which they write only relatively insignificant pieces. In some sense, of course, our students are equally dependent on our library to construct complete programs. The difference is that our library provides general purpose primitives similar to the system library rather than providing support that is limited to a particular programming assignment. As a result, students perceive the experience of using our library as similar to the experience of writing programs using only system supplied libraries and enjoy the associated satisfaction.

Another technique that has been used to eliminate complexity from the first encounters students have with programming is the use of "micro-worlds". For example, the object-oriented version of Karel [2] provides an environment in which students can write short programs to manipulate robot objects through methods. In our view, the graphics library we provide has many of the advantages of a micro-world. It employs a limited and simple vocabulary of commands. It enables beginning programmers to get immediate feedback regarding the behavior of their programs. There is little real difference between telling Karel to move and telling one of our graphical objects to move. The graphics library, however, has the additional advantage that it remains a useful tool throughout the course for the development of a wide range of programs.

Our graphics library is particularly similar to the Widget package developed by Roberts [7, 8]. Both systems are based on a collection class (the "canvas" or "collage") to which a student can add instances of state-

ful graphical objects. Roberts's package is more general than ours, providing a CompoundWidget class designed to let the programmer extend the set of supported graphical shapes. On the other hand, Roberts does not provide mechanisms to enable beginners to construct event-driven programs. His package includes a "waitForClick" method apparently designed to support the "turn taking" approach to user interaction advocated by Wolz [12].

Conner et al [4] advocate an object-first approach that depends on a graphics library, NGP, that also integrates event-driven programming. Their graphics library also includes a collection of GUI components and library-specific ways of laying out components. Both graphics and GUI components react to events using an event-driven model similar to that of Java 1.0. Behavior is associated with objects by defining a subclass of the component that overrides the default behavior. In particular, this approach to event-driven programming unnecessarily mixes GUI appearance and behavior with the application behavior. Our own library is both simpler and more restricted in scope, and makes it easier to migrate to the Java 1.1 event model.

Readers familiar with the work of Lynn Andrea Stein [10] and her Rethinking CS101 project at MIT will recognize the impact of her thinking on this project. Our approach is simultaneously more radical and more conservative than Stein's. When we looked at early versions of the text she is developing for her course, we were surprised to find that the sequence of topics in the beginning of the course was quite conservative. Chapters 1 through 6 cover the traditional topics of built-in data types, expressions, and statements (including conditionals and loops). Chapter 7 introduces classes, objects, and methods. Chapter 9 introduces threads for animate objects, while event-driven programming is not covered until chapters 15 and 16. Instead, we take a more radical approach and introduce event-driven programming in the first week of the term, with discussions of concurrency by week 4. On the other hand, we are more conservative in that we do not introduce some of the more sophisticated applications such as networking, client-server interaction, etc., that Stein includes in her course.

7 Conclusion

The use of our ObjectDraw library has enabled our course to focus on the key concepts of object-oriented programming without overwhelming students with the complexity of using raw Java in the first few weeks. While the graphics classes were used unchanged throughout the term, students were weaned from the simplified event-driven model presented early in the course, to the more complex world of listeners with more varied events and GUI components. By the second course, the library is not used by the students at all.

Our library allowed us to focus on objects and methods from day one, and provided students with the tools to apply these techniques without excessive overhead. While we are continuing to refine the library, we feel that it is an important tool for introducing novices to object-oriented programming.

References

[1] Bailey, D. A., and Bailey, D. W. *Java Elements.* McGraw Hill, 2000.

[2] Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming.* John Wiley and Sons, 1996.

[3] Buck, D., and Stucki, D. J. Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. In *Proceedings ACM SIGCSE Symposium 2000* (2000), pp. 75–79.

[4] Conner D. B, Niguidula D, A. v. D. Object-oriented programming: Getting it right at the start. In *OOPSLA Educators' Symposium, Portland, OR* (1994).

[5] Horstmann, C. *Computing Concepts with Java.* John Wiley and Sons, 1998.

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

[7] Roberts, E. The Widget package for Java. Tech. rep., Stanford University, 2000. http://cse.stanford.edu/java/widget/index.html.

[8] Roberts, E., and Picard, A. Designing a Java graphics library for CS 1. In *Proceedings of the 3rd annual ITiCSE* (1998), pp. 213–218.

[9] Slack, J. M. *Programming and Problem Solving with Java.* Brooks/Cole, Thomson Learning, 2000.

[10] Stein, L. A. What we've swept under the rug: Radically rethinking CS1. *Computer Science Education 8*, 2 (1998), 118–129.

[11] Stein, L. A. *Interactive Programming in Java.* Morgan Kaufmann Publishers, 2001.

[12] Wolz, U., Weisgarber, S., Domen, D., and McAuliffe, M. Teaching introductory programming in the multi-media world. In *Proceedings ITiCSE* (1996), pp. 57–59.