

A library to support a graphics-based object-first approach to CS 1

Kim B. Bruce*, Andrea Danyluk, and Thomas Murtagh
Williams College

May 23, 2000

Abstract

In this paper we describe a library that we have developed which 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. The old version of the course covered relatively traditional CS 1 material using the procedural language Pascal. A strength over previous versions of this course was the reliance on the use of simple computer graphics and animations to make the programs more interesting and to provide visual cues to errors in programs.

A primary goal of the new course was to introduce the concepts of object-oriented programming and design in a simple and well-motivated fashion that would allow students to gain extensive hands-on experience in mastering the programming process. Other priorities included introducing event-driven programming early in the course and using graphics extensively. As we developed the course, we found that these seemingly disparate goals complemented each other. Our goals can be summarized 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. Based on our successful experience in the earlier procedural version of this course, we found graphics an important tool both because students were able to create more interesting programs, and because graphic displays allowed students to receive visual feedback when they made programming errors.
- Introduce event-driven programming early in the course. Most of the 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. Our desire to introduce event-driven programming early was motivated, in part, by Stein [Sla00a].

*Corresponding author: Kim B. Bruce, Department of Computer Science, Williams College, Williamstown, MA 01267. kim@cs.williams.edu, (413) 597-2273, FAX: (413) 597-4116.

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

2 Obstacles

Initially, the item in our list of goals that seemed most daunting was the desire to emphasize the use of an object-oriented style from the very start of the class. Before a student can begin to use a language's mechanisms for defining and using classes in meaningful ways, the student must master a significant amount of material. The student must learn to define methods. If the examples used are to be rich enough to adequately motivate the use of objects, the student must know how to declare and use parameter names within a method. 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.

Other introductory courses and texts have addressed this problem in two main ways. The first and least satisfactory way is to accept the need to delay the presentation of object-oriented programming facilities and to attempt to compensate by giving an abstract overview of the notions of "class" and "object" before concretely discussing programming constructs. Unfortunately, in our opinions, such attempts to present the philosophy underlying object-oriented design are largely lost on students who have little or no concept of the nature of computer programming in any style.

Another approach is to introduce programming basics by providing examples of the use of pre-defined classes of objects. The goal in this approach is to familiarize the students with the notion of accomplishing work by sending messages to objects. The fact that the students use the objects without understanding how they are designed has the additional, potential benefit of suggesting the appropriate level of encapsulation/abstraction to be used once students learn to define their own classes.

This approach is the basis for several "micro-worlds" designed to introduce object-oriented programming. The object-oriented version of Karel [BSRP96] provides an environment in which students can write short programs to manipulate robot objects through methods. Faculty at Wellesley College have created several micro-worlds to illustrate different aspects of object-oriented programming. Approaches that make use of micro-worlds early in a course must eventually drop them in order to provide students with a complete, practical introduction to programming. An unfortunate possible consequence is that students may view their exposure to the micro-world as an amusing interlude with little relevance to their "serious" introduction to programming.

To avoid the need to use and then abandon a set of objects to introduce programming, others have searched for sets of objects that were built in to the existing programming framework that could serve as archetypal "objects" during the first few weeks of the course. Arnow and Weiss [AW00] make use of the Java String class and its methods early in their text. Unfortunately, Strings in Java have the shortcoming that all String methods produce new objects rather than modifying existing objects. This will tend to leave students with a skewed version of the object concept.

An alternative to the approaches described above is to design from scratch a collection of classes that can serve as a fundamental programming tool throughout the course. Since they are designed from scratch, like micro-worlds, these classes can provide an introduction to object-oriented programming at the appropriate level.

3 An Object-oriented Graphics Library

Our decision to design a set of classes that would provide an early introduction to the object model was motivated by an apparently unrelated design goal for our course: the desire to incorporate graphics and animations in the students' programming projects. This had been a technically simple thing for us to accomplish when our course was taught in Pascal on Macintoshes. The Quickdraw graphics library was simple enough to be accessible to beginners.

The Java AWT does not provide an interface that is as appealing for an introductory course. The need to do all actual drawing in a single `paint` method implies that any program that updates the screen 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.

Others have solved this problem by creating libraries that enable students to write programs that can invoke simplified drawing primitives. 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 [Hor98], [BB00], for example.)

The resulting libraries generally fail to exhibit object-oriented concepts. Their designs typically parallel the underlying Java AWT Graphics class. There is generally a single object with a name like "pen" (or "turtle" in the case of Slack's turtle graphics [Sla00b]) 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

The desire to incorporate graphics into our course provided the motivation to develop an environment in which students could begin working with a rich and practical collection of classes of objects. It did not, however, directly address one requirement for the introduction of classes. Before constructing classes students must be introduced to the mechanics of defining methods and using formal parameters. An approach to handling this arose from another independent goal for our course: the desire to emphasize event-driven programming.

One can introduce Java programming by having students construct simple applications. In this case, students begin by placing all their code in a method named `main`. As the complexity of the code in `main` increases, one would typically introduce the ability to define private methods and teach students how to *procedurally* decompose the code they want to place in `main` into small, logically-structured private methods. The problem with this approach is that in following it, one is teaching a procedural rather than an object-oriented approach. Students are led to think of methods as a mechanism to decompose larger tasks rather than as the means to describe the possible behaviors of an object.

In our course, from the very start we had students construct programs in a framework based on the Java Applet class. Their first programs were described as a class that extended a `WindowController` class from our library that itself was a slight extension of the Java Applet class. Our `WindowController` extends the Applet class in several simple ways. 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 slightly simpler parameters. Most of our event-handling methods receive a single parameter describing the coordinates where the mouse event occurred, rather than a more complex “Event” object.

As a result of this approach, our students never imagined the idea of a large main program. From the start, their programs were actually class definitions that consisted of lists of short method definitions. While they were not conscious of the possibility of generalizing the use of the class construct to define new objects, they did view methods as a means of describing how a single object (their program) should respond to outside stimuli. This view remained clear to them throughout the course. After years of pleading with students writing in Pascal to decompose their code into shorter procedures, it was shocking to find that this was simply no longer an issue.

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 used formal parameters when defining event-handling methods, but didn’t have to worry about where the actual values came from. They accepted the idea that the system somehow provided this information to their methods. At the same time, they were actively using actual parameters when invoking the constructors and methods of our graphics library. By the time we introduced the definition of new classes of objects, students were comfortable with both notions and were well prepared to deal with the idea that a formal name written in one part of their code would 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 provided an excellent way to prepare students for the introduction of classes. By the time we began asking students to define their own classes, they had already used all of the required language mechanisms. Instead of explaining parameter passing or class syntax, we could focus on the role of objects.

5 Sample Lab Assignments using the ObjectDraw Library

While there is not enough space here to discuss our teaching strategies in detail, we provide a sketch of sample programs assigned to students 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 the graphics classes in an environment that allows them to use buttons, pop-up menus, and dialog boxes to construct graphical objects and send messages.

Lab 1. The first lab after the tutorial lab has students write a “laundry sorter”. A colored rectangle representing an item of clothing is drawn on the top of the screen, with three bins drawn below it. The three bins are for light, dark, and colored clothing. The user is to drag the laundry item to the appropriate bin. If the user drops the item in the correct bin, a new item is drawn in a randomly-chosen color. If it is dropped in the incorrect bin, the item is redrawn at the top of the screen. At the time this program is assigned, students have just been introduced to conditional statements. Because students use event-driven programming, no loops are necessary. All of the interesting parts of the program are contained in methods corresponding to mouse-button presses, drags, and releases.

Lab 2. After two full weeks of classes, the assigned lab tests students’ abilities to design and implement classes. The assignment is to design and implement a magnet class, where a magnet is represented as an elongated rectangle with the north and south poles located near each end. Two magnets are drawn on the screen, and a user can drag either one around the screen by clicking and dragging it. If the two magnets are close to each other, the moved magnet 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: graphical objects moving within the drawing canvas. More fundamentally, however, students are introduced to the notions of loops and a simple form of concurrency. Students define classes that extend `ActiveObject` which is provided in our ObjectDraw library to handle the setting up of Runnable objects. The lab involves implementing a simple game in which a user tries to drop a ball into a target box. The game has a number of difficulty levels that are defined by the height from which the ball must be dropped and the size of the box. This lab requires students to implement a number of classes: the Controller that allows for user interaction, a Ball class, and a Target class. This lab serves as an exercise in thinking carefully about parameters.

Lab 4. The following week’s lab is to program the *Frogger* game. In this program, cars move across the screen to form four lanes of traffic. The user controls a frog’s attempts at hopping across the road by clicking on the mouse button in the direction the frog is to hop. If the frog is hit by a car, it “dies”, and then is resurrected to try again. Each car is controlled by a separate thread, encapsulated as an `ActiveObject`.

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

6 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.

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

- [AW00] David Arnow and Gerald Weiss. *Introduction to Programming Using Java: An Object-Oriented Approach*. Addison-Wesley, 2000.
- [BB00] Duane A. Bailey and Duane W. Bailey. *Java Elements*. McGraw Hill, 2000.
- [BSRP96] Joseph Bergin, Mark Stehlik, Jim Roberts, and Richard Pattis. *Karel++: A Gentle Introduction to the Art of Object-Oriented Programming*. John Wiley and Sons, 1996.
- [Hor98] Cay Horstmann. *Computing Concepts with Java*. John Wiley and Sons, 1998.
- [Sla00a] James M. Slack. *Interactive Programming in Java*. Morgan Kaufmann Publishers, 2000.
- [Sla00b] James M. Slack. *Programming and Problem Solving with Java*. Brooks/Cole, Thomson Learning, 2000.