Using Graphics to Support the Teaching of Fundamental Object-Oriented Principles in CS1

Carl Alphonce
Dept. Comp. Sci. & Eng.
University at Buffalo, SUNY
Buffalo, NY 14260-2000
alphonce@cse.buffalo.edu

Phil Ventura
Dept. Comp. Sci. & Eng.
University at Buffalo, SUNY
Buffalo, NY 14260-2000
pventura@cse.buffalo.edu

ABSTRACT

Teaching object-oriented programming in CS1 is hard. Keeping the attention of CS1 students is perhaps even harder. In our experience the former can be done successfully with very satisfying results by focusing on the fundamental principles of object-orientation, such as inheritance, polymorphism and encapsulation. The latter can be done by having students create graphical event-driven programs. Care must be taken, however, since teaching graphics can easily distract students and certainly takes time away from the fundamentals being taught. We use Java as a vehicle for OO instruction, but rather than expose CS1 students to the intricacies of Swing we employ an elegant and small graphics package called NGP. NGP allows students to create event-driven graphical programs using only inheritance and method overriding. We describe how we use NGP to enhance rather than detract from our teaching of fundamental OO principles.

Categories and Subject Descriptors

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

General Terms

Design

Keywords

Object-orientation, CS1, Java, NGP, graphics

1. CONTEXT OF EXPERIENCE

1.1 The course

We have been teaching object-orientation in CS1 using Java for many years. The course has evolved a great deal since its early Java-based offerings. The course now emphasizes design in an objects-first approach.

Objects are introduced to students in the very first day of classes, and object-oriented fundamentals such as inheritance, polymorphism and encapsulation are introduced to the students within the first few weeks.

By the end of the course students are able to write reasonably sophisticated event-driven graphical programs. The final project typically requires students to implement a well-known computer game such as Tetris or Nibbles.

1.2 The students

The course enrolls 200 to 300 students per semester, spread over three to four lecture sections and twelve to sixteen lab sections. Although our CS1 course has no formal course prerequisites, approximately half of the students have prior programming experience, though only rarely is this experience truly object-oriented.

Our students, like the majority of today's students, are well acquainted with the use of computers and therefore with graphical event-driven programs. Writing text-based programs rarely captures their attention. In fact, it typically does just the opposite.

2. DESCRIPTION OF EXPERIENCE

2.1 Piquing and keeping students' interest

Teaching object-oriented programming in CS1 is hard. Keeping the attention of CS1 students is perhaps even harder. Making course materials and course projects appealing to students is a prerequisite to learning. In early August of this year a thread (*Simplifying Java for CS1*) discussing the use of text I/O arose on the SIGCSE list. We feel compelled to quote a highly relevant comment by Ian Utting in this thread:

> In the olden days, when I learnt programming, printing out a triangle made of asterisks (on the ASR-33 console) was a significant achievement. Today's students, bought up on cinema-quality shoot-em-ups, are much less impressed.

One of the authors has a similar recollection. Printing a Celsius-to-Fahrenheit conversion table was a significant achievement. User input was often not considered at all. Today's students have grown up in an environment in which computers are a part of everyday life. Text-based programs

Copyright is held by the author/owner.
OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

are not only unimpressive to these students, they are foreign to them. Students use graphical event-driven systems such as web browsers, word processors, automated banking machines, self-service checkouts and even pay-at-the-pump systems at gas stations every day.

2.2 Our philosophy

Like many other educators, we have found that using graphics in a CS1 course can capture and hold the interest of our students. A significant challenge for educators is to make course materials interesting to their students without giving up course content and without getting bogged down in irrelevant details of a graphics package.

There are, of course, many ways in which one can do this. *Alice* (`www.alice.org`) is an excellent example of an environment in which students can explore programming concepts in a graphically rich environment. The intent of this paper is not to survey existing environments or promote a particular one (though we do discuss the graphics library we have found useful). Instead our goal is to highlight how the use of graphics in the classroom can serve the dual purpose of keeping the attention and interest of students while simultaneously supporting instruction of fundamental object-oriented concepts.

In our environment we use Java as a vehicle for OO instruction, but rather than expose CS1 students to the intricacies of Swing we employ an elegant and small graphics package called NGP ("Nice Graphics Package"), developed at Brown University by Andries van Dam.

2.3 We're not teaching the library

A concern often expressed, one echoed in the SIGCSE mail list discussion mentioned above, is that non-standard libraries are a waste of time since students will not use them outside the scope of the one course. Our response to this is that we are not teaching students the library, we are teaching students object-orientation *using the library as a supportive mechanism* to accomplish this. We are no more interested in teaching students AWT/Swing in the first course than we are in teaching them NGP. Trying to use raw AWT/Swing in the first course is more complex than using NGP because of the large amount of background understanding of object-oriented concepts required to use it. Students cannot write graphical event-driven programs using AWT/Swing until much later in the course, when they *already understand* the object-oriented concepts.

An obvious way around this is to provide students with classes which hide the complexity of AWT/Swing. This is what NGP does for us. It allows us to focus on using graphical event-driven programs to deepen our students' understanding of object-oriented concepts.

2.4 NGP in brief

NGP provides basic yet very functional graphical containers and components. Here we give a thumbnail sketch of NGP. For those who are interested Brown University has a more comprehensive NGP tutorial on the web:

`www.cs.brown.edu/courses/cs015/2002/Docs/tutorial`

Basic containers include (among a handful of others) `Row` and `Column`. These are essentially `JPanel`s with layout managers already installed. A `Row` arranges items placed within it in a row, while a `Column` arranges its contained items in a column. The parent (containing) container of any NGP container or component is given as an argument in its constructor. The superclass constructor `add`s the item to the parent container automatically. NGPs pallet of containers is not as varied or as flexible as that found using raw AWT/Swing, but a surprising amount of very reasonable layout can be accomplished using inly `Row`s and `Column`s.

`NGP.Component` classes, such as `PushButton`, provide event-driven reactivity by in effect making instances be their own event handlers. For example, a `PushButton` object provides a `release()` method which a subclass can override to define a behavior. The `actionPerformed` method of (an inner class event handler of) the superclass calls the superclass' `release()` method, which is defined with an empty body. This complexity is hidden from the students by design. To specify what should happen when a button is clicked a student needs only to subclass the `PushButton` class and override the `release()` method.

The same approach is taken with `NGP.Graphics` classes (shapes and images) which can be placed in a special container called a `DrawingPanel`. Objects instantiated from `NGP.Graphics` classes such as `FilledRectangle` and `Filled Ellipse`, are represented by shapes and are reactive. In the simplest case a student needs only to subclass the `Filled Rectangle` (or similar) class and override the `react()` method to specify what should happen when the mouse is clicked over the shape.

Using such containers and components students are able to create graphical event-driven applications without needing to know about things such as layout managers or event listeners. Students are able to build "cool" programs while learning about inheritance, the importance of being able to specify a method in a superclass but define it in a subclass, polymorphism and polymorphic dispatch, the role of interfaces, and so on.

2.5 Teaching fundamental object-oriented principles

A good starting point when teaching students about a new concept is to ground it in something our students are already familiar with. Luckily, concepts such as inheritance and polymorphism are natural to students from everyday life.

Inheritance is seen in typologies of many sorts. An example surely many educators have used is that of animals, mammals, reptiles, dogs, cats and other creatures arranged into an inheritance hierarchy.

Polymorphism and polymorphic dispatch is also fairly easy to explain in everyday terms. The first author frequently tells his class,[1]

> Assume that all pets have tails. Suppose I know that John has a pet and that Mary also has a pet. Unbeknownst to me John has a cat and Mary has a dog. I only know that each one has a pet. If I pull John's pet's tail it growls at me. If I pull Mary's pet's tail it hisses at me and scratches me. The same message results in different behaviors.

Students understand this.

Understanding these concepts and applying them in a programming environment is a bit trickier. Graphical reinforcement of the concepts is therefore quite helpful, especially if it

[1]No pets were harmed in the making of this example.

helps students tie the abstract programming notion to their grounded real-world understanding of the concept. The goal of this paper is to demonstrate one way in which this can be done using graphical event-driven programs to support the teaching of these concepts. A key component of the approach is to use a graphical package which provides such pedagogical support. We have found the NGP library to be effective in this regard, though other libraries or home-grown approaches could work just as well.

2.6 Creating objects

On the first day of class we present and discuss the notion of an object, noting that objects have properties and behaviors, and show how objects can be created. The students' first programming exercise requires them to create instances of pre-defined classes. These classes are graphical in nature and thus provide visual feedback.

At this stage of the course we provide students with two files as a starting point for their laboratory work. These two files are an `html` file for running an applet as well as a skeleton for an Applet which they are instructed to fill in:

```
package Example1;
public class Applet extends EdSymp.Applet {
    public Applet () {
    }
}
```

The superclass `EdSymp.Applet` is a subclass of the NGP-provided `Applet` class and adds a static `DrawingPanel` for displaying `NGP.Graphics` objects.

Since the `DrawingPanel` is static we can easily define specializations of `NGP.Graphics` objects to display automatically on that (single) `DrawingPanel`. For example, we can define an `EdSymp.Circle` class, a subclass of the `NGP.Graphics.FilledEllipse` class, which displays a red circle on the `Applet`'s `DrawingPanel` at a random location as follows:

```
package EdSymp;
public class Circle
       extends NGP.Graphics.FilledEllipse
{
   public Circle() {
      super(EdSymp.Applet._drawingPanel);
      setDimension(new java.awt.Dimension(20,20));
      setCenterLocation(EdSymp.Applet._drawingPanel.
                                             randomPoint());
      setColor(java.awt.Color.RED);
    }
}
```

Students do not see this code, however. In order to create an instance of the `EdSymp.Circle` class they need only add the code to instantiate this class in the skeleton we provide:

```
package Example1;
public class Applet extends EdSymp.Applet {
    public Applet () {
        new EdSymp.Circle();
    }
}
```

Figure 1 shows the result of running this applet.

Of course, creating a single object is not as interesting as creating multiple objects:

```
package Example2;
public class Applet extends EdSymp.Applet {
    public Applet () {
        new EdSymp.Circle();
        new EdSymp.Circle();
        new EdSymp.Circle();
    }
}
```

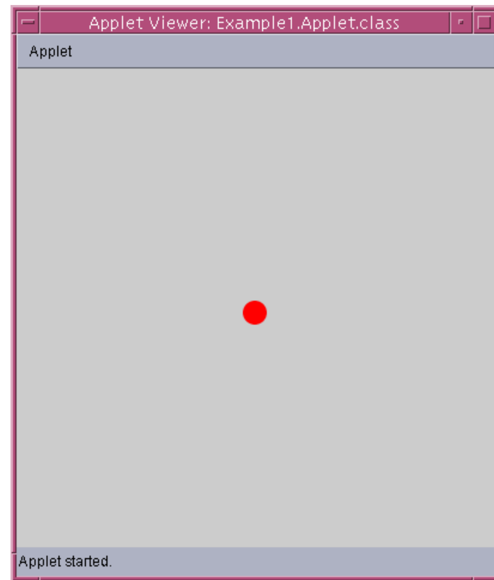

Figure 1: Applet with one `EdSymp.Circle` object represented

Doing this results in a display such as that shown in figure 2. Because each `EdSymp.Circle` object displays at a randomly generated position on the `Applet`'s `DrawingPanel` the students receive visual reinforcement that the three different class instantiations they put into their source code really does result in three distinct objects, each with a different value for the object's location property.

None of this is new or revolutionary. The point to note is how little code the students need to be exposed to in order to make this happen. To this point we have only considered object creation. To demonstrate inheritance we have students create a subclass of an existing class, perhaps an NGP component such as `PushButton` or one we provided to them like `EdSymp.Circle`. To define a class of circles similar to `EdSymp.Circle` but whose instances are `java.awt.Color.WHITE` a student may write:[2]

```
package Example3;
public class MyCircle extends EdSymp.Circle {
    public MyCircle () {
        super();
        this.setColor(java.awt.Color.WHITE);
    }
}
```

and modify their `Applet` to create a few instances of each kind of circle:

```
package Example3;
public class Applet extends EdSymp.Applet {
```

[2]The examples in the paper are representative of examples we discuss in lecture. They are, however, selected and modified to assist in the paper presentation and figure printing. We also discuss more active, behavior-based examples whose screenshots do not reproduce well on a static, printed page.

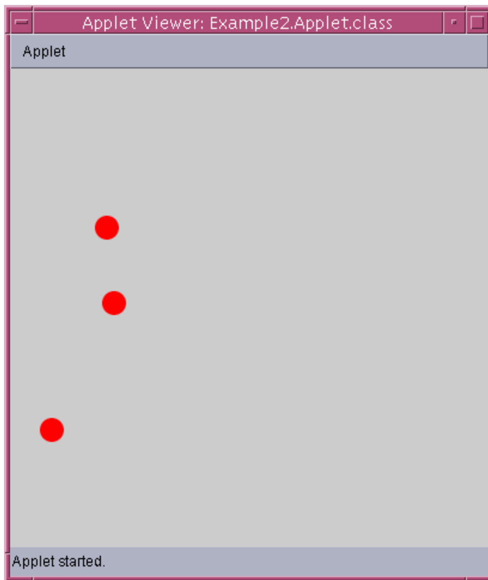

Figure 2: Applet with many `EdSymp.Circle` objects represented

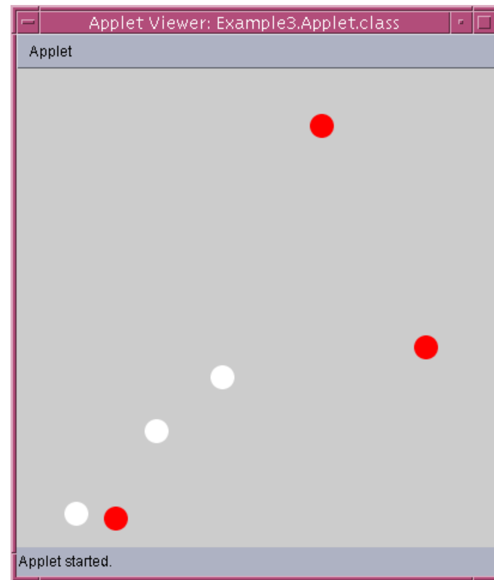

Figure 3: Applet showing representations of objects of two different classes

```
        public Applet () {
            new EdSymp.Circle();
            new EdSymp.Circle();
            new EdSymp.Circle();
            new Example3.MyCircle();
            new Example3.MyCircle();
            new Example3.MyCircle();
        }
    }
```

to produce what is shown in figure 3. Students receive visual reinforcement of what they have written in code, with very little overhead.

2.7 Polymorphism

An example we often use to introduce polymorphism is a state button. A state button is a `PushButton` which can be in different states. A very simple example is a button whose text alternates between "On" and "Off" when clicked. A class diagram for this is shown in figure 4. Of course, once the state pattern is set up it is straightforward to incorporate more complex behaviors into the states if so desired.

This example is interesting because, though small, it touches on all of the following concepts:

- object creation
- inheritance
- overriding
- polymorphic dispatch

and allows us to talk about the state pattern. Because NGP hides the mechanics of event handling students need only write the following code to create such a button class.

```
    package Example4;
    public class StateButton
           extends NGP.Components.PushButton
    {
        private ButtonState _state;

        public StateButton() {
            super(EdSymp.Applet2._column, "");
            _state = new OffState();
            _state.activate(this);
        }

        public void release() {
            _state = _state.next();
            _state.activate(this);
        }
    }
```

Note that the code which defines the behavior of the button when clicked is simply:

```
        public void release()
        {
            _state = _state.next();
            _state.activate(this);
        }
```

Rather than get bogged down with the details of AWT/Swing event handlers students can focus on the relevant concepts being reinforced.

Condensed somewhat in format to conserve space, the rest of the code required for this example is shown below:

```
    public interface ButtonState {
       public ButtonState next();
       public void activate(StateButton b);
    }

    public class OffState implements ButtonState {
       public ButtonState next() {
          return new OnState();
       }
       public void activate(StateButton b) {
          b.setText("Off");
       }
    }
    public class OnState implements ButtonState {
       public ButtonState next() {
          return new OffState();
```

```
    }
    public void activate(StateButton b) {
       b.setText("On");
    }
 }
```

To demonstrate the power of polymorphism in programming we borrow a project developed by Stephen Wong named BallWorld. Although a web search of "BallWorld" pulls up many different projects with this name and with similar visual appearance, his is the only one we've found which stresses good object-oriented design, through the use of design patterns and polymorphic dispatch. His original description can be found at

`www.exciton.cs.rice.edu/cs150/labs/lab1/`
`www.exciton.cs.rice.edu/cs150/labs/lab2/section2.htm`
`www.exciton.cs.rice.edu/cs150/labs/lab3/section2.htm`

BallWorld has proven to be a very rich environment in which to explore the power of polymorphism and a variety of design patterns. BallWorld is a simulation of a set of bouncing balls within a virtual world. BallWorld becomes especially interesting when one starts to vary the behaviors of the balls, combine them, and permit them to change dynamically in response to event-driven user input.

In order to do this students must be able to,

- handle event-driven user input,
- encapsulate behaviors as objects,
- set up polymorphic dispatch, and
- employ relevant design patterns.

Moreover, the student must be able to show the results of this in a graphical manner. The basic design which we want students to aim for is shown in figure 5.

Event-driven user input is handled using the simple mechanism mentioned above involving subclassing an existing class with the event-driven mechanism built in and overriding the magic method. There is nothing special students need to do to get the graphical display since the Ball class we provide does the relevant work for them. Simplifying, our `Ball` class is essentially an `NGP.Graphics.FilledEllipse` which knows how to move. The whole simulation is driven by an `NGP.Timer`, which again follows the general event-handling paradigm laid out earlier: the timer has a dedicated event handler which calls a "magical method" in the `Timer` class. To make a special-purpose `Timer` class requires only that the `NGP.Timer` class be subclassed and the magical method overridden.

Students need to create a subclass (a `BehaviorBall`) which adds to the basic `Ball` class a `Behavior` as a property. The `Behavior` can be a `Null` behavior, in which case the `Behavior Ball` behaves like an ordinary `Ball`. The behavior can also be composite, in which case all `Behaviors` of the composite must act on the `BehaviorBall`.

A pull-down menu of available behaviors allows the user to select an arbitrary number of individual behaviors. The composite of the selected behaviors is the "current behavior". Whenever a `BehaviorBall` is created its initial behavior is the "current behavior".

The `BehaviorBalls` need to be reactive too. When clicked they must change their behavior dynamically to whatever is now the currently selected composite behavior in the menu. The reactivity of the balls is achieved in the now familiar way of simply overriding the appropriate method from the superclass.

3. CONCLUSION

Visual feedback is important to students - they *see* polymorphism working: no matter what specific behavior is associated with a `BehaviorBall` it acts appropriately: the timer calls an `update` method on the `BehaviorBall`, which dispatches to the `BehaviorBall`'s `Behavior`, which in turn acts on the `BehaviorBall`.

Students can *see* composite behaviors at work. They *see* event dispatch (dynamic changing of behavior), and they *know* why things are happening - because they wrote the code! That is to say: they don't know the details of the event handling, but they do know how the change in behavior occurs in response to a particular event. The code they wrote accomplishes the action of the behaviors on the `BehaviorBalls`, or the dynamic changing of `Behavior` acting on a `BehaviorBall`. Students are able to focus on the interesting object-oriented concepts rather than on the particulars of the graphics library or the event-handling mechanism. These things will be important in the second semester course. By the time students take that course they understand enough about object-orientation and design patterns to unwrap NGP and see the workings "under the hood".

What is the point? The point is that through the use of appropriate exercises graphics can be used to accomplish two goals: (i) keep students interested (once they see how the pieces fit together students are *excited* about `BallWorld`), and (ii) reinforce the fundamental principles taught by making the relationship between the graphical components and the underlying program components and relationships as transparent as possible.

4. WHAT OTHER OO EDUCATORS CAN LEARN

Our goal in this paper has been to highlight how the use of graphics in the classroom can serve the dual purpose of keeping the attention and interest of students while simultaneously supporting instruction of fundamental object-oriented concepts.

There is no doubt that today's students feel at home working with and writing graphical event-driven applications. As educators we should leverage this as much as possible in our teaching. Although significant hurdles can exist in using the raw graphics libraries of languages not mainly designed for teaching, these can be overcome. In the specific case of Java we feel the raw AWT/Swing classes require too much pre-existing knowledge of object-orientation and design patterns to be helpful as a tool in *teaching* object-orientation and design patterns to beginning students. We have found the NGP library useful because it wraps raw AWT/Swing classes in a sensible way to make available to students a very usable graphical library at an appropriate level of abstraction. With this we are able to get students started very early on in CS1 writing graphical event-driven programs while learning about fundamental object-oriented concepts and good software design through design patterns.

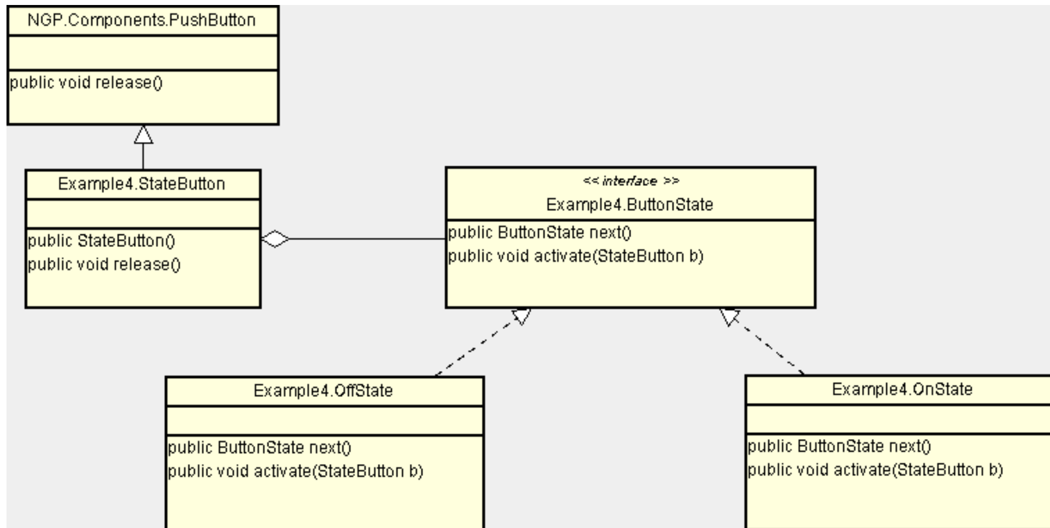

Figure 4: The StateButton UML class diagram

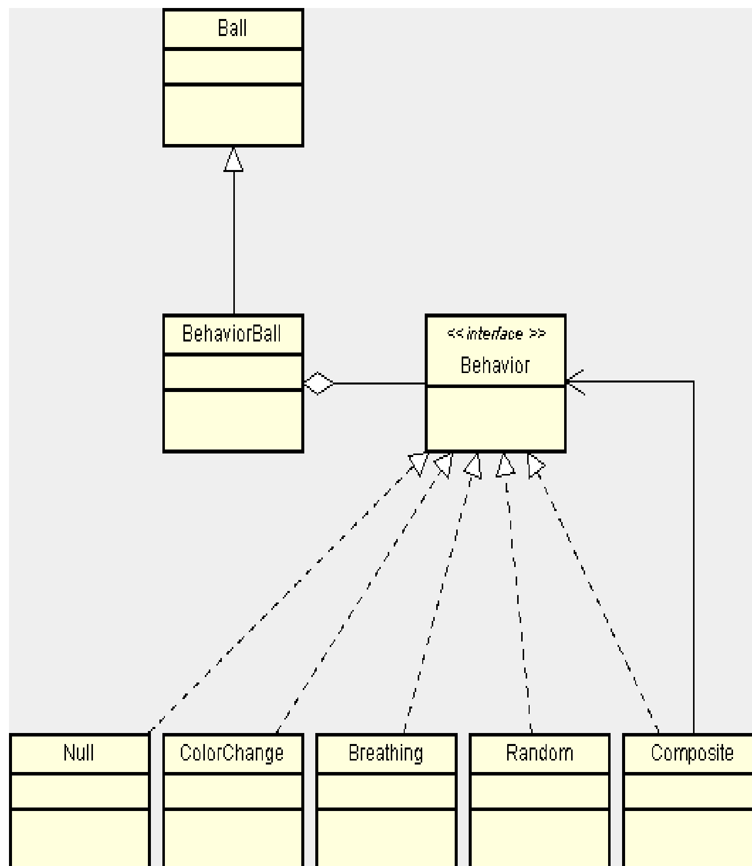

Figure 5: The BallWorld UML class diagram