Objects From the Beginning - With GUIs

Viera K. Proulx
College of Computer Science
Northeastern University
Boston, MA 02115
1-617-373-2225
vkp@ccs.neu.edu

Jeff Raab
College of Computer Science
Northeastern University
Boston, MA 02115
1-617-373-5876
jmr@ccs.neu.edu

Richard Rasala
College of Computer Science
Northeastern University
Boston, MA 02115
1-617-373-2206
rasala@ccs.neu.edu

ABSTRACT
We describe a way to introduce objects at the beginning of the first CS course through the use of objects that have significant nontrivial behavior and interactions with other objects. We will describe four introductory laboratory projects and an outline for introductory lectures on object oriented programming that illustrate the need for private member data, constructors and accessor member functions, and prepare students for writing object oriented programs in Java with graphical user interfaces.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects.* H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *graphical user interfaces.* K.3.2 [**Computers and Education**]: Computer and Information Science Education – *Computer science education.*

General Terms
Your general terms must be any of the following 16 designated terms: Algorithms, Design, Human Factors, Languages.

Keywords
CS1, objects, GUIs.

1. INTRODUCTION
Objects First! is a mantra for the first programming based computer science course. The series of four laboratory assignments presented in this paper introduces students to classes and objects by providing a comprehensive interactive GUI for exploring the member data state and the object behavior in response to different actions, gradually illustrating the most important concepts in object oriented programming. The Java Power Tools [6,7] allow the programmer to build a robust graphical user interface without altering the nature of the actual Java program. As a result, students can see complete working classes from the beginning and explore the behavior of objects through a viewer that itself is a subject of study.

ITiCSE'02, June 24-26, 2002, Aarhus, Denmark.

The paper starts with our view of the skills and concepts that students need to master to begin working with objects and classes. We follow with a framework for class definitions for students to follow. The four labs described in the subsequent sections follow these principles. In the first lab students explore object behavior without any concern for actual code. In the next lab they observe the changes in the member data in response to actions that invoke member function calls and learn the syntax of such calls. In the third lab students read the code for a complete program following a detailed tutorial. All three labs have a creative component where students add some design of their own and get prepared to write their own segments of code. In the fourth lab students extend an existing class by implementing a function that paints a scalable image and use several such images to create a mosaic.

The use of GUIs is an integral part of these labs, both for exploring the behavior of the objects of study and for developing the mental model of the communication between the objects and the views that represent some snapshot of the internal state. The distinction between the internal state and the external view helps motivate a number of issues, such as the distinction between private and public member data and functions, the need for preserving integrity of the object state, and the techniques for assuring such integrity.

2. OTHER 'OBJECTS FIRST' WORK
Though Java and especially the Swing libraries provide extensive support for building GUIs, such tasks are ill suited for students in introductory courses without further assistance. In addition, the support for text input in Java is minimal making even the simplest program difficult. There have been numerous attempts to address these problems that fall into two categories. The first category consists of small toolkits that allow the students to perform simple IO. These include breezyGUI [5], simpleIO and its GUI extension [4], and the input processing functions in [2]. Students are limited to interacting with objects in small snapshots at a time.

Others have built entirely separate environments for student work, such as MiniJava [8], Graphics Library [1], and BlueJ [3]. In [1] students are restricted to working with graphics based objects at first, limiting their view of how objects can be used. In [8] students work with a (pedagogically sound) variation of the Java language, but interact with objects mainly through a console. The BlueJ environment [3] allows the user to construct an object and inspect the behavior and data of that object. Thus, BlueJ acts in some ways as an intelligent debugger. None of these approaches encourages the use of GUIs programmed in Java, the viewing of user-defined subsets of the object state, or the use of algorithm animation. In all

cases, students must eventually "graduate" to dealing with the full mess of Java without any hints about how to use object-oriented principles to encapsulate the complexities of building a full scale program with a graphical user interface.

[3] presents guidelines for introducing objects first using BlueJ, but argues against the use of GUIs. The thrust of the argument there is that GUIs are idiosyncratic and time consuming. This argument loses force if one has a toolkit such as the Java Power Tools that leverages object-oriented principles to dramatically reduce the effort needed to build a GUI. If you can build a high quality GUI in an hour, then time is no longer a big issue .

We believe that well designed GUIs can so enhance the student experience that students will understand why the techniques of object-oriented programming are powerful and worth learning.

3. THE TEACHING STRATEGY

In learning object oriented programming in Java, there are many different concepts, ideas, and skills that students must master almost concurrently. Each of these skills presents a different type of mental challenge. Students often get stuck not knowing what is the real stumbling block in a particular situation.

The key concepts, ideas, and skills that students need to grasp before writing programs on their own are the following:

- create a mental model of what is a class and what is an instance object of this class
- create a mental model of the class organization: the member data and the member functions as well as the information hiding aspects
- understand the organization of Java code: the statements, brackets, parentheses, data declarations, method declarations, etc.
- understand, in the context of Java, what is a type/class, an instance of an object/variable, a value, and a reference
- learn the details of the syntax: member function calls, new object creation, etc.
- understand the logistics of an algorithm that is needed to perform a given action
- understand the encoding of algorithms as member function definitions
- understand the encapsulations that classes provide:
    - assuring integrity of member data
    - hiding implementation details
    - encapsulating complex tasks

The pedagogy should focus on one task at a time if at all possible, giving students the time needed to get familiar with new ways of thinking and organizing ideas not to mention expressing them in a new language.

Once we start the discussion of the organization of a typical class definition, it helps to formalize and explain the role of the different constituents.

A Java class definition typically consists of several key parts. The most important to the novice are the following:

- **constants** used to name default state information and other helpful information
- **member data** that describe the state of an object instance
- **constructors** that are used to create a new instance of an object in this class
- **accessor** member functions (set and get) that allow the user to modify the state of an object instance in a manner consistent with the class constraints
- **action member functions** that perform the tasks for which the class has been designed
- **helper functions** (utility functions) that encapsulate subtasks of accessors and action functions

This may look like a formidable list, but is quite manageable if the first classes and objects presented use each of these constituents in a meaningful way. The first four items are mainly responsible for creating and maintaining the state of an object instance. The action functions are the reason for creating the class. The helper functions encapsulate the low-level tasks and cannot be accessed from the outside of the class. This division of a class into constituent parts helps students to see the distinction between the class API (the part that the caller of the class can see) and the internal implementation of the class.

4. THE FIRST LECTURES

The first lectures should begin with scenarios that explain the ideas of objects and algorithms using real world examples that are familiar to students and easy to understand. An example that may be used is a boom box. It consists of three major objects, the radio, the tape player, and the CD player. These objects share some behavior but also have individual features. Moreover, the sound controls are independent of the music source. No matter what is the source of the music, the same controls are used for volume, speaker balance, treble, and bass. In this example, we can talk about the fact that the user interface (API) is similar for all boom box devices while the internal implementation differs widely from manufacturer to manufacturer. We also know that we do not need to understand the implementation to be able to push the buttons and turn the dials. As a thought exercise, students may be asked to describe the functionality of a boom box device by specifying the member data component objects and the member functions needed for each of these objects.

5. THE FIRST FOUR LAB ASSUGNMENTS

The first labs introduce students to object-oriented programming by starting with the observation of object behavior and gradually increasing the exposure to program syntax and structure.

Each of the four labs uses a GUI to provide an environment for exploring objects and classes in a progressively greater detail. In the first lab, students just observe the behavior of an object-oriented program and learn what method invocation means. In the second lab, students observe changes in the member data of a turtle object in response to a method invocation. Students create small drawings using the GUI to activate the appropriate sequence of turtle methods. In the third lab students study the Java source code, learn how to read text input from a GUI, and learn how to display results. They also modify a small amount of code and learn about the need for safeguarding the integrity of input and member data. In the fourth lab, students extend a class following a given pattern and write code that creates and uses several objects to create interesting graphics images.

5.1 The First Lab: Picture Explorer

In the first lab, students need to get familiar with the particular computer system, the compiler idiosyncrasies, and learn how to extract information from a running program to write a lab report. There is no time yet for thinking about algorithms and program creation. However, this is a great time for introducing the kind of thinking that needs to precede any programming. Specifically, students are presented with an algorithmic problem that they solve by doing computations on paper and then verify by observing a program that uses their results.

In this lab, students work with a complete program with a full graphical user interface that allows them to type in the name of a gif image file and display the image in the graphics window. The GUI allows the user to choose the location of the picture and modify its width and height. The lab document describes the GUI components and their behavior, explains the key components of the **Picture** class, and describes a series of tasks that reinforce the student's understanding of the program behavior.

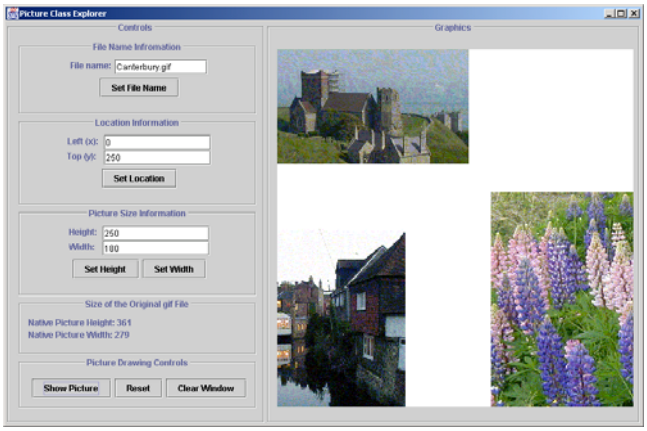

Figure 1. Snapshot of Picture Explorer Application.

Students are asked to show an image in the top left corner of the window, to find out its size by recording the displayed member data information, then display the image in its original size in the bottom right corner of the window, then center it, and finally display it in some desired size. The students perform the needed calculations on paper, record their values into a lab report, and verify their computations by observing the displayed images. They also paste snapshots of their program screens into the lab report.

Although they work with the member data and member functions of a **Picture** class, students are not required to read the code for their lab work. However, once they experience the behavior of a Picture object, this simple class makes a great first example of the code to be presented in the classroom. For example, we may show them how the action button **Set Location** calls the member function **setLocation**, passing two **int** arguments extracted from the two text field views named **xTFV** and **yTFV**:

```
public void setLocation() {
   int left = xTFV.demandInt();
   int top  = yTFV.demandInt();

   myPicture.setLocation(left, top);
}
```

5.2 The Second Lab: Turtle Explorer

In this lab, students again explore the behavior of an object instance in a given class, however, now they observe the current state of member data and see the syntax of the member function calls. The object is a familiar Logo-like **Turtle**, and the member functions mimic the typical Logo functions: **step**, **turn**, **showTurtle**, **hideTurtle**, **setPaint**, **penUp**, **penDown**. Students add again a bit of creative work by making the **Turtle** draw a simple figure of their own design in the graphics window.

The GUI window has a display that shows in the left panel the current state of the member data - displayed in fields that cannot be modified by the user. This reinforces a student's understanding that these data items are hidden from the outside world.

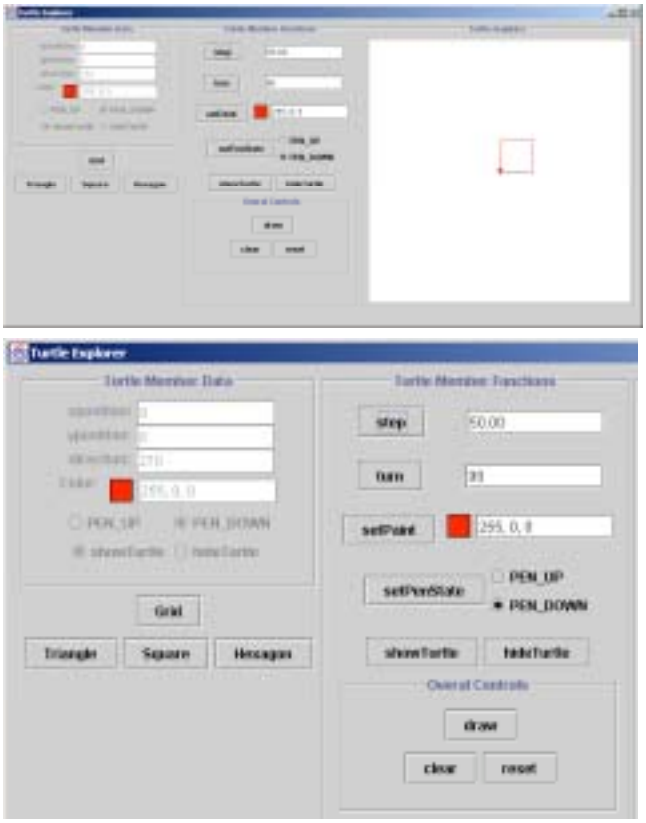

Figure 2. Snapshot of Turtle Explorer Window and a Close-Up of the GUI.

The center panel has an action button for each member function, together with user editable fields that supply the arguments to these functions (as appropriate). If the student modifies the color choice, for example, but does not click the **setPaint** button, the color in the member data display stays at its previous value.

Each time the student activates a member function button, the actual member function call with its correct syntax is displayed in a separate console window and the Turtle performs the desired action in the graphics panel. As an example, we have reproduced the transcript from a simple session that draws a square.

```
myTurtle.step(50.0);
myTurtle.turn(90.0);
```

```
myTurtle.setPaint(255, 0, 0);
myTurtle.step(50.0);
myTurtle.turn(90.0);
myTurtle.step(50.0);
myTurtle.turn(90.0);
myTurtle.step(50.0);
```

The student's task is to explore the **Turtle** behavior, experiment with simple drawings, to write down an algorithm for creating several given drawings, and finally, to make their own design.

To help with creating a drawing, the main application program includes a **draw** button that calls a **draw()** member function. Initially, this function is empty and nothing happens when the button is pressed. To provide a definition of the **draw()** member function, students may copy the transcript of their drawing history from the console window and insert it into the body of the **draw()** function in the application class. After the next compilation, their work will be encapsulated as a function.

In the rest of the lab exercise, students read the code of the **Turtle** class and of the action functions that call the member functions in the **Turtle** class. Thus they become familiar with the syntax, understand the meaning and behavior of member data, the role of accessor functions and constructors. They also practice simple algorithmics without even realizing that they are doing it. And, by now, they are begging for loops, so they do not have to repeat the same drawing code over and over again!!

We should add the fact that the **Turtle** class is actually quite complex and includes many other functions that interact with the appropriate graphics window and even do some automatic scaling. However, students are given a printout only of the relevant parts of the **Turtle** class definition to read in this lab. We do not hide the fact that the class is more complex. This, too, is a lesson to learn: one does not have to understand all of the details to be able to use an appropriate subset of a class's utility. The lab document that describes the lab tasks includes questions about the code, forcing students to read and think about the code organization

5.3 The Third Lab: Ticket Seller

At this point students are ready to add some real statements to an existing program. The Ticket Seller program simulates a box office that is selling tickets for one of three shows with general admission in three categories of ticket prices: adult, student, and child.

The program consists of two classes, the Show class and the main application class. The Show class records the title of the show (movie), the capacity of the theater, the prices of tickets in each category, and the number of tickets sold in each category as requested by the following functions:

available	returns **boolean** with the request given either as the total tickets or as the number of tickets in each category
price	returns the price of an order for a given ticket request
sell	records the requested ticket purchase provided the show is not sold out.

The **TickerSeller** application auto-updates all displays that are affected by a particular action: the show sales statistics, the price list, and the number of tickets available.

There is a large amount of information recorded here. But the problem is one that students understand very well and the algorithmic task of keeping track of sales is very straightforward. Because all of the interaction with the user is through the GUI displays, the action code is not overburdened with extensive I/O statements that deter from understanding of the program logic. Doing a similar program using the sequential I/O dictated by a console interface would be an unmanageable nightmare!!

Figure 3. Snapshot of Ticket Seller Lab Application

The code for creating a Graphical User Interface with the Java Power Tools is very compact and clean. Using the JPT toolkit, each GUI component is created in one statement and the relevant parts are combined into tables using code that is very readable. The main part of the lab document guides students through reading the code (by following the narrative that explains various sections of the code), instructs them to make small modifications, and then asks them to observe the behavior of the program in response to various user actions.

The main program, constructs the GUI, and includes three short action member functions that are activated by the corresponding buttons. One action clears all user input fields. The next two extract the user's request for tickets and either ask for the price of requested tickets or actually perform the purchase (delegating the work to the member functions of the **Show** class).

There are also three actions in the Show selection part of GUI that select the **Show** (object) for which the tickets are sold. The fourth action allows the user to reset the **Show** (object) statistics before selling tickets for the next performance of the show.

What do students learn students learn here? There are three **Show** objects, each with the same behavior but different member data. The need for member data that is not controlled

or even accessible by an outside caller becomes quite clear. The only legitimate way to modify the member data of the **Show** class that records the total number of tickets sold in each category is by making a purchase, that is, by calling the **sell** function. The responsibilities of an object and its class are illustrated in a setting that is conceptually very clear yet rich in object interactions and actions.

Another lesson is about assuring the integrity of the member data. What happens when the user requests a negative number of tickets or tries to construct an instance of the **Show** object that charges `$-5` for student tickets? Students add code to the appropriate constructors and set functions so that negative prices or requests for a negative number of tickets will not be permitted.

Once this code is entered and repeated several times, students are then asked to refactor the code by creating a simple private helper function **adjust(int n)** that returns either the original number, or zero, if **n** is negative.

The code narrative also explains the design of GUI based programs: the definition of text fields, the extraction of data from text fields, the definition of action buttons and their corresponding actions, and the creation and installation of GUI components. With the proper narrative, students are not concerned about the details, yet they see how a GUI element such as a button is created and "connected" to the corresponding function invocation.

The ticket seller exercise can be extended in later courses. A full ticket seller lab (that sells tickets for an entire movie megaplex and maintains many statistics) would be a good exercise for a data structures course or an object oriented design course.

5.4 The Fourth Lab: Scaled Picture

Now that students understand the geometry of the graphics window and the structure of the simple **Picture** class, they will create in this lab a picture object using Java2D Graphics objects: rectangles, ellipses and lines with different colors. The picture object must be able to scale its size and shift its location. We provide an abstract base class **ScaledPicture** similar to the first **Picture** class. It contains the member data needed to record the title, location, and size of the picture and the appropriate constructors and accessor functions. The one abstract function that must be implemented is **showPicture**.

Students first study two examples of concrete classes that implement **ScaledPicture**. They then build a new concrete **ScaledPicture** class according to specifications given in the laboratory exercise. Finally, they are asked to design their own concrete **ScaledPicture** class entirely as they wish.

The GUI provides feedback by displaying the various pictures in different shapes and sizes. To see how the objects are created and used, students must also implement a **Mosaic** action in the main application that will create a collage of at least three pictures.

As a variation, we also include a **ScaledImage** class that extends the **ScaledPicture** class but displays any given gif image.

6. LESSONS LEARNED

The examples presented here break the mold of traditional "Hello World" exercises, that use simple I/O with perhaps an assignment statement: the "Here is some very short working code" approach. To be truly Objects First, one has to use objects that matter in a situation where they are useful. This is not possible as long as we are constrained by the sequential I/O model of console-based user interactions. We believe that it is not possible to do Objects First without also doing GUI First!

We are aware that people who use BlueJ achieve some of these goals by effectively using BlueJ to provide the interactive interface that is not being provided by the Java program itself. We dislike this approach because every interaction is forced into a standard mold of "create an object, execute a method on the object, and then inspect the object to see the results". We strongly prefer to provide students with a quality graphical user interface that is designed specifically for each laboratory exercise.

We break another mold too in introducing objects first. The first four labs contain more code reading than most textbooks have in an entire first course. Students understand simple algorithms. They know how to compute the price of the ticket order. Why is it so hard to write a program that performs the calculations? The answer is that from the student perspective, they have to write it in Swahili using the Cyrillic alphabet. The key issue is to supply code that is interesting, well written, and conceptually simple, so that students in reading it can comprehend the role of different statements even if they may not know all the details of language or even the precise syntax. Gradually by reading code, then modifying code, then creating new code, students will learn the language of programming in much the same way that every child learns its native language, by in-depth exposure and repetition. It is only after knowing what the language is and how to use it that students will be enabled to study its nuances in detail.

6.1 Online Materials

The Java Power Tools, labs, tutorials and sample files URL is:

http://www.ccs.neu.edu/jpt

This work was partially supported by NSF grant DUE-9950829

7. REFERENCES

[1] Bruce, K. B., Danyluk, A., and Murtagh, T. P., *A Library To Support a Graphics-Based Object-First Approach to CS1*, SIGCSE Bulletin, 33(1), 2001, 6-10.

[2] Horstman, C., and Cornell, G., Core Java 1.2, SunSoft Pres, Mountain View, CA, 1999.

[3] Koelling, M., and Rosenberg, J., *Guidelines for Teaching Object Orientation with Java*, SIGCSE Bulletin, 33(3), 2001, 33-36.

[4] Koffman, E., and Wolz, U., A Simple Java Package for GUI-like Interactivity, SIGCSE Bulletin, 33(1), 2001, 11 - 15.

[5] Lambert, K. A., and Osborne, M., JAVA Complete Course in Programming & Problem Solving, South-Western Educational Publishing, Cincinnati, OH, 2000.

[6] Raab, J., Rasala, R., and Proulx, V. K., *Pedagogical Power Tools for Teaching Java*, SIGCSE Bulletin, 32(3), 2000, 156-159.

[7] Rasala, R., Raab, J., and Proulx, V. K., Java Power Tools: *Model Software for Teaching Object-Oriented Design*, SIGCSE Bulletin, 33(1), 2001, 297-301.

[8] Roberts, E., *An Overview of MiniJava*, SIGCSE Bulletin, 33(1), 2001, 1-5.