Teaching Object-Oriented Concepts Through GUI Programming

Jesse M. Heines and Martin J. Schedlbauer
Dept. of Computer Science
University of Massachusetts Lowell
One University Ave., Lowell, MA 01854
1-978-934-3634

[heines, mschedl]@cs.uml.edu

ABSTRACT

It is difficult to teach object-oriented programming (OOP) from a language perspective, even to experienced programmers. Complex syntaxes obscure concepts and make it difficult for learners to get a real "feel" for OO architecture. This is a classic example of "not seeing the forest for the trees." OOP is best taught within a context of an application or software framework. Graphical user interface (GUI) programming provides a particularly effective vehicle for this purpose because it is relevant to virtually all applications and provides immediate feedback on the correctness of OO structures through tangible, visual results. We have built a GUI Programming course that focuses on the OO aspects of building user and application programmer interfaces (APIs). This paper presents an overview of our approach and some of the techniques we use in that course.

1. OOP AND STUDENT EXPERIENCE

Object-oriented programming (OOP) has now been taught at the undergraduate level for about two decades [1, 7]. Our experience, however, is that while students who complete these courses can "walk the walk and talk the talk," they have trouble applying OO concepts in project-based courses where they must solve problems outside the domain in which they were introduced to OOP. If they have not fully internalized the concepts, students will not see the connections between the techniques they learned and the problems they encounter in the project course's application domain.

We propose that one reason for this disconnect is that students lack experience with large programs. Without such experience, students don't see the real benefit of OOP and therefore don't internalize the concepts. Consider, for example, the often-cited pillars of OOP [2, 5] and their relationships to student programs:

- *Extendibility* – Homework assignments generally consist of small programs that may extend a class or two, but they seldom fit together into a larger, hierarchical class structure.
- *Scalability* – Student programs are seldom longer than a few hundred or at most a thousand lines. Thus, they don't see the benefits of the OOP structure that become readily apparent in programs several thousand lines long.
- *Maintainability* – Students typically work alone and seldom look at others' code except to copy it (a large problem in this day of Google and the Web). The only code they may study in detail is that of the instructor, and even then they often fail to notice subtleties of design and therefore fail to emulate them. Foremost in this regard is their failure to document code, regardless of how carefully and completely the instructor may document his or hers.
- *Reusability* – Students generally consider the programs they develop to be "throw aways." Seldom, if ever, do they reuse their own code in an OOP fashion (cutting-and-pasting code from one program to another doesn't count), and even more rarely do they reuse classmates' code or provide code for their classmates to use (again, unauthorized copying doesn't count).

2. OOP AND GUI PROGRAMMING

To teach OOP effectively and give students firsthand experience with its benefits, one needs a substantial framework. We have found that GUI programming provides a particularly good teaching framework. It is sufficiently complex to challenge even our most advanced students, yet its visual nature and tangible results put OOP within the reach of those students who have more trouble grasping the concepts.

Four other factors support our decision to offer a two-semester GUI Programming course sequence with heavy emphasis on objects.

- *The availability of excellent examples* – The large Java Swing (JFC) and .NET (and its predecessor MFC) class hierarchies required to do GUI programming are themselves excellent examples of large scale, well designed OO frameworks. Significant learning can take place merely by browsing and poking around these hierarchies.
- *The availability of excellent tools* – NetBeans, Eclipse, Visual Studio .NET, and other less well known IDEs are mature tools that all contain form designers and code generators. The availability of these tools not only allows students to focus on semantic relationships rather than syntactic details, but also cuts down the number of lines of code that students need to write themselves and reduces the number of errors in student developed applications.
- *The availability of (mostly) excellent documentation* – One can't do GUI programming without knowing how to use and navigate the API documentation, as there are simply too many classes to know and too many methods to choose from. Learning to navigate the API documentation is therefore a critical skill for today's students. More importantly, learning to

write quality API documentation is an even more critical skill. The extensive documentation on the literally hundreds of GUI-related classes and thousands of GUI-related methods provides not only an excellent tool, but also an excellent guideline for the level of documentation needed for a non-trivial application. As students work with the existing documentation they are quick to identify where that documentation falls short, and professors can easily turn such discoveries into lessons on the value of quality API documentation.

- *The inclusion of essential design patterns* – GUI programming also makes use of many of the design patterns identified by the "Gang of Four" [3, 4, 6]. Horstmann [4] dedicates an entire chapter in his book *Object-Oriented Design & Patterns* to the "patterns that arise in the Swing user interface toolkit and the Java collections library." Such patterns include observers (the MVC architecture and listeners), strategies (layout managers), composites (UI components and containers), and decorators (scroll panes and borders). To these one might add singletons (calendars), factories (borders), and commands (menus).

3. LECTURE TOPIC EXAMPLES

3.1 "Building Bridges" Between Classes

One of the major concepts that students struggle with in GUI programming is the relationship between objects that represent on-screen components (views) and objects that represent the data behind those objects (not only models, but also the data structures used to populate the models, either during instantiation or afterward). There are many cases in which one needs to "build a bridge" between various classes or methods or objects.

Students must, of course, understand the interaction between a GUI component and its corresponding model. For example, consider the steps necessary to set the text of the root node of a `JTree` component. If you're doing straight Java coding, you can do this when you call the overloaded `JTree` constructor that takes a `DefaultMutableTreeNode` as an argument:

```
DefaultMutableTreeNode root =
   new DefaultMutableTreeNode(
      "91.461 Fall 2005" ) ;
JTree jtreeCourse =
   new javax.swing.JTree( root ) ;
```

When working with NetBeans, however, the code generator always calls the default `JTree` constructor:

```
jtreeCourse = new javax.swing.Jtree();
```

The root node is set at this point, and there is no method in the `JTree` class to set it to something else. Thus, asking students to create a `JTree` with the root node text set to a specified value often throws them. If they search carefully, they eventually find the `setRoot` method in the `DefaultTreeModel` class. But how do they get a reference to an object of that class and how do they use it to set the text of the `JTree` root node?

The answer is easy if one understands OOP accessor methods, casting from a parent to a child, and the fact that changing the model automatically changes the rendition in the view. Taking this in steps for clarity, one first gets a reference to the `Tree-Model` for the `JTree`:

```
TreeModel tmCourse = jtreeCourse.getModel() ;
```

Next, one casts that `TreeModel` to a `DefaultTreeModel`:

```
DefaultTreeModel dtmCourse =
   (DefaultTreeModel) tmCourse ;
```

One can then set the root of the tree to the `DefaultMutable-TreeNode root` constructed earlier:

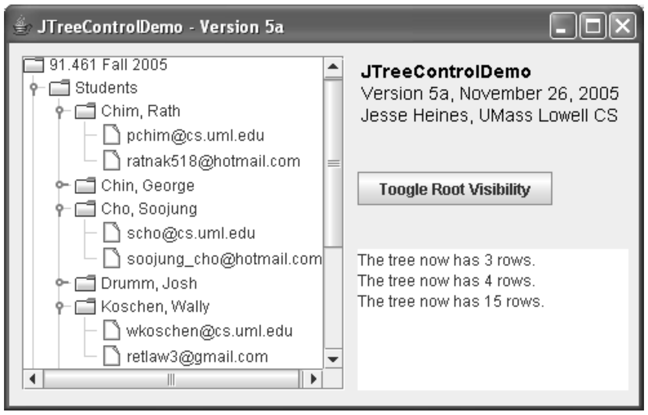
```
dtmCourse.setRoot( root ) ;
```

However, even though that's just three lines of code, they encompass a lot of concepts to understand! Such examples provide a rich platform for teaching the OOP principles involved, and students can easily *see* if they "build the bridge" successfully because the text of the `JTree` root node will change to the text they specified if they do it correctly. Note that discussion of this topic also gets students into using the API, a critical skill (as discussed earlier) in today's programming environments.

3.2 Using Auxiliary Classes

Another concept that students struggle with is the use of objects they create themselves. Even though they quickly grasp the syntax of how to create a class and then instantiate an object of that class, students often don't see the need to do this in a "real" application. Since those "real" applications are typically small, they find it easier to just use regular variables (or at most arrays) in their main classes rather than to go to the trouble of creating a class to store their data.

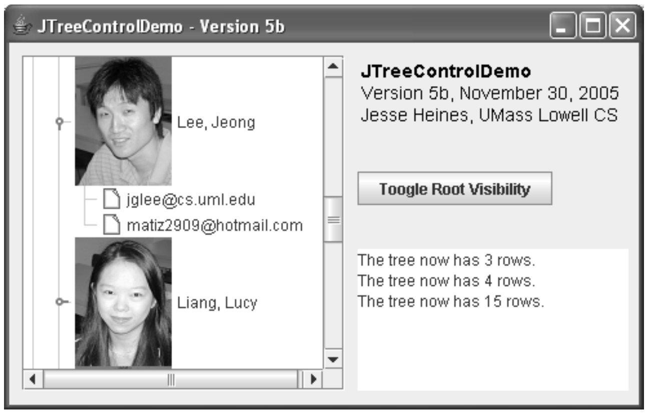
Sticking with `JTree`, here's an example of how auxiliary classes might be introduced. Consider the tree shown in Figure 1. It is pretty straightforward to create a 2D array of the students' names and e-mail addresses and create the `DefaultMutableTreeNode` objects needed to populate this `JTree`. One could, of course, create a `Student` class and use that to populate the tree, but doing so seems contrived in such a small example.



```
The tree now has 3 rows.
The tree now has 4 rows.
The tree now has 15 rows.
```

Fig. 1. Sample `JTree` control populated with students' names and e-mail addresses.

The auxiliary class is not contrived when students are shown how to create a `JTree` like that shown in Figure 2. Here one could use code identical to that in the previous example and manipulate the tree nodes after they have been created, but it is much more elegant to create one's own class that extends `DefaultMutable-TreeNode` to encapsulate the tree node data and then use a custom cell renderer (a class that extends `DefaultTreeCell-Renderer`) to display the node. The interaction of the single GUI tree control object (`JTree`), the multiple tree node objects


JTreeControlDemo - Version 5b

JTreeControlDemo
Version 5b, November 30, 2005
Jesse Heines, UMass Lowell CS

The tree now has 3 rows.
The tree now has 4 rows.
The tree now has 15 rows.

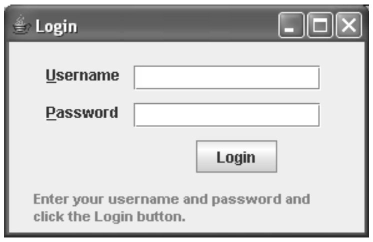
Fig. 2. Sample `JTree` control in which students' pictures have been added to their respective nodes.

(derived from `DefaultMutableTreeNode`) that contain the tree's data, and the tree node renderer (derived from `Default-TreeCellRenderer`) that displays the tree nodes as desired provides a rich example of the power of OOP that students can readily see and appreciate.

4. ASSIGNMENT EXAMPLES

4.1 Creating Custom Dialog Boxes

Assignments are of course intended to reinforce topics discussed in class. After doing some work with simple GUI controls such as text boxes and radio buttons and check boxes, the first real challenge that students face is to use these controls in a custom dialog box (`JFrame` or `JDialog`) and somehow get the results of the user's choices in that dialog box back into data structures defined in the main class. Even a simple login dialog box like that shown in Figure 3 makes a nice first example.



Enter your username and password and click the Login button.

Fig. 3. A simple login dialog box, but the OOP challenge is to pass the username and password entered here back to the main class so that they can be validated.

There are several ways to do this, of course, but the method we teach is to pass a reference to the instance of the calling class in the parameter list of an overloaded constructor for the custom dialog box, and then to use that reference to call public accessor methods in the calling class that get and set its variables and/or manipulate its objects. This method stresses the implications of private data and public methods in two classes that have neither inheritance nor encapsulation relationships.

These are clearly very basic OOP concepts, but few students really grasp their significance. As mentioned earlier, they can "walk the walk," but they still have trouble applying basic concepts in real applications. The beauty of doing so in a GUI context is that they get immediate feedback on whether the data was exchanged when they try to use it back in the main class.

4.2 Formatting Text

We have often felt that one could teach the entire GUI programming course using a single common, yet surprisingly complex example: word processing. We never did this because we feared that students would get bored spending an entire semester adding features to a single application. The opportunity to use such a familiar application to illustrate OOP and GUI programming principles is, however, simply irresistible.

Building on the concepts learned in creating custom dialog boxes, one popular assignment we have used asks students to apply those concepts to formatting text in a `JEditorPane`. All sorts of custom dialog boxes are possible in this application, and it is interesting to discuss which work best as modal dialog boxes and which work best as modeless, as well as the many human factors considerations involved in grouping various text attributes into a single dialog box.

One of the very first issues – choosing the font to use – provides an excellent example of the singleton pattern, because one must have an instance of the `GraphicsEnvironment` class to call the `getAvailableFontFamilyNames` method. The constructor for the `GraphicsEnvironment` class is protected, so it cannot be called directly. There is only one graphics environment anyway, thus a singleton is appropriate.

```
GraphicsEnvironment ge =
    GraphicsEnvironment.
        getLocalGraphicsEnvironment() ;
String[] strFontNames =
    ge.getAvailableFontFamilyNames() ;
```

One could even go so far as building a font chooser dialog box, but most students have their hands full just populating a `JComboBox` with all the font names and then applying the chosen font to the selected text. Figure 4 shows the formatting dialog box submitted by one student for this assignment.

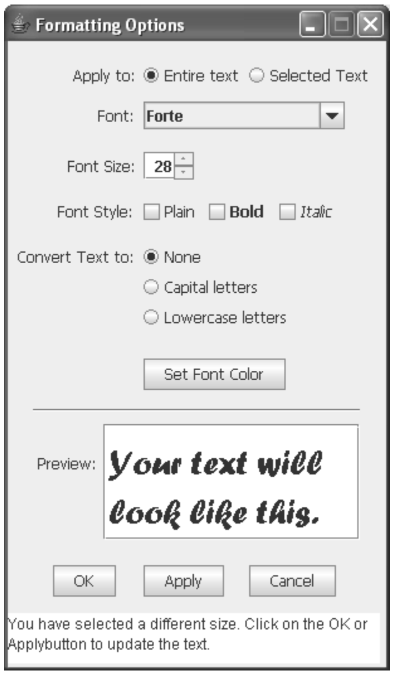
4.3 Additional Examples

All class notes, examples, and assignments for our course may be found online by going to http://teaching.cs.uml.edu/~heines, clicking the Teaching button at the top of the page, and then clicking the links for the courses named GUI Programming I and II. (The course numbers have changed over the years.)

Our GUI programming course sequence is also a senior capstone project course, so much of the second semester (GUI Programming II) is spent on software engineering and human factors issues. At the time of this writing (the Spring 2007 semester), we are teaching a compressed version of the sequence, so classes devoted to project plan reviews and student presentations may also be found in GUI Programming I.

5. EXAM QUESTION EXAMPLES

Our exams mirror the course's OOP emphasis, focusing on concepts rather than the details of specific controls or the mechanics of putting controls on the screen. We typically give open book exams and allow students to use their own laptops or systems that we provide to access the Java API during exams. Figure 5 presents a set of questions typical of those on our exams.


Formatting Options

Apply to: [✓ Entire text] Selected Text

Font: Forte

Font Size: 28

Font Style: ☐ Plain ☐ Bold ☐ Italic

Convert Text to: [✓ None] Capital letters Lowercase letters

Set Font Color

Preview: Your text will look like this.

OK Apply Cancel

You have selected a different size. Click on the OK or Applybutton to update the text.

Fig. 4. A student-designed dialog box for specifying text formatting options.

6. TEACHING GUI PROGRAMMING WITH WEB-BASED VS. STANDALONE TECHNOLOGIES

This discussion would not be complete without addressing the question of teaching GUI programming using HTML and its related technologies (scripting languages, cascading style sheets, etc.) vs. Java (or a similar language such as C#).

Many students understandably want to learn web-based technologies. The first problem with those technologies is that the GUIs one can build in them are far less rich than those in standalone environments, unless one uses proprietary products such as Flash. But more importantly, scripting languages, though somewhat object-oriented, tend to implement OOP in non-standard ways and with clearly "shoehorned" syntax. (Consider, for example, how constructors are implemented in JavaScript.)

Our philosophy in using Java (or some other highly object-oriented standalone language) is that if students learn the concepts that underlie GUI programming, they can apply them in any language, even those that have not yet been invented.

7. REFERENCES CITED

[1] Beck, K., & Cunningham, W. (1989). *A laboratory for teaching object oriented thinking*. Proceedings of a Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA-89), pp. 1-6. New Orleans, LA.

[2] Chu, W.C., Lu, C.-W., Shiu, C.-P., & He, X. (2000). *Pattern-based software reengineering: a case study*. Journal of Software Maintenance: Research and Practice **12**(2):121-141.

[3] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software* (2nd ed). Upper Saddle River, NJ: Addison-Wesley Professional Computing Series.

[4] Horstmann, C. (2004). *Object-Oriented Design & Patterns*. Hoboken, NJ: John Wiley & Sons.

[5] Mari, M., & Eila, N. (2003). *The impact of maintainability on component-based software systems*. Proceedings of the 29th Euromicro Conference, pp. 25-32.

[6] McConnell, S. (2004). *Code Complete* (2nd ed). Redmond, WA: Microsoft Press.

[7] Sims-Knight, J.E., & Upchurch, R.L. (1992). *Teaching object-oriented design to nonprogrammers*. Proceedings of OOPSLA-92 Educators' Symposium. Vancouver, British Columbia, Canada.

The following questions pertain to controlling what happens when the user presses the `Tab` key, which is known in Java as the *focus traversal policy*.

1. Why is it important to worry about what happens when the user presses the `Tab` key?

   Because an application in which pressing the `Tab` key causes focus to jump all over the place is confusing to the user and looks unprofessional.

2. One way to set focus to a component is call the `grabFocus()` method. What class is `grabFocus()` a member of? (Use the Java API to answer this and other questions in this section.)

   JComponent

3. The documentation says that "client code" should not use method `grabFocus()`. What method does the API recommend using instead of `grabFocus()`?

   requestFocusInWindow

4. A better way to implement your own focus traversal policy is to create a class that extends the built-in `FocusTraversalPolicy` class. Why must this be a separate class from your application's main class, which typically extends `JFrame`?

   Because a class can only extend one other class.

5. We have seen three ways to create an instance of a class that you might use for your own focus traversal policy. Name two that you might use to create a `FocusTraversalPolicy` class for use in your application.

   Create an anonymous inner class.
   Create an named inner class.
   Create a separate external class.

6. Look up `FocusTraversalPolicy` in the Java API and scroll down to the listing of its methods. You will see that five of the six methods are abstract. What implication does this have for the class that you create that is derived from the built-in `FocusTraversalPolicy` class?

   You must implement each of the abstract methods or you will not be able to instantiate your class.

7. Once you have defined a focus traversal policy, you must associate it with your `JFrame`. What method performs this function, and what class is that method a member of?

   setFocusTraversalPolicy in class java.awt.Container

Fig. 5. Sample test questions.