How a Language-based GUI Generator Can Influence the Teaching of Object-Oriented Programming

Prasun Dewan
University of North Carolina
Department of Computer Science
Chapel Hill, NC 27599
001-919-962-1823
dewan@unc.edu

ABSTRACT
A language-based direct-manipulation user-interface generator automatically creates a visualization of an object directly from its class, allows users to edit the visualization, and invokes methods in the object in response to these edits. Such a generator can change, and we argue, improve the lectures and assignments on programming conventions, methods, state, constructors, preconditions, MVC, polymorphism, graphics, structured objects, loops, concurrency, and annotations. We have built such a generator, which has several novel features for teaching such as interactive instantiation of a class, interactive invocation of methods and constructors that take arbitrary arguments, visualization of objects representing records, sequences, table and graphics, use of preconditions to disable/enable user-interface elements, and automatic generation of model threads. We have been working on and using such a generator for teaching CS 1 and CS 2 for about a decade.

Categories and Subject Descriptors

General Terms
Documentation, Design, Human Factors, Standardization

Keywords
MVC, User-interface generator, preconditions, concurrency

1. INTRODUCTION
Instructors of introductory programming are faced today with two main choices: use an interpretive functional language such as Scheme or ML; or use a compiled object-based programming language such as C# or Java. Though decreasing in popularity, the first choice continues to be a valid alternative because it allows students to write functions, and hence focus on programming logic, from day one. Once they have written a function, they can see what it does by simply calling it interactively – the interpreter performs the messy tasks of mapping user-input to function calls and displaying return values of these calls, freeing them from learning about I/O altogether. Object-oriented programming requires the teaching of either console I/O or (user-interface) toolkits. Usually the latter variation is chosen as it allows the creation of graphical user-interfaces (GUIs). In comparison to the functional-language-with-interpreter approach, the OO-programming-with-toolkit approach is the more popular choice because it teaches the powerful and practical concepts of object-based programming and allows generation of GUIs.

The BlueJ [1] and Dr. Java systems have shown a path to overcome this problem by allowing methods of objects to be invoked interactively. However, these systems have two related problems. First, they come with their own programming environments. Second, they support command-based interaction, wherein a user manipulates an object by entering a command and parameters, using the command line and/or menus/dialogue boxes. A more modern alternative is to support direct-manipulation user-interfaces [2], where a user interacts with an object by editing a visual representation of it. Such interfaces are not only more compelling to the students, but also can intuitively display complex concepts. These two problems are related as programming environments mimic the interaction style offered by programming languages.

User-interface generation [3] seems to offer a solution to this problem. It is based on the insight that certain types of conventional program values can be automatically visualized by certain kinds of user-interface objects. For instance, any primitive type can be displayed as a string, an int as a slider, an enum as a combo-box or spinner, a record as a form, and an array, sequence, and map as a list, table, or tree. It is possible to edit these visualizations to change the underlying values.

However user-interface generation is apparently at odds with object encapsulation – it requires knowledge of the structure of the visualized object, while encapsulation hides these details. Two workarounds have been tried to overcome this problem. One is to require programmers to manually create an explicit description of the object structure in a non object-oriented language (e.g. [4]). This description must be manually mapped to the implementation of the object, which is an error prone and tedious task. Moreover, it is hard enough to teach programming to students in one language – one cannot expect them to use and learn two very different languages simultaneously. The second workaround is to break encapsulation and require programmers to declare object components as public variables [5], which essentially asks programmers to do conventional programming in an object-oriented language.

What is required, thus, is a generator that overcomes the apparent difficulty above to create an editable visualization of encapsulated objects directly from the descriptions of these objects in a
standard programming language. We refer to such a generator as a language-based direct-manipulation GUI generator. In this paper, we describe how we have adapted and applied the author’s research on such a generator for Java, to the teaching of programming in CS 1 and CS 2 classes. The difference between the two classes in our university is that students in CS 2 are required to have seen some programming before, which may not have been object-oriented. Thus, students in both classes are introduced to object-oriented programming. Freshmen take both classes, and usually the majors take CS 2.

2. STATELESS OBJECTS

When ObjectEditor is used, the first program students see is not the traditional “hello world” printer. Instead it is a stateless class that implements a mathematical function (Figure 1). Students in CS1 see a class without a package declaration, while those in CS2 see the packaged version shown here. Without ObjectEditor, students would have to use a main method to exercise the function. But this approach has the well-known problem that it uses the concepts of arrays of static methods, arrays, and string – none of which CS 1 students know on day one. Thus, they have to be presented the main method as a magic recipe.

Figure 2 shows how the functional language interpreter approach has been adapted in ObjectEditor. Among other items, the tool provides the “new(String)” menu item (a), which can be selected to create a form allowing the entry of the name of a class to be instantiated (b). The result of pressing the button on the form is creation and visualization of a new instance of the entered class (c). The class provides the menu, ABMICalculator, containing the single item, “Calculate BMI (double, double)”, which can be selected to create a form to enter the parameters of the corresponding method. Pressing the button invokes the method, and displays a visualization of the Double result (e). This visualization allows interactive invocation of the Double methods on the result (e). The student can use this facility to interactively explore the functionality of a Double. As we see above, Fig 2(a) and (b) are very similar to 2(c) and (d), respectively. The reason is that in Figure 2(a), ObjectEditor is used to visualize and edit itself. In general, the visualization of an object of class C creates a menu named C with items for invoking the methods of the class (Figure 2(a, c, e)). Selecting a menu item creates a form to enter the parameters of the corresponding method, and creates a visualization of the result of the method (Figure 2(c, e)).

Interactive class instantiation and object editing allow CS 1 students to write and test a useful function on the first day. In addition, they allow method invocation to be learnt at an intuitive level – assigning actual parameters to formal parameters defined by a method declaration is essentially filling items in a form to request a service that defines the structure of the form and reads the filled values. This analogy is made concrete by ObjectEditor. Furthermore, the names of method menu items such as “new(String)” and “Calculate BMI (double, double)” make the abstract idea of a method signature concrete to both CS 1 and 2 students. Students see that a signature is useful to distinguish between different methods. Finally, both CS 1 and 2 students also use ObjectEditor to test the functionality of an object without multiple iterations of the run-modify program steps. To try out different values of the parameters of a method, they do not have to write modify a main class that calls the method. They can simply call the method repeatedly with different parameters.

CS 2 students do not use the ObjectEditor for creating and displaying an object. Instead, they do so directly from a main method, as shown in Figure 3.

Once students learn how to define and invoke functions, they can be taught functional programming to the depth desired by the instructor. At some point, however, state must be introduced.
3. BEANS AND CONSTRUCTORS
To motivate state, we take a stateless object, such as the example above, and point out the effort required to specify all actual parameters of a method in each invocation of it, even if some of these invocations share some of these values. For example, the height parameter must be specified in each of the two invocations of the calculateBMI method shown in Figure 4, even though it does not change.

Figure 4 The problem of re-entering height
We then take a stateful version of some stateless object to show how values can be remembered. The code and visualization of the stateful version of the BMI calculator class is shown in Figure 5. The figure shows how the seemingly impossible problem of breaking up an encapsulated object can be addressed. The generation process, which students understand intuitively, is formally explained in terms of Java Bean conventions for defining getters and setters of typed properties. They are able to derive on their own that getters are invoked to display property values and setters to reflect user edits to them. They are able to understand that programming conventions make the code understandable to both humans and tools such as ObjectEditor.

Figure 5 Displaying Bean Properties
As we see in Figure 5, a stateful object can have constructors, which can also be invoked interactively. If a class has more than one constructor, then when it is instantiated using ObjectEditor (Figure 2(a)), a menu of all of its constructors is displayed to the user (Figure 6, top window). Choosing one of these results in a form similar to the one created to invoke any method (Figure 6, bottom window). Invocation of the constructor results in the newly created instance to be shown. As we see here, ObjectEditor concretely visualizes the abstract idea of multiple constructors and their use in initializing objects.

Figure 6 Interactive Constructor Invocation
ObjectEditor also makes the distinction between stateless and stateful objects very clear. Different stateless instances of a class all look the same, and have no main window to display the components of the instance, while different stateful instances of a class have main windows that can diverge based on the constructor used and methods called on them. Once students see state, they are shown that executing the Show Class Name command in ObjectEditor (Figure 2(a)), allows a property to be displayed that remembers the last class they entered. As we see later, students use preconditions to themselves dynamically display properties or enable menu items.

4. DYNAMIC TYPES & POLYMORPHISM
Not all stateful objects use Bean conventions. Essentially, Bean conventions allow decomposition of an encapsulated object that implements a conventional record type. Students also create and/or use objects that represent arrays, (variable-length) sequences, sets, and (hash) tables – four other important data types taught in CS 1 and 2 classes. Arrays are language-defined, so they pose no problem – the predefined constructs for setting and getting their elements can be used to create editable visualizations of them. One approach to handling sequences, sets, and tables is to require these to be instances of predefined interfaces such as the Java Set, List and Map interfaces, respectively. However, this approach has two related problems. First, it does not give students experience in implementing their own interfaces for these types. Second, predefined interfaces may contain too few or many methods for a particular application. In both classes in our institution, they make extensive use of history objects, which are sequences that allow addition but not replacement or insertion of elements. The Java List interface provides the union of all methods that may be used in sequences, and thus cannot support various kinds of constrained sequences. Therefore, as for records, we rely on conventions for defining dynamic objects. We have taken two steps to address the fact that no standard Bean-like conventions have been defined for these and other types discussed below. (a) We allow conventions to be defined by the instructor – the exact details are beyond the scope of his paper. (b) We have defined conventions based on popular Java types. For example, we recognize an object as a variable-length readonly sequence of type <ElementType> if it defines the following two methods:

```java
type int size();
<ElementType> get(int index)
```
Here the words in angle brackets are placeholders. If a class also defines the following optional method:
then the elements of the sequence can be edited by the user. We define several other conventions for sequences and other dynamic types, which are details we omit because of space constraints.

Figure 7 shows the use of the convention above to visualize and edit an instance of a Java vector. It also brings out another use of ObjectEditor—visualization of the abstract concept of polymorphism. Figure 7(a) visualizes a new, and hence empty, instance of a Vector. Suppose we call add(Object) to add a new element to this instance (Figure 7(b)). As we saw when we invoked the calculateBMI() method, a dialogue box is displayed to enter the actual parameter of the method. However, there is an important difference—the type of the parameter is an object type rather than a primitive type. Unlike a primitive type, an object type is a polymorphic type in that it can stand for a set of types, which includes all types that have an IS-A relationship with it.

Therefore, before entering the parameter value, the user must first specify the class of the parameter. ObjectEditor allows the user to type the exact class name. It also provides a drop-down menu listing commonly-used predefined class names that have an IS-A relationship with the declared type and those the user has explicitly entered in previous invocations of the method. In this example, the user selects the String class as the parameter type (Figure 7(b)) and then enters the actual string to be used as the parameter (Figure 7(c)). The new string is then displayed as a dynamic element of the Vector (Figure 7(d)).

One of the first questions students ask when they encounter ObjectEditor is “why is the type field a combo-box”? They are told to wait until they are taught polymorphism, and once this abstract anticipated concept is covered, they can visualize it as a combo-box representing a type choice.

In this example, the value of the chosen type is atomic in that it can be entered by directly editing the parameter text field. To be complete, ObjectEditor should also be able to handle arbitrary parameter types, which could include structured types such as ABMISpreadsheet or Vector. Figure 8 shows how a structured value is entered. In this example, we use an extended version of ABMISpreadsheet of Figure 5, which has two additional properties, BMIClassification (enum) and OverWeight (bool), respectively, and an additional method, RestoreWeightAndHeight, which resets the height and weight to their initial values.

Let us call the Vector add(Object) method again, but this time, let us enter ABMISpreadsheet as the class of its parameter (Figure 8(a)). As this type is structured, it has no standard parsable String representation. ObjectEditor instantiates such a parameter class and recursively creates an editor for the new instance (Figure 8(b)). We can now use this editor to manipulate the object (Figure 8(c)), and execute the File → Done command to commit the parameter value (not shown here). At this point, the edited object is assigned as the actual parameter of the dialogue box, with its string representation (that is, the return value of the toString() method defined on all objects) being shown as the parameter value (not shown here). The user can then press the button in the parameter dialogue box to invoke the add(Object) method. The visualization of Vector changes to show the new item (Figure 8(d)). This, such interaction shows the relationship between structured parameter types in the code and cascaded dialogue boxes in the user interface.

5. PRECONDITIONS

Yet another important abstract concept visualized by ObjectEditor is the concept of preconditions. Consider an arbitrary target method whose signature is decomposed as follows:

<AnyType> <M>(<T1> p1, <T2> p2, …, <Tn> pn)

A method with the signature:

public boolean pre<M>(<T1> p1, <T2> p2, …, <Tn> pn)

is considered its precondition method. Thus, the following method:

public boolean preGetBMI() { return height >= 0 && weight >= 0; }

is the precondition method of the target method:

public double getBMI() {
    assert preGetBMI();
    return weight / (height*height);
}

If a target method is a getter for some property P, then ObjectEditor does not display P until its precondition method returns false. If a target method is not a getter or a setter, then it is shown in the method menu ObjectEditor creates for the class of the object. The menu item for the method, however, is not enabled until the corresponding precondition method returns true. It is because of these two rules that the BMI, BMIClassification, and OverWeight properties are not shown when the height or weight are 0 (Figure 8(b)), and the RestoreWeightAndHeight menu item is disabled if the height and weight have their initial values (Figure 8(b)). The impact of preconditions on disabling and enabling of widgets have encouraged our CS 1 and 2 students to use assertions in the manner shown above. Before ObjectEditor, students have disliked putting assertions in code, mainly because they did not see concrete benefit of doing so.

6. GRAPHICS

It is attractive to teach students about (2-D) graphics objects such as Point, Line, and Rectangle for several reasons. First they are familiar with some of these concepts from earlier geometry courses. Second they form excellent examples to show that there
might be multiple useful representation of the same real-world object – for example, a rectangle may be represented by two diagonally opposite end points or an end point and a width and height. Third, they form the basis of projects displaying graphics.

To provide the last benefit, ObjectEditor displays a graphics object by the image it represents in a 2-D plane, relying again, on (customizable by the instructor) programming conventions to determine the kind of the graphics object. These conventions are layered on top of the Bean conventions. A class whose name contains the substring Point and defines the X and Y named int properties (using Bean conventions) is recognized as a point type. A type whose name contains the substring “Line”/"Rectangle”/"Oval”/"Text”/"Image” and defines the X, Y, Width, and Height integer properties is recognized as a line/rectangle/oval/text/image-file type. These four properties define the bounding rectangle of the object. The label and image types must, in addition, define a String property giving the text to be used as the label and the name of the file to be used as the image, respectively. All of these types can define several additional optional properties such as Color that determine attributes of the shapes.

Figure 8 shows the code for the type Line and two visual representation of an instance of the type. Both user-interfaces display the geometric shape the object represents. The top user-interface also contains an optional tree window showing a textual representation of the object, which is useful for debugging purposes.

![Figure 9](image)

**Figure 9 Graphics Object With Two Visualizations**

We see above a single graphics object displayed in the graphics window. In general, a visualized object can be decomposed into a tree based on the Bean, sequence, set, and table patterns (we currently do not support arbitrary graphs). All graphics objects in this tree reachable from the top-level object visualized are displayed in the graphics window. In addition to the graphics window, ObjectEditor also creates a tree window to the left of the graphics window (Figure 9) and a main window on top of the graphics window (Figure 11(a)). The tree without the graphics objects is displayed in the main window, and the whole tree displayed in the tree window. ObjectEditor provides user-interface commands and API calls to hide/show these three optional windows.

**7. INCREMENTAL COMPOSITION**

Developers can incrementally build the tree of desired objects. At each stage in this development, ObjectEditor provides a user-interface to visualize and interact with the part of the tree developed so far. This, in turn, has allowed both CS 1 and 2 students to implement assignments that add up to a large end-semester project. Figure 10(a-d) shows some of the stages in a Spreadsheet project - First they developed an evaluator of expressions involving only numbers (a). Then this evaluator became a part of an expression list (b), which then became part of an expression matrix (c), which finally became a spreadsheet in which expressions involved both numbers and variables (d). Similarly, Figure 10(e-h) shows some of the stages in a Halloween CS 2 simulation project. An atomic graphic object representing a candy (e), is composed into a sequence of candies and added to a candy container (f), which is composed with a stick avatar representing a candy taker (g), which is then composed with a sequence of houses with candy (h).

**8. INCREMENTAL COMPOSITION**

Developers can incrementally build the tree of desired objects. At each stage in this development, ObjectEditor provides a user-interface to visualize and interact with the part of the tree developed so far. This, in turn, has allowed both CS 1 and 2 students to implement assignments that add up to a large end-semester project. Figure 10(a-d) shows some of the stages in a Spreadsheet project - First they developed an evaluator of expressions involving only numbers (a). Then this evaluator became a part of an expression list (b), which then became part of an expression matrix (c), which finally became a spreadsheet in which expressions involved both numbers and variables (d). Similarly, Figure 10(e-h) shows some of the stages in a Halloween CS 2 simulation project. An atomic graphic object representing a candy (e), is composed into a sequence of candies and added to a candy container (f), which is composed with a stick avatar representing a candy taker (g), which is then composed with a sequence of houses with candies (h).

A project-based approach is particularly appropriate to teach object-oriented programming as the benefits of such programming are more evident in such projects. Our CS 1(2) students typically create projects with twenty (forty) different classes and interfaces. They are able to do create such large projects because they do not have to worry about details of input/output. This approach has also excited students, as many students go beyond the sample project provided by the professor as a guide. See by comparing Figures 10(1h) and (i).

**9. TOOLKITS AND Model View Controller**

The use of ObjectEditor does not preclude the teaching of toolkits. The tool can work as a set of training wheels that are
later removed when students were taught to create their own user-interface classes. The objects visualized by it are essentially models whose views and controllers are provided by it. These models can later be attached, unchanged, to custom views and controllers written using a toolkit. For example, in one of the CS 1 classes, students first created a turtle-graphics model that was automatically manipulated by ObjectEditor (Figure 11(a)) and then created their own view and controller to interact with the unchanged model (Figure 11(b)). Thus, students concretely see the power of MVC and separation of concerns.

ObjectEditor uses annotations to specify values of customizable attributes. To illustrate, one feature many students ask is a way to influence the (a) stacking order of overlapping properties in the graphics window, and (b) order of properties in the main and tree windows. These two orders can be specified by annotating the getter method for the property with the annotation:

```
@Position(int)
```

where the parameter gives the position of the property.

Annotations are also used to document or override the kind of data type implemented by an object. For example, a class implementing line conventions can be associated with the annotation:

```
@StructurePattern(StructurePatternNames.LINE_PATTERN)
```

to indicate that it implements a Line graphics type. Such an annotation allows ObjectEditor to determine if the class follows the conventions of a Line graphics type, and give errors otherwise, which is useful when students are learning the conventions. Thus, ObjectEditor provides a setting to motivate and explain to them the concept of annotations.

10. ANIMATION AND THREADS

It is attractive to animate actions on graphics objects, such as a turtle moving to its next position and a candy moving from the container of a house to that of the avatar. Such an animation involves writing a loop of the kind shown in Figure 12.

```
public synchronized void animatedMoveX(
    int increment, int steps, int pauseTime) {
    for (int step = 0; step < steps; step++) {
        sleep(pauseTime);
        moveX(increment);
    }
}
```

The animate method will not achieve the desired effect if ObjectEditor waits for it to complete. For the animation to work, this method must be active concurrently with the paint() method in ObjectEditor, which draws the animating object on the screen. If the two methods were executed serially, one after the other, then the paint method would be executed before or after the animate method is called, when the animating point is always at its resting position. Thus, it would never show any of the positions the object takes while the method is executing.

By default, ObjectEditor executes a programmer-defined method as part of the thread that handles events and paints. It forks a new thread for invoking a method if the method does not return a value and the keyword `synchronized` appears in its header, as shown above. Even in a manual implementation of an animating user interface, an animation method should have this keyword to prevent concurrently executing painting and animation methods from stepping on each other’s toes as they access shared data structures. Thus, ObjectEditor does not require this keyword to be used in a way that is inconsistent with the semantics of the animation method. CS 2 students write their own threads for animation, and see animation as an application of concurrency. CS 1 students see it as an application of loops.

11. ANNOTATIONS

ObjectEditor allows extensive facilities for customizing the user-interface, which is beyond the scope of this paper, and are not advertised to the students unless they specifically ask for them.