GUIGraph – Editing Live Object Diagrams for GUI Generation Enables New Pedagogy in CS1/2

Duane Buck
Otterbein University
Westerville, Ohio
614-823-1793
dbuck@otterbein.edu

ABSTRACT
The GUIGraph software tool supports a new pedagogy, motivates students, and solves early user-interface issues. Regardless of the type of curriculum, it can provide an initial, intuitive introduction to object-oriented thinking, even before coding is discussed. By editing a UML-like object diagram, the student creates and links virtual Java Swing objects representing a user-interface, and can instantly view and manipulate its realization. GUIGraph is unique in that the student specifies an object structure to be created, equivalent to writing complex source code. When requested, GUIGraph generates the Java source code of an abstract class that constructs the object structure. The student then completes the coding of a concrete class that implements its application specific abstract methods. The functionality of the application is cleanly separated from its user-interface, which helps build design intuition, and iterative refinement of the user-interface is supported by regenerating the abstract class.

Categories and Subject Descriptors
D.1.7 [Programming Techniques]: Visual Programming

General Terms
Design, Human Factors, and Languages.

Keywords
Object-graph, User-interface, and Pedagogy.

1. INTRODUCTION
GUIGraph is a software tool supporting Java Swing development that is being made freely available by the author. While GUIGraph is intended for general use, its design was motivated by pedagogy. It utilizes a live object-graph editor that allows students to build a deep level of comprehension of object structures while synthesizing an application’s interface.

Traditional curricula have relied on the student’s imagination to visualize the object structure implied by source code. Recently, software tools (e.g., BlueJ [2], jGRASP [6] and others) have provided the capability for direct object visualization and manipulation, and the education community has recognized their importance. [7] Those tools are most applicable to objects-early curricula. GUIGraph is a very different tool, but it also provides opportunities for the student to benefit from experiences with live objects, and due to its intuitive nature it has the advantage of being applicable independently of the type of curriculum.

With GUIGraph, the student directly manipulates a UML-like object diagram to define a user-interface, and immediately sees its realization. Because the student does not write traditional source code, GUIGraph has the potential to provide new learning experiences early in the curriculum as students intuitively build object structures to create behaviors.

This paper is structured as follows: Section 2 discusses the potential impact of GUIGraph on pedagogy in the early curriculum. Section 3 describes technical aspects of GUIGraph and its design choices. Section 4 describes object-graph editing within GUIGraph. Section 5 gives examples of two object-graphs and their realizations. Section 6 discusses classroom experiences using GUIGraph. Section 7 compares GUIGraph to other user-interface generators and “object workbench” tools. Section 8 is devoted to how GUIGraph might evolve and makes a call to educators to help shape its direction. After the conclusion is presented in Section 9, Sections 10 and 11 provide acknowledgements and references.

2. PEDAGOGY
Copyright 2011 ACM 978-1-4503-0697-3/11/06...$10.00..Copyright 2011 ACM 978-1-4503-0697-3/11/06...$10.00. With GUIGraph, students specify the creation of object structures without writing traditional source code. They see the realization immediately, and can also extend the generated abstract class with their own methods in a concrete class. GUIGraph is applicable at the beginning of both control-structures-early and objects-early curricula. With an objects-early curriculum, the advantages of live object manipulation are obvious. With control-structures-early, the use of GUIGraph early-on provides an easier transition to objects and classes later. The students are doubly motivated because GUIGraph presents objects in an intuitive way and they are working within a familiar and interesting domain.

An additional benefit of using GUIGraph early in the curriculum is that students create realistic applications starting with their first assignment. There is no need for unrealistic dialog-based interfaces using static methods and console I/O or a series of dialog boxes. Using GUIGraph, the student naturally learns to separate the user-interface from the functionality of the application. In fact, a good source of assignments is to reimplement textbook examples using GUIGraph and cleanly separate those two aspects of the application.

GUIGraph has been under development for several years, and has been used extensively for the past two years. Our curriculum is “inside/out” [4] where the student first implements methods in the context of a class. The student sees a design and implements methods specified in the design. Originally, GUIGraph was seen as a tool for general use, including use by instructors to create
shells for assignments. Its pedagogical importance was recognized later, and that is now its primary motivation.

In my courses a user-interface may evolve over multiple assignments, and GUIGraph has been designed with this in mind. When requested, GUIGraph generates the source code for an abstract Java class that constructs the diagram’s object structure. Because the interface is fully defined in the GUIGraph file representing the object diagram, and the application specific code is contained in a Java concrete class extending the abstract class (and possibly in other application classes), there is never a need to edit the generated abstract file. It is safely regenerated whenever the interface is changed, so there is no need for additional tool support. GUIGraph works equally well with any integrated development environment or with a text editor and the command line. The separation of the user-interface from the implementation of the functionality also helps the student to build design intuition.

3. TECHNICAL OVERVIEW
GUIGraph presents the user with a graph editor with which they create a UML-like object diagram. The graph nodes are primarily virtual Java Swing objects. Directed edges connect the nodes to show aggregation. Some node types are shared and allow multiple impinging edges, although cycles are not possible. Graph manipulations are performed exclusively with the mouse. The left-button is used to select nodes and move them for visual appearance (having no relevance to their meaning). When a node is selected, configuration information for the node may be viewed and edited at the bottom of the window. The right-button brings up a context menu used to create and link nodes.

Although the coverage of the Java Swing library is not complete, a majority is supported. Fortunately, since the introduction of BoxLayout to the Java library, the complex and unintuitive GridBagLayout is no longer required for sophisticated layouts. It is not currently supported in GUIGraph.

An attempt has been made to make the virtual Swing objects and their structural hierarchy correspond as closely as possible to actual Swing objects and their structural hierarchy, as long as that does not interfere with making GUIGraph consistent and easy to learn. For much of Swing, the goals are consistent. The parts of Swing which are inconsistent, hard to learn, or tedious have been “virtualized” and appear to be consistent and simple in the graph. GUIGraph uses three object-graph abstractions from raw Swing, as described below. In reality, everything is done purely within Swing when the graph is realized. As a result, unlike some GUI generators, no special runtime library is required.

The first abstraction is the virtual object that is a component of a JTabbedPane. In raw Swing, there is only a tab index that is used to refer to a particular tab and manipulate its attributes, instead of having a “JTab” object. In GUIGraph the type of this “object” node is int with an annotation indicating it is the tab index. The tab’s attributes are attached to the node, and the int may be given a Java identifier to use in the concrete class to identify the tab when JTabbedPane method invocations are required.

The second abstraction is that you may add any number of JToolBar objects directly to a virtual JFrame (or other top-level container) using the context menu command “Add JToolBar.” In Swing, each JToolBar object must be added into its own JPanel with BorderLayout, which is then added into the Center of an enclosing JPanel. The ultimate content of the JFrame is added into the Center of the inner most JPanel. This complex structure is built automatically by GUIGraph when the interface is realized.

The third abstraction in GUIGraph is that JPanels are unified with their layout manager into one object node. This is because the layout manager has an important influence on how the JPanel is used. The node’s icon pictures the JPanel’s layout.

When a graph is constructed, it starts with a top-level container. Currently, the supported top-level containers are JFrame, JDialog, and JPanel. Note that JPanel is provided so that users can utilize GUIGraph to design part of a larger user-interface. The user can then add content to the container, and set its attributes. At any time, the user can display the current realization. There is also the option to have the realization updated after each graph edit, which works particularly well. The graph itself is saved for later editing: The GUIGraph file (with suffix “.gff”) becomes one of the application’s “source” files.

Figure 1: Diagram of NameConfigAbstract
Attributes of a top-level node include the name of its generated abstract class and its package (if any). This information is used when the user requests generation of the abstract class or an initial concrete class.

The user decides which Swing objects are made available to the concrete class by giving each of those object nodes a Java identifier. The user can also add a “Method” attribute to any Swing object that supports listeners. This causes the method to be declared as abstract, and code to be inserted into the constructor to...
make the connection. The initial generated concrete class has a stub for each method that displays a message dialog.

Because code generation is potentially confusing, care has been taken to minimize possible loss of work. Each generated source file begins with a date and time generated comment. When generating, if the file already exists, it is examined to see if it has been modified. If so, the user is advised of the possible conflict. The abstract class also begins with a comment warning that the file should not be edited because it may be regenerated.

![Figure 2: Realization of NameConfigAbstract](image)

The graph editor was designed to be both robust and intuitive, and evolved with experience. Each node corresponding to a virtual Swing object is represented as rectangle. At the top of the rectangle, standard UML object diagram notation is shown including its type and (optional) Java identifier. Below it in the rectangle is an icon evocative of the object it represents. Often these are the bit-maps rendered by an actual Swing object. When a node is selected, its configuration data is shown at the bottom of the window. The configuration data is specialized to the node type, and the node’s icon visually reflects the configuration data. The icon changes whenever the configuration data is updated.

While configuration data applies only to one node type, attributes are not specialized to one node type. To reduce visual distraction, attributes are shown with nodes drawn as rectangles with one line of text, and they have incoming edges labeled with the name of the attribute. The type of an attribute is not shown explicitly, but may be deduced from the syntax of its text. Java identifiers are shown verbatim. Strings are shown with the text inside double-quotes, and Dimension objects are shown as ordered pairs.

When editing the diagram, everything is accomplished with the mouse. The left button is used to select a node or drag it to a new (visual) location in the diagram. The right button brings up a context menu. A context menu allows new nodes (virtual swing objects and attributes) to be created and linked to the identified node. It also provides the option of linking existing nodes by first selecting one of the nodes and then pointing to the other node while bringing up the context menu. In addition, one may unlink a node from incoming edges, duplicate or delete a single node, or its subtree. Right-dragging is like left-dragging, except that the node and its subtree are moved.

When developing a complex interface, the diagram can become busy, so the editor allows parts of the diagram to be collapsed and expanded when needed. The context menu has a “Frame Children” checkbox that when selected causes a frame to be drawn around the identified node and its children. If a child is itself framed or is a shared node, it is not included in the parent’s frame. Double-clicking the frame will collapse it so that only the parent is shown and outgoing child edges are shown connecting to the parent. Double-clicking a collapsed frame expands it again. When a frame is selected, it is brought to the front. This scheme allows the user to utilize the available diagram area more efficiently while observing the overall structure.

![Figure 3: Diagram of GUIGraphAbstract](image)

## 4. THE GRAPH DIAGRAM EDITOR

The graph editor was designed to be both robust and intuitive, and evolved with experience. Each node corresponding to a virtual Swing object is represented as rectangle. At the top of the rectangle, standard UML object diagram notation is shown including its type and (optional) Java identifier. Below it in the rectangle is an icon evocative of the object it represents. Often these are the bit-maps rendered by an actual Swing object. When a node is selected, its configuration data is shown at the bottom of the window. The configuration data is specialized to the node type, and the node’s icon visually reflects the configuration data. The icon changes whenever the configuration data is updated.

While configuration data applies only to one node type, attributes are not specialized to one node type. To reduce visual distraction, attributes are shown with nodes drawn as rectangles with one line of text, and they have incoming edges labeled with the name of the attribute. The type of an attribute is not shown explicitly, but may be deduced from the syntax of its text. Java identifiers are shown verbatim. Strings are shown with the text inside double-quotes, and Dimension objects are shown as ordered pairs.

When editing the diagram, everything is accomplished with the mouse. The left button is used to select a node or drag it to a new (visual) location in the diagram. The right button brings up a context menu. A context menu allows new nodes (virtual swing objects and attributes) to be created and linked to the identified node. It also provides the option of linking existing nodes by first selecting one of the nodes and then pointing to the other node while bringing up the context menu. In addition, one may unlink a node from incoming edges, duplicate or delete a single node, or its subtree. Right-dragging is like left-dragging, except that the node and its subtree are moved.

When developing a complex interface, the diagram can become busy, so the editor allows parts of the diagram to be collapsed and expanded when needed. The context menu has a “Frame Children” checkbox that when selected causes a frame to be drawn around the identified node and its children. If a child is itself framed or is a shared node, it is not included in the parent’s frame. Double-clicking the frame will collapse it so that only the parent is shown and outgoing child edges are shown connecting to the parent. Double-clicking a collapsed frame expands it again. When a frame is selected, it is brought to the front. This scheme allows the user to utilize the available diagram area more efficiently while observing the overall structure.

## 5. EXAMPLES

Parts of the GUIGraph interface were developed within GUIGraph itself. The examples shown below display GUIGraph diagrams (“.ggf” files) used in generating GUIGraph’s interface.

### 5.1 The Java Identifier JPanel

Figure 1 shows the GUIGraph application editing the object diagram corresponding to the NameConfigAbstract class. Figure 2 shows the realization provided while editing the graph.
At the lower left of Figures 1 and 3 can be seen the result, within the GUIGraph interface itself, of the generated abstract class.

5.2 The GUIGraph Application JFrame

Figure 3 shows the object diagram for the abstract class (GUIGraphAbstract) used to generate the GUIGraph application’s JFrame. Note that the Action nodes are shareable within a GUIGraph diagram, like the actual Swing objects. Figure 4 shows the user-interface’s realization while editing. The Edit menu is shown opened to demonstrate that the realization may be manipulated and that the same Actions are available from both the toolbar and menu. The generated abstract file is extended by the GUIGraph concrete class to produce the interface shown in Figures 1 and 3. The Center and South of the “Content” JPanel are added by the constructor of the GUIGraph concrete class by using the configBorder Java identifier.

Figure 4: Realization of GUIGraphAbstract

6. CLASSROOM EXPERIENCES

GUIGraph has taken a central place in my courses for the past two years. I use it extensively in CS1 and CS2, as well as in other courses. In practice, it takes only minutes to create significant interfaces, even with the initial learning curve. As a project, one of my Practicum classes produced a video demonstrating the development of a Fahrenheit to Centigrade converter using GUIGraph. I have used this short video to introduce GUIGraph with success in CS1 and other courses.

It is liberating for students to be able to implement realistic applications. GUIGraph works so well that a question arises: Do we really need to devote much class time learning to hand-code GUIs? My tentative answer is no, unless it is motivated by another topic, like studying a design pattern. After using GUIGraph, hand-coding is like running in lead boots. It is analogous to the question of whether or not, and if so when, you should teach assembly or machine language. When studying textbook GUI chapters, I have the students implement the exercises in GUIGraph and examine the generated code, rather than write the code from scratch.

7. RELATED WORK

7.1 GUI Generation

User-interface generation is an area of intense interest. A detailed review is beyond the scope of this paper, so I summarize my observations here. Generators fit into three genres. The first utilizes a pictorial representation of the interface (referred to here as WYSIWYG, what you see is what you get). The second uses a textual representation of the user-interface elements and their relationships. Finally, research projects have implemented GUI generators based on higher level models of applications and their usage scenarios. It is interesting that popular CS1 Java textbooks do not utilize GUI generation. [3] This may be indicative of the complexity of GUI generators, and/or that using them does not contribute to the course’s curricular learning outcomes.

7.1.1 WYSIWYG Based Generation

With WYSIWYG based generation, user-interface components are placed onto a form. In some cases, positioning is absolute, which has very limited value for most applications because resizing of windows is expected. In more sophisticated systems, an attempt is made to deduce the relationships of the components and determine how they should be positioned as the window size changes. Some WYSIWYG generators (such as GLADE [1]) use a two-dimensional hierarchical containment structure.

WYSIWYG based generation is similar to GUIGraph in that it has an iconic representation of the widgets, and they can be given properties such as runtime identifiers and action methods. The explicit representation of the hierarchical containment structure can capture some of the meaning of GUIGraph’s object-graph.

The fact that GUIGraph makes no pretense to be WYSIWYG gives it advantages over the WYSIWYG based systems. With WYSIWYG systems, the user must coordinate viewing the information in two or three windows, and shared components cannot be shown explicitly in a visual two-dimensional containment hierarchy. A GUIGraph object diagram shows the complete description in one view, and instantly displays the realization separately. This is superior because it displays the actual interface, not a simulation, the user can manipulate the interface and observe its behavior when resizing its window, and immediate feedback helps to reduce the learning curve.

7.1.2 XML Based Generation

With XML based generation, a user-interface specification is written conforming to a DTD definition defining the language. The user-interface code is generated by processing the XML file. In some cases, multiple platforms can be targeted with the same specification. Some examples of XML based generators are: SwiXml, Thinlet, XUL, XULUX, Jelly, and SwingML.

XML based generation is similar to GUIGraph in that the object structure is specified by the developer and is directly mapped to the generated code. It is dissimilar in that the object structure is not visual, and tedious text editing is required that is similar to writing source code.1

7.1.3 Model Based Generation

An example of this type is a generator that takes a description of the application data and lays out forms for data entry. These generators are based on research into models describing applications. For more about this research area, see the survey by Paulo Pinheiro da Silva [9]. He says model based generators “are often hard to learn and use” and talks about the need to overcome the complexity by “providing features such as graphical editors, assistants and design critics to support UI designers.” It appears model based generators are targeting a sophisticated audience and are not applicable in the early computer science curriculum.

7.2 Object Visualization/Manipulation Tools

Below, GUIGraph is contrasted with important educational tools that overlap with it pedagogically. It is worth noting that the Scratch [10] tool, although very different, is in one fundamental

---

1 Some WYSIWYG based systems output specifications that target text based generation.
way more similar to GUIGraph than those tools: The user creates objects (sprites) that become a permanent part of their application.

7.2.1 Blue and BlueJ
The Blue environment object workbench is the earliest example of an object visualization/manipulation tool. [8] At the ACM SIGCSE 2010 conference, Blue was referenced as one of the most influential tools in the history of computer science education. [7] The BlueJ environment is derived from Blue. Like Blue, it displays an object workbench and is intended to support an objects-early curriculum. [2]

Although both GUIGraph and BlueJ display and manipulate objects, they are not otherwise similar. The least significant difference is that while BlueJ uses UML-like notation for object nodes, it does not show the runtime structure by displaying the edges of an object-graph. A more significant difference is that although a user could build a user-interface runtime structure in BlueJ by creating Swing objects and invoking their methods to connect them, it would be tedious and similar to writing code. The most significant difference is that nothing is remembered permanently: BlueJ’s object workbench is useful for experimenting, not for implementing.

7.2.2 jGRASP
The jGRASP environment also has an object workbench. [6] Instead of being pictured as nodes in a UML-like object diagram, objects are shown textually in a list. However, an advantage of jGRASP over BlueJ is that it supports object structure visualization. When a parent object is opened, its children are displayed under it as indented sublists, showing a visual tree structure. Otherwise, the jGRASP workbench is similar to BlueJ’s and the more significant differences noted above apply, making it very different from GUIGraph.

Recently, an object-graph visualization capability was added to jGRASP [5], but it is intended to display and animate certain types of linked data structures, not to manipulate object structures.

8. FUTURE DIRECTIONS
It is possible to increase GUIGraph’s coverage of the Java Swing framework, and I plan to do that, although there is not a pressing need for it. The constructor of a concrete class can easily extend the generated code with any missing feature.

Perhaps there is a greater need for similar support for other languages used in education. Because of its design, much of the GUIGraph code-base could be shared by generators for other platforms. Software similar to GUIGraph could be developed for another specific platform, such as Python’s pyGTK. Another avenue is to develop software similar to GUIGraph that outputs a specification for an XML based generator. In that way, GUIGraph would support all the platforms supported by that code generator. I have yet to investigate the best approach and I solicit feedback from the community.

Although I believe GUIGraph is applicable in the workplace, my primary interest is in educational usage. I am interested in collaborating with educators to apply and refine GUIGraph.

9. CONCLUSION
GUIGraph is a GUI generator that is motivated by pedagogy. It is easy to learn and utilize even before the student starts to write source code. Similar to the venerable “object workbench” tools, the student learns how object structures create behavior by direct experience. However, similar to Scratch, using GUIGraph the student may implement, rather than just experiment, because unlike runtime objects, the object-graph may be saved. With GUIGraph, there is no need to expose the student to dysfunctional user-interface coding early in the curriculum. GUIGraph separates the user-interface from the application’s functionality, building design intuition. It turns a serious curricular difficulty, dealing with the user-interface, into an advantage, motivation for studying object structures. Because it is highly intuitive, GUIGraph provides new opportunities for enhanced learning outcomes in CS1 and CS2 regardless of the type of curriculum.

10. ACKNOWLEDGMENTS
The initial classroom version of GUIGraph was developed with the support of Otterbein University while I was on sabbatical leave during the autumn of 2008.

I thank my students for their enthusiastic support of GUIGraph. Their feedback proved to be invaluable as GUIGraph evolved. I particularly thank the students of my winter 2010 Practicum class for improving the code dealing with reading and writing the “ggr” file, developing icons, and producing an introductory video.

11. REFERENCES

---

2 The object structure may be explored by opening viewers.

3 If a node is shared, it appears multiple times in the tree. If the structure has cycles, the result is that the tree can be harmlessly opened to an unlimited depth as a cycle is traversed.