Interface Builder

Jason Chalecki
Human-Computer Interaction Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
chalecki@andrew.cmu.edu

ABSTRACT

Implementation of a graphical user interface for a non-trivial application tends to be tedious and error prone. A simple, visual GUI builder was implemented to explore issues with the current state of graphical application development. Various implementation issues are discussed in relation to enabling a GUI builder tool to be easily extensible to prevent discouraging stagnation of GUI controls. Additionally, design/interaction observations are discussed in the context of making the user's experience efficient and integrating with the rest of the development process.

Keywords

Graphical User Interface (GUI), Interface layout, GUI development, GUI tools.

INTRODUCTION

For over two decades, a Graphical User Interface (GUI) has been a common method of interacting with computer systems. However, they tend to be very tedious to implement due to the many different concerns that need to be addressed. Drawing non-trivial objects, i.e. anything more complicated than a basic shape, tends to require a lot of code. There are lots of different logical mechanisms in which the user could provide input. Providing rich interaction with these mechanisms, or controls, generally requires writing a lot of case specific code. The user is typically allowed to drive the order in which input is given, which requires more flexibility in the application. Again, this results in more code. As the volume of code increases, it gets increasingly more likely to commit coding errors, thus programming GUIs tends to be error prone. Graphical control libraries or toolkits have mitigated a lot these issues by providing implementations for commonly used input and output controls. This relieves the programmer of having to implement much of the "small picture" functionality associated with an application. However, the programmer still generally has to worry about where the controls are positioned on the screen, which still tends to require a good bit of code to implement. Within this paper, the implementation of a tool to allow developers to generate code for a graphical layout through visual creation of that layout is explored. Before this is done, however, some alternative or similar solutions to the problem are briefly reviewed.

RELATED WORK

There have been several classes of solutions to mitigate this problem. Some are more like "syntactic sugar" for the initial problem in that the programmer still needs to consciously think about each and every detail, but will need to write less to specify them. Some reduce the amount of effort required by trying to provide intelligent default configurations whenever possible. Others reduce the amount of effort required by moving the specification mechanism into the problem domain, i.e. allowing the implementation to be made visually.

Resource Files

One of the earliest methods of addressing the issue of simplifying the implementation of graphical layout of controls was through a simple declarative programming language. This approach was common in Microsoft Windows® applications, where it was typically known as a resource file. (The resource file could be used for other purposes, as well.) In this file, a programmer would succinctly specify the size and location, in addition to other control-specific configuration parameters such as the text to be shown for a label control. A separate, provided program would then convert the declarative description into instructions that would display the described interface. While this was method was a lot simpler than implementing it by hand, it was still tedious for the programmer to figure out the exact pixel locations and sizes for each and every control that was to be displayed.

Layout Managers

A somewhat evolutionary solution to this new problem was conceptually very simple. The code library would make intelligent assumptions for parameters that were not explicitly specified by the programmer. This approach is typically known as a layout management. The layout manager would usually take care of configuration with regard to size and position. Many controls would expose a preferred size for its configuration. For example, a button would calculate its preferred size to be that which is just big enough to be able to fit its entire label. The layout manager would use these sizes, as long as enough space was available. For positioning concerns, the programmer would instruct the layout manager as to the relative positions of control. This would typically mean determining the direction in which subsequent controls are displayed, e.g. vertically or horizontally, and the relative order of the controls. Container controls were typically available so that this approach could be used to recursively and simply build up complex and interesting layouts.

Visual Layout Tools

While the approach provided by layout management systems did in fact relieve the programmer from the burden of dealing with somewhat inconsequential, details such as at which exact pixel location should a control be drawn, it still kept the solution to a conceptually visual problem outside of the visual domain. This would generally mean much iteration over the exact configuration of the layout manager and controls so that the displayed interface would appear as desired. A general solution to the initial problem that also addressed the concern arose as graphical applications, themselves, became more popular. In this solution, the programmer visually draws the interface of the application exactly as it should appear. The tool providing this functionality typically provides a palette of the controls that are available. The programmer creates new control instances and directly places them in a representation of the application that is being built. As this is the solution explored by this paper, issues arising with this approach will be discussed later in this paper.

DESIGN

As previously mentioned, approach to dealing with the issues associated with layout of graphical controls explored in this paper is that of a visual layout tool, which is named Interface Builder. Consistent with this, the primary design philosophy was to allow interaction in its most natural modality whenever possible. This typically meant enabling input through direct manipulation.

The most salient examples of where this philosophy was evident are the resizing and moving of controls. All controls could be positioned on the canvas, which represented the display associated with the interface being designed, through clicking down on them and holding with the pointer device and dragging them to the desired location. For controls that were fully resizable, a narrow region around the boundary of the control could be clicked and held and then dragged to increase or decrease the controls size in that dimension (or both dimensions is a corner was clicked). For controls that were only resizable in

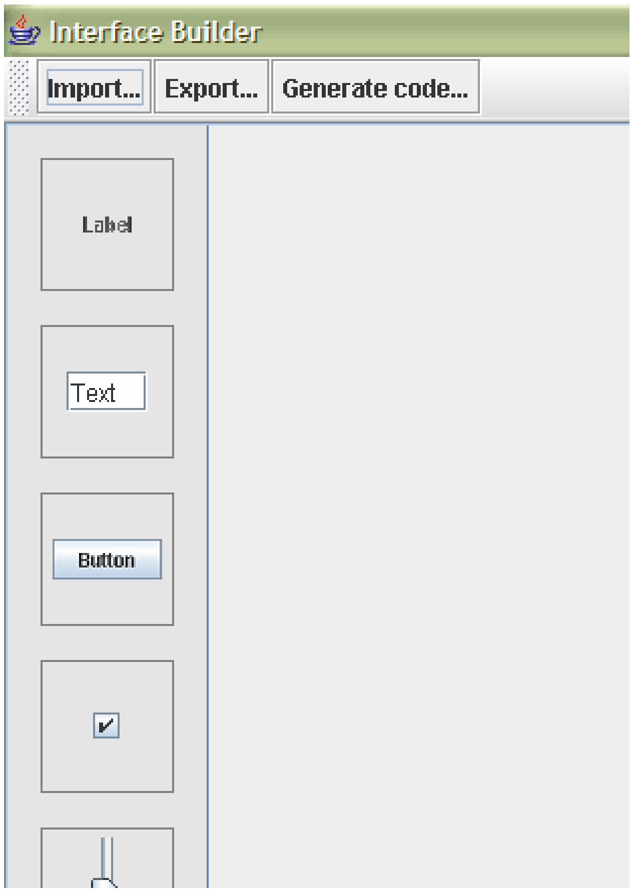


Figure 1. The palette used to create new controls.

one dimension, e.g. a text field can be made wider or narrower but not taller or shorter, the ability to resize was only available on the appropriate edges.

The other example of this is the manner in which the z-order, that is, the order determining which control is on top when controls overlap, is manipulated. Here all controls are presented in a vertical list. Similar to the method for moving controls, the representation of controls, which form the list, can be "grabbed" and moved to the desired new position.

Interface Definition Tasks

There are several core tasks involved with defining an interface for a graphical application. They will be described briefly, each followed by a description of the feature(s) in Interface Builder that is used to accomplish them.

Creating a new control

The first task that must be completed before any other task may start is deciding on a control that is needed and adding it to the application. Interface Builder provides a palette of available controls on the left edge of the application (See Figure 1). A control type is chosen by clicking and holding on one of the items in the palette. A new instance of the

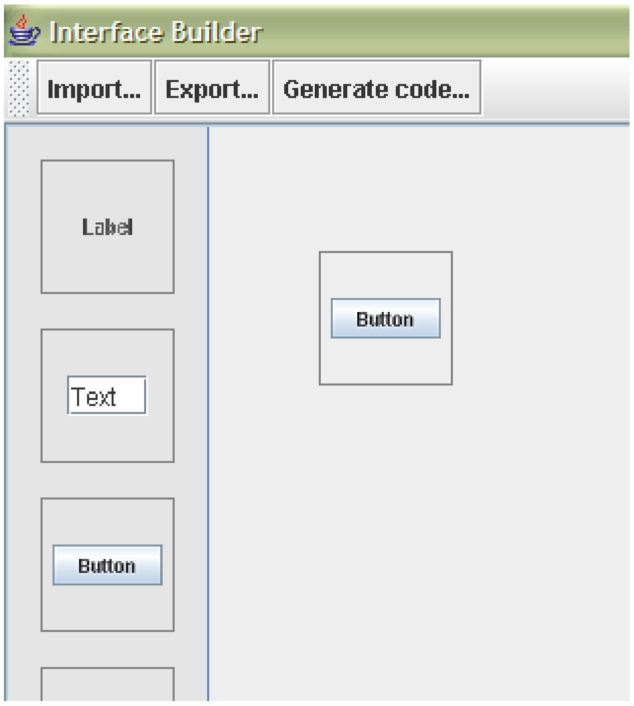


Figure 2a. A button palette item is held above the canvas.

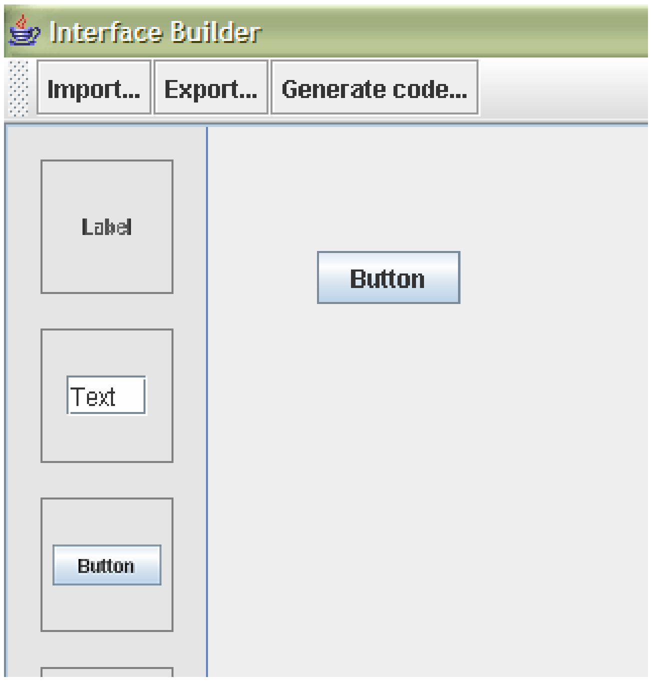


Figure 2b. A button palette item is released over the canvas, creating a new button control.

control type is created by dragging the held item into the layout pane and releasing it there (See Figure 2a & 2b). Feedback is provided to indicate that no action will be taken if the item is released in the palette (See Figure 3). The

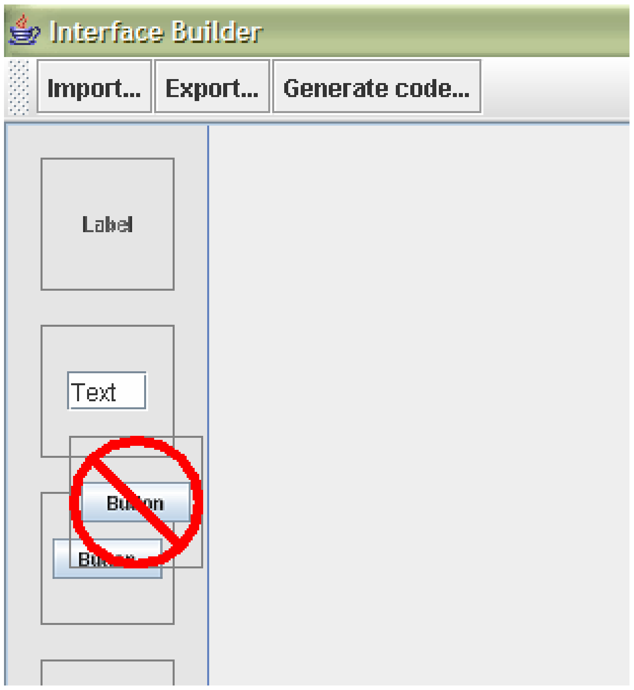


Figure 3. Feedback indicating no action will be taken if the item is released here.

initial position of the new control instance is at the location where the item was released.

Positioning and sizing
After a control is created, it can be positioned and sized through direct manipulation on the canvas. The canvas is the large area of Interface Builder adjacent to the palette and roughly occupying the center of the application (See Figure 4). It represents the display area of the interface being designed. The gesture used for resizing is similar, if not identical to that used to resize windows in most windowing operating systems, i.e. grab and drag a corner or edge. Any part of the control that does not enable the control to be resized when grabbed is used to enable the control to be moved.

Determining z-order
When multiple controls exist, there is the possibility that some controls will overlap. To determine which control is on top, the z-order can be manipulated. The z-order pane, located in the lower right corner of Interface Builder is used to manipulate the z-order of controls. A representation of each control is depicted in a vertical list in the z-order pane. A control that is above another control in z-order will be higher in the list, i.e. more towards the top. Having the list be vertical provides a better match with users' conception of above and below. To change the relative z-order of a control, that control can be dragged to the new desired position (See Figure 5).

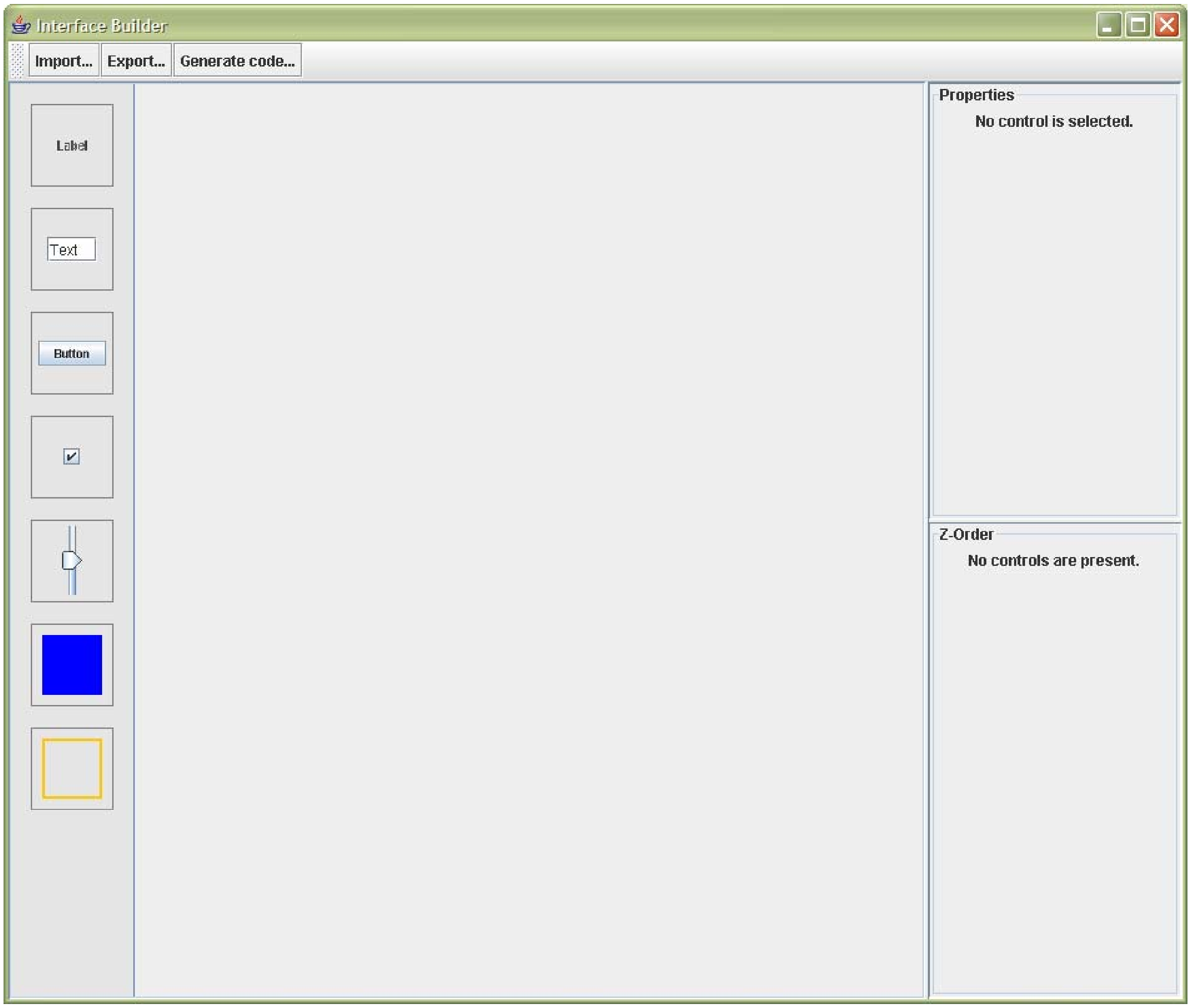


Figure 4. The Interface Builder application: the palette is on the left, the properties pane is in the upper right, the z-order pane is in the lower right, and the canvas is in the center.

Configuring parameters
In addition to size and position, most controls have other properties that can be configured to affect the control's appearance and behavior. Interface Builder includes a property editor, located in the upper right corner of the application. The property editor exposes all properties on the control, such as the label on a button, as textual values (See Figure 6). Properties that are read-only are not editable. Property values are updated in real time so they always display the control's current configuration. Size and location are also just properties so they are visible, which is helpful for high precision size or location tasks (and the values can be directly set, depending on the control type.)

Generating code
After the interface is completely defined, the code to create the interface must be generated. Interface Builder provides a simple function to generate Java code in a user specified file.

Supporting tasks
Often defining an interface is a complex task that requires time and several iterations. To add these needs, Interface Builder includes the capabilities to export the current interface definition and load it back at a later time so that the interface definition process can be interrupted and resumed.

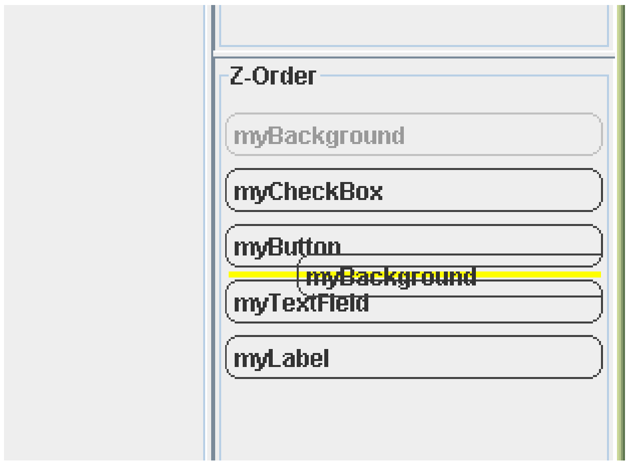


Figure 5. The myBackground control is about to be reposition below the myCheckBox and myButton controls.

IMPLEMENTATION

Implementation of the Interface Builder additionally revolved around two more principles: control extensibility and editor extensibility.

Control Extensibility

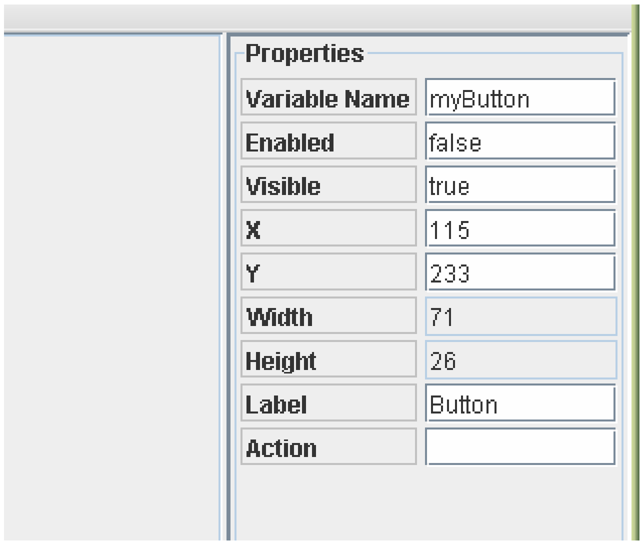
There rate of innovation in the sphere of control types seems very slow. While this is probably not due to visual layout tools being static in their control selection, the tools used do, in large part, define how work is accomplished. Additionally, due to implementation resource constraints, the full suite of common controls was not able to be supported. (Currently the supported controls are: label, text field, button, check box, vertical slider, a filled rectangle and an outlined rectangle.) To address the first possible concern, and in anticipation of the second, Interface Builder was architected from the beginning to facilitate addition of new supported control types.

Control interface

The primary mechanism that accomplishes this goal is the use of a very simple contract that each control must follow. Principle in this contract is the aspect of control properties. Each control provides two methods to get and set each of its properties by name. The values are exposed as strings. Additionally, a method must be provided that enumerates the names of the controls properties. In addition to these three methods, functionality must be provided to save the control and generate Java code based on its current configuration.

Control factory interface

The control interface contract addresses the tasks of configuring a control and generating Java code for an interface. However, before a user could ever get to that point, controls must be created. To provide this functionality, a control factory must be provided for each type of control. The control factory is responsible for



Variable Name	myButton
Enabled	false
Visible	true
X	115
Y	233
Width	71
Height	26
Label	Button
Action	

Figure 6. The properties for a button control, which has intrinsic, i.e. not directly mutable, size.

instantiating new control instances in a default configuration. It additionally provides the functionality to load previously saved controls. Finally, it must provide a canonical depiction of its control type. This is used, for example, by the palette to show what controls are currently supported.

Editor Extensibility

There will always be better and different ways to accomplish a task. Interface Builder prepares for this by encouraging a loose coupling between controls and an editor and between editors. Interface Builder currently has three control editors: the layout pane, the properties pane, and the z-order pane. (The palette is conceptually not quite an editor, and in implementation, together with the canvas, it is part of the layout pane.) By encouraging loose coupling, Interface Builder facilitates modification of itself to add new and better editors as they arise.

Loose coupling

The mechanism by which a loosely coupled implementation is achieved primarily consists of a central object (hereafter "manager") and an event notification framework (See Figure 7). The manager keeps track of all controls currently defined as well as all control factories that are available. Editors register with the manager to receive notification about interesting events such as controls being added, control properties being changed, or a control being made the selection. Additionally, editors may register with the manager to receive notification about control factories, such as new ones being made available. The layout pane does this on behalf of the palette to expose the available factories to the user so that new controls may be instantiated.

An aspect created by control extensibility aids in the loose coupling. By having each property be discoverable and

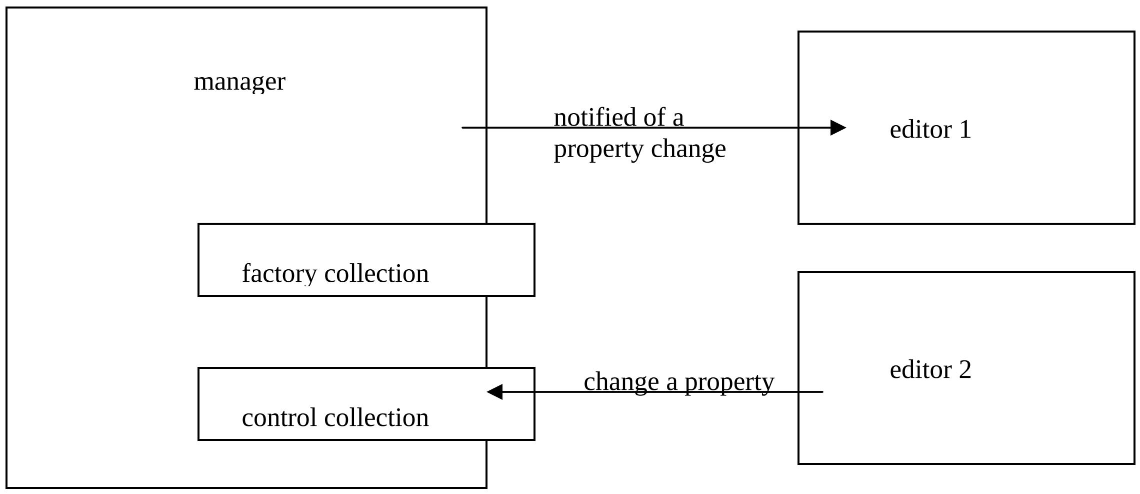


Figure 7. A simple example of communication flow when a property on a control owned by the manager is modified by an editor.

exposed in a common, simple format, existing editors can typically work well on new types of controls.

ISSUES ENCOUNTERED

By in large, there were very few issues encountered during the implementation of Interface Builder. There was one issue of interest from a design perspective and one issue of interest from an implementation perspective.

Design Issue

The design issue encountered concerned the z-order pane. While the vertically oriented list is probably a natural presentation for working with the z-order of controls (which would require user testing to confirm), it was unclear how to best associate each entry in the list with the corresponding control. A simple way would possibly be to show a scaled version of the control as the entry. This has issues in that some different control may not be visually distinguishable. Or, more subtly, some different controls may only be visually distinguishable by size, which would be destroyed by scaling. The current implementation simply shows a property of the controls that is required to be globally unique, i.e. the variable name used in generated Java code. This is sufficient to distinguish between controls, and in small interfaces with well named variables, it is probably even an acceptable solution. However, it is unlikely that this solution is satisfactory for the interfaces that are present in most widely used graphical applications. A possible solution would be to combine the two approaches since multiple cues will make it easier to recall. For the specific combination of editors currently supported, adding highlights, or a similar device, to controls immediately above and below where the active control would be reordered to would probably be very helpful. (The active control is currently made the selection so that provides a cue.) However, care would need to be taken in designing this capability so that the layout pane and z-order pane do not become tightly coupled.

Implementation Issue

The implementation issue concerned the layout pane. The single biggest amount of time and code was spent on implementing move and resize functionality for controls in the layout pane. It was very tedious to implement all of the cases that need to be handled to support moving and, especially, resizing a control. While a lot of the code is very similar, it was still different enough that it could not be easily factored in to shared code. Yet none of the code was specific to any type of control. Nor even was any code aware of the fact that controls were anything more than a rectangular region. It seems that it would be possible to develop generic controls, or possibly, wrappers for controls, that provide basic functionality to move and resize rectangular controls. This is explored a little further in “Future Directions”.

FUTURE DIRECTIONS

Implementation of Interface Builder highlighted several opportunities for further exploration and investigation.

User Testing

First, and most importantly, user testing should be performed on the current implementation of Interface Builder. There are likely issues with the z-order pane. Several solutions have been proposed. User testing would be beneficial in determining which solution would be best (as well as if there is even an issue to begin with).

Direct Manipulation Toolkit

As mentioned in the "Issues Encountered" section, implementing support for moving or resizing of items on screen is cumbersome and tedious. Unfortunately, these are also some of the more natural direct manipulation techniques for graphical applications. It seems unfortunate that current applications must "reinvent the wheel" to support them. (Or worst still, applications may simply not even offer them as input methods.) It appears fruitful to investigate adding generic support for moving and resizing controls to a windowing toolkit. A first attempt would probably restrict this to rectangular controls, with supporting controls of arbitrary shapes being a logical next step.

Active Interfaces

An interface that does not do anything is not that useful. As helpful as aiding in designing and showing an interface might be, there is still a lot of work involved in connecting controls to application specific behavior. Interface Builder currently only addresses this problem by allowing the user to type (or more likely, paste) Java code that will get hooked up to standard event handlers for controls. This is still error prone in part because some of the assumptions of the code are probably dependent on the interface. If the interface changes, there is no way to update the code automatically or even warn the user. The user must track this on their own. Providing the user with a direct and visual way to associate actions with controls in a way where the tool, e.g. Interface Builder, understands the actions and can monitor them for common errors would almost certainly be of great benefit. For some actions, such as setting another control's property to a different value could be done in an almost entirely rich semantic manner, such as by demonstration or by creating different states of the interface and transitioning between them. Invoking application specific logic will probably be less compelling in its experience but more useful to users.

CONCLUSION

While graphical applications have been around, and even common for many years, development tools for creating these applications appear to still be relatively immature. In a visual layout tool, a simple flexible interface for controls and an architecture promoting loose coupling between the major components was found to greatly facilitate initial implementation. It is hypothesized that these characteristics will also aid in its ongoing usefulness by allowing it to quickly grow and adapt to new user needs. The difficulty in implementing simple move and resize direct manipulations, as well as the agnostic way in which it was implemented, led to the conclusion that addition of such functionality to a windowing toolkit would be possible and beneficial. Finally, observations were discussed regarding how the next step in aiding graphical application development appear to be in helping connect the interface to application specific logic, and any tool that can successfully do that will be a great benefit to programmers. How this connecting of interface to logic should be enabled is still an open question.