Pedestal: A Modern Approach To Traditional Framework Design

Joshua Juran

Freelance Software Developer

Abstract

Application frameworks for the Macintosh abound, but applications that lack seemingly standard (or at least highly desirable) features, such as advanced editing keys (e.g. Shift-arrow, forward delete, etc.), or exhibit other annoying defects (like flickering graphics) are even more plentiful. Perhaps the developers of these programs find that existing frameworks don't meet their needs. This paper is not only a discussion of what these needs might be and how they could be addressed, but also a case study of Pedestal, an open-source C++ Mac application framework, in development.

Introduction

Pedestal is a framework for developing Mac OS applications in C++. While it may appear similar (in some respects) to existing frameworks, Pedestal was designed not with similarity to or compatibility with them in mind.

Pedestal is traditional in the sense that it consists of C++ classes that are compiled and linked into the application, as opposed to a dynamically interpreting runtime environment. What makes Pedestal modern is its vision.

Jean-Louis Gassée has described the role of a Macintosh developer as someone who stands inside a swimming pool underneath a user and reaches up to place a palm at the surface right where the user is about to step, so that the user has the experience of walking on water. Pedestal's job is to let the developer do this without getting sore arms.

The primary design goals of Pedestal are fidelity to the Mac interface and spirit (and accuracy in being faithful), elegance in interface and implementation, and extreme modularity in the class hierarchy, trading economy of numbers for ruthless factorization. In other words: There are more classes, but they're simple, lightweight, and they do their job efficiently and effectively. Not only that, but only the functionality an application needs has to be compiled in — for example, the visual containment subsystem fully supports scrolling content, but only if you need it. Even the nesting of views itself is optional — it's either seamlessly integrated or inconspicuously absent. The design of Pedestal's visual subsystem is one of the areas this paper discusses in detail.

Visual Subsystem

Introduction

Most frameworks, in the process of representing a visual hierarchy, resort to a pair of abstract concepts which they call 'views' and 'panes'. Although the names match, the actual concepts they're meant to denote usually vary across frameworks. While this variance is not problematic in and of itself (unless you're attempting to unify the frameworks), there is a disturbing lack of clarity around what the concepts actually are within a particular framework. This is evident from examining the class hierarchy. For example, there may be an abstract View class, which may support subviews or handle mouse clicks. Now, consider a checkbox: A checkbox is a kind of button, which is a control, which is a pane, which is a view. So a checkbox inherits mouse clicks — but it also inherits subviews, which is inappropriate, or at least superfluous. On the other hand, an offscreen view is a view that generally speaking will have subviews (so there's something to be drawn offscreen) but has no need to handle mouse clicks. Although it might never result in erroneous runtime behavior, these inconsistencies nevertheless indicate a flaw in the design of the inheritance model. While it is possible to write working applications based on such a framework, the discrepancy between the model (e.g. the checkbox class) and that which is modeled (the conceptual checkbox that Mac users are familiar with) is a stumbling block for the developer. It is possible to be ever mindful and careful and never stub one's toe, but one is less likely to walk away unscathed than if the block weren't there in the first place.

Views and Panes

Pedestal's primary design goals are accuracy / fidelity, elegance, and modularity. Nowhere are these three more evident than in Pedestal's visual model. The key distinction is the relationship between views and panes. Rather than the traditional incestuous liaisons espoused by already-existing frameworks, Pedestal introduces a respectful but no less intimate courtship. The revolutionary arrangement is this: Views and panes are distinct, disjoint types. One does not inherit from the other, nor do they share a common base (with the possible exception of a framework-wide root class). Furthermore, the abstract View class does not contain panes (or other views), and doesn't even know what panes are. The View class is a paragon of innocence — or at least ignorance. The Pane class, however, does know about views — in fact, every pane is contained by one, which is its superview. But it doesn't contain any panes or views either.

How can this be? How can anything be accomplished in such an austere model? The answer becomes apparent upon examination of a coterie of subclasses.

Nesting

The first question concerns nesting, without which there is no hierarchy. Although the View class (PedView) doesn't contain anything, all view objects can contain panes, by virtue of some derived class. One example is PedWindow, which is defined has having exactly one pane. The mechanism for adding several panes where only one can fit involves a subview. The subview class (PedViewSub), a subclass of PedView, must be further derived from to define exactly how it contains panes (for example, a scroller contains up to two scrollbars and one content pane). The subview is associated with (*not* contained by) a subview-pane (PedPaneSubView) which is installed in the window. The containment of a pane by a view and the link between a subview and its subview-pane are *not* the same relationship. Likewise, a chicken laying an egg and an egg hatching a chicken are inequal operations (and note that the combined process is different from a live birth). In this case, however, we are not interested in the question of which came first (the view, if you're wondering), but instead the two-step manner in which they're nested.

Scrolling

Scrolling is a fairly complex process, involving several different activities. The content has to be redrawn and the scrollbars' values (and occasionally their maxima) must be adjusted. To further complicate things, there are different ways to cause scrolling to occur, with different procedures for implementing scrolling in each case. For example, if the user moves the scroll box, you just compare the scrollbar's value before and after and the difference is how much to scroll the content. However, if the user clicks in a scroll arrow or a paging region, you call ::TrackControl(), which repeatedly calls your ControlActionProc. Instead of sampling the control value to get the scroll distance, you supply it your own. You redraw immediately, instead of waiting for the next update event. But if you're scrolling to make a selection visible, then the distance is calculated and you scroll all at once, redrawing later (as with dragging the scroll box) but you still have to set the control value on the scroll bar. In all cases, if you've just scrolled away an area below the bounds of the content, you need to recalibrate the scroll bar's maximum. Finally, there are applications like MacPaint and Stickies which feature scrolling, but lack scroll bars.

Pedestal's approach is to divide the work. The scrollbar class (PedScrollbar) is a subclass of PedControl which is in turn derived from PedPane. A scrollview (PedViewScroll) is a subview that has one pane (the content pane) and a scroll position that determines which part of the pane is displayed. Scrolling does not affect the bounds of the content pane, just the scroll position. (It helps to think of the scrollview as an actual scroll. The document may be several feet long, but you only see a foot or so at a time.) Rounding out the set is the scroller class (PedViewScroller). A scroller doesn't actually perform any scrolling on its own, but it manages the interaction between the scrollbars and the scrolling object. Usually this object is the scrollview, but if the content pane manages its own scrolling (e.g. a TextEdit pane), then a customized scroller is called for. So a scroller object will be an instance of either PedViewScrollerSimple or PedViewScrollerTE.

Command Subsystem

Pedestal has a command hierarchy similar to that of other frameworks, though with some slight differences. Instead of bearing anthropomorphic names like Bureaucrat and Commander, Pedestal's units of control are simply called 'tasks'. Another, perhaps less nominal difference is that the tasks represented by the abstract Task class (PedTask) are not comingled with views or panes. A task represents some process of finite duration that has a discrete beginning, middle, and end. While some screen elements may be associated with tasks, they are distinct: A click in the close box is interpreted as a command to close the window by the Window class, but the fulfillment of that command (or its cancellation, as well as the invocation of a 'save changes' dialog to determine which) is handled by a task (specifically, an agent).

Tasks may have subtasks to whom subsets of the parent task's responsibility is delegated. The root task is the application itself, whose purpose is to provide some core functionality to the user, and host an interface to expose that functionality. Subclasses include Agents (PedAgent), which manage windows (e.g. setting the window's name, deciding to close the window immediately or ask the user first), Documents (PedDocument), which model documents as we know them (usually with an Agent subtask and a file reference), and Operations (PedOperation), which oversee some operation such as transfering a file or performing a search. While it's quite plausible for an operation to be a kind of task, the derivation of the other subclasses is less intuitive — and explained thus: Whereas a file transfer has a point of initiation, a point of completion, and the work done in between, an application, agent or document is at some point opened, later closed, and in the interim is processing events. Both have a beginning, middle, and end. The conceptual difference is that an operation is 'getting stuff done' while the user is idle, and the tasks which merely respond to events have nothing to do then, so a document appears as a static 'thing', rather than a running process.

In addition to responding to events, tasks can get time in between events to perform chores. A chore (PedChore) is not a task. Rather than having a beginning, middle, and end, the chore is atomic — accomplished in one function call — for example, calling ::TEIdle() or drawing a frame of an animation. In order to be run, a task's chores must be installed either in its repeat queue or its idle queue. The repeat queue is run in between every event, and the idle queue is run only after a null event. A queue is run by sending the appropriate message to the root task, which runs the queue and forwards the message to all of its subtasks.

I/O Subsystem

Pedestal's I/O subsystem is exemplary of the degree of modularity prescribed by the design goals. A supposedly simple operation such as reading a file involves no less than five distinct classes of objects (though only three are instantiated directly): First, the file is located using a file system reference (class PedFSRef). The reference is used to create a file data-source (class PedDataSourceFile), which creates a raw access path (PedAccessRaw) to the file. There are two kinds — PedAccessData and PedAccessRF, for reading the data and resource forks respectively. (There's also PedAccessRes, which is for accessing resource files through the Resource Manager, and therefore can't be used for a data source.) A data source is simply that — it's a source of data that is accessed a bufferfull (PedBuffer) at a time. Finally, an input stream (in this case a buffered input stream (PedStreamInputBuffered)) is created to draw from the data source (or if it's a text source, PedStreamInputBufferedText). An input stream of any kind (PedStreamInput) is read one byte at a time, at which point the application is free to do with it as it sees fit, possibly using a dispenser object (e.g. PedDispenserString) to package the data in some desireable form.

This is even more useful than it may at first appear. Simply by calling GetByte() on a buffered input stream (or using a dispenser, which will do the same thing), blocks of the file are read in and buffered as necessary, and those buffers are automatically deleted after their data have been read. But why is this any better than using the File Manager's built-in buffering?

The advantage of having such highly factored behavior becomes evident upon examining other possibilities for its use. A data source could be an HTTP stream instead of a file. One data source could be passed (through an input stream) to another one which compressed or decompressed it along the way. A special kind of access path could read the 'MacBinary' fork of a file (by internally reading both data and resource forks and performing the encoding). An input stream could return random bytes, null characters, or end-of-file (analogous to Unix's /dev/random, /dev/zero, and /dev/null respectively). In addition, the text input stream transparently converts newlines (the lack of which feature has caused no end of mischief).

An equally full-featured output system is in the works.