Cocoa Interfaces Using the Apple Interface Builder (IB) and Clozure Common Lisp (CCL)

Version 2.0 April 2010

Paul Krueger, Ph.D.

Introduction

This tutorial provides a guide to creating Cocoa interfaces for Clozure Common Lisp (CCL) programs. It is not a comprehensive guide to Cocoa; that is simply too big an undertaking and there is ample documentation from many other sources. What I've tried to do is gradually introduce you to the Cocoa/Lisp development world so that you can go explore the many possibilities on your own with a good sense of how those things might be integrated into Lisp. For the sake of simplicity I have taken a fairly narrow path and not tried to explain or use every possible bell and whistle that Cocoa provides. Hopefully I will have provided enough of a foundation that each of you can individually explore the possibilities on your own.

Although I have tried to be as factual as possible, it is entirely likely that there are things here which are in error. Please notify me of any such things (email plkrueger (at) comcast.net) and I will make corrections as quickly as possible. All of this was done with the Leopard operating system (Mac OSX version 10.5.8), Developer Tools version 3.1.4 and CCL version 1.5-dev-r13388M-trunk (DarwinX8664). Since all of this is a moving target, the diagrams and examples may look slightly different on your system.

Whenever I add a new project I will change the major version number of this document. Minor version number changes indicated corrections without adding a new project. There is also an associated document called "Revision Notes" that has a synopsis of the content changes for each revision.

There are a number of different objectives that you might have for development of graphical user interfaces for CCL programs and it's important to understand what the goals were for this work so that you know whether to keep reading or look for an alternative.
The opinions expressed below are just that, opinions, and others will certainly disagree. So with that in mind, in order of importance to me, the goals that I had for selecting an approach to user interface development were:

1. Usable either to create stand-alone executables or interactively from within a standard Lisp Read/Eval/Print/Loop (REPL)
2. Looks native to the platform
3. Easy blending with Lisp code
4. Development effort commensurate with interface complexity
5. Cross-platform/OS portability

Goal 1: Usable for stand-alone executables or within standard Lisp REPL

One of the primary reasons that I find Lisp to be such a productive language is the ability to try snippets of code and quickly modify/correct them. Standard non-interpretted languages, for the most part, require a sort of code/build/execute/debug loop that I find to be a productivity killer. Modern IDE's like Apple's Xcode can make this somewhat tolerable, but it would never be my first choice. User interfaces, like most code, require debugging and being able to modify them on the fly from within a REPL is simply the easiest method I know for rapid deployment. Given all that, I don't want to do anything that would preclude creating stand-alone executables that use my interfaces. I want the best of both worlds.

Goal 2: Looks native to platform

I've been developing user interfaces of one sort or another for about four decades. One thing that I've observed over that time is that users of systems get used to a particular look and feel on whatever platform they have and won't readily tolerate an interface that departs too much from that. I personally find it annoying when an application behaves in a way that is different from others on the same platform. I don't want my own code to generate that same feeling of annoyance. I suppose this is changing somewhat as web/browser-based applications become more prevalent, but I still believe that for anything that runs natively, as my code will, adherence to standards is a must.

Goal 3: Easy blending with Lisp code

There are various ways that interfaces can be created on each platform. There are innumerable packages that exist to make this easy. For example X-windows, TCL/TK, Java Swing, and many more. None of these is particularly easy to use with Lisp although there certainly have been credible attempts to make them so (e.g. CLX and Garnet). One reason that none of these attempts has been widely adopted is that they often become obsolete rather quickly. That is because

making those interface packages easily accessible to Lisp developers often entails a fair amount of bridge or translation code which becomes a burden for anyone to maintain. User interface packages tend to change rather rapidly and as a consequence the Lisp interface can quickly become out-of-date. The alternative is often to make a bridge that is NOT easy for Lisp developers and that's not much better.

Goal 4: Development effort matches user interface complexity

Developing user interfaces is generally not the main focus of my work. Sometimes I just need something quick and making format calls that print in the listener is just fine. But as the application gets more complex so do my needs for visualization, for changing control parameters, seeing output, etc. I want the effort that needs to be put into those interfaces to be as little as possible. I want intuitive easy-to-use tools to assist me in their development.

Goal 5: Cross-platform/OS portability

So basically I'm a Macintosh developer. This goal almost isn't on my radar, but I certainly recognize that for others it is a must. While I didn't go looking for a solution that made this a high priority I was pleased to find that there is some potential for cross-platform portability with the approach that I have taken although it is clearly not available today (January 2010). More on that later.

The Search for a Solution

I thought it might be useful to recount my thinking and processes for deciding how I wanted to build user interfaces and use them within Lisp. If you don't care how I got to my approach, you can skip to the next section.

My platform of choice is the Apple Macintosh. So I started my search by trying to understand how user interfaces are constructed in that environment. That very quickly led me to Cocoa and I decided early on that any reasonable approach must use it. I learned much of what I know about Cocoa by working my way through

"Cocoa Programming for MAC® OS X", Third Edition, by Aaron Hillegass

and I strongly urge you to buy the book and do the same. This book uses Apple's Xcode and Interface Builder (IB) tools to create progressively more complex user interfaces using Objective-C. Each of his projects introduces one or more new Cocoa and/or Objective-C concepts. What I've tried to do here is to adopt a similar approach, but from a Lisp perspective.

Although I hope my examples and this accompanying documentation will give people a head-start, there just isn't any substitute for understanding how Cocoa works and there are many resources for doing so. I expect that many or most of the people reading this may not have any knowledge of Objective-C and will, therefore, shy away from this approach. All I can say to persuade you is that as much as I like Lisp (at last count I've developed programs in 29 different languages and Lisp is still my favorite) Objective-C is a pretty nice language that you should be able to pick up reasonably easy if you have any C background whatsoever. I assure you that it is nothing like C++ if that helps any.

As I worked through the Hillegass book I became more and more fond of the way that Apple's Interface Builder helped with the design of user interfaces. I didn't need to worry about screen layout or any number of different things that I've always had to consider previously while creating user interfaces. It was, all things considered, pretty easy.

So having zeroed in on Cocoa and having become somewhat familiar with how things worked, I went back to CCL's release to see what they had done to make this easy to use. I first found the easygui code in the CCL release: ... /ccl/examples/cocoa/easygui. In the previous version of Lisp that I used for Macintosh development (MCL), there was a rather nice assortment of user interface objects that you could put together for your own use and it seems to me that the easygui interfaces are an attempt to do something similar for CCL. But as I started to use it, I became convinced that it isn't the best approach either for a couple of reasons.

In some sense the easygui approach represented a step backwards from the easy-to-use drag-and-drop Xcode/IB environment that I had been using. For anything more than a fairly simple window I once again had to think about all the myriad placement and relationship factors that were so easy to do using IB. I didn't really want to go back to a world where I had to *compute* every aspect of the user interface rather than just dragging and dropping and clicking check-boxes to define behavior. The nature of user interfaces has become enormously more capable and commensurately complex over time. That means that to use Cocoa as intended to develop full-featured interfaces would require mastering an enormous number of classes, methods, and interactions between them.

The second problem I had with the easygui approach is that I found myself spending time trying to figure out whether and how easygui had implemented one or another of the classes or features that are described in the massive amount of Cocoa documentation that Apple provides. That mapping challenge seemed like it would be an ongoing problem for anyone who aspired to make more complex user interfaces. When you couple that with the fact that Cocoa is a constantly changing target, it just seemed to me that Easygui would require a very substantial amount of ongoing maintenance and documentation.

So I went back to Xcode thinking that perhaps I could integrate CCL into that environment. I made some progress in this direction, but a little voice kept nagging at me that I wasn't going to be able to meet my #1 goal with this approach. Xcode is entirely built around the idea of the code/build/execute/debug loop and that just wasn't what I wanted. So after a short time looking at this approach I abandoned it as well.

One day it occurred to me that I didn't need to use Xcode in order to get the advantages of IB. I don't claim this as an entirely new thought, I was just a bit slow in coming to the realization that this was not only possible, but could be done in a way that satisfied all of my goals. I quickly found that I could design my interfaces within IB and save them to NIB files that could be loaded into Lisp rather easily. But what about the ease of integration of this approach with Lisp? What would the Lisp code have to look like as I implemented more complex interfaces?

It is at this point that I have to stand up and applaud Randall Beer, who originally created the Objective-C bridge code in the CCL release, as well as all those who have made it what it is today. This is simply a stupendous achievement that deserves all the recognition anyone can give it. I suspect that it wouldn't have been all that difficult to just use FFI to access the libraries, but that alone would not have made it easy to integrate into Lisp. The real achievements were:

1. The ability to create Lisp classes that inherit from Objective-C classes
2. Creating slots in these classes that are directly accessible to Objective-C code
3. Permitting the creation of Lisp methods that are callable from Objective-C
4. Making it easy to call Objective-C methods using Lisp syntax
5. Permitting Lisp to create, store, and access Objective-C objects just as you would Lisp objects
6. Making it easy to access Objective-C runtime constants and variables
7. Being able to instantiate Lisp classes that inherit from Objective-C classes from Objective-C
8. Automatic name translation between Objective-C and Lisp
9. Being able to call "make-instance" for Objective-C classes
10. Doing many type translations automatically

and I'm sure there are other features that I'm missing.

Initially I would build one of the Hillegas projects using Xcode and Objective-C and then go back to Lisp and create something similar. After doing just a few of these it became apparent that I no longer had to do the Xcode version. I could read the book and go directly to Lisp to create a counterpart. I could modify or augment those behaviors by going directly to the relevant Apple documentation and using it to inform my Lisp development. I didn't have to worry about what the mapping was between Objective-C and Lisp because for the most part it is a 1-1 mapping that uses relatively simple name translation rules.

So let's review what we have against my goals:

1. I can clearly define interfaces that I load into a standard CCL IDE and modify in a REPL environment. I routinely redefine methods while a window is up and immediately see the modified behavior. To change the look of an interface I switch to IB, modify and save, go back to Lisp and create a new instance of the window to see the changes. That's faster than I could figure out how to modify code to generate a new interface as we did with MCL. At this point in time (January 2010) I have not yet made a stand-alone program, but it is clear to me that nothing I have done precludes this possibility. If you want to do this I suggest that you download Mikel Evins' APIS code and start with that:

    http://explorersguild.com/mikelevins/Apis_1_0.zip

Integrating my examples into a stand-alone program using Mikel's framework would mostly consist of creating a main menu nib I think, but until I do everything I can't say what else might be required. I expect to create a project or two to do this before I'm done with this effort.

2. Clearly Cocoa is the way to get a native look and feel on Macintosh systems and I can now access all of Cocoa's capabilities without wondering how some intermediate layer might have implemented those capabilities.

3. The CCL-supplied Objective-C bridge code makes it a pleasure to integrate Lisp with the Objective-C runtime environment. Again, I can't say enough about how well this was done. We have a true Lisp-friendly *interface* to Objective-C rather than a *new layer on top of it*. So as the underlying Cocoa libraries and documentation change over time, we can make immediate use of them without waiting for some maintainer to figure out how to incorporate those changes into a Lisp layer.

4. I find that developing user interfaces can be done very quickly. The easy part is the initial layout using IB and specification of Lisp classes to interface to it. Depending on the complexity of what you are trying to do this is minutes to hours in duration. Typically I require a longer amount of time to debug the behavior that I want, but this is where the Lisp REPL environment proves its worth. Rather than finding a bug and rebuilding as would be required in an Xcode world, I can very quickly examine objects, modify methods, and generally do the sorts of things that developers always do in Lisp.

5. As I said initially, portability was not my main criteria, but as I browsed though some of the CCL Objective-C code I

found references to "Cocotron", so I did a bit of research to see what that was all about. It turns out that this is an open-source project that aims to provide an Objective-C compiler and runtime environment for a multitude of platforms and OS types. It also has implementations of many of the same Cocoa classes and methods provided for the Macintosh, but as Gary Byers said in an email (12/3/09) to openmcl-devel@clozure.com:

"Cocotron does a fairly good job of implementing a large subset of Cocoa and it's under active development, but it's not too hard to run into things that're implemented in Cocoa but not (yet) present in Cocotron or to run into things that're implemented differently (and in some cases incorrectly.)"

That same email provides instruction about how to load Cocotron and try it with CCL if you're really brave. So I think there is some promise of future portability, but as of January 2010 it isn't there yet. For more information about Cocotron see:

http://www.cocotron.org/Info.

Background Reading

CCl provides very nice documentation with their distribution that describes (among other things) their Objective-C bridge code. Assuming you have CCL installed in your applications folder as I do, you can find it at:

file:///Applications/ccl/doc/ccl-documentation.html

In addition to reading the Hillegas book, if you are not already familiar with Apple's Interface Builder, it would be a good idea to read:

http://developer.apple.com/documentation/DeveloperTools/Conceptual/IB_UserGuide/IB_UserGuide.pdf

although if you work through the examples that I have here you should be able to pick up what's going on fairly quickly. You will have to download Apple's developer tools to get IB. I'm not going to describe how to do that here. Apple's website can tell you how to do that and in all likelihood if you're building your own CCL environment you've already done that anyway.

Apple's developer website has an enormous wealth of information. I find it useful to get the latest documentation and have it on my own system. You can do this by running Xcode and using the help facility to subscribe and download the documentation sets. Although this works I would recommend that you get the AppKiDo application. AppKiDo is a reference tool for Cocoa programmers. The latest release, with source code, can be downloaded from

http://homepage.mac.com/aglee/downloads

AppKiDo is free. It creates a very easy way to access the reference documentation that is included with the Apple's developer tools (you will have to run Xcode at least once and use the help facility to subscribe to and force a download of the documents). I have found this to be an extremely easy way to find out more about specific classes or methods or constants that might otherwise be difficult to locate within Apple's documentation.

Setting up your CCL environment

I'm going to presume that you have downloaded the CCL distribution and built the Cocoa-based CCL IDE. To make life a bit easier you may want to set up your environment to make it simple to tell Lisp where to find files. There are some simple things you can do in a ccl-init.Lisp file that you put into your home directory that will make it easier to use my examples. First you can create a logical host directory that points either to my InterfaceProjects directory or to an equivalent directory of yours where you will build up your own versions of my examples. Second, you can tell CCL to search within that directory when we "require" something. Mine looks like the following:

```
;; ccl-init.Lisp
;;
;; loaded by ccl at startup

;; Set up the logical host named "ip" so I can easily use it in pathnames
;; This is basically my Interface Projects" directory. Within this directory I
;; create various project directories and within them I put Lisp code that I might want
;; to "require" in Lisp.

(setf (logical-pathname-translations "ip")
      '(("**;*.*" "ccl:contrib;krueger;InterfaceProjects;**;*.*")))

;; extend the module search path so that we can just "require" a module and ccl will find it
;; among our various projects. "InterfaceProjects" (now logical host "ip:") is a
;; special sub-directory within which I put all my cocoa interface examples, so I put that
;; in the module search path specifically.

(setf *module-search-path* (nconc *module-search-path*
                                  (list (pathname "ip:*;"))))
```

So once that's all done you can either start with my code and read about what I did, try to develop your own from scratch, or modify mine.

If you just want to see what all the fully functional project windows will look like before beginning you can, at this point, start CCL and have a dialog like the following in the listener:

```
Welcome to Clozure Common Lisp Version 1.5-dev-r13174M-trunk  (DarwinX8664)!
? (require :simplesum)
:SIMPLESUM
("NIB" "SIMPLESUM")
? (ss:test-sum)
#<SUM-WINDOW-OWNER <SumWindowOwner: 0x127b8170> (#x127B8170)>
? (require :speech-controller)
:SPEECH-CONTROLLER
("SPEECH-CONTROLLER")
? (spc:test-speech)
#<SPEECH-CONTROLLER <SpeechController: 0x127c44c0> (#x127C44C0)>
? (require :package-view)
:PACKAGE-VIEW
("PACKAGE-VIEW")
? (pv:test-package)
#<PACKAGE-VIEW-CONTROLLER <PackageViewController: 0x13f68bc0> (#x13F68BC0)>
? (require :loan-calc)
:LOAN-CALC
("DATE" "LOAN-CALC")
? (lnc:test-loan)
#<LOAN-CONTROLLER <LoanController: 0x13f9fa90> (#x13F9FA90)>
? (require :loan-document)
:LOAN-DOCUMENT
("MENU-UTILS" "LISP-DOC-CONTROLLER" "NIB" "DATE" "DECIMAL" "LOAN-WINDOW-CONTROLLER" "NS-
STRING-UTILS" "NSLOG-UTILS" "LOAN-PRINT-VIEW" "LOAN-CALC" "LOAN-DOCUMENT")
?
```

The loan-document project has no test function per se. It adds a few menu items that you can use to test it. See a much longer discussion for Project 7. Feel free to play around with them; it may help you understand what is happening when we get to the development details.

Each of these projects is defined within its own package and a separate package is also defined for various utility functions that are common to several projects. So it is safe to load any of these into your environment without fear of name conflicts with anything you have done.

From this point on I will assume that you have an "ip" logical host defined that resolves to some directory with subdirectories for each of the projects we'll discuss. If you do something different you'll have to modify my instructions accordingly. In all cases below where I describe how to build something using IB, you can just open up the corresponding nib file from my example directory if you would prefer.

Warnings:

There are a few things that a lisp user might expect to do which are a bit different when you have to interact with the Objective-C runtime. At least one person tripped across one of these while following along in version 1.0 of this tutorial, so it seemed prudent to provide some warnings before you get started.

The first warning is that we will be defining a number of Objective-C classes as we go on. In Lisp you can define a class, modify the source for it in some way, and then redefine it with the new description. Common Lisp explicitly handles this in very nice ways. Objective-C does essentially nothing here. You just can't define an Objective-C class, modify it, and then redefine it at runtime. Sorry, but if you need to do this you'll need to restart Lisp and reevaluate the source to get the new definition. Bummer. On the bright side, you CAN dynamically redefine Objective-C method definitions at runtime and get the effect that you expect (mostly). You may find that the fact that a class has a method defined has been squirreled away somewhere, so if you define a new method you may need to create a new instance of that class to get it to find your new method. If this is confusing, don't worry about it right now. All will become clear later.

Although all files load fairly quickly, you may decide to compile them. If you do, you will note that the compiler will generate a warning for any Objective-C method that we define that is not a standard one from the library. Example:

;Compiler warnings for "ip:Simple Sum;simplesum.lisp.newest" :

; In SIMPLESUM::|-[SumWindowOwner doFirstThing]|: Undefined function NEXTSTEP-FUNCTIONS:|doSum:|

This in fact occurs in a method that is in the process of defining that doFirstThing: method, but the compiler chooses to warn us that it is not a preexisting function name from the Objective-C library.

Let's get started!

Project 1: Hello World

Key Concepts: Interface Builder, NIB and XIB files, NIB loading, NIB file owner

We must begin by talking generally about what IB does. Basically it is a graphical design tool that lets you put together standard Apple interface objects and to specify how those objects will interact with other classes that a program defines. It packages up that design into a file that can be loaded at runtime by an application. When the file is loaded, instances of all of the objects it describes will be created and linked together to recreate the interface that you designed. The objects that it creates can include instances of classes that you defined in your application.

Start by executing IB. You will see three windows:

1. An inspector window that will initially be titled "Attributes".
2. A Library window that allows you to select interface elements and drag them onto your design.
3. A window titled "Choose a template"

The third window presents you with a dialog that asks you what sort of interface you want to build. Select "Cocoa" in the leftmost column and "Window" from the options provided in the rightmost column. When you select this the dialog will disappear and two additional windows will come up:

4. An untitled *nib document window* that represents all of the objects that will be contained in the NIB file that you will load into Lisp.
5. A window titled "Window" that is the main window object that you will design and depicts will be displayed in Lisp.

If you click on window #5, the inspector window (#1) will now show the attributes of your window object. Click in the "title" box in the inspector window and change the title to something you like, say "Hi there!". You will see that change immediately reflected in window #5 which now has your title. Now in the Library window (#2) click on the "Inputs & Values" folder. Click and drag a "Label" object from the library window to your "Hi there!" window. Double-click it and type "Hello World!". That's it we're done with this interface. Now we just have to save it and load it into Lisp.

There is the normal sort of "Save as ..." dialog that you might expect, but we need to talk about the difference between XIB files and NIB files. XIB files are XML files which IB uses to represent this interface. XIB files are nice because you can look at them and see what's there pretty directly and even modify them if you're brave enough. But applications can't directly load XIB files. Instead, they must load a compiled form of them called NIB files. For our purposes we never need XIB files, so I typically save as a NIB file initially and forget about it. In the Save dialog you will see a "File Type" choice. I typically use "NIB 3.x" because I don't have to worry about supporting an older version of OSX. Save it to someplace that you can easily access it or use the one that I provided in my example directory. I suggest that you un-check the box that hides the file extension. That way you won't be confused about what sort of file it is (NIB or XIB) and will know the right name to provide to Lisp.

Navigate to the "Hello World" subdirectory of your InterfaceProjects directory and save the file as "hello.nib". Once you have saved the NIB, the name of the nib document window will change to reflect the saved name.

Start the CCL IDE and follow along with the listener dialog shown below:

```
Welcome to Clozure Common Lisp Version 1.5-dev-r13174M-trunk  (DarwinX8664)!
? (require :nib)
:NIB
("NIB")
? (iu:load-nibfile (truename "ip:Hello World;hello.nib"))
(#<LISP-APPLICATION <LispApplication: 0x1cec80> (#x1CEC80)> #<NS-WINDOW <NSWindow:
0x14220420> (#x14220420)>)
T
?
```

Your "Hello World" window will pop up. Congratulations! you just built your first Lisp Cocoa user interface.

When the NIB file that we created is loaded, all of the user interface objects that we defined using IB are created. In this case the window and the label within it are created and linked together properly so that they display in exactly the same

way that we defined. Also note that you can resize the window, that it will respond to the close menu and otherwise acts just as you would expect of any window. Also note that when this window is selected that any menu items that are NOT relevant are automatically grey-ed out. This is just what you would want without ever having to specify anywhere what are and are not relevant menu items for this window.

Let's look under the hood at the load-nibfile function since that is all that we used for this project. This is my modified version of some example code that CCL provides.

```
;; nib.Lisp
;; Start with some corrected functions from .../ccl/examples/cocoa/nib-loading/HOWTO.html

(defpackage :interface-utilities
  (:nicknames :iu)
  (:export load-nibfile))

(in-package :iu)

;; Note that callers of this function are responsible for retaining top-level objects if
;; they're going to be around for a while and them releasing them when no longer needed.

(defun load-nibfile (nib-path &key (nib-owner #&NSApp) (retain-top-objs nil))
  (let* ((app-zone (#/zone nib-owner))
         (nib-name (ccl::%make-nsstring (namestring nib-path)))
         (objects-array (#/arrayWithCapacity: ns:ns-mutable-array 16))
         (toplevel-objects (list))
         (dict (#/dictionaryWithObjectsAndKeys: ns:ns-mutable-dictionary
                nib-owner #&NSNibOwner
                objects-array #&NSNibTopLevelObjects
                (%null-ptr)))
         (result (#/loadNibFile:externalNameTable:withZone: ns:ns-bundle
                                                            nib-name
                                                            dict
                                                            app-zone)))
    (when result
      (dotimes (i (#/count objects-array))
        (setf toplevel-objects
              (cons (#/objectAtIndex: objects-array i)
                    toplevel-objects)))
      (when retain-top-objs
        (dolist (obj toplevel-objects)
          (#/retain obj))))
    (#/release nib-name)
    ;; Note that both dict and objects-array are temporary (i.e. autoreleased objects)
    ;; so don't need to be released by us
    (values toplevel-objects result)))

(provide :NIB)
```

You can find a good description of the function that this was based upon in CCL's documentation:
"file:///Applications/ccl/examples/cocoa/nib-loading/HOWTO.html"
so I won't reproduce that detail here, but only talk about a few differences. I have provided a couple of &key parameters to this function that were not in CCL's example code. The "nib-owner" key permits you to specify what object will "own" the nib objects that are loaded. You'll learn much more about that in the next project, but for now we just used the default owner which is the NSApplication object that was created for the Lisp IDE. Cocoa applications will have just one such object and the Objective-C variable denoted by #&NSApp contains a reference to it. The "retain-top-objs" key tells the function whether to "retain" the objects. You'll learn more about that in the next project as well. Since we don't care about keeping track of the objects created when the NIB was loaded both of the defaults are perfectly reasonable choices.

Project 2: simplesum

Key Concepts: Lisp defined File's Owner class, Outlets, TextFields, Buttons, Actions and Targets, foreign slots, Lisp action functions

In this project we'll create a fairly simple interface that lets a user enter two numbers and click a button to add them and display the result in the window. You may already know something (and perhaps be confused about) Apple objects like

documents and window controllers. For this project we won't be using any of those. We'll just create the interface graphically and create a single simple Lisp class that owns it. Even with this simple example there are lots of little things to learn about how this all works.

Let's begin as we did for the first project by opening up IB and selecting "New..." from the File menu. Select the Cocoa Window template. To be consistent with the names that I have in my example code, name the new file "sumLisp" in the "Simple Sum" project directory within the "InterfaceProjects" directory. Remember to save it as a "NIB 3.x" file rather than as a XIB file (which is the default).

Recall that when a file is loaded at runtime the application must specify an object that "owns" it. Within IB you will see a *proxy* object called "File's Owner". This object is NOT created by the NIB loading function as other objects are. Instead, the object specified as the owner when the file is loaded is used in its place. What this lets us do is create relationships between interface objects and the proxy File's Owner object within IB that are recreated at NIB loading time between our real File's Owner object and the newly instantiated interface objects. There are various sorts of relationships that can be created and we'll be exploring more of them as we do different projects. One of the simplest sorts of relationships is a simple reference link so that, for example, we can initialize a slot in our File's Owner instance that contains a reference to a particular text box in the interface.

Note: I've discovered that if you are using Interface Builder version 3.2 or later, then the process for adding an object that will be defined in Lisp is somewhat different than it is documented below. As currently shown in this tutorial, the process is to put a new generic object in the document window and then change its class name in the identity inspector before adding outlets and actions via the inspector window. Starting in IB version 3.2 that functionality has been moved to the Library window. There you should select the "Classes" tab and select the class you want to subclass. Then from the action menu select "New Subclass" and enter the name of the Lisp subclass. Choose not to generate source code. Then you can use controls in the library window to add outlets and actions as discussed in the rest of this tutorial.

For this project we will be using a Lisp class that inherits from an Objective-C class as the File's Owner. We'll see later how to define such a class in Lisp. It is easy to tell IB what the class of the "File's Owner " object will be when the NIB is loaded. Click on the "File's Owner" object and then in the inspector window click on the Identity icon (looks like a circle with a small "i" in it) so that information about the object's identity is shown. I called this class "SumWindowOwn" and if you want to follow along you should type in the same name. See Figure 2.1 Below:

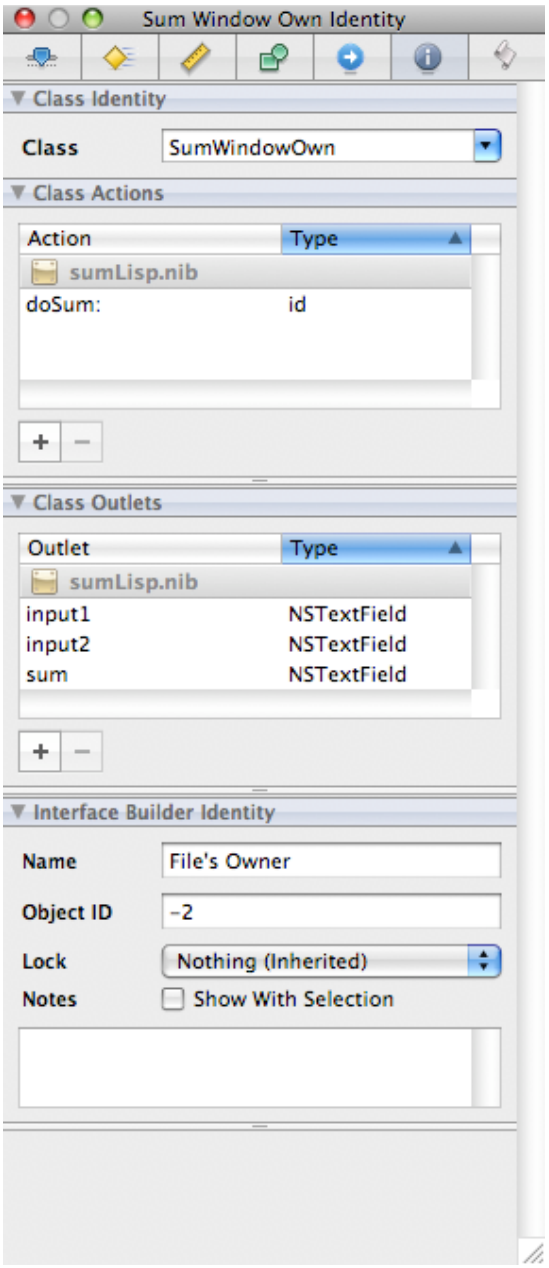

Figure 2.1 SumWindowOwn identity information

Warning: if you later decide that you want to change the name of your owner class (or any Lisp class used in IB), you will lose most of the information that you entered in the inspector window because it now thinks that you just changed the class of the File's Owner. So pick a class name and don't change it.

I suppose this is a good point to talk about naming in Objective-C and the translation of those names to Lisp. Objective-C uses a set of conventions to automatically generate names for various purposes. You will learn those conventions as we go along. Since Objective-C names are case-sensitive and Lisp names are not, some sort of translation is required in order to integrate the two environments. The CCL Objective-C bridge does those translations automatically for you whenever needed, but you need to understand how they will be done so that you can specify names appropriately.

Our first example is the class name that we specified: "SumWindowOwn" (note the capitalization). Lisp will translate that to the case insensitive Lisp name "sum-window-own". You can read the CCL documentation for more detailed information about the translation rules, but gist of it is that except at the beginning of a name, wherever there is a capital letter the Lisp version will insert a hyphen.

Now let's design the interface. As we did before, select your window and give it a title in the inspector window. From the library window select and drag three "Text Fields" and one Button of your choice to your window and arrange them any way that pleases you. Double-click on your button to rename it to "Sum" If you like, select a text field and use the inspector to explore various options you can select for its appearance. When you get done it might look something like the Figure 2.2 below:

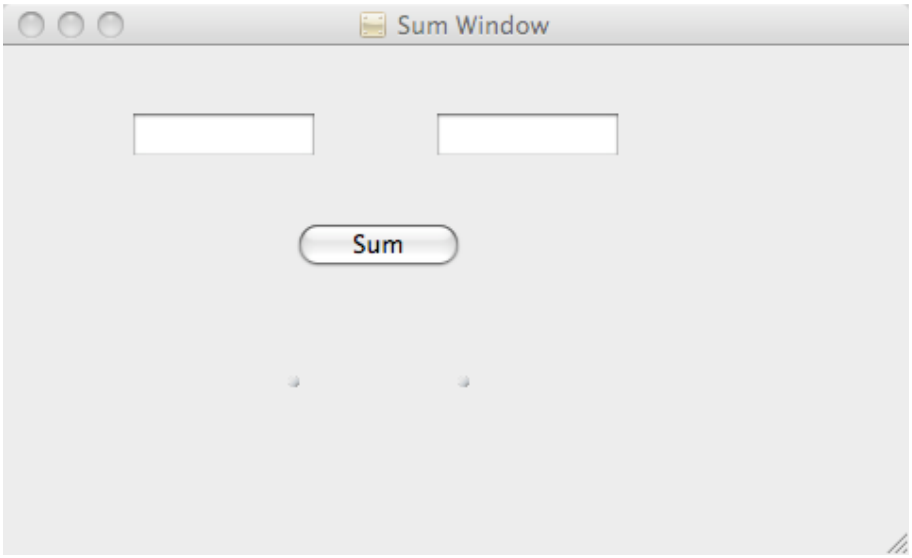

Figure 2.2 Our Sum application window

Note that my third text field is barely visible (only visible at all because I clicked on it) because I used the inspector to make it's border invisible and deselected the "Draws Background" box. I also deselected the "Selectable" and "Editable" boxes so that the user cannot click within it to do anything. When you change an attribute of a view object, IB tries to make it look exactly the way that it will at runtime so that you can easily visualize it. That's true even if it makes your object invisible. Sometimes that can make it hard to find when you need it, so you may want to wait until the design is mostly complete before setting attributes that make something invisible.

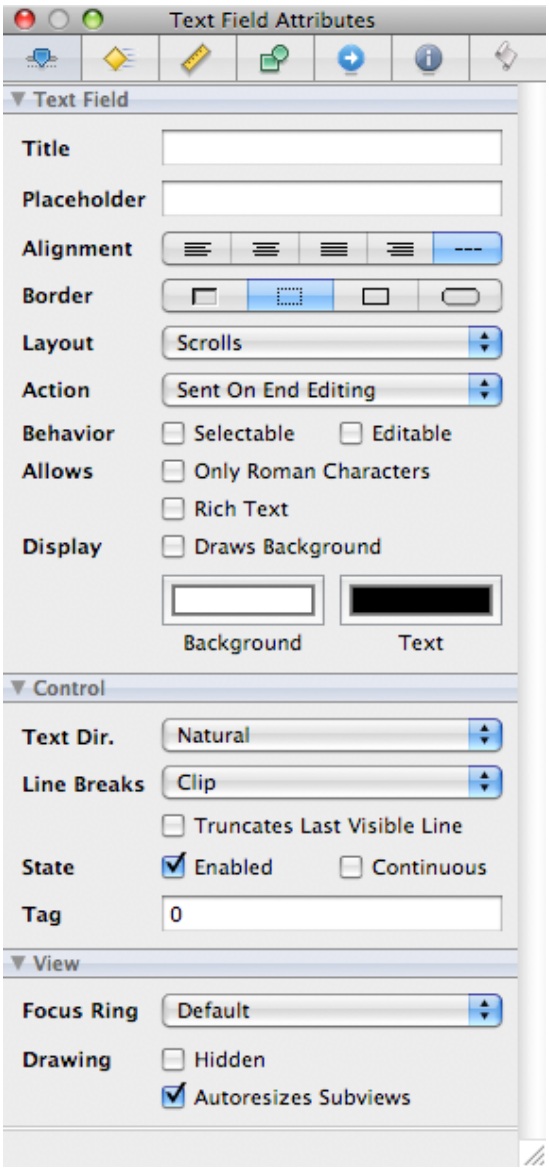

Figure 2.3 Setting attributes for the output text field

Next we'll set up a couple of different relationships between the File's Owner object to these fields. The first type of relationship is the reference link that I mentioned above. Apple has a special name for slots that contain such references; they are called "Outlets". So the idea is that we will specify to IB what outlets the File's Owner class will have and then link those to the objects that we want linked at runtime. Refer back to Figure 2.1.

In the Class Outlets part of the window click the "+" button to add a new outlet. Edit the name to be "input1" and the type to be NSTextField. Note that IB will help with the latter by giving you selectable options after you've typed the first few letters. Do the same for the input2 and sum outlets. Remember that later we will make slots in our Lisp sum-window-own class that correspond to these names. Our objective is that at runtime our input1 slot will contain a reference to the first text field, our input2 slot will contain a reference to the second, and our sum slot will contain a reference to the third.

So far we have just created the outlets, now we need to cause them to be linked to their corresponding text fields. You do that by control-clicking on the "File's Owner" object in the nib document window (Figure 2.4) and dragging to the first edit field in the application window (Figure 2.2). When you release the button IB will let you select which of the File's Owner outlets should be linked to this text box. Select "input1". Now control-click and drag to link the other two text boxes to their respective outlets.

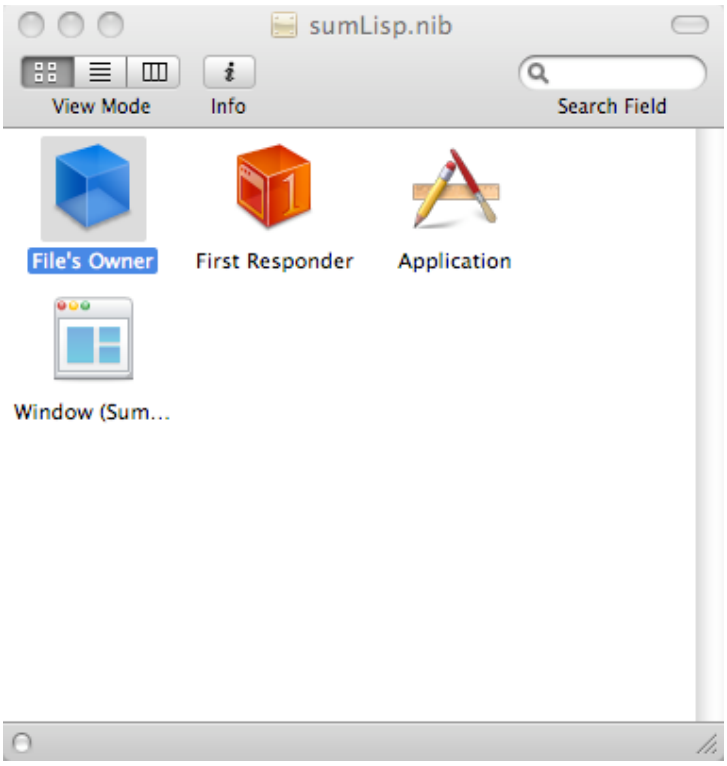

Figure 2.4 Project window for the sumLisp NIB

We need to make one more sort of link and that is so that when the button is clicked something actually happens. In later projects we'll see other ways to accomplish this, but for now it's good to understand just what happens when a button gets pushed. It's basically pretty simple; the button sends an action message that we define to a target that we define. Refer once again to Figure 2.1. The "Action" sub-window is where we tell IB what action messages our class will respond to. Click the "+" button and add a new action. Edit the "MyAction1:" name that is generated and change it to "doSum:". Pay attention to the capitalization and to the use of the ":" at the end of the name. The capitalization is only important insomuch as we need to duplicate it when we define a corresponding method in Lisp. The ":" is required. This is a rule in Objective-C. I don't want to discuss the way that Objective-C functions are named here, but it is important that you understand it before going much further. This discussion can be found in the CCL documentation or any number of books about Cocoa. You can just follow along blindly and you may pick up on what's going on, but feel free to stop and read up on it before proceeding.

Now that we have told IB about our action we need to tell the button what action message to send and where to send it. This is fairly simple. Control-click on the button and drag to our File Owner's object in the nib document window. A dialog will pop up that shows the possible received actions that we might want the button to send. We have only one, namely "doSum:", so select it. This has set the button's target to our File's Owner object and indicated that it should send a "doSum:" message when the button is pushed. When the action message is sent, the button will also pass a pointer to itself as an argument. In this case we don't need that, but in future projects we'll see how this can be helpful.

That's pretty much it on the IB side of things. Save your file and let's go to Lisp. In general I suggest that you just open my source files and follow along with the discussion (simplesum.Lisp in this case), but feel free to type it all in yourself. I find that I sometimes learn things more thoroughly when I do them myself. If you do that, just make sure that you include things that may be in my source (such as "require" statements) that I may not talk specifically about.

At this point, if you have not already done so, I urge you to read section 13 of the CCL documentation:
file:///Applications/ccl/doc/ccl-documentation.html#The-Objective-C-Bridge
Actually the whole document is a pretty good read and contains lots of goodies that you should know about.

Let's start with the class definition for the object that will be our File's Owner when we load the NIB file:

```
(defclass sum-window-owner (ns:ns-object)
  ((input1 :foreign-type :id
           :accessor input1)
   (input2 :foreign-type :id
```

```
          :accessor input2)
   (sum :foreign-type :id
        :accessor sum)
   (nib-objects :accessor nib-objects :initform nil))
  (:metaclass ns:+ns-object))
```

You'll see that the class name is the Lisp version of the name that we specified in the IB inspector window for the File's Owner classname (SumWindowOwner). This class inherits from the root class for all Objective-C objects; namely NSObject. Again, we have used the Lisp version of that name. The :metaclass specification is per CCL's recommendation and you can read their documentation for a discussion of it.

We have four slots and the first three correspond to the outlets that we specified to IB. We have declared them to be a :foreign-type and take a value of :id. This makes them accessible to Objective-C just as if they were a slot defined for some native Objective-C class. The ":id" part of the specification just says that the slot will contain a generic pointer to an Objective-C object of some sort (which you should already know if you really read the CCL documentation!) Note that CCL is handling the name translation for you. We specified the Outlet names in lower case to IB and Lisp effectively upper-cases all names, so even though it may not be readily apparent, there is a name mapping going on here. When the NIB file is loaded using an Objective-C function it will look for a slot in File's Owner named "input1" and will find ours and set the value. And we can access that slot from Lisp in the normal way. Nice!

The fourth slot, nib-objects, is there to contain the top-level objects that are instantiated by the NIB loading function. We'll see how we use that in just a bit. Next let's look at some of the things we do during initialization of this class:

```
(defmethod initialize-instance :after ((self sum-window-owner)
                                       &key &allow-other-keys)
  (setf (nib-objects self)
        (load-nibfile
         (truename "ip:Simple Sum;sumLisp.nib")
         :nib-owner self
         :retain-top-objs t))
  ;; we do the following so that ccl:terminate will be called before we are garbage
  ;; collected and we can release the top-level objects from the NIB that we retained
  ;; when loaded
  (ccl:terminate-when-unreachable self))
```

As in the Hello World project we call the load-nibfile function, but we do a few things differently. First we set the nib-owner to the object we are creating so that it becomes the "File's Owner" object that is linked to the interface objects, just as we specified to IB. We save the list of objects that it returns in our nib-objects slot. As the File's Owner this object is responsible for retaining these top-level objects so that they don't get deallocated at runtime and also releasing them when we're done with them. We told the load-nibfile function to retain them for us by setting the key parameter :retain-to-objs to t. To assure that they are released properly we must ensure that before this object is garbage collected a method is called to do the release. We tell CCL to do that with the call to ccl:terminate-when-unreachable. The function that is called is:

```
(defmethod ccl:terminate ((self sum-window-owner))
  (dolist (top-obj (nib-objects self))
    (unless (eql top-obj (%null-ptr))
      (#/release top-obj))))
```

and this simply calls the Objective-C method #/release on each of the objects that we had previously retained. If you don't do this everything may appear to work properly, but at the least the CCL IDE will terminate abnormally when you quit it. That's usually a sign that there are memory management problems (typically a memory leak) that could bite you, so it's best to do everything properly. In future projects we typically won't discuss this function, but be assured that it is there.

The last method we need to define is the action method that we told the button to call when it was pushed. You'll recall that in IB we set that to "doSum:". It is defined as follows:

```
(objc:defmethod (#/doSum: :void)
                ((self sum-window-owner) (s :id))
  (declare (ignore s))
  (with-slots (input1 input2 sum) self
    (#/setIntValue: sum (+ (#/intValue input1) (#/intValue input2)))))
```

There are a few things to note about this definition. First, there are two different macros that can be used to define

Objective-C methods. There are also two ways to invoke Objective-C functions and I will use the one shown here. If I had to characterize the differences I suppose I would say that one form is more consistent with the way that Objective-C programmers would call the functions and one seems somewhat more like natural Lisp syntax. It is not difficult to translate and after trying both I am simply more comfortable with the more Lisp-like syntax and will use that consistently throughout this tutorial. If you feel more comfortable with the other syntax you should be able to translate what I provide quite easily.

Note that the function defined here is named #/doSum:. The #/ reader macro honors the case of the name that follows and does whatever is necessary to register the name with the Objective-C runtime system. In this way I can be sure of defining functions that have exactly the name that I want. Note the ":" at the end of the name. This is part of every Objective-C function that passes one or more parameters to the target (in addition to the target argument itself). No return value is required so the return value is specified as :void. The method is defined for the sum-window-owner so that is the first argument and a pointer to the sending object is also passed as an argument that we called "s" and defined to be a generic Objective-C pointer (which we don't need and therefore ignore).

The function itself simply sets the value for the interface object that is pointed to by the pointer in our sum slot. How did I know to call "#/setIntValue:"? That is easily found by looking at the instance methods available to NSTextField objects. If you haven't done so yet, this is a good time to start the AppKiDo application and search for NSTextField. Click on it and then click on the "ALL Instance Methods" line to see every possible thing that you could do. Impressive, isn't it?

An astute programmer might be wondering what happens when there isn't anything in those input fields. What will the #/intValue calls return? Initially I put all sorts of additional code here to initialize the fields to 0 if they were blank. But after experimenting a bit I discovered that what you get in that case is 0 so it was all unnecessary. Cocoa does lots of things that make life easy and this is just one small one.

If you are following along in simplesum.lisp you will see the function #/doFirstThing next. Ignore this function for now. It is actually used as the target of a menu-item in project #3 and will be explained when we get to that point.

Finally I typically create a test function for classes that I define and for this one it's pretty simple:

```
(defun test-sum ()
  (make-instance 'sum-window-owner))
```

If you execute this in the REPL by typing (ss:test-sum) your interface will pop up and you can add numbers to your heart's content. Project 2 is complete.

Project 3: menus

Key Concepts: First Responder and responder chains, delegates, menu actions, menu creation

It's actually fairly straight-forward to add a menu object to your IB definition and load it into the CCL IDE. The problem is that it will *replace* the existing menu which is not exactly what we're trying to accomplish. Nevertheless, there are times when we would like to add our own menu to the menubar or modify menus that already exist. That's what we'll explore in this project.

We've already seen that our new windows will automatically respond to existing menu choices whenever they are supported. How exactly does that work? You may have noticed a red box object in your nib document window called "First Responder". This is another proxy object somewhat like File's Owner. But in this case the actual first responder object is dynamically determined while your application is running. All windows and view objects within them are organized as a hierarchical set of containers. Every time the application user clicks on the screen somewhere the lowest-level user interface element in that location is given the opportunity to become the first responder. If it chooses to decline the responsibility then it's superview is given the opportunity and so on up the line until some object accepts. This chain of responsibility called the Responder Chain is dynamically redefined every time a click is made. Why do we care about this? There are many reasons, but in this case it is because the action messages sent by menu items are often targeted at the First Responder proxy. At runtime the actual messages are sent to whichever object is currently the First Responder. If it has a method defined for that action, it performs it. If not, the message is passed up the responder chain until some object does respond to it.

Responder chains also can include *delegate* objects so we need to understand what these are. Many classes, such as NSWindow and others have a delegate outlet. When a delegate-owning object (that is an object with a delegate outlet that contains a pointer to some real object) is passed an action message it will first check to see whether a method corresponding to that message has been implemented by the delegate. If so, it passes the message to it. (i.e. it delegates responsibility for the response). If no such method exists either the object will respond itself or pass the message along to the next object in the responder chain.

You may notice that menu-items which are not relevant for our windows are disabled when our window is active. That is because every menu-item checks to see whether ANY object in the current responder chain can respond to its action message. If none of them can, then it is disabled.

What we're going to do here is dynamically add a menu with sub-menus to the menubar and then set one of our objects to respond to one of those menu items. Go back to IB and open the "sumLisp.nib" file that we used for project #2. Locate the Window object in the nib document window. Control-click on this object and drag to the "File's Owner" object just above it. When the pop-up appears select the "delegate" outlet. What we have now done is make the File's Owner a delegate for the window. If you now control-click on the window object you can see the link that has been made (Figure 3.1).

Figure 3.1 Window outlets showing that File's Owner is the delegate

There is a bit of complexity to adding new menus and sub-menus, but thanks to Apple's sample code (Objective-C of course) it turned out to be not all that difficult. I suppose the key things to understand is that there are NSMenu objects and NSMenuItem objects. NSMenu's may only contain NSMenuItem's never subordinate NSMenu objects. NSMenuItem's may contain subordinate NSMenu objects. So in order to create a new menu, put it into the menubar (which is just the main menu), and have subordinate menu items we must create a new NSMenuItem that we make subordinate to the main NSMenu object, then make a new NSMenu object subordinate to the new NSMenuitem we created, and finally add a series of subordinate NSMenuItem's to our new NSMenu. It might be easier to see what the code is doing than it is to parse that last paragraph!

I created a make-and-install-menu function to make this all a bit simpler:

```
(defun make-and-install-menu (menu-name &rest menu-item-specs)
  (let* ((ns-menu-name (ccl::%make-nsstring menu-name))
         (menuitem (#/initWithTitle:action:keyEquivalent:
                    (#/allocWithZone: ns:ns-menu-item
                                      (#/menuZone ns:ns-menu))
                    ns-menu-name
                    (%null-ptr)
                    #@""))
         (menu (#/initWithTitle: (#/allocWithZone:
                                  ns:ns-menu (#/menuZone ns:ns-menu))
                                 ns-menu-name))
         (main-menu (#/mainMenu #&NSApp)))
    (dolist (mi menu-item-specs)
      (destructuring-bind (mi-title mi-selector &optional (mi-key "") mi-target) mi
        (let* ((ns-title (ccl::%make-nsstring (string mi-title)))
               (action-selector (get-selector (string mi-selector)))
               (ns-key (ccl::%make-nsstring (string mi-key)))
               (men-item (#/addItemWithTitle:action:keyEquivalent: menu
                                                                   ns-title
                                                                   action-selector
                                                                   ns-key)))
          (when mi-target
            (#/setTarget: men-item mi-target))
          (#/release ns-title)
          (#/release ns-key))))
    ;; Link up the new menuitem and new menu
    (#/setSubmenu: menuitem menu)
    (#/release menu)
    ;; Now tell the main menu to make this a sub-menu
```

```
    (#/addItem: main-menu menuitem)
    (#/release ns-menu-name)
    (#/release menuitem)
    menu))
```

A menu-item-spec is a list of the form:
(menu-item-name menu-item-action menu-item-key-equivalent menu-item-target)
where the first three are all strings and the last must be an object that will be the target of the menuitem's action message when it is selected. Normally the target is left nil and the message is sent up the first responder chain, but for some menuitems that we will create later we want it to be sent to a specific target that we will specify.

I also created the function make-and-install-menuitems-after to install new menuitems at a specific location within an existing menu. This will be used in Project #7.

```
(defun make-and-install-menuitems-after (menu-name menu-item-name &rest menu-item-specs)
  (let* ((ns-menu-name (ccl::%make-nsstring menu-name))
         (main-menu (#/mainMenu #&NSApp))
         (menuitem (or (#/itemWithTitle: main-menu ns-menu-name)
                       (error "~s is not a valid menu title" menu-name)))
         (sub-menu (#/submenu menuitem))
         (ns-menu-item-name (ccl::%make-nsstring menu-item-name))
         (insert-index (#/indexOfItemWithTitle: sub-menu ns-menu-item-name)))
    (dolist (mi menu-item-specs)
      (destructuring-bind (mi-title mi-selector &optional (mi-key "") mi-target) mi
        (let* ((ns-title (ccl::%make-nsstring (string mi-title)))
               (action-selector (get-selector (string mi-selector)))
               (ns-key (ccl::%make-nsstring (string mi-key)))
               (men-item (#/insertItemWithTitle:action:keyEquivalent:atIndex:
                          sub-menu
                          ns-title
                          action-selector
                          ns-key
                          (incf insert-index))))
          (when mi-target
            (#/setTarget: men-item mi-target))
          (#/release ns-title)
          (#/release ns-key))))
    (#/release ns-menu-item-name)
    (#/release ns-menu-name)))
```

If you have all the paths to my code set up as previously described then from the listener you can type:

```
? (require :menu-utilities)
```

and this will be loaded. To demonstrate how to call this function you can use the following test-menu function:

```
(defun test-menu ()
  (make-and-install-menu "New App Menu"
                         '("Menu Item1" "doFirstThing")
                         '("Menu Item2" "doSecondThing")))
```

When you execute this in the listener by typing (iu:test-menu) a new menu titled "New App Menu" will be added to the menubar with two sub-items titled "Menu Item1" and "Menu Item2". If you click on the menu you will see that both menu items are disabled. That is because no object in the responder chain knows anyting about the action messages "doFirstThing" or "doSecondThing".

Recall that we previously linked our the simplesum File's Owner object as the delegate for the Sum Window object. What that means is our Lisp File's Owner object is now in the responder chain for any click within our window. So let's go back to our Lisp code and add a method that will respond to the action method "doFirstThing" that we defined for the first menu item:

```
(objc:defmethod (#/doFirstThing :void)
                ((self sum-window-owner) (s :id))
  (declare (ignore s))
  ;;; test function for menu tests
```

```
  (#/doSum: self (%null-ptr)))
```

This simple function just calls the same action function that is invoked by clicking on the "Sum" button, namely it adds the two input values and displays the new sum. Once this function is defined you will see that if you first click in a Sum window and then click on the "New App Menu" that "Menu Item1" is now enabled and that if you select it the display will act just as if you had clicked on the Sum button.

There is also a second test function:

```
(defun test2-menu ()
  (make-and-install-menuitems-after "File" "New"
                                    '("New myDoc" "newMyDoc")))
```

which installs a "New myDoc" menuitem immediately after the "New" menuitem within the "File" menu. You can experiment with adding key equivalents and specific targets as well although you will also do some of that in a later project.

These are obviously pretty simple examples, but they demonstrate what is possible. For the ambitious developer I would refer you to Apple's example xcode project called "MenuMadness". It pretty thoroughly shows almost every sort of menu option and possibility that anyone could ever want. I think that you'll find that the functions there are readily ported to Lisp.

Project 4: Speech

Key Concepts: Speech controller, Radio Buttons, Memory management, runtime view modification

At this point the use of IB should be somewhat more intuitive, so this fun little project should be a piece of cake. Basically it demonstrates how to configure and use radio buttons and gives a small taste of how views can be modified at runtime.

Start a new window NIB in IB or open my nib file: ...InterfaceProjects/Speech/SpeechView.nib. Define an interface that has a NSTextView field where we will type in things to be said by the speech synthesizer. Add a couple of buttons to start and stop speaking and label them as you wish. Add a "Radio Group" that you will find in the "Inputs & Values" folder in the Library window. Also add a label above the radio group and change it to say "Voice" (or whatever you'd like it to say). Initially the radio group will have two buttons, but we need 24 to accommodate all of the possible voices of Apple's speech synthesizer (You can discover that quite easily for yourself. Go into CCl and in the listener type:

```
? (#/count (#/availableVoices ns:ns-speech-synthesizer))
24
```

If yours says something different you may need a different number of radio buttons and you should modify this project appropriately.

So we want to change the Radio Group to have 24 buttons and to be arranged nicely. 24 is a nice number because we can create a nice four-column six-row rectangular array. To do that click on the button group to select it. You need to be a bit careful here because this object is actually an arrangement of nested views and it's easy to select something other than what you want. It took me a while to figure out how IB decides what you are trying to select, but it's really quite a simple rule. The first time you click IB will select the outermost container at that point. If you click again within that region it selects the next most deeply embedded view object. Click again and you will get something even more deeply embedded if it exists. If you accidentally get too deep, just click outside the object somewhere and then start over.

In this case, we want to select the "Matrix" object that contains all of the radio buttons so click accordingly and make sure that the title in the inspector window reflects the fact that you have selected the matrix. In that window click on the "Attributes" icon (far left) and change the number of rows and columns to suit your needs. You can drag and resize the matrix in the design window to change its visual characteristics. If you click on the "Size" icon in the inspector window you can change many things including how the matrix reacts to dynamic window-resizing. Play with the options a bit to see how this works and pick something that suits your personal sense of esthetics.

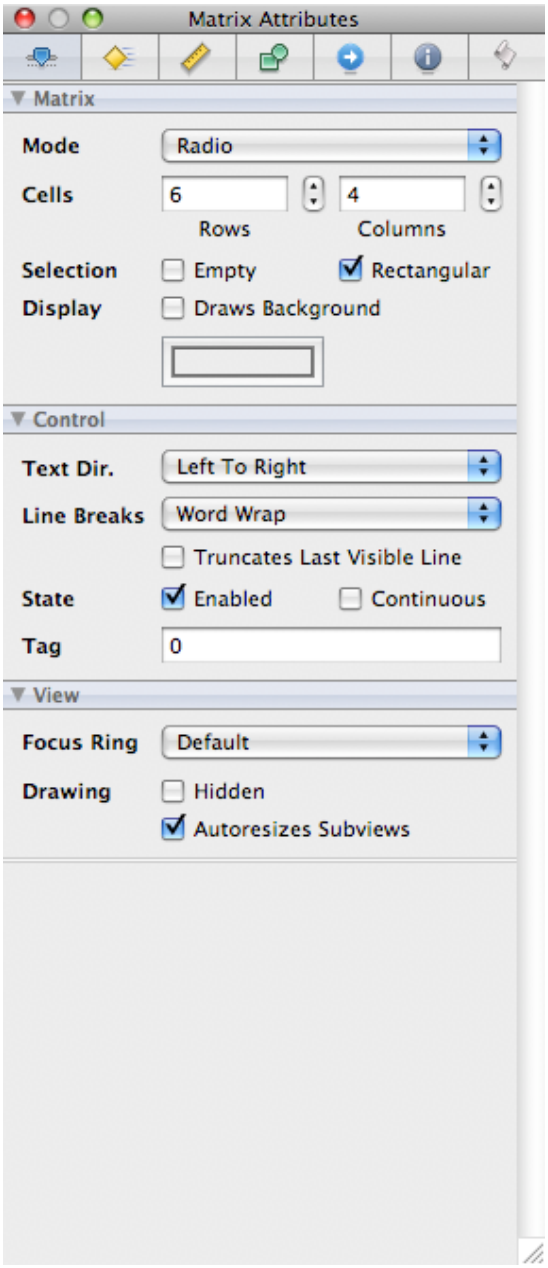

Figure 4.1 Radio Button Matrix attributes

When you get done you newly designed window might look something like mine (Figure 4.2):

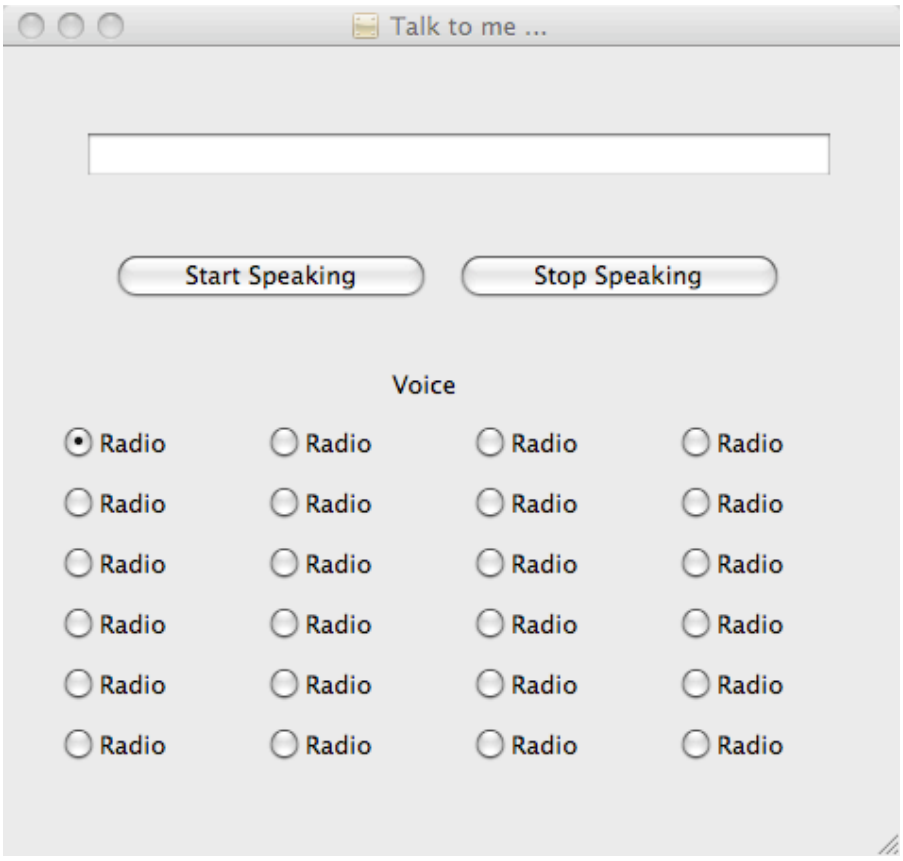

Figure 4.2 Speech window designed in IB

What about the labels on the radio buttons? Well, we certainly could go ahead and put labels on them within IB, but that would require us to figure out how to map them to the real voices in our Lisp code. It's easier to set those titles from within Lisp in a way that makes it easy to do this.

Next we need to specify, as we have done previously, what sort of object will be the File's Owner and what relationships it will have with other interface objects. Click on the File's Owner object in the nib document window and on the identity icon in the inspector window. Change the class name to "SpeechController" and add the actions and outlets as shown in Figure 4.3.

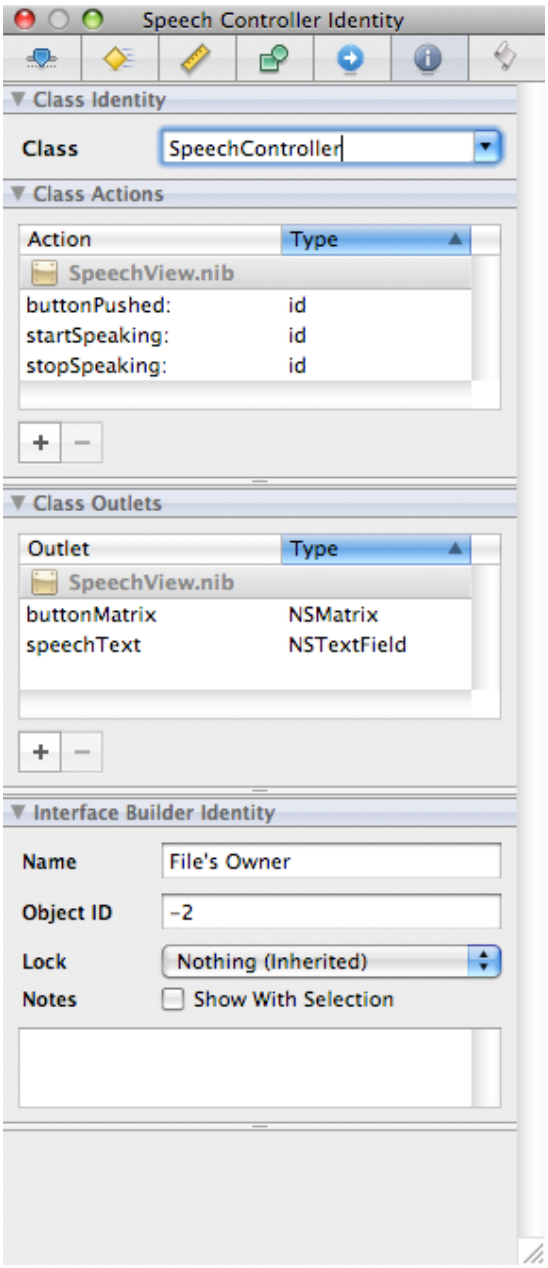

Figure 4.3 SpeechController Identity Inspector

Next we'll want to link our File's Owner object appropriately. Control-click and drag from your "Start Speaking" button to the File's Owner object in the nib document window. Select the startSpeaking action from the window that pops up. Similarly link the "StopSpeaking" button. Finally do the same for the Radio Button Matrix and link it to the "buttonPushed" action. To make sure that the outlet slots are set up in our real File's Owner object when the NIB is loaded we need to link them here. So control-click and drag from the File's Owner object to the Radio Button Matrix. In the pop-up window select the buttonMatrix outlet. Similarly link from the File's Owner to the text box where the user will type what they want said to populate the "speechText" outlet. When you get done you can control-click on the File's Owner object to see the links or click on the "Connections" icon in the inspector window to see all the connections you have made. It should look something like Figure 4.4 below:

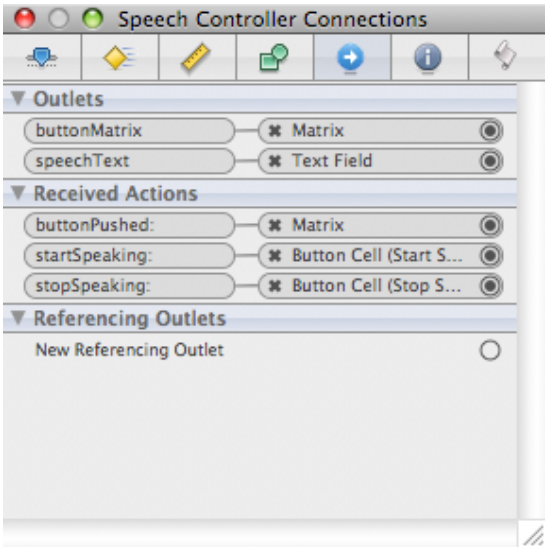

Figure 4.4 Inspector Connections Window

That's all we need for this project, so save the NIB file and we'll work on the Lisp code in CCL. You can open up my example file:

...InterfaceProjects/Speech/speech-controller.Lisp

or create your own if you like. As before I'm only going to discuss new concepts, so if you do start with your own you may want to compare with mine when you're done to make sure you haven't overlooked something. Let's start with the class definition:

```
(defclass speech-controller (ns:ns-object)
  ((speech-text :foreign-type :id :accessor speech-text)
   (button-matrix :foreign-type :id :accessor button-matrix)
   (speech-synth :accessor speech-synth
                 :initform (make-instance ns:ns-speech-synthesizer))
   (voices :accessor voices
           :initform (#/retain (#/availableVoices ns:ns-speech-synthesizer)))
   (nib-objects :accessor nib-objects :initform nil))
  (:metaclass ns:+ns-object))
```

As before, we translate the classname that we gave to IB from SpeechController to speech-controller. We created :foreign slots that are equivalent to the outlets that we defined in IB, again with appropriate name translation. In addition we create a normal Lisp slot to contain a pointer to our speech synthesizer instance and one to contain a pointer to an Objective-C array that contains all of the possible voices for the synthesizer. Memory management is often confusing so I think it's worth a short discussion about why the #/retain call was used for the voices slot, but not for the speech-synth slot initialization. The difference is that when you call make-instance the resulting object is already retained as part of the creation process. You still own it and must release it when done, but you don't have to retain it a second time. Conversely, functions that return things like arrays typically don't retain the objects they are returning. They leave it up to the caller to decide whether to do that or not. Instead they will #/autorelease the object before returning it to you. What this does is tell the Objective-C runtime to release the object sometime later (typically during the next event loop). So it's safe to use for a while, but if you want to keep it around long-term, you must #/retain it yourself to avoid having it disappear suddenly. In future projects you'll see a few instances where our Lisp-defined functions that are callable from Objective-C will #/autorelease the object that they return for exactly the same reason.

```
(defmethod initialize-instance :after ((self speech-controller)
                                       &key &allow-other-keys)
  (setf (nib-objects self)
        (load-nibfile
         (truename "myLisp:InterfaceProjects;Speech;SpeechView.nib")
         :nib-owner self
         :retain-top-objs t))
  ;; get all of the voice strings and set the names of the radio buttons
  (dotimes (i (#/count (voices self)))
    (multiple-value-bind (col row) (floor i 6)
      (#/setTitle: (#/cellAtRow:column: (button-matrix self) row col)
```

```
                  (#/objectForKey:
                   (#/attributesForVoice: ns:ns-speech-synthesizer
                                          (#/objectAtIndex: (voices self) i))
                   #&NSVoiceName))))
  ;; Make sure that the initial voice selected for the speech syntesizer matches
  ;; the radio button that is selected at startup. To do that we'll just call our
  ;; own buttonPushed: method.
  (#/buttonPushed: self (button-matrix self))
  ;; we do the following so that ccl:terminate will be called before we are
  ;; garbage collected and we can release the top-level objects from the NIB
  ;; that we retained when loaded
  (ccl:terminate-when-unreachable self))
```

The initialize-instance :after method is similar to previous ones with the exception that we also initialize all of the titles of those radio buttons that we defined in IB. We create a straightforward mapping from the linear voice index to the two-dimensional array of radio buttons. We go down each column and then across rows. One of the design choices we could have made would have been to translate all those voices to Lisp strings and save them that way rather than keeping around a pointer to an NSMutableArray that contains NSStrings. The reasons that I chose not to do that were that Lisp no longer needed those strings so there was no point in converting them to Lisp strings and we continued to need them when a radio button was pushed to set the new voice. I also could have used a Lisp array to contain those NSStrings, but then I would have had to worry about appropriate #/retain and #/release calls that are handled automatically by NSMutableArray. Deciding how to represent data is one of those ongoing tasks that must be done.

```
(defmethod ccl:terminate ((self speech-controller))
  (when (speech-synth self)
    (#/release (speech-synth self)))
  (when (voices self)
    (#/release (voices self)))
  (dolist (top-obj (nib-objects self))
    (unless (eql top-obj (%null-ptr))
      (#/release top-obj))))
```

Note that the ccl:terminate method calls #/release for both the speech-synth and voices objects.

```
(objc:defmethod (#/startSpeaking: :void)
                ((self speech-controller) (s :id))
  (declare (ignore s))
  (with-slots (speech-text speech-synth) self
    (let ((stxt (#/stringValue speech-text)))
      (when (zerop (#/length stxt))
        (setf stxt #@"I have nothing to say"))
      (#/startSpeakingString: speech-synth stxt))))
```

In our #/startSpeaking method we just get the string from the text box and give it to the speech synthesizer (unless it's blank in which case we provide a bit of humor). Note the use of the #@ reader macro. This creates a Objective-C string that is a constant. So we don't have to #/retain or #/release or #/autorelease it. It will be around for the duration of our program and can be re-used as many times as needed.

```
(objc:defmethod (#/stopSpeaking: :void)
                ((self speech-controller) (s :id))
  (declare (ignore s))
  (with-slots (speech-synth) self
    (#/stopSpeaking speech-synth)))
```

In this method we tell the synthesizer to stop speaking. In that way the user can abort ongoing speaking.

```
(objc:defmethod (#/buttonPushed: :void)
                ((self speech-controller) (button-matrix :id))
  (let ((row (#/selectedRow button-matrix))
        (col (#/selectedColumn button-matrix)))
    (#/setVoice: (speech-synth self)
                 (#/objectAtIndex: (voices self) (+ row (* col 6))))))
```

This method is called when a radio button is pushed. We just locate the button within the matrix and convert that row and column into an offset into the voices array. We then set the voice for the speech synthesizer accordingly.

As before we define a test function and after you eval'ed all the code you can run it:

```
(defun test-speech ()
  (make-instance 'speech-controller))
```

Type (spc:test-speech) in the listener to run it.

Challenges:

If you play around with this program a bit you may notice a bug. If you push a radio button while the synthesizer is talking the new voice will not be set properly. Clearly you can't do two things simultaneously with the synthesizer, so the #/setVoice call is ignored. How do we fix this? The right way is to ensure that it doesn't happen in the first place by disabling the radio buttons while the synthesizer is talking and then re-enabling them when it's done. After project #6 you'll know more about how to enable and disable controls, but feel free to work it out on your own.

You've probably also noticed that using radio buttons to list the names of all the voices isn't a particularly good design. If the number of voices is changed at some future time then the interface is broken and must be fixed. A much better alternative might be to have a scrollable list of voices from which to pick. After project #5 you'll know something about how to create such a view and select from it. Feel free to come back and modify this project.

If you really like the idea of radio buttons, then there is another alternative that could be tried. Here I'll admit that when I started this project I intended to do something a little more ambitious. Rather than just changing button titles, I intended to dynamically define how many radio buttons should be made and where to put them within a space that I defined within IB. See Apple's demonstration code called "ButtonMadness" for an example of this technique. I'll confess laziness and a desire to get on to other sorts of interfaces that were of more interest to me. But if you'd like to do something like this I believe it will provide both a good challenge and a good example of the sort of runtime alteration of the interface that is possible.

Project 5: PackageView

Key Concepts: TableViews, Lisp Data Sources. Accessor functions accessible from Objective-C objects, window controllers

So far the projects that we've done have simply used user input to do some easy things. In this project we'll be creating an interface to show data that originates in the Lisp system; namely information about packages that have been created and what other packages either use them or are used by them. You'll learn how easy it is create Lisp functions that provide data for NSTableViews that are part of our display window. We've previously seen how you can create :foreign slots that are directly accessible from Objective-C. For this project you'll see how to create accessor functions that create something like a *virtual* slot, which gives us an enormous amount of flexibility in how we provide data to interface objects.

Let's start with the interface design. Open IB and either create a new window project or load my example NIB:
...InterfaceProjects/PackageView/packageview.nib
If you created your own, then save it as packageview.nib as I did.

Click on the design window and in the attributes view of the inspector window rename it as: "Lisp Packages" (or something you like better). We'll be adding three "Table Views" to the window which you can find by selecting the "Data Views" folder in the library window. Click and drag three of them to the Lisp Packages window and use the handles on those objects to resize and arrange them roughly as shown in Figure 5.1 below. The first window will have two columns and the other two will have one. To set the number of columns click on the view and ...

Oops, be careful here. When you click on the table view note that the inspector says that you've selected a scroll view. That's because a Table View is actually composed of a moderately complex hierarchy objects that have been design to work together seamlessly. Click multiple times in various spots within the Table View and you'll see these different objects reflected in the inspector window's title. Cocoa defines a large number of reusable object classes. IB prototypes that we include in our interface are often composed of several of these classes combined into a useful pattern. This re-use of primitive classes makes for a very consistent look and feel. It also facilitates easy maintenance because bugs need only be fixed in one place rather than in many. Once this is recognized it becomes easy to see why we don't really want to have to set up all of these relationships programmatically. Cocoa provides ways to do that if you really want to, but personally I prefer to drag and drop a prototype interface element that is composed of a set of fully debugged objects.

OK, so what you have to do to edit the number of columns in a Table View is to click twice on it so that the inspector window shows that you're looking at a "Table View". The attributes view within the inspector window will allow you to set the number of columns that you want. To change the column titles you should double-click on each column header and edit its title directly there.

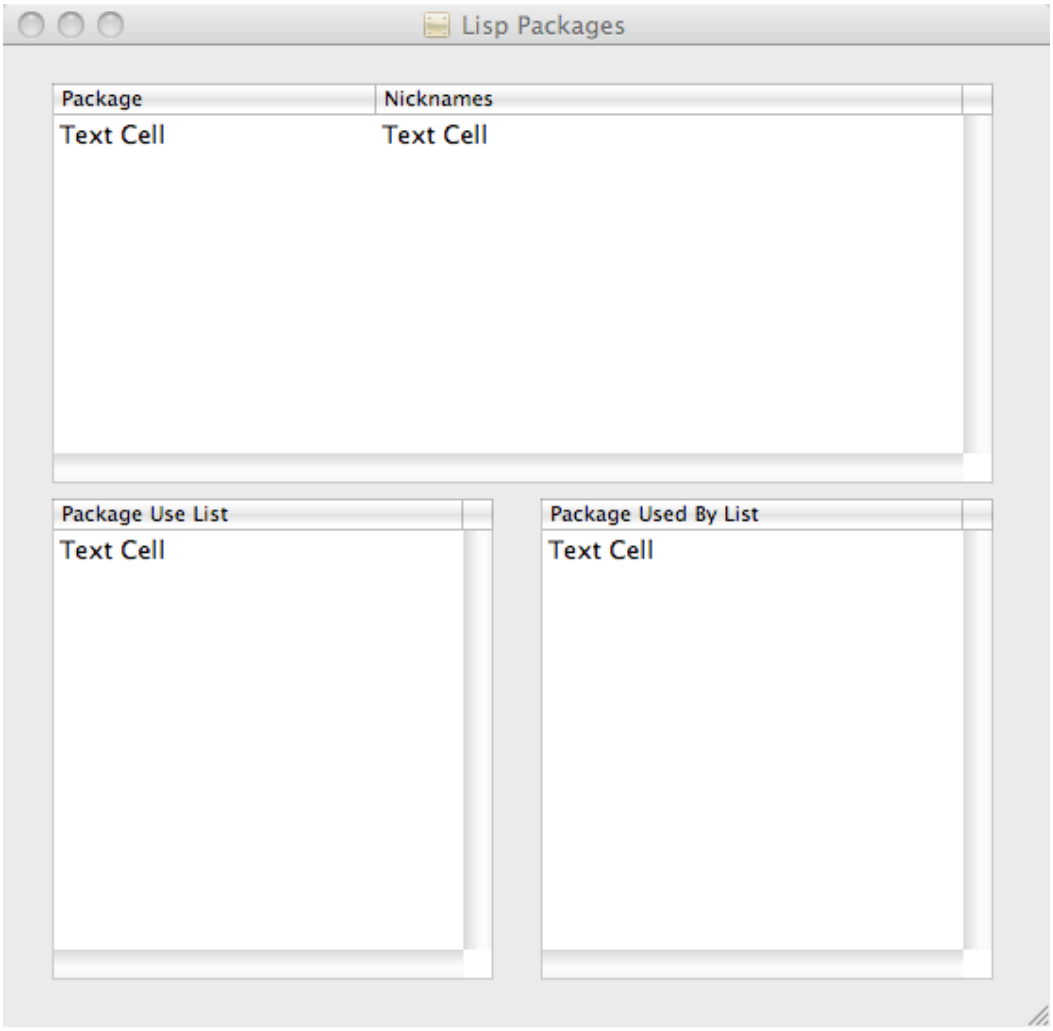

Figure 5.1 Window design for the PackageView project

At this point your window should look pretty much as shown in Figure 5.1 above. Now we need to set up the relationships that will allow each table view to retrieve the data that it will display. It's important that you re-read and understand that last sentence. In the Cocoa paradigm display objects that are not used for user input generally retrieve their data automatically. It's not typically the case that a function executing somewhere tells the display how to change. Instead, if a program knows that the data displayed in some view is no longer valid it will tell the view to reload its data. This permits very fast updates. For example if you are displaying a very large table where most of it is currently invisible due to the position of the scrollbar, then the Table View object is smart enough to only reload the visible portions of its data. This also relieves the application code from having to be aware of how the scrollbar is set at each moment in time (although as we'll see in the next project it can certainly make itself aware if that's desirable).

Now we need to make connections so that our Lisp File's Owner class will know how to locate the Table Views in our window (to tell them to reload occasionally) and so that the Table Views know where to go to get their data. Let's start with the File's Owner object. In the Inspector identity view change the name of the class to "PackageViewController". Remember that this must be the first thing you do and don't change it unless you want to re-enter everything else later.

Now add three outlets and title them as shown in Figure 5.2 below:

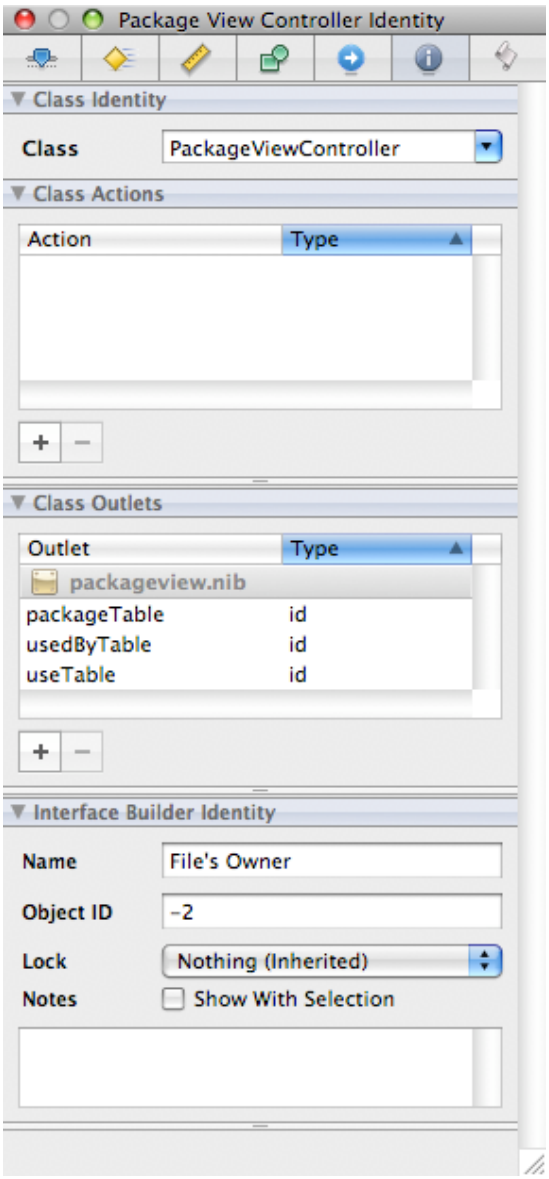

Figure 5.2 PackageViewController Identity

In previous projects we did a control-click and drag to set up a relationship between two objects. Here we'll do an entirely equivalent thing using a process that can sometimes be faster when you have lots of things to connect up to each other.

First click twice on the first Table View (so that the inspector shows that you have selected the Table View rather than the Scroll View). Now control-click on that field. A pop-up window will show you all of the outlets for the scroll view. it will look like Figure 5.3 below:

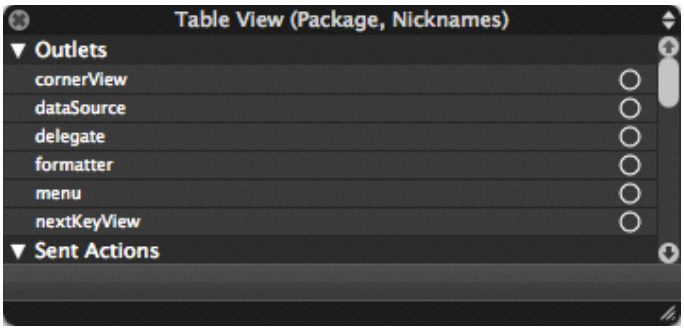

Figure 5.3 Pop-up outlets for the first Table View

If you move your cursor over any of the little circles corresponding to an outlet a "+" sign will appear. To link that outlet simply click within the circle and drag to the object you want to link it with. We'll link both the delegate and data source outlets for each Table View to the File's Owner object. Go ahead and make all of those links.

Next we will make the File's Owner object the window's delegate. Either control-click on the window and then click-and-drag from the delegate outlet to the File's Owner or directly control-click and drag from the window to the File's Owner and then select the delegate outlet. The methods are equivalent.

Now we'll create links from the File's Owner object. Control click on it to bring up a window that shows all of its outlets. Link the three outlets to their corresponding Table View objects. When you get all done the pop-up display for the File's Owner should look like Figure 5.4. Note that this also shows the outlets in other object that reference the File's Owner. Since everything we have done creates links to or from the File's Owner we can verify that we have done everything correctly all in this one place. It should look like Figure 5.4 below:

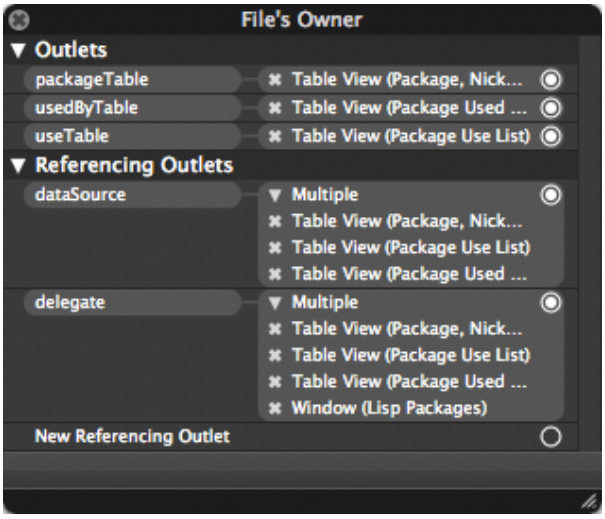

Figure 5.4 File's Owner Outlet pop-up

Note that this is pretty much identical to what you would see in the Connections view of the Inspector window when looking at the File's Owner object, so there are multiple ways to see the same data.

We're going to do one last thing in IB before we move to the Lisp code and that is to set up column ids for the two columns in the top package table. We do this because when the table asks our code for data it passes in a column object rather than just a column number. This done because tables have the capability to re-order their columns, so a simple number just doesn't suffice. So to know which of the two columns is asking for data, we need to set up an identifier that our Lisp code can query when it gets the request.

To do this click on the top Table View, once to select the Scrollable View, a second time to select the Table View, and a third time somewhere inside the first column but below where the words "Text Cell" are displayed. You have now selected the Table Column object. In the Attributes view of the inspector window you can now set the "identifier" string. We will just set it to "1". This will look as follows:

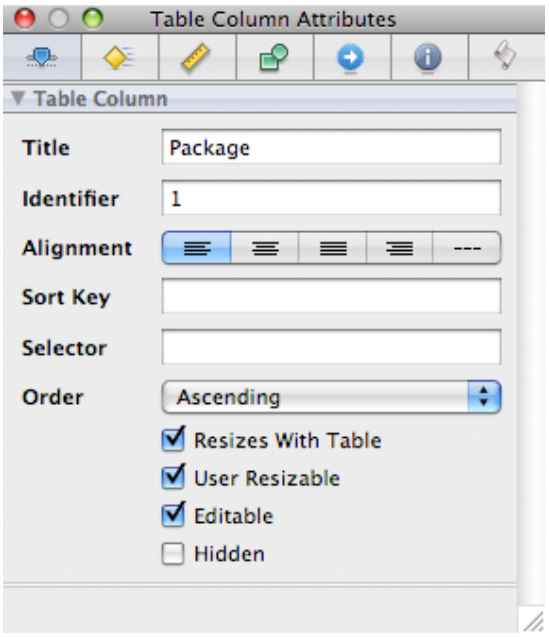

Figure 5.5 Setting the Table Column Identifier to 1

Similarly, set the identifier for the Nicknames column to 2.

Save the NIB file and let's go create some Lisp code to work with this interface.

As before you can open up my sample code and follow along or make your own. The source file for this project is:
"...InterfaceProjects/PackageView/package-view.Lisp"

The class definition is straight-forward. We have slots that correspond to the outlets that we created in IB and slots to contain the data that will be displayed in the Table Views. All of the latter are normal Lisp slots that will contain normal Lisp data. But instead of having a slot to keep track of the top-level objects from the NIB file, we have a slot to keep track of a NSWindowController object. Note that we could easily continue to track the NIB objects as we have in previous projects, but we are setting the stage for more complex relationships that will come in future projects. For this project we will use a generic NSWindowController to load the NIB for us and to keep track of the objects loaded. When the window is closed the NSWindowController will take care of releasing them. Of course, since we created the NSWindowController, we will have to take care of releasing it when we are garbage-collected.

```
(defclass package-view-controller (ns:ns-object)
  ((package-table :foreign-type :id :accessor package-table)
   (use-table :foreign-type :id :accessor use-table)
   (used-by-table :foreign-type :id :accessor used-by-table)
   (current-package :accessor current-package :initform nil)
   (current-nicknames :accessor current-nicknames :initform nil)
   (all-packages :accessor all-packages :initform nil)
   (current-package-use-list :accessor current-package-use-list :initform nil)
   (current-package-used-by-list :accessor current-package-used-by-list :initform nil)
   (window-controller :accessor window-controller :initform nil))
  (:metaclass ns:+ns-object))
```

The initialize-instance :after method sets up the list of all packages. Note that we really wouldn't have to save it here, but doing so prevents having to call the list-all-packages over and over again. And creating an array to hold it means that we can access it faster when responding to a request from a Table View for that data.

Normally window controllers are made to be the owner of the windows that they load, but in this case we specify that the package-view-controller entity is the owner. That way, all the links that we set up in IB to the File's Owner object will be references to the package-view-controller and not to the ns-window-controller.

There are many ways to skin a cat and we could just as easily have made package-view-controller be a subclass of NSWindowController. Arguably for this simple example that might be the way to arrange things. The lesson I'm beginning to teach here is about the normal division of responsibilities that Apple suggests when using Cocoa for more

complex interfaces and more complex data. They promote a Model/View/Controller (MVC) paradigm where Models are data models, Views are user interfaces, and Controllers map back and forth between the two. For very simple applications such as we have seen so far it is typically possible to combine the Model and Controller functionality into a single object class. When we get to the Document architecture in project #7 and beyond you will see a more distinct division of labor between models and controllers. If we want to present multiple views for a given set of data (perhaps with multiple windows), then this division becomes almost mandatory. As the data itself becomes even more complex, there may be reasons for creating multiple ways to organize it (confusingly these are referred to "views" in the database literature), thus adding another layer to the abstraction.

```
(defmethod initialize-instance :after ((self package-view-controller)
                                       &key &allow-other-keys)
  (let ((pkgs (list-all-packages)))
    (setf (all-packages self) (make-array (list (list-length pkgs))
                                          :initial-contents pkgs)))
  (let ((nib-name (ccl::%make-nsstring
                   (namestring (truename "ip:PackageView;packageview.nib")))))
    (setf (window-controller self)
          (make-instance ns:ns-window-controller
            :with-window-nib-path nib-name
            :owner self))
    ;; Now make the controller load the nib file and make the window visible
    (#/window (window-controller self))
    (#/release nib-name))
  ;; we do the following so that ccl:terminate will be called before we are garbage
  ;; collected and we can release the window-controller that we created
  (ccl:terminate-when-unreachable self))
```

The ccl:terminate function is similar to those we have seen previously, but only release the window controller.

```
(defmethod ccl:terminate ((self package-view-controller))
  (#/release (window-controller self)))
```

In previous projects we just let the nib loading function find and initialize our slots directly. We didn't have to define a function to do so. But here we are going to do things a bit differently. When the nib-loading function wants to set one of our slots it will first look for an appropriately named accessor function to do the job. The naming convention is critical and must be adhered to if we want the Objective-C runtime to locate our function. If the slot is named "mySlot" in IB, then the nib-loading function will look for an accessor function named "setMySlot:" All of the capitalization is critical. A read accessor for that slot must be named "mySlot" although for this project we won't need a read accessor.

So why do we need a slot-write-accessor for this project. It turns out that sometimes Table View objects can be initialized and begin to make requests to their data sources before all of the links specified in the NIB file have been instantiated (I found this out the hard way). So our methods that respond to those requests have to do some something benign. In our case we will just tell the Table View that it doesn't have any data to display. That's all well and good, but then we have to correct that after the links ARE created by telling the Table View that it needs to reload its data. That is, we want a side-effect to setting the link. So we'll create methods to have that effect:

```
(objc:defmethod (#/setPackageTable: :void)
                ((self package-view-controller) (tab :id))
  (setf (package-table self) tab)
  ;; Table may already have initialized before this link was set. Tell it to reload
  ;; just in case.
  (#/reloadData tab))

(objc:defmethod (#/setUseTable: :void)
                ((self package-view-controller) (tab :id))
  (setf (use-table self) tab)
  ;; Table may already have initialized before this link was set. Tell it to reload
  ;; just in case
  (#/reloadData tab))
```

When we set the dataSource outlet in each Table View to point to our File's Owner object we were, in effect, promising that this object would conform to a data source protocol that Cocoa defines: NSTableDataSource. If you look at the documentation for this protocol (AppKiDo works well for this) you will see all of the required and optional methods that must be implemented. Since we are not allowing columns to be sorted and the user cannot edit the columns, this becomes a pretty simple protocol. We must respond to a request for the number of rows in the table and another for the

value to be displayed in a particular row and column of the table.

Let's first talk about the request for the number of rows. Our method is as follows:

```
(objc:defmethod (#/numberOfRowsInTableView: #>NSInteger)
                ((self package-view-controller) (tab :id))
  (cond ((eql tab (package-table self))
         (array-dimension (all-packages self) 0))
        ((eql tab (use-table self))
         (if (current-package-use-list self)
           (array-dimension (current-package-use-list self) 0)
           0))
        ((eql tab (used-by-table self))
         (if (current-package-used-by-list self)
           (array-dimension (current-package-used-by-list self) 0)
           0))
        (t
         ;; We can get called before the links are initialized. If so, return 0
         0)))
```

This method will be called for each of the tables so we need to determine which table called us and respond appropriately. Making this determination is one of the two reasons why we needed those links to the Table View objects in the first place. If this had been the only reason then we might have elected to do things a bit differently. We could, for example, have set a tag for each table in IB and then queried the object that called this function for its tag to find out which one it was. But since we occasionally want to tell a specific table when to reload, we needed the link anyway and it constitutes an easy identifier.

The second function we need to create is the "tableView:objectValueForTableColumn:row:". Each of the colons in the method name corresponds to another parameter that we need to accept (in addition to the first target parameter). Note that the row parameter returns an Objective-C NSInteger object, but that we use it later just as if it was a Lisp number. The type translation is done for us automatically.

```
(objc:defmethod (#/tableView:objectValueForTableColumn:row: :id)
                ((self package-view-controller)
                 (tab :id)
                 (col :id)
                 (row #>NSInteger))
  (let ((ret-str nil))
    (cond ((eql tab (package-table self))
           (let ((col-id (ccl::Lisp-string-from-nsstring (#/identifier col))))
             (setf ret-str (ccl::%make-nsstring
                            (if (string= col-id "1")
                              (package-name (svref (all-packages self) row))
                              (format nil
                                      "~{~a~^,~}"
                                      (package-nicknames
                                       (svref (all-packages self) row)))))))))
          ((eql tab (use-table self))
           (setf ret-str (ccl::%make-nsstring
                          (if (current-package-use-list self)
                            (package-name (svref (current-package-use-list self) row))
                            ""))))
          ((eql tab (used-by-table self))
           (setf ret-str (ccl::%make-nsstring
                          (if (current-package-used-by-list self)
                            (package-name (svref (current-package-used-by-list self) row))
                            ""))))
          (t
           (error "~s is not a linked view (~s, ~s, or ~s)"
                  tab
                  (package-table self)
                  (use-table self)
                  (used-by-table self))))
    (#/autorelease ret-str)
    ret-str))
```

This method will return an NSString object that the table can use to set the value to be displayed. So we need to create one. But now we need to think about memory management. We can't release it before we return it obviously because it might be reclaimed before the Table gets it. So instead we "autorelease" it, which basically marks the object to be released sometime later. This gives the Table that made this call a chance to do whatever it wants with the NSString. Whether that object retains it itself or copies it or whatever is irrelevant to us; we have done our part to make sure that the reference count for it is correct.

Other things to note in this function are that we don't have to worry about being asked for an invalid array reference because we already told the table how many rows it had. Also, we use a standard CL format function to create a string with all the class nicknames.

You will also recall that we made our FIle's Owner object the delegate for each of the tables. We only care about one of the delegate methods: "tableViewSelectionDidChange:". This is called whenever the user clicks on an object in the table. When the user clicks in the top table we want the other two tables to immediately reflect that choice. So we use this method to find out when that happens, to find out what was selected, and to update the other two tables and tell them to reload themselves.

```
(objc:defmethod (#/tableViewSelectionDidChange: :void)
                ((self package-view-controller) (notif :id))
  (let ((tab (#/object notif)))
    (when (eql tab (package-table self))
      ;; change the other two tables to reflect the package selected
      (let* ((pkg (svref (all-packages self) (#/selectedRow (package-table self))))
             (pkgs-used (package-use-list pkg))
             (pkgs-using (package-used-by-list pkg)))
        (setf (current-package-use-list self)
              (make-array (list (list-length pkgs-used)) :initial-contents pkgs-used))
        (setf (current-package-used-by-list self)
              (make-array (list (list-length pkgs-using)) :initial-contents pkgs-using))
        (#/reloadData (use-table self))
        (#/reloadData (used-by-table self))))))
```

You may be wondering at this point why we made our File's Owner object the delegate of the other two tables. Good question! In fact it isn't necessary for the way we have defined things so far. At one point I considered adding the ability for a user to select a package in either of the two bottom windows to make that the selection in the top one. I decided that was a bit of overkill, but if you like the idea it could easily be implemented within this function. Have at it!

And, as always, I defined a simple test function:

```
(defun test-package ()
  (make-instance 'package-view-controller))
```

If you eval all of this or just do (require :package-view) in the browser to load my code and then do (pv:test-package) in the browser you should see a nice interactive display of all the Lisp packages and their use connections.

Project 6: Loan Calc

Key Concepts: Bindings, Number formatters, Slider Controls, control enabling/disabling, control hiding/showing, continuous updating, window controller functionality

In this project we will develop something that begins to look like a complete application. It will do various sorts of loan calculations. Loans can be characterized by the starting value, the interest rate, the duration, and the monthly payment. Given any three of these you can calculate the fourth and our project will do exactly that. Ok, actually there are some combinations of three of these that result in no possible value for the fourth, but we'll discuss and manage those cases a bit later.

We will continue to explore the division of responsibility advocated by the *model/view/controller (MVC)* paradigm. In the last project we used a window controller to keep track of window objects, but nothing else. In this project it will become the NIB File owner and will assume responsibility for various ongoing window operations. As the File's Owner object, the window controller must provide an access path to the data (i.e. the model). To do that we will create our own window-controller subclass which contains a pointer to a loan object which maintains the data.

This project will take advantage of *bindings* that we can set up between interface fields and class slots. Bindings are created using Objective-C's *Key Value Coding (KVC)* mechanism. I'll provide a short introduction to it here, but I'd

strongly suggest that you consult other resources for more information. Assuming you have Apple developer tools loaded onto your system in the normal location you can look at:

file:///Developer/Documentation/DocSets/com.apple.ADC_Reference_Library.CoreReference.docset/Contents/Resources/Documents/documentation/Cocoa/Reference/CocoaBindingsRef/CocoaBindingsRef.html

or online you can access the same document at:

http://developer.apple.com/mac/library/documentation/Cocoa/Reference/CocoaBindingsRef/CocoaBindingsRef.pdf

and from either of these you'll see references to other documents.

KVC requires that two methods be callable for any KVC-compliant object:

```
(id)valueForKey:(NSString *)key
(void)setValue:(id)value forKey:(NSString *)key
```

If you think of this as access to object slot values by passing in the name of the slot as a parameter (i.e. as if by calling #'slot-value in Lisp) you won't be too far off the mark although there are differences in syntax and semantics as we'll see.

The class NSObject (from which all of our classes so far inherit) has a default implementation of these methods which simply calls the corresponding accessor methods. That is, for a call to valueForKey: to an object, obj, passing it an NSString with the value "mySlot" it will make a call equivalent to (#/mySlot obj). For a call to setValue:ForKey: to an object obj passing it an NSObject reference, say objref, and an NSString with the value "mySlot" it will make a call equivalent to (#/setMySlot: obj objref). Again note that all of the name conventions must by adhered to so that the methods will be invoked correctly. In the absence of accessor methods, KVC will try to find a slot of the same name as the key and access it directly. In our classes this would work with slots that are declared to be :foreign. If there are no slots or accessors with the name specified then the receiver calls itself with the function #'valueForUndefinedKey:. It is, therefore, possible to handle such exceptions in any way we want by defining our own version of that function.

What this practically means for us is that we can define Lisp slots for our classes (i.e. not :foreign) and provide Objective-C accessor methods for them and make our Lisp slots KVC compliant. We could even define accessor functions for a non-existent slot to create something like a *virtual slot* if desired.

KVC also supports access via a *key path*. A key path is basically a dot-separated list of successive keys. Often objects are connected via links between their slots and this provides a mechanism to follow those links to a final destination. We will use that mechanism here to link the values of user interface elements through the window controller and via its link to the Loan object to "slots" in the Loan object. KVC additionally supports access to and through collection objects like arrays and sets, but we'll have to wait for a future project to see uses for that.

Now we're ready to go build our interface. In IB you can create a new window project or open up "...InterfaceProjects/Loan Calc/loan.nib". Make your interface window look like the following:

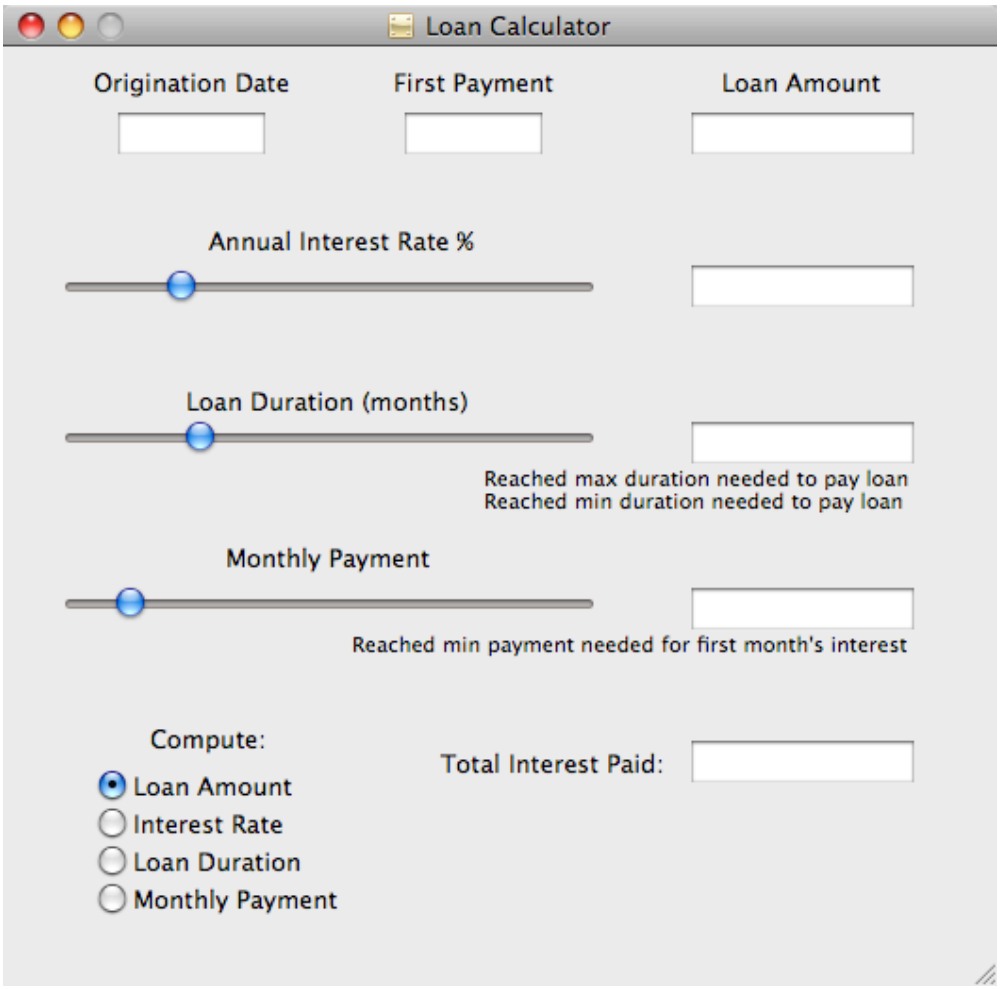

Figure 6.1 The Loan Calculator window

To do that you will need to place 7 Text Fields 11 Labels, 3 Horizontal Sliders (which you can find by clicking on the "Inputs & Values" folder in the Library window), and one Radio Group. Change the window title to something you like as we did previously. Click on the radio group once and in the Attributes view of the inspector window change the number of rows in the matrix to 4. To change the name of the first button click on it (once or twice depending on what is currently selected) until you have selected the "Button Cell". Change its title using the attributes view of the inspector window as shown in Figure 6.2 below.

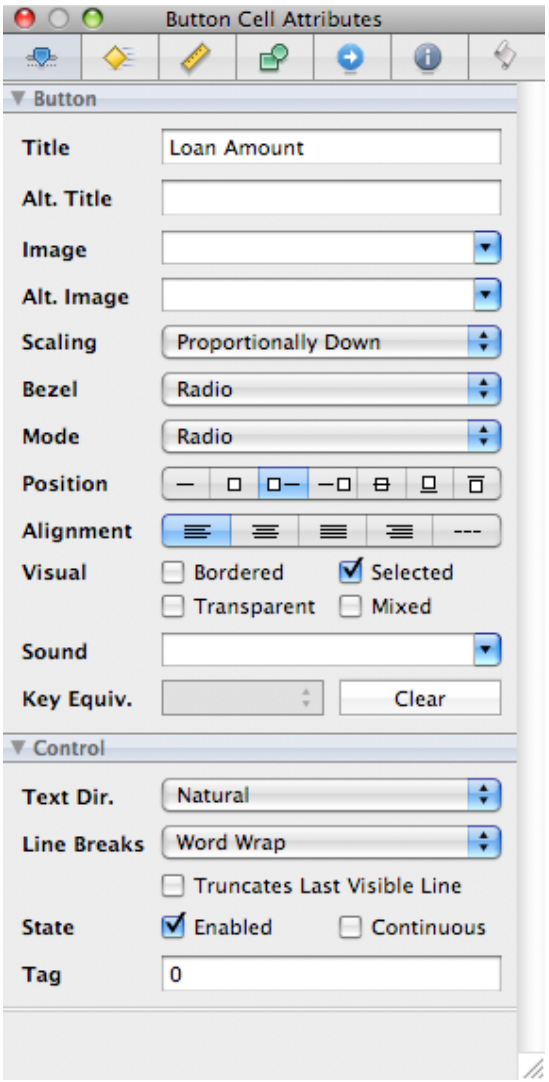

Figure 6.2 Radio Button Attributes

Similarly change the titles for the other radio buttons.

The three long labels with a smaller font that are situated under the Loan Duration and Monthly Payment boxes require some explanation. At runtime these will be conditionally displayed when certain combinations of loan variables occur. We'll talk about how to make that happen in a bit. For now place the labels, click on them, and in the Size view of the Inspector window (looks like a small ruler) select "Small" for the size. You can actually position the two labels under the Loan Duration box on top of each other since we will assure that only one of them is visible at any given time.

When you've completed this the overall look of the interface should be pretty much what you want.

Next we'll make sure that the text boxes operate the way we want them to. The text boxes to the right of each slider will show a numerical representation of the slider's value. I'll refer to each Text Field using the label that we gave it in the interface or the label of the corresponding slider.

Let's start with the Origination Date Text Field. This will contain a date and we want it to be easily editable by the user. We are going to make our life a little easier by attaching a *formatter* to this text box. A formatter is basically what it sounds like, namely a process that intervenes in the display of a value to make it look the way we want and assures that what a user enters into this field conforms to the format that we desire. Click on the Formatters folder in the library window and drag a Date Formatter over the Origination Text Field and drop it. Now when you click on that text field you will see below it a small version of the date formatter icon that you saw in the library window. When you click on that icon you have selected the Date Formatter object and can modify its characteristics in the inspector window. Select it now and modify the Date Style to be Short Style as shown in Figure 6.3 below.

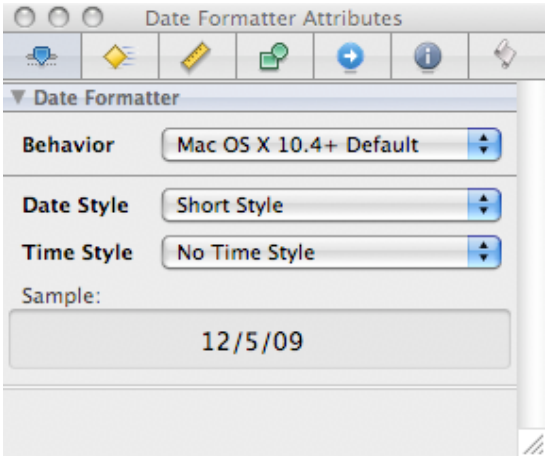

Figure 6.3 Modifying the Date Formatter

Similarly, add a Date Formatter to the First Payment Text Field and also change its Date Style to Short Style.

The Loan Amount Text Field will be a dollar amount. Drag a Number Formatter from the library window and drop it on top of this text field. Then edit it in the inspector attributes window. Change its Style to Currency as shown in Figure 6.4.

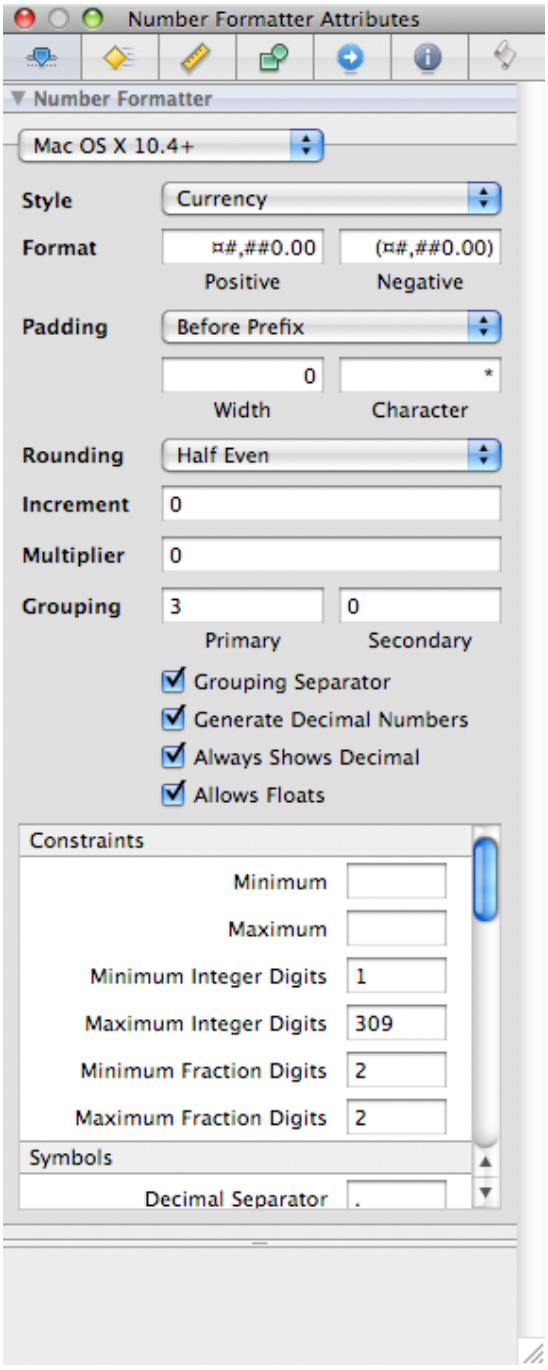

Figure 6.4 Modifying the number formatter

You will also check the boxes "Generate Decimal Numbers" and "Always Shows Decimal" for this formatter. The second of these requires that a decimal point always be displayed and entered to have a valid amount. If you don't want this, it won't change the application code at all, so you can omit that if you want to allow values without a decimal point. The first checkbox causes the formatter to create and pass NSDecimalNumber objects to Lisp when the field is modified. The normal default is to pass NSNumber objects. So what's the difference?

Whenever an application works with relatively large dollar amounts it can run into problems with the number of resolvable digits if it chooses to represent them using floating point values. You tend to lose cents or have them rounded in funny ways when you translate to or from a string representation such as we see in the user interface Text Fields. Lisp users have a relatively easy resolution to this problem because we can simply represent currency amounts as "the number of cents" and use very large fixnums. There are almost always enough digits to represent any value we might

need. C programmers are not so lucky and have to create arrays of shorts or some such thing to represent these values and then create special functions to operate on them. Ugly, but what can you do? Apple decided to make this easier for developers by defining a class called NSDecimalNumber and providing functions to operate on instances of it. While we could just use such objects directly within Lisp, it would be a real pain to have to use the relevant Objective-C functions to manipulate them for every arithmetic operation that we wanted to use.

Instead we will define some Lisp functions to translate back and forth from NSDecimalNumbers to Lisp integers. We'll examine them in detail when we get to the discussion of Lisp functions. But for now we have to assure that the dollar values that we enter and display are accurately moved between the user interface and Lisp by causing the formatter to generate NSDecimalNumber objects. So make sure that this box is checked.

Also add Number Formatters to the Monthly Payment and Total Interest Paid Text Fields and change their style to Currency. Be sure to check the same boxes for "Generate Decimal Numbers" and "Always Shows Decimal". Don't be confused by the "Allows Floats" checkbox. That does not refer to the type of object passed back and forth. Rather, that indicates whether the display will show values with a decimal point or only whole numbers. Since dollar values definitely have decimal values, you still want to select "Allows Floats".

Next add a Number Formatter to the Annual Interest Rate % Text Field and edit it in the inspector attributes window. First change its Style to be Percent. Note the Multiplier field is set to 100 by default. This will multiply any value that we set for this field by 100 before displaying it and divide it by 100 before returning it to any call that asks for the numerical value of the text field. For example, if we set the value to .10 it will be displayed as 10% just as we intend. Next we have a design decision to make as to how many decimal digits (if any) we want to allow. I have seen some credit card interest rates recently shown as 4 digit numbers so that's what I used, but do whatever seems reasonable to you. Set that in the "Maximum Integer Digits" field in the Constraints field of the Number Formatter attributes view. When you get done this should look as shown in Figure 6.5.

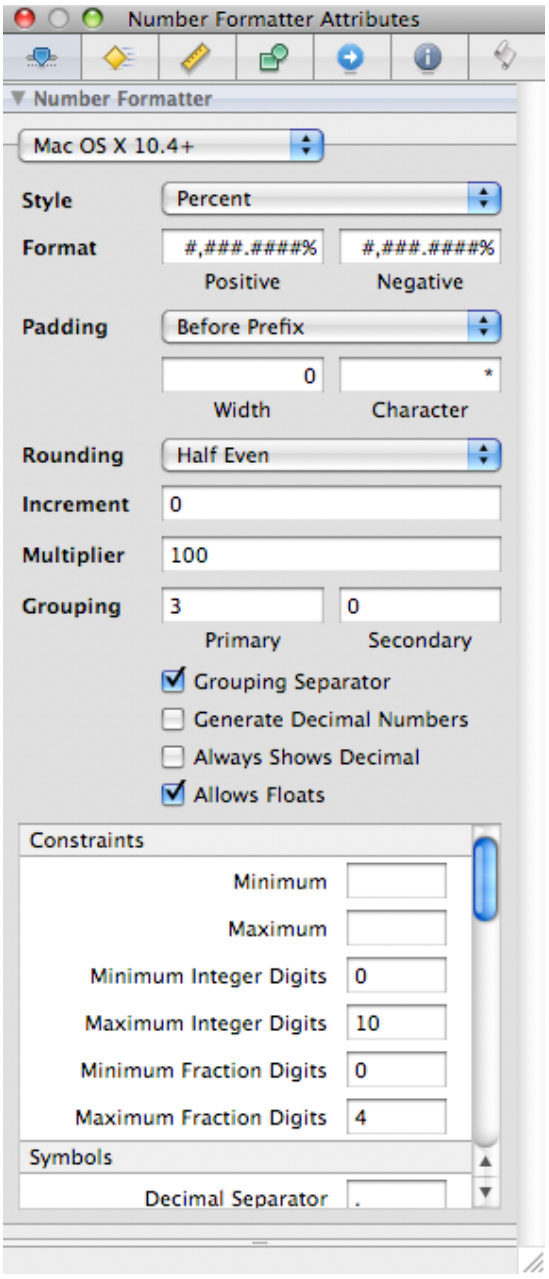

Figure 6.5 Annual Interest Rate Formatter

Although we could have used NSDecimalNumbers for this object too, there typically are not enough digits to cause a problem when translating back and forth to Lisp. But if you're concerned about that, feel free to modify the formatter and the relevant Lisp code to use them here too.

Finally add a number formatter to the Loan Duration Text Field. We want these values to always be integers, so we de-select the "Allows Floats" check box. In mine I also de-selected the other text boxes since we never need a separator and don't need to use NSDecimalNumbers to accurately transmit integers back and forth.

The look of our interface is complete and now we have to make connections similar to those we have made in previous projects. For the loan calculator we want our user to be able to select which of the four loan variables to compute and input values for the other three to compute it. When the user selects the "Interest Rate" radio button, for example, we would like to disable input for both the Horizontal Slider and the Text Field that correspond to that parameter. To do that our File's Owner object will need to have pointers to those controls that we want to enable and disable. First click on the File's Owner object in the nib document window and in the identify view of the inspector window change the name of the class to "LoanController". Now add the following outlets to contain those links: durSlider, durText, intSlider, intText,

loanText, paySlider, and payText. Note that although we could be precise about identifying the type of objects that these outlets will point to (as we did in previous projects), it is not necessary. We can use the "id" type which just means that it will be an NSObject of some kind and everything will work just fine.

Also add an outlet titled "window". We will use this as a link to the window object.

We'll also add an action method to be called when a radio button is pushed. Call it "buttonPushed". When you get done the identity view should look as in Figure 6.6 below.

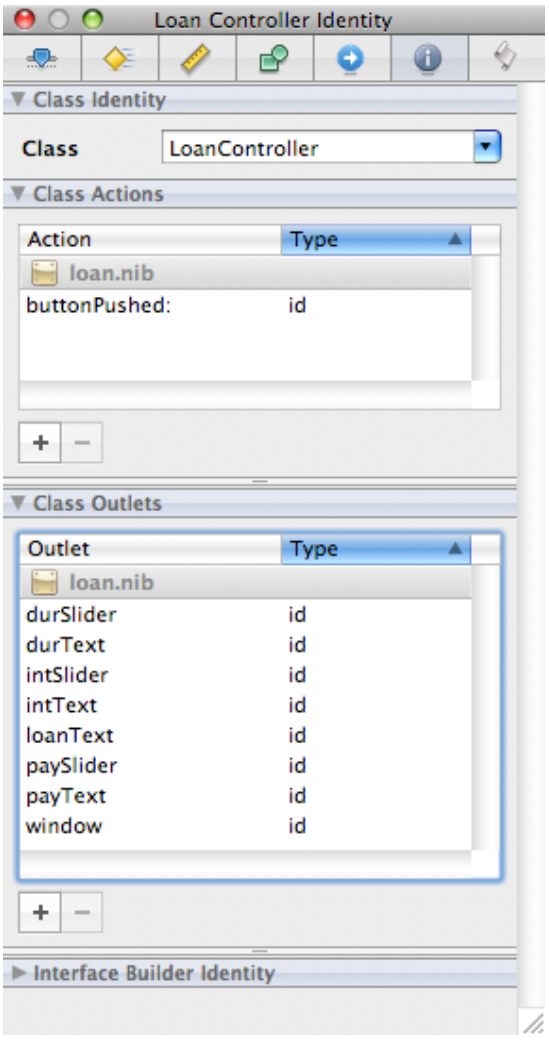

Figure 6.6 Loan Controller Identity

Now connect up those outlets to their corresponding display objects just as we have done previously. Control-click on the File's Owner object and you will see all of the outlets that you just added in the inspector window. For each outlet, click and drag from the circle next to it to the corresponding Text Field or Horizontal Slider in the Loan Calc window.

Similarly control-click and drag from the Radio Group to the File's Owner object and connect it to the buttonPushed: received action that we defined.

Similarly control-click and drag from the File's Owner object to the window object in the document window and connect it to the "window" outlet that we defined. While we're at it, control-click and drag from the window object to the File's Owner object and connect it to the window's "delegate" outlet. This will permit our loan-controller object to receive delegate messages. In particular, we will want to know when the window is closed by some means such as the user clicking on the window's close button or selecting "Close" from the file menu.

Next we need to make connections that are used by the display elements to report changes in their values that the user makes. But for this project we are going to take advantage of KVC and use *bindings* rather than setting targets and action

messages. What we will do is bind the value of a display field (e.g. a Text Field) to the value of a slot in a Loan object which is pointed to by the "loan" slot in the File's Owner (which at runtime will of course be an instance of our Lisp loan-controller class). If a binding such as this exists, then whenever KVC becomes aware that a value has changed on either side (i.e. either in the interface field or in the Loan class slot that it is bound to) it causes the value in the other bound object to be changed accordingly. This will save us a bunch of coding because we no longer have to be explicitly notified when something changes and then go query its value to see what the change was so that we can reflect it within our own structures in some way. We also don't have to explicitly set the value for the interface object when we compute some new value that should be displayed. In effect, the Loan object is now largely oblivious to the interface objects. As long as it is KVC compliant for all the slots that some interface might want that's all it needs to worry about. As we will see shortly, we can even bind multiple user interface objects to the same slot and they will both reflect the value in the slot.

Let's start by clicking on the Origination Date Text Field. In the Inspector window select the bindings view (little green square and circle joined together). If you can't already see the Value binding fields, click on the arrow to the left of the word "Value" to cause them to appear. Then click on the "Bind to:" box and select "File's Owner". In the Model Key Path field type in "loan.originationDate". As always capitalization must be precise. "loan" is the name of the slot in the File's Owner in which there is a link to a loan object. "originationDate" must be a KVC compliant "slot" in that loan object to which this Text Field will be bound. As we will see when we get to the Lisp code, we won't actually use an Objective-C slot, but instead will create accessor functions that make our class KVC-compliant for originationDate.

There is one more thing that we will do for any of our input objects and that is to click on the "Continuously Updates Value" box. This causes the bound value to be changed in the Loan object anytime the user makes any valid change to the field (i.e. one that the field's formatter accepts). Why would we do this rather than wait until the user is done editing? The main reason is to make our interface more responsive. Text Fields don't end editing until a user types a return in the field to indicate completion or clicks within another editable field. Leaving the field (even to click on a slider for example) is not enough to signify that editing is complete. So you can end up with one value appearing in the display and another value in your slot even though they are bound together. Trust me, this can lead to confusion. When you are done with the Origination Date Text Field it should look as in Figure 6.7 below.

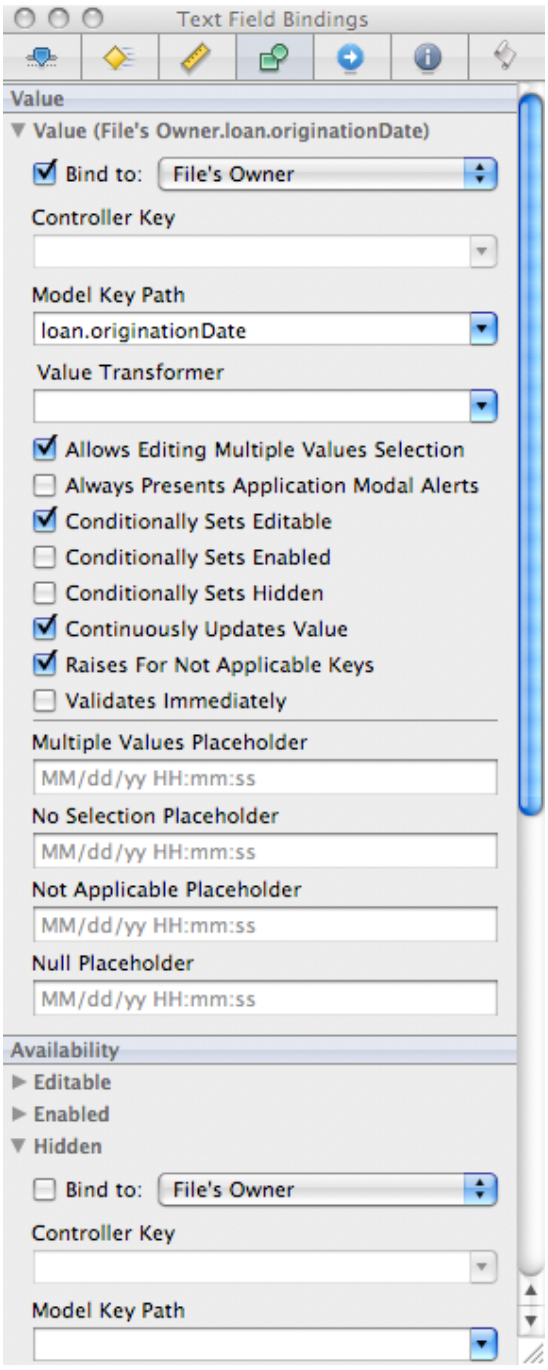

Figure 6.7 Origination Date Text Field Bindings

The First Payment Text Field is a bit different in that we don't want it to be editable. We will set it to be one month after the loan origination date. You may not want to make such a restriction, but the loan calculation will become somewhat more complex. Feel free to change that if you want to experiment and understand how to change the lisp code. Assuming you want to do it my way, select the Loan Payment Text Field and in the inspector attributes view deselect the "Selectable" and "Editable" check boxes so that this is merely a display field. Then in the bindings view bind its value to the "loan.firstPayment" slot of the Loan object (i.e. set the Model Key Path to "loan.firstPayment"). It is not necessary to check the "Continuously Updates Value" checkbox of course since this isn't an input field.

Similarly bind the values for the other Text Fields to a File's Owner slot:

"Loan Amount"	Text Field to	"loan.loanAmt"	slot
"Annual Interest Rate"	Text Field to	"loan.interestRate"	slot

"Loan Duration"	Text Field to	"loan.loanDuration"	slot
"Monthly Payment"	Text Field to	"loan.monthlyPayment"	slot
"Total Interest Paid"	Text Field to	"loan.totInterest"	slot

For the first four check the "Continuously Updates Value" checkbox and for the last make it an un-editable display-only field, just as we did for the First Payment Text Field.

We have three more bindings to do for those three small labels (one "Reached max ..." and two "Reached min ... ") that will only be displayed when a particular relationship holds among our loan variables. Under normal circumstances we want to hide these fields. We will only make them visible when Lisp decides that the conditions are right . Every view object has a boolean "hidden" attribute that causes the object to be invisible when that attribute is "Yes" and visible when it is "No". Cocoa with IB provides a very easy way to do this and that is to bind the "hidden" attribute of the Text Field to some slot. So we'll do just that; we'll bind them to appropriate slots in the Loan object which indicate that the corresponding condition exists. Click on the "Reached max ..." Text Field and in the Inspector window click on the bindings view. Then click the arrow next to "hidden" to expose the binding fields for the hidden attribute. Check the "Bind to:" box and select File's Owner. In the Model Key Path field enter "loan.maxDur".

When the condition is #&YES for the loan we want the Text Field to be displayed, but that corresponds to a hidden value of #$NO. So we need to negate the value of the loan object field in order to make it the value of the Text Field's hidden attribute. To do that select the "NSNegateBoolean" choice for the Value Transformer field. This will do exactly what you might expect and hide the field when the corresponding loan condition is false and display it when true.

Finally select "Yes" in the Null Placeholder field so that in the absence of information from our File's Owner object it will assume that the field is hidden. This is probably overkill since these fields will always be initialized whenever the Loan object exists, but in theory that need not be the case, so it's a good idea to think about what default value makes sense when creating bindings. When you get done, that binding information should look like Figure 6.8 below. Similarly bind the "Reached min payment ..." Text Field to the slot "loan.minPay" and the "Reached min duration ..." Text Field to the slot "loan.minDur".

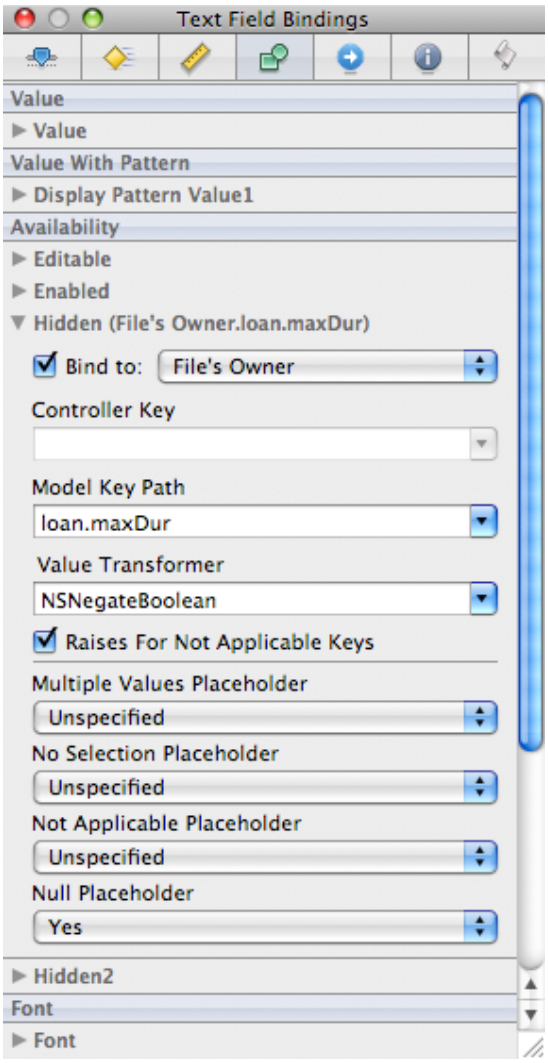

Figure 6.8

Now let's consider the Horizontal Sliders. We want them to reflect the same values as their corresponding Text Fields. We want changes to either control to be immediately reflected in the value displayed by the other as well as in the slot value in our File's Owner object. Making this happen is rather easy. We simply bind the value of the Horizontal Slider to the same slot as its corresponding Text Field. Then we have three things bound to a common value (the Horizontal Slider, the Text View, and the slot). When the value is changed in any of these, it will be immediately changed in the other two.

Start by clicking on the Annual Interest Rate slider. In the Attributes view of the Inspector window set its minimum and maximum values to 0 and .5 respectively (unless you want your annual interest rate to go above 50%, in which case I'll be happy to loan you some money). Set the current value to any initial value that you would like. Then click on the bindings view and bind the value of the slider to the "loan.interestRate" key path of File's Owner, just as we did for the corresponding Annual Interest Rate text field.

At this point we would love to click on the "Continuously Updates Value" checkbox if it existed, but it doesn't. Without that the slider value won't be reflected either in the text box or in the loan interestRate slot until you complete the movement of the slider and are no longer clicking on it. This is disconcerting for users because they would have no idea what value they would be setting the slider to as they moved it. Checking the technical documentation for sliders we discover that this attribute is available for sliders, so at least we know that what we want to do is possible. As you'll see a bit later, we will set this attribute explicitly in our Lisp code. Why IB (at least the version that I have) doesn't provide this capability I have no idea, but it's no problem to get the effect that we want.

In a similar fashion set the minimum and maximum values for the "Loan Duration" slider to 0 and 500 respectively (or whatever values you deem reasonable.) Again set the current value to some good initial value. Next set the minimum

and maximum values for the "Monthly Payment" slider to 0 and 10000 respectively and the current value to your desired starting value.

Note that we are not using NSDecimalNumbers to transfer information between the slider and Lisp. There are a couple of reasons for this. The first is that it doesn't seem to work. You can actually attach a formatter to a slider (and then select the checkbox that tells it to generate NSDecimalNumbers), but it doesn't seem to actually do anything and what Lisp gets is an NSNumber. So the first reason is that we can't use them and luckily the second reason is that we don't need them. That's because slider representations are crude at best and there is no confusion if the slider position is slightly different than what is actually in the Lisp class slot. What is important is keeping text strings that the user can see consistent with what is in corresponding Lisp slots.

To double-check that we've linked everything correctly, control-click on the File's Owner object. The pop-up should like very much like Figure 6.9 below.

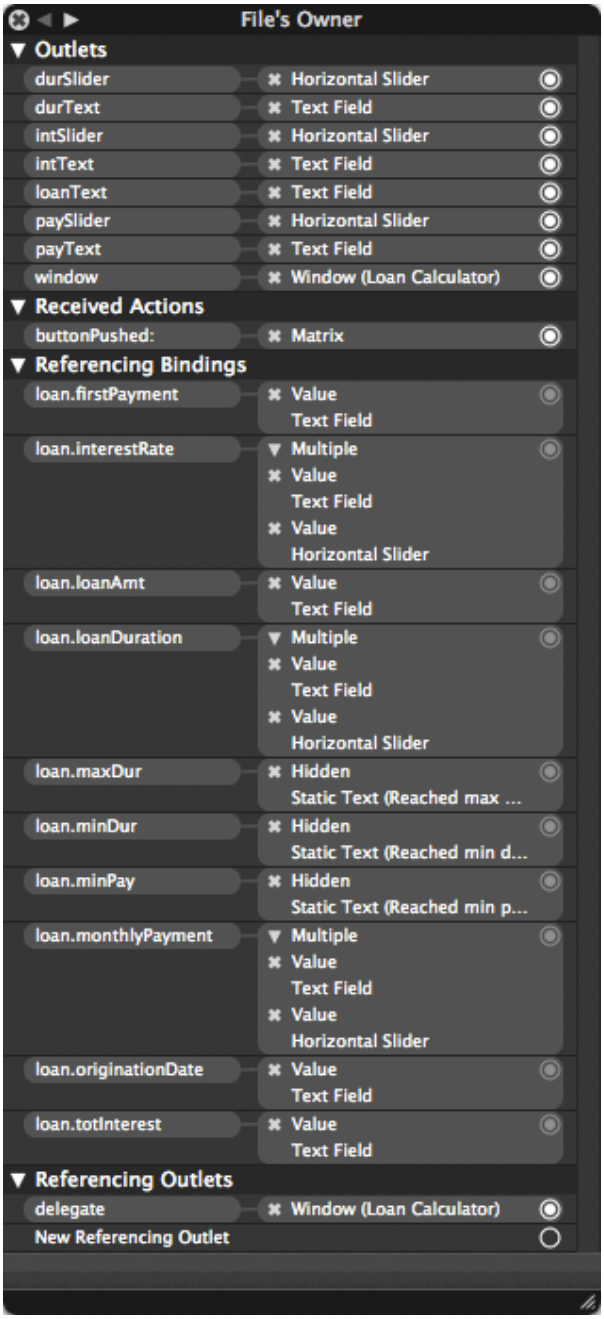

Figure 6.9 LoanController links

If everything looks ok, we can save the NIB file and move on to some Lisp code. You can open up my example file: "...InterfaceProjects/Loan Calc/loan-calc.lisp" or start your own.

Program Flow

The overall control flow of this application is something like the following:

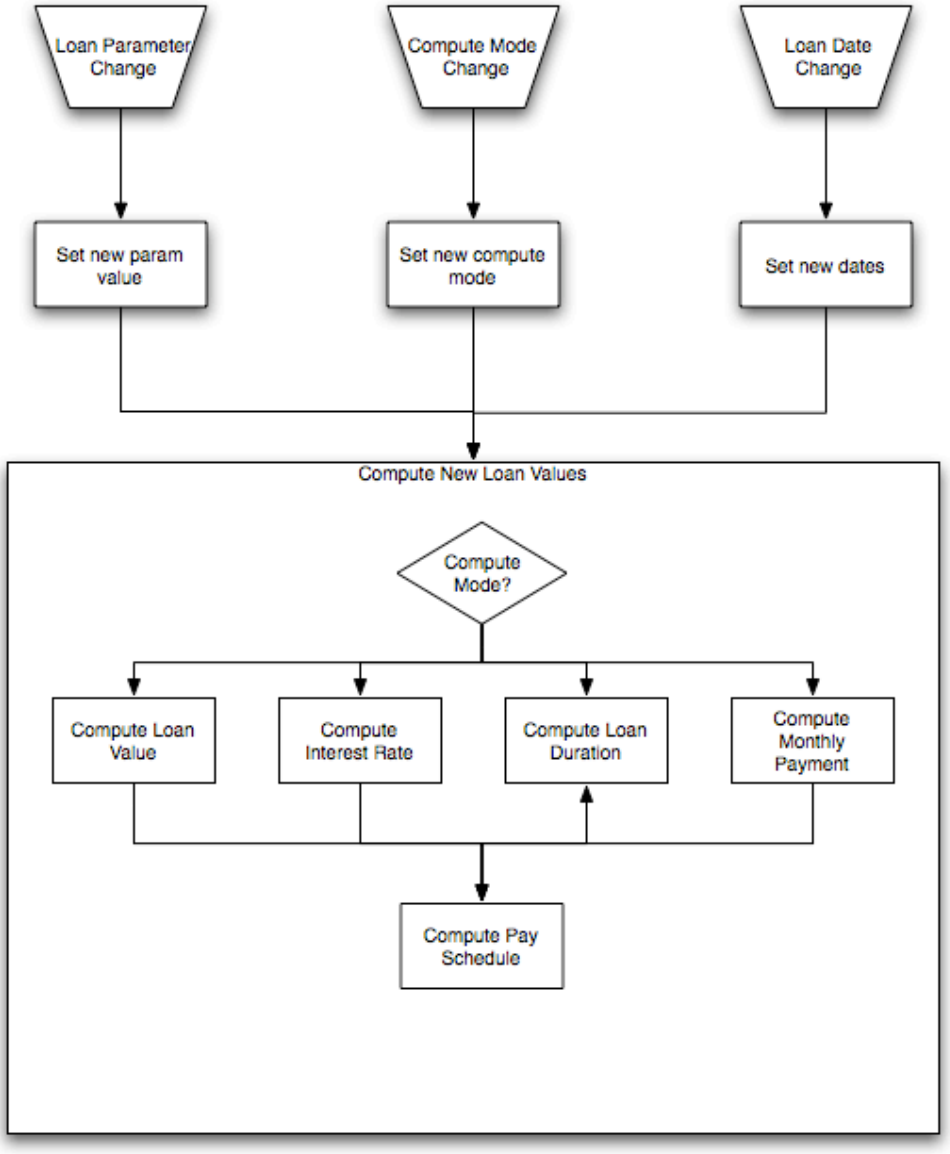

Figure 6.10 Loan Program flow

Users make changes in the loan window are first converted and saved as Lisp values. Depending on the current compute mode we re-compute the variable that is free to move based on the combination of other values. Then we compute the whole pay schedule.

There are various places in this process where we may change a value displayed in the loan window. It should be obvious that anytime we compute the new value of a loan parameter then we will cause the display to be updated. What is less obvious is that there are various combinations of parameters that may cause us to change a loan parameter other than the one we were explicitly directed to compute. For example, suppose we are told to compute the loan duration while the user manipulates the loan amount, interest rate, and monthly payment. Suppose the user increases the interest rate to a point where the monthly payment is no longer even as large as the first month's interest? Clearly the

user is adjusting the interest rate, so they must want it higher. But if we left the monthly payment alone then the loan could never be paid off. Not the least of our problems would be an infinite loop when we tried to compute a payment schedule. So in this case we choose to arbitrarily adjust the monthly payment so that it minimally covers the first month's interest plus $1 to start paying off the loan principle. There are other cases where we must adjust the loan duration either up (to pay off a loan) or down (because the loan is paid off earlier than the duration the user requested). So we change these values dynamically as necessary to maintain a consistent set of loan parameters. In these cases where we adjust a value that wasn't dictated directly by the user, we will display a small banner explaining why we did so. These are the banners that we created that are conditionally hidden in the loan window.

Utility Functions

The first thing to note in loan-calc.lisp is that there is a:

```
(require :date)
```

statement. This includes a utility file that I have built up over a long time which does various things with Lisp dates. Rather than isolate out just what is necessary for this project I've included all of it. But there are only a few functions that you will use and I won't discuss them much. Their names pretty much indicate what they do. The curious should be able to figure them out just by reading the code. Most are very simple.

Next there is a:

```
(require :decimal)
```

statement. This is a utility file that provides the two functions shown below.

```
(defun lisp-to-ns-decimal (int-val &key (decimals 2))
  ;; construct an NSDecimalNumber object with the given int-val and number of decimals
  ;; For example if you have a dollar amount 123.45 represented by the fixnum 12345
  ;; you would call (make-ns-decimal 12345 :decimals 2) to get a corresponding
  ;; NSDecimalNumber object. This object is the responsibility of the caller and a
  ;; call to #/release must be made when the caller is done with it.
  (unless (typep int-val 'fixnum)
    (error "Input must be a fixnum"))
  (#/decimalNumberWithMantissa:exponent:isNegative:
   ns:ns-decimal-number
   (abs int-val)
   (- decimals)
   (if (minusp int-val) #$YES #$NO)))
```

The lisp-to-ns-decimal function simply creates an instance of NSDecimalNumber. The input must be a fixnum and if it represents a number with other than 2 decimal digits you should specify the number using the :decimals keyword argument. Note that the function called to create the NSDecimal number is a class function of the NSDecimalNumber class. It is one of the so-called *convenience* functions that are provided by some cocoa classes. According to Apple's documentation:

http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.pdf
"Many classes provide methods of the form +className... that you can use to obtain a new instance of the class. Often referred to as “convenience constructors”, these methods create a new instance of the class, initialize it, and return it for you to use. You do not own objects returned from convenience constructors, or from other accessor methods."

So the upshot of this is that the NSDecimalNumber object that is returned is a temporary object. If you want to keep it longer than the span of the function that made the call you must retain it yourself and release it whenever you're done with it. Later discussion in this project will show other ways that we use other similar convenience functions.

```
(defun lisp-from-ns-decimal (ns-dec-obj &key (decimals 2))
  ;; This function returns a fixnum that corresponds to the NSDecimalNumber
  ;; or NSNumber that is passed in as the first argument.
  ;; The result will be scaled and rounded to represent the desired
  ;; number of decimal digits as specified by the :decimals keyword argument.
  ;; For example, if an NSNumber is passed in which is something like 123.45678
  ;; and you ask for 2 decimal digits, the returned value will be the integer 12346.
  (let* ((loc (#/currentLocale ns:ns-locale))
         (lisp-str (ccl::lisp-string-from-nsstring
                    (#/descriptionWithLocale: ns-dec-obj loc)))
```

```
        (str-len-1 (1- (length lisp-str)))
        (dec-pos (or (position #\. lisp-str) str-len-1))
        (dec-digits (- str-len-1 dec-pos))
        (dec-diff (- decimals dec-digits))
        (mantissa-str (delete #\. lisp-str)))
    (cond ((zerop dec-diff)
           (read-from-string mantissa-str))
          ((plusp dec-diff)
           (read-from-string (concatenate 'string
                                          mantissa-str
                                          (make-string dec-diff :initial-element #\0))))
          (t ;; minusp dec-diff
           (let ((first-dropped (+ (length mantissa-str) dec-diff)))
             (+ (if (> (char-code (elt mantissa-str first-dropped)) (char-code #\4)) 1 0)
                (read-from-string (subseq mantissa-str 0 first-dropped)))))))))
```

The lisp-from-ns-decimal function takes either an NSDecimalNumber or an NSNumber as input and converts it to a fixnum that is returned. This asks the object for its string representation and then reads in the string after removing the ".". Finally it is scaled and rounded to set the number of implicit decimal digits in the returned fixnum to whatever was requested by the :decimals keyword argument. There is a fair amount of string manipulation going on here just to avoid the use of any floating point operations. There may be a more efficient process than this and the reader is invited to provide a better alternative. I suspect that a more efficient function would necessarily have to differentiate between NSDecimalNumber and NSNumber inputs. In any case, this seems to work fast enough to avoid any interface lag.

Loan-Controller Functionality

The next section of loan-calc.lisp concerns the loan-controller class and its functionality. First is the class declaration:

```
(defclass loan-controller (ns:ns-window-controller)
  ((loan :foreign-type :id :accessor loan)
   (loan-text :foreign-type :id :accessor loan-text)
   (int-text :foreign-type :id :accessor int-text)
   (dur-text :foreign-type :id :accessor dur-text)
   (pay-text :foreign-type :id :accessor pay-text)
   (int-slider :foreign-type :id :accessor int-slider)
   (dur-slider :foreign-type :id :accessor dur-slider)
   (pay-slider :foreign-type :id :accessor pay-slider))
  (:metaclass ns:+ns-object))
```

The first slot, "loan", will link the loan-controller instance to an instance of the loan class that we will discuss below. After that are the normal sort of :foreign slots that point to interface objects we've come to expect. They are used to contain the links that we created in IB from the File's Owner object to the various Text Fields and Horizontal Sliders. We'll use those to selectively enable and disable those controls at appropriate times.

Next we define a custom initialization function #/initWithLoan for ourselves:

```
(objc:defmethod (#/initWithLoan: :id)
                ((self loan-controller) (ln :id))
  (setf (loan self) ln)
  (let* ((nib-name (ccl::%make-nsstring
                    (namestring (truename "ip:Loan Calc;loan.nib"))))
         (init-self (#/initWithWindowNibPath:owner: self nib-name self)))
    init-self))
```

The caller of this function must supply the loan object pointer that will be used. This init function first calls the #/initWithWindowNibPath:owner: method to set the path to our custom NIB file. When we discuss the loan methods below we will explain how the loan and loan-controller objects cooperate to open and close windows. This will also help you understand what Document objects do when we get to the next project.

When a radio button is pushed we will change which of the loan variables is computed from the others. The function that is called by the radio button matrix is shown below.

```
(objc:defmethod (#/buttonPushed: :void)
                ((self loan-controller) (button-matrix :id))
  (with-slots (loan loan-text int-text dur-text pay-text int-slider
```

```
                dur-slider pay-slider) self
  (let ((cm (#/selectedRow button-matrix)))
    (unless (eql cm (compute-mode loan))
      (case (compute-mode loan)
        (0 (#/setEnabled: loan-text #$YES))
        (1 (#/setEnabled: int-text #$YES)
           (#/setEnabled: int-slider #$YES))
        (2 (#/setEnabled: dur-text #$YES)
           (#/setEnabled: dur-slider #$YES))
        (3 (#/setEnabled: pay-text #$YES)
           (#/setEnabled: pay-slider #$YES)))
      (setf (compute-mode loan) cm)
      (case cm
        (0 (#/setEnabled: loan-text #$NO))
        (1 (#/setEnabled: int-text #$NO)
           (#/setEnabled: int-slider #$NO))
        (2 (#/setEnabled: dur-text #$NO)
           (#/setEnabled: dur-slider #$NO))
        (3 (#/setEnabled: pay-text #$NO)
           (#/setEnabled: pay-slider #$NO)))
      (compute-new-loan-values loan)))))
```

We first determine which user interface objects are currently disabled and re-enable them. Then we disable those that correspond to the value that we are now being asked to compute. Note that KVC will know what we did because we called the appropriate Objective-C #/set... accessors. So we do not have to explicitly call #/willChangeValueForKey: and #/didChangeValueforKey:. And of course we set the slot value that contains the compute mode in the attached loan object. Note that the Loan object (which controls data) does not have to be aware of which radio button is pushed or even the fact that we're using radio buttons to select the compute mode.

The next method we'll discuss, #/awakeFromNib, is called as a result of us setting the window's delegate outlet to point to the File's Owner class in IB. As its name implies, this method is called after the window has been loaded from the nib and initialized.

```
(objc:defmethod (#/awakeFromNib :void)
                ((self loan-controller))
  (#/setEnabled: (loan-text self) #$NO)
  ;; set the sliders to update continuously so that the text boxes reflect the current value
  ;; Note that we can set this in IB for text boxes, but not, apparently, for sliders
  (#/setContinuous: (int-slider self) #$YES)
  (#/setContinuous: (dur-slider self) #$YES)
  (#/setContinuous: (pay-slider self) #$YES))
```

Here we initialize some interface attributes. Since we have set our default radio button to be to calculate the loan value we will disable input into that Text Field for now. As discussed above, when the user selects a different radio button we'll change which controls are enabled and disabled.

Perhaps this is a good point to discuss another subject. When you were binding the values of various display fields you may have noticed that in addition to binding their values you can bind to a slot that specifies whether they are enabled or disabled. We could certainly have used that mechanism here either by defining a :foreign slot for each control and making sure that it was initialized properly or by providing accessor functions. But then when a button was pushed we would have to set those slot values rather than just directly setting the enabled property of the object. The point is that whether the object should be enabled or not is entirely a property of the interface state and not the data, so there really isn't any advantage of binding that property to a slot rather than just setting it directly.

Next the function sets that Continuous update property for the three sliders that we discussed previously. If you want to see the difference in functionality, omit those calls and observe how the interface operates when a slider is moved.

The next loan-controller method we'll discuss is also called as a result of its being the window delegate.

```
(objc:defmethod (#/windowWillClose: :void)
                ((self loan-controller) (notif :id))
  (declare (ignore notif))
  (when (loan self)
    ;; Tell the loan that the window is closing
    ;; It will #/autorelease this window-controller)
```

```
    (window-closed (loan self))))
```

Once the window is closed we want the loan-controller object to go away as well (unless you want to re-show the window at some later time). When a window controller object that loaded a nib file is deallocated it then releases all of the top-level nib objects that were created when the nib file was loaded. These notably include the window object which then releases all of the interface objects that it points to. These may, in turn, release more deeply embedded objects. So if we want all of that memory to be reclaimed, it is important to make sure that the loan-controller object goes away. So the first thing we do is call the loan's "window-closed" method to tell it what is happening. We'll talk about that method later, but at this point just recognize that it has a link to this loan-controller object which must be released in order for it to be garbage collected and that's one of the things that the window-closed method will do.

That's the end of the loan-controller functionality. We'll now switch from talking about the loan-controller class to discussing the loan class and other subsidiary loan functions.

Loan Functionality

Most of the loan computation code is built around a basic loan equation:

MonthlyPayment = Loan * (MonthlyInterest + (MonthlyInterest / ((1 + MonthlyInterest)^LoanDuration - 1))).

We can easily rearrange this to derive the loan value from the other parameters. To compute interest from the other values, the code searches because there isn't an algebraic solution for it. To compute the loan duration we create a complete payout schedule and just look at its length.

First we define a utility function that helps with these calculations:

```
(defun pay-to-loan-ratio (mo-int loan-dur)
  ;; Just computes the MonthlyPayment/Loan ratio from the basic loan equation given above
  (if (zerop mo-int) 0
    (+ mo-int (/ mo-int (1- (expt (1+ mo-int) loan-dur))))))
```

With a little mathematical manipulation you can see that this is derived from the basic loan equation above.

The loan class is defined as follows:

```
(defclass loan (ns:ns-object)
  ((loan-amount :accessor loan-amount :initform 0)
   (interest-rate :accessor interest-rate :initform 0)
   (loan-duration :accessor loan-duration :initform 0)
   (monthly-payment :accessor monthly-payment :initform 0)
   (origination-date :accessor origination-date :initform (now))
   (first-payment :accessor first-payment :initform (next-month (now)))
   (pay-schedule :accessor pay-schedule :initform nil)
   (window-controller :accessor window-controller :initform nil)
   (compute-mode :accessor compute-mode :initform 0)
   (max-dur :foreign-type #>BOOL :accessor max-dur)
   (min-dur :foreign-type #>BOOL :accessor min-dur)
   (min-pay :foreign-type #>BOOL :accessor min-pay))
  (:metaclass ns:+ns-object))
```

You'll see several Lisp slots (i.e. not defined as :foreign-type) defined first. At first blush they seem to correspond to the slot names that we bound to the various IB view object values. That was an intentional naming choice on my part to keep them straight, but as Lisp slots they are not accessible to Objective-C functions. Later I will discuss how we make use of them.

After that is the "window-controller" slot in which we will keep a pointer to the instance which controls the window as described previously.

That is followed by the compute-mode slot which keeps track of which variable is being computed given the values of the others.

Next are three slots with a :foreign-type of #>BOOL. These are the slots that we bound to the "hidden" attribute of the three longer messages that we only want to appear when we detect certain conditions. In some sense they indicate the presence or absence of those conditions so we have not really exposed the user interface to the loan object.

```
(defmethod initialize-instance :after ((self loan)
                                       &key &allow-other-keys)
```

```
  (setf (window-controller self)
        (make-instance (find-class 'loan-controller)
          :with-loan self))
  (#/showWindow: (window-controller self) self))
```

This function simply creates the loan-controller object that will manage the display window and then tells it to show its window.

Recall that we used bindings for those three condition messages. Initially we want them all to be hidden because the corresponding conditions haven't been detected yet (because we don't even have values to calculate anything with yet). All :foreign-type slots are always initialized to 0, which conveniently corresponds to #$NO, so we don't have to do anything special to initialize those conditions.

If we DID have to initialize those conditions, we would also have to make sure that KVC was made aware of the changes, so that they would be given to the bound user interface objects. Generally we can assure that KVC is aware of changes made to slots in any of three ways. First we can explicitly call #/setValue:forKey: for these slots. Second, we can define appropriately named Objective-C accessor functions for these slots and use them. Third, we can surround the Lisp call which sets the slot with a prior call to #/willChangeValueForKey: and a subsequent call to #/didChangeValueForKey:. Later we will see how that third method is used both for changes to these condition slots and also for the value bindings that we made even though there is no actual :foreign-type slot slot that contains those values.

The next methods we will discuss are those accessor methods which are called when KVC detects that a user interface object has changed its value or that one of our slot values has changed. Let's start with the former set. When either a slider is moved or a value in a Text Field is changed KVC will call the corresponding #/setValue:forKey: method for the bound "slot" that we specified in IB. The default implementation of that method will call an appropriately named Objective-C accessor method. If the slot was named "mySlot", then the method called would be "setMySlot:". So we first define a group of these accessors.

```
(objc:defmethod (#/setLoanAmt: :void)
                ((self loan-controller) (amt :id))
  (setf (loan-amount self) (lisp-from-ns-decimal amt))
  (compute-new-loan-values self))
```

Some things to note here. In IB we bound the Loan Amount Text Field to the slot "loanAmt" in File's Owner. An appropriately named write accessor for that slot is "setLoanAmt". We know that the object passed in as the new value will be an NSDecimalNumber because we told the formatter for that Text Field to generate such numbers. So we can just take it and convert it to a Lisp fixnum and put it in a conveniently named slot in our loan-controller object. It is now a Lisp value and can be manipulated normally. After doing this we trigger the side-effect of computing any new loan values. What gets computed will depend on the compute-mode field. As you may recall, this value is set by the loan-controller object when a new radio button is selected in the Loan window.

```
(objc:defmethod (#/setInterestRate: :void)
                ((self loan-controller) (rate :id))
  (setf (interest-rate self) (#/floatValue rate))
  (compute-new-loan-values self))
```

In this case we will be passed an NSNumber object and we will ask for its float value which we put into our Lisp slot before computing new loan values.

```
(objc:defmethod (#/setLoanDuration: :void)
                ((self loan-controller) (dur :id))
  (setf (loan-duration self) (#/longValue dur))
  (compute-new-loan-values self))
```

Durations are never float values (remember that we de-selected the "Allows Floats" checkbox in the formatter for this field in IB), so we can ask for a long from the NSNumber and get it.

```
(objc:defmethod (#/setMonthlyPayment: :void)
                ((self loan-controller) (pay :id))
  (setf (monthly-payment self) (lisp-from-ns-decimal pay))
  (compute-new-loan-values self))
```

The monthly payment is another dollar amount, so we use NSDecimal numbers to move values from the Text Field. But wait a minute, we also bound the monthly payment slider to this same value and it cannot provide NSDecimal numbers. Well, as we mentioned when we discussed the

lisp-from-ns-decimal function, it can also handle NSNumber objects and has the nice side-effect of rounding the floating point input value that we get to two decimals and putting it into the same Lisp internal format that we want to use for currency values.

```
(objc:defmethod (#/setOriginationDate: :void)
                ((self loan-controller) (dt :id))
  (let ((new-dt (ns-to-lisp-date dt)))
    (setf (origination-date self) new-dt)
    (#/willChangeValueForKey: self #@"firstPayment")
    (setf (first-payment self) (next-month new-dt))
    (#/didChangeValueForKey: self #@"firstPayment"))
  (compute-new-loan-values self))
```

When we set the loan origination date, we convert it to the corresponding Lisp data format. We also want to set the first payment date to be exactly one month later so we use the "next-month" function from the date.lisp file. Note that we have to notify KVC that we are changing the firstPayment "slot" so that will change the value of the First Payment Text Field in the Loan window. Note that an alternative here would have been to just directly call the write accessor:

(#/setFirstPayment self dt)

which would have let KVC know what was going on. That would have required two separate conversions of the date to Lisp so I elected not to do that.

```
(objc:defmethod (#/setFirstPayment: :void)
                ((self loan-controller) (pay :id))
  (let ((new-pay (ns-to-lisp-date pay)))
    (setf (first-payment self) new-pay))
  (compute-new-loan-values self))
```

This method is not, strictly speaking, necessary. We made the First Payment Text Field un-editable so this should never be called. But who knows how we might decide to change things in the future, so we created it just in case it is needed.

The corresponding reader accessor functions are shown below and are fairly straight-forward. We just have to make sure that the Objective-C objects that we create to represent values are released at some future time so we don't have a memory leak. So in all cases we use the same class convenience functions that we discussed in the context of the lisp-to-ns-decimal code (or in some cases functions that return the results of these functions). These all return objects that are not owned by us, so we do not need to do anything about releasing them here. There is also no point in trying to cache values here for future use because the only time that these methods will be called is when KVC becomes aware that they have changed.

```
(objc:defmethod (#/loanAmt :id)
                ((self loan))
  (lisp-to-ns-decimal (loan-amount self)))

(objc:defmethod (#/interestRate :id)
                ((self loan))
  (#/numberWithFloat: ns:ns-number (float (interest-rate self))))

(objc:defmethod (#/loanDuration :id)
                ((self loan))
  (#/numberWithInt: ns:ns-number (loan-duration self)))

(objc:defmethod (#/monthlyPayment :id)
                ((self loan))
  (lisp-to-ns-decimal (monthly-payment self)))

(objc:defmethod (#/originationDate :id)
                ((self loan))
  (lisp-to-ns-date (origination-date self)))

(objc:defmethod (#/firstPayment :id)
                ((self loan))
  (lisp-to-ns-date (first-payment self)))
```

There is one last reader accessor to discuss and that is for the Total Interest Text Field. We created Lisp slots that corresponded to each of the other Text Fields, but for this one we did not (mostly just to illustrate that you don't actually need to do so.)

```
(objc:defmethod (#/totInterest :id)
                ((self loan))
  (lisp-to-ns-decimal (reduce #'+ (pay-schedule self)
                              :key #'seventh
                              :initial-value 0)))
```

The #/totInterest accessor function computes the total interest from the current payment schedule and returns it as a NSDecimalNumber object whenever the Total Interest Text Field requests it. As always, the call to lisp-to-ns-decimal uses two decimal digits as the default assumption and encodes the value that we give it accordingly.

As shown in the simple flowchart for our loan application and in the set... accessor code above, virtually any change the user makes results in a call to compute-new-loan-values. Here is that function:

```
(defmethod compute-new-loan-values ((self loan))
  ;; For the sake of expediency we assume monthly componding
  ;; The basic equation governing these computations is
  (with-slots (compute-mode interest-rate loan-duration monthly-payment
                            loan-amount pay-schedule) self
    (case compute-mode
      (0
       ;; compute the loan amount
       (unless (or (zerop interest-rate)
                   (zerop loan-duration)
                   (zerop monthly-payment))
         (#/willChangeValueForKey: self #@"loanAmt")
         (setf loan-amount
               (round (/ monthly-payment
                         (pay-to-loan-ratio (/ interest-rate 12)
                                            loan-duration))))
         (set-pay-schedule self)
         (#/didChangeValueForKey: self #@"loanAmt")))
      (1
       ;; compute the interest rate
       (unless (or (zerop loan-amount)
                   (zerop loan-duration)
                   (zerop monthly-payment))
         (#/willChangeValueForKey: self #@"interestRate")
         (setf interest-rate
               (* 12 (/ (floor (* 1000000 (compute-int-rate self)))
                        1000000)))
         (set-pay-schedule self)
         (#/didChangeValueForKey: self #@"interestRate")))
      (2
       ;; compute the loan duration
       (unless (or (zerop interest-rate)
                   (zerop loan-amount)
                   (zerop monthly-payment))
         (#/willChangeValueForKey: self #@"loanDuration")
         (set-pay-schedule self)
         (setf loan-duration
               (list-length pay-schedule))
         (#/didChangeValueForKey: self #@"loanDuration")))
      (3
       ;; compute the monthly payment
       (unless (or (zerop interest-rate)
                   (zerop loan-amount)
                   (zerop loan-duration))
         (#/willChangeValueForKey: self #@"monthlyPayment")
         (setf monthly-payment
               (round (* loan-amount
                         (pay-to-loan-ratio (/ interest-rate 12)
                                            loan-duration))))
         (set-pay-schedule self)
         (#/didChangeValueForKey: self #@"monthlyPayment"))))))
```

When the program is first started, all of the input fields have 0 values and we cannot compute a loan value. So the first thing we do, for whatever compute mode the user has selected, is to make sure that we have non-zero values for the variables that are needed to compute it. Another thing that we do for all compute modes is let KVC know that we have changed the value that should be displayed in the corresponding Text Field by calling the appropriate #/willChange... and #/didChange methods. In all compute modes we will also call the set-pay-schedule method that we will discuss below.

The computations for loan amount or monthly payment have easy closed-form solutions that are derived from the basic loan equation. The loan duration can be directly derived from the length of the payment schedule so we just call that first and then set the value of the loan-duration slot. We use a search algorithm to compute the interest rate and round the value returned to the number of digits we want to display. There is a separate method to do the search (compute-int-rate) which is shown below.

```
(defmethod compute-int-rate ((self loan))
  ;; Find a monthly interest rate that makes the rest of the values work.
  ;; There isn't an algebraic solution for the interest rate, so let's search for it.
  ;; Find a suitable search range and do a binary search for it. Even for large interest
  ;; rates the number of search iterations should be minimal.

  (with-slots (loan-amount monthly-payment loan-duration interest-rate) self

    ;; First we'll check to see whether the monthly payment is great than the loan amount.
    ;; If so we'll set the interest rate directly so that the loan is paid off in one month.
    ;; This avoids some ugly arithmetic overflow things that can happen when interest rates
    ;; go off the charts
    (let ((max-monthly-rate (/ $max-interest-rate$ 12)))
      (if (>= monthly-payment loan-amount)
        (min max-monthly-rate (1- (/ monthly-payment loan-amount)))
        (let ((imin (max 0 (min max-monthly-rate
                                (/ (- (* monthly-payment loan-duration) loan-amount)
                                   (* loan-duration loan-amount)))))
              ;; imin is basically a rate that would result in the first month's interest as
              ;; the average interest paid for all months. Since we know it must be greater
              ;; than this, we have a guaranteed lower bound. But we cap it at our allowed
              ;; monthly maximum interest.
              (imax (min max-monthly-rate
                         (- (/ monthly-payment loan-amount) .000008333)))
              ;; imax is a rate that would result in the first month's interest being
              ;; minimally smaller than the payment. Since we must pay off in a finite
              ;; duration, this is a guaranteed maximum. We cap it the allowed maximum
              ;; monthly rate.
              (target-p-l-ratio (/ monthly-payment loan-amount)))
          (unless (>= imax imin)
            (error "Max int = ~8,4f, Min int = ~8,4f" imax imin))
          (do* ((i (/ (+ imin imax) 2)
                   (/ (+ imin imax) 2))
                (p-l-ratio (pay-to-loan-ratio i loan-duration)
                           (pay-to-loan-ratio i loan-duration)))
               ((<= (- imax imin) .000001) imax)
            (if (>= target-p-l-ratio p-l-ratio)
              (setf imin i)
              (setf imax i)))))))
```

This method uses a binary search to find the interest rate that produces something very close to the target payment to loan ratio. We can compute that target from the user-specified payment and loan value parameters. The initial bounds for the interest rate are computed as discussed in the code comments.

At one time I used a more precise upper bound that for some reason resulted in extremely slow performance in circumstances where the computed rate eventually reached the upper bound. I decided to not chase that down, but I expect that it was caused by deriving a rate that resulted in an extremely long loan payout schedule. The method here seems to work pretty well and responsively. Consider it a challenge to do something more precise.

The final method we will discuss computes a detailed payment schedule given all the loan parameters. Since we never display this anywhere you may wonder why it exists. There are four reasons. First it makes it very easy to determine the

loan duration when we need to compute that from the other values. Second, I found it to be a useful debugging tool. I could print out the payment schedule in the listener just to make sure that everything looked good. I uncovered several bugs that way. Third, I wanted to provide a challenge for you to add functionality to this application. And fourth, I wanted to have something more substantial to print out when we get to Project #7.

```
(defmethod set-pay-schedule ((self loan))
  ;; create a detailed payment schedule for the loan using daily compounding of interest
  ;; Payments are on the same date of each month, but the number of days between payments
  ;; varies because the number of days in each month varies.
  ;; We compute accurate interest compounded daily for the actual number of days.
  (let ((monthly-interest (/ (interest-rate self) 12))
        (payment (monthly-payment self))
        (sched nil)
        (display-min-pay-banner nil))
    (prog1
        (do* ((begin (loan-amount self) end)
              (begin-date (first-payment self) end-date)
              (end-date (next-month begin-date) (next-month begin-date))
              (int (round (* begin monthly-interest))
                   (round (* begin monthly-interest)))
              (end (- (+ begin int) payment) (- (+ begin int) payment)))
             ((not (plusp end))
              (progn
                (push (list (short-date-string begin-date)
                            (/ begin 100)
                            (/ int 100)
                            (/ payment 100)
                            (short-date-string end-date)
                            (/ end 100)
                            int)
                      sched)
                (setf (pay-schedule self) (nreverse sched))))
          (when (>= end begin)
            ;; oops, with this combination of values the loan will never
            ;; be paid off, so let's set a minimum payment required
            ;; Display a field that tells user the minimum payment was reached
            (setf display-min-pay-banner t)
            (#/willChangeValueForKey: self #@"monthlyPayment")
            (setf (monthly-payment self) (1+ int))
            (#/didChangeValueForKey: self #@"monthlyPayment")
            ;; now patch up our loop variables and keep going
            (setf payment (monthly-payment self))
            (setf end (1- begin)))
          ;; put the last payment into the list
          (push (list (short-date-string begin-date)
                      (/ begin 100)
                      (/ int 100)
                      (/ payment 100)
                      (short-date-string end-date)
                      (/ end 100)
                      int)
                sched))
      (#/willChangeValueForKey: self #@"totInterest")
      ;; we'll make the total interest field call our accessor
      ;; to generate a new amount
      (#/didChangeValueForKey: self #@"totInterest")
      (if display-min-pay-banner
        (progn
          ;; Set a condition that says the minimum payment was reached
          (setf display-min-pay-banner t)
          (#/willChangeValueForKey: self #@"minPay")
          (setf (min-pay self) #$YES)
          (#/didChangeValueForKey: self #@"minPay"))
        (progn
          ;; otherwise reset that condition
```

```
    (#/willChangeValueForKey: self #@"minPay")
    (setf (min-pay self) #$NO)
    (#/didChangeValueForKey: self #@"minPay")))
;; If we happen to be computing the interest rate, then
;; the combination of loan-amount and monthly payment will
;; determine a maximum interest rate. This, in turn,
;; determines a maximum loan duration. If the duration was set
;; longer than this by the user, we will reset the
;; lone duration value to the maximum needed.
;; If, on the other hand, the monthly payment is set so low that
;; the interest rate approaches 0, then we may have to adjust the
;; loan duration up to the minimum needed to pay the loan.
;; Let's start by resetting our two "duration" conditions and then we'll
;; set them if conditions dictate.
;; Reset a condition that indicates the max duration was reached
(#/willChangeValueForKey: self #@"maxDur")
(setf (max-dur self) #$NO)
(#/didChangeValueForKey: self #@"maxDur")
;; Reset a condition that indicates the min duration was reached
(#/willChangeValueForKey: self #@"minDur")
(setf (min-dur self) #$NO)
(#/didChangeValueForKey: self #@"minDur"))
(let ((duration-diff (- (loan-duration self) (list-length (pay-schedule self)))))
  (unless (or (eql (compute-mode self) 2) (zerop duration-diff))
    ;; i.e. we're not calling this function just to determine the loan duration
    ;; and we have to adjust the loan duration
    (if (plusp duration-diff)
      (progn
        ;; change the loan-duration value to what it must be
        (#/willChangeValueForKey: self #@"loanDuration")
        (setf (loan-duration self) (list-length (pay-schedule self)))
        (#/didChangeValueForKey: self #@"loanDuration")
        (when (> duration-diff 2)
          ;; If we're one-off just fix it and don't post a message
          ;; This can occur almost anytime because of numerical issues
          ;; Display a field that tells user the max duration was reached
          (#/willChangeValueForKey: self #@"maxDur")
          (setf (max-dur self) #$YES)
          (#/didChangeValueForKey: self #@"maxDur")))
      (progn
        ;; change the oan-duration value to what it must be
        (#/willChangeValueForKey: self #@"loanDuration")
        (setf (loan-duration self) (list-length (pay-schedule self)))
        (#/didChangeValueForKey: self #@"loanDuration")
        (when (< duration-diff -2)
          ;; If we're one-off just fix it and don't post a message
          ;; This can occur almost anytime because of numerical issues
          ;; Display a field that tells user the min duration was reached
          (#/willChangeValueForKey: self #@"minDur")
          (setf (min-dur self) #$YES)
          (#/didChangeValueForKey: self #@"minDur")))))))
```

This is a pretty simple function that starts with the initial loan value, calculates and adds the interest on that amount for one month, subtracts the payment to derive a new loan value and iterates until the loan value drops to zero or below. As it proceeds it keeps a record of all the payments. You may note that we put the monthly interest paid into each record twice; once as a float value that we can print directly and once as internally represented with a fixnum. The former is used in our payment schedule print function and the latter is used to compute the total interest for the interface. We previously discussed how this value is used to compute the total interest paid whenever the Total Interest Text Field asks for it.

Computing the pay schedule itself is the easy part of this function and is taken care of by the "do*" function. The more difficult aspects of this method concern the detection of conditions that require that we limit or change some of the user-specified values. Under these circumstances we need to let the user know what we did and why so they don't get frustrated by a seemingly unresponsive control.

One condition will be detected the first time through the do loop; namely whether the loan payment is less than or equal to the amount needed to pay off the first month's interest. Typically this will occur as the user increases the interest rate for a fixed loan value.If we allowed that condition to continue we would be in an infinite loop because the outstanding loan value would not decrease over time. If this case is detected, the monthly payment will be set to an amount that guarantees the loan will be paid off eventually and we set a flag indicating that we have set the minimum monthly value. After we have computed the whole schedule we will use that flag to set the condition value in the Loan object. That will, in turn, trigger the display of a banner in the user interface.

The second condition that we detect is when the loan duration specified by the user is longer than necessary to pay off the loan. This can arise as the user increases the monthly loan payment for a fixed loan value. To be as informative as possible to users, we show how the duration is modified as the monthly payment is adjusted and set a conditional value in the loan object to indicate what is being done. That value, as we have seen, is used to display an informational message in the Loan window.

The third condition is very similar to the second, but occurs when the payment is reduced to the point where the duration specified by the user is insufficient for the combination of other user-defined and computed parameters. So the code automatically increases the duration and sets the relevant condition in the loan object. This results in the display of an appropriate banner to explain what is being done.

That's all the code that is necessary. As has been typical of our projects so far, our test function is pretty simple:

```
(defun test-loan ()
  ;; up to caller to #/release the returned loan instance
  ;; but only after window is closed or crash will occur
  (make-instance 'loan))
```

Type (lnc:test-loan) in the listener to run this. Note that this function returns a loan object that has a retain count of 1, so it is the responsibility of the user to call the #/release function on this object.

Give it a try.

Challenges:

One of the things that might be useful for users is to see the whole loan payout schedule. Add a button to the window (or a menu item or both) that the user can select to open up a whole new window in which to display the loan-payout schedule. Create a separate NIB file for this window and load it on demand.

In Canada and other countries installment loan interest charges are not compounded monthly. Modify the code to allow for a different compounding schedule.

Project 7: Loan Document

Key Concepts: Documents, Document archiving, Undo manager, Printing, Open Panels

This project will be a straight-forward extension of the previous one. But we will transform loans into a standard document that can be opened, saved to a file, printed, and closed as one would expect. It will support "undo" and "redo" operations in a normal way. Most of the complications are caused by trying to add a new type of document into the existing CCL IDE without having to change it any way. You will see that for the most part we will meet this goal. The one way in which our documents will be different from others is it will not be possible to double-click on them in the finder and get them to open up in Lisp. We will, of course, be able to open them from within lisp. We could remedy this by appropriate editing of resources within the CCL IDE's bundle and also assuring that our lisp code is always loaded at runtime, but that would violate our rule for not making changes to the standard environment. The next project will be to make a stand-alone application that implements this same loan document functionality. At that time it will be possible to double-click .loan documents to open them.

The user interface for this project is basically identical to that of Project 6. We will make one small addition that will be explained when we talk about supporting "undo" and "redo" functionality. As before, either create your own project named "Loan Document" or just follow along with mine. Copy the nib file from the previous project and save it as "loandoc.nib" within the project. Open it up and add a new outlet to the "File's Owner" object (loan-controller). Name that outlet OrigDateField. Control-click and drag from the File's Owner object to the loan origination text field and link it to that new outlet. That's it for NIB file changes. Save it and we'll move on the the lisp code.

It's first necessary to understand the relationship between various classes in a normal application that supports documents. I will provide a brief introduction here, but if you want the whole story I suggest that you read Apple's "Document-Based Applications Overview". If you have the documentation sets downloaded as discussed previously you

can find this on your own system in:

file:///Developer/Documentation/DocSets/com.apple.ADC_Reference_Library.CoreReference.docset/Contents/Resources/Documents/documentation/Cocoa/Conceptual/Documents/Concepts/WindowClosingBehav.html#//apple_ref/doc/uid/20000027

Or on Apple's web-site at:

http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/Documents/Documents.html

Every document-based application will create a single instance of an NSDocumentController. The first instance created will be used as the shared instance for any later reference. Most document-based applications (including the CCL IDE) make a subclass of NSDocumentController that is unique to the application. They arrange to create an instance of it as one of the first things that the application does when it starts up. Since we do not want to interfere in any way with the functionality of that class while operating within the IDE, we will leave that class alone. When a user requests that a file be opened or that a new file be created, it is functionality within the NSDocumentController that handles that request. For the most part it knows what to do because of a resource file that is put into the application's bundle. Once again, we don't want to modify that file, so we can't directly make use of that functionality to open or create a new type of document. You will see below that I instead created a sort of pseudo document controller which does many of the same things that a normal document controller would do, but only for our new class. I tried to do that in such a way that if we decide to create a stand-alone application, we can replace our custom document controller with the more standard one and everything will work just as one would normally expect. See the next project for how that is done. We use our pseudo document controller to do a few things that we can't otherwise do and then register our documents with the standard NSDocumentController and let it do the rest of the things that it can do just fine (like saving and closing files).

The NSDocumentController instance manages instances of NSDocument. Each type of document will have its own defined subclass and we will do the same for our loan document. NSDocument objects in turn manage NSWindowController objects (one per window needed to display a single document). We will have a single loan window, so we will have only a single NSWindowController (just as we did for the previous project). Each NSWindowController will manage a single window that is defined in a NIB file (again, just as we did previously).

We will discuss the flow of control for various processes as we go through the lisp code, but for a more complete discussion, refer to the document cited previously.

For this project I refactored all the code into separate source files for each major class. The "loan-document.lisp" file contains code relevant to the loan-document class. This is basically the same as the loan class in the previous project, but now inherits from the NSDocument class and supports functionality that traditional NSDocument classes support. The "lisp-document-controller.lisp" source file supports the creation of the pseudo document controller class discussed above. As previously stated, a stand-alone application would not require this. The "loan-window-controller.lisp" source file contains code relevant to our window controller class. Finally, the "loan-print-view.lisp" file contains code for a custom view that is used for printing loan documents.

We will start with "lisp-document-controller.lisp". One design goal for this class was to make it as easy as possible to later migrate from "within-IDE" document functionality to "stand-alone" document functionality. Another was to be able to support multiple different types of documents with this one single class. To address the second goal, this class is "document-class agnostic". It does not know or care what class of document is being created. If you wanted to support two or more different types of document you could do so with this class. In a standard stand-alone document application, only a single instance of NSDocumentController is used. For our purposes we will use a single instance of the lisp-doc-controller for each *class* of document that is to be supported. For this project we will need only one instance that will be created when the loan document class is loaded.

```
(defclass lisp-doc-controller (ns:ns-object)
  ((document-class :accessor document-class :initarg :doc-class)
   (menu-class-name :accessor menu-class-name :initarg :menu-class)
   (file-ext :accessor file-ext :initarg :file-ext)
   (doc-ctrlr :accessor doc-ctrlr)
   (open-panel :accessor open-panel)
   (type-ns-str :accessor type-ns-str)
   (ext-ns-str :accessor ext-ns-str)
   (type-array :accessor type-array)
   (documents :accessor documents :initform nil))
  (:default-initargs
    :doc-class nil
    :menu-class nil
    :file-ext nil)
  (:metaclass ns:+ns-object))
```

The document-class slot will contain a reference to the class of document that will be controlled by this controller. This

will be set by the caller when the instance is created. In the same manner the caller will set the name to be used in custom menu items that we will add to do those things that cannot be done by the standard menu items. The initializing call must also provide the file extension to be used when saving or loading this type of document.

The doc-ctrlr slot contains a reference to the shared document controller instance that is used by the CCL IDE. We will register new loan documents with this shared document controller so that other menu items will work properly. As mentioned previously, one of the things that our document controller will have to manage is opening up loan documents. To do that we use an NSOpenPanel object. We only need to create one once, so that is put into the open-panel slot. At various times we need to pass instances of NSString to Objective-C functions, so we cache the strings needed in the type-ns-str, ext-ns-str, and type-array slots. The documents slot will keep track of all instances of our loan documents. This is not currently used in any way and does not delete objects when loan documents are closed. It was primarily used for early debugging. It is entirely possible that the mac-ptrs there could become invalid when a document is closed, so if you decide to modify the code to use this slot in any way, be careful.

We start to do more interesting things in an "initialize-instance :after" method:

```
(defmethod initialize-instance :after ((self lisp-doc-controller)
                                       &key menu-class file-ext &allow-other-keys)
  (ccl:terminate-when-unreachable self)
  (when menu-class
    (setf (type-ns-str self) (ccl::%make-nsstring menu-class))
    (setf (ext-ns-str self) (ccl::%make-nsstring file-ext))
    (setf (doc-ctrlr self) (#/sharedDocumentController ns:ns-document-controller))
    (setf (open-panel self) (make-instance ns:ns-open-panel))
    (#/retain (open-panel self))
    (setf (type-array self)
          (#/arrayByAddingObject: (make-instance ns:ns-array) (ext-ns-str self)))
    (make-and-install-menuitems-after "File" "New"
                                      (list (concatenate 'string "New " menu-class)
                                            "newDoc"
                                            nil
                                            self))
    (make-and-install-menuitems-after "File" "Open..."
                                      (list (concatenate 'string "Open " menu-class "...")
                                            "openDoc"
                                            nil
                                            self))
    (make-and-install-menuitems-after "File" "Print..."
                                      (list (concatenate 'string "Print " menu-class "...")
                                            (concatenate 'string "print" menu-class ":")
                                            nil
                                            nil))))
```

First we create those NSString objects described earlier. The type-ns-str for this project will contain "Loan", the ext-ns-str will contain "loan", and the type-array slot will just contain the "loan" NSString.

Next we create three new menu items. You can look at the menu-utils.lisp file to see how the make-and-install-menuitems-after function works. Basically it locates a menu item specified by the name of the top-level menu and the name of the menu item and then adds a new menu item following it. In this case we create three new menu items that will be named "New Loan", "Open Loan" and "Print Loan...". Other types of document that you might create could similarly add menu items for other types of document. In a stand-alone application we would use the standard New, Open, and Print... menu items instead. This is one of the compromises that must be made to operate within the IDE without modifying it in any way.

Note that both the "New ..." and "Open ..." send action messages directly to this instance of lisp-doc-controller. The new "Print..." menu item, however, specifies nil as the target. What this means is that the target will be the "First Responder". In this way the action message will be sent to whichever window is the top window at the time. In a document-based application, the document object is also in the first responder chain, so we can write an appropriate Objective-C method for our loan-document class that will handle the message. The action message for the print object is specialized to each type of class. In that way, if multiple document types were being supported by multiple instances of lisp-doc-controller, the print action message would be appropriate for each type document. Also recall that menu items are automatically enabled and disabled according to whether there is any object in the responder chain that will accept that action message. That means that our custom loan print menu item will only be enabled when a loan window is the active window.

Why did we not just use the action "printDocument:" as standard document applications do? The reason is that there is a default implementation of this method for all NSDocument objects, including those that represent Hemlock documents within the IDE. But the default method relies on the user overriding other print methods and the NSDocument classes defined for Hemlock windows do not do so. So if we used the "printDocument:" action our print menu-item would be enabled for all windows, but would do nothing for any but our own.

If the user should ever cause this instance of lisp-doc-controller to be garbage collected we want to deallocate those NSString objects that we created, so we define an appropriate method to do this.

```
(defmethod ccl:terminate ((self lisp-doc-controller))
  (#/release (type-ns-str self))
  (#/release (ext-ns-str self))
  (#/release (type-array self))
  (#/release (open-panel self)))
```

Next we must create the newDoc and openDoc methods that will be called when corresponding menu items are selected.

```
(objc:defmethod (#/newDoc :void)
                ((self lisp-doc-controller))
  (let ((new-doc (make-instance (document-class self))))
    (push new-doc (documents self))
    ;; register the document with the shared controller so that things like
    ;; "save" and "close" will work properly
    (#/addDocument: (doc-ctrlr self) new-doc)
    (#/makeWindowControllers new-doc)
    (#/showWindows new-doc)))
```

This method just mimics what a real NSDocumentController would do when a New menu item is selected. It makes an instance of the new document and registers it with the "real" NSDocumentController. Then it calls the same methods that an NSDocumentController would to cause the document to make its window controllers and then show those windows.

```
(objc:defmethod (#/openDoc :void)
                ((self lisp-doc-controller))
  (let ((result (#/runModalForTypes: (open-panel self) (type-array self))))
    (when (eql result 1)
      (let ((urls (#/URLs (open-panel self))))
        (dotimes (i (#/count urls))
          (let ((doc (make-instance (document-class self))))
            (setf doc (#/initWithContentsOfURL:ofType:error:
                       doc
                       (#/objectAtIndex: urls i)
                       (type-ns-str self)
                       (%null-ptr)))
            (if doc
              (progn
                (pushnew doc (documents self))
                (#/addDocument: (doc-ctrlr self) doc)
                (#/makeWindowControllers doc)
                (#/showWindows doc))
              (#_NSRunAlertPanel #@"ALERT"
                                 #@"Could not open specified file ... ignoring it."
                                 #@"OK"
                                 (%null-ptr)
                                 (%null-ptr)))))))))
```

The openDoc method also mimics what a real NSDocumentController object would do. It runs the openPanel in such a way that it will only permit the selection of files with a .loan extension (or whatever extension was specified when the lisp-doc-controller was created). The returned value will be 1 if the user selected a file or 0 if the selection was cancelled. In theory a user can select multiple files to open, although the default values for the open panel permit only single files. But to be on the safe side we will iterate through the set of files selected and create a new document for each one. Note that we first create an instance of our class and then call the standard init method initWithContentsOfURL:ofType:error:. Since init methods need not return the same object that was allocated, we need to do the setf to make sure we use whatever object is returned by the init method. We do not need to worry about what it does. When we look at loan-document methods we will see what must be done to make sure that this is done correctly.

If the init should fail we run an alert panel with a single OK button that simply indicates that the file could not be open. If we wanted to be more precise about why it couldn't be open, we could modify the init call to provide a ptr to an error object that would be created by the init function. Then we could use that error object to alert the user about what happened.

That's all the functionality required for our lisp-doc-controller class. Since the changes to the window controller class defined in the previous project are fairly minimal, we will look at that next. First note that to avoid confusion I changed the name of the class from "loan-controller" in the previous project to "loan-win-controller" for this one. It would not have been necessary to do this since they are safely interned within separate packages, but I wanted to avoid any confusion.

The new loan-win-controller class definition has only one small modification from the loan-controller class in the previous project. We add a single slot:

```
  (orig-date-text :foreign-type :id :accessor orig-date-text)
```

This will keep a reference to the origination date text field. You will recall that we set this link up when we modified the nib file.

The only other changes are in the #/wakeFromNib method which now looks like:

```
(objc:defmethod (#/awakeFromNib :void)
                ((self loan-win-controller))
  (#/setEnabled: (loan-text self) #$NO)
  ;; set the sliders to update continuously so that the text boxes reflect the current value
  ;; Note that we can set this in IB for text boxes, but not, apparently, for sliders
  (#/setContinuous: (int-slider self) #$YES)
  (#/setContinuous: (dur-slider self) #$YES)
  (#/setContinuous: (pay-slider self) #$YES)
  ;; tell the text cells not to handle undo
  (#/setAllowsUndo: (#/cell (loan-text self)) #$NO)
  (#/setAllowsUndo: (#/cell (int-text self)) #$NO)
  (#/setAllowsUndo: (#/cell (dur-text self)) #$NO)
  (#/setAllowsUndo: (#/cell (pay-text self)) #$NO)
  (#/setAllowsUndo: (#/cell (orig-date-text self)) #$NO))
```

We called #/setAllowsUndo: with an argument of #$NO for the cell sub-object of each of the editable text fields. When we discuss the undo functionality provided by our loan-document class we will see how undo is handled. What we are doing here is disabling the default undo methods that come with all NSTextFieldCell objects. Basically, these handle the undo and redo for text editing operations. So you would see something like "Undo Typing" in the Edit menu. But the undo/redo semantics for our Loan objects are a bit more subtle than that and we don't want these default undo operations to interfere with what we want to do. This is another one of those attributes of objects that arguably should be settable within IB, but isn't.

Those are all of the changes needed for our loan-win-controller, so we will next discuss the loan-document class. The first thing needed in the loan-doc.lisp file are the appropriate "require" statements.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :menu-utils)
  (require :decimal)
  (require :date)
  (require :lisp-doc-controller)
  (require :loan-win-cntrl)
  (require :loan-pr-view)
  (require :ns-string-utils))
```

You have seen most of these previously. The loan-print-view functionality will be discussed later. The ns-string-utils.lisp file contains a number of useful routines that we will examine quickly here.

```
(defun ns-to-lisp-string (ns-str)
  (if (plusp (#/length ns-str))
    (%get-cstring (#/cStringUsingEncoding: ns-str #$NSUTF8StringEncoding))
    ""))
```

This converts an NSString object to a lisp string. This is occasionally useful when the Objective-C bridge functions do not automatically do the conversion for you.

```
(defun lisp-str-to-ns-data (lisp-str)
  (with-cstrs ((str lisp-str))
    (#/dataWithBytes:length: ns:ns-data str (1+ (length lisp-str)))))
```

This function converts a lisp string to an NSData object. This will be useful when we need to pass an object that represents the state of our loan objects to an Objective-C method. We will basically put a lisp form into a string and then encode the string into an NSData object that gets passed. We will do that both when we save a file and when we specify how to undo an operation.

```
(defun ns-data-to-lisp-str (nsdata)
  (%get-cstring (#/bytes nsdata)))
```

This function reverses the process and creates a lisp string from an NSData object.

```
(defun lisp-object-to-ns-data (obj)
  (lisp-str-to-ns-data (format nil "~s" obj)))
```

Completing the encoding process, this function creates a string that represents a lisp object. For this project that will be a simple list of values.

```
(defun ns-data-to-lisp-object (nsdata)
  (read-from-string (ns-data-to-lisp-str nsdata)))
```

And completing the decoding process, this function reads the lisp object from the string retrieved from the NSData object.

```
(defun lisp-to-temp-nsstring (string)
  ;; creates a string that is not owned by caller
  ;; so no release is necessary
  (with-encoded-cstrs :utf-8 ((s string))
    (#/stringWithUTF8String: ns:ns-string s)))
```

This is a function that I should have created for previous projects. In several of those projects we created NSStrings that were used as arguments to Objective-C functions and then #/released them before exiting the function. Here we use one of the class convenience functions described earlier which create an object of the desired type (here an NSString) that we do not own and therefore do not need to release later.

The loan-doc class definition is identical to the definition of the loan class in the previous project with the exception of a single new slot:

```
 (last-set-param :accessor last-set-param :initform nil)
```

This slot will be used in the implementation of the "undo" functionality and will be discussed later.

Once the loan-doc class is defined we create an instance of the lisp-doc-controller class that will be used to manage all loan-doc instances.

```
(defvar *loan-doc-controller* (make-instance 'lisp-doc-controller
                                :doc-class (find-class 'loan-doc)
                                :menu-class "Loan"
                                :file-ext "loan"))
```

We set the class to be our loan-doc class, the menu text to be "Loan" and the file extension of be "loan" (note that the "." is not needed as part of the extension).

All of the functionality used to compute the values of loan parameters is exactly the same as the previous project and will not be further discussed here. The remainder of this discussion will focus on new functionality added to the loan-doc class for window-controller management, undo/redo, open/save, and printing operations.

The first method we will discuss is #/makeWindowControllers.

```
(objc:defmethod (#/makeWindowControllers :void)
                ((self loan-doc))
  (let ((lwc (make-instance 'loan-win-controller
               :with-loan self)))
    (#/setShouldCloseDocument: lwc #$YES)
    (#/addWindowController: self lwc)))
```

This method will be invoked by the lisp-doc-controller object that we just created whenever a user elects to create a new loan document. This is a standard NSDocument method that is typically defined by a subclass in order to create multiple windows for a single document and is normally called by the shared document controller. When just a single window is needed, a standard document subclass would normally override #windowNibName to specify which nib file from within the application's bundle should be used to create that window. In our case, the nib file that we want to use is outside of the CCL IDE's bundle, so we need to do something a bit different. As we did in the previous project, we invoke the #'initWithLoan method that we defined for the loan-win-controller class. In addition we tell the window controller that when the window closes it should also close the document. Finally we add the window controller to ourself. I'm not entirely sure what all this accomplishes, but the Apple documentation is quite clear that this is something that should be done if you override this method.

Note that the window controller will now have two slots that point to our loan-doc object: document and loan. We do not necessarily need both. The document slot is necessary because it is used for standard NSWindowController functionality, so we could change all of the KVC paths used to bind user interface objects to use the document slot rather than the loan slot and do away with the latter altogether. Part of the reason I didn't do this is just laziness. I didn't want to go back into IB and change all of the binding paths. But as I considered doing this I also realized that the document slot in the lisp-window-controller would not be set until sometime after the nib file was loaded whereas the loan slot was set prior to this event. IB provides a way to specify default values to use if a key-path returns a null value and presumably I would have needed to set all of these as well to prevent problems when the newly created interface objects began to ask for values via a slot that was not yet defined. Rather than walk down that path I chose to take the easy route and have two different slots.

Undo/Redo Functionality

Before examining the next loan-doc methods we need to discuss how "undo" and "redo" normally work and how we want to implement them for our loan documents. Apples' functionality is really quite clever. Each document has its own NSUndoManager instance that coordinates undo and redo operations whenever a window owned by that document has the current focus. The way this works is that whenever an operation that we want to undo is done, it creates the invocation that would be needed to reverse the action being taken and registers that action with the undo manager. That action should itself be "undoable". Assuming that the reverse action is in fact undoable, then when it is invoked it will create and register its own reverse action. The undo manager knows that it is in the process of doing an undo when this happens and puts the reverse action on the "redo" stack instead of on the "undo" stack. All of this seems fairly straight-forward, but we need to think about how it applies to our loan application.

The first problem that we have applying undo functionality to this application is that setting a new value will always result in new values for other parameters as well as the one changed. At the very least the dependent variable being computed is changed and frequently others are as well. So just changing the value back to what it was will not always result in an overall state that was the same as it was previously. What we really need to do is to capture the entire state of the loan just before we change something and restore that state to undo the effects of the change.

The second problem is how to manage the size of the undo stack. Recall that we made all the slider and text-box updates continuous to provide instantaneous feedback to the user about the effects that are generated by the change. If we regarded each of these individual changes as an action that could be undone, then there would be a huge number of actions on the undo stack and that would make the undo functionality virtually unusable. To address this problem we will treat a sequence of changes to any single parameter as if it were a single un-doable action. So we can move a slider back and forth as much as we want and then undo to set the state of the loan back to what it was before we started to move it.

Note that changing a slider and then subsequently changing the text field that sets the same parameter will be treated as a single action. Arguably you might want to treat these as separate actions. That could be done by creating a different bind target for each control and then in the corresponding Objective-C methods each could update the same slot and notify the other that the slot had changed. Feel free to make this change if you want.

We will encapsulate the state of a loan as a simple list that contains the relevant parameter slot-values. The reverse action for every change we make will be to set the state of the loan back to what it was, as indicated by the list of slot-values. When a reverse action is first created it will capture the current state of the loan as a list of parameters and give it to the undo manager as an NSData object that should be passed back as a parameter to the undo method that we will name "setLoanState:". This method will, of course, register its own reverse action with the undo manager; that being simply to set the state back to its current setting. Let's now look at the methods that are used to do all this.

```
(defmethod create-undo ((self loan-doc) undo-name &optional (always-undo nil))
  (when (or always-undo (not (eq (last-set-param self) undo-name)))
    (let ((undo (#/undoManager self))
          (st (lisp-object-to-ns-data (get-loan-state self))))
      (#/setLoanState: (#/prepareWithInvocationTarget: undo self)
```

```
                    st)
      (unless (#/isUndoing undo)
        (#/setActionName: undo undo-name))
      (setf (last-set-param self) undo-name))))
```

The create-undo method is called as the first thing from every action that changes a loan parameter value. It is here that we note whether the action being done is the same as the previous action or something new. The last-set-param slot is used to track the las action. If it is the same, then we do not create and register a new reverse action and simply return. The last one that we registered will suffice to return the state to what is was previously. Otherwise we prepare the undo manager by calling #/prepareWithInvocationTarget: . This alerts it to accept the next message that is sent to it and package it up to be sent later if the user elects to undo. The second argument to the #/prepareWithInvocationTarget: method tells the undo manager where to send the message to cause the undo. In this case, it is simply sent back to this instance. We register the #setLoanState method as our undo method by calling it with the undo manager as its first argument. The undo manager will package up the message and any parameters sent along (the st NSData object in this case) and save them for use when an undo is invoked. At that time it will send the message to the target object (which we set as self when we prepared the undo manager) along with the st parameter. After we register the undo message we set up the string that will be shown in the undo menu item to make it easier for the user to know exactly what is being undone. We only need to do that if we are not in the process of undoing something. In that case the undo manager will take the title from the current undo, which is exactly what we want to happen.

Note that the reverse of a #/setLoanState action is another #/setLoanState action and in this case we always want to create an appropriate reverse. So create-undo provides an optional parameter that lets us specify that an undo should be registered even if the previous action was the same.

```
(objc:defmethod (#/setLoanState: :void)
                ((self loan-doc) (st :id))
  (create-undo self nil t)
  ;; called when user does an "undo"
  (set-loan-state self (ns-data-to-lisp-object st)))
```

This is the method that is actually called by the undo manager whenever the user elects to undo or redo something. It first creates its own undo and makes sure that it is always created by calling create-undo with the always-undo parameter set to t. Then it calls the lisp set-loan-state method to do the real work using the parameter list recovered from the NSData object.

```
(defmethod get-loan-state ((self loan-doc))
  ;; returns a list of loan state values suitable for use by set-loan-state
  (list (loan-amount self)
        (interest-rate self)
        (loan-duration self)
        (monthly-payment self)
        (origination-date self)
        (first-payment self)
        (max-dur self)
        (min-dur self)
        (min-pay self)))
```

The get-loan-state method simply gathers up the slots that we need to reproduce a loan's state and puts them into a list.

```
(defmethod set-loan-state ((self loan-doc) state-list)
  (setf (last-set-param self) nil)
  (#/willChangeValueForKey: self #@"loanAmt")
  (setf (loan-amount self) (pop state-list))
  (#/didChangeValueForKey: self #@"loanAmt")
  (#/willChangeValueForKey: self #@"interestRate")
  (setf (interest-rate self) (pop state-list))
  (#/didChangeValueForKey: self #@"interestRate")
  (#/willChangeValueForKey: self #@"loanDuration")
  (setf  (loan-duration self) (pop state-list))
  (#/didChangeValueForKey: self #@"loanDuration")
  (#/willChangeValueForKey: self #@"monthlyPayment")
  (setf   (monthly-payment self) (pop state-list))
  (#/didChangeValueForKey: self #@"monthlyPayment")
  (#/willChangeValueForKey: self #@"originationDate")
  (setf  (origination-date self) (pop state-list))
```

```
  (#/didChangeValueForKey: self #@"originationDate")
  (#/willChangeValueForKey: self #@"firstPayment")
  (setf  (first-payment self) (pop state-list))
  (#/didChangeValueForKey: self #@"firstPayment")
  (#/willChangeValueForKey: self #@"maxDur")
  (setf   (max-dur self) (pop state-list))
  (#/didChangeValueForKey: self #@"maxDur")
  (#/willChangeValueForKey: self #@"minDur")
  (setf   (min-dur self) (pop state-list))
  (#/didChangeValueForKey: self #@"minDur")
  (#/willChangeValueForKey: self #@"minPay")
  (setf   (min-pay self) (pop state-list))
  (#/didChangeValueForKey: self #@"minPay")
  (#/willChangeValueForKey: self #@"totInterest")
  (setf   (pay-schedule self) nil)
  (#/didChangeValueForKey: self #@"totInterest")
  (compute-new-loan-values self))
```

The set-loan-state method first sets the last-set-param slot to nil, thus assuring that a subsequent change to any parameter value will initiate the generation of a new undo entry. Then it takes a list of loan parameters and uses them to set the corresponding loan values. Finally it calls the compute-new-loan-value method to compute the new loan-schedule slot. Note that it makes sure to notify KVC that the corresponding key paths have been changed so that any user interface objects that are bound to them will be changed as well. Also note that any argument given to set-loan-state is used exactly once (for a couple of reasons, not the least of which is that it is newly created from an NSData object). So there is no problem with destroying it using the pop calls.

To complete the undo functionality we must assure that every method that initiates a change to the loan state calls the create-undo method as the first thing that it does. For example:

```
(objc:defmethod (#/setLoanAmt: :void)
                ((self loan-doc) (amt :id))
  (create-undo self #@"set loan amount")
  (setf (loan-amount self) (lisp-from-ns-decimal amt))
  (compute-new-loan-values self))
```

This method is identical to its counterpart in the previous project except for an additional call to create-undo that occurs at the beginning of the method. The string provided would result in the undo menu showing "Undo set loan amount". Similar changes must be made to all of the #/set... methods.

Open/Save Functionality

The next functionality that we will implement for our loan-document class is support for opening and saving files. First let's talk about the "save" menu item. Typical save functionality is to enable that menu item when a document has been modified and disable it when it has not. The NSUndoManager tracks this and can manage this for normal documents. It is not entirely clear to me what the CCL IDE does to implement save, but the default seems to do some fairly strange things. For normal lisp documents it appears to always be enabled, even when the document is unmodified (for example right after we save it). And for custom documents such as ours it often seems to be disabled even when it is modified. A little experimentation showed that the default save functionality worked just fine even when the menu item was disabled, so the solution was to simply validate the menu item ourselves. The following function does that.

```
(objc:defmethod (#/validateMenuItem: #>BOOL)
                ((self loan-doc) (item :id))
  (let* ((action (#/action item)))
    (cond ((eql action (ccl::@selector #/saveDocument:))
           (#/isDocumentEdited self))
          (t (call-next-method item)))))
```

This is a standard method that is called on the first responder for every menu item when a menu is opened. Since our loan-doc instance is in the First Responder chain we can implement it for the loan-doc class and simply enable the menu item whenever our document has been edited and disable it otherwise. This works just like a normal application. When some other type of window has the focus the no loan-doc instance will be in the first responder chain and normal validation of the save menu-item will be done.

When an application is saved we want to force it to be saved with a ".loan" extension. To do that we define the method below.

```
(objc:defmethod (#/prepareSavePanel: #>BOOL)
                ((self loan-doc) (panel :id))
  (#/setRequiredFileType: panel #@"loan")
  #$YES)
```

This method is called just before the save panel is displayed to a user. We simply set the required file type and it will automatically be filled in for the user as part of the file name.

There are basically three ways that documents can participate in saving and loading documents. In each case a pair of methods must be overridden by the document. Which one is selected depends on how involved the document wants to be in the process. The easiest approach is to just worry about what data needs to be loaded or saved and let the NSDocument default methods worry about the rest. This is the approach taken here. To do this we will override the #/readFromData:ofType:error: and #/dataOfType:error: methods. Consult the Document-Based Application Architecture document described previously for other ways to implement document saving and loading functionality.

```
(objc:defmethod (#/dataOfType:error: :id)
                ((self loan-doc) (dtype :id) (err (:* :id)))
  (declare (ignore dtype err))
  (lisp-object-to-ns-data (get-loan-state self)))
```

This method is called when a user elects to save a file. Having implemented the undo functionality, this is now a pretty easy thing to do. We get the loan state and package it up as an NSData object. Recall that when we discussed the lisp-object-to-ns-data function we explained that it creates a string that represents a lisp object that is then used to create the NSData object. What happens to it after that is not something that we have to concern ourselves with too much. As long as we are given the same object back when that file is opened, we will be just fine. In fact, it turns out that for NSData objects such as this the output file will simply contain the text that we gave it. It is possible to use TextEdit or most any other text editing application (including CCL itself) to open it up and look at it.

If saving and restoring our document required more complex and/or interconnected lisp objects, then we would certainly want to do something other than create a simple list. In this case it might be appropriate to use lisp's make-load-form to generate a set of loadable forms that could be used when the file was opened. Perhaps in a later project I will do something like that.

```
(objc:defmethod (#/readFromData:ofType:error: #>BOOL)
                ((self loan-doc) (data :id) (dtype :id) (err (:* :id)))
  (declare (ignore err dtype))
  (set-loan-state self (ns-data-to-lisp-object data))
  #$YES)
```

This method is invoked when a loan file is opened. It simply converts the NSData object back to a lisp list and uses it to reassign data values to the document's slots, much as we did for undo actions.

And that's it! You can now easily save and load .loan documents.

Print Functionality

The last of the new document functions we will discuss is printing. In a typical document-based application the selection of the print menu item results in sending a "printDcoument:" message to the first responder which ultimately reaches the document being displayed in the active window. In non-document-based applications the print menu item will typically send a "print:" message to the first responder. All NSTextFields will respond to a print: message by printing themselves. The print menu item in the CCL IDE uses the "print:" method rather than :printDocument. This works just fine when all the windows have a single NSTextField which can then print itself. But this is clearly not going to work for our loan window. Whichever text field happened to be the first responder at the time would simply print itself. I pondered various ways that I might alter those individual print: methods to change that behavior, but in the end decided that the easiest solution was simply to create a "Print Loan..." menu item that does what we want it to do. I decided not to just make it call #/printDocument: because it would then be applicable to all document objects and those defined for the CCL IDE had no functionality to support printing. So I set up the menu item to create a unique message for the loan class that only a loan-doc would respond to. Therefore, when a window for a document other than a loan-doc was active, the Print Loan... menu item would be disabled.

```
(objc:defmethod (#/printLoan: :void)
                ((self loan-doc) (sender :id))
  (declare (ignore sender))
  (#/printDocument: self self))
```

The method above is invoked when the Print Loan... menu item is selected. You will recall that we set up that menu-item when the lisp-doc-controller was initialized. This method, in turn, calls the standard #/printDocument: method on itself. That will make life easier when we create a stand-alone loan application in the next project. The #/printDocument: method will in turn invoke the `#/printOperationWithSettings:error:` method shown below.

```
(objc:defmethod (#/printOperationWithSettings:error: :id)
                ((self loan-doc) (settings :id) (err (:* :id)))
  (declare (ignore err settings))
  (let ((pr-view (make-instance 'loan-print-view
                   :loan self)))
    (#/printOperationWithView:printInfo: ns:ns-print-operation
                                         pr-view
                                         (#/printInfo self))))
```

This method returns an NSPrintOperation object which controls printing operations. We create it by specifying the view that will get drawn and providing a NSPrintInfo object. The latter is an object that contains information about things like paper size, number of copies, print margins, etc. Although it is possible to create a custom version of this object, we simply use the default version that is shared by all documents. As far as the loan-doc object is concerned, this is all that it needs to do to provide printing. The NSView object that we provide will be an instance of our custom loan-print-view class. That is defined within the loan-print-view.lisp source file that we will examine right after we talk generally about printing.

The printing support within Cocoa makes it quite easy to print an existing view. Basically you can hand the view to the printing subsystem and it will arrange for the view to draw itself on a page rather than on a screen. If what is in your screen view exactly matches what you want to print and fits onto a single page (or can be automatically divided into pages), then life is pretty easy. However if your view is much larger than a single page and programmatic control is needed to decide how pages should be composed or you want to print data with a different format than is used on the screen, then some code is needed to make that happen. To demonstrate how to do this in lisp we will print a loan document with all the basic parameter data at the top of the first page and with details about the payout schedule printed one line per month at the bottom of the first page and onto subsequent pages.

Cocoa printing functions are designed to handle views that are larger than a single page. By default, Cocoa's printing facility will try to map the view that you give to it onto multiple pages. For things like text documents that default works quite well. For many other applications the developer will want to specify how that mapping occurs. Cocoa provides a method for the application to specify which rectangle within the view corresponds to each page number. We will see a bit later how we take advantage of that mechanism to do something a bit different. Once you specify which rectangle within the view is to be printed for a given page, the printing functionality will try to map that rectangle onto the available space on a page. By default it positions it at the upper left corner of the page. For our purposes this will work just fine, but be aware that there are ways that this can be modified by the programmer. A view to be drawn can also be clipped or scaled in either the horizontal or vertical direction or both. For more information about how to implement printing see "Printing Programming Topics for Cocoa":

http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/Printing/Printing.html

Our loan-print-view class is entirely new, so we will examine it and its methods in some detail. The first lines you will see in the file loan-pr-view.lisp specify some required files.

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :date)
  (require :nslog-utils))
```

You probably looked at the date.lisp utility file when we used it earlier. We will use another function from that file here to generate an appropriate string to print.

The nslog-utils.lisp utility file is worth a look because of its use in debugging. In fact, these functions are not actually used in the final code. I left calls to them in the code as comments so that you can see how I went about seeing what was going on. For a normal lisp program, the programmer might find it useful to insert format statements to print out significant values while a function is running. Unfortunately, you can get some pretty strange results if you try to do that within an Objective-C function that is called in response to some event. I crashed CCL more than once before I got it through my head that this just wasn't a very smart thing to do. There is, however, a way to do something equivalent and that is to invoke the same #_NSLog function that Objective-C programmers do in their code. This will create an entry into the console log that can be easily viewed using Apple's Console application. I added some utility functions that make calling this a bit easier. These are largely self-explanatory, so I show them here without further explanation. I encourage you to create your own versions of these for other sorts of objects.

```
(defun log-rect (r &optional (log-string ""))
  (#_NSLog (lisp-to-temp-nsstring (concatenate 'string
                                               log-string
                                               "x = %F y = %F width = %F height = %F"))
           #>CGFloat (ns:ns-rect-x r)
           #>CGFloat (ns:ns-rect-y r)
           #>CGFloat (ns:ns-rect-width r)
           #>CGFloat (ns:ns-rect-height r)))

(defun log-size (s &optional (log-string ""))
  (#_NSLog (lisp-to-temp-nsstring (concatenate 'string
                                               log-string
                                               "width = %F height = %F"))
           #>CGFloat (ns:ns-size-width s)
           #>CGFloat (ns:ns-size-height s)))

(defun log-float (f &optional (log-string ""))
  (#_NSLog (lisp-to-temp-nsstring (concatenate 'string
                                               log-string
                                               "%F"))
           #>CGFloat f))

(defun interleave (l1 l2)
  (let ((lst1 (if (listp l1) l1 (list l1)))
        (lst2 (if (listp l2) l2 (list l2))))
    (if (atom l1)
      (setf (cdr lst1) lst1)
      (if (atom l2)
        (setf (cdr lst2) lst2)))
    (mapcan #'(lambda (el1 el2)
                (list el1 el2))
            lst1
            lst2)))

(defun log-4floats (f1 f2 f3 f4 &optional (log-strings '("" "" "" "")))
  (#_NSLog (lisp-to-temp-nsstring (apply #'concatenate 'string
                                         (interleave log-strings "%F ")))
           #>CGFloat f1
           #>CGFloat f2
           #>CGFloat f3
           #>CGFloat f4))
```

Now we'll continue with the loan-print-view.lisp file.

```
(defclass loan-print-view (ns:ns-view)
  ((loan :accessor loan
         :initarg :loan)
   (attributes :accessor attributes
               :initform (make-instance ns:ns-mutable-dictionary))
   (page-line-count :accessor page-line-count
                    :initform 0)
   (line-height :accessor line-height)
   (page-num :accessor page-num
             :initform 0)
   (page-rect :accessor page-rect))
  (:metaclass ns:+ns-object))
```

Since we are not going to print anything that looks very much like what is displayed in our window, we will define a brand new view that we will give to the printing functions. We saw previously that an instance of this view was created in the loan-doc object's #/printOperationWithSettings:error: method. This view class contains a slot that will link to the loan object so that we can retrieve the data to be printed. It also has slots that contain various values and objects necessary to printing. Each of these will be discussed as they are used.

```
(defmethod initialize-instance :after ((self loan-print-view)
                                       &key &allow-other-keys)
```

```
  ;; assure loan still exists if user closes the window while we are printing
  (#/retain (loan self))
  (ccl:terminate-when-unreachable self)
  (let* ((font (#/fontWithName:size: ns:ns-font #@"Courier" 8.0)))
    (setf (line-height self) (* (+ (#/ascender font) (abs (#/descender font))) 1.5))
    (#/setObject:forKey: (attributes self) font #$NSFontAttributeName)))
```

When initializing the loan-print-view we first make sure that the loan-doc object won't go away if the user decides to print a loan and then close the loan-doc window before printing has been completed. This is done using a #/retain call on the loan-doc. When this object is reclaimed by garbage collection we will do the corresponding #/release.

Next the function specifies a font to use. I picked a fixed-width font to make things line up on the page a bit more uniformly and a size that makes lines fit well, but feel free to experiment to find choices that you like better. Next we set the line-height that should be used for each line based on the font. There are possibly more precise ways to do this, but they seemed to involve more futzing around than I wanted to do and the technique shown here seemed to work quite nicely. The only attribute that we will set for the text that we will be printing is the font. If you want to make something bold or italic or otherwise modify the attributes, this might be a good place to do that.

```
(defmethod ccl:terminate ((self loan-print-view))
  (#/release (loan self))
  (#/release (attributes self)))
```

As previously noted, the ccl:terminate method will release objects that it owns.

```
(objc:defmethod (#/knowsPageRange: :<BOOL>)
                ((self loan-print-view) (range (:* #>NSRange)))
  ;; compute printing parameters and set the range
  (let* ((pr-op (#/currentOperation ns:ns-print-operation))
         (pr-info (#/printInfo pr-op))
         ;; (pg-size (#/paperSize pr-info))
         ;; (left-margin (#/leftMargin pr-info))
         ;; (right-margin (#/rightMargin pr-info))
         ;; (top-margin (#/topMargin pr-info))
         ;; (bottom-margin (#/bottomMargin pr-info))
         (image-rect (#/imageablePageBounds pr-info))
         (pg-rect (ns:make-ns-rect 0
                                   0
                                   (ns:ns-rect-width image-rect)
                                   (ns:ns-rect-height image-rect))))
    ;; (log-size pg-size "pg-size: ")
    ;; (log-4floats left-margin right-margin top-margin bottom-margin
    ;;              (list "Margins: left = " " right = " " top = " " bottom = "))
    ;; (log-rect image-rect "imageable rect: ")
    (setf (page-rect self) pg-rect)
    ;; (log-rect pg-rect "my page rect: ")
    (#/setFrame: self pg-rect)
    ;; (log-float (line-height self) "Line Height: ")
    (setf (page-line-count self) (floor (ns:ns-rect-height pg-rect)
                                        (line-height self)))
    ;; start on page 1
    (setf (ns:ns-range-location range) 1)
    ;; compute the number of pages for 9 header lines on page 1 and 2 header
    ;; lines on subsequet pages plus a line per payment
    (let* ((pay-lines-on-p-1 (- (page-line-count self) 7))
           (other-pages-needed (ceiling (max 0 (- (list-length (pay-schedule (loan self)))
                                                  pay-lines-on-p-1))
                                        (page-line-count self))))
      (setf (ns:ns-range-length range)
            (1+ other-pages-needed))))
  #$YES)
```

The #/knowsPageRange: method is called to let the user do completely custom pagination if desired. If this method returns #$YES, then the view will subsequently be asked for the rectangle to use for each page via calls to the #/rectForPage method. As a side-effect, this method must set the page range in the range parameter that is passed in.

I did quite a bit of experimenting to figure out the relationship between various rectangles and things like margins. Unfortunately that was a (rather rare) necessity because I was not able to get a very clear picture from any of Apple's documentation. You will see vestiges of my search included as comments in the source code. I decided to leave all of them there so that I could discuss what I found out and perhaps help others to understand a bit better. #/paperSize simply returns the exact paper size available in points in the user space. So for my 8.5" x 11" paper it returned a size structure with the width = 612.000000 height = 792.000000. This is exactly what would be expected at 72 points per inch, which is what all of Apple's drawing assumes. Note that always having 72 points per inch doesn't pose a drawing limitation of any kind because coordinates can be specified as fractions of points.

The left and right margins both returned 72 points (i.e. 1") and both top and bottom margins returned 90 points (i.e. 1.25"). As near as I can tell, the default print functions do nothing with these values. I believe that they are there only to provide information for developers who want to lay out their own pages. The way that you would use these is described in the next paragraph.

The #/imageablePageBounds call resulted in

imageable rect: x = 18.000000 y = 40.000000 width = 576.000000 height = 734.000000

This turned out to be the most useful value to have as it precisely specifies the maximum width and height of a view rectangle that will fit onto a page. The x and y values specify where the user view will be placed on the page. You don't need to worry about that unless you want your drawing routine to honor the specified margins. If so, then the drawing routine would need to know both the requested margin, as returned by the #/...Margin calls and the minimal margin that you will get regardless, as specified by the x and y values returned by the #/imageablePageBounds call. It would have to subtract the minimal margin values from the requested margin values to derive an additional amount of margin that the drawing routine should provide in order to see the requested margins on the printed page. Then rectangles used within the drawing routine would have to be positioned accordingly. I did not do that for this project, which results in printing that is rather close to the edges of the pages. Feel free to adjust the printing routines to respect requested margins.

For this project both the page-rect slot and the frame rect of the view are set to a rectangle positioned at 0,0 with the same bounds as the imageable page. This is the maximum space that can be printed by the printer. We will see how this is used below. The page-line-count slot is set to a computed value that is the number of lines that can be fitted into the space available on the imageable page. Finally, to determine the number of pages that will be needed we first determine the total number of loan schedule lines that can be printed on the first page (i.e. after allowing for the fixed information at the top of the page) and then divide the number of additional lines needed by the number of lines per page. We set the range parameter to be the total number of pages required.

```
(objc:defmethod (#/rectForPage: #>NSRect)
                ((self loan-print-view) (pg #>NSInteger))
  (setf (page-num self) (1- pg))
  (page-rect self))
```

The #/rectForPage: method is called to get a rectangle for a particular page number. Page numbers start with 1, so we subtract 1 to make it 0-relative and set the page-num slot, which will be used by the drawing routine to decide what to print. The rectangle returned is always the same, namely the one that we previously computed and saved which is positioned at 0,0 and has the size of the imageable space on a printer page.

```
(objc:defmethod (#/isFlipped #>BOOL)
                ((self loan-print-view))
  ;; we compute coords from upper left
  #$YES)
```

Normally coordinates in views have the origin at the lower left corner. For our purposes computation is a little easier if we have the origin in the upper left, so the #/isFlipped method is overridden to return #$YES. The Cocoa drawing environment makes several changes to accommodate this, including changes to how fonts are drawn so that they have the correct orientation. Note that this also affects how rectangles are specified, namely the position specified now refers to the upper left corner rather than the lower left corner.

```
(objc:defmethod (#/drawRect: :void)
                ((self loan-print-view) (r #>NSRect))
  (with-slots (loan attributes page-line-count line-height page-num page-rect) self
    (ns:with-ns-rect (line-rect (ns:ns-rect-x r)
                                (- (ns:ns-rect-y r) line-height)
                                (ns:ns-rect-width r)
                                line-height)

      ;; (log-rect r "draw rect: ")
```

```
 (labels ((draw-next-line (str)
            (incf (ns:ns-rect-y line-rect) line-height)
            (#/drawInRect:withAttributes:
             (lisp-to-temp-nsstring str)
             line-rect
             attributes))
          (draw-next-payment (sched-line)
            (draw-next-line
             (format nil
"~1{On ~a balance = $~$ + interest of $~$ - payment of $~$ = ~a balance of $~$~}"
                     sched-line))))
   (when (zerop page-num)
     ;; print all the basic loan info
     (draw-next-line (format nil
                             "Loan ID: ~a"
                             (ns-to-lisp-string (#/displayName loan))))
     (draw-next-line (format nil
                             "Amount: $~$"
                             (/ (loan-amount loan) 100)))
     (draw-next-line (format nil
                             "Origination Date: ~a"
                             (date-string (origination-date loan))))
     (draw-next-line (format nil
                             "Annual Interest Rate: ~7,4F%"
                             (* 100 (interest-rate loan))))
     (draw-next-line (format nil
                             "Loan Duration: ~D month~:P"
                             (loan-duration loan)))
     (draw-next-line (format nil
                             "Monthly Payment: $~$"
                             (/ (monthly-payment loan) 100)))
     ;; draw spacer line
     (incf (ns:ns-rect-y line-rect) line-height))
   ;; print the appropriate schedule lines for this page
   (let* ((lines-per-page (- page-line-count (if (zerop page-num) 7 0)))
          (start-indx (if (zerop page-num) 0 (+ (- page-line-count 7)
                                                (* lines-per-page (1- page-num)))))
          (end-indx (min (length (pay-schedule loan))
                         (+ start-indx lines-per-page 1))))
     (dolist (sched-line (subseq (pay-schedule loan) start-indx end-indx))
       (draw-next-payment sched-line)))))))
```

The #/drawRect: method is where things really happen. The rectangle passed into the routine tells the view where to draw. In our case that rectangle will always be the same as the one that we provided in the #/rectForPage method, but that isn't necessarily always the case. If, for example, we had provided a rectangle that was larger than the imageable area, then the Cocoa printing functions would have decided whether to clip or scale the page according to parameters set in the NSPrintInfo object that we provided. If the rectangle was to be clipped, then the rectangle passed to this method could have been smaller than the one we provided in the #/rectForPage method. So its probably always a good idea to make use of the rectangle parameter rather than the one that you expect to get.

This method makes use of the ability of an NSString to draw itself within a specified rectangle using specified attributes. Basically what we do here is create such a rectangle at the top of the page that spans the width available and as each line is printed we move it down the page. We move the rectangle down prior to printing, so it is initialized above the actual position desired for the first line.

Two utility functions are defined within our #/drawRect: method using the labels construct: draw-next-line and draw-next-payment. The draw-next-line function increments the line rectangle and then converts a lisp string parameter into an NSString object and tells it to draw itself in that rectangle. The draw-next-payment function formats a single line from the payment schedule and then calls draw-next-line to print it. A single list of values is passed as a parameter to this function so the format function uses the "~1{...}" construct to iterate over that list. The iteration count specifier, 1, is needed because there are more values in the list than we actually want to use and we don't want the format statement to begin using them for a second iteration. If that happened it would run out of parameters and trigger an error.

The rest of the #/drawRect: method is straight-forward. Header lines are first formatted and printed if we are printing page zero. Then we extract the appropriate sub-sequence of loan payment lines from the loan-doc's pay-schedule and print

each of them on a subsequent line. Each payment-line is a list of values, so it is easy to just pass that list to the draw-next-payment function as discussed above. Arguably we should grab the pay-schedule and copy it once at the beginning of printing so that if the user changes some parameter in the window while printing is being done it won't affect what is printed. In practice it all seemed to happen pretty quickly, but if you are concerned about this, go ahead and make this change as well.

And that is all of the code needed to turn our loan application into a full-fledged document application that will run under the CCL IDE. Just by virtue of doing a "(require :loan-doc)" in the listener you will see three new menu-items in the File menu. You can use them to make a new loan document that you can then save and subsequently open and of course you can print any loan document. Just remember that if you accidentally choose the "Print ..." method that CCL provides while a loan window is active, the result will be that only one text field will be printed.

Challenges:

Modify the lisp-document-controller's ccl:terminate method so that if it should ever by garbage-collected it would remove the menu items that were added; thus removing all trace of our document class. It should also close any open documents at that time. To do that appropriately it will need to know which documents are still open. Modify the loan-document object to update the appropriate lisp-document-controller object so that it can remove the document from its documents slot when it is closed.

Modify the print routines to respect the requested margins.

Project 8: Lisp Controller

In Project 5 where we built the Package viewing window, it was necessary to implement Objective-C methods for supplying the proper elements from our lisp arrays as demanded by the NSTextView interface objects. As you might expect, doing this sort of thing is fairly common, both in the Objective-C and Lisp worlds. To help developers, Cocoa provides a number of "data controller" classes. The idea is that you can, for example, hand an NSArray to an NSArrayController and it will provide all of the relevant interface methods that make it act as a data source for an NSTableView. These controllers also have "add" and "remove" methods that can be invoked by interface buttons in order to create and add or remove an Objective-C object from the collection. This is all well and good if you happen to have an NSArray or other Objective-C container class and if every sort of thing that you want inside your collection is an Objective-C object, but that's not so useful for Lisp programmers.

We could, of course, just use those Objective-C container classes directly if wanted to go to a lot of trouble. But in my opinion you might as well just use Objective-C directly if you want to go that route. Instead, in this project we'll create an analog to an Objective-C controller that knows how to use Lisp arrays, lists and hash-tables as container objects and which can instantiate any sort of Lisp object and add it to the collection in response to a user-interface control action. In other words, it will act very similarly to a standard Objective-C controller, but be much more useful to Lisp programmers.

In addition, we would like our Lisp controller to be easily accessible from within IB, so I have provided an Xcode project that you can build. It will create a framework that you will need to install and an IB Plugin module that you will need to load into Interface Builder. That will allow you to directly select and configure a lisp-controller object as part of your interface design.

I'm not going to document the construction of the Xcode proejct for the IB plugin module here. If I get enough requests for a tutorial about that I will create one. Otherwise you can use the project I provided as an example for your own work or feel free to modify what I've done for your own purposes. If you do either of those you will want to consult:
http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/IBPlugInGuide/IBPlugInGuide.pdf
I will, of course, discuss the Lisp code for those controller objects a little later in this tutorial.

Some developers might prefer to just use my lisp-controller object and not care about understanding what goes on behind the scenes. To accommodate those people I have created a reference and tutorial for the lisp-controller:
...ccl/contrib/krueger/Interface Projects/Documentation/LispController Reference
Before describing the Lisp code that underlies the lisp-controller object it is best of the reader experiments with it first in order to understand its capabilities. Use the reference manual to install and run some of the examples to get a better feel for how this works. Go ahead, I'll wait here while you go do that.

Right, now that you have a better feel for what the controller can do we'll take a look at how all that is accomplished. I will say just a little bit about the Objective-C plugin module, so you understand the approach that I took. A "normal" plugin would be defined by specifying all of the controller object's behavior in Objective-C. That would be turned into a Framework that would then be loaded into any program that desired to use the plugin objects. But in our case we want to specify the plugin's functionality using Lisp and not via Objective-C. So what I actually did was to provide just enough functionality in Objective-C so that IB could create instances of our lisp-controller plugin object. Luckily for us, what actually happens is that when we select a lisp controller object, add it to the interface, and then save it away in a NIB file,

IB will "encode" the object into an archive that is part of the NIB file and then reconstitute it by "decoding" the archive at runtime. So at runtime we can specify a Lisp definition for the "LispController" class and as long as it knows how to create itself by decoding the archive, everything will work out fine.

If you look at the Xcode project you'll see a minimal implementation of the LispController class that has slots for the various parameters that we want to set within IB and knows how to archive itself and recreate itself from an archive and that's about it. All of the real functionality is implemented in the Lisp version of that class and that's what we will look at next.

Lisp Controller Class

Open up the file "ip:Utilities;lisp-controller.lisp" and follow along with the description below. I'll warn you that the functionality described here is somewhat complex. Consequently we will jump around within the Lisp code quite a bit in order to hopefully make the explanation comprehensible. We'll first discuss the basic data structures, then look at how they are initialized, and finally look at how the Lisp controller responds to request made by the interface object.

Let's start with the lisp-controller class itself:

```
(defclass lisp-controller (ns:ns-object)
  ((root :accessor root)
   (root-type :accessor root-type)
   (gen-root :accessor gen-root)
   (objects :accessor objects)
   (types :reader types)
   (reader-func :accessor reader-func)
   (writer-func :accessor writer-func)
   (count-func :accessor count-func)
   (select-func :accessor select-func)
   (edited-func :accessor edited-func)
   (added-func :accessor added-func)
   (removed-func :accessor removed-func)
   (delete-func :accessor delete-func)
   (add-child-func :accessor add-child-func)
   (children-func :accessor children-func)
   (type-info :accessor type-info)
   (obj-wrappers :accessor obj-wrappers)
   (column-info :accessor column-info)
   (nib-initialized :accessor nib-initialized)
   (view-class :accessor view-class)
   (can-remove :foreign-type #>BOOL :accessor can-remove)
   (can-insert :foreign-type #>BOOL :accessor can-insert)
   (can-add-child :foreign-type #>BOOL :accessor can-add-child)
   (owner :foreign-type :id :accessor owner)
   (view :foreign-type :id :accessor view))
  (:metaclass ns:+ns-object))
```

If you looked at the Objective-C version of this class that was used to build the IB plugin, you will see that although there are some similarities in naming, they are very different objects. Thats not a problem because the Objective-C version is used when objects are instantiated during an IB session and this Lisp version is used at application runtime. As long as they both understand a common archived format, there is no problem.

The root slot will contain the main Lisp object to be displayed. Typically this will be an array or list or hash-table, but it can be an object of some sort as well. If you worked through the examples, the meaning of this value should be clear to you.

The root-type is the type of the root object. If the lisp-controller is configured to generate a root object, then it will use this root-type to find the appropriate initialization form (as specified in IB).

The gen-root slot is a boolean value that specifies whether or not the lisp-controller should generate the root object or whether one will be provided by the implementor in some way. If the latter was specified in IB, the lisp-controller will wait until the root slot has been set before displaying anything.

The objects slot contains a cache of the immediate children of the root object. This exists strictly to minimize the time required to respond to a view object's request for information that should be put into the table.

The types slot contains an ordered list of all the types that were specified within IB plus some default types like list and array that are always acceptable. The order is determined by sub-type relationships. This is important when the lisp-controller needs to determine what type of object it has been given so that it can use an appropriate function to find its children or initialize an instance of it.

The reader-func, writer-func, count-func, delete-func, add-child-func, and children-func slots may each contain an override function that is used if the Lisp programmer wants to replace lisp-controller functionality with some alternative. Although the default lisp-controller behavior is fairly comprehensive, there may well be situations where the developer will want to provide a more customized alternative. These are all specified in IB when configuring the lisp-controller object.

The select-func, edited-func, added-func, and removed-func slots may each contain a notification function that is invoked whenever the lisp controller does the corresponding action. These could be used for whatever purposes the developer might have.

The type-info slot will contain an instance of an "assoc-array" and it's probably worth a small digression at this point to talk about what that is because it plays a fairly prominent role in the lisp-controller code.

Assoc-arrays are defined in the file "ip:Utilities;assoc-array.lisp. It is an object that implements a sparse multi-dimensional associative array that can be indexed by arbitrary lisp-objects. I've found this to be a fairly useful tool to use when organizing data. Without something similar to this, it would be necessary to define several additional classes and accessors to maneuver between them. I think their utility will become more clear as we go through the rest of the lisp-controller code.

Assoc-arrays were discussed more in the LispController Reference document, so I'm not going to describe them in any detail here. The reader is invited to read through the code to understand it. I will only say that the implementation is done using nested hash-tables and is fairly straight-forward.

So back to the lisp-controller object ...

The obj-wrappers slot is used when displaying things in an NSOutlineView. An NSOutlineView keeps track of objects at each level of the hierarchy and as each one is expanded it will request its children so that they can be displayed as well. Unfortunately, those must be Objective-C objects. The lisp-controller arranges to wrap each lisp object inside an Objective-C object just so that it can be delivered to the NSOutlineView. If a displayed object is then contracted and re-expanded, the same Objective-C objects must be used for its children. So in the obj-wrappers slot we keep an assoc-array that contains the association between an arbitrary lisp object and a corresponding Objective-C "wrapper" object. This could be a simple hash-table, but an assoc-array was used to make the code look similar to other usage.

The column-info slot contains another assoc-array object that maintains information about each column in the associated NSTableView or NSOutlineView object. This is initialized after the NIB has been loaded by looking at the table and extracting information about each column. As you saw when you went through the examples, some of this information is in the form of lisp constructs that are interpreted in different ways, depending on the type of Lisp object being displayed. We will expand on this idea later when we examine how this slot is initialized.

The nib-initialized slot is a boolean that tells us that the NIB file has been completely initialized. This is necessary to avoid problems during the brief period after the lisp-controller object has been created and before it has been completely initialized by the NIB loading mechanism.

The view-class slot contains the class of the associated NSTableView or NSOutlineView object. Slightly different functionality is required for the two, so the value is saved here and referenced as needed.

The can-add, can-remove, and can-add-child slots are foreign slots that may be bound to by interface elements to indicate whether they should be enabled. For example, if the lisp-controller is in a state where the interface should permit the addition of a child to the root object, then the value of can-add will contain #$YES. If some interface button is conditionally enabled via a link to the canAdd path of the lisp-controller object, then it will be enabled. If the state is such that addition should not be allowed (as for example when no parent is currently selected), then the slot will contain #$NO and the button would be disabled. Similarly, the can-remove and can-add-child slots indicate whether it is currently valid to remove the currently selected object and add a child to the currently selected object, respectively.

The owner slot will typically contain a reference to the FileOwner object that was used to initialize the NIB file. This must be set in IB by ctrl-dragging from the lisp-controller object to the FileOwner object and setting the owner outlet. The only time this is used is when calling notification functions. This value of the owner slot is passed as one of the arguments. This may be useful if a developer defines a single notification function that is to be used by several different windows. Having the value of the owner can be used to disambiguate where the call originates.

Finally, the view slot contains a reference to the NSTableView or NSOutlineView object that is associated with this controller. This is set in IB by ctrl-dragging from the lisp-controller to the view object and setting the view outlet.

Lisp Controller Initialization

These are the fundamental data structures of the lisp-controller. Next we will look at how all of their values are initialized. As previously mentioned, many of these values are specified in IB, some for the lisp-controller object and some for the view object. We'll first look at how those specified for the lisp-controller object are extracted from the archived object that was part of the NIB.

When the NIB file is loaded, space is allocated for a lisp-controller object and the Objective-C function #/initWithCoder is called to initialize it. We'll get to the details of our lisp implementation of that method in just a bit, but the first thing it does is call another Objective-C method called #/init. this is mostly done so that if a lisp-controller object is created by calling (make-instance 'lisp-controller) from some Lisp context that a meaningful object is constructed. Normally that will not be the case, but it was useful to have around while I was debugging the code. That #/init method is shown below:

```
(objc:defmethod (#/init :id)
                ((self lisp-controller))
  ;; need to do this to initialize default values that are needed when
  ;; this object is instantiated from Objective-C runtime
  (unless (slot-boundp self 'nib-initialized)
    (setf (nib-initialized self) nil))
  (unless (slot-boundp self 'select-func)
    (setf (select-func self) nil))
  (unless (slot-boundp self 'edited-func)
    (setf (edited-func self) nil))
  (unless (slot-boundp self 'added-func)
    (setf (added-func self) nil))
  (unless (slot-boundp self 'removed-func)
    (setf (removed-func self) nil))
  (unless (slot-boundp self 'add-child-func)
    (setf (add-child-func self) nil))
  (unless (slot-boundp self 'delete-func)
    (setf (delete-func self) nil))
  (unless (slot-boundp self 'reader-func)
    (setf (reader-func self) nil))
  (unless (slot-boundp self 'writer-func)
    (setf (writer-func self) nil))
  (unless (slot-boundp self 'count-func)
    (setf (count-func self) nil))
  (unless (slot-boundp self 'objects)
    (setf (objects self) nil))
  (unless (slot-boundp self 'root)
    ;; note that we have to set root slot to avoid
    ;; calling accessor functions. This is a
    ;; special case and the only place where we
    ;; want to set the root to something that
    ;; doesn't match the root-type specified in IB
    (setf (slot-value self 'root) nil))
  (unless (slot-boundp self 'root-type)
    (setf (root-type self) t))
  (unless (slot-boundp self 'types)
    (setf (types self) nil))
  (unless (slot-boundp self 'type-info)
    (setf (type-info self) (make-instance 'assoc-array :rank 2)))
  ;; Now set up some default type info for standard types
  ;; These can be overridden in the lisp-controller setup
  ;; within Interface Builder.
  ;; Typically users would define their own types and
  ;; specify values for them rather than overriding these
  ;; but it is permissable to do so.
  (setf (assoc-aref (type-info self) 'hash-table :child-key) #'children)
  (setf (assoc-aref (type-info self) 'ht-entry :child-key) #'children)
  (setf (assoc-aref (type-info self) 'list :child-key) #'children)
  (setf (assoc-aref (type-info self) 'vector :child-key) #'children)
```

```
  (setf (assoc-aref (type-info self) 'hash-table :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'ht-entry :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'list :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'vector :child-setf-key) #'(setf children))
  (setf (assoc-aref (type-info self) 'hash-table :child-type) 'ht-entry)
  (setf (assoc-aref (type-info self) 'list :child-type) 'list)
  (setf (assoc-aref (type-info self) 'vector :child-type) 'vector)
  (setf (assoc-aref (type-info self) 'hash-table :initform)
        '(make-hash-table))
  (setf (assoc-aref (type-info self) 'list :initform)
        nil)
  (setf (assoc-aref (type-info self) 'vector :initform)
        '(make-array '(10) :adjustable t :fill-pointer 0 :initial-element nil))
  (unless (slot-boundp self 'obj-wrappers)
    (setf (obj-wrappers self) (make-instance 'assoc-array :rank 1)))
  self)
```

The first part of the #/init method just sets up default values for simple slots. In most cases that value is nil. The first somewhat interesting thing is when we set the default value for the type-info slot. We set a value that is a two-dimensional assoc-array. The first dimension will be indexed by a Lisp type. The second dimension will be indexed by a keyword value that specifies what type of information is contained in the value. Various default values are then put into the assoc-array for the standard Lisp types list, vector, and hash-table. The type ht-entry is defined specifically to handle hash-table children and will be discussed later. The user is free to override these defaults in IB by specifying other values. Normally the :child-key, :child-setf-key, and :child-type should be pretty good defaults, but the :initforms would be overridden in IB if these types were used.

Note that the column-info slot is not initialized at this point. We should not do that because we do not yet know that the view slot is valid. So obviously we cannot go look at it. That will happen sometime later after we have been notified that the NIB was initialized by a call to #/awakeFromNib.

The normal way that lisp-controller objects will be created is via a call to #/initWithCoder:. As previously mentioned, this first calls the #/init method. Once this basic, default initialization has been completed we begin to extract information from the archive by making calls to the decoder object that is given to us.

```
(objc:defmethod (#/initWithCoder: :id)
                ((self lisp-controller) (decoder :id))
  ;; This method is called when the Nib is loaded and provides values defined
  ;; when the NIB was created
  (#/init self)
  (with-slots (reader-func writer-func count-func select-func edited-func
               add-child-func root-type gen-root added-func removed-func
               children-func type-info delete-func) self
    (let ((type-info-array (#/decodeObjectForKey: decoder #@"typeInfo")))
      (dotimes (i (#/count type-info-array))
        ;; for each type specified in IB by the user
        (let* ((row-array (#/objectAtIndex: type-info-array i))
               (ns-str-type-name (#/objectAtIndex: row-array 0))
               (type-name (nsstring-to-sym ns-str-type-name))
               (child-type (nsstring-to-sym (#/objectAtIndex: row-array 1)))
               (child-func-str (ns-to-lisp-string (#/objectAtIndex: row-array 2)))
               (child-func (find-func child-func-str))
               (reader-sym (and child-func (function-name child-func)))
               (writer-form `(setf (,reader-sym thing) new-val))
               (child-writer-func (and child-func
                                       (valid-setf-for writer-form)
                                       (eval `(function (lambda (new-val thing)
                                                         ,writer-form))))))
          (when child-type
              (setf (assoc-aref type-info type-name :child-type) child-type))
          (when child-func
            (setf (assoc-aref type-info type-name :child-key) child-func))
          (when child-writer-func
              (setf (assoc-aref type-info type-name :child-setf-key) child-writer-func)))))
    (let ((initform-array (#/decodeObjectForKey: decoder #@"initforms")))
      (dotimes (i (#/count initform-array))
```

```
      ;; for each initform specified in IB by the user
      (let* ((row-array (#/objectAtIndex: initform-array i))
             (ns-str-type-name (#/objectAtIndex: row-array 0))
             (type-name (nsstring-to-sym ns-str-type-name))
             (initform (ns-to-lisp-object t (#/objectAtIndex: row-array 1))))
        (when initform
            (setf (assoc-aref type-info type-name :initform) initform)))))
  (let ((sort-info-array (#/decodeObjectForKey: decoder #@"sortInfo")))
    (dotimes (i (#/count sort-info-array))
      ;; for each sort predicate and key specified in IB by the user
      (let* ((row-array (#/objectAtIndex: sort-info-array i))
             (ns-str-type-name (#/objectAtIndex: row-array 0))
             (type-name (nsstring-to-sym ns-str-type-name))
             (sort-key (nsstring-to-func (#/objectAtIndex: row-array 1)))
             (sort-pred (nsstring-to-func (#/objectAtIndex: row-array 2))))
        (when sort-pred
          (setf (assoc-aref type-info type-name :sort-pred) sort-pred))
        (when sort-key
          (setf (assoc-aref type-info type-name :sort-key) sort-key)))))
  (setf root-type (nsstring-to-sym (#/decodeObjectForKey: decoder #@"rootType")))
  (setf (types self) (delete-duplicates (list* root-type
                                               'ht-entry
                                               (mapcar-assoc-array #'identity type-info))))
  (setf reader-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"readerFunc")))
  (setf writer-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"writerFunc")))
  (setf count-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"countFunc")))
  (setf select-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"selectFunc")))
  (setf edited-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"editedFunc")))
  (setf added-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"addedFunc")))
  (setf removed-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"removedFunc")))
  (setf delete-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"deleteFunc")))
  (setf add-child-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"addChildFunc")))
  (setf children-func (nsstring-to-func (#/decodeObjectForKey: decoder #@"childrenFunc")))
  (setf gen-root (#/decodeBoolForKey: decoder #@"genRoot")))
self)
```

You can think of an archive as something like a Lisp hash-table. That is, you can specify a key and some data to associate with it. That's what our Xcode-defined LispController Objective-C method "encodeWithCoder:" did when it was called by IB. Here we are just doing the reverse by specifying a key and getting back the Objective-C object that was archived. We then convert that object into something that is more usable within Lisp.

In the IB version of the LispController class I defined three arrays that corresponded to the three tables that you see when you inspect a LispController object in IB. That made it pretty easy to implement. But in Lisp I wanted to pull all of this information into the single assoc-array that is the value of the type-info slot. No problem, we simply extract each of the NSArray objects from the archive and then iterate through them to extract relevant information. This is then put into the type-info assoc-array via calls of the form

(setf (assoc-aref type-info type-name <some keyword>) converted-value)

In many cases the value that we originally put into the NSArray was a simple NSString, but in Lisp we want to treat it as either a lisp form, symbol, string, or the name of a function. To make this easy, several utility functions were defined to convert appropriately. The function nsstring-to-sym creates a Lisp symbol. The function nsstring-to-func returns a Lisp function (if it exists). The function ns-to-lisp-string converts an NSString to a Lisp string. Finally ns-to-lisp-object will take an NSString and effectively do a read-from-string to turn it into an arbitrary lisp object. This is useful for things like initialization forms.

Slots for notification and override slots are similarly populated from archived values. Empty NSStrings are converted to nil values in functions like nsstring-to-func.

Once this function is complete, all of the values specified for the LispController in IB have been transferred to the corresponding runtime instance of lisp-controller.

There is additional initialization that occurs once the NIB has been completely loaded and all the objects are linked together. At that time the lisp-controller object can examine the associated view object to retrieve additional information that the developer provided when the view's column IDs were set. The #/awakeFromNib method is called to inform the lisp-controller that the NIB was completely loaded, so we define a function to manage that additional initialization:

```
(objc:defmethod (#/awakeFromNib :void)
                ((self lisp-controller))
  (setf (nib-initialized self) t)
  (unless (eql (view self) (%null-ptr))
    (setf (view-class self) (#/class (view self)))
    (init-column-info self (view self))
    (when (gen-root self)
      ;; create the root object
      (setf (root self) (new-object-of-type self (root-type self))))
    (when (objects self)
      (setup-accessors self))))
```

This function sets the view-class slot and then initializes the column-info slot by calling init-column-info:

```
(defmethod init-column-info ((self lisp-controller) (view ns:ns-table-view))
  (with-slots (column-info) self
    (let* ((tc-arr (#/tableColumns view))
           (col-obj nil)
           (idc nil)
           (col-count (#/count tc-arr)))
      (unless tc-arr
        (ns-log "#/tableColumns returned nil for view")
        (return-from init-column-info))
      (setf column-info (make-instance 'assoc-array :rank 2))
      (dotimes (i col-count)
        (setf col-obj (#/objectAtIndex: tc-arr i))
        (setf (assoc-aref column-info col-obj :col-indx) i)
        (setf (assoc-aref column-info i :col-obj) col-obj)
        (setf idc (ns-to-lisp-string (#/identifier col-obj)))
        (setf (assoc-aref column-info col-obj :col-string) idc)
        (setf (assoc-aref column-info col-obj :col-val)
              (read-from-string idc nil nil))
        (setf (assoc-aref column-info col-obj :col-title)
              (ns-to-lisp-string (#/title (#/headerCell col-obj))))
        ;; find any formatter attached to the data cell for this column and
        ;; use info from it to help us translate to and from lisp objects
        ;; appropriately
        (let ((formatter-object (#/formatter (#/dataCell col-obj))))
          (unless (eql formatter-object (%null-ptr))
            (cond ((typep formatter-object 'ns:ns-date-formatter)
                   (setf (assoc-aref column-info col-obj :col-format) :date))
                  ((typep formatter-object 'ns:ns-number-formatter)
                   (cond ((#/generatesDecimalNumbers formatter-object)
                          (let ((dec-digits (#/maximumFractionDigits formatter-object)))
                            (setf (assoc-aref column-info col-obj :col-format)
                                  (list :decimal dec-digits))))
                         (t
                          (setf (assoc-aref column-info col-obj :col-format)
                                :number)))))))))))
```

First this method gets all the table's columns from the view. This is returned in an NSArray object. That array is traversed and information about that column is put into the assoc-array that is in the column-info slot of the lisp-controller. That assoc-array is a two-dimensional assoc-array that is indexed by the column object pointer and a keyword that indicates the type of column information that is being accessed. The value is some Lisp value that is appropriate for that type of information. The information kept includes the column number (:col-indx keyword as the second index), a reference to the Objective-C column object (:col-obj keyword as the second index), the column's identifier as an NSString (:col-string keyword as the second index), the Lisp object that results from reading the identifier string (:col-val keyword as the second index), and the title of the column that was specified in IB as a Lisp string (:col-title keyword as the second index).

In addition this method will examine any data formatters that were attached to the column to see if they can be used to determine how Lisp data should be packaged when given to the view and how Lisp data should be extracted from objects that are given to the Lisp code. That information is stored using the :col-format keyword as the second index to the assoc-array.

The next thing that the #/awakeFromNib method does is to initialize the root object if that has been specified. It does that

by calling the new-object-of-type method. We'll defer discussion of that method until later when we consider how new objects are created.

Finally the #/awakeFromNib method calls setup-accessors to initialize appropriate accessor methods for each column for every possible type that might be contained in any row of the table. The general idea is that if we can find a way to apply the accessor that was specified for the column to the object displayed in any particular row, then we will do that to retrieve the Lisp data to be displayed in that column for that row.

```
(defmethod setup-accessors ((self lisp-controller))
  ;; This method must be called to initialize the column value
  ;; accessor functions for a lisp-controller.
  ;; It is called after NIB loading has been done.
  (with-slots (reader-func column-info type-info types) self
    (unless reader-func
      (dolist (col (mapcar-assoc-array #'identity column-info))
        (let ((col-id (assoc-aref column-info col :col-val)))
          (dolist (typ types)
            (setf (assoc-aref type-info typ col)
                  (reader-writer-pair typ col-id))))))))
```

First setup-accessors iterates across all columns. For each column it iterates across all of the known types (both default and user-specified) and tries to makes sense of the column identity as a read accessor for that type. If it can, then it also tries to construct a corresponding write accessor. Those accessors are saved in the type-info assoc-array indexed by the type and column object reference as a dotted pair. Let's take a look at how those accessors are created:

```
(defun reader-writer-pair (typ col-val)
  (let* ((reader-form nil)
         (writer-form nil))
    (cond ((null col-val)
           ;; reader justs return the object itself
           ;; leave the writer-form null
           (setf reader-form 'row))
          ((and (eq col-val :key) (eq typ 'ht-entry))
           ;; used for the key value in a hash table
           (setf reader-form '(ht-key row))
           (setf writer-form '(setf (ht-key row) new-val)))
          ((and (eq col-val :value) (eq typ 'ht-entry))
           ;; used for the value in a hash table
           (setf reader-form '(ht-value row))
           (setf writer-form '(setf (ht-value row) new-val)))
          ((eq col-val :row)
           (setf reader-form 'row)
           (setf writer-form '(setf row new-val)))
          ((numberp col-val)
           (cond ((subtypep typ 'vector)
                  (setf reader-form `(aref row ,col-val))
                  (setf writer-form `(setf (aref row ,col-val) new-val)))
                 ((subtypep typ 'list)
                  (setf reader-form `(nth ,col-val row))
                  (setf writer-form `(setf (nth ,col-val row) new-val)))
                 ((eq typ 'ht-entry)
                  ;; Index if the value is a sequence
                  (setf reader-form `(when (typep (ht-value row) 'sequence)
                                       (elt (ht-value row) ,col-val)))
                  (setf writer-form `(when (typep (ht-value row) 'sequence)
                                       (setf (elt (ht-value row) ,col-val) new-val))))
                 ((subtypep typ 'hash-table)
                  ;; use the number as a key into the hash table and return the value
                  (setf reader-form `(gethash ,col-val row))
                  (setf writer-form `(setf (gethash ,col-val row) new-val)))
                 (t
                  ;; index if row is any other type of sequence
                  (setf reader-form `(when (typep row 'sequence)
                                       (elt row ,col-val)))
                  (setf writer-form `(when (typep row 'sequence)
```

```
                                (setf (elt row ,col-val) new-val))))))
      ((and (symbolp col-val) (fboundp col-val))
       (cond ((eq typ 'ht-entry)
              ;; Assume the function applies to the value
              (setf reader-form `(,col-val (ht-value row)))
              (when (valid-setf-for reader-form)
                (setf writer-form `(setf (,col-val (ht-value row)) new-val))))
             (t
              (setf reader-form `(,col-val row))
              (when (valid-setf-for reader-form)
                (setf writer-form `(setf (,col-val row) new-val))))))
      ((symbolp col-val)
       (cond ((subtypep typ 'hash-table)
              ;; use the symbol as a key into the hash table and return the value
              (setf reader-form `(gethash ,col-val row))
              (setf writer-form `(setf (gethash ,col-val row) new-val)))))
      ((and (consp col-val) (eq (first col-val) 'function))
       (let ((col-val (second col-val)))
         (when (and (symbolp col-val) (fboundp col-val))
           (cond ((eq typ 'ht-entry)
                  ;; Assume the function applies to the value
                  (setf reader-form `(,col-val (ht-value row)))
                  (when (valid-setf-for reader-form)
                    (setf writer-form `(setf (,col-val (ht-value row)) new-val))))
                 (t
                  (setf reader-form `(,col-val row))
                  (when (valid-setf-for reader-form)
                    (setf writer-form `(setf (,col-val row) new-val)))))))))
      ((consp col-val)
       ;; accessors are lisp forms possibly using keywords :row, :key, and :value
       ;; which are replaced appropriately
       (setf reader-form (nsubst 'row ':row
                                 (nsubst '(ht-key row) :key
                                         (nsubst '(ht-value row) :value
                                                 col-val))))
       (when (valid-setf-for reader-form)
         (setf writer-form `(setf ,col-val new-val)))))
  (when *lisp-controller-debug*
    (ns-log (format nil "Reader-form: ~s~%Writer-form: ~s" reader-form writer-form)))
  (cons (and reader-form
             (eval-without-errors `(function (lambda (row) ,reader-form))))
        (and writer-form
             (eval-without-errors `(function (lambda (new-val row) ,writer-form))))))
```

If no column-identifier was specified in IB, then the row-object itself will be displayed in the column. This might be useful for single column tables or when the row-object can itself be used as an identifier of sorts for the row (i.e. printing the object results in some meaningful identifier). No write accessor is created in this case.

If the root object being displayed is a hash-table, then a number of special things are done to represent it correctly. Essentially the hash-table is converted into a list of ht-entry objects, where each such object represents a key-value pair. Each row-object displayed would then be an ht-entry object. The developer doesn't really need to worry about that. The only thing they need to know is that they can use the keyword :key to retrieve the key from the key-value pair and the keyword :value to retrieve the value. If a column identifier is either of these two keywords then the read and write accessors are set to forms that read and write the ht-entry slots respectively. The ht-entry objects have functionality that reflect changed keys and values back to the original hash-table. For example, when the user edits a column value for which the identifier :value has been set, then the ht-entry setf function that is called will also modify the corresponding value in the root hash-table. If you are curious about exactly how this works, examine the methods for the ht-entry class.

If the keyword :row is used, it refers to the row-object itself. If used alone as the column identifier (i.e. not embedded in some other form) then the row object is displayed in that column, just as it would be if no column identifier was specified. But in this case we assume that the developer is doing so knowledgeably, so we allow for the possibility that setting the row object itself to a new value may be possible. So we will create a setf form and if that turns out to be a valid form then it will be used to write a new value.

Next we consider that case where the column identifier is a number. Generally speaking we will use that number as an

index into the row-object. For sequences like lists and vectors that has an obvious interpretation. If the row-object is an ht-entry (i.e. it represents a key/value pair from a hash-table), then we will assume that the value is some type of sequence and index into that. It is possible that the row-object could itself be a hash-table. In that case we treat the number as a key into that hash-table and return the result.

Next we consider column identifiers that are Lisp symbols that have a function binding. If the row-object is an ht-entry we will apply that function to the value of the key/value pair. Otherwise we will apply the function to the row-object itself.

If the column identifier is a symbol that does not have a function binding then the only likely use is as a key into a hash-table. So if the row-object is a hash-table we will use it that way. For any other type it will be ignored. Note that this can be a source of error if a symbol is intended to specify a function, but is spelled incorrectly or does not specify a package correctly. In that case it would be silently ignored.

If the column identifier was specified using something of the form #'func or equivalently (function func) then it is used in the same way as a symbol with a function binding.

If the column identifier is any other sort of list, then we assume that it is an accessor form of some sort. The keywords :row, :key, and :value can be used within that form to refer to the row-object or the key of an ht-entry row-object or the value of an ht-entry row-object, respectively. We substitute for those keywords to create read and write accessors.

In all cases we validate that the write form is valid before saving it. This is done by calling the valid-setf-for function shown next.

```
(defun valid-setf-for (read-form)
  (multiple-value-bind (a b c func-form d) (get-setf-expansion read-form)
    (declare (ignore a b c d))
    (or (not (eq (first func-form) 'funcall))
        ;; this must be a built-in function, so assume setf works
        ;; otherwise make sure the function name specified is fboundp
        (let ((func-name (second (second func-form))))
          (and (typep func-name 'function-name) (fboundp func-name))))))
```

This checks to make sure that the function that would be called by expanding the setf form is either a built-in function or is a user-specified function that has a function binding.

The last thing that the reader-writer-pair function does is evaluate the forms. Any errors that occur while evaluating a form are ignored and nil is returned for that evaluation. This hopefully results in accessors that can be funcalled as needed.

Lisp Controller Handling of NSTableView Calls

At this point we have examined all of the initialization functions and can now consider what happens when the end-user interacts with the user interface and objects there subsequently make calls to the lisp-controller. Some calls are made by virtue of the lisp-controller being the data-source for an NSTableView or NSOutlineView. Others are made because the lisp-controller is the delegate object for the view. Both links should always be made between the view and the lisp-controller. We will examine the calls made by each of these types of views.

An NSTableView calls the methods
    #/numberOfRowsInTableView:
    #/tableView:objectValueForTableColumn:row:
    #/tableView:setObjectValue:forTableColumn:row:
on the lisp-controller as its data source. It calls the methods
    #/tableView:shouldEditTableColumn:row:
    #/tableViewSelectionDidChange:
on the lisp-controller as the view's delegate.

The #/numberOfRowsInTableView: method calls the specified override method if it exists. Otherwise it simply returns the length of the objects slot. You will recall that this contains the children of the root object. This tells the view how many rows should be displayed.

The #/tableView:objectValueForTableColumn:row: method returns an appropriate Objective-C object to be displayed in the column and row specified. Here is the method:

```
(objc:defmethod (#/tableView:objectValueForTableColumn:row: :id)
                ((self lisp-controller)
                 (tab :id)
```

```
                (col :id)
                (row #>NSInteger))
  (declare (ignore tab))
  (let ((ns-format (assoc-aref (column-info self) col :col-format)))
    (lisp-to-ns-object (col-value self (elt (objects self) row) col) ns-format)))
```

This is fairly simple, but calls two helper functions, lisp-to-ns-object and col-value, that provide additional functionality. We will look at those next. The col-value method finds the appropriate lisp value to be displayed:

```
(defmethod col-value ((self lisp-controller) obj col-obj)
  ;; Get the lisp value for some column for an object
  ;; return "" if there isn't one so the display doesn't
  ;; have "nil" for columns without values.
  (let* ((obj-type (controller-type-of self obj))
         (reader-func (or (reader-func self)
                          (car (assoc-aref (type-info self) obj-type col-obj)))))
    (if reader-func
      (funcall reader-func obj)
      "")))
```

The obj parameter is the object being displayed in the selected row and the col-obj parameter is the Objective-C column object for the selected column. If a reader override function has been specified, then it is used. Otherwise the reader functions that were constructed as part of the initialization are used. The choice of which function to use depends on the type of the row object and the column where it will be displayed.

After the lisp object has been retrieved, it must be converted to an appropriate Objective-C object for display. That is the role of the lisp-to-ns-object function which is defined in ip:Utilities;ns-object-utils.lisp:

```
(defun lisp-to-ns-object (lisp-obj &optional (ns-format nil))
  ;; convert an arbitrary lisp object to an appropriate NSObject so
  ;; that it can be displayed someplace
  (cond ((ccl::objc-object-p lisp-obj)
         ;; it's already an NSObject so just return it
         lisp-obj)
        ((eq ns-format :date)
         ;; assume lisp-obj is an integer representing a lisp date
         (lisp-to-ns-date lisp-obj))
        ((and (consp ns-format) (eq (first ns-format) :decimal))
         (cond ((typep lisp-obj 'fixnum)
                (lisp-to-ns-decimal lisp-obj :decimals (second ns-format)))
               ((typep lisp-obj 'number)
                (lisp-to-ns-decimal (round (* (expt 10 (second ns-format)) lisp-obj))
                                    :decimals (second ns-format)))
               (t
                (lisp-to-ns-decimal 0 :decimals (second ns-format)))))
        ((integerp lisp-obj)
         (#/numberWithInt: ns:ns-number lisp-obj))
        ((typep lisp-obj 'double-float)
         (#/numberWithDouble: ns:ns-number lisp-obj))
        ((floatp lisp-obj)
         (#/numberWithFloat: ns:ns-number lisp-obj))
        ((null lisp-obj)
         #@"")
        (t
         (lisp-to-temp-nsstring (if (stringp lisp-obj)
                                  lisp-obj
                                  (format nil "~s" lisp-obj))))))
```

This function uses the ns-format argument, if provided, to guide the decision. If the Lisp object is already an Objective-C object, then it is just returned. If we know from the ns-format argument that it is a date, then an NSDate object is created (see the lisp-to-ns-date function defined in ip:Utilities;date.lisp). If we learned from the formatter attached to a column that the use of NSDecimalNumber objects is desired and if the Lisp object is a fixnum, then we assume that the Lisp object uses the format defined in decimal.lisp which was discussed for Project 6. If the Lisp object is some other kind of number, then we round it appropriately and convert it to an NSDecimalNumber. In the absence of an ns-format value Lisp numbers are converted to an appropriate Objective-C numeric object. In all other cases we just print the Lisp object to a

string that is converted to an NSString for display.

The next method needed to support NSTableViews is #/tableView:setObjectValue:forTableColumn:row:. This is called when a user modifies the value displayed within some row/column. The implementation of this is:

```
(objc:defmethod (#/tableView:setObjectValue:forTableColumn:row: :void)
                ((self lisp-controller)
                 (tab :id)
                 (val :id)
                 (col :id)
                 (row #>NSInteger))
  ;; We let the user edit the table and something was changed
  ;; try to convert it to the same type as what is already in that
  ;; position in the objects.
  (declare (ignore tab))
  (let* ((row-obj (elt (objects self) row))
         (old-obj (col-value self row-obj col))
         (ns-format (assoc-aref (column-info self) col :col-format))
         (new-val (ns-to-lisp-object old-obj val ns-format)))
    (if (writer-func self)
      (funcall (writer-func self)
               new-val
               (root self)
               row
               (assoc-aref (column-info self) col :col-val))
      (set-col-value self row-obj col new-val))
    (when (edited-func self)
      (let* ((row-obj (object-at-row self row))
             (edited-obj (if (typep row-obj 'ht-entry)
                           (list (ht-key row-obj) (ht-value row-obj))
                           row-obj)))
        (funcall (edited-func self)
                 (owner self)
                 self
                 (root self)
                 row
                 (assoc-aref (column-info self) col :col-indx)
                 edited-obj
                 old-obj
                 new-val)))
    ;; re-sort and reload the table
    ;; unfortunately we probably have to do this for every change since
    ;; we don't know what affects the sort order
    (sort-sequence self (objects self))
    (#/reloadData (view self))))
```

This function must convert the new value, represented as an Objective-C object, into a usable Lisp object. The tricky decision is what sort of object that should be. This is done by the ns-to-lisp-object function which makes use of whatever information is available. That includes information from any formatter that has been attached to the field (we retrieved that information as part of lisp-controller initialization described earlier) and the type of Lisp object that was displayed in the field to begin with. In general we try to convert back to the same type as was originally there, but if the user really wants to replace a number with a symbol or string, we allow that to happen. Note that the use of a formatter within IB can prevent the user from changing to an invalid type and that is the mechanism that should be used to force the user to adhere to a particular type when editing. Let's take a closer look at the ns-to-lisp-object function:

```
(defun ns-to-lisp-object (old-lisp-obj ns-obj &optional (ns-format nil))
  ;; convert an arbitrary NSObject object to an appropriate lisp object.
  ;; Often done so that it can replace the old-lisp-obj when edited
  ;; An empty string @"" returns nil if old-lisp-obj is not a string
  (cond ((ccl::objc-object-p old-lisp-obj)
         ;; the old value was an NSObject so just return the new value
         ns-obj)
        ((typep ns-obj 'lisp-ptr-wrapper)
         ;; just strip the wrapper and return the original object
         (lpw-lisp-ptr ns-obj))
```

```
        ((typep ns-obj 'ns:ns-decimal)
         (if (floatp old-lisp-obj)
           ;; convert the decimal to a float
           (#/doubleValue ns-obj)
           ;; otherwise convert it to an appropriate lisp integer with assumed
           ;; decimals (see ip;Utilities;decimal.lisp)
           (if (eq (first ns-format) :decimal)
             (lisp-from-ns-decimal ns-obj :decimals (second ns-format))
             (lisp-from-ns-decimal ns-obj))))
        ((typep ns-obj 'ns:ns-number)
         (read-from-string (ns-to-lisp-string (#/descriptionWithLocale: ns-obj (%null-ptr)))
                           nil nil))
        ((typep ns-obj 'ns:ns-date)
         (ns-to-lisp-date ns-obj))
        (t
         (let ((str (ns-to-lisp-string ns-obj)))
           (if (stringp old-lisp-obj)
             str
             (read-from-string str nil nil))))))
```

If the old lisp object, i.e. the one that was retrieved to be displayed initially, was already an Objective-C object, then we will simply return the new Objective-C object. This allows Lisp developers to display and store Objective-C instances if they desire. When we discuss the functionality needed to support NSOutlineViews we will see that it sometimes we just want to package up Lisp objects in an Objective-C wrapper and unwrap them when then are sent back to us. That wrapper is a lisp-ptr-wrapper instance. If we get one of those, the Lisp object that it encapsulates is returned.

If the value is an NSDecimal, then we convert it to either a floating point value (if the old value was a float) or to the internal scaled decimal fixnum format implemented in ip;Utilities;decimal.lisp and described previously in this document.

Any other type of NSNumber is just converted by reading from its string representation. NSDate objects are converted to Lisp date integers. Anything else that we get is converted to a Lisp string. If the previous value was a string it is just left as a Lisp string. Otherwise we read from that string and return whatever was read. In this way we can retrieve any Lisp form that could be typed into a Lisp listener window. No evaluation of that form occurs of course. Such forms could include numbers, symbols, lists, vectors, etc.

After the new value has been set, the lisp-controller will call any notification function that was specified when the lisp-controller was configured in IB.

The lisp-controller implements two additional methods that are called by virtue of it being the delegate of an NSTableView: #/tableView:shouldEditTableColumn:row: and #/tableViewSelectionDidChange:. The first is called when a user clicks in a particular row/column and the lisp-controller must either deny or grant permission to edit the field. If no function has been found to write that column for the type of the current row-object, then we will deny permission to edit the field. If the developer finds that it is not possible to edit a field that should be editable, this may be the cause. For some reason the lisp-controller has been unable to construct a function to write the value. The function to do all this is below.

```
(objc:defmethod (#/tableView:shouldEditTableColumn:row: #>BOOL)
                ((self lisp-controller)
                 (tab :id)
                 (col :id)
                 (row #>NSInteger))
  (declare (ignore tab))
  ;; allow editing if there is a function available to setf a new value
  (if (or (writer-func self)
          (let ((obj-type (controller-type-of self (elt (objects self) row))))
            (cdr (assoc-aref (type-info self) obj-type col))))
    #$YES
    #$NO))
```

The #/tableViewSelectionDidChange:. method tells us that the user changed the previous selection. This may result in a new field being selected or no field being selected. The function to deal with this is:

```
(objc:defmethod (#/tableViewSelectionDidChange: :void)
                ((self lisp-controller) (notif :id))
  (let* ((tab (#/object notif))
```

```
        (row-indx (#/selectedRow tab))
        (col-indx (#/selectedColumn tab)))
    ;; enable/disable buttons that remove current selection
    (#/willChangeValueForKey: self #@"canRemove")
    (if (minusp row-indx)
      (setf (can-remove self) #$NO)
      (setf (can-remove self) #$YES))
    (#/didChangeValueForKey: self #@"canRemove")
    ;; enable/disable buttons that want to add a child to
    ;; the current selection
    (set-can-add-child self row-indx)
    ;; User code to do something when a cell is selected
    (when (select-func self)
      (let* ((row-obj (and (not (eql row-indx -1)) (object-at-row self row-indx)))
             (col (assoc-aref (column-info self) col-indx :col-obj))
             (selected-obj (cond ((and (minusp col-indx) (minusp row-indx))
                                  nil)
                                 ((minusp col-indx)
                                  row-obj)
                                 ((minusp row-indx)
                                  (assoc-aref (column-info self) col :col-title))
                                 (t
                                  (col-value self row-obj col)))))
        (funcall (select-func self)
                 (owner self)
                 self
                 (root self)
                 row-indx
                 col-indx
                 selected-obj)))))
```

Recall that we allow buttons that add or remove new objects to be selectively enabled depending on what is currently selected. Here we set the lisp-controller slots that control that functionality. Then, if the developer specified a notification function to be called when the selection is changed, we gather up the necessary arguments and call it.

Lisp Controller Handling of NSOutlineView Calls

In a manner similar to what NSTableView objects do, NSOutlineView objects call the methods
#/outlineView:numberOfChildrenOfItem:
#/outlineView:child:ofItem:
#/outlineView:isItemExpandable:
#/outlineView:objectValueForTableColumn:byItem:
#/outlineView:setObjectValue:forTableColumn:byItem:
on the lisp-controller as its data source. It calls the methods
#/outlineView:shouldEditTableColumn:item:
#/tableViewSelectionDidChange:
on the lisp-controller as the view's delegate.

NSOutlineView objects present a table view, but each object in the table can potentially be expanded into a subordinate set of objects that are displayed indented from its parent. In turn, each of these may also be expandable. The user controls whether or not the subordinate list is expanded and shown by clicking on a small arrow next to the item. One example of this, the Lisp Class browser, was given in the List-Controller reference document.

To implement this functionality the NSOutlineView asks its data source for an initial set of objects (the children of nil) and then for each of those top-level objects it asks whether it is expandable. If so, it asks for the number of children and then asks for each child object by number. In general the Lisp developer would like those objects to be Lisp objects rather than Objective-C objects, so the lisp-controller arranges to encapsulate Lisp objects within an Objective-C object of type lisp-ptr-wrapper. Putting Lisp objects into lisp-ptr-wrapper objects and extracting them is automatic and is hidden from the developer who is using the lisp-controller object. We will first look at how the lisp-ptr-wrapper functionality is implemented. The class definition from ns-object-utils.lisp is:

```
(defclass lisp-ptr-wrapper (ns:ns-object)
  ((lpw-lisp-ptr :accessor lpw-lisp-ptr)
   (lpw-depth :accessor lpw-depth)
   (lpw-parent :accessor lpw-parent))
```

```
  (:metaclass ns:+ns-object))
```

The lisp-ptr-wrapper object keeps track of both the object and its parent. The lpw-depth slot was originally intended as a way to permit users to specify a maximum depth of expansion for objects, but I later reconsidered this because I couldn't find a good use for it. Maybe sometime in the future it will come back if the need arises.

```
(defun make-ptr-wrapper (ptr &key (depth 1) (parent nil))
  (let ((lpw (make-instance 'lisp-ptr-wrapper)))
    (setf (lpw-lisp-ptr lpw) ptr)
    (setf (lpw-depth lpw) depth)
    (setf (lpw-parent lpw) parent)
    lpw))
```

This function will look a bit odd to Lisp developers. Why not just provide initargs for all those slots and use make-instance with appropriate keywords? The answer is that the lisp-ptr-wrapper class is an Objective-C class. So calling make-instance with keywords will be translated into a call on a function whose name is determined in part by the names of the keyword arguments. This is a nice convention that makes it easy for Lisp developers to use existing Objective-C classes with specialized init functions, but is not really what we want or need here. We certainly could have created a C init function with the proper name, but all calls would have to provide all of the arguments. So we simply chose to create the make-ptr-wrapper function instead to allow keyword arguments to be optionally provided.

We already saw one use of lisp-ptr-wrapper objects in the ns-to-lisp-object function. It merely extracted the lpw-lisp-ptr and returned it. We'll see other uses as we go through other functions needed to support NSOutlineViews. The first method we'll discuss is #/outlineView:numberOfChildrenOfItem:.

```
(objc:defmethod (#/outlineView:numberOfChildrenOfItem: #>NSInteger)
                ((self lisp-controller)
                 (olview :id)
                 (item :id))
  (declare (ignore olview))
  (cond ((typep item 'lisp-ptr-wrapper)
         (length (children-of-object self (lpw-lisp-ptr item))))
        ((eql item (%null-ptr))
         (length (objects self)))
        (t
         0)))
```

This method makes use of the function children-of-object to retrieve a sequence that has a length and returns that object. If the specified item is nil, then what is being requested is the count of the top-level objects. Since they are cached in the objects slot, we can use it directly. Let's take a closer look at children-of-object:

```
(defmethod children-of-object ((self lisp-controller) obj)
  ;; Get the children of an instance of some type
  (let* ((obj-type (controller-type-of self obj))
         (child-key (assoc-aref (type-info self) obj-type :child-key))
         (children-object nil))
    (if (children-func self)
      (setf children-object (funcall (children-func self) (owner self) self obj))
      (if child-key
        (setf children-object (funcall child-key obj))))
    ;; if the children object is a hash-table, expand it into an ht-entry list
    (when (typep children-object 'hash-table)
      (setf children-object (children children-object)))
    (sort-sequence self children-object)))
```

In IB, the developer might have specified an override function to be called to get the children of an arbitrary object. If so we call it to determine the child object. Alternatively, the developer might have specified a child key to be used to retrieve the children of a particular type of object. If so we use that key. The final possibility is that neither of these was specified in IB and some default is used. Defaults are available for lists, vectors, and hash-tables. The children of a list or a vector is simply the object itself. That is, all elements of the sequence are presumed to be children.

The child object returned must be a list, a vector, or a hash-table. If it is a hash-table, then it is re-represented as a list of ht-entries. This conversion is handled automatically without the user ever needing to know that it has happened. See the *Hast-table Representation* section below for a more complete description of how hash-tables are represented.

The next method needed to support NSOutlineViews is #/outlineView:child:ofItem:.

```
(objc:defmethod (#/outlineView:child:ofItem: :id)
                ((self lisp-controller)
                 (olview :id)
                 (child #>NSInteger)
                 (item :id))
  (declare (ignore olview))
  (with-slots (obj-wrappers objects) self
    (cond ((typep item 'lisp-ptr-wrapper)
           (let* ((parent (lpw-lisp-ptr item))
                  (parent-depth (lpw-depth item))
                  (children (children-of-object self parent))
                  (child-ptr (elt children child)))
             (or (assoc-aref obj-wrappers child-ptr)
                 (setf (assoc-aref obj-wrappers child-ptr)
                       (make-ptr-wrapper child-ptr
                                         :depth (1+ parent-depth)
                                         :parent parent)))))
          ((eql item (%null-ptr))
           (let ((child-ptr (elt objects child)))
             (or (assoc-aref obj-wrappers child-ptr)
                 (setf (assoc-aref obj-wrappers child-ptr)
                       (make-ptr-wrapper child-ptr :depth 1 :parent nil)))))
          (t
           (%null-ptr)))))
```

This method requests the return of an Objective-C object that represents the Nth child, where N is and integer specified by the child parameter. If the parent object is a lisp-ptr-wrapper, we first extract the Lisp parent from the lisp-ptr-wrapper object that we are given. Then we find its children and extract the Nth child from that sequence. If the parent is nil, then we are being asked for a top-level child. We already have these cached in the objects slot, we we can directly find the Nth child.

In either case, we next want to return the child within its own lisp-ptr-wrapper. We either find such an object that we previously created or construct a new one and add it to the assoc-array that is in the obj-wrappers slot of the lisp-controller.

Outline views display a small arrow next to objects that can be expanded. To find out whether a displayed object can be expanded it calls the #/outlineView:isItemExpandable: method:

```
(objc:defmethod (#/outlineView:isItemExpandable: #>BOOL)
                ((self lisp-controller)
                 (olview :id)
                 (item :id))
  (declare (ignore olview))
  (cond ((eql item (%null-ptr))
         ;; root object
         #$YES)
        ((typep item 'lisp-ptr-wrapper)
         (if (children-of-object self (lpw-lisp-ptr item))
           #$YES
           #$NO))
         (t
          #$NO)))
```

The root object is always expandable, so if we are handed a null pointer the answer is always yes. Otherwise, if the object has children we say it is expandable and if it doesn't it is not expandable. Note that this may change if the developer provides a way to add children to an object. Not being expandable now does not mean that an object is never expandable.

NSOutlineViews also call methods to retrieve and set selected values. These are very similar to the NSTableView calls described earler. The former is implemented via the #/outlineView:objectValueForTableColumn:byItem: method and the latter by the #/outlineView:setObjectValue:forTableColumn:byItem: method.

```
(objc:defmethod (#/outlineView:objectValueForTableColumn:byItem: :id)
```

```
               ((self lisp-controller)
                (olview :id)
                (col :id)
                (item :id))
  (declare (ignore olview))
  (let ((ns-format (assoc-aref (column-info self) col :col-format)))
    (lisp-to-ns-object (col-value self (lpw-lisp-ptr item) col) ns-format)))
```

The item parameter provides the row object and the col parameter tells us which column we want to display. When the lisp-controller was initialized we gathered and/or constructed everything necessary to display each possible type of row-object in each possible table column. We earlier examined the col-value function and saw how it finds the right Lisp value and also examined how the lisp-to-ns-object function creates an appropriate Objective-C object for display.

```
(objc:defmethod (#/outlineView:setObjectValue:forTableColumn:byItem: :void)
               ((self lisp-controller)
                (olview :id)
                (val :id)
                (col :id)
                (item :id))
  (let* ((row-obj (lpw-lisp-ptr item))
         (old-obj (col-value self row-obj col))
         (ns-format (assoc-aref (column-info self) col :col-format))
         (new-val (ns-to-lisp-object old-obj val ns-format)))
    (if (writer-func self)
      (funcall (writer-func self)
               new-val
               (root self)
               row-obj
               (assoc-aref (column-info self) col :col-val))
      (set-col-value self row-obj col new-val))
    (when (edited-func self)
      (let* ((row (#/rowForItem: olview item))
             (edited-obj (if (typep row-obj 'ht-entry)
                           (list (ht-key row-obj) (ht-value row-obj))
                           row-obj)))
        (funcall (edited-func self)
                 (owner self)
                 self
                 (root self)
                 row
                 (assoc-aref (column-info self) col :col-val)
                 edited-obj
                 old-obj
                 new-val)))))
```

If the developer specified an override function to retrieve a column value, then we also may have constructed a corresponding function to set a new value. If so, we call it. Otherwise, we may have found a value setf form for a specified column and row object type. If so, we call set-col-value to invoke it. After the value has been modified, we then invoke a notification function that the Lisp developer may have provided in IB. This can take any additional action that is desired.

There are two lisp-controller methods that are called by virtue of its being a delegate for an NSOutlineView: #/outlineView:shouldEditTableColumn:item: and #/tableViewSelectionDidChange:. The second we already examined for NSTableViews. The first is:

```
(objc:defmethod (#/outlineView:shouldEditTableColumn:item: #>BOOL)
               ((self lisp-controller)
                (olview :id)
                (col :id)
                (item :id))
  (declare (ignore olview))
  ;; allow editing if there is a function available to setf a new value
  (if (or (writer-func self)
          (let ((obj-type (controller-type-of self (lpw-lisp-ptr item))))
            (cdr (assoc-aref (type-info self) obj-type col))))
    #$YES
```

```
    #$NO))
```

This is called by the NSOutlineView to determine whether editing of the selected row/column should be permitted. This method simply checks to see whether some method exists for writing a new value and if so, permits editing.

Hash-table Representation

Hash-tables are handled in a special way by the lisp-controller. They are effectively treated as a sequence of key/value pairs. Each pair is encapsulated as an ht-entry object. The class definition for this is:

```
(defclass ht-entry ()
  ((ht-key :reader ht-key :initarg :key)
   (ht-value :reader ht-value :initarg :value)
   (ht :accessor ht :initarg :ht))
  (:default-initargs
    :key (gensym "KEY")
    :value nil
    :ht nil))
```

In addition to the key and value, an ht-entry keeps a pointer back to the original hash-table from which it was derived. In this way, if the user modifies a key or value using the user interface we can translate that into an appropriate action in the original hash-table.

One of the things that we want to do is keep track of which hash-tables we have already turned into lists of ht-entry objects. Since this is both time and space consuming we wouldn't want to do it over and over again whenever the user interface requested the children of an object that happened to be a hash-table. To keep track of this we will use, appropriately enough, a hash-table which contains keys that are hash-table references and associated values which are a list of ht-entry objects. We encapsulate the function definitions for children, (setf children), add-to-child-seq, and delete-from-child-seq within a let that creates this hash-table. In that way each of those functions can modify the list of ht-entry objects and/or the hash-table they are derived from.

```
(let ((ht-hash (make-hash-table)))
  ;; in order to treat hash-tables as containers like lists and vectors we
  ;; need to define a few functions which use a cache of the "children" of
  ;; a hash-table so that we don't need to recreate the whole list every time
  ;; a new child is added

  (defmethod children ((parent hash-table))
    (or (gethash parent ht-hash)
        (setf (gethash parent ht-hash)
              (let ((ht-list nil))
                (maphash #'(lambda (key val)
                             (push (make-instance 'ht-entry
                                     :key key
                                     :value val
                                     :ht parent)
                                   ht-list))
                         parent)
                ht-list))))

  (defmethod (setf children) (new-value (parent hash-table))
    (setf (gethash parent ht-hash) new-value))
```

The new-value parameter will be a list of ht-entry objects. So we just set the value corresponding to the parent hash-table in ht-hash to this list.

```
  (defmethod add-to-child-seq (parent (seq list) (thing ht-entry))
    (with-slots (ht ht-key ht-value) thing
      (setf (gethash ht-hash parent) (cons thing seq))
      (setf ht parent)
      (setf (gethash parent ht-key) ht-value)))
```

To add a child (which will be an ht-entry instance) we first just add it to the value corresponding to the parent hash-table used as a key in ht-hash. Then we set the ht field in the ht-entry so that it now points to the correct parent hash-table. Finally we actually add the new entry to the parent hash-table.

```
  (defmethod delete-from-child-seq ((seq list) (thing ht-entry))
    (with-slots (ht ht-key) thing
      (remhash ht-key ht)
      (delete-from-list seq thing)))
```

To delete a child (ht-entry) we remove it from the list and also remove the corresponding entry from the parent hash-table.

```
) ;; end of hash-table functions within let
```

The methods for making changes to existing ht-entry objects are shown below.

```
(defmethod (setf ht-key) (new-key (self ht-entry))
  (block set-it
    (let ((new-key-exists (not (eq :not-found (gethash new-key (ht self) :not-found)))))
      (when new-key-exists
        ;; They are redefining a key to be one that already exists in the hash-table
        ;; first verify they want to do this
        (let* ((alert-str  (lisp-to-temp-nsstring
                            (format nil
                                    "Continuing will reset value for existing key: ~s"
                                    new-key)))
               (res (#_NSRunAlertPanel #@"ALERT"
                                       alert-str
                                       #@"Cancel key change"
                                       #@"Continue and change key"
                                       (%null-ptr))))
          (unless (eql res #$NSAlertAlternateReturn)
            ;; they don't want to continue
            (return-from set-it))))
      ;; change the value for the existing key
      (setf (gethash new-key (ht self))
            (gethash (ht-key self) (ht self)))
      ;; and then remove the old key that was changed both from the hash table
      ;; and from the list of keys
      ;; new keys are always put at the end of the list unless a sort predicate
      ;; has been specified.
      (remhash (ht-key self) (ht self))
      (setf (slot-value self 'ht-key) new-key))))
```

This method is called if the user has modified a displayed field that corresponds to the key of a hash-table entry. This is perhaps an unusual thing to want to do, so the function displays an interactive dialog that requests the user to verify that this is actually what they want to do. If so, then the effect is the same as:
1. adding a new hash-table entry with the the new key and an associated value that is the same as that held by the old key
2. deleting the hash-table entry the corresponds to the old value of the key.
These changes are reflected both in the list of ht-entry objects and in the hash-table from which they are derived.

```
(defmethod (setf ht-value) (new-val (self ht-entry))
  (setf (gethash (ht-key self) (ht self)) new-val)
  (setf (slot-value self 'ht-value) new-val))
```

Setting the new value of an ht-entry is more straightforward. It sets the slot in the ht-entry as you might expect and has the side-effect of changing the corresponding value in the hash-table from which this entry was derived.

Support for Insertion and Deletion of Child Objects

Three additional methods exist to support:
1) The insertion of a new child into the root object - #/insert:
2) The insertion of a new child into the currently selected object - #/addChild:
3) The deletion of a selected object - #/remove:

```
(objc:defmethod (#/insert: :void)
                ((self lisp-controller) (button :id))
```

```
  (declare (ignore button))
  ;; insert a new object into the root object
  (unless (root self)
    ;; need to create a root object
    ;; may still be null if root type is 'list
    (setf (root self)
          (new-object-of-type self (root-type self))))
  (let ((new-children (add-child-to self (root self))))
    (when (null (root self))
      ;; special hack for root objects that are lists and may be null
      (setf (root self) new-children)
      (setf (objects self) new-children)))
  (#/reloadData (view self)))
```

The real work of this method is accomplished with a call to add-child-to. There are some special considerations needed both before and after this call. Prior to calling it we want to make sure that the root object has been created. So if it is currently nil (which may be a valid value and may not be), we go ahead and create a new root object. The function add-child-to will return a list of children that includes a new child object. Typically the new child is spliced into the existing list of children for the root, so there will not be any additional processing required. However there is one special case that needs additional handling. If the root object started out as nil, then it is necessary to set both it and the objects slot to have the value of the new list of children (which will have only a single object). Finally this function tells the table to reload itself so that the new value will be reflected correctly.

Now let's examine the add-child-to method:

```
(defmethod add-child-to ((self lisp-controller) parent)
  (let* ((parent-type (controller-type-of self parent))
         (child-type (assoc-aref (type-info self) parent-type :child-type))
         (child-key (assoc-aref (type-info self) parent-type :child-key))
         (child-initform (and child-type (assoc-aref (type-info self) child-type :initform)))
         (set-child-func (assoc-aref (type-info self) parent-type :child-setf-key))
         (new-children nil)
         (new-child nil))
    (if (and child-type child-key child-initform set-child-func)
      ;; we've got everything we need to set the child ourselves
      (let ((children (funcall child-key parent)))
        (setf new-child (eval child-initform))
        (setf new-children
              (funcall set-child-func
                       (add-to-child-seq parent children new-child)
                       parent))
        (when (subtypep (controller-type-of self parent) 'hash-table)
          (setf (ht new-child) parent)
          (setf (gethash (ht-key new-child) parent) (ht-value new-child))))
      ;; else see if there is a user-specified function to add a child
      (when (add-child-func self)
        (setf new-children (funcall (add-child-func self) parent))
        (when new-children
          (setf new-child (elt new-children (1- (length new-children)))))))
    (when (added-func self)
      ;; notify by calling the function specified in IB
      (let ((last-child (if (typep new-child 'ht-entry)
                          (list (ht-key new-child) (ht-value new-child))
                          new-child)))
        (when last-child
          (funcall (added-func self)
                   (owner self)
                   self
                   (root self)
                   parent last-child))))
    (sort-sequence self new-children)))
```

First off, all the information needed to create a new child is gathered together to assure that we have everything needed. That includes:
1) The type of the parent object because that determines what type of child it will have

2) The type of the child, given the type of the parent
3) The key that is used to retrieve the children from the parent
4) The initform for the type of the child
5) The function used to set the children of the parent

If we have all of those pieces, then we go ahead and create a new child and add it to the list of children for the parent. If the new child is an ht-entry we set the parent hash-table for it and make sure that the parent hash-table is updated.

If we don't have all those pieces, but do have an override function that can be called to add a new child, then we call it.

Next, if the developer identified a notification function that was to be called after a new child was added, then we go ahead and call it.

Last, but not least, we sort the sequence of children so that the new child will be displayed in the correct location relative to other children and return that list from the function.

The #/addChild: method is similar to the #/insert: method, but will only add a child to the currently selected object. So it is never called to add a child to the root. Therefore we can ignore the special case considerations that were needed in the #/insert: method to handle root peculiarities. That makes this method much simpler and should be self-explanatory.

```
(objc:defmethod (#/addChild: :void)
                ((self lisp-controller) (button :id))
  (declare (ignore button))
  ;; add a new child to the currently selected item
  (let* ((row-num (#/selectedRow (view self)))
         (parent (object-at-row self row-num)))
    (add-child-to self parent))
  (#/reloadData (view self)))
```

Finally we will consider the method called to delete selected objects:

```
(objc:defmethod (#/remove: :void)
                ((self lisp-controller) (button :id))
  (declare (ignore button))
  (let ((row-num (#/selectedRow (view self))))
    (multiple-value-bind (child parent) (object-at-row self row-num)
      (when parent
        (remove-child-from self parent child)
        (#/reloadData (view self))))))
```

The bulk of the work done here is by the call to remove-child-from, so let's have a look at that function;

```
(defmethod remove-child-from ((self lisp-controller) parent child)
  (let* ((parent-type (controller-type-of self parent))
         (child-key (assoc-aref (type-info self) parent-type :child-key))
         (set-child-func (assoc-aref (type-info self) parent-type :child-setf-key))
         (parent-is-root (eq parent (root self)))
         (new-children nil))
    (if (delete-func self)
      (setf new-children (funcall (delete-func self) (owner self) self parent child))
      (when (and child-key set-child-func)
        (let ((children (funcall child-key parent)))
          (setf new-children
                (funcall set-child-func
                         (delete-from-child-seq children child)
                         parent)))))
    (when (and parent-is-root (null new-children))
      ;; The only time this actually does something is when the
      ;; objects were a list and we just deleted the last child.
      (if (listp parent) (setf (root self) nil))
      (setf (objects self) nil))
    (when (removed-func self)
      (funcall (removed-func self) (owner self) self (root self) parent child))
    (sort-sequence self new-children)))
```

If the developer specified an override function to delete objects, then we just call that function and let it return a list of new children that we'll use. Otherwise, assuming that we can gather up all the necessary functions we will call the delete-from-child-seq function to remove the child object. If the parent of the selected object is the root object, then it is possible that we are removing the very last object from the root. In that case only we may need to set the root object if it was a list and to set the objects slot to nil as well.

Finally we will call a delete notification function if one was specified and then re-sort the new list of children and return it.

The delete-from-child-seq function is implemented somewhat differently for different types of sequences. We already looked at the version used when the children are a list of ht-entry objects. For vectors, the object is removed and the remaining elements of the vector are moved down. For lists, the object is removed in a way that leaves all references to the list valid (as long as the last item in the list isn't removed). The reader is invited to find these functions and examine them to see exactly how they are implemented.

That's it for the projects so far. Others topics that need corresponding projects include ...

Stand-Alone Applications, Array Controllers, KVC paths, Quartz Graphics, OpenGL Graphics, Core Data,

Debugging Hints

1. Although you can redefine Objective-C methods at runtime, there are occasions when the presence or absence of such a function is cached. If, for example, you add a new delegate method for a class D, don't expect that method to be called for any already instantiated example of class D. That's because any object that has a D-class object as its delegate will, at the time the delegate outlet is set, determine which delegate methods D-class objects support and never call any others. If you create a new instance of D, and it is made a delegate, then your new method will be called.

2. If you run into Objective-C runtime exceptions that make it necessary to "Force Quit" CCL, it's likely that what you have done is prematurely release some Objective-C object. If you unnecessarily retain one you may get a memory leak, but that typically does not cause a runtime problem. Premature releasing, on the other hand, will almost surely cause you grief. If you prematurely release an object that has slots which are bound somehow, KVC will log a message indicating that this happened. Open up the console application to see these messages.

3. If you add a format statement in your code somewhere to help with debugging, the output may or may not go to the listener window. If the function is called in an Objective-C method it may display in the AltConsole window associated with this Lisp session. But as discussed above, using format statements within Objective-C functions should probably be avoided altogether. Replace them with calls to #_NSLog instead

4. Memory management of Objective-C objects can be a fairly tricky proposition at times. I discovered that objects are sometimes retained in places where you might not expect. For example, any direct or indirect use of the #/window function for window controller objects (or any object derived from it) results in an increase in the reference count of the window. I suspect this is actually a bug in Apple's code, but who knows, there may be some undocumented reason for this. One technique that can help resolve issues like this is to use the #/retainCount function which will tell you what the current count is for any Objective-C object. Apples discourages its use because it doesn't say anything about what objects did those retains, but I found it useful at times.

5. If some runtime event results in calling a lisp function which errors in some way you will typically see some sort of error in the AltConsole window. Sometimes just doing a simple :pop command will get you out of the situation, but it is easy to get into a situation where you just re-trigger the error. To get out of this situation type :R to find available restarts and pick the one that just goes on to the next event as in the following dialog.

```
> Error: value #1=((2 3 2 2) . #1#) is not of the expected type (SATISFIES CCL::PROPER-LIST-P).
> While executing: (:INTERNAL LISP-CONTROLLER::I-[LispController numberOfRowsInTableView:]I), in process
Initial(0).
> Type :POP to abort, :R for a list of available restarts.
> Type :? for other options.
1 > :r
>  Type (:C <n>) to invoke one of the following restarts:
0. Return to break level 1.
1. #<RESTART ABORT-BREAK #x493A1D>
2. Process the next event
1 > (:C 2)
Invoking restart: Process the next event
```