

CCL Cocoa Developer Tools Tutorial

Version 1.1 February 2011

Copyright © 2011 Paul L. Krueger All rights reserved.

Paul L. Krueger, Ph.D.

Overview

What I'm trying to achieve

Fundamentally the tools described in this document were created to expedite the process of creating *Document-Based* Lisp/Cocoa applications. My intention was to create a workflow that would be familiar to Lisp developers. What that means to me is that I don't have to sacrifice the very interactive development and debug process that is familiar to Lisp programmers. This is very different from the Xcode paradigm used by C, C++, Objective-C, Ruby, ... developers on Macintosh platforms. For those developers there are distinct program / build / run & debug stages through which the developer iterates. In Lisp one can make a program change and (for the most part) immediately test the results of that change. All of the stages are combined in a single interactive environment, making the developer much more productive. For those of you who are seasoned Lisp developers I'm not saying anything that you don't already understand. For those who haven't tried it yet, let me say that this brief discussion doesn't begin to do justice to the benefits of Lisp for rapidly prototyping new applications.

What may not be so familiar to Lisp developers is all of the machinery needed to develop document-based Cocoa applications. There is a wealth of documentation available from Apple and I will make no attempt to even summarize all of it here. What I will do is cover just enough for developers to use the tools provided to create their own document-based applications.

All of this work is built on top of the CCL IDE and depends on it. If you can't run that on your platform, then this code will be of no use to you. The processes described throughout this document will help a developer build an application which evolves from bits of isolated Lisp code to a full-blown stand-alone application that can be executed just like any other on the system. Along the way we'll create documents with all that that entails, then add interactive windows that can be used to display and/or edit the document, and finally incorporate menus to provide additional user controls.

It is not trivial to run a complicated document-based application inside of the existing CCL IDE, but I have made an attempt to make this look as natural as possible. There are still a few limitations to that environment, but they are actually fairly minor. Once the developer gets to the point where those limitations matter, it is possible to create a stand-alone executable that contains the REPL, and the AltConsole, but which otherwise IS the target application. That permits further debugging. Once the application functionality has been completely debugged, a stand-alone application without any IDE functionality can be built and executed. After experimenting with this a bit, my current belief is that the stand-alone code that includes IDE resources will typically not be needed. Nevertheless, it is supported if you want to create one.

The tools always attempt to do something reasonable at each stage of development. For example, if you haven't yet created a main menu for your application, then the tools will provide some simple menu items that will let you create and load your documents. For some applications this will be enough to do the bulk of debugging that is needed. Once you DO create your own application main menu, the tools will permit you to toggle back and forth between the CCL menus and your application's menus or even see both at once if you like.

If you have not yet defined a window for your document, the tools will provide a proxy window so you can at least see it pop up and check what happens to your document when that window is closed. If you have not used any of the print mechanisms provided by the tools to print your specific document type, then the tools will provide a default print mechanism that just prints out the slot values of your document. The idea is that you will be able to progressively add features to your document-based application and be able to test it at any stage using defaults that make some sense.

While the tools provide an interface for routine application development tasks, more complex applications may still need to use other Apple tools. In some cases the Lisp tools provide a way to execute those other Apple tools and automatically re-import changes made within them. An example of this is being able to invoke Apple's Property List Editor to do advanced editing of an application's Info.plist. If you don't know what that is, don't worry about it. We'll talk more about it later and you may never need to know more about it than what is visible from a lisp document window.

Apple espouses something called the *Model/View/Controller (MVC)* paradigm. The user is encouraged to read Apple's documentation on the topic, but roughly speaking you can think of a *Model* as those objects and data structures that represent the state of the document. These are the things that you might want to save in a document and open again later; i.e. the data. The *View* consists of those objects that are used to provide an interface between your document and a user so that they can view and/or edit it. View objects include windows and their various sub-views as well as controls of all sorts both within windows and in menus. *Controllers* are those objects that act as a conduit between the Data objects and the View objects. Of necessity they need to understand and access the structures of both Data and View objects.

The general idea behind this paradigm is to be able to develop each of these distinct areas of functionality independently and to do so in a way that makes re-use easy. For the most part, all View objects have been provided by Apple as part of Cocoa. It is possible for a developer to create new types of view objects, but this is unnecessary for most non-graphical applications. For applications that need to perform graphics operations (and I will provide an example of this in this document) it will be necessary to create a custom view class and call Objective-C methods to do the necessary work. While it would certainly be possible to create lisp methods to hide all of the Objective-C syntax, I don't view this as a desirable thing to do. I want to give developers complete access to all Cocoa functionality and the ability to use Apple's documentation in a very direct way. Of course any developer who wants to create their own graphics layer on top of the Cocoa calls is free to do so.

One objective of this work is to make it possible for developers to define all of the Model objects as standard Lisp objects that are manipulated with standard Lisp methods. I'll let you be the judge of how successful I have been at accomplishing that objective. To that end I have introduced some classes that hide much of the complexity that would otherwise be required in a Lisp Model layer. Those classes include *lisp-document*, which should be inherited by any Document class that you create for yourself, and a couple of *application delegate* classes that will be discussed later. Since data is represented in the Objective-C world by a variety of Objective-C object types, it is frequently necessary to convert back and forth from Lisp to Objective-C data structures. I have provided a number of different Lisp functions that do this conversion and for the most part this is done automatically without the developer ever having to be aware that it occurred. When manual conversion is required by a developer who wants to directly utilize Objective-C functions and needs to provide parameters to them, the use of conversion functions is very straight-forward.

That leaves the Controller layer. Apple also provides a number of standard Controller objects, but these are not so useful to Lisp developers because for the most part they assume that Data objects are standard Objective-C objects. With care it is possible to use them, but I found this to be so taxing that I created a *lisp-controller* class that combines the capabilities of several of Apple's controller objects into a single controller. This plays an important role in making it possible to create a Model layer that is standard Lisp.

For those who only want to understand the role of the lisp-controller and how it makes things like binding from view objects to slots within ordinary Lisp instances, you will want to read the "LispController Ref". For those who want a deeper understanding of exactly how the List Meta-Object Protocol (MOP) was used to implement that binding, refer to the "Lisp-KVO Reference & Tutorial 1.1". Those who want a better understanding about how to use Apple's Interface Builder (IB) with Lisp should read the "InterfaceBuilderWith CCL Tutorial3.0" document.

The remainder of this document will describe what tools are available and provide examples of how to use them. It will NOT describe the internal implementation of the tools themselves other than what is provided in the other documents referenced in the last paragraph. At some future point I may add such a description to the *Cocoa Interfaces Using the Apple Interface Builder (IB) and Clozure Common Lisp (CCL)* tutorial if there is sufficient demand. The tools implementation itself is a document-based application that is run under the IDE and uses many of the capabilities that we'll be discussing here, so it provides a (currently undocumented) third example of what sorts of things can be done. Hopefully once this tutorial is completed the reader will be able to look at the source code for the tools and understand how they make use of the same functionality.

Lisp App Documents (.lapp)

Cocoa developers who use languages other than Lisp use Xcode to define and build their applications. An Xcode project contains a complete description of an application including what source code is included, how and where it is built, what other resources are needed, etc. It saves all that information into an Xcode project document. The Lisp Tools described in this tutorial create a similar sort of document for Lisp applications, although what needs to be saved is quite different of course. When the tools are loaded into the CCL IDE you can create a new Lisp App document or open one that was previously saved, just as you can with any other document. A single window is provided to edit these documents. This window is shown in Figure 1 below. For convenience, all windows (other than panels) shown in this document are at 60% of actual size.

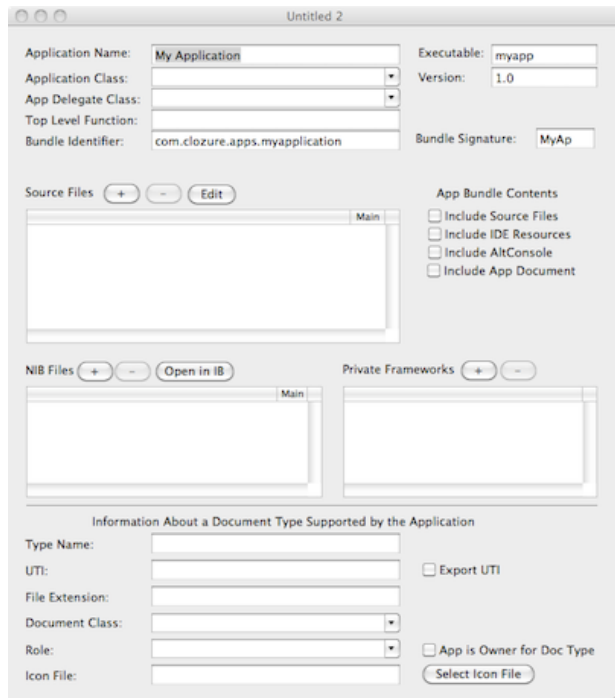


Figure 1 A Lisp Application Document Window

As we move from simple to progressively more complex Lisp applications we'll use more and more of the fields shown in this window.

Application Bundles

Applications on Apple systems are contained in *bundles*. A bundle is basically a directory (folder) that is displayed in the finder as if it were a single discrete file. Bundles can be moved around and otherwise treated just as if they were single files. There are bundles of various sorts used for different purposes. For example, NIB files are actually bundles.

Application bundles have a specified sub-directory structure and some required files including the actual executable file that is run when the application is opened by a user. All of this will be discussed as needed below. For now, just recognize that as the application evolves from snippets of Lisp code to a complete stand-alone application, we'll gradually add more things to our application bundle. The developer tools will allow you to create and populate a corresponding application bundle without requiring that you understand too much about what is in it. For the curious, you can open up a bundle in the finder and view it just like any other folder by control-clicking on an application file and selecting "Show Package Contents" from the pop-up menu. Once you do that you can move files to or from a bundle just as you would for any other folder that you can view in the finder.

The Lisp Application document keeps track of where its corresponding bundle is located. You can create a new bundle at any time just by removing or renaming the old one. As the tools change the contents of a bundle, they may modify existing contents, but they never remove anything that was there previously. This permits advanced users to manually move additional resources into a bundle without fear that the tools will remove them. The downside, of course, is that if you WANT to remove something from a bundle, you must do so manually as well or start all over with a new bundle.

It is also possible to create a new Lisp Application Document and associate an existing application bundle with it. The Lisp Application will adjust its parameters to reflect what it finds within that bundle. In this way, you can duplicate an existing bundle to get a head-start on a new application and then modify it as necessary.

We'll have more to say about bundles when we get to the point where they are needed. For our purposes that will come when we add a document window to the application.

Getting Started with Lisp App Tools

Initializing the tools in the IDE

Initializing the Lisp App Developer Tools within the CCL IDE is pretty easy. CCL now sets up logical pathname translations that will find anything in any contrib subdirectory, so the following should work out of the box:

```
Welcome to Clozure Common Lisp Version 1.7-dev-r14466M-trunk (DarwinX8664)!
? (require :lisp-app-doc)
;Compiler warnings for "/Applications/ccl/cocoa-ide/builder-utilities.lisp" :
; In MAKE-DOCTYPE-DICT: Unused lexical variable EXPORTABLE-AS
;Compiler warnings for "/Applications/ccl/cocoa-ide/builder-utilities.lisp" :
; In MAKE-DOCTYPE-DICT: Unused lexical variable BUNDLEP
;Compiler warnings for "/Applications/ccl/cocoa-ide/builder-utilities.lisp" :
; In MAKE-DOCTYPE-DICT: Unused lexical variable ICON-FILE
;Compiler warnings for "/Applications/ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Utilities/save-panel.lisp" :
; In SAVE-PANEL: Undefined function NEXTSTEP-FUNCTIONS:|setNameFieldStringValue:|
; In SAVE-PANEL: Undefined function NEXTSTEP-FUNCTIONS:|setDirectoryURL:|
;Compiler warnings for "/Applications/ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Utilities/open-panel.lisp" :
; In OPEN-PANEL: Undefined function NEXTSTEP-FUNCTIONS:|setDirectoryURL:|
:LISP-APP-DOC
("BUILDER-UTILITIES" "FILE-MONITOR" "NS-STRING-UTILS" "ASSOC-ARRAY" "NS-BINDING-UTILS" "UNDO" "KVO-SLOT" "NSLOG-
UTILS" "DATE" "DECIMAL" "NS-OBJECT-UTILS" "LIST-UTILS" "ALERT" "LISP-CONTROLLER" "WORDS" "OPEN-PANEL" "SAVE-PANEL"
"CLASS-CONVERT" "LISP-APP-DELEGATE" "SELECTOR-UTILS" "NIB" "MENU-UTILS" "LISP-DOC-CONTROLLER" "LISP-BUNDLE" "LISP-
DOCUMENT" "LISP-APP-WIN-CONTROLLER" "UTILITY" "LISP-APP-DOC")
?
```

Some of the warnings are generated by loading the CCL file builder-utilities.lisp. The warnings from my save-panel.lisp file are caused by the inclusion of calls with definitions in OSX 10.6, but not included in the Objective-C library available in CCL 1.7. They aren't actually called because the function checks to make sure that the target will respond to them before making the call. There are probably ways to work around this with proper compiler directives.

At the end of the lisp-app-doc file it calls (ad::install-lisp-app-tools) which installs the tools for you. In effect, the tools are an application that is being run under the IDE. So if you are looking for a fairly complex example of what can be done inside the IDE, then have a look at the code in the directory: .../ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Cocoa Dev.

If you get to the point where you want the tools to be routinely available when you start up the CCL IDE, then you can easily accomplish this by creating a ccl-ide-init.lisp file in your home directory. The one in mine has the following content:

```
;; ccl-ide-init.lisp

;; Put this in your home directory to make sure the that the CCL logical directory points
;; to the right place on your system. Then you can move the application bundle wherever you
;; want and anything that needs access to ccl: will (e.g. using #/ constructs in the
;; listener) will work properly.
(ccl::replace-base-translation "ccl:" (truename "/Applications/ccl"))

;; Load and install the Developer tools, but only in the IDE and not in any other
;; application bundle that might also load this init file
(let* ((bundle (#/mainBundle ns:ns-bundle))
      (id (ccl::lisp-string-from-nsstring (#/bundleIdentifier bundle))))
  (when (and (> (length id) 23)
            (string= "com.clozure.Clozure CL-" (subseq id 0 23)))
    (handler-bind ((warning (lambda (c)
                              (muffle-warning c))))
      (require :lisp-app-doc))))
```

This first changes the translation to the ccl: logical directory to the actual location on my system. The default way of setting this translation is to set it to whatever directory contains the IDE bundle. Since by default that IS the CCL directory everything works fine. But later when we create stand-alone applications that include IDE resources, we may well wish to put those bundles in some other directory. There are some things that you may wish to do in the finder that load resources from the ccl directory (e.g. invoking Objective-C functions using a #/ syntax). Unless we assure that it is set correctly, those will fail. If your ccl directory is not in /Applications, you will of course want to modify the code to point to the correct location.

The next thing that is done is to load and install the tools by requiring :lisp-app-doc. The handler-bind form surrounding it is necessary to cause the warnings that you saw above to be ignored. We also make sure that we only load the tools for the IDE itself and not any other application that we may later create which uses the same top-level function (and which therefore loads the ccl-ide-init.lisp file). We do that by making sure that the tools are only loaded for the particular bundle that contains the IDE.

Requiring :lisp-app-doc makes available everything described in the remainder of this document.

Tool Description

There is a single window which is used to edit a **Lisp Application Document (LAD hereafter)**. That is the window shown in Figure 1 above. In addition, installing the tools results in the addition of a new menu: the Dev menu shown in Figure 2 below.

Dev	
New Bundle	⌘ B
Initialize Bundle	⌘ I
Use Bundle ...	⌘ U
Edit Info.plist	⌘ P
Install Executable	⌘ E
Load App Under IDE	⌘ L
Run App Stand-Alone	⌘ R
Create .h For Interface Builder	▶
Create Lisp Source From .h ...	⌘ S
✓ CCL Menus	⌘ C
App Menus	⌘ A

Figure 2: The Dev menu

I'll give a brief description of the function of each selection here and additional discussion of when and how they are used will follow later.

Menu items affecting the application bundle

New Bundle: Used to create a bundle and initialize it with the latest information from your LAD

Initialize Bundle: Used to update an existing bundle with the latest information from your LAD

Use Bundle: Used to attach an existing bundle to a LAD. Document information will be changed to reflect what is specified in the bundle.

Edit Info.plist: This will make sure the Info.plist in the application bundle is updated with current information from the LAD window and then open that Info.plist file in Apple's Property List Editor application. Whenever that document is subsequently saved, changes made will automatically be re-imported into the LAD and reflected in the window. An "undo" item will appear in the Edit menu that will allow you to revert the Info.plist back to the state it had prior to the re-importation of the changes.

Install Executable: Put an executable into the application bundle. This executes a subordinate CCL instance and interactively tells it how to initialize itself and then save itself as an executable in the appropriate place within your application bundle. By doing this interactively rather than just launching lisp blindly with arguments telling it what to do, we can detect problems that occur (for example as a result of loading user files) and display an appropriate alert window so the developer can see what happened.

Menu items for running the application

Load App Under IDE: Does whatever is reasonable given the state of the LAD to load the application defined by the currently selected LAD window. This may include loading source files, loading a MainMenu NIB, creating additional menuitems for your application, and creating a lisp-doc-controller to manage your application inside the CCL IDE.

Run App Stand-Alone: This launches your stand-alone application just as if you had double-clicked on it within a Finder window.

Menu items for transferring files to and from Interface Builder

Create .h For Interface Builder: Generate a .h file for a selected Lisp class which can be imported into IB to provide actions and outlets for a Lisp class that you wish to use in IB. The sub-menu of this choice lists all known Objective-C subclasses and lets you select one for which you want to create a corresponding .h file that can be loaded into Interface Builder and used there.

Create .lisp Source From .h: Create a prototype Lisp Defclass form for any class that you defined in IB and exported and opens up a standard document window containing that form in the CCL IDE.

Menu items for manipulating displayed menus

CCL Menus: This toggles the inclusion of CCL menus in the menubar. When you get to the point of wanting to see what your application might look like and have a complete Main Menu defined for it, you can select this to remove CCL menus. The Dev menu is never removed so that you can always get back to whatever menu set you desire. CCL menus are automatically restored if an application window with displayed menus of its own is closed.

App Menus: This toggles the inclusion of application menus for the current LAD. If a Main Menu NIB is loaded as the result of selecting "Load App Under IDE" in the Dev menu, then toggling the App Menus will alternately remove or add the Application menubar to the currently shown menubar. In this way you can see either or both of the CCL and Application menubars with the Dev menu always shown in either case. Note that the leftmost Menu will always be labelled with "Closure <something>" regardless of the label provided in the NIB.

There are also three new menu choices added to the standard CCL File menu: "New Lisp Application", "Open Lisp Application", and "Print Lisp Application". As you might expect, these allow you to create a new LAD, open up a saved LAD, and print a LAD respectively.

The Squiggles Example

"Squiggles" is a sample application provided by Apple. It is a simple graphical application. It is NOT a document-based application and does not provide any way to save anything that is created. It also does not provide any sort of undo functionality or printing capability. We will convert this application to Lisp and use it as a running example of how to progressively refine an application to make into a stand-alone application that can save and open and print squiggle documents. Throughout the remainder I will refer to the Lisp version of this application as Squiggle (singular) to distinguish it from the Apple sample application Squiggles (plural).

The Apple Objective-C project can be found at <http://developer.apple.com/library/mac/#samplecode/Squiggles/Introduction/Intro.html>. The interested reader can download it and examine it in Xcode, but that is not necessary to do so in order to follow along with the Lisp example that is provided here. I selected this example for two reasons. First, it is a graphical application and demonstrates how to get started using Cocoa graphics in a Lisp application. Second, it combines the Model and Controller functionality into a single class. For many applications this is sufficient and it is not necessary to create one or more separate window controller classes. I wanted to demonstrate how to make that work with the Application Development Tools and show that this does not prohibit creating a document-based application.

Throughout the remainder of this document I will evolve the Squiggle example to show how a developer might expand it from some simple Lisp Code to a complete stand-alone document-based application complete with undo and print capabilities.

The Loan Example

NOTE: In order to be able to check the tools into the Clozure SVN repository as soon as possible, I have elected to omit the Loan example from this first version of the Tutorial. It will be added with the next update.

Developers who have worked through my InterfaceBuilderWith CCLTutorial3.0 document will recognize this example as a recurring theme there. In that document I progressively added new capabilities to demonstrate how things actually work under the hood. While familiarity with that example may be useful, it is not strictly necessary to follow along here. Using the new tools described in this document, implementation of the Loan example is much simpler and much more lisp-like. For the reader who wants to better understand what might be going on behind the scenes, it would still be useful to work through the tutorial. For the reader who just wants to get on with building a Lisp application and not worry too much about how it works, this document should be sufficient.

Rather than confuse the reader by using two separate examples throughout the main functional tutorial, this example will be presented in its entirety at the end of this document. It will demonstrate a few additional features that are not seen in the Squiggle example.

Creating / saving / opening lisp-app-doc documents (LADs)

To open a new LAD, select "New Lisp Application" from the CCL File menu. A window will open showing a newly initialized LAD. LADs can be saved using normal menu options from the File menu. LADs will be saved in a file with a .lapp extension. To open a saved LAD, use the "Open Lisp Application" selection in the File menu. Note that without changes to the CCL Application bundle's info.plist, it will not be possible to double-click a .lapp file and have it open automatically in the CCL IDE. Perhaps at a later date I'll add a section in the Advanced Topics section of this document that describes how to create a new version of the IDE that has this ability. Or perhaps by the end of reading this document the user will be able to figure out for themselves how to do that.

To follow along with the Squiggle example, at this time you can select "New Lisp Application" from the File menu. The window you initially see will look like Figure 1 above. Change the fields so that they look like Figure 3 below:

Figure 3: Beginning the squiggle application document

The application name will determine the name of the bundle when we eventually create it. Changing this field at any time will result in an automatic re-naming of the application bundle on disk. Most of the other changes indicated here are not yet used and will be discussed as they become relevant. Save this document to disk and give it a name that you can remember. Remember, at this point we are only saving the equivalent of an Xcode project file that describes our application, not the application itself.

Managing Lisp Source

I have provided the ability to include Lisp source files inside an application bundle. Why? My thinking was that by doing this, such a bundle could be moved to a new platform where the LAD could be opened (it can also be included as a resource in the bundle) and the source could be loaded from

the bundle into the CCL IDE on that platform. Then a new executable could be added to the bundle. This has not been tested, but the possibility exists. There is no problem with creating a stand-alone application that does not include source code in the bundle. Whether or not source code is copied to the bundle is controlled by the "Include Source Files" checkbox in a LAD window. If you DO decide to include source files within the bundle, I recommend that you create the bundle in a directory that CCL will not search via any known load path. That's because it will find your source within the bundle and if there are two legal paths to a single required file, CCL will ignore both of them and tell you that there is no way to load the file.

When a bundle containing source files is loaded (as is done when you ask the tools to run your application under the IDE) the source files will be loaded into Lisp. As a preamble to this the tools will set up a bundle-specific logical directory (named the same as the bundle) that can be used to designate a directory in load or require statements. For now I suggest that you do not include source within your bundle, but just manage it in whatever fashion you normally use. It is never *necessary* to include source files within bundles.

However you *do* want to specify what source files your application uses by adding them to the Source Files table so that the tools assure that they are properly loaded before running your application either within the IDE or as a stand-alone application. If you have a single source file that requires or loads all others that are needed, then that is the only file that must be included in the source file table (unless you also want to save everything in the bundle, in which case all files that you want included must be added to the table).

Currently there is NO other integration of source management or definition tools (e.g. asdf). That's probably a good project for someone who is ambitious.

Loading and testing source files under the IDE

To include a new source file, simply click on the "+" button above the source file table in a LAD window and a standard open dialog will pop up to allow you to select which document you want to add. You can do this as many times as you want. You may choose to designate a "main" source file by double-clicking in the Main column in the corresponding row of the source code table and then typing anything at all (other than a space). A main source file is presumed to require or load all other source required for your application. Or you may leave the Main column blank for all source files in the table. When you hit return or otherwise complete editing of the Main field for a source file the word "YES" will appear in the column. You can have only one main source file, so if you change some other source file to be main, the first will no longer indicate that it is main.

When you select "Load App Under IDE" from the Dev menu the source files will be loaded either from the bundle (if you elect to include source files in the bundle) or from their original locations (if source files are not included in the bundle). If source files are included in the bundle, then you *must* designate a Main source file which is the only one loaded. If you do not designate any of the source files as Main, then the first one in the table is assumed to be the Main source and will be the only one loaded. If source files are not included in the bundle, then the Main source will be required if it exists. Otherwise each of the individual source files will be loaded in the order in which they appear in the table.

Source files are only loaded if none of the Objective-C classes which they define already exist. It is not possible to redefine Objective-C classes at runtime, so the tools will not load any of the source if they already exist. This is one of the unfortunate aspects of the Objective-C runtime. If you need to redefine an Objective-C class, then you must quit the IDE and restart it before reloading the source files. You can, however, reevaluate the definition of Objective-C methods at runtime.

Once source has been loaded into the IDE, any normal Lisp actions can be taken in the Lisp listener window just as you might for any other Lisp code.

Alternatively, you can load source in any way you desire using normal commands in the listener window. This will work just fine if you are only running your application within the IDE. Assuming, however, that eventually you will want to create a stand-alone application (either with or without IDE resources included), then you will want to add one or more source files to the source table in the LAD window. Assuming that you are following along with the Squiggly example, add the source file ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Squiggly/squiggly-doc1.lisp to your table. Click in the Main column next to it and type in any non-space string and complete editing by hitting the enter key or just clicking elsewhere in the window. Then save the LAD. I used the name "sq-a" for mine and saved it in the same Squiggly directory where the source files reside. The .lapp extension is used for saved Lisp application documents and is added automatically. Your LAD window should now look like Figure 4 below:

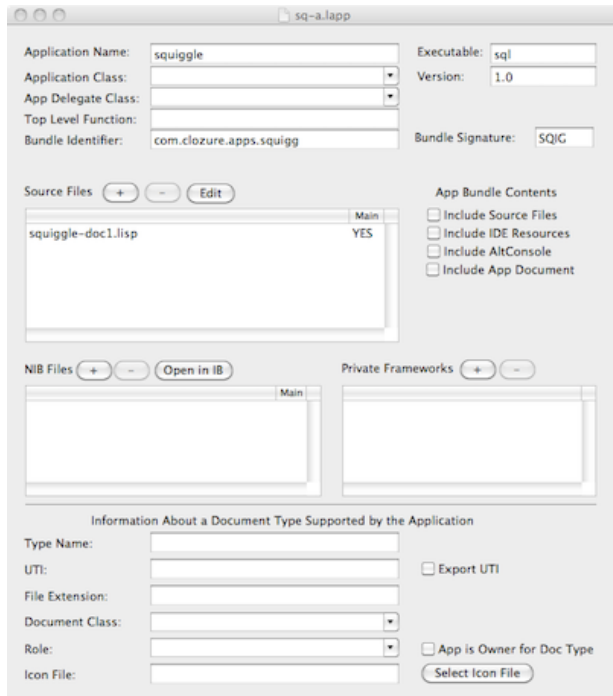


Figure 4: Saved LAD window with a single source file identified

If you now select "Load App Under IDE" from the Dev menu you will first be asked where you want to save the Application bundle. Pick any convenient directory, but I would recommend that you make that outside of any directory that might be automatically searched by CCL for source files. After that, the file `squiggle-doc1.lisp` will be "required", which will, in turn, require additional source files that will be loaded. If you like, you can verify that this occurred by typing `(find-class 'sq::squiggle-doc nil)` in the listener which should return something that looks like:

```
#<OBJECT-CLASS SQUIGGLE::SQUIGGLE-DOC (#x18933B40)>.
```

Since we have not yet specified anything about the document that our application will manage, running the application results in nothing other than loading the source code. If you were to make the same menu selection again (i.e. "Load App Under IDE") an alert would pop up telling you that the tools couldn't find anything to do. That is because source that includes Objective-C classes can only be loaded once. The Objective-C runtime does not have any facilities for redefining or removing existing classes, so unfortunately any changes to class definitions that you make and wish to be reflected in your current runtime environment require that you exit the CCL IDE and restart it before "Load App Under IDE" can do anything meaningful. The developer tools check to see whether loading source would result in the redefinition of Objective-C classes and simply don't do the reload if this would be the case. As we'll see later, it may be entirely appropriate to load the app multiple times to test modifications that you might make to your application's menu or windows or bundle and the developer tools permit you to do this without reloading the source files. You can also open up source files, make changes to Objective-C functions within those files and cause those definitions to be re-evaluated. Runtime modification of Objective-C functions IS supported by the Objective-C runtime, so a complete restart of the IDE is not necessary in order to modify them.

I will add one little caveat to the previous paragraph however. Some objects that have delegate slots that point to other objects will cache a list of what messages (methods) their delegates respond to at the time the connection is made. Adding a new delegate method sometime after that will be ignored and the objects themselves must be re-created so before they will recognize that their delegate can now respond to an additional method.

One of the side effects of loading the application's source code is that the LAD window now knows about any new document subclasses that are defined in that source code. In this particular case that is the class `sq::squiggle-doc`. If you click the pull-down menu next to the Document Class text field in the bottom section of the LAD window you will see that this class appears as a possible selection for the field.

Managing A Document With Your Application

In this section we'll talk about documents and how to add them to your application. For more background the interested developer should refer to Apple's *Document-Based Application Overview*.

To make life a bit simpler for Lisp developers I created the `lisp-document` class (`...ccl/contrib/krueger/Interface Projects/Utilities/lisp-document.lisp`). Although it is not strictly necessary that any document class you define inherit from the `lisp-document` class, the application tools do assume that any document class implements substantially all of the methods defined for the `lisp-document` class. So if you decide not to inherit from the `lisp-document` class, you should probably look at the implementation for `lisp-document` and make sure that you implement suitable counterparts. The `lisp-document` class **only** adds methods and does not add any data structures to your class. So inheriting from `lisp-document` will not increase the size of your document class instances. The rest of this tutorial will assume that any document class you define will inherit from `lisp-document`.

The features supported by the `lisp-document` class include:

- Automatic support for saving a document to disk and reloading it
- A lisp-friendly interface to the undo manager
- The ability to specify the name of the nib for the main document window
- Generation of a proxy window if you have not yet defined a window for your document
- The ability to specify a window-controller class or use a default window-controller
- Simple interfaces for printing either text or graphical depictions of your document

We will illustrate how to use these features with our squiggle example.

Creating a lisp-document subclass

You can begin to define a new document-based application by creating a class that represents your document. That class can define slots that contain any sort of data you desire. Make sure that your class inherits from the class lisp-document which is defined in the file

...ccl/contrib/krueger/InterfaceProjects/Utilities/lisp-document.lisp

You will want to add something like: (require :lisp-document) to your source file and will probably want to assure that whatever package you use to define your document class also uses the interface-utilities package. Something like "(use-package :iu)" suffices or if you define your own package you might want that definition to look something like:

```
(defpackage :my-pkg
  (:nicknames :mp)
  (:use :iu :ccl :common-lisp))
```

You can see an example of such a definition in ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Squiggle/squiggle-doc1.lisp as shown immediately below.

```
(defpackage :squiggle
  (:nicknames :sq)
  (:use :iu :ccl :common-lisp))

(in-package :sq)

(defclass squiggle-doc (lisp-document)
  ((squiggle-list :accessor squiggle-list :initform nil)
   (rotations :accessor rotations :initform 1)
   (squiggle-view :foreign-type :id :accessor squiggle-view)
   (squiggle-win :foreign-type :id :accessor squiggle-win)
   (back-color-well :foreign-type :id :accessor back-color-well)
   (back-color :accessor back-color :initform (blueColor ns:ns-color))
   (rotation-slider :foreign-type :id :accessor rotation-slider))
  (:metaclass ns:+ns-object))
```

This example conflates some of the data and controller functionality by including it in a single class instance. I did this both because it demonstrates what is possible and because that was what was done in Apple's squiggles example. The :foreign-type slots represent the controller aspects of the class. We'll see later when we create a window to display our squiggle document that these will point to view objects in the display. We could easily have included these slots in a separate window-controller object to avoid the inclusion of any :foreign-type slots in our document class.

As discussed earlier, the squiggle example is derived from an Apple sample application called squiggles. In addition to squiggle-doc1.lisp, it is implemented in squiggle.lisp and squiggle-view.lisp. A squiggle instance is an object that represents one line drawn in the window. A squiggle-view instance represents the view object that is displayed in the window. When thinking about the *model/view/controller* paradigm, a squiggle-view falls squarely into the *view* category. So when you look at squiggle-view.lisp you will see many Objective-C function calls. This is consistent with the philosophy that data objects should be primarily Lisp, view objects will have both Lisp and Objective-C elements with potentially quite a bit of the latter, and controller objects will span the two worlds.

Opening and Saving Documents

To support opening and saving documents, each Cocoa document class must normally implement the two methods #/readFromData:ofType:error: and #/dataOfType:error:, but these are implemented for you by the lisp-document class. This default implementation will likely be sufficient for most applications. We'll see in just a bit how to direct some of what it does.

A squiggle-doc will manage a collection of squiggles as well as parameters for how they are to be displayed. When you save a squiggle-doc, you will be saving all of the values contained in its slots. The good news is that all of this saving and restoring of slot values is completely automated for lisp-document instances. The only thing that a developer really needs to do is to assure that the slots archived to disk are only those that are really needed to reconstruct the document data when it is reloaded. Any slots that can be reconstructed from the others need not be part of the saved document. By default, all slots that are not :foreign-type slots are saved. This may or may not be the correct set for you. To specify which slots should be saved for your application you can implement an archive-slots method for your document:

```
(defmethod archive-slots ((obj your-class))
  (list 'slot-a 'slot-c 'slot-d))
```

Note that in the squiggle example, the squiggle-list slot will contain a list of squiggles, each of which is an instance of the squiggle class. The automatic archiving will manage all the conversion of these objects as well. As needed, you can define an archive-slots method for any other class to specify which of its slots must be saved when it is archived. So the archive-slots method is generic for all standard-instances and not just instances of lisp-document.

In the case of both our squiggle-doc and squiggle instances, the default archive-slots method will be just fine. Note that some of the values stored in squiggle slots are Objective-C objects. That is also just fine; these will be archived along with everything else and restored properly when the document is reloaded. All data type conversion needed to archive Lisp data structures and to restore them when the document is re-opened from the saved file is done automatically.

As you might imagine, not all data type conversions are completely determined by the type itself. For example, if a slot contains either a Lisp list or a Lisp array, it is saved off as an NSArray object. When that NSArray is converted back to Lisp how do the tools decide what sort of conversion is needed? In fact, the archiving process saves off not only the object itself, but also saves the original type of the object. When the saved value is reloaded and converted to a Lisp value that saved type is used to guide the conversion process. This assures that when a file is restored, it will be exactly the same as it was when saved. The conversion process also manages multiple references to the same object. Thus, if two slots contain references to the same object (i.e the slot values are eq) that will also be true upon restoration of the objects.

Although the type conversion between Lisp and Objective-C is now quite comprehensive, it is certainly possible that some conversion may not be done properly. Please let me know about any problems encountered in this area and I'll try to make an appropriate fix.

Supporting Undo

It's probably natural to think about "undo" functionality as something that lets you erase some action that you just took via a user interface. But in fact "undo" provides a way to revert the state of some object (a document in our case) back to a previously existing state, erasing all changes made between those two states. So it is not really interface functionality at all, but rather functionality that is associated with a document object. You just use an interface to see what state changes are possible and select them. Apple's support for undo starts with an undo-manager object that is associated with an object (typically an instance of `NSDocument`). To implement undo functionality, an application action that changes the state of the object should register a counter-acting action with the undo-manager. That counter-action should restore the previous state of the document if it is invoked. If the user subsequently selects the undo menu-item, then that action is called to restore the previous state. In a normal Cocoa program, all such actions are given in the form of Objective-C messages and message parameters that will be sent. That is a bit cumbersome for Lisp developers so I created three Lisp mechanisms that translate from more normal Lisp idioms into appropriate Objective-C calls.

The first mechanism for providing undo functionality is to use the `set-undo` macro. This takes the form:

```
(set-undo <target> <undo-closure> <undo-string>) where
<target> is the lisp-document instance being modified.
<undo-closure> is a function of 0 arguments that when funcalled will undo the change being made, and
<undo-string> is a string that names the action that will be undone or redone.
```

Typically the undo-string should name the action. So in a method that is adding something to a view, the undo-string should be something like "add". When in the process of "undoing" the set-undo method will ignore this string and the underlying Objective-C method will take the action name for the redo menu item from the undo menu item. This macro will work with any target that is a subclass of `NSDocument`.

For more complex state changes, this mechanism provides a way to make arbitrary changes and is a good choice. In many instances however, the change being made only affects a single slot value for a document and the undo action consists of simply setting the slot back to the value it had previously. To support such easy changes I added two additional Lisp mechanisms to support undo for modifications to individual slots.

The second mechanism supporting undo is the `update-with-undo` function. This takes the form:

```
(update-with-undo <accessor-name> <instance> <new-value> &key :undo-name <undo-string> :test <equality-test>) where
<accessor-name> is the name of some accessor for an object instance. This accessor must define both read and write methods (i.e. both (symbol-function <accessor-name>) and (symbol-function (list 'self <accessor-name>)) must return valid functions.
<instance> must be the object to which the accessor is applicable.
<new-value> is an arbitrary lisp value
<undo-string> is a string that names the action that will be undone or redone
<equality-test> must be a function that takes two values of the sort that will be set in the slot and returns t if they are the same and nil otherwise.
```

The `update-with-undo` function is a convenience function that updates an instance slot only if the value is different from whatever was there previously (using the equality test specified or `#'equal` if none is provided). It makes an appropriate `set-undo` call if the new value differs from the existing slot value. The instance should be a subclass of `NSDocument`. This call would typically be made from some lisp function that wants to update a slot and have that change reflected as an "undoable" action in the document's window if and only if an actual change to the slot's value occurred.

Some slots may be designed to only be updated via bindings made to them from user interface view objects. We'll see how that works a bit later, but for now just understand that it is possible for such changes to occur without requiring the developer to provide any lisp code other than the use of an additional slot option (`:kvo`) that provides a name for the binding-target. To add an automatic undo capability for such changes, any slot defined in an `NSDocument` sub-class can add a slot option of the form:

```
:undo <undo-string> where
<undo-string> is defined as previously.
```

This will automatically register an appropriate undo action with the document's undo-manager anytime the slot is changed via a key-value binding. For example, if a user changes the value in some field in the document's window that is bound to a slot with this option, then an automatic undo method will be registered that would set the value back to what it was previously.

It is possible to use any or all of these three undo mechanisms for a single document.

The Squiggle example provides some simple examples of how the `set-undo` function can be used. A squiggle is a structure that is effectively an arbitrary curve drawn by a user in the squiggle window by clicking and dragging the mouse. Its structure will be initialized when the user first clicks in a window and extended as the mouse is moved while the button is held down. The structure is complete when the mouse button is released. For purposes of this application we will treat creation of the entire squiggle as an undo-able event. In addition we will allow the user to set the number of rotations and select a background color. Therefore there are four methods that we want to be undo-able: `add-squiggle`, `remove-squiggle`, `set-rotations`, and `set-back-color`. These are shown below:

```
(defmethod remove-squiggle ((self squiggle-doc) (sq squiggle))
  (set-undo self
    #'(lambda ()
        (add-squiggle self sq))
    "remove squiggle")
  (setf (squiggle-list self) (delete sq (squiggle-list self)))
  (#/setNeedsDisplay: (squiggle-view self) #YES))

(defmethod add-squiggle ((self squiggle-doc) (sq squiggle))
  (set-undo self
    #'(lambda ()
        (remove-squiggle self sq))
    "add squiggle")
  (setf (squiggle-list self) (cons sq (squiggle-list self)))
  (#/setNeedsDisplay: (squiggle-view self) #YES))
```

The `set-undo` calls in each of the methods above simply calls the other; to undo an add you remove the squiggle and vice versa. The `set-undo` call includes a function of no arguments that would reverse the effects of the action currently being taken. Selecting "undo" in the squiggle window (once we get around to creating one) results in the invocation of that function.

```

(defmethod set-rotations ((self squiggle-doc) rot)
  (let ((old-rot (rotations self)))
    (set-undo self
      #'(lambda ()
          (#/setIntValue: (rotation-slider self) old-rot)
          (set-rotations self old-rot))
        "set rotations")
      (setf (rotations self) rot)
      (#/setNeedsDisplay: (squiggle-view self) #$YES)))

(defmethod set-back-color ((self squiggle-doc) col)
  (let ((old-col (back-color self)))
    (set-undo self
      #'(lambda ()
          (#/setColor: (back-color-well self) old-col)
          (set-back-color self old-col))
        "set background color")
      (setf (back-color self) col)
      (#/setNeedsDisplay: (squiggle-view self) #$YES)))

```

Each of the two functions above provides a closure that reverses its actions. Don't worry too much at this point about the actual actions taken. These mostly make sure that the display is kept in synch with the state of the squiggle document.

File Types and Extensions

A document's *file type* is a string that identifies what sort of document it is. Documents can be encoded and stored on disk. The same document can sometimes be stored using different encoding formats. The *file extension* identifies the format of the disk file. Every document-based application must specify the types of document that it supports. For each such type it must identify the extensions that it supports for that type and the document class that represents it. A UTI is a *Universal Type Identifier*. This is Apple's currently recommended way to identify a document. It must be a dot-separated list of names that intentionally looks a lot like a reversed URL. Although in this example I have used a string here that incorporates Clozure, you will probably want to use a string that reflects your own company or personal identity. If you export the UTI, then that definition is made public on the system and is available for import into other applications. When an application specifies that it is the Owner of the document type, then double-clicking on such a document in the Finder will generally result in opening up that document in that application. If no application claims to own a particular document type then the system has a set of rules for finding applications that may be able to open and edit it and will open one of those.

When a file is opened in an application, only files with the supported extensions for the file types supported by the application can be selected in the open dialog. The extension of the file opened is mapped to the class which implements the corresponding type of document and an instance of that class is created and told to initialize itself using data from the file. When a document is saved, its file type determines the allowable extension(s) for the file on disk. In a typical application this information is stored (along with lots of other information) in the Info.plist file that resides in the application's bundle. The developer tools will automatically create and initialize this file for you given information that you enter in the LAD window.

In addition to these normal mechanisms, the developer tools also use the file type and file extensions that you entered in the LAD window for similar purposes when running the application under the IDE.

Initializing the Application Bundle

As discussed earlier, all applications are contained in application bundles. The tools automatically create such a bundle for you whenever you select "Initialize Bundle" from the Dev menu or when you run your application (either under the IDE or stand-alone). The basic directory structure is set up and then populated according to settings in the LAP window. At this point don't worry too much about what a bundle looks like. We'll discuss that more as we add functionality to the application that causes the tools to make additions to the bundle.

Loading your Document Application in the CCL IDE

Next we'll add some information about the Squiggle document into the LAD window. First select "Load App Under IDE" from the Dev menu if you have not previously done so. Since we have not yet created a bundle for this application, a pop-up window will ask you where you want to put the bundle. You can select any convenient directory, but if you have any idea that you might later want to add source code to it, then I'd recommend selecting a directory that is outside of any area that CCL might search for source files. That's because both the original source files and those that have been copied into the bundle would be found if you try to require a file with that name. When two paths are found, require treats that as if no files were found and will tell you that no path exists to the file.

The only other thing that the tools can do at this point is to load the specified source files, so that's what happens. We have yet to tell the tools about anything else related to the squiggle application. One side-effect of loading source files is that information about the squiggle-doc class that is defined in that squiggle-doc1.lisp is now known to CCL.

Next fill out the remaining fields for the squiggle document type at the bottom of the LAD window as shown in Figure 5 below. You will see that when you want to fill in the Document Class: field you can just use the pull-down-menu attached to that field and select the SQUIGGLE::SQUIGGLE-DOC class. That is now available as a choice because we previously loaded the source file.

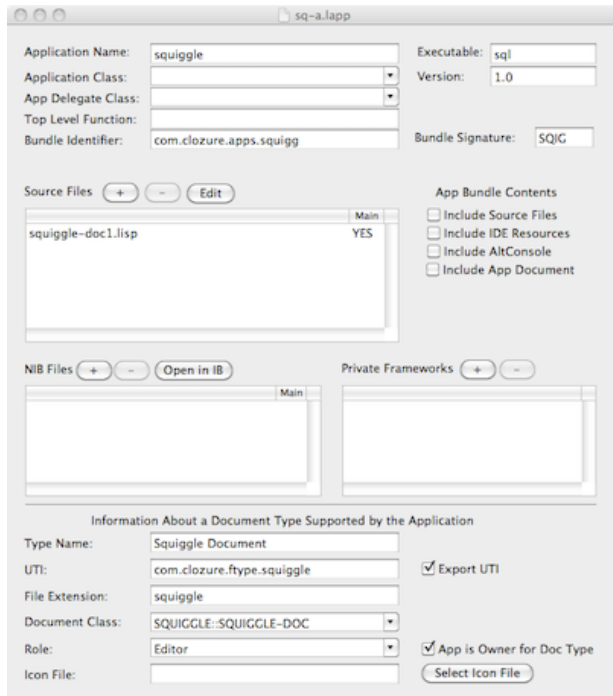


Figure 5: LAD window with Squiggle Document information

Now once again select the "Load App Under IDE" menu item in the Dev menu. The source will not be reloaded because the tools know that this was already done once and that trying to do it again would result in trying to redefine an Objective-C class, which the runtime does not permit. We have yet to tell the tools anything about what sort of window should be used to display a squiggle document or what sort of menu items we might want, but we have told it enough about your document to let you create one, save it, reload it, etc. So after you have loaded the application you will see that the tools have provided three new menu items in the File menu: "New Squiggle Document", "Open Squiggle Document" and "Print Squiggle Document" to let you begin to debug your application. Your File menu should look much like Figure 6 below:

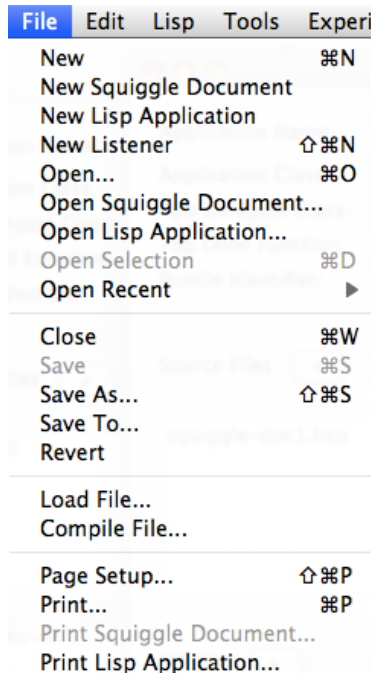


Figure 6: File menu with additions for Squiggle Documents

Note that the item for Printing Squiggle Documents is grayed out since we only want that to be enabled when the top document is a squiggle document that can respond to the print command.

So next let's create a squiggle document using the "New Squiggle Document" menu item to see what happens. In a normal application you would expect this to open up a blank window for the document (whatever that means for that type of document). Of course we have yet to define any sort of

window for displaying a squiggle document, so by default the `lisp-document` class methods will create a proxy window for us. This is just a placeholder for a real document window, but it does serve a useful purpose. If you click on it to make it the first responder, you will see that the "Print Squiggle Document" menu item is now enabled. In addition you can now save your document, close it, and reload it. We haven't yet created any way to actually edit that document in a normal window, but that doesn't mean you can't interact with it. You still have the listener window at your disposal and can use that to view or modify slots in your document object. Of course we need get a pointer to the document instance somehow. There are any number of ways that you might be able to find the document, but the `docs-of-type` function was created for just this purpose. Do the following in a listener window:

```
? (iu::docs-of-type "Squiggle Document")
(#<SQUIGGLE-DOC <SquiggleDoc: 0x194b2e40> (#x194B2E40)>)
```

This returns a list of all open documents of the type specified (which must designate some sort of `lisp-document`). You can manipulate that result in the usual ways. The type string argument should be the same string that was used to define the document type in the LAD window (which was used to create the menu items). If you have more than one such window open then obviously you may need to inspect them to determine which is which. You can then manipulate the list returned in any way you want. Here's a simple example:

```
? (describe (first *))
#<SQUIGGLE-DOC <SquiggleDoc: 0x194b2e40> (#x194B2E40)>
Class: #<OBJC:OBJC-CLASS SQUIGGLE::SQUIGGLE-DOC (#x1706CB00)>
Wrapper: #<CCL::CLASS-WRAPPER SQUIGGLE::SQUIGGLE-DOC #x302001035A4D>
Instance slots
SQUIGGLE::SQUIGGLE-LIST: NIL
SQUIGGLE::ROTATIONS: 1
SQUIGGLE::BACK-COLOR: #<NS-COLOR NSCalibratedRGBColorSpace 0 0 1 1 (#x1A4230)>
NS:ISA: #<OBJC:OBJC-CLASS SQUIGGLE::SQUIGGLE-DOC (#x1706CB00)>
SQUIGGLE::SQUIGGLE-VIEW: #<A Null Foreign Pointer>
SQUIGGLE::SQUIGGLE-WIN: #<A Null Foreign Pointer>
SQUIGGLE::BACK-COLOR-WELL: #<A Null Foreign Pointer>
SQUIGGLE::ROTATION-SLIDER: #<A Null Foreign Pointer>
?
```

The "print Squiggle Document" menu item will also be active when you click in a proxy window for a squiggle document. The `lisp-document` class provides a default implementation that prints out the values of each slot in your document in text form. If you print a LAD by selecting the "print Lisp Application" menu item, that default behavior is used and you'll get a printout of the values in the slots of the `lisp-app-doc` instance that supports that window. We'll see a little later how to modify the default printing behavior to present something more meaningful and/or with a better format.

If you would like to test the save and restore functionality, then at this point you could change any of the slot values for non-foreign slots (because for this object those are the slots that are saved), select "save as" to save it somewhere, close it, and then reopen and examine the reopened document (a different object than the one previously open of course) and verify that your change was saved and restored correctly.

OK, so now we've got our own little mini-application running inside the IDE with a document defined that can be saved and restored. This required almost no effort on our part. Perhaps it's time that we add a better way to edit the document; namely a window.

Adding A Window To Your Application

At this point we want to define a window in which to view and modify a squiggle document. For our purposes a graphical window is best. We will use Apple's interface Builder application (IB) to do this. Readers who have worked through the `InterfaceBuilderWith CCLTutorial3.0` will already be quite familiar with this process although some new facilities have recently been added that are not reflected in that document so you might want to pay attention.

As you build a window for your application there are two different ways of working and each has its advantages. The first way is to define all the Lisp classes you may want to reference from IB and import definitions into IB (after some automated translation that we will discuss shortly). If you have used IB previously, you are aware that you will need establish connections between objects using their *outlets* and define how interface element trigger *actions* for objects. These correspond roughly to foreign-slots and methods in Lisp objects respectively.

The second way to work is to start in IB. You can define object classes there with outlets and actions as needed and then import those class definitions into Lisp (after some automated translation that we will discuss shortly). You can also work in both directions if desired, importing into IB, making changes, exporting back to Lisp, etc.

There are two sorts of translation functions that are available in the Dev menu: "Create .h for Interface builder" and "Create Lisp Source from .h ...". There is a sub-menu for the first of these that lists Objective-C classes that are known to Lisp (excluding standard classes which would make that menu VERY long). If you select any one of these, a corresponding .h file will be created and named and saved by using a standard save-file panel (you do not need to specify the .h extension). If you like, you can open and examine this .h file using any editor that can handle text files (including CCL of course). So, for example, you can create a .h file that corresponds to our `squiggle-doc` class by selecting it from the sub-menu. Go ahead and do that and save the result someplace where you can find it again. If you open this in some text editor you will see something like the following:

```
@interface SquiggleDoc : LispDocument {
    IBOutlet id squiggleView;
    IBOutlet id squiggleWin;
    IBOutlet id backColorWell;
    IBOutlet id rotationSlider;
}
@end
```

There are no actions defined here because we have yet to create any method definitions for this class. Note that the superclass is `LispDocument`, which is the form of the name that corresponds to the `lisp-document` class. In just a bit we'll explain how to import this .h file into IB.

As we start to use IB, you can either create your own NIB by following along with the explanation below or just open up the one I have already defined in `...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Squiggle/SquiggleWin.nib`.

Window Definition in Interface Builder (IB)

Start IB and create a new document using the MacOSX window template. From the File menu select "Read Class Files" and navigate to the .h file that you just created and load it. If you now look at the Library window and select the "classes" pane, then you can scroll down to the SquiggleDoc class and select it. If you look at the outlets defined for it you will see something that looks like Figure 7 below.

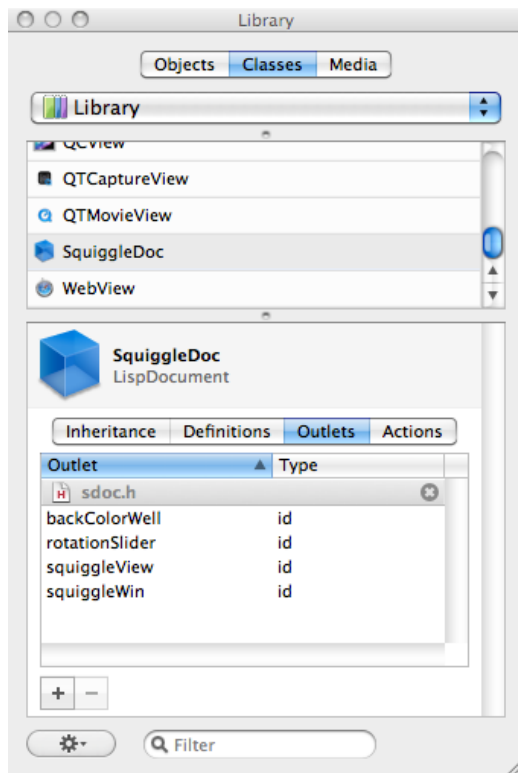


Figure 7: Outlets for SquiggleDoc class as seen in Interface Builder

If you next examine the inheritance panel for the SquiggleDoc class you will see that IB understands that SquiggleDoc inherits from LispDocument, but has no idea what a LispDocument is. Although it's not really necessary for our purposes, we could rectify this in any of several ways. We could have created a .h file that corresponds to the LispDocument class and loaded it into IB when we loaded the SquiggleDoc class. Or we can just directly tell IB what class LispDocument inherits from. Let's do that by clicking on the "Click to set superclass" button in the inheritance pane and selecting NSDocument as the superclass. The inheritance pane for the SquiggleDoc class will now look like Figure 8.

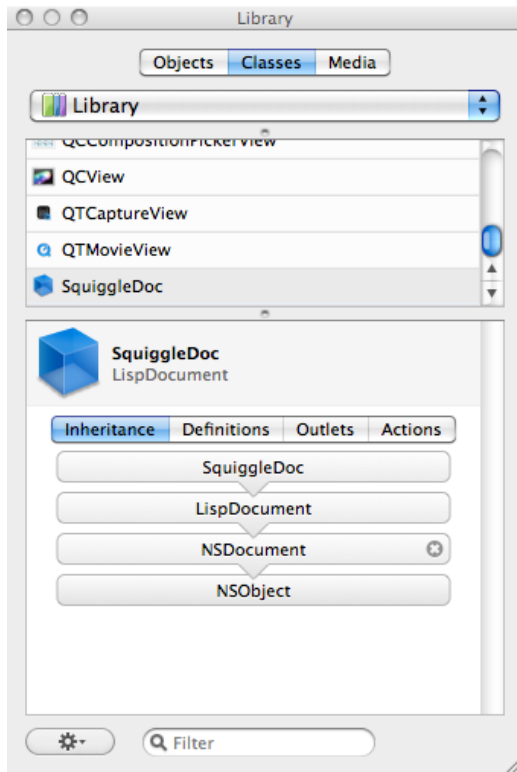


Figure 8: Completed inheritance for the SquiggleDoc class

Before proceeding further, let's save this NIB file. I named mine SquiggleWin and saved it in the same directory where we have the source files, but you can name yours however you want and put it wherever you want; just don't forget what you called it and where you put it, because we will need that information later. When you save the file you will want to change the form of the saved file from the default .XIB to .NIB which is directly usable by an application at runtime. For Xcode developers that translation is only done at build time, but since we won't have a formal build process and often want to see the results of changes to our user interface almost immediately, using the NIB format is just better all around.

The NIB file we are creating will be the one that is used to display a SquiggleDoc. You will recall that we decided to control that window directly with methods defined for the squiggle-doc class rather than using a separate window-controller class of some sort. That is, we combined the data and controller aspects into that single class. That means that when this NIB file is loaded into our application (typically in response to the user selecting some form of "New ..." or "Open ..." menu item), that the owner of the objects created will be a squiggle-doc instance. We tell IB this by clicking on the File's Owner object in the IB document window. Then in the IB inspector window click on the "object identity" icon to open up that pane. If you don't see an inspector window, double-click on the File's Owner object and one will be shown. The "object identity" icon is one farthest to the right in the inspector window and is a little circle with an "i" in it. Now you can set the class for the File's Owner object by selecting SquiggleDoc from the pull down menu attached to the "Class" text field as shown in Figure 9 below:

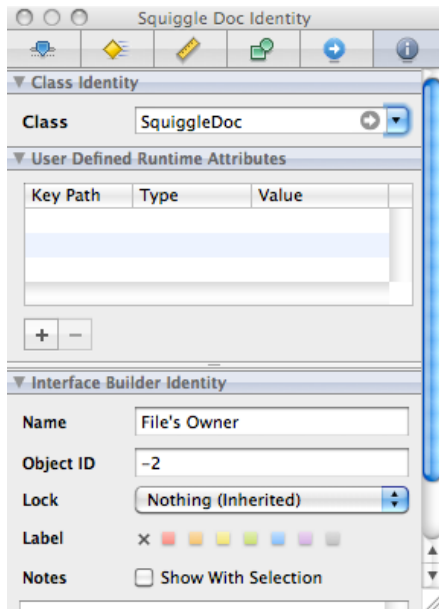


Figure 9: Setting the Class of the File's Owner

OK, we're all set to start defining the actual contents of the window. We're going to create a fairly simple window with four elements in it. The main one will be a custom view object in which we will graphically display our squiggles. The next will be a slider control that determines how many times we will rotate and re-display each individual squiggle line. The next control will be a color well that determines the base color for the background and finally we add a label that says what the color well is for.

To make it a bit easier to see where we're going I'll first show you what the Squiggle display window will look like when we get done. Figure 10 shows the completed window (at 60% of actual size as I have done with all non-panel windows shown in this document).

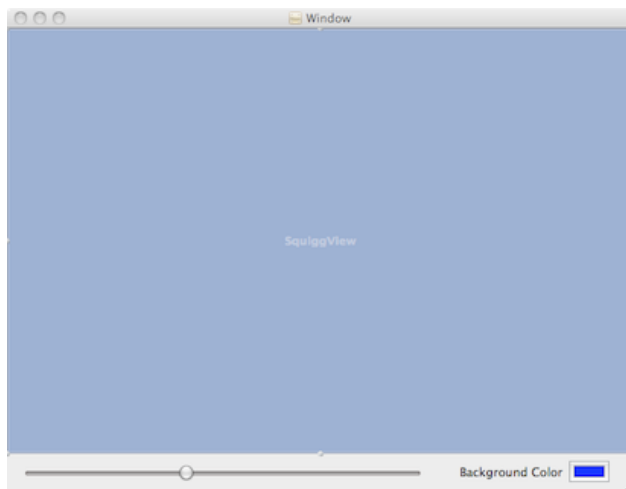


Figure 10: Squiggle Display Window

Let's start with the custom view object. Go to the Library window again. Click on the Classes button and then select the `NSView` class. We are going to define a new custom subclass of `NSView`. To do that, click on the additional actions button in the lower left corner of the Library window and select "New Subclass...". In the dialog window that pops up, set the name to "SquiggleView" but leave the "Generate Source Files" box unchecked for now. We'll actually create those after we have made one more addition to the `SquiggleView` class. Click on the outlets button for the `SquiggleView` class, then add a new outlet by clicking the "+" button and name it "document". An outlet is just a slot in a class that nominally points to some other object. In our case we'll want the `SquiggleView` to be able to extract various parameter values from the document so that it can display everything as intended so we will add an outlet that will contain a pointer to the squiggle-doc instance that contains it. Once you have added this outlet, the Library window should look much like Figure 11 below:

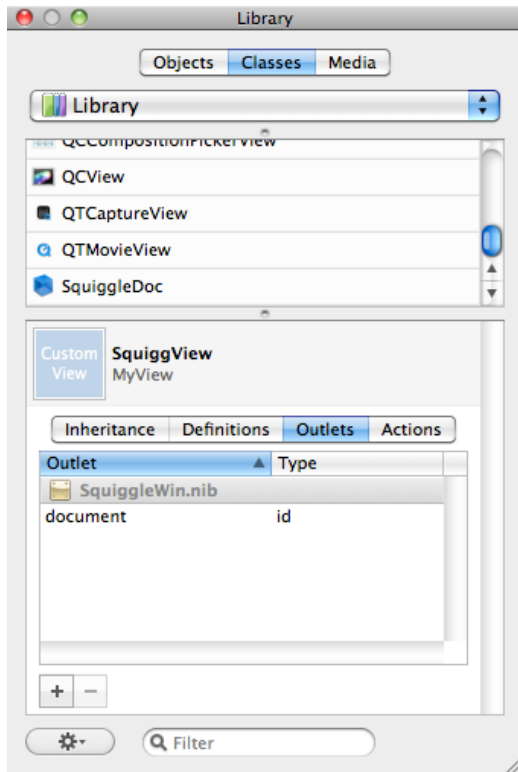


Figure 11: SquiggDoc outlets

Now click the additional actions button in the lower left and select the "Generate class files" item. You will be asked where to save the files. This will actually save both a .m and a .h source file. We don't really care about the .m, we'll only want the .h, so we can just ignore the .m. Make sure that the box that specifies also saving a .h is checked in the save dialog box. We now have the start of our own custom view class. From the Library window drag a SquiggView to the window that is being designed. Resize the window to something you like and place the SquiggView in some fashion similar to what is shown in Figure 10 above.

Drag a Horizontal Slider object to the window and position and size it as desired. You will also want to specify the minimum and maximum values that the slider can take. You do that by clicking on the slider and then selecting the attributes pane in the inspector window (small icon farthest to the left in the inspector window) and editing the min and max fields. For our purposes a minimum of 0 and maximum of 100 seem to work pretty well. The current value is irrelevant because we'll set that later from our Lisp code. Add a label object by dragging one from the Library window and title it "Background Color". Finally drag a Color Well object to your window.

You will also want to make sure that the behavior of all of these objects is reasonable if your window is re-sized. You do that using the "size" pane in the Inspector window (little ruler icon). I'm not going to go into any detail here about how to do that, so if you aren't sure refer to either the InterfaceBuilderWithCCLTutorial3.0 tutorial or to Apple's Interface Builder documentation. The size browser display in IB actually does a pretty good job of dynamically showing you what will happen and if you're still not sure, then choose "Simulate Interface" from the file menu and you can resize the displayed menu to see what happens.

If you are new to IB and have difficulty with any of these actions, I suggest that you spend some time going through the InterfaceBuilderWithCCLTutorial3.0 which provides many more examples of how to work in IB.

The next step is to link up these display objects to the SquiggDoc. At runtime when the NIB is loaded and these objects are created, the links that we specify here will result in pointers being placed in appropriate slots to form corresponding links between the real objects. You can make single connections between objects by control-clicking on one and dragging to another. The pop-up that appears when you release the mouse button allows you to specify which outlet to use for the link in the starting object. To make multiple connections from a single object it is a bit easier to just control-click on that object and then drag as needed from the pop-up that is displayed. We'll do that now for connections from the File's Owner object. First Control-click on the File's Owner object. The window that pops up should look like Figure 12 below:



Figure 12: File's Owner before making connections

Click and drag from the small circle at the right end of the backColorWell outlet line to the color well object that was added to the window. Similarly link the rotationSlider outlet to the horizontal slider we added to the window, the squiggleView outlet to the SquiggView object in the window, and both the squiggleWin and window outlets to the window itself. When you complete that, the window should look like Figure 13 below:



Figure 13: File's Owner after making connections

It's reasonable to ask why we created two separate links to the window object. This was mostly laziness on my part. The window slot is inherited from the NSDocument class. In order to access it we would use the Objective-C `#window` method. We could easily define a Lisp method to make this call for us if we wanted to avoid having to use Objective-C syntax. Instead I just added a normal Lisp slot called SquiggleWin and defined a Lisp accessor for it.

Now let's add some additional individual connections that are needed. Link the "document" outlet in the SquiggView object by control-clicking on the SquiggView, dragging to the File's Owner object, and then selecting "document" from the pop-up. You'll recall that we added this slot to the SquiggView class so that the view could retrieve parameters from the document object. Similarly link the "delegate" outlet from the window to the File's Owner object. Window delegates are given the opportunity to handle certain sorts of messages that are sent to windows. In our particular case we are not going to implement any of the defined delegate methods for NSWindow objects, so we could get along without this link, but it is a very good idea to get into the habit of making it anyway. That way if you later decide that you need to add a delegate method, then the link will already exist. Now if you control-click on the File's Owner object the pop-up should look like Figure 14 below:



Figure 14: File's owner with all links completed

The last step in the interface design is to specify what happens when the controls are modified. When a user changes the horizontal slider or picks a new color from the color well we want to be told about it. We obviously want to do something when a user clicks in the SquiggView as well, but in that case the SquiggView object itself can decide what to do. We'll see a bit later how we specify that when we define the SquiggView class in Lisp. To specify what a control tell the File's Owner object when it changes, we need to define an appropriate message that the squiggle-doc object will accept. Such messages are called "Actions" in IB.

Let's go back to the Library Window and select the SquiggleDoc class. Click on the Actions button to show its actions. Then add two actions (using the "+" button) called "setBackColor:" and "setRotationCountFromSlider:". Note the ":" at the end of each name. When actions are sent from a control they pass a pointer to themselves as the only argument. When you have added these definitions, the Action pane for the SquiggleDoc class should look like Figure 15 below:

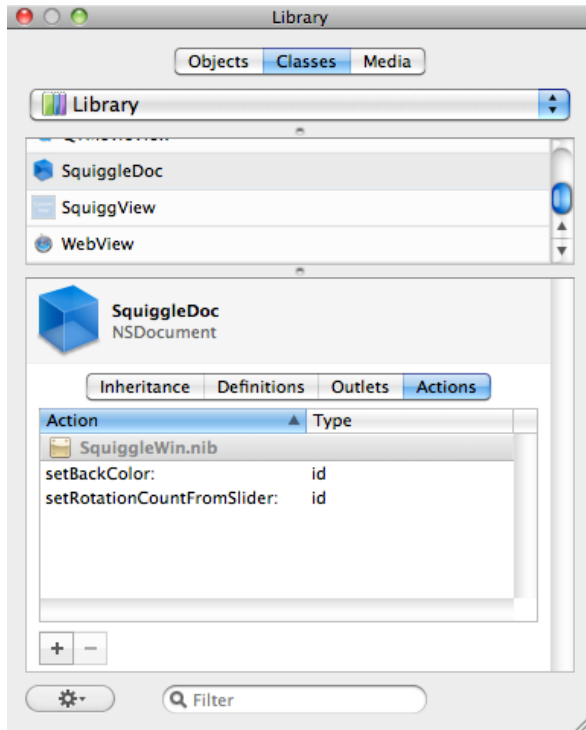


Figure 15: SquiggleDoc Actions

We have effectively changed the definition of the SquiggleDoc class to add two new methods. Later, in the Lisp definition for the corresponding squiggle-doc class we'll define an implementation for each of these methods. That's pretty simple in this case since we are only adding two methods. If there were a large number of controls we might want to make sure that we don't forget any of them and also make sure that everything is spelled the same way in IB and in Lisp. One way to assure that is to choose the "Write updated class files" item from the additional actions menu and let IB generate a new .h. The resulting file will look a lot like the old one, but additionally reflects the new methods that we added. Go ahead and do that now. Later in Lisp we'll see how to import that back in a form that is suitable for use there.

Now tell the controls to use these actions by control clicking on the slider and color well objects in the window and dragging to the File's Owner object. When the mouse button is released, select the appropriate action from the pop-up that appears. If all has gone well, you can control-click on the File's Owner object and the pop-up should now reflect the actions that you specified in the "Received Actions" section; as shown in Figure 16 below:



Figure 16: File's Owner with all action connections made

This completes the definition of the Squiggle user interface in IB. Make sure that you save the NIB file and go back to the CCL IDE to continue the definition of the application.

Window Definition in the CCL IDE

There are a number of steps that we'll now take to add a window to our Squiggle application. First we'll import the definitions for the SquiggView and SquiggleDoc classes that we created or modified in IB and use them to create corresponding Lisp source code. Then we'll identify the NIB file we created so that the tools know where it is and what it is named. Finally we'll test to see whether we can run the application under the IDE and have it use our new window definition correctly.

In the Dev menu select "Create Lisp Source From .h ..." and direct it to the .h file that you generated in IB for the SquiggView. This will open up an untitled Lisp source window under the IDE that should look like Figure 17 below:

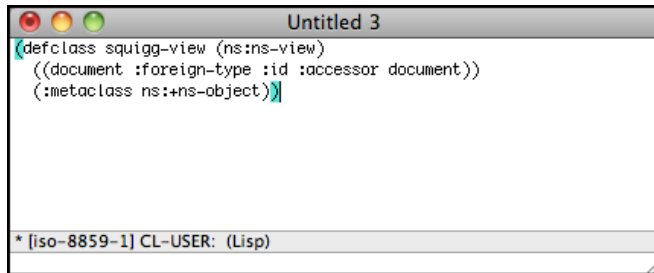


Figure 17: Source for the squigg-view class

Similarly generate source for the squiggle-doc class from the .h saved in IB for the SquiggleDoc class. This will look like Figure 18 below:

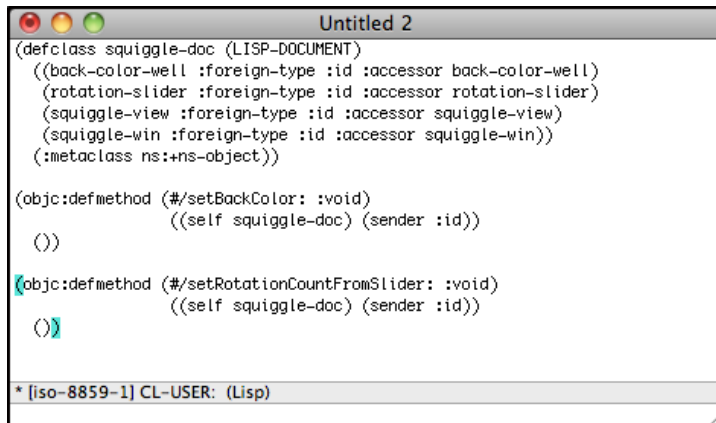


Figure 18: Source for the squiggle-doc class

In the case of the squigg-view class, we didn't get a lot of help, but it is at least a start on what needs to be defined. We'll add significantly to it in a moment. Since we already have a source file for the squiggle-doc class we will simply do a copy and paste from the document that we just opened to the squiggle-doc1.lisp file that we already created. Copy just the new methods that we defined.

If you want to follow along with the files that I've already created, then I suggest that at this point you change the source file from squiggle-doc1.lisp to squiggle-doc.lisp. That is the completed form of the document. You can change this in the LAD window by deleting the squiggle-doc1.lisp line and add the squiggle-doc.lisp source. Be sure to click in the Main column and make the main file. When we get to the point of wanting to load this file you will have to restart the IDE to do so because as noted previously it isn't possible to redefine Objective-C classes at runtime. Feel free to save your LAD document, restart the IDE, and re-open the LAD document now. Assuming you have done that, let's move on.

First let's take a brief look at the Lisp source that will now comprise the Squiggle application: squiggle-doc.lisp, squiggle.lisp, and squigg-view.lisp.

Let's start with squiggle-doc.lisp. Since we have previously discussed much of what is in this file, we'll just describe what needs to be added here.

First, immediately after the squiggle-doc class definition we add the following:

```

(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :squigg-view))

```

This adds a require statement for the squigg-view.lisp file that we will create to contain everything needed to draw squigg-views. Next we add methods to support our new window.

```

(defmethod window-nib-names ((self squiggle-doc))
  (list "SquiggleWin"))

```

This method is called from the lisp-document's #/makeWindowControllers method. That Objective-C method is called as part of the process of opening a document. You can override this method for your document class if you know what you are doing. Otherwise, the lisp-document default method will do what needs to be done in most cases. Feel free to look at how that is defined in lisp-document.lisp. One of the things that it will do is load one or more NIB files that correspond to the window. You use it to specify the names of the NIB files that are to be loaded to create the new windows needed for this document. Whether your document requires only a single window or more than one window, this method should return a list. Each element of the list of names returned by window-nib-names can be a simple name as shown here, in which case it is assumed to be the name of a NIB file that is located within the application's bundle. Or it can be a completely determined absolute pathname, in which case the NIB can be located anywhere on the system. For an application that you want to distribute to others you'll obviously want to put the nib files into the bundle so that they are available for loading.

If the window-nib-names method returns nil, as the default lisp-document method does, then the Objective-C document method #/windowNibName will be called. If that method returns a non-null NSString, then that name will be used as the name of a single NIB to be used. This is basically the same mechanism that Objective-C developers use to specify a NIB file for their documents.

There is another lisp-document method that I will mention here, even though we will not need to override it for the squiggle-doc class. That method is called document-window-controller-classes. This method returns a list of document-controller classes that should be used to control each NIB named

in the list returned by the window-nib-names method. By default a plain old NSDocumentController is used if a single NIB is named and not special document controller class is needed. If you have multiple NIB files or need a special document controller class for any NIB file, then you must override the document-window-controller-classes method as well.

If you need to do anything special as part of window creation, then you should override the Objective-C `makeWindowControllers` method for your document class. There are a few things that this method needs to do, so I suggest that if you find you need to do this, that you read Apple's documentation on the subject and/or use the version defined for the lisp-document as a template.

```
(defmethod document-is-nib-file-owner ((self squiggle-doc) nib-name)
  (declare (ignore nib-name))
  t)
```

The method `document-is-nib-file-owner` overrides a corresponding method defined for the `lisp-document` class that returns a default value of `nil`. This is also called from the `lisp-controller's` `makeWindowControllers` method. When a NIB file is loaded, the call must specify what object will be the File's Owner object for the named NIB name specified. Most of the time that will be an `NSWindowController` instance of some sort, but in our case we have decided to incorporate controller functionality directly into the document class and make it the File's Owner object. Overriding the `document-is-nib-file-owner` method to return `t` tells the method to make the document the File's Owner instead of specifying a window controller as the File's Owner. A generic `NSWindowController` is still created to perform most of its duties, but the links to the File's Owner object that are made at the time the NIB is loaded are made to our `Squiggle` document instance rather than to that window controller.

```
(objc:defmethod (#/awakeFromNib :void)
  ((self squiggle-doc))
  (#/setIntValue: (rotation-slider self) (rotations self))
  (#/setColor: (back-color-well self) (back-color self)))
```

This method is used to fulfill some of the responsibilities normally associated with a window controller, namely to initialize values for display objects that need such initialization. This method is called for all objects created when a NIB is loaded as well as for the File's Owner object. It is only called after all links specified in the NIB file have been made between the new objects.

Next we will complete the definitions for those action methods that we created in IB that are sent by the controls. Those two methods were called `setRotationCountFromSlider:` and `setBackColor:`. Recall that we previously defined Lisp methods called `set-rotations` and `set-back-color` to modify our Lisp document object. So all we need to do now is to link up the Objective-C methods that will be called by our controls with those Lisp methods.

```
(objc:defmethod (#/setRotationCountFromSlider: :void)
  ((self squiggle-doc) (sender :id))
  (set-rotations self (#/intValue sender)))

(objc:defmethod (#/setBackColor: :void)
  ((self squiggle-doc) (sender :id))
  (set-back-color self (#/color sender)))
```

Note that we could easily have directly incorporated the substance of the Lisp methods directly into these Objective-C methods and eliminated the Lisp methods altogether. I deliberately did not do that to emphasize the distinction between *model* functionality that we want to maintain as straight Lisp code and *controller* functionality that is used to link views and models together. These new Objective-C methods are clearly controller functions which are triggered by view objects and then call one or more model methods to reflect the view changes in the model.

Next we will look at a couple of additional methods to support the creation of squiggle objects:

```
(defmethod start-new-squiggle ((self squiggle-doc) point)
  ;; this will be invoked on mouse-down to start a new squiggle.
  (let ((sq (make-instance 'squiggle)))
    (add-point sq point)
    (add-squiggle self sq)))

(defmethod continue-squiggle ((self squiggle-doc) point)
  ;; when the mouse is dragged, continue by adding a point and
  ;; invalidate the view so it redraws
  (add-point (first (squiggle-list self)) point)
  (#/setNeedsDisplay: (squiggle-view self) #YES))
```

These methods will be called respectively when a mouse-down or mouse-dragged event occurs within the `squigg-view` object that we created in IB. We'll see where those calls are made later when we discuss `squigg-view` lisp functionality.

Before completing our discussion of the `squiggle-doc.lisp` source we will add methods to support printing. A short digression about the printing facilities that are made possible by the `lisp-document` class is in order.

Printing can be a fairly involved operation. The nitty gritty details are discussed in the **Project 7: Loan Document** section of the "InterfaceBuilderWithCCLTutorial3.0" document. What I have done for the `lisp-document` class is to create a `lisp-app-doc-print-view` class that can easily be used to print documents using either text or graphics. If you wish to define your own print view class, it is only necessary to override the definition of the `print-view-class` method in your `lisp-document` subclass to return the `NSView` sub-class you wish to use for printing your document. That class should take a ".doc" initialization argument that will contain the document instance. You could use the implementation of the default class (defined in `lisp-app-pr-view.lisp`) as a template for your own if desired. Or you could drop even lower down and override the Objective-C method `printOperationWithSettings:error:` for your document class and then do whatever you want in whatever way you want within that method. In the printing discussion below I will assume that you have decided to use the default methods provided by the `lisp-document` and `lisp-app-doc-print-view` classes.

The `lisp-app-doc-print-view` class calls several methods that must be defined for the document class used to initialize it. There are default methods defined for the `lisp-document` class which can be overridden in subclasses to customize how documents are printed. Documents that wish to print themselves in a customized way as text objects should override the method `print-lines` and documents that wish to print themselves in a graphic way should override both the `print-lines` and `print-graphic` methods. The `print-graphic` method is only called if the `print-lines` method returns `nil`, so to print a `lisp-document` subclass as a graphic, it should override the `print-lines` method to assure that it returns `nil`.

The `print-lines` method is declared as: `(defmethod print-lines ((self lsp-document) lines-per-page) ...)`. This method should simply return a list of lines to be printed for the document. The `lines-per-page` argument can be used to help the method decide where to add page header or footer lines if it desires to do so. The default implementation of this method creates lines to display the document's slot names and values. You can specify the font to be used by overriding the method `font-name` for your `lisp-document` subclass which should return a string with the font name. Similarly, you can specify the size of the font by overriding the method `font-size`. The defaults for these methods for the `lisp-document` class are "Courier" and 8.0 respectively. The `lines-per-page` argument value will be computed automatically for the specified font and font size. It is up to the application to assure that any individual line returned is not longer than can be printed. It will simply be truncated if it is too long. The default `print-lines` method that prints slot values for `lisp-document` instances demonstrates one way that long lines might be broken automatically into multiple lines.

The `print-graphic` method is passed two arguments, a view and a rectangle within the view in which the graphic should be drawn. The rectangle may be the entire view or any small part of it. This is provided as a way to improve efficiency. If it is easy for your drawing function to limit what it draws to that rectangle, then it should do so. But it is not necessary to limit drawing to the rectangle. You can draw the whole view and only what is inside the rectangle will be used. For drawing in print views I suspect that the rectangle will almost always be the entire view so you wouldn't gain too much by restricting your drawing to that rectangle. This is essentially the same sort of call made for on-screen window drawing and I imagine that the rectangle was included here simply to make it easy to use the same method that you use for drawing in a window for printing. In fact, as you'll see in a second, we do exactly that for our `squiggle-view`.

Let's now look at how we support printing in the `squiggle-document` class. We define two methods as follows:

```
(defmethod print-lines ((self squiggle-doc) lines-per-page)
  (declare (ignore lines-per-page))
  nil)
```

As discussed above, making `print-lines` return `nil` triggers the call to `print-graphic`.

```
(defmethod print-graphic ((self squiggle-doc) pr-view rect)
  (draw-in-view-rect pr-view
    self
    (/#/bounds (squiggle-view self))
    rect))
```

Our `print-graphic` method calls a method that we'll examine in a moment. This method draws our squiggles in the print view. I defined that method to take four arguments and we will look at those in a little more detail when we discuss that method. This completes our discussion of `squiggle-doc.lisp`.

Next let's take a look at exactly what a squiggle is and how it is created. The `squiggle` class and its associated methods are defined in the `squiggle.lisp` source file. A squiggle object is effectively an arbitrarily shaped curve that the user draws in a window. To create one, the user will click the mouse and drag it around however it pleases them. When the mouse button is released the curve is complete. The implementation of this object will make use of the Cocoa class `NSBezierPath`. This is a class that effectively collects an arbitrary sequence of graphics commands and stores them within itself. When you want to actually draw it, there is a command to do so which carries out that sequence of drawing commands. As we'll see later, you can also transform that curve in arbitrary ways before drawing it if desired. To make things a little more interesting we'll give each line that the user creates a new random color and a random thickness (within a small range).

```
(defun random-color ()
  (let ((red (random (gui::cgfloat 1.0)))
        (green (random (gui::cgfloat 1.0)))
        (blue (random (gui::cgfloat 1.0)))
        (alpha (+ (gui::cgfloat 0.5) (random (gui::cgfloat 0.5)))))
    (/#/colorWithCalibratedRed:green:blue:alpha: ns:ns-color
      red
      green
      blue
      alpha)))
```

This `random-color` function is a utility function that does just what it promises and generates a random `NSColor` object.

```
(defclass squiggle ()
  ((path :accessor path :initform nil)
   (color :accessor color :initform (/#/retain (random-color)))
   (thickness :accessor thickness :initform (+ 1.0 (random 3.0)))))
```

The `squiggle` class itself consists of just the `NSBezierPath` instance, the color, and the thickness.

```
(defmethod add-point ((self squiggle) point)
  (with-slots (path) self
    (if path
      (/#/lineToPoint: path point)
      (progn
        (setf path (make-instance ns:ns-bezier-path))
        (/#/moveToPoint: path point)))))
```

The only other method for squiggles is the one that either starts it at the point specified by its argument or extends it to that point. That's all there is to a squiggle.

Finally we'll take a look at how all this is pulled together to display something interesting. That is defined in the `squigg-view.lisp` file.

```
(defclass squigg-view (ns:ns-view)
  ((document :accessor document :foreign-type :id)
   (:metaclass ns:+ns-object))
```

A `squigg-view` is simply an `NSView` that has a pointer to an associated document.

```
(objc:defmethod (#/drawRect: :void)
  ((self squigg-view) (rect #>NSRect))
  ;; this is broken up so the same draw function can be called for printing
  (draw-in-view-rect self (document self) (#/bounds self) rect))
```

The Objective-C method draw-rect: is called for all views that are to be displayed in a window. In our version, we'll just call the same draw-in-view-rect method that we used for printing.

```
(defmethod draw-in-view-rect ((self ns:ns-view) (doc squiggle-doc) bounds rect)
  (declare (ignore rect))
  (let ((rotations (rotations doc))
        (initial-transform (#/transform ns:ns-affine-transform))
        (transform (#/transform ns:ns-affine-transform))
        (my-bounds (#/bounds self))
        (gradient (#/initWithStartingColor:endingColor:
                    (#/alloc ns:ns-gradient)
                    (#/blackColor ns:ns-color)
                    (back-color doc))))
    ;; Draw the background gradient.
    (#/drawInRect:angle: gradient my-bounds 45.0)
    ;; For printing, center everything on the print view. This does nothing in a window where
    ;; mybounds and bounds are the same.
    (#/translateXBy:yBy: initial-transform
      (/ (- (ns:ns-rect-width my-bounds) (ns:ns-rect-width bounds)) (gui::cgfloat 2.0))
      (/ (- (ns:ns-rect-height my-bounds) (ns:ns-rect-height bounds)) (gui::cgfloat 2.0)))
    (#/concat initial-transform)
    ;; Create a coordinate transformation based on the value of the rotation slider to be repeatedly applied below.
    ;; Rotate around the center point of the displayed view by translating before rotating and then translating
    ;; back. Note that if we rotated around the center of the print view as well, we'd get something that looked
    ;; pretty different when printed than it did in the original window.
    (#/translateXBy:yBy: transform
      (/ (ns:ns-rect-width bounds) (gui::cgfloat 2.0))
      (/ (ns:ns-rect-height bounds) (gui::cgfloat 2.0)))
    (#/rotateByDegrees: transform (/ (gui::cgfloat 360.0) rotations))
    (#/translateXBy:yBy: transform
      (/ (ns:ns-rect-width bounds) (gui::cgfloat -2.0))
      (/ (ns:ns-rect-height bounds) (gui::cgfloat -2.0)))
    ;; for each rotation draw all the squiggles
    (dotimes (i rotations)
      (dolist (squiggle (squiggle-list doc))
        (let ((path (path squiggle)))
          (#/setLineWidth: path (thickness squiggle))
          (#/set (color squiggle))
          (#/stroke path)))
        ;; now apply the transform to rotate a little more for the next iteration
        (#/concat transform))))
```

The draw-in-view-rect method does all of the heavy lifting for our display. I'm not going to go through this line-by-line, but I will point out some of the more interesting features of this function to help you understand why it does things the way it does. To make the window visually more interesting the background is drawn as a color gradient from the lower left to the upper right corner of the view. The gradient goes from black to a base color that the user can select using the color well that we added to the window. Over this background a squiggle window draws the collection of squiggles that the user created. It then rotates each of those squiggles a little bit about the center of the display and displays them again. The value that the user sets on the slider at the bottom of the screen determines how many rotations will be done. A simple calculation determines how big each rotation should be in order to make sure that the sum of all rotations is 360 degrees.

I initially used the drawing routine defined in Apple's squiggles demonstration application pretty much as written without looking too closely at what it did. But that application didn't support printing and I discovered when I applied the same function to printing that the printed document didn't look all that much like what I was seeing in the window. A short inspection of the code explained why. What is drawn is sensitive to the current shape and size of the window. The squiggle positions are all designated relative to the lower-left corner and then rotated about the center of the window. If you change the window's size or relative dimensions, then the center of the window (and hence the center of rotation) also moves. You can see that yourself once you have a real squiggle window displayed just by changing its size and relative dimensions. Since a print view is, in general, a different shape than the display window, what is printed didn't look the same as what was displayed on screen. I found that undesirable, so I modified the drawing function to use the display's rectangle to determine the center of rotation at all times. I did, however, translated everything for the print view so that the printed version is centered correctly. By doing these things the printed view looks much more like the displayed view. Note that drawing the gradient must be different in the print view just to make sure that the entire rectangle is covered. This does make the printed view look somewhat different than the displayed view, but in my judgment this was the most pleasing choice. You may of course make different choices in your version if desired.

I will leave it to each of you to check out Apple's documentation for the NSAffineTransform class if you want to understand more about what it does and how transformations are used within drawing functions. Suffice it to say that we define a transform that rotates a scene a little bit and apply that transform to the current drawing environment at the end of each iteration. That causes the drawing that is done in the next iteration to be rotated slightly more than what was drawn in the previous one.

```
(objc:defmethod (#/mouseDown: :void)
  ((self squigg-view) (event :id))
  (unless (eql (document self) (%null-ptr))
    ;; convert from the window's coordinate system to this view's coordinates and start a new squiggle there
    (start-new-squiggle (document self)
      (#/convertPoint:fromView: self
        (#/locationInWindow event))
```

```
(%null-ptr))))))
```

The `#/mouseDown:` method is called, obviously enough, when a mouse-down event occurs in the squigg-view. It simply converts the point to a coordinate within the squigg-view and then calls the document method `start-new-squiggle` that we have already seen.

```
(objc:defmethod (#/mouseDragged: :void)
  ((self squigg-view) (event :id))
  (unless (eql (document self) (%null-ptr))
    ;; convert from the window's coordinate system to this view's coordinates and start a new squiggle there
    (continue-squiggle (document self)
      (#/convertPoint:fromView: self
        (#/locationInWindow event)
        (%null-ptr))))))
```

The `#/mouseDragged:` method is called periodically as a mouse is dragged with the button held down. Our method simply tells the document to add to the squiggle currently being defined.

Some of you may be wondering at this point why we didn't just collect all the squiggles in the squigg-view object itself rather than making them belong to the squiggle-doc. I suppose the easy answer is that this is what Apple's example code did. But a little reflection makes one realize that by doing this all of the data associated with the object is kept in one place, namely the squiggle-doc instance. That made it pretty easy to save and restore. If we had permitted the squigg-view to own the squiggles, then we would have had to save the squigg-view when we saved the document. That could have worked too, but would have meant saving a view object that we would have had to add to a window in some fashion when the document was loaded and this all gets fairly messy as you start to think about it. In general it's best to keep views separate from data and use data to cause the views to display in different ways.

```
(objc:defmethod (#/isOpaque #>BOOL)
  ((self squigg-view))
  #YES)
```

The `#/isOpaque` method tells the runtime that this view completely fills its rectangle. So if there were potentially something behind it, the system would not bother drawing it first, but would leave it blank. It's a small efficiency measure.

This concludes our discussion of all the source needed for the Squiggle application.

We're getting close now to actually seeing our application do something interesting in a window. The last thing we need to do before trying it is to tell the tools about our window NIB so that it will put it into the bundle for us. You can do that by clicking the "+" button associated with the NIB table and using the open dialog that pops up to select the NIB file that you saved from IB. This will NOT be the main NIB. The main NIB designates the one that is first loaded when the application is started. It is usually assumed that the main NIB will contain a main menu that will be displayed for the application. We'll see in how to do that after we test our window behavior.

At this point your LAD window should look like Figure 19 below:

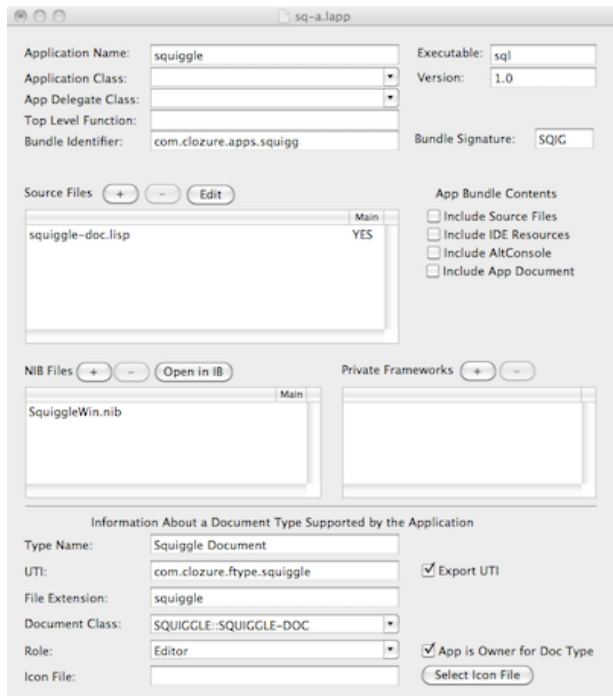


Figure 19: LAD window with NIB defined

Loading an app with windows under the IDE

We're now ready to try out the application under the IDE. You do this by selecting "Load App Under IDE" from the Dev menu. Once you have done this

you will see the same new Squiggle Document menu items in the File menu that we did previously. But since we have added a window definition, Lisp code to draw in it, and specified how to print it graphically, the behavior of those menu items will be much different. Go ahead, select "New Squiggle Document" from the File menu and play around with it. Create your own squiggle lines by clicking in the window and dragging the mouse around. Try writing your name or something like that. Then change the rotation slider to see what happens. Change the window's shape to see what effect that has. Save your squiggle document, close it, print it, open one from disk or whatever you want.

I will warn you that there are a few things that will not work entirely as expected. For example, do not try to open a Squiggle Document using the "Open Recent" menu choice, even though recent squiggle documents will appear there. Also, if you select "Open" from the file menu and select a .squiggle document you will see that it does not open in a squiggle window. Neither of these things work as expected because although we have gone out of our way to make it look like the CCL IDE now understands the relationship between squiggle documents on disk and the squiggle-document class, the reality is that it does not. That can't really happen until we make some changes to the Info.plist in the application bundle. And since we don't want to go mucking around inside the CCL IDE bundle, we can't REALLY make that happen. However, once we create a stand-alone application things will work more normally.

If you are wondering how this is all made to work under the IDE, be patient for a bit and a short overview will be provided after we show how to add a main menu to your application.

Adding A Main Menu To Your Application

While we now have a document-based application that provides a nice window for making changes to it, we are missing one ingredient for a complete application and that is a main menu that will be displayed. In this section we will add one and also show how to test it under the IDE. To create one we will go back to IB.

Creating a main menu is fairly easy in IB. Choose New from the File menu and select as shown in Figure 20 below:

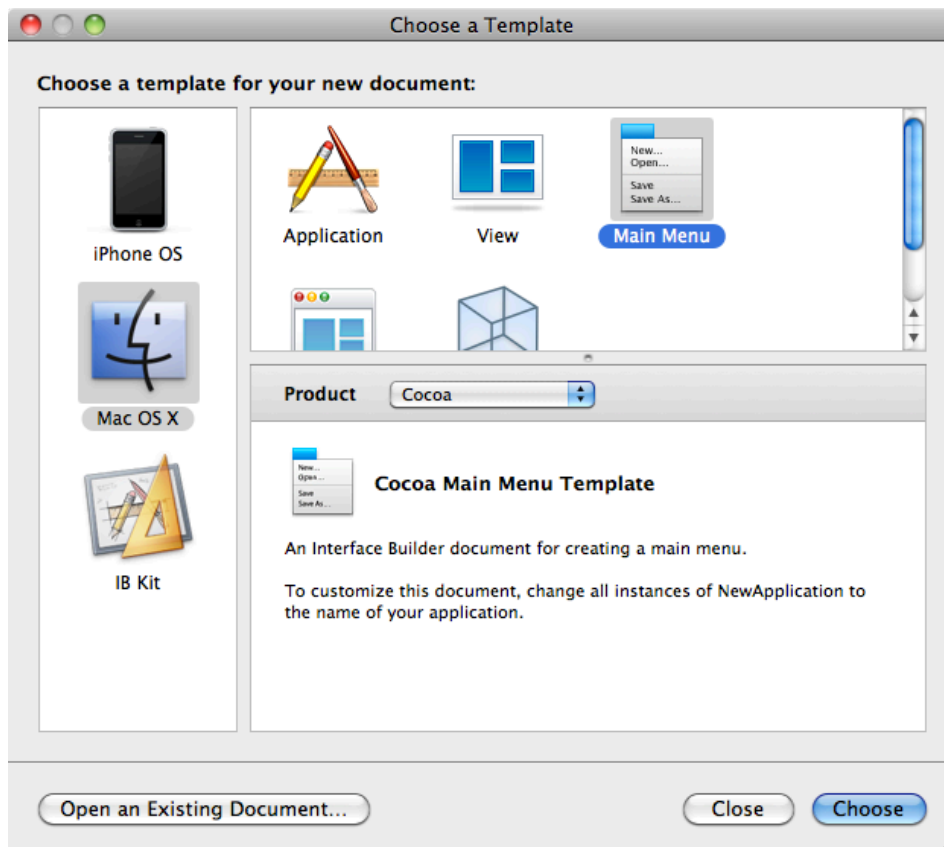


Figure 20: Making a new Main Menu in IB

When you make this choice you will see a main menu design window that should look pretty familiar. See Figure 21 below:

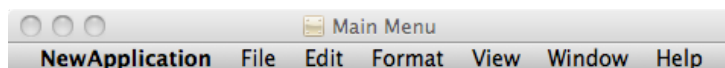


Figure 21: The new Main Menu

If you click on a menu item it will expand to show sub-menus which can be individually selected. You can modify any of those to suit your liking. I suggest that initially you make the following changes. First change the name of the application menu from "New Application" to "Squiggle". Just double click directly on it to edit and make the change. Click on the Squiggle menu that you just renamed to show its menu items. Change all instances of "NewApplication" to "Squiggle" in those names as well.

Next delete the Format and View menus because we will have no need for those.

Finally we have to connect up a few things that are not connected in the initial default Main Menu. Click on the File menu to show its menu items. Control-click on the "Open" item and drag to the red First Responder object in the NIB document window. Select "newDocument:" as the action from the window that pops up when you release the mouse button. Similarly, connect the the "Open ..." item to the "openDocument:" method of the First Responder and the "Print Document ..." item to the "printDocument:" method of the First Responder.

That's enough for now. Save your file (remember to use NIB format rather than XIB which is the default in the save dialog).

Next go back to the CCL IDE and add your new NIB to the NIB table for the Squiggle application. Also make it the main NIB by clicking in the main column and typing anything at all. "YES" will appear. Assuming that you just left the NIB with the name MainMenu, your LAD window should now look like the following:

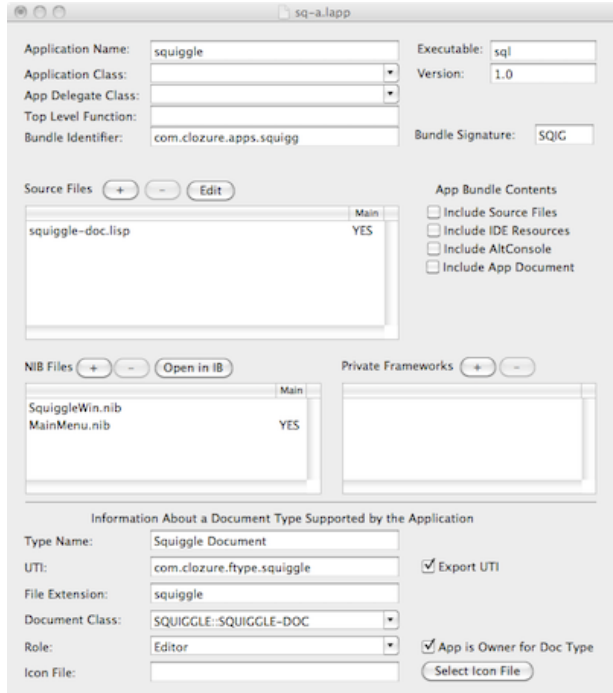


Figure 22: The LAD window with a MainMenu NIB.

OK, now select "Load App Under IDE" from the Dev menu. It is not necessary to restart the IDE before doing this; the tools are smart enough to reload an application correctly and remove all traces of previous loads (e.g. the additional menu items that were created). You'll now see that the CCL IDE menubar has been replaced by the new main menu that we just defined. The only little glitch there is that the leftmost menu is still named "Closure <something>" rather than Squiggle as we defined it in IB. This is done automatically and I couldn't seem to change it, so let's just call this one of those little annoyances and live with it. All items within that menu are our new items.

In addition to the main menu for the Squiggle application, we also still have the Dev menu hanging around as the last menu at the right end of the menubar. We need this to be able to continue to debug our application. At this point two more of the choices in the Dev menu come into play. When you click on the Dev menu you will now see that the "App Menus" item is checked and the "CCL Menus" item is not. The inclusion or exclusion of these menus from the menubar can be toggled by selecting from the Dev menu. You can see either or both or neither set. In all cases the Dev menu will remain as the rightmost menu (although when it is the ONLY menu shown it will still have that annoying "Closure ..." title).

Note that when the CCL menus are not shown you will also not have access to any of the keyboard shortcuts that are provided to select them. Also, if both sets of menus are shown, then any keyboard shortcuts common to both sets will be assigned to menuitems within the first set shown.

We now have a fairly complete application running under the IDE. You can test out functionality triggered by actions you take in your window or menus. When things don't go entirely as expected (which I can assure you will happen) you can inspect or otherwise manipulate your code and/or the objects it creates just as you would normally debug any Lisp code. You will likely become familiar with the AltConsole and how to interact with it. This will pop up anytime there is some sort of exception that occurs when processing an event. So even though it is executing along through your Lisp code, the AltConsole will be where information is displayed, not in a listener window. I'll admit that I'm not yet very good with it. I find that for the most part I can figure out what is going by the displayed message and judicious use of the :b command to get a backtrace in the AltConsole window. In many cases a subsequent :pop command can get you back into a running state within CCL. Sometimes other available choices will let you move on to process the next event, which may also get you back in operation.

You can do many things that Objective-C programmers cannot easily do. You can invoke Objective-C functions directly from a listener window, you can modify function or method source, reevaluate it, and immediately see the effects in the changed behavior of your running application. You can go off to IB, modify a NIB definition, go back to CCL and update the bundle, open a new instance of that window and immediately see the changed effects (incidentally, accomplishing that last feat required defining a new NSBundle subclass that gets NIBs and other resources directly from the bundle each time they are requested rather than just using a cached version that was created at the time the bundle was first loaded). You will have a truly dynamic way to define a Cocoa application.

Application Delegates

Although we are not going to specify an application delegate right now, we will be designating one later and it is useful to understand what they do

before discussing what's going on to run an application under the IDE. Many, but not all, applications designate an application delegate object. The use of delegates is an Apple invention that allows developers to modify or augment the default behavior of many of their standard objects (applications, windows, various sorts of controls and view objects, etc.). When certain designated messages are sent to one of those standard objects it gives its delegate a chance to handle it if it so desires. If not, then the standard object handles it itself. Delegates for the NSApplication object often implement things like menu actions that open up preference panes or special windows of one sort or another within the application. The CCL IDE uses an application delegate of the class `lisp-application-delegate` for some of these sorts of things. In your application you can choose to do similar things with your own application delegate class.

You can specify what sort of object should be the application delegate in either of two ways. First you can add a delegate object to your MainMenu NIB that is defined in IB. To do that you would first create a new sub-class of NSObject in IB (or import a .h that you created for a Lisp-defined app delegate class). Drag an instance of that class to the document window and create a link from the File's Owner to that instance by control-clicking on the File's Owner and dragging to your object. Choose "delegate" from the popup that is shown when the mouse button is release. In theory you could drag from either the File's Owner object or from the Application object that is shown since in a stand-alone application they will be the same object. However, as we'll discuss in a moment, when your application is loaded under the IDE we will use another object as the File's Owner and for our purposes we want that object to have the delegate link. If you instead linked your delegate to the application object then when we loaded this NIB the CCL IDE's application object would get a new delegate and many things would no longer work very well inside the IDE.

The other way to specify what sort of object you want to be your delegate is to specify a delegate class in the LAD window. When we create a stand-alone Squiggle application we will do that. This results in a value being put into the Info.plist in the application bundle. That value is used to create an application delegate object when a Lisp Cocoa application initializes itself. The notification message `#applicationWillFinishLaunching` is normally sent by the NSApplication object to its delegate just before the event loop is started for the application. This notification will also be sent to any application delegate that you define when you run your application under the CCL IDE. As we'll see later, the prototype application delegate classes that have been defined for your use implement that Objective-C method by calling the Lisp function `application-will-finish-launching`. This is merely a little convenience. For application delegates that you define you can either inherit from the one of the provided classes and then implement that Lisp method or directly implement the Objective-C method to do whatever other initialization may be desired for your application.

What's going on when an application is run under the IDE?

So how exactly did all this work? For those who are curious I'll provide a brief synopsis here about what is going on to make your application run under the CCL IDE. If you don't really care and just want to move on to creating your own stand-alone application, feel free to skip the remainder of this section.

When you select "Run App Under IDE" from the Dev menu, a `lisp-doc-controller` object is created for your application. This object performs many useful roles. In the simple case where you do not have a main menu specified it will create those default menu items that we saw previously for New, Open, and Print and add them to the default File menu. It sets itself up as the target for the actions of those menus so that when they are selected it will take the intended action in a fashion very similar to what an `NSDocumentController` would. It knows how to mimic the normal message flow for documents in a more complete application.

The `lisp-document` class also contributes some default behavior that makes it possible to run a less than fully specified application under the IDE. If you have not yet defined a NIB to use for the document window, then it creates a proxy window as we saw previously. And it provides default printing.

Things get a little more interesting when the application definition is complete with both a window and a main menu. In this case, the controller first acts something like an NSApplication object. A main NIB is normally loaded by an NSApplication at startup with itself as the File's Owner for that NIB. Our controller object will do much the same for a main nib file specified for your application. If that NIB contains a MainMenu, the NIB loading mechanism will automatically replace the main menu in the application with the one that you specified. Since we would still like to have both the standard CCL menus and our Dev menu available after that, the controller arranges to save those off where they can be found later and added back to the menubar.

If you have specified an object to be the delegate object for your application by linking it to the File's Owner in your nib, then it will be created and linked as the delegate of the controller automatically when the NIB is loaded. If you specified a delegate class in the LAD window, the controller object will make an instance of that class and set its own delegate link to that instance. The controller object will also send the `#applicationWillFinishLaunching` notification message to the delegate if it exists and has implemented that method.

Our objective is to be able to create a main menu definition in IB that we can use without change both when running under the IDE and later as part of a stand-alone application. That required a little slight of hand. Normally the messages sent when the New and Open menu selections are made end up with whatever object is the shared `NSDocumentController` object. In the CCL IDE this would be a `HemlockDocumentController` object. But that object will treat such messages as requests to open up a CCL document, not one of our application documents. So we need to intercept the message at some point and decide whether it originates from one of our newly added application menu items or from one of CCL's menu items. If it is the former, it will be acted on it directly to open one of our application's documents and if it is the latter then the message will be passed on to the shared controller for action.

The messages from the menu items of interest are directed at the First Responder object, so intercepting them requires understanding how the next responder chain works and what the order of precedence is. For now it is sufficient to understand that such messages eventually go to the application object, which gives its delegate a chance to handle them, and then on to the shared document controller. So in our case, we can add new methods to the application delegate object for CCL, recognizing that they will take precedence over corresponding methods defined for the shared controller. We create these additional methods at runtime when the tools are installed, so they do not exist in the default CCL IDE. We first hand those messages to a controller method which checks to see whether the sender of the message is any of our newly installed application menu items (information about menu items is cached when they are first loaded). If the message originated from one of the new menu items, then we give it to the appropriate `lisp-doc-controller` and let it mimic the actions of an `NSDocumentController` just as it did previously. If the message does not originate from one of the new menu items, then it is passed on to the shared document controller for normal action.

Note that this only happens for selected standard messages. What about menu messages that you would normally want your own application delegate to handle? In theory those could either be directed at the First Responder or directly at the File's Owner (which would give its delegate a chance to handle them). To make this work when your application is run under the IDE, your menu items must do the latter and set the File's Owner as the target of their messages. This will assure that your delegate is given the chance to handle them either when run under the IDE or run as a stand-alone application. Or you could add your delegate object to the MainMenu NIB, link it as the delegate of the File's Owner as previously described, and then target your menu actions directly at that delegate object. Lots of ways to get the job done ...

Creating a stand-alone application without the IDE

You now have your application running pretty well under the IDE and want to create a complete stand-alone application that can be run independently or even moved to a different system without CCL installed and run there. This will be a fairly easy task.

The Lisp top-level code which starts up Cocoa applications will create an instance of a subclass of `NSApplication`. This must be specified in the bundle's Info.plist file. The tools will do this for you using the class specified in the Application Class field of the LAD window. For stand-alone applications that do not include IDE resources it is sufficient to use the `NSApplication` class itself. For applications that DO include IDE resources you will want to use the `gui::lisp-application` class, just as the CCL IDE does. You can choose either of these using the pull-down menu attached to the "Application Class" text field in the LAD window. For now select `ns:ns-application`.

The Lisp startup code will also create an application delegate object if necessary. We have already seen that it is possible to replace this object by specifying one in the LAD window or specifying and linking one to the File's Owner object in the main NIB file. For a stand-alone application to function properly it MUST do one of these. In effect we will provide one to replace the default used by the CCL IDE. I have provided a default class called `app-dev::simple-lisp-app-delegate`. If you use the pull-down menu next to the "App Delegate Class" field in the LAD window you can select this class. That menu will show all classes that derive either from the `lisp-application-delegate` or the `app-dev::simple-lisp-app-delegate` classes. If you define your own delegate class that does not inherit from either of those, you can type its name in directly. Now your LAD window should look like Figure 23 below:

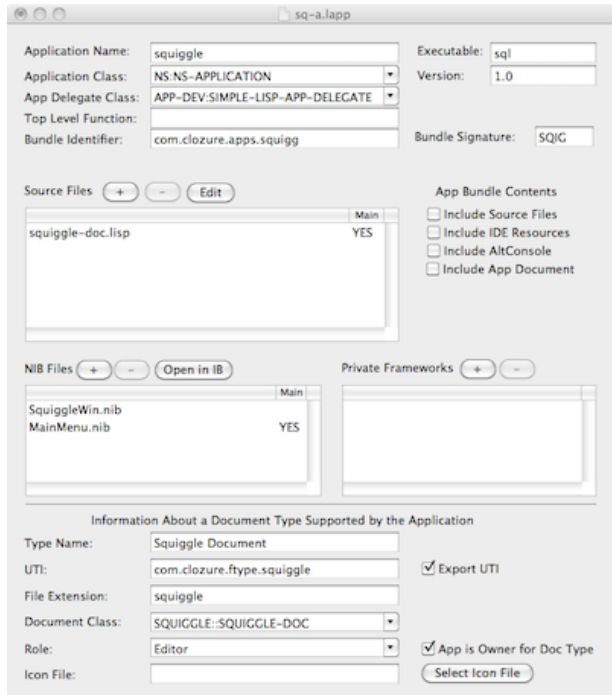


Figure 23: Complete LAD window for stand-alone Squiggle application

The only thing left to do is to install an executable into the bundle. When the application bundle is double-clicked in the finder, that executable inside the bundle will be run. The tools create this executable using the CCL save-application function, but they do so within a separate instance of the CCL command-line application that is used so that the one running our tools remains up and running while we do so. We open up a pipe to that subordinate CCL instance and interactively tell it what to load to properly initialize itself for saving. If any error occurs during that process it will be captured and an alert box will open to tell the user what happened. This is a bit more responsive than the alternative of just launching a separate CCL instance with batch commands telling it what to load and how to save itself.

Select "Install Executable" from the Dev menu to start this process. It first causes the subordinate Lisp to load all the Cocoa functionality which can take a bit of time, so be patient. You will know when this is complete because a new IDE window for the subordinate Lisp will briefly open up on your system. The tools will then instruct the subordinate Lisp to load your code along with a small amount of helper code. Finally it will tell the subordinate Lisp to save itself into the application bundle in the proper location under the name specified for the executable in the LAD window. Once this has been done, you can either double-click on the bundle in a Finder window to execute it or just select "Run App Stand-Alone" from the Dev menu. If you were to make that Dev menu selection before installing a valid executable, obviously nothing would happen. Note that sometimes the application bundle icon in the finder will appear as if it is not executable. But the tools "touch" the bundle after the executable is installed, so unless something has gone seriously wrong you will be able to double-click on the bundle icon and it will execute even if it does not initially look like that will work. Thereafter it should appear as a normal application in Finder windows.

You can reinstall a new executable at any time if you find that changes are needed to your Lisp code. You can edit NIB files and then use the "Initialize Bundle" menu selection to cause that NIB file to be moved into the bundle. It is not necessary to install a new executable after doing that. You will have to re-execute your application for it to recognize that a NIB file within its bundle has been changed.

Congratulations! You just created your own stand-alone Lisp application.

Creating a stand-alone application including IDE resources

It is also possible to create a stand-alone application that includes IDE resources. If you find it is the case that you are unable to debug a problem when running your application within the CCL IDE then you want access to a Listener window in your stand-alone application. Or perhaps you just want a Listener for other reasons in your own application. In either case you can easily create such an application. Note that many of the functions that you have access to within the CCL IDE depend on loading other things that are found within the CCL directory. Assuming that you have defined your

ccl-ide-init.lisp file as discussed earlier in this document, that won't be a problem. But you will not be able to take this application to another system where CCL is not installed and expect everything to work correctly (although many things will).

Assuming that this is to be a temporary thing, I'd suggest that you save your LAD under a new name. I called mine squigIDE just to make it clear what I was doing.

First you will want to make sure that the application class is changed to `gui::lisp-application` and your delegate class is either `gui::lisp-application-delegate` or some some class of your own that inherits from that class. Without doing those things, many of the normal CCL functions will not work correctly.

Next you should decide which of the CCL menu items you may also want to include in your application's main menu since you won't be able to switch back and forth as we did when running within the IDE. If you control-click on the IDE bundle (called "Clozure CL64.app" on my system and likely something similar on yours) in a Finder window and select "Show Package Contents" from the pop-up, a Finder window will open to show the bundle contents. Double-click on "...Contents/Resources/English.lproj/MainMenu.nib" within that directory to open up CCL's main menu in IB. You can copy and paste individual menu items from there into your own menu if you so desire. For demonstration purposes I did that with the "New Listener" menu item under the File menu. After doing so, make sure to link your new menu items in the same way that they are in the CCL MainMenu. In the case of the "New Listener" menu item, that means that I wanted to send the "newListener:" message to the First Responder object. But in my MainMenu.nib the First Responder object doesn't understand that it should respond to this item, so we must add it. Do that by clicking on the First Responder object in your NIB document window, clicking on the attributes icon (farthest left) in the browser window, and clicking the "+" button to add that message to the list of First Responder actions.

Then "Save As ..." your modified NIB file under a new name. I called mine Main Menu2.nib. In your LAD window remove MainMenu from the NIB table using the "-" button, and replace it with MainMenu2 using the "+" button.

Next check the "Include IDE Resources" and "Include AltConsole" buttons.

Also change the Application Name, Bundle Identifier and Bundle Signatures so that the OS knows that this is a different application than the Squiggle application we previously created. I called those squigIDE, com.clozure.apps.squigide, and SQIDE respectively.

To avoid other problems, uncheck the "Export UTI" and "App is Owner for Doc Type" checkboxes at the bottom.

When you have done all that, your LAD window should look like Figure 24 below:

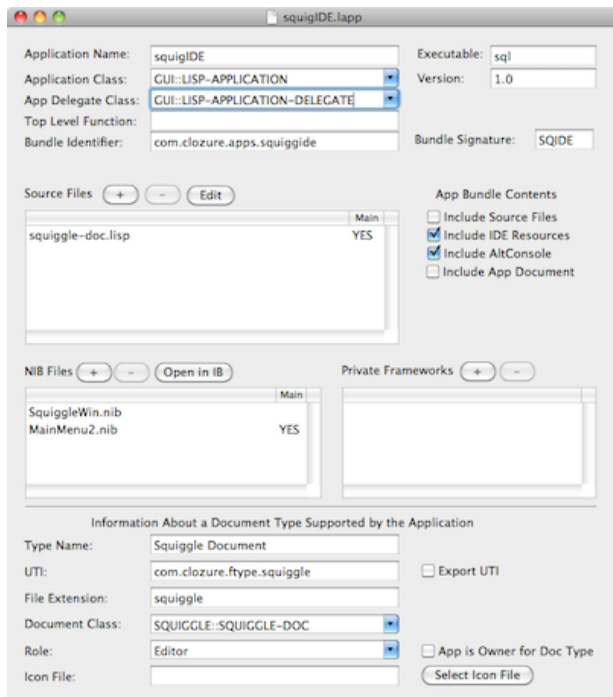


Figure 24: LAD window for stand-alone app that included IDE resources

Next create a brand new bundle for this application by selecting the "New Bundle" item under the Dev menu. Save it in some convenient location.

Finally, select "Install Executable" from the Dev Menu to put an executable into the new bundle. Assuming that all went well, your application is now executable by double-clicking it in the finder or selecting "Run App Stand-Alone" from the Dev menu. A Listener window will open as it normally does when the CCL IDE is started. Within that application you can use all of the Squiggle menu items and also select the "New Listener" item to have it do what you would normally expect.

Example 2: The Loan Application

This second example illustrates some additional features provided by the Application Tools and various adjunct classes and methods. Specifically you will see an example that shows more direct binding of interface object values to Lisp slots, the use of text for printing the document, and the use of a bound-slot-modified method that captures the entire state of the document instance and arranges for it to be restored when an undo is requested. The separation between Model and Controller functionality is also more clearly delineated in this example.

Let's start by creating a new LAD for our Loan application. All referenced files are found in:

...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/LoanStandAlone

Select "New Lisp Application" from the File menu in the CCL IDE. Modify the window to match Figure 25 below. Add the source file load-app-document.lisp to the source table and make it the main source file.

Figure 25 shows the initial LAD (Loan Application Document) window. The window is titled "Untitled 4". It contains several sections for configuring the application:

- Application Name:** LoanCalc
- Application Class:** (empty)
- App Delegate Class:** (empty)
- Top Level Function:** (empty)
- Bundle Identifier:** com.closure.apps.loancalc
- Executable:** lcalc
- Version:** 1.0
- Bundle Signature:** LOAN

Below these fields are three sections for managing files and frameworks:

- Source Files:** A table with a "Main" column. The table is currently empty.
- NIB Files:** A table with a "Main" column. The table is currently empty.
- Private Frameworks:** A table with a "Main" column. The table is currently empty.

There are buttons for adding, removing, and editing files in each of these sections. Additionally, there are buttons for "Open in IB" and "Select Icon File".

On the right side, there is a section titled "App Bundle Contents" with four checkboxes:

- ☐ Include Source Files
- ☐ Include IDE Resources
- ☐ Include AltConsole
- ☐ Include App Document

At the bottom, there is a section titled "Information About a Document Type Supported by the Application" with several fields and checkboxes:

- Type Name:** (empty)
- UTI:** (empty)
- File Extension:** (empty)
- Document Class:** (empty)
- Role:** (empty)
- Icon File:** (empty)
- ☐ Export UTI
- ☐ App is Owner for Doc Type
- Select Icon File** button

Figure 25: Initial LAD for the Loan Application

As with the squiggle example, if we now choose "Load App Under IDE" from the Dev menu, a dialog will ask you where you want the bundle to be created. Select any convenient location. The only other thing that will happen is that the source code will be loaded. This will make the loan-doc::loan-doc class known. This will permit us to go ahead and fill in the document information at the bottom of the LAD window. Also save your document so we don't lose it. Any name and location is fine. I called mine "loandoc". This should now look like Figure 26 below.

Figure 26 shows the LAD window after adding document information. The window is titled "loandoc.lapp". The fields and sections are the same as in Figure 25, but with updated values:

- Application Name:** LoanCalc
- Application Class:** (empty)
- App Delegate Class:** (empty)
- Top Level Function:** (empty)
- Bundle Identifier:** com.closure.apps.loancalc
- Executable:** lcalc
- Version:** 1.0
- Bundle Signature:** LOAN

The **Source Files** table now contains one entry:

	Main
loan-app-document.lisp	YES

The **App Bundle Contents** section remains the same.

The **Information About a Document Type Supported by the Application** section has updated values:

- Type Name:** Loan Calculation
- UTI:** com.closure.files.loancalc
- File Extension:** loan
- Document Class:** LOAN-DOC::LOAN-DOC
- Role:** Editor
- Icon File:** (empty)
- ☒ Export UTI
- ☒ App is Owner for Doc Type
- Select Icon File** button

Figure 26: Loan App LAD with document information provided

If you now select "Load App Under IDE" from the Dev menu, the tools have enough information about your document to create the default File menu items: "New Loan Calculation" "Open Loan Calculation" and "Print Loan Calculation". If you select "New Loan Calculation" a proxy window will be created. From a listener window you could then access the underlying loan-doc instance and test all of its functionality by directly setting values in its slots and exercising various methods that have been defined for it. You could then use the new File menu items to save, close, re-open, and print your loan document.

We will now stop to take a look at the loan-app-document.lisp source. I am not going to examine the algorithms for computing loan values; that is covered in some detail in Project 6 of the *InterfaceBuilderWithCCLTutorial* document. If you contrast the code there with the implementation in the loan-app-document.lisp file you will see that although it is substantially the same, there are a few simplifications that are made possible in the latter source file because of the use of a simpler binding mechanism for values. We were also able to bind directly to Lisp slots that were :foreign-type in the earlier implementation and more normal slots in the implementation described and used here. Let's start with the loan-doc class definition.

```
(defclass loan-doc (lisp-document)
  ((loan-amount :accessor loan-amount
                :kvo "loanAmount"
                :initform 0)
   (interest-rate :accessor interest-rate
                  :kvo "interestRate"
                  :initform 0.0)
   (loan-duration :accessor loan-duration
                  :kvo "loanDuration"
                  :initform 0)
   (monthly-payment :accessor monthly-payment
                     :kvo "monthlyPayment"
                     :initform 0)
   (origination-date :accessor origination-date
                     :kvo "origDate"
                     :initform (now))
   (first-payment :accessor first-payment
                   :kvo "firstPayment"
                   :initform (next-month (now)))
   (total-interest :accessor total-interest
                   :kvo "totInterest"
                   :initform 0)
   (pay-schedule :accessor pay-schedule
                  :initform nil)
   (max-dur :accessor max-dur
             :kvo "maxDuration"
             :initform nil)
   (min-dur :accessor min-dur
             :kvo "minDuration"
             :initform nil)
   (min-pay :accessor min-pay
             :kvo "minPay"
             :initform nil)
   (compute-mode :accessor compute-mode
                  :initform 0)
   (last-slot-set :accessor last-slot-set
                  :initform nil))
  (:default-initargs
   :doc-class nil
   :menu-name nil
   :file-ext nil)
  (:metaclass ns:+ns-object))
```

Perhaps the first thing to notice is that there are no :foreign-type slots. This is made possible by binding mechanisms that permit us to bind interface element values directly to Lisp slots. The :KVO slot options each specify a string that can be used to bind that slot. Note also that we do NOT use the :undo slot option that was discussed earlier. That is because in general any change to a slot value may result in changes to other slot values as well. And simply setting the slot's value back to what it was previously is not guaranteed to exactly restore the previous state. We will see in a bit how this is handled more generally.

I want to discuss a few of the more specialized slots and the nature of the values that they will hold so that you have those in mind when we get to the point of designing the user interface window for this document.

Several of the slots represent dollar values. There are a couple of different ways that we might represent these values. First, we could use floating-point values. Since dollars always have a couple of places to the right of the decimal, this seems somewhat natural. But business application developers have known for a very long time that this can cause problems. Since the internal binary representation is never exact, you can end up representing a value like \$31.17 as 31.1699999998 or something along those lines. There are other problems that pop up when calculating with such numbers. The other way to represent dollar figures is to use integers with the implicit assumption that what is represented is the value in cents. Every display of such values must be adjusted accordingly. This is typically what a business application developer would do with any currency values.

In the Objective-C world programmers are provided with the NSDecimalNumber class. This effectively keeps an exact integer value for the number and provides conversion to and from other numeric representations. It is displayed in the normal fashion with decimal point included. We will represent currency values as cents in our Lisp slots and specify the use of NSDecimalNumbers for some of the corresponding interface fields. You will see that the conversion back and forth between these at runtime is completely automatic.

The slots origination-date and first-payment are date fields. We will store dates using the normal Lisp date/time encodings. For corresponding user interface elements we will specify a date format. When the binding is made between these the interface object and our Lisp slot, the use of the Date format by the interface element will alert the binding code to expect a Lisp date encoding in the Lisp slot and it will then convert it back and forth appropriately when changes are made on either side of the binding.

The max-dur, min-dur, and min-pay slots signify true or false values for corresponding states of the loan. These will be used to enable and disable the appearance of text in the loan window that alerts the user to these conditions. User interface elements expect these to be of the type BOOL. In Lisp that corresponds fairly naturally to the use of t and nil. And in fact we will use t and nil in this way. The binding code will automatically convert back and forth between these values too.

There are several parameters that occur in the fundamental equation that describes a loan. The compute-mode slot is a controllable parameter that determines which of the variables is treated as the *dependent variable* (i.e. it is solved for using the existing values of the others). We'll see later that the user is able to specify that in the loan window. The parameter selected will also become un-editable. Changing any other variable will result in recomputation of the dependent variable. We don't bind directly to this variable because we want there to be some side-effects in the window (namely not allowing further editing of the field that corresponds to the dependent variable). To maintain the separation between *model* and *controller* functionality, we will let the window controller object (described shortly) to handle changes to the interface and then set the compute-mode for the document.

We will revisit much of this when we get to the design of the loan window in Interface Builder. Next we'll look at some of the lisp-document methods that are overridden for our loan-doc subclass.

```
(defmethod archive-slots ((self loan-doc))
  '(loan-amount interest-rate loan-duration monthly-payment origination-date
    first-payment max-dur min-dur min-pay))
```

The archive-slots method specifies which slots should be included when our document is saved to disk (and of course which slots should be set when it is re-opened later).

```
(defmethod document-did-open ((self loan-doc))
  ;; We just set the state of the document from disk
  (setf (pay-schedule self) nil)
  (compute-new-loan-values self))
```

The document-did-open method is called immediately after an instance has been initialized by opening it from a saved disk file. You would use this method to complete any initialization of other slots that might be required after your specified archive slots have been set. In this case we use the values that were restored to compute the values of a complete pay schedule for the loan, which also computes the total interest as a side-effect.

```
(defmethod window-nib-names ((self loan-doc))
  (list "LoanWin"))
```

This specifies a list of the names of NIB files that describe windows that will be created to represent our document. Most applications will only specify a single name, but multiple names are possible.

```
(defmethod document-window-controller-classes ((self loan-doc))
  (list (find-class 'loan-win-controller)))
```

The list provided by the document-window-controller-classes method corresponds to the list of window NIB names. Each item must be a subclass of NSWindowController. We will look at the code that implements the loan-win-controller class in just a bit.

```
(defmethod print-lines ((self loan-doc) lines-per-page)
  (declare (ignore lines-per-page))
  ;; Default method to support printing a document with text.
  ;; This just prints the values of all the document slots.
  ;; If you are printing graphically, this should be overridden
  ;; to return nil
  (let ((lines nil)
        (blank-line ""))
    ;; print all the basic loan info
    (push (format nil
                  "Loan ID: ~a"
                  (ns-to-lisp-string (#/displayName self)))
          lines)
    (push (format nil
                  "Amount: $~$"
                  (/ (loan-amount self) 100))
          lines)
    (push (format nil
                  "Origination Date: ~a"
                  (date-string (origination-date self)))
          lines)
    (push (format nil
                  "Annual Interest Rate: ~7,4F%"
                  (interest-rate self))
          lines)
    (push (format nil
                  "Loan Duration: ~D month~:P"
                  (loan-duration self))
          lines)
    (push (format nil
                  "Monthly Payment: $~$"
                  (/ (monthly-payment self) 100))
          lines)
    ;; draw spacer line
    (push blank-line lines)
    ;; print the appropriate schedule lines for this page
```

```

(dolist (sched-line (pay-schedule self))
  (push (format nil
    "~1{On ~a balance = $-~$ + interest of $-~$ - payment of $-~$ = ~a balance of $-~$}"
    sched-line)
    lines))
(nreverse lines)))

```

We will print our document as text. To do that, it is only necessary to pass back a list of the text lines. At the top of the printed document we'll print a summary of the Loan parameters. After that we'll print a line for each payment to be made that shows the accrued interest from the last payment, the payment amount, and the remaining loan balance after the payment.

The methods below are not overrides of lisp-document methods. They are used to control the computations that occur when changes are made to loan values.

```

(defmethod bound-slot-will-be-modified ((self loan-doc) slot-name)
  ;; Gets called when any slot in a loan-doc is about to be modified via a KVO connection
  ;; Any other way that the slot is set does NOT result in calling this method
  (unless (eq slot-name (last-slot-set self))
    ;; we treat multiple serial modifications to the same slot as a single undoable event
    (create-undo self (format nil "set ~a" (string-downcase (symbol-name slot-name)))))
  (setf (last-slot-set self) slot-name))

```

The bound-slot-will-be-modified is called just before a bound slot's value is modified. This gives the application a chance to save its state so that the action can be undone. And that's exactly what we do in our function. If we were interested in only a particular slot, we would have added a qualifier to the slot-name argument. But in this case, we want to know when any bound slot has been changed. The first thing we want this method to do is to arrange to "undo" the action that is about to be triggered by setting the slot value. In general, when a user changes the value of an interface element we want the view to be responsive and change instantly. The alternative is to wait until a user "commits" the change by entering a return or exiting the control. To get that level of responsiveness we will set the controls to continuously update their values. That means that we will likely see a number of sequential changes for a slot value. We will treat that sequence as a single undo-able event. So we save the state of our document the first time that a new slot is about to be changed and ignore subsequent changes (as far as saving the state goes). The last-slot-set value keeps track of what was last set. We will examine what the create-undo method does in just a bit.

```

(defmethod bound-slot-modified ((self loan-doc) slot-name)
  ;; Gets called after any slot in a loan-doc is modified via a KVO connection
  ;; Any other way that the slot is set does NOT result in calling this method
  (when (eq slot-name 'origination-date)
    ;; also set the first-payment date
    (setf (first-payment self) (next-month (origination-date self))))
  (setf (pay-schedule self) nil)
  (compute-new-loan-values self))

```

The bound-slot-modified is called immediately after a slot has been modified via some binding connection. At this point we want to re-compute loan values. If the modified slot was the origination-date, then we also update the first-payment slot as a side-effect.

The following methods are used to implement "undo" for the loan-doc class.

```

(defmethod create-undo ((self loan-doc) undo-name)
  ;; We use the same Objective-C object that is archived to disk when the document
  ;; is saved to represent the state of the document at any given time.
  (let ((st (while-converting
    (std-instance-to-objc self))))
    (set-undo self
      #'(lambda ()
        (set-loan-state self st))
      undo-name)))

```

The create-undo method is called when a slot is about to be modified. Its job is to create some object that represents the current state of the document and then set up an undo function that restores that state when the undo action is selected from the Edit window by the user. This is almost exactly what data conversion functions do when we save a document to disk. Of course in this case we don't want to save anything to disk, we just want to keep create something that can be referenced in a closure that we create to do the undo. It turns out that the conversion functions do exactly this. They turn an instance into an NSDictionary before it is archived off to disk. When it is unarchived that NSDictionary object is reconstituted and the conversion routines use it to initialize an object of the appropriate class. So we will use the same method that the conversion routines do (std-instance-to-objc) to convert our instance into an NSDictionary. The "while-converting" macro is necessary to initialize the overall context for the conversion. The set-undo call is one that we previously discussed. This registers an appropriate undo message such that the specified closure will be invoked when the user chooses to undo something.

```

(defmethod set-loan-state ((self loan-doc) st)
  ;; The current state of the loan is captured in the same type of object
  ;; that is archived to disk when the document is saved. We use that
  ;; object to initialize the loan when its state is restored.
  (create-undo self "set loan state")
  (setf (last-slot-set self) nil)
  (while-unconverting
    (objc-to-std-instance st self))
  (setf (pay-schedule self) nil)
  (compute-new-loan-values self))

```

The set-loan-state method is called when the undo item is selected from the edit menu in our user interface. Note that the first thing it does is set up its own undo. The Objective-C methods are fairly clever and they know that when an undo message is registered during the course of handling an undo message, then the registered undo action is actually a "redo" and gets put onto that stack. That's how "redo" options show up in an edit menu. To set the loan's state this method uses the objc-to-std-instance method to convert the NSDictionary object that was previously saved into an appropriate set

of slot values. That method can either instantiate a new instance of the saved class or use the re-converted values to set the slots for an existing object. We use the second option here to have it automatically set the slot-values for the "self" instance. Then we recompute the loan values in much the same way that we did when we opened a new document.

The rest of the methods in the `loan-app-document.lisp` file are specific computational methods for loans and as described previously are discussed in Project 6 of the *InterfaceBuilderWithCCLTutorial* document.

Next we will look at the `loan-win-ctrl.lisp` source file. This file implements controller functionality for our application. It knows about the design of the interface as well as the methods available for setting loan parameters.

```
(defclass loan-win-controller (ns:ns-window-controller)
  ((loan :accessor loan :initform nil)
   (orig-date-text :foreign-type :id :accessor orig-date-text)
   (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)
   (field-control :foreign-type :id :accessor field-control))
  (:metaclass ns:+ns-object))
```

The `loan-win-controller` class is a subclass of `NSWindowController`. It contains a pointer to the loan object that the window represents and pointers to many of the user-interface fields that will be created.

```
(objc:defmethod (#/awakeFromNib :void)
  ((self loan-win-controller))
  ;; the #/document for a NSWindowController is set when the controller is
  ;; added to the document via a #/addWindowController: call, which is done
  ;; by a lisp-document method. So we can just go get it here and put it into
  ;; a lisp slot which makes it a bit easier to use.
  (setf (loan self) (#/document self))
  (setf (root (field-control self)) (loan self))
  (#/setEnabled: (loan-text self) #$NO))
```

The `awakeFromNib:` method sets the `loan` slot value to be a duplicate of its own Objective-C slot `#/document`. That is merely a little convenience that lets us use a `with-slots` construct and could be eliminated in favor of always calling the `#/document` method. The main thing that this method does is set the root value of the lisp-controller object to be the loan document. The lisp-controller object mediates all bindings between interface objects and Lisp slots. We'll see when we get to the point of defining our window in IB, how we create and configure that lisp-controller object. Setting the root for the lisp-controller simply tells it what object to start with when following binding paths. For this application all the bindings will also end at that object, but in general that need not be the case.

```
(objc:defmethod (#/buttonPushed: :void)
  ((self loan-win-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))))))
```

The only other method we need in our window controller class handles the choice of independent variable that was discussed earlier. It enables any previously disabled fields and disables the appropriate fields for the new compute mode.

A `loan-win-controller` will be the File's Owner object for the window that we will define. So before we leave Lisp to go design the window, we will export the description of our class in a form that IB can import. From the Dev menu select "Create .h For Interface Builder" and select "LOAN-WIN-CONTROLLER" from the pulldown. Save as whatever name you like. I called my lwc (do not type in the .h extension, that will be added automatically for you.) That file will look as follows:

```
@interface LoanWinController : NSWindowController {
    IBOutlet id origDateText;
    IBOutlet id loanText;
    IBOutlet id intText;
    IBOutlet id durText;
```

```

IBOutlet id payText;
IBOutlet id intSlider;
IBOutlet id durSlider;
IBOutlet id paySlider;
IBOutlet id fieldControl;
}
- (IBAction)buttonPushed:(id)sender;
@end

```

Next let's have a look at the actual interface design. To follow along with my existing design you can double-click "LoanWin.nib" or open the Interface Builder application and create your own as we go. In Interface Builder let's first import that definition that we just created. From the File menu select "Read class files ..." and navigate to the .h file that you just created and select it. This will make the definition for the LoanWinController class available for use inside IB. If you now select the LoanWinController class in the Library window Classes pane, you can look at its Outlets and Actions to assure yourself that we have indeed made IB aware of our class definition.

Much of the following will look familiar to anyone who went through Project 6 from the InterfaceBuilderWithCCLTutorial document. But there are some subtle differences as well. I used a newer version of Interface Builder for this project, so that will also make things look a bit different. If you are still using the older version of IB, you may want to refer to the other project at times, but the binding and connectivity described here is what you want to achieve.

Start by creating a new Cocoa window project in IB. Make your interface window look like the following:

Figure 27: 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. 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 28 below.

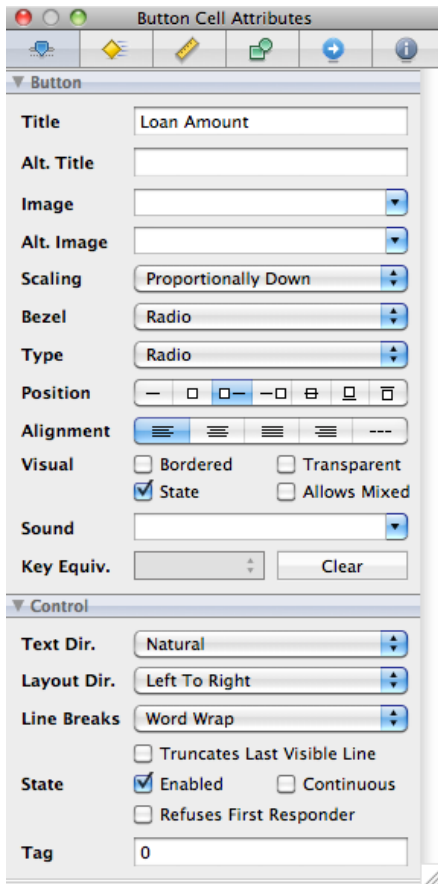


Figure 28: Radio Button Attributes

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 under the Objects pane of 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 select the M/d/yy preset (or whichever one you prefer) as shown in Figure 29 below.

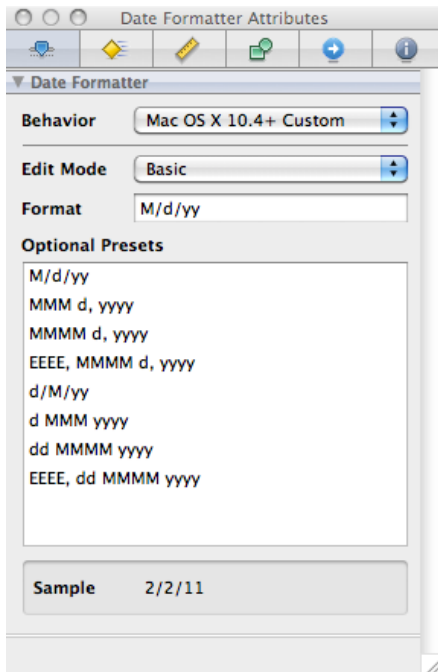


Figure 29: Modifying the Date Formatter

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

Date formatters are required for another reason. When we later establish a binding between these text fields and a Lisp instance slot, the presence of this formatter will cause the Lisp functions that manage data conversions for bindings to assume that the Lisp value is a *universal-time* value as normally encoded in Common Lisp (i.e. as if created by `get-universal-time` or similar functions).

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 30.

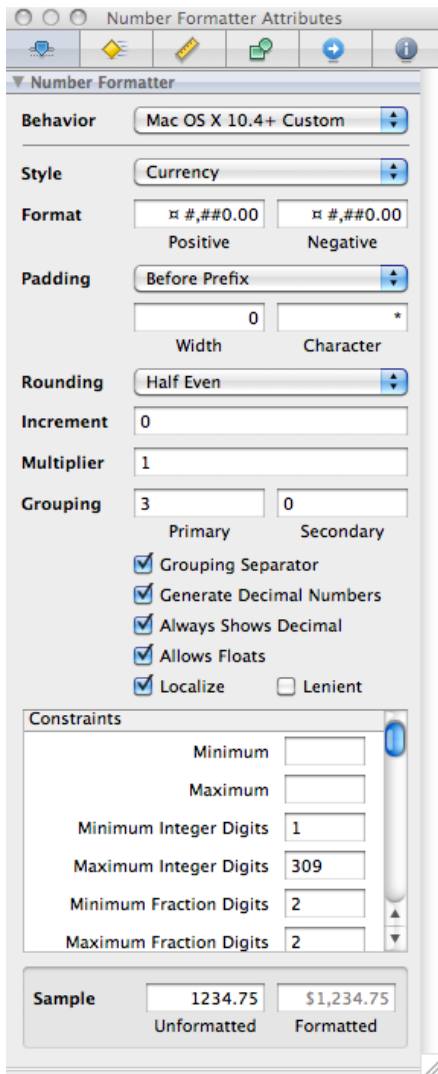


Figure 30: Modifying the number formatter

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 won't examine them in detail here. We refer the interested reader to the detailed description within the description of Project 6 in *InterfaceBuilderWithCCLTutorial*. But for now we have to assure that the dollar values that we enter and display are accurately converted between the user interface and Lisp by causing the formatter to generate `NSDecimalNumber` objects. So make sure that this box is checked. This is the hint that the Lisp binding routines will use when managing the conversion of bound values.

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 would 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. But for this application we will change this to 1 and the Lisp code

will divide by 100 to get a real interest rate. This will give us a bit better precision when using floating point values. 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 Fraction Digits" field in the Constraints field of the Number Formatter attributes view. When you get done this should look as shown in Figure 31.

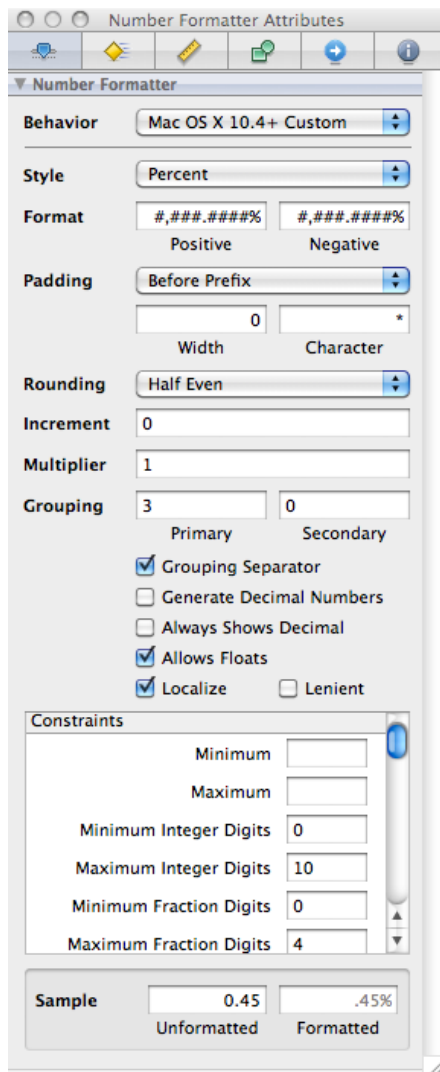


Figure 31: Annual Interest Rate Formatter

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.

We must also configure the sliders appropriately. Start by clicking on the "Annual Interest Rate %" slider. Using the attributes pane of the Inspector window, set its minimum and maximum values to 0.0 and 50.0 respectively. If somebody wants a loan and is willing to pay a higher annual percentage than that, please send me a private email. In the control section of that window make sure that the continuous checkbox is selected for the State attribute. This assures that values are continuously sent to our application as the slider is moved. Without this, the value would not change until the slider was released. If you are using an older version of Interface Builder, you may have to set this on the "Slider Cell" rather than for the Slider itself. To do that click on the slider once to select it and a second time to select the Slider Cell. The inspector window's title will tell you what object has been selected. Then make sure that the "continuous" checkbox is checked for the slider cell.

Similarly configure the minimum and maximum values for the Loan Duration slider to 0 and 500 and make sure it is also set to continuously update its values. Also configure the minimum and maximum values for the Monthly Payment slider to 0.0 and 10000.0 and make sure it is also set to continuously update its values.

The look of our interface is complete and now we have to establish five sorts of links. First we need to create links between our File's Owner object (which as you will recall will be an instance of our `LoanWinController` class) and various interface objects that it will need to manage at runtime. bindings between interface fields and Lisp slots. Second, we need to connect the action of those radio buttons to a particular method in our File's Owner object. Third, we need to link the File's Owner object as the window's delegate. Fourth, we need to link the File's Owner object to a

LispController object. We'll discuss its role a bit later. Finally, we need to establish bindings between interface object values and the values from corresponding Lisp object slots.

Before we can do any of these, we need to set the type of that File's Owner object. Select it in the NIB document window and in the browser window select the Identity pane (farthest right in the browser window). Use the pull-down for the Class field to select LoanWinController.

Let's start with the links from the File's Owner object to various interface objects. As described earlier, we want our user to be able to select which of the four loan variables will be treated as the independent variable and compute its value from those of the other three. 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. We already saw that in the code for the loan-win-controller class. We will establish pointers from the fields of our File's Owner object to those controls that we want to enable and disable. Control-click on the File's Owner object and you will see all of the outlets that were defined in the Lisp code and imported into IB. 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.

Next 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.

Next 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. Note that this outlet was inherited from the NSWindowController class. 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. While we won't actually implement any delegate methods in our subclass, there may be such methods implemented in the NSWindowController class that we inherited from, so we will make the link to assure that everything goes smoothly.

Next, from the Library Window Classes pane find the LispController class. If you can't find it, make sure that you have previously installed the LispController plugin as described in the *LispController Reference* document. This object is needed to mediate all bindings between interface objects and Lisp slots. Drag a LispController instance from the Library window to the document window. I then clicked on the name and called it "Field Controller". Connect the fieldControl outlet of the File's Owner object to this controller just as you connected other outlets.

While we're at it, let's also configure the LispController instance. This is actually pretty easy since we're only using it as an intermediary for bindings. Click on the LispController object named Field Control and in the attributes pane of the inspector window set the Root Type to "Idoc::loan-doc". Also make sure the "Generate Root" check-box is NOT selected. At runtime the LispController will assure that the value of the root that is set is of that type. That's all that is needed to use a LispController as a first pathway for bindings.

When all of these links have been made, you can control-click on the File's Owner object and see something that resembles Figure 32, shown below.

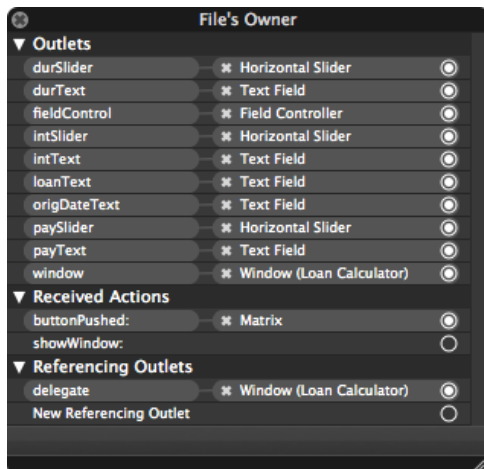


Figure 32: Connections to the File's Owner object

Finally we need to make binding connections between the display elements and fields in our loan-doc object. Recall that in the definition of the loan-doc class we specified a :kvo option for most of the slots. This will allow us to bind interface elements using the string specified with each of those options. If a binding such as this exists, then whenever 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-doc object is now largely oblivious to the interface objects. As long as it is *Key-Value Coding* compliant for all the slots that some interface might want that's all it needs to worry about. And as luck would have it, using a :kvo slot option is all that is necessary to assure such compliance. 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 select "Field Controller" (or whatever you named the LispController object you added to the interface) BEFORE clicking on the "Bind to:" box. If you click the Bind to box first, then the default value for the bound object will be used. That is typically "Shared User Defaults Controller" and IB will immediately add an object of this type to your NIB file. By selecting the bound-to object first, you avoid having to go delete the object that you accidentally added. In the Model Key Path field type in "root.origDate". If you look back at the loan-doc class definition you will see that "origDate" was the specified :kvo string to use for the origination-date slot. As always capitalization must be precise. Using "root" specifies that the first step in the keypath access whatever object is the root object of the LispController. You will recall that the loan-win-ctrl object's #awakeFromNib method sets this value to the loan-doc instance that is represented in the window. Additionally click in the "Continuously Updates Value" checkbox. This makes sure that the value is updated in the bound slot even if you don't leave the Text Field after editing it. This is highly desirable. The alternative results in displayed values sometimes being different than what is in the corresponding slot-value. Setting this attribute is entirely equivalent to checking the "continuous" checkbox for the

State attribute in the attribute inspector window while inspecting the text field. Older versions of Interface Builder only permitted the continuous attribute to be set in the Binding window, which is why I have described it that way here. When you finish this, the value binding for this field should resemble Figure 33 below.

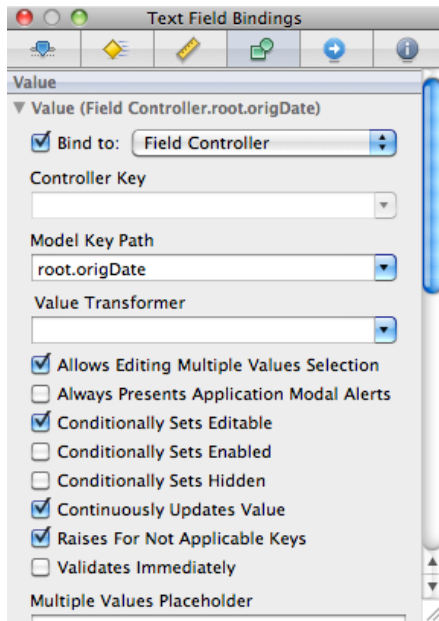


Figure 33: Value binding for Origination Date field

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 "root.firstPayment" slot of the Field Controller (i.e. set the Model Key Path to "root.firstPayment").

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

"Loan Amount"	Text Field to	"root.loanAmount"	slot
"Annual Interest Rate"	Text Field to	"root.interestRate"	slot
"Loan Duration"	Text Field to	"root.loanDuration"	slot
"Monthly Payment"	Text Field to	"root.monthlyPayment"	slot
"Total Interest Paid"	Text Field to	"root.totlInterest"	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. Careful now, we are not binding the object's value field as we did with other controls. Make sure you are looking at the binding fields for the Hidden attribute. Select the Field Controller object as the Check the "Bind to:" box and . In the Model Key Path field enter "root.maxDuration".

When the lisp-value of this condition is t for the loan the converted value returned by the binding will be the Objective-C value \$YES. In this case we want the Text Field to be displayed, but that corresponds to a hidden value of \$NO. So we need to negate the value returned by the binding 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 34 below.

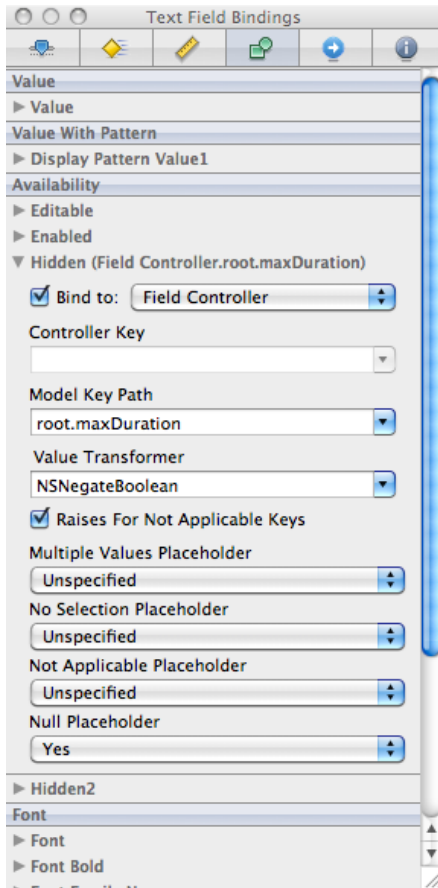


Figure 34: Binding the Hidden attribute for the "Reached max duration ..." text field

Similarly bind the "Reached min payment ..." Text Field to the slot "root.minPay" and the "Reached min duration ..." Text Field to the slot "root.minDuration". If you have not already done so, you may want to position the text fields for min and max duration on top of each other. Since only one of them will be displayed at a time, this won't be a problem at runtime.

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 loan-doc 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. Then open the bindings pane in the inspector window and bind the value of the slider to the "root.interestRate" key path of the Field Controller, just as we did for the corresponding Annual Interest Rate text field. We can't click on a "Continuously Updates Value" checkbox because it doesn't exist, but that doesn't matter because we already selected the continuous attribute for the slider itself when we initialized it. Without that the slider value wouldn't be reflected either in the text box or in the loan-doc 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.

In a similar fashion bind the values for the Loan Duration and Monthly Payment sliders to the same fields that we bound their corresponding text boxes. But DO NOT try to add a formatter to the slider and set it to use NSDecimal numbers as we did for the corresponding text box. 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. When the slider requests the value from the bound slot, the binding conversion routines will determine that SOME object that is bound to the slot is using NSDecimal numbers and return an NSDecimal object that represents the slot value. The good news is that sliders are perfectly happy to accept such objects and interpret them properly. When the slider sends a new value to Lisp, it is sent as an NSNumber object. The conversion routines know that the lisp slot must be consistent with an NSDecimal interpretation and convert any object appropriately. So everybody stays happy.

To double-check that we've linked everything correctly, control-click on the Field Control object. The pop-up should look very much like Figure 35 below.

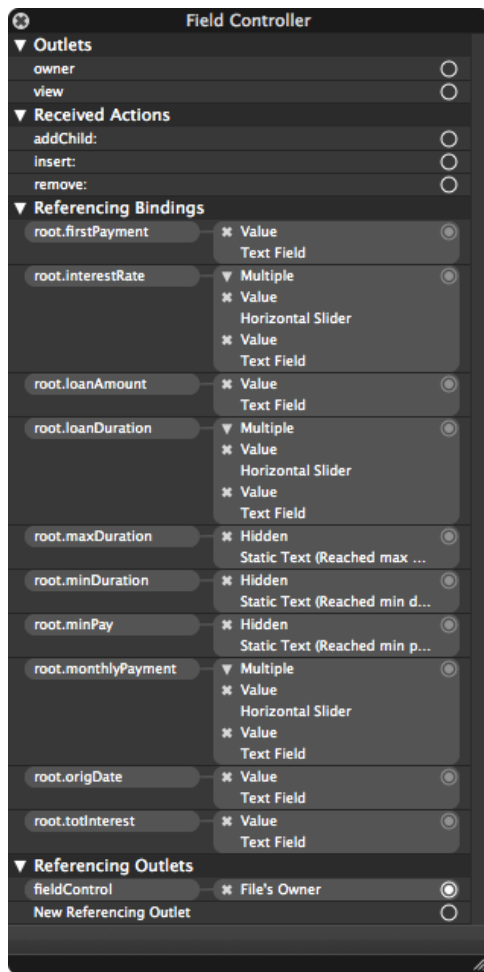


Figure 35 Field Control Links

That completes the configuration of the LoanWin NIB file. Save it and let's return to the CCL IDE to test it out.

Click on the + button for the NIB table and navigate to and select your LoanWin.nib file (or whatever you named yours, although if you named it something other than this you should also edit the window-nib-names method in your source code to match this name). Then select "Load App Under IDE" from the Dev menu. After you do that you can then select "New Loan Calculation" from the file menu and you should see your window pop up. At that point you should test all of the functionality to make sure that everything works as expected. Save a loan document, close it and re-open it, whatever suits your fancy.

The next step in our application development is to create a MainMenu NIB. This process is pretty much identical to what we did for the squiggle application. Rather than duplicating that discussion here, I will just refer you back to the section titled "Adding A Main Menu To Your Application" earlier in this document. For the sake of argument I will assume that you named your main menu NIB file "LoanMainMenu.nib". Click on the + button for the NIB table and select this saved NIB file. Your LAD window should now look like Figure 36 below.

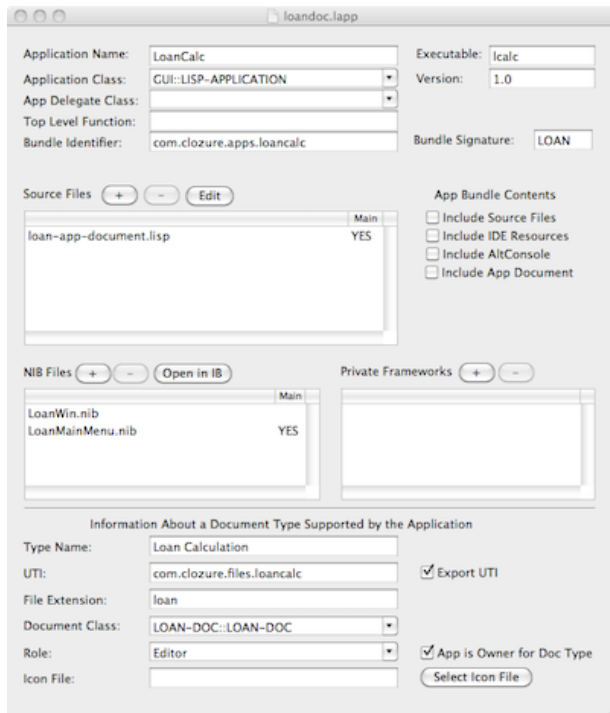


Figure 36: Fully configured LAD window for the Loan application

If you now select "Load App Under IDE" from the Dev menu, the CCL menus in the menu bar will be replaced by those from your new LoanMainMenu.nib file. Select "New" from the File menu and if all goes well it will open up a Loan window for you. As before, you can test anything you want, but presumably at this point your testing would focus on menu item functionality. As we did when testing the squiggle application, you can toggle application and CCL IDE menus off and on using appropriate selections from the Dev Menu (which remains visible no matter what other menus are shown).

The last step, as you might imagine, is to create a stand-alone version of the loan application. All that is required to do this is to install an executable into your application bundle. Do this by making your Loan LAD window the top window and then selecting "Install Executable" from the Dev menu. As before, this may take a bit while everything is loaded into the subordinate Lisp that is run, but after this is complete you can select "Run App Stand-Alone" from the Dev menu to run your new program. Equivalently you can double-click on it in the Finder.

That concludes discussion of the Loan Calculation Application.

Advanced Topics

Creating a custom application class

In most cases, you should not have a need to create a custom application class. But if you are reading this section perhaps you have such a need. The LAD window lets you specify the class that should be used. The only advice I'll give is that if you are planning to include IDE resources, then your class should inherit from `gui::lisp-application`. Otherwise you're on your own.

Creating a custom application delegate class

This is a more common thing to do. Such a class is a good place to implement any methods that implement the action messages sent from custom menu items that are not document-specific. For example, implementing application preferences or search windows or whatever. You can also do these things in a custom application class as the CCL IDE does, but using the delegate class is a bit more common and avoids the need to create a separate class for both the application and its delegate.

Supporting additional document types in your application

Each document type that an application supports must be described in the Info.plist file in the application's bundle. The LAD window currently only supports the definition of a single document type. Arguably the section for document type definition at the bottom of the LAD window should be replaced by a table of document types. When one of those is selected for editing, a separate document definition window should pop up to let you do that. If someone really wants to make such a mod, I'd be happy to see that.

In the meantime it's not too onerous to directly edit the Info.plist file and make additions yourself for additional documents. I suggest that you add one document in the LAD window and then select "Edit Info.plist" from the Dev window. It will open in Apple's Property List Editor application and you can use the existing fields that are defined for your single document to add additional documents. Apple's "Information Property List Key Reference" is a good source of information about what goes into each of the values in an Info.plist.

Custom top-level functions

For the most part I think that the default top-level function provided by Clozure for Cocoa applications works pretty well. Working through what happens when you designate a different top-level function to the save-application function is an interesting exercise. I did this for myself and captured what I learned in ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Documentation/toplevel-function.rtf. That may help you if you believe you have a

need to create your own. Within the `lisp-app-doc.lisp` source file I also defined a somewhat simpler version of Clozure's toplevel function called `simple-toplevel-function` that can use as a starting point for developing your own. WARNING: this has not been tested and I make no guarantee about how well this will work.

Adding executables for other platforms

I have not tried to do this, so this should be taken as a suggestion. If you have a need to do this, let me know what you found out and I'll include it here for others.

Assuming that you want to add an executable for a different platform to your bundle and that you have such a platform available with both the CCL implementation and requisite native implementation tools available, you may be able to move your application bundle to that platform and simply install a new executable for that platform. By checking the "Include Source Files" and "Include App Document" buttons in a LAD window, everything else needed to re-create your application will be put into the bundle. You should then be able to move the entire bundle to the other platform. You will have to then extract the `.lapp` document from that bundle (that's the saved LAD) and open it in the native CCL. You can then select "Use Bundle" from the Dev menu to direct it at the application bundle. You will then either have to move the source files from the bundle to some convenient location or use them where they are, but in either case you would need to change the source table to point to the correct location. Then installing an executable should hopefully do the right thing.