CCL Cocoa Developer Tools Tutorial

Version 2.1 July 2017

Paul L. Krueger, Ph.D.

Overview

What I'm trying to achieve

The tools described in this document were created to expedite the process of creating stand-alone *Document-Based* Lisp/Cocoa applications. It can also be used to create stand-alone applications without documents. 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, Swift, 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 summarize any significant part 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. In addition, the tools now require that you be running at least version 1.9 of the CCL IDE and OSX 2.7 (Lion) or higher. 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.

This document is NOT an in-depth tutorial about how to design and implement windows procedurally in CCL. To learn that (which is really a prerequisite for building document-based applications using this document) you should first work through the

UserInterfaceTutorial document. This will tell you pretty much everything you will need to know to implement windows procedurally and have them usefully interact with Lisp data. That includes the use of custom view and controller classes that were created to make Lisp programming easier as well as how to use the Lisp constraint interface, do data conversions, bind values between view attributes and Lisp slots and more.

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. Running an application under the Lisp CCL IDE provides a very rich debugging environment. Once the developer gets to the point where the few limitations actually matter, it is possible to create a stand-alone executable for testing purposes.

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 instances of your main document class. For many 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. The Lisp tools provide a way to invoke Apple's Xcode 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. Typical Apple application developers would use Interface Builder functionality within Xcode to design interfaces. With this version of the tools, I have instead adopted the strategy described in the *UsertInterfaceTutorial* document which constructs all interfaces procedurally. This makes it unnecessary to use any part of Xcode for simple Lisp application development. You may still be required to use

Xcode if your application is very complex or if you need to prepare it for inclusion in Apple's App Store (although I hope to add the ability to do this completely within Lisp in the near future). *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 easier. For the most part, all View objects have been provided by Apple as part of Cocoa. I have adde a number of additional view classes that make it somewhat easier to use what Apple provides. In some ways, those classes are similar to the sorts of pseudo-views that Interface Builder provides for Objective-C users. It is possible for a developer to create additional 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 one or more custom view classes 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. I have created a single coerce-obj method that takes two arguments: an object to be converted and a type to which it should be converted. This is intentionally similar to the Lisp coerce function. Sometimes you may not exactly know what the best type is for a particular conversion. You can always do something like:

```
      (coerce-obj <lisp object> 'ns:ns-object)
```
or
```
       (coerce-obj <ns:ns-object> t)
```
which will convert a Lisp object to an Objective-C object or vice versa (respectively).

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. See the *LispController Reference* for a more detailed discussion of how this works and the *UserInterfaceTutorial* for example code that demonstrates how to include lisp-controller objects in interface designs.

I have also extended Apple's binding protocol to include Lisp slots when proper slot options are specified as part of class definitions. See the *Lisp-KVO Reference and Tutorial* for a complete description of how that works and the *UserInterfaceTutorial* for example code that demonstrates how to do slot bindings.

What's not here yet

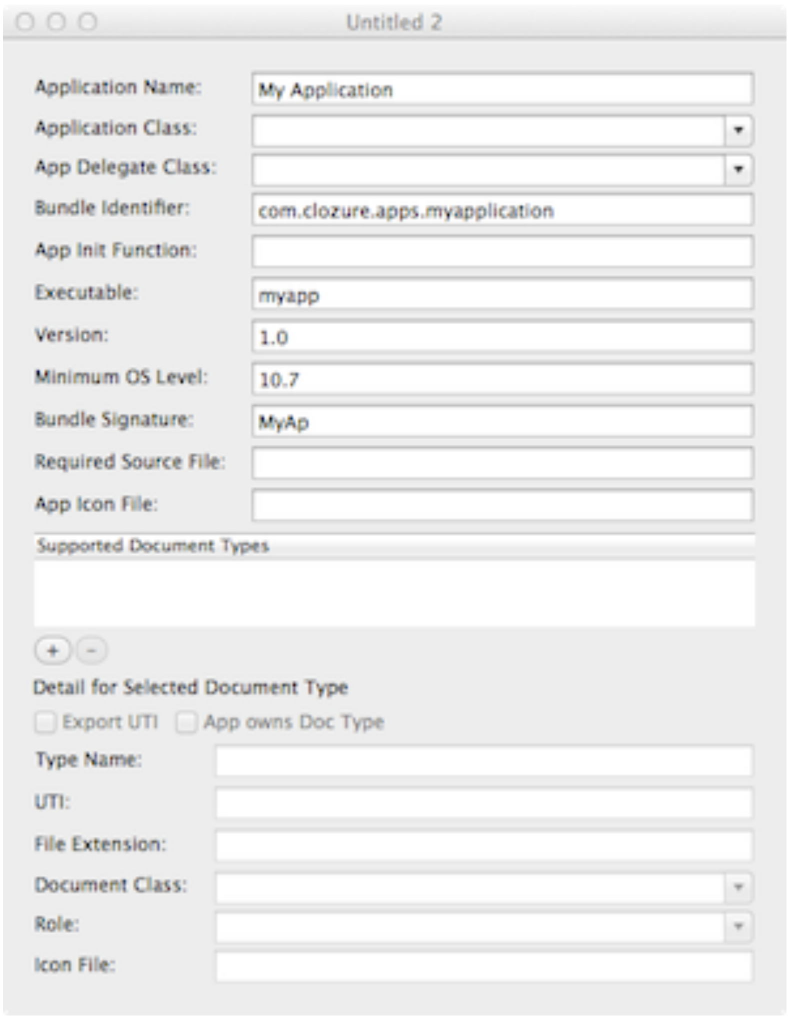
The tools described here get you all the way to the creation of a stand-alone app that you can run on your own computer or one that can run the same code. It does not support codesigning, sandboxing, or other procedures that might be necessary to get an application ready for the App Store. If you want to do those things, you'll have to do things manually. At some time in the near future I expect those will also be supported by tools.

Next ...

The remainder of this document will describe what tools are available and provide two examples of how to use them. It will NOT describe the internal implementation of the tools themselves. The tools implementation is itself 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) 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 (primarily contained in the "...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Cocoa Dev/" folder) 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 below. For convenience, all windows (other than panels) shown in this document are at 60% of actual size.


Untitled 2

Application Name:	My Application
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.myapplication
App Init Function:	
Executable:	myapp
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	MyAp
Required Source File:	
App Icon File:	

Supported Document Types

Detail for Selected Document Type

Export UTI App owns Doc Type

Type Name:	
UTI:	
File Extension:	
Document Class:	
Role:	
Icon File:	

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 (with a ".app" extension) have a specified sub-directory structure and some required files including the actual executable 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. The following should work out of the box:

```
Welcome to Clozure Common Lisp Version 1.11-store-r16714
(DarwinX8664)!

CCL is developed and maintained by Clozure Associates. For more
information
about CCL visit http://ccl.clozure.com.  To enquire about
Clozure's Common Lisp
consulting services e-mail info@clozure.com or visit http://
www.clozure.com.

? (require :install-app-tools)
:INSTALL-APP-TOOLS
("INTERFACE-PACKAGES" "IU-CLASSES" "NS-STRING-UTILS" "LC-
CLASSES" "NSLOG-UTILS" "UNDO" "KVO-SLOT" "SELECTOR-UTILS"
"ASSOC-ARRAY" "BINDING-UTILS" "TAGGED-DATES" "DATE" "ALERT"
"DECIMAL" "ATTRIBUTED-STRINGS" "NS-OBJECT-UTILS" "COERCE-OBJ"
"LIST-UTILS" "FILE-DIRECTORY-UTILS" "OBJC-INITIALIZE" "CCL-
ADDITIONS-FOR-COCOA-TOOLS" "BUILDER-UTILITIES" "FILE-MONITOR"
"OBJC-METHOD-INFO" "LV-CLASSES" "DEV-TOOLS" "LISP-CONTROLLER"
"WORDS" "OPEN-PANEL" "SAVE-PANEL" "CLASS-CONVERT" "LISP-APP-
DELEGATE" "INTERACTIVE-APP" "NIB" "MENU-UTILS" "LISP-APP-DOC-PR-
VIEW" "LISP-APP-PR-VIEW" "LISP-BUNDLE" "LISP-WINDOW-CONTROLLER"
"WINDOW-CONTROLLER" "QUICK-WINDOW" "LISP-DOCUMENT" "DOC-
CONTROLLER-HASH" "LISP-DOC-CONTROLLER" "VIEW-UTILS" "CONSTRAINT-
LAYOUT" "TEXT-VIEWS" "SCROLL-VIEW" "WINDOW-UTILS" "ORGANIZED-
BOX-VIEW" "BUTTON" "LISP-APP-WIN-CONTROLLER" "CUSTOM-APP-INIT"
"UTILITY" "THREAD-SAFE-QUEUE" "LISP-APP-DOC" "INSTALL-APP-
TOOLS")
?
```

If desired, you can make this happen every time you start up the CCL IDE by adding:

```
      (require :install-app-tools)
```

to your ccl-ide-init.lisp file in your home directory. This loads quite a bit of code, so it can take a small additional amount of time to start up. If you get to the point where you do this often enough, it may be of value to use CCL's save-application function to create a version of the IDE with the bulk of this code already loaded. To do that you should start up the normal CCL IDE and then

```
      (require :lisp-app-doc)
```

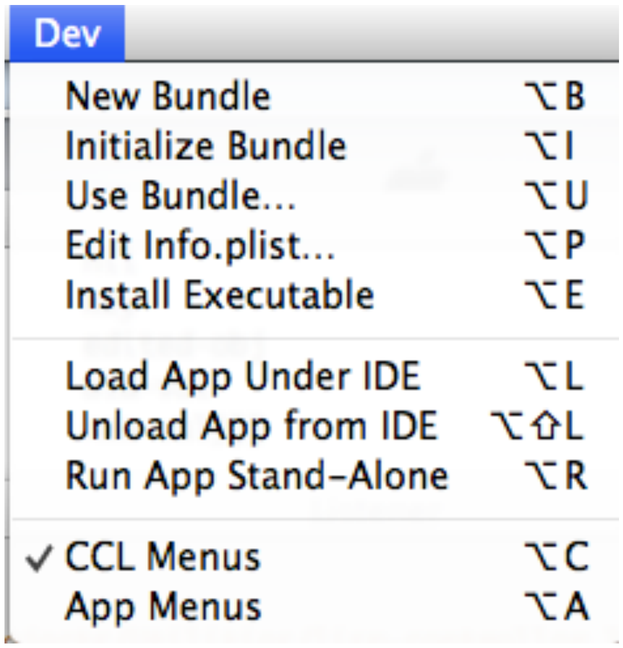
This will load virtually everything needed to install the tools. Then use save-application to save this. Then the use of (require :install-app-tools) will happen very quickly whether you do it manually after the IDE opens or execute it as part of a ccl-ide-init.lisp file.

Some of you may be wondering why you can't just require :install-app-tools and then save the application. The reason is that part of the install process adds a new menu to the main menu bar. This would not be preserved in an image created by save-application, so it is necessary to execute the install code to create that menu each time it is needed.

Requiring :install-app-tools 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 the figure above. In addition, installing the tools results in the addition of a new menu: the Dev menu shown below.



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. All applications are contained in application bundles. The tools automatically create such a bundle for you whenever you select "New 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 LAD 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.

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 Xcode 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, executing the app init function, 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 in a Finder window.

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 within your application initialization method, you can select this to toggle the inclusion or exclusion of 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 most recently loaded app. If a Main Menu is created and installed by the app initialization method (typically 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 "Clozure <something>" regardless of the label provided when you created the menu. Note that although it is possible to run multiple applications under the IDE simultaneously, only the main menu of the most recently run app can be swapped in and out via this menu choice.

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 Squiggle Example

"Squiggles" is a sample application provided by Apple. It is a simple graphical application. As provided by Apple, 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 transform it into a stand-alone application that can save, 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).

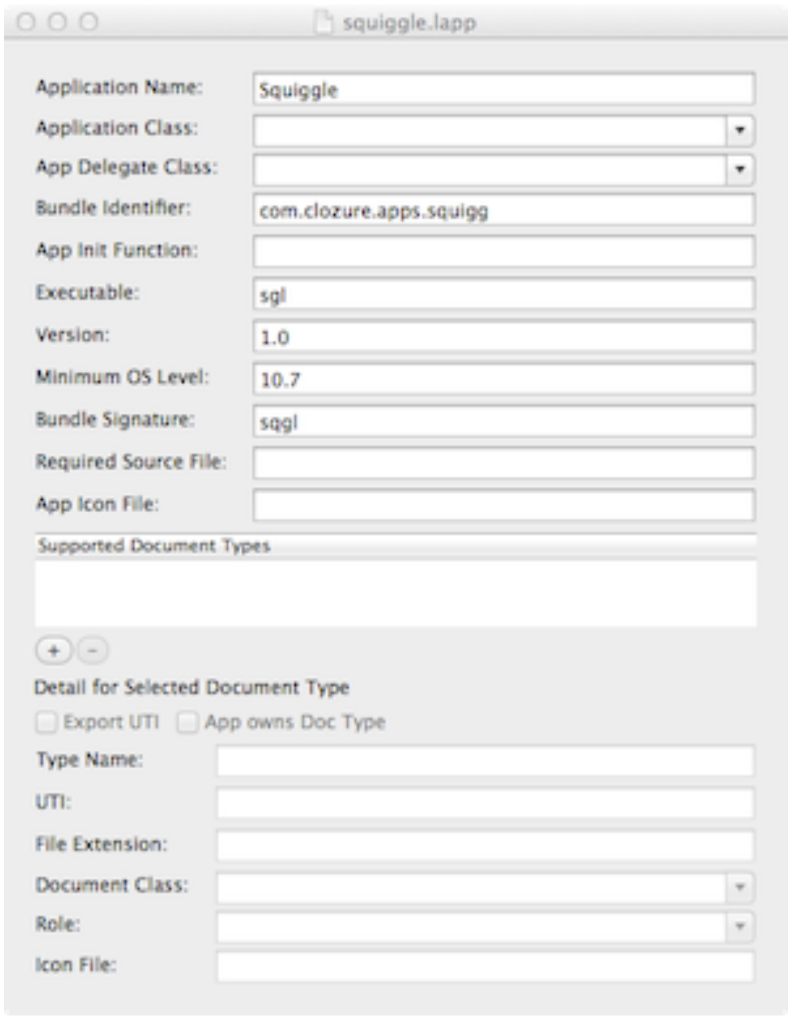
The Apple Objective-C project can be found at http://developer.apple.com/library/mac/#samplecode/Squiggles/Introduction/Intro.html. The interested reader can download it and examine it in Xcode, but that is not necessary to follow along with the Lisp example that is provided here. I selected this example 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.

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 and save this document to disk. Give it a name that you will remember. Remember, at this point we are only saving the equivalent of an Xcode project file that describes our application, not the application itself. As you can see, I called mine squiggle.lapp. I saved it in the same Squiggle directory where the source files reside, but the location doesn't really matter as long as you can find it again.


squiggle.lapp

Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	
App Icon File:	

Supported Document Types

Detail for Selected Document Type

Export UTI App owns Doc Type

Type Name:	
UTI:	
File Extension:	
Document Class:	
Role:	
Icon File:	

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.

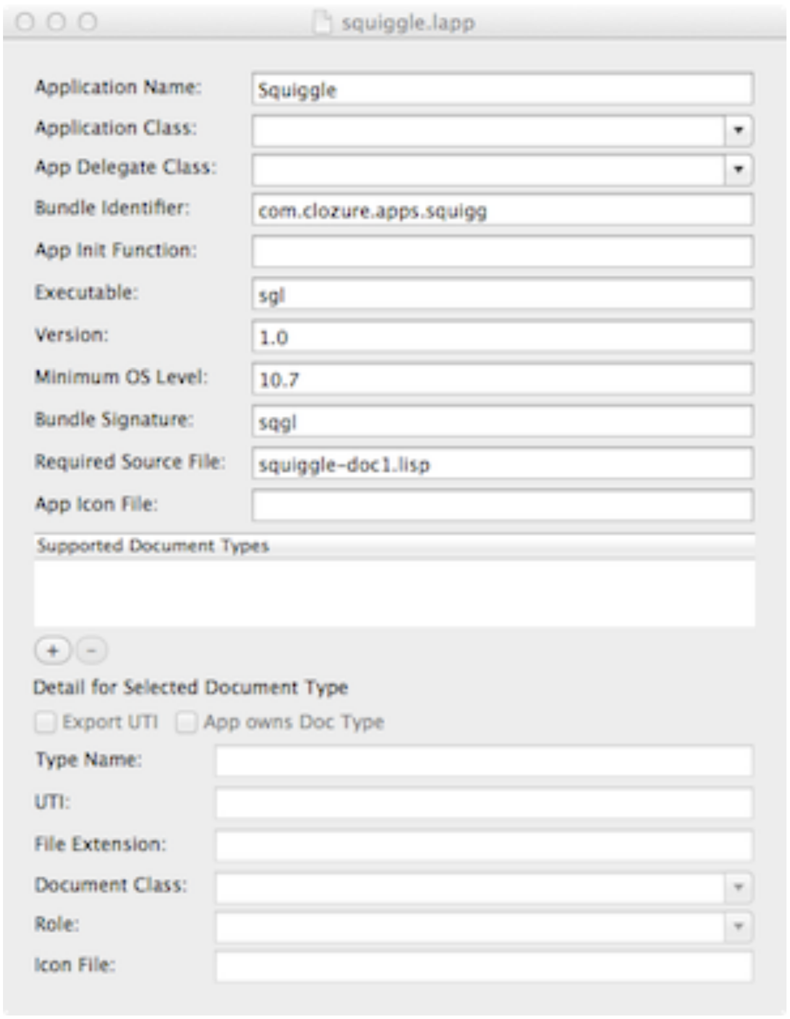
Loading and testing source files under the IDE

I manage all of my own source code so that for any given application it is only necessary to load a single file, which manages the load of everything else needed (primarily by the use of lisp *require* forms). The Lisp tools described here make the assumption that you will do the same. You need only provide the name of a single required lisp source file. The best way to do this is to click in the *Required Source File* box and press the return key with nothing entered. This will pop up a standard open dialog that allows you to navigate to the source file you wish to use. Don't be concerned that only the last part of the fie path you selected is displayed, the entire path is recorded in the LAD. You will want to specify that source file so that the tools assure that your source is properly loaded before running your application either within the IDE or as a stand-alone application.

You may, of course, enter a complete path name in the field. If no file currently exists at the path specified you will receive a pop-warning, but the name will be accepted (just in case you are entering the name of a file that you expect to create presently). However, if you subsequently do something that requires the source to be loaded, you will receive a Lisp runtime error if the file cannot be found.

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.

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 specify a required source file in the LAD window. Assuming that you are following along with the Squiggle example, click in the *Required Source File* field, hit the return key with nothing entered in the field and navigate to the source file …CocoaInterface/Example Projects/Squiggle/squiggle-doc1.lisp and select it. Then save the squiggle LAD. The .lapp extension is used for saved Lisp application documents and is added automatically. Your LAD window should now look like the figure below:


squiggle.lapp

Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	squiggle-doc1.lisp
App Icon File:	

Supported Document Types

Detail for Selected Document Type

Export UTI App owns Doc Type

Type Name:	
UTI:	
File Extension:	
Document Class:	
Role:	
Icon File:	

Saved LAD window with the required source file identified

The squiggle-doc1.lisp source file is just a restricted subset of the full application that is defined in squiggle-doc.lisp. We'll start with this in order to demonstrate some of the default things that the tools will do for you before you everything defined for the complete application.

When you select "Load App Under IDE" from the Dev menu (do that now if you are following along) you will first be asked where you want to save the Application bundle. Pick any convenient directory. After that, the source file that you specified will be required. If it was previously loaded by you manually, then nothing more is done and you will get a warning that the attempt to load the app accomplished nothing. If it was not previously provided, then it is loaded.

Once your source has been loaded, re-loading the app will NOT cause the source to be reloaded, even if it has been modified. This is because generally speaking any code that you will want to load will have defined one or more Objective-C classes and if an attempt is made to redefine an Objective-C class after modifying its :foreign slots, you will get a runtime error. It used to be the case that ANY redefinition of an Objective-C class caused such an error, but this is no longer true. If only Lisp slots are modified, then it is perfectly acceptable to redefine an Objective-C subclass. But since the tools have no idea what was modified in your source code, I have elected to never automatically re-load source to avoid any problems. You may, of course, reload your source manually if desired and you are positive that no Objective-C slots were modified. If you need to redefine an Objective-C class after modifying an Objective-C slot, then you must quit the IDE and restart it before reloading the source files. Generally speaking, you can dynamically redefine Objective-C methods without problems. I will add one little caveat to the previous sentence 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.

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. If you like, you can verify that the load occurred by typing (find-class 'sq::squiggle-doc nil) in the listener which should return something that looks like:

```
     #<OBJC:OBJC-CLASS SQUIGGLE::SQUIGGLE-DOC (#x23E3E820)>.
```

Since we have not yet specified a document or a window or any menu items for our application, loading the application at this stage 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. As we'll see later, it may be entirely appropriate to load the app multiple times to test modifications that you make to your application's menu or windows or bundle. The developer tools permit you to do this without reloading the source files.

Managing Documents Within 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*.

First I'll discuss documents in general and the special lisp-document and lisp-window-controller classes that you will need to use. Next I'll examine the document class for our squiggle example. Finally I will show how to add documents to your application by entering values in the LAD window.

Documents in general and the lisp-document and lisp-window-controller classes

A document is an in-memory repository for data pertaining to some particular object of interest. Documents can typically be stored on disk and opened by an application when that data is needed. In Objective-C the ns:ns-document class contains a wide variety of useful methods. Developers will create custom subclasses with additional methods to implement any other functionality that is needed to maintain the data for a specific type of document. Among other things, ns:ns-document classes can do the following:

- Save whatever data is necessary to reconstitute the document in a form suitable for external storage
- Reconstruct themselves from a previously saved external representation
- Create whatever window controllers are needed to visually represent (and potentially edit) the document's data
- Manage the destruction of window controllers as appropriate
- Print their data in some suitable manner
- Keep track of changes to document data
- Maintain an "undo" capability.

Developers of custom ns-document subclasses must interact appropriately with this functionality to assure that their documents are handled correctly. Much of what is done automatically by these Cocoa functions relies on the assumption that document data is represented using Objective-C data structures. Without the tools described in this document, a Lisp developer of a custom ns-document subclass must choose between using similar Objective-C data structures (in which case you might as well code directly in Objective-C) or translating data back and forth as needed.

To make life a bit simpler for Lisp developers I created the lisp-document class (...ccl/contrib/cocoa-ide/krueger/Interface Projects/Utilities/lisp-document.lisp), the lisp-window-controller class (...ccl/contrib/cocoa-ide/krueger/Interface Projects/Utilities/window-controller.lisp), and a large number of data conversion functions which are neatly wrapped up in a single "coerce-obj" method. I hope you will find that together these make it entirely reasonable to create new document classes where all the data is represented in typical Lisp fashion using entirely ordinary Lisp data.

Although it isn't 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 with lisp data slots to disk and reloading it
- A lisp-friendly interface to the undo manager
- 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 multiple classes if the document requires multiple windows) or use a default lisp-window-controller

- Simple interfaces for printing either text or graphical depictions of your document

The features supported by the lisp-window-controller-class include:

- The ability to specify a process for creating a window without the requirement of a NIB (as default window-controllers do).
- Automatic memory management for window-specific objects created.

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

Creating a lisp-document subclass for the squiggle app

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/cocoa-ide/krueger/InterfaceProjects/Utilities/lisp-document.lisp
You will want to add something like: (require :lisp-document) to your source file which will in turn require all non-view-related source that your application might need. For this example (from ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Squiggle/squiggle-doc1.lisp) this looks like:

```
(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :lisp-document)
  (require :squiggle-classes)
  (require :squiggle))
```

Thanks to the advice of one lisp-programming colleague, I've come to see the wisdom of defining all classes used for a project in a single file. This avoids circular requirements that might otherwise arise. Since I also tend to use a custom package for each project, this turns out to be a useful place to define the project's package as well. In the case of the squiggle example, I have created the squiggle-classes.lisp file to contain this information. Let's first look at that file. In that file you will find the following package definition:

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

In addition to the standard :ccl and :common-lisp packages, our squiggle package will use the interface-utilities (:iu) and lisp-views (:lv) packages. The former contains almost all non-view functionality that I provide to developers and the latter contains the custom view subclasses and related functionality.

Next we make sure that the code which follows is within the :squiggle package:

```
(in-package :sq)
```

And then define the class for our squiggle document:

```
(defclass squiggle-doc (lisp-document)
  ((squiggle-list :accessor squiggle-list
                  :initform nil)
   (rotations :accessor rotations
              :initform 1
              :kvo "rotations"
              :undo "set rotations")
   (squiggle-view :accessor squiggle-view
                  :initform nil)
   (squiggle-win :accessor squiggle-win
                 :initform nil)
   (back-color-well :accessor back-color-well
                    :initform nil)
   (back-color :accessor back-color
               :initform (#/blueColor ns:ns-color)
               :kvo "backColor"
               :undo "set background color")
   (rotation-slider :accessor rotation-slider
                    :initform nil)
   (window-frame :accessor window-frame
                 :initform (list 200 200 400 400))
   (notif-handler :accessor notif-handler
                  :initform nil))
  (:metaclass ns:+ns-object))
```

This example uses a window with a few simple views that require little management once the window is constructed. Therefore there was no need to create a custom lisp-window-controller subclass. What little management IS required will be done by the squiggle-doc class itself. That does NOT mean that no window-controller is used. By default, a lisp-document will use the lisp-window-controller class for any window controllers that are needed. As a consequence of not having a custom window controller class, there are a few slots in the squiggle-doc class (specifically squiggle-view, squiggle-win, back-color-well, rotation-slider) that might ordinarily end up in a custom window-controller 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 rather than conflating document and window-controller functionality as is done in this example.

Note that even though those slots will point to Objective-C objects, it is not necessary that the slots containing those values BE Objective-C slots. Normal Lisp slots are perfectly happy to contain pointers to Objective-C objects. This makes it unnecessary to include any slots with :foreign-type declarations in our document class. The only time that :foreign-type slots are required is when it becomes necessary for an Objective-C method to directly access a slot in one of our classes. Because I have also extended the Objective-C binding protocol to work with special lisp slots that have a :kvo declaration, it is seldom necessary to use :foreign-type slots.

As discussed earlier, the squiggle example is derived from an Apple sample application called squiggles. In addition to squiggle-doc class, it also uses two additional classes: squigg-view and squiggle, defined as follows:

```
(defclass squigg-view (ns:ns-view)
  ((document :accessor document
             :initarg :document))
  (:default-initargs
    :document nil)
  (:metaclass ns:+ns-object))

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

A squigg-view instance represents the view object that is displayed in the window. As you can see, there is essentially no new data associated with this view. It is mostly a place to attach the functionality needed to draw our squiggle-document. When thinking about the *model/view/controller* paradigm, a squigg-view falls squarely into the *view* category. So when you look at squigg-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.

A squiggle instance is an object that represents an arbitrary single path drawn by a user by clicking and dragging in the squigg-view. That will make more sense to you when you run the application and see what it does. Cocoa makes it quite easy to capture such paths into a single object (an ns-bezier-path) that we will talk about later. There is no Lisp counterpart for this object, so we'll simply use what Cocoa provides. Similarly, we will use Cocoa ns-color objects to hold colors. You might wonder why the squigg-view does not own the squiggle objects that it displays. This is entirely consistent with the idea that the document's data belongs to the document, not to the view that displays it. As you will also see, keeping the document's data together in the document class will make it very easy to archive the document to disk.

Opening and Saving Documents

To support opening and saving documents, a Cocoa document class must normally implement the following two methods:

- #/readFromData:ofType:error:
- #/dataOfType:error:

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 is done by the lisp-document's version of these methods.

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. The reason for excluding :foreign-type slots is simply that they often contain pointers to things that can be reconstructed when the document is loaded from disk. So it is just a useful heuristic, not an absolute restriction. The default set of slots to be saved 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 class:

```
(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. As noted previously, that is also just fine. These will be archived along with 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 (with appropriate translation of the sequence elements also done). When that NSArray is converted back to Lisp how do the tools decide what sort of conversion is needed? In fact, the archiving process defined by lisp-document methods 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, or eql in the case of macptrs) that will also be true upon restoration of the objects. Circular references between objects are also permitted and handled appropriately.

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 ns:ns-document). 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 <something>". The methods called by an undo closure should typically be "undoable" themselves. That is, they should make their own call to "set-undo". The undo manager recognizes when such a call is made while in the process of undoing and puts the undo method on the redo stack rather than the undo stack. When in the process of "undoing" Apple's undo-manager will ignore the undo-string argument and the underlying Objective-C method will take the action name for the redo menu item from the undo menu item. The set-undo macro will work with any target that is a subclass of ns:ns-document.

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 'setf <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 ns:ns-document. 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. Such slots must always have some form of the :kvo slot-option defined. You can see several examples of that in the *UserInterfaceTutorial* document and we'll see how that works for this project a bit later. For now just understand that it is possible for the developer to specify via a binding how a change to some attribute of a view object is automatically reflected as a corresponding change to the value of a lisp slot (and vice versa). To add an automatic undo capability for such changes, any :kvo slot can also have a slot option of the form:

```
        :undo <undo-string> where
```

<undo-string> is defined as previously.

If such a slot is defined for a subclass of ns:ns-document, then any change to the slot's value as a result of a binding between it and a view object will automatically register an appropriate undo action with the document's undo-manager. 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 may be the case (and IS the case for our squiggle project) that a document may contain references to other objects (squiggle objects for this project). Since squiggle objects are not ns-document objects, they do not have their own undo manager automatically defined. What we would like to do is register changes to the slot values in squiggle objects with the document's undo manager. The method undo-target can be defined for instances of any class to specify which object owns the undo-manager to be used. So as you will see for this project, we will define undo-target for squiggle instances that will return the squiggle-document that owns the squiggle. That will result in registering undo actions for squiggle class slots that have both :kvo and :undo options with the squiggle-document's undo manager.

It is possible to use any combination 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 changes that we want to be undo-able: add-squiggle, remove-squiggle, set-rotations, and set-back-color. The last two can be undone simply by changing their slot values back to their previous value and updating the view to reflect the changes. To effect the change it was only necessary to define the back-color and rotations slots in the squiggle-document with appropriate :kvo and :undo options as shown previously in the class definition. You may want to go back and look at that definition. The add-squggle and remove-squiggle implement slightly more complex state changes because they require manipulating a list that is contained in a slot and those changes are made as the result of more complex user actions. Therefore, we implement methods for those actions and incorporate set-undo calls as shown below:

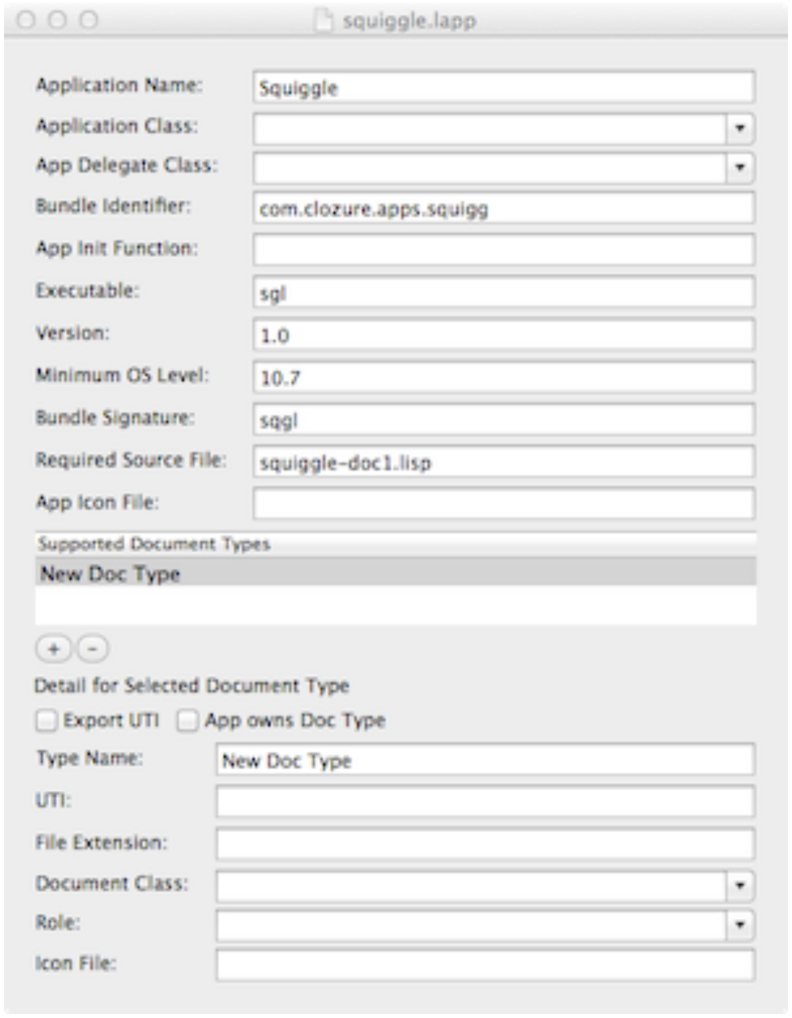
```
(defmethod remove-squiggle ((self squiggle-doc) (sq squiggle))
  (set-undo self
            #'(lambda ()
                (add-squiggle self sq))
            "remove squiggle")
  (setf (squiggle-list self) (delete sq (squiggle-list self)))
  (when (squiggle-view self)
    (#/setNeedsDisplay: (squiggle-view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

Adding documents to the LAD window

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. So first select "Load App Under IDE" from the Dev menu if you have not previously done so. In this particular case the tools will learn about the class sq::squiggle-doc.

Next let's go ahead and add a document to the squiggle application. Click the "+" button underneath the *Supported Document Types* table in the squiggle.lapp LAD window. Then select the *New Doc Type* entry that appears in the table. The window should now look like the one below:



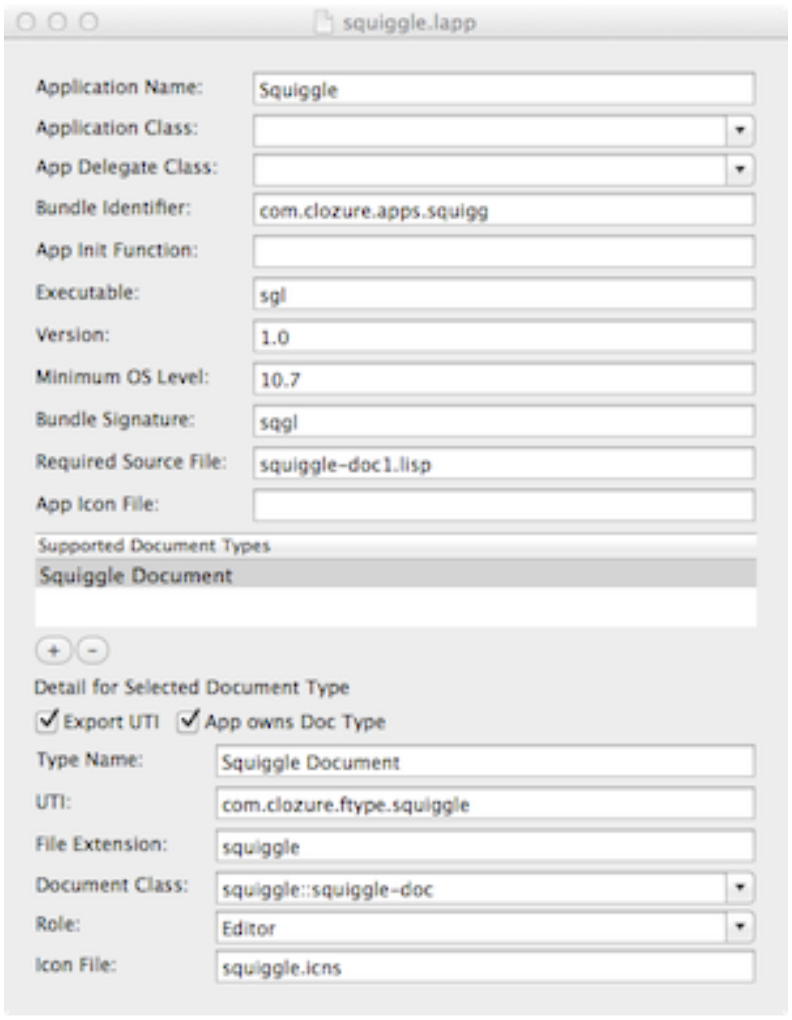
Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	squiggle-doc1.lisp
App Icon File:	

Supported Document Types
New Doc Type

Type Name:	New Doc Type
UTI:	
File Extension:	
Document Class:	
Role:	
Icon File:	

LAD window with new document type

The fields at the bottom of the LAD window will now be enabled and you can use them to edit information about the new document type. We'll go through those fields one-by-one. When we get done your window will look like the following:


squiggle.lapp

Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	squiggle-doc1.lisp
App Icon File:	

Supported Document Types

Squiggle Document

Detail for Selected Document Type

✓ Export UTI ✓ App owns Doc Type

Type Name:	Squiggle Document
UTI:	com.clozure.ftype.squiggle
File Extension:	squiggle
Document Class:	squiggle::squiggle-doc
Role:	Editor
Icon File:	squiggle.icns

LAD window after adding squiggle document information

A document's *file type* is a string that identifies what sort of document it is. Documents are typically 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.

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.

A UTI is a *Universal Type Identifier*. This is Apple's currently recommended way to identity 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 that incorporates Clozure, you will probably want to use a string that reflects your own company or personal identity.

If you export the UTI, as specified by checking the *Export UTI* checkbox, 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 with the *App owns Doc Type* checkbox, then double-clicking on such a document in the Finder will generally result in opening up that document in the 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.

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 the squiggle class defined in the source file appears as a possible selection for the field. That happened as a result of loading the squiggle source when we previously ran the squiggle app under the IDE.

Since the squiggle app will be able to edit Squiggle Documents, we will use the Role field pull-down to select "Editor".

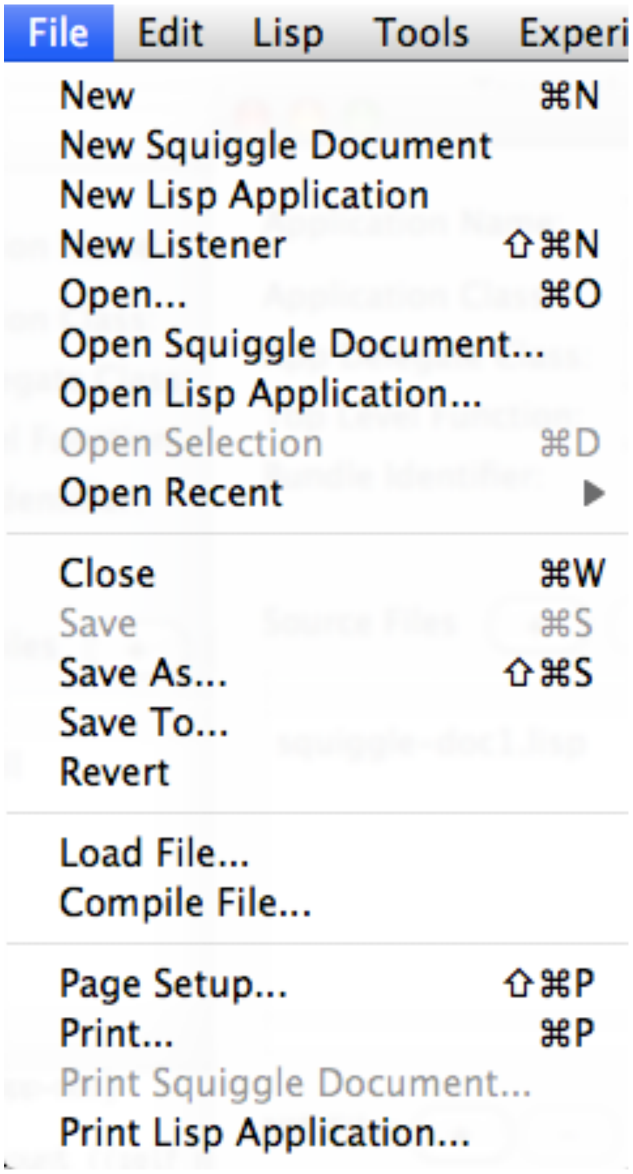
In addition to these normal mechanisms, the developer tools also use the file type and file extensions that you entered in the LAD window for similar purposes when running the application under the IDE.

To enter the path for the document icon file you can use the return key while the field is blank and then navigate to an icon file. There are any number of ways to create an icon file using publicly available freeware applications. I've found that the freeware application FastIcns (http://projects.digitalwaters.net/index.php?q=fasticns_download) works pretty well for me. I've created a few .icns files using that program that you can choose from. These are located in
        ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Cocoa Dev/Icon Templates.
The squiggle.icns file was created using the Grab utility that is bundled with Apple software to capture a section of a real squiggle window and save it as a .tiff file which was then given to FastIcns which created the .icns file. Very easy.

Loading your Document Application in the CCL IDE

Now that we have defined document information for our application, you should 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. 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. And to facilitate other debugging it will even create a default window for us.

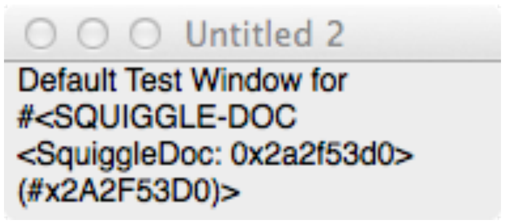
After you have loaded the application you will see that the tools have provided three new menu items in the File menu: "New Squiggle Document", "Open Squiggle Document" and "Print Squiggle Document" to let you begin to debug your application. Your File menu should look much like the figure below:



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 and we have not yet created a squiggle window.

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 window will look something like the following (actual size):



Default Test Window for #<SQUIGGLE-DOC <SquiggleDoc: 0x2a2f53d0> (#x2A2F53D0)>

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. If you actually print your document at this point the default print function for the lisp-document class will print out a list of the object's slots and value. We'll see a little later how to modify the default printing behavior to present something more meaningful and with a better format.

In addition you can now save your document, close it, and reopen 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: 0x2a2f53d0> (#x2A2F53D0)>)
```

This returns a list of all open documents of the type specified (which must designate some sort of lisp-document). The name of the type is the same name that you assigned to the document type in the LAD window. You can manipulate the list of documents that is returned in the usual ways. 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: 0x2a2f53d0> (#x2A2F53D0)>
Class: #<OBJC:OBJC-CLASS SQUIGGLE::SQUIGGLE-DOC (#x20E98920)>
Wrapper: #<CCL::CLASS-WRAPPER SQUIGGLE::SQUIGGLE-DOC
#x302001776C2D>
Instance slots
SQUIGGLE::SQUIGGLE-LIST: NIL
```

```
SQUIGGLE::ROTATIONS: 1
SQUIGGLE::SQUIGGLE-VIEW: NIL
SQUIGGLE::SQUIGGLE-WIN: NIL
SQUIGGLE::BACK-COLOR-WELL: NIL
SQUIGGLE::BACK-COLOR: #<NS-COLOR NSCalibratedRGBColorSpace 0 0 1
1 (#x20E9F950)>
SQUIGGLE::ROTATION-SLIDER: NIL
SQUIGGLE::WINDOW-FRAME: (200 200 400 400)
SQUIGGLE::NOTIF-HANDLER: NIL
NS:ISA: #<OBJC:OBJC-CLASS SQUIGGLE::SQUIGGLE-DOC (#x20E98920)>
?
```

Another way to retrieve the document associated with a window is to do something like the following:

```
? (setf doc (#/document (iu::window-titled "Untitled 2")))
#<SQUIGGLE-DOC <SquiggleDoc: 0x2a2f53d0> (#x2A2F53D0)>
?
```

That might be easier to use if you don't know exactly what the document type is for some window or if you have several windows open and want to make sure to get the document associated with a particular one.

If you would like to test the save and restore functionality, then at this point you could change any of the slot values, click on the displayed test window, 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.

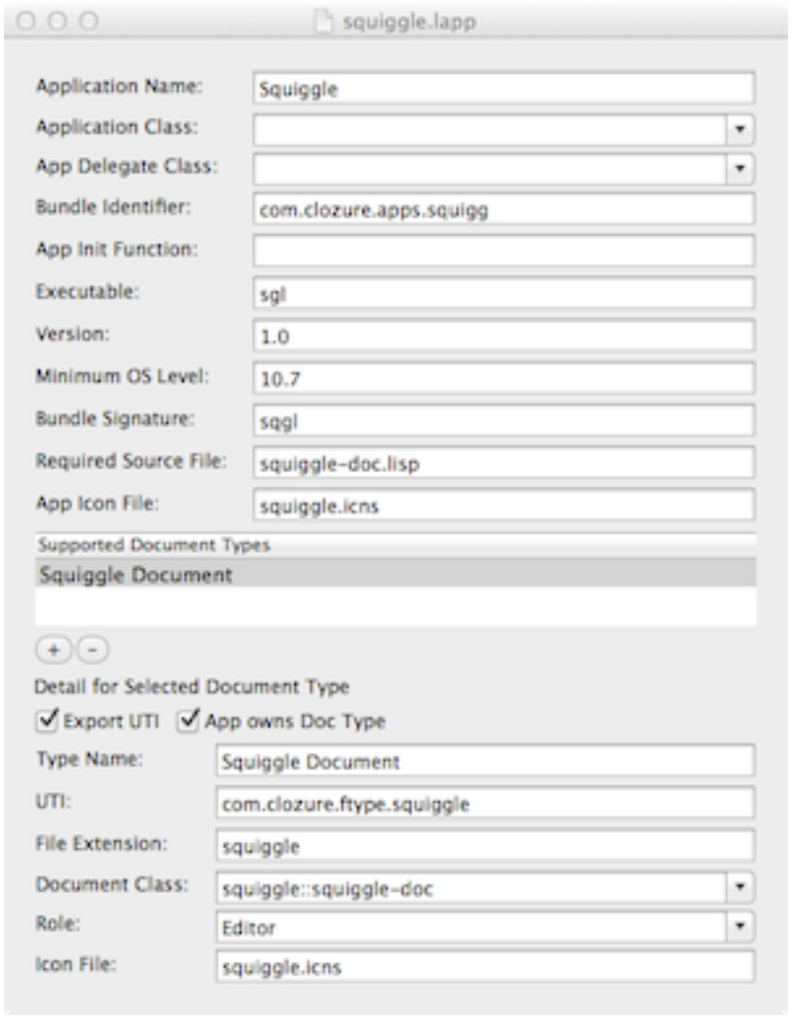
OK, so now we've got our own little mini-application running inside the IDE with a document defined that can be saved and restored. This required almost no effort on our part. Perhaps it's time that we add a better way to edit the document; namely a window.

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 define a window-definition function to do this. Readers who have worked through the *InterfaceBuilderWith CCLTutorial* will already be quite familiar with this process and if you haven't yet looked at that document, this might be a good time to do so.

At this point we'll switch the source file from squiggle-doc1.lisp to squiggle-doc.lisp. This contains the full final source for the squiggle application. Change that in the LAD window, remembering to clear the field and type the return key to bring up an open dialog that lets you search for the file. While we are at it, let's select an application icon file. Select the same file that we used for the document or use another, at your discretion. Your window should now look as follows:


squiggle.lapp

Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	squiggle-doc.lisp
App Icon File:	squiggle.icns

Supported Document Types
Squiggle Document

Detail for Selected Document Type
[✓ Export UTI] [✓ App owns Doc Type]

Type Name:	Squiggle Document
UTI:	com.clozure.ftype.squiggle
File Extension:	squiggle
Document Class:	squiggle::squiggle-doc
Role:	Editor
Icon File:	squiggle.icns

squiggle app LAD window with final source specified

As previously described, the source is contained in the following files:

- squiggle-doc.lisp
- squiggle-classes.lisp
- squiggle.lisp
- squigg-view.lisp

We've already discussed squiggle-classes.lisp, so let's look at squiggle-doc.lisp and discuss it in detail.

```
(objc:defmethod (#/dealloc :void)
                ((self squiggle-doc))
  (when (notif-handler self)
    (#/release (notif-handler self))
    (setf (notif-handler self) nil))
  (call-next-method)
  (objc:remove-lisp-slots self))
```

The #/dealloc method is one that is called for all Objective-C objects when they are about to be garbage-collected. One good reason you might want to have one for an Objective-C subclass that you define is that instances own other Objective-C objects that need to be have
#/release called to avoid memory leaks. In this case we have a notification handler object that fits that description. We'll talk about what that does in just a bit. The call-next-method invocation is important so that dealloc methods for parent classes also gets invoked. The (objc:remove-lisp-slots self) call tells CCL that Lisp slots associated with this object may be garbage-collected.

You need to have a little bit of background knowledge to understand why these two calls are made in this order. When CCL creates a new Objective-C class that contains Lisp slots, it arranges to store a reference to those slots so that they are not garbage-collected. The idea is that you want to make sure that they don't disappear until the Objective-C instance is garbage collected. To find out exactly when that occurs, CCL will automatically create a #/dealloc method that calls `(objc:remove-lisp-slots self)`. If you need to create a #/dealloc method for your own reasons, then you need to make sure that you also make that call or the Lisp slots for that object will never be garbage-collected. The reason to do the `(call-next-method)` first, is that your class might inherit from another class that also has lisp-slots which has its own dealloc. When you call remove-lisp-slots, ALL the lisp slots are removed, including those inherited from super classes. Those slots might contain things that are referenced by #/dealloc methods defined for superclasses. So you want to let the #/dealloc methods for superclasses do their thing before removing lisp slots. And if one of those parent classes removes the lisp slots before it is done here, that causes no problems.

```
(defmethod archive-slots ((obj squiggle-doc))
  (declare (ignore obj))
  (list 'squiggle-list 'rotations 'back-color 'window-frame))
```

The archive-slots method specifies what slots must be conserved for a class when it is saved to disk. We don't need the various pointers to view objects because those will be reconstituted when the document is reopened. We do save the window-frame so that we can restore it to its original size.

```
(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 remove-squiggle and add-squiggle methods are substantially the same as those we saw previously. I did remove the check for a valid view because that should never happen once we have a window, but that is a small efficiency that wasn't really necessary.

```
(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)))
```

The start-new-squiggle method will be called when there is a mouse-down event in the squigg-view (you'll see how that happens later). It creates a new squiggle and adds it to the squiggle-doc's list of squiggles. It also adds the mouse-down point to the squiggle.

```
(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))
```

The continue-squiggle method is called periodically as the user drags the cursor while holding down the mouse button. It just adds the point to the squiggle that is currently being defined.

```
(defnotification ((self squiggle-doc) (slot rotations) new-val)
  (declare (ignore new-val))
  (when (and (slot-boundp self 'squiggle-view)
             (squiggle-view self))
    (#/setNeedsDisplay: (squiggle-view self) #$YES)))

(defnotification ((self squiggle-doc) (slot back-color) new-val)
  (declare (ignore new-val))
  (when (and (slot-boundp self 'squiggle-view)
             (squiggle-view self))
    (#/setNeedsDisplay: (squiggle-view self) #$YES)))
```

The defnotification macro is defined in kvo.lisp. It provides a way for program to be notified when a :kvo slot has been modified in any way whatsoever. That includes being set via any sort of lisp call (including "(setf (slot-value ...))") or via any binding made between the slot and a property of a view object. Sometimes it is useful to do something just when slots are modified by a binding and there are other ways to do that. This method is useful when you want to know about any changes, no matter how they occurred. In this particular case, we use the method to update the display when either the rotation slot or the back-color slot is modified.

```
(defmethod frame-changed ((self squiggle-doc) notif-name win
notif-info)
  (declare (ignore notif-name notif-info))
  ;; called when the window size is changed
  (let* ((frame (coerce-obj (window-frame self) 'ns:ns-rect))
         (old-size (ns:make-ns-size (ns:ns-rect-width frame)
(ns:ns-rect-height frame)))
         (new-frame (#/frame win))
         (new-size (ns:make-ns-size (ns:ns-rect-width new-frame)
(ns:ns-rect-height new-frame))))
    (unless (equal-size-p old-size new-size)
      (set-undo self #'(lambda ()
                         (setf (window-frame-size win) old-size)
                         (setf (window-frame self) frame))
                "window size change"))
    (setf (window-frame self) new-frame)))
```

The frame-changed method is called whenever the window changes size. We'll see later how we set things up so that this will occur. It's possible that the method might also be called just as a result of changing the window's position on the screen. So we check to make sure that the size actually did change and if so set up and appropriate undo. Then we update the slot in the squiggle-document so that when the document is saved and later reopened the window will be created with its previous size.

Next in 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 Apple's documentation and the *UserInterfaceTutorial* document provides enough detail for Lisp developers that adopt the lisp-document class. What I have done 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 ns:ns-view 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 lisp-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 by 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 (draw-in-view-rect) that we'll examine in a moment. The draw-in-view-rect 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.

The next method we will discuss is the one that actually builds the Application's window. For applications that use a custom window-controller class, I would expect this method to be defined in the same source file where that window-controller's methods are defined. See the *UserInterfaceTutorial* for examples of this. Since we do not have a custom window-controller, we will define the build method in squiggle.doc.

Window build methods are called by lisp-window-controllers when they are asked to load their windows. In a typical Objective-C application window loading would involve loading a NIB file (which automatically constructs the objects that it describes). But I have adopted a strategy where all windows are built procedurally as described in the *UserInterfaceTutorial* document.

In general, the tasks that a window build method must undertake include:

1. Create all views needed within a window.
2. Create a window, adding the subviews to it.
3. Constrain both the size and position of the views in the window to achieve the desired layout behavior.
4. Create any data controllers needed.
5. Create any data source objects needed.
6. Link up or bind together views, controllers, and data objects as appropriate.
7. Return the window and a list of other objects that need to be managed to the window controller.

```
(defmethod make-squiggle-win ((wc lisp-window-controller))
  (let* ((doc (#/document wc))
         (squig-view (make-instance 'squigg-view
                       :document doc))
         (slider (make-instance 'ns:ns-slider
                   :continuous t
                   :max-value 100.0
                   :min-value 1.0))
         (back-label (make-instance 'label-view
                       :title "Background Color"))
         (color-well (make-instance 'ns:ns-color-well
                       :bordered t
                       :frame-size (list 40 20)
                       :color :blue))
         (win (make-instance 'ns:ns-window
                :resizable t
                :frame (window-frame doc)
                :content-subviews (list squig-view slider back-
label color-well)))
         (cv (#/contentView win)))

    (setf (squiggle-view doc) squig-view)
    (setf (squiggle-win doc) win)
    (setf (rotation-slider doc) slider)
    (setf (back-color-well doc) color-well)

    (bind slider "value" doc "rotations")
    (bind color-well "value" doc "backColor")

    ;; Add view size constraints
    (constrain-to-natural-size color-well)
    (constrain (= (height slider) (height color-well)))
    (constrain (>= (width slider) (height slider))) ;; forces
slider to be horizontal

    ;; Add view position constraints
    (anchor cv squig-view (list :top :left :right))
    (anchor cv slider (list :left))
    (anchor cv color-well (list :right :bottom))
    (constrain (= (top slider) (+ 6 (bottom squig-view))))
    (order-views :views (list :border slider 12 back-label 2
color-well :border)
                 :orientation :h
                 :align :center-y)

    ;; Set up things so that the squiggle doc gets notified when
the window's frame changes size so
    ;; that it can save that value off and restore it when a
squiggle-doc is reopened.
    (setf (notif-handler doc) (make-instance 'notification-
handler
                                :from win
                                :notifications (list #
$NSWindowDidResizeNotification
                                                     #
$NSWindowDidMoveNotification)
                                :target doc
```

```
                                    :func #'frame-changed))

    (values win (list squig-view slider back-label color-
well))))
```

To understand how this method fits into the global picture, it helps to understand the order in which various things are done in a document-based cocoa application. When the user selects the "Open" menu-item the shared-document-controller for the application will create an openpanel and let the user select what file should be opened. It then creates an instance of the ns-document sub-class that was archived to that file and tells it to initialize itself using the file. Then it tells that document to create any window controllers that are needed to display itself. Finally, each window-controller is told to load itself. When a lisp-window-controller is told to load itself, it calls a build function that was provided when it was initialized (discussed shortly) and passes itself as the single argument. Since window-controllers and ns-documents keep mutual references. It is easy for the build routine to find the document for which the window is being built. That can make it easy to set up any reference pointers or bindings that may be needed.

The make-squiggle-win method first creates the squigg-view, slider, label, and color-well objects that will be displayed in the window. It then creates the window and adds those objects as subviews of its content-view.

Next it sets the slots in the squiggle-document that point to view objects. After that it binds the values of the rotation slider and background color-well objects to the corresponding slots in the squiggle-document. That causes the slot to be automatically updated when a user manipulates those views in the window.

After that, constraints are added to force objects to behave as intended, especially when the user manually resizes the window. The *UserInterfaceTutorial* document provides many examples of how constraints can be used and debugged, so I'm not going to say much more about them here. Even if you don't completely get constraints yet, I think the intent of these should be fairly clear. Creating window layouts with constraints is something of a an art form that takes time to master. It is easy to either under-specify constraints, which makes the layout ambiguous, or over-constrain which can make it impossible for the runtime to simultaneously satisfy all of those specified.

The penultimate thing done in the build routine is to set things up so that the squiggle-document gets notified when the window's frame changes sizes. Rather than go into detail about how Cocoa notifications work, let me discuss the Lisp interface to notifications that I created to make life a bit easier. I created the notification-handler class. It takes :from and :notifications arguments that together specify which notifications we want to know about (i.e. those that originate from the :from argument and are of the types specified in the :notifications list). The :target and :func arguments together tell the notification handler what to do when such a notification arrives, namely call the :func specified using the :target as the first argument passed. Three other arguments are also passed to that function: the name of the notification, the object it came from, and any additional information that was incorporated into the notification. Earlier we discussed the frame-changed method and how it simply uses the notification to trigger a re-setting of the window-frame slot in the squiggle-document.

The last thing the build method does is to return the window along with a list of all the views instantiated as part of the build process. The lisp-window-controller will call #/awakeFromNib on all those objects which respond to that message (since many view objects do special things when that is called) and make sure that they stay around until the window is closed. That is, it *owns* those objects, much as a standard window-controller will own the objects that are created when a nib is loaded.

```
(defun init-squiggle-app (app)
  ;; The function that gets called to initialize the application
  ;; It creates and installs a main menu.
  (let ((*app-name-for-menus* "Squiggle"))
    (#/setMainMenu: app (standard-main-menu))
    (set-windows-menu)))
```

Eventually we will set the "App Init Function" field in LAD window and when we do, we will specify the init-squiggle-app function shown above. This will be discussed more thoroughly in the *Adding A Main Menu To Your Application* section below.

```
(defmethod window-build-funcs ((self squiggle-doc))
  (list #'make-squiggle-win))
```

After a document is created (whether it be a new on or one loaded from a file) it will be told to create whatever window-controllers are needed to display its content. Base functionality in the lisp-document class will take care of doing that, but it needs to two things to work correctly:

- A list of lisp-window-controller sub-classes; one for each window that needs to be created, and
- A list of the corresponding build-methods for each of the windows.

To provide the former you should override the *document-window-controller-classes* method for your document class. The default method returns a list with a single item: lisp-window-controller. To provide the latter you should override the *window-build-funcs* method for your class. The default for this method returns a list with a single item: #'make-default-lisp-window. It was these defaults that resulted in popping up a proxy window before we had provided any window-building functionality. By defining the window-build-funcs method for the squiggle-doc class we change how its windows are built.

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. We saw the squiggle class definition previously in the squiggle-classes.lisp file. Functionality for that class is contained in the squiggle.lisp 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 ns:ns-bezier-path. 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 ns:ns-color object.

Just to have it handy, the class definition found in the squiggle-classes.lisp file was:

```
(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 ns:ns-bezier-path 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.

Again, from the squiggle-classes.lisp file we have:

```
(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 will use 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 example 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 the code explained why. What is drawn is sensitive to the current shape and size of the view within which it is drawn. The squiggle positions are all designated relative to the lower-left corner and then rotated about the center of the view. If you change the view's size or relative dimensions, then the center of the view (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 view, 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, translate 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 (#/isOpague #>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 code.

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.

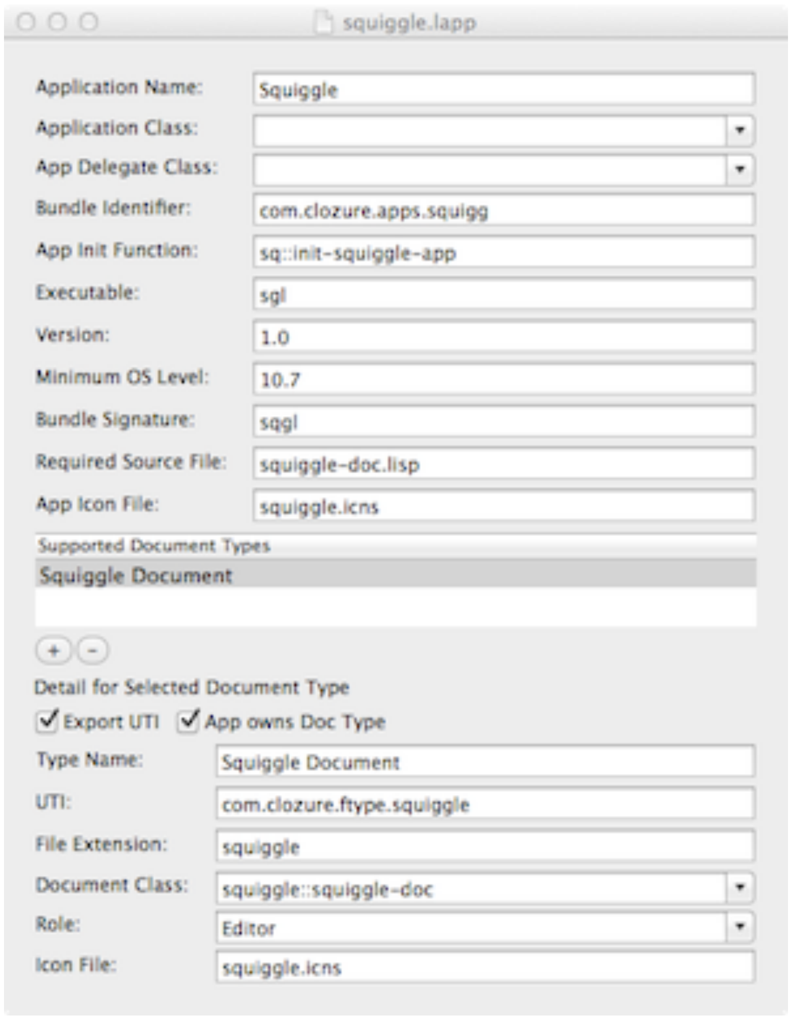
If you are wondering how this is all made to work under the IDE, be patient for a bit and a short overview will be provided after we show how to add a main menu to your application.

Adding A Main Menu To Your Application

We previously saw the definition for the following function in squiggle-doc.lisp:

```
(defun init-squiggle-app (app)
  ;; The function that gets called to initialize the application
  ;; It creates and installs a main menu.
  (let ((*app-name-for-menus* "Squiggle"))
    (#/setMainMenu: app (standard-main-menu))
    (set-windows-menu)))
```

At this point you can go ahead and add that specification to the LAD window's field. That window will now look as follows:



Application Name:	Squiggle
Application Class:	
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.squigg
App Init Function:	sq::init-squiggle-app
Executable:	sgl
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	sqgl
Required Source File:	squiggle-doc.lisp
App Icon File:	squiggle.icns

Supported Document Types

Squiggle Document

Detail for Selected Document Type

✓ Export UTI ✓ App owns Doc Type

Type Name:	Squiggle Document
UTI:	com.clozure.ftype.squiggle
File Extension:	squiggle
Document Class:	squiggle::squiggle-doc
Role:	Editor
Icon File:	squiggle.icns

squiggle window with app init function specified

I need to provide a bit of background information here to help you understand what this does.

When a CCL Lisp application starts up, its top-level function is run. For the most part the default top-level function used for the CCL IDE will work just fine for our Lisp applications, but it does assume that the application has a main NIB file that will be loaded. The Lisp start up code creates an ns:ns-application object and then looks in the info.plist file in the application's bundle to find the name of that nib file. Loading main NIB files typically results in setting the Main Menu for the application and it may additionally create other sorts of objects. Apps built using the methodology described here and in the accompanying *UserInterfaceTutorial* will not have such a NIB file, so we must replace its functionality by calling what I've called the App Init Function.

We could have defined a whole new top-level function that accomplished this, but since the existing version mostly does what we want it do do, I instead redefined the function gui::init-cocoa-ide which is called by that top-level function. You can find the new version in the file custom-app-init.lisp. This file will only be loaded into a new application that is created, never into the CCL IDE. It is substantially the same as the original version, but will look for either a main nib file or an app init function specified in the application's info.plist file and use whichever is present (possibly both). The specified app init function will be run right right after the ns:ns-application is created and at a minimum should set the main menu for the application, just as a would be done by loading a main nib file.

For the Squiggle application we will specify the app init function shown above. Setting the corresponding field in the LAD window will cause the name of the function to be put into the Info.plist for our app so that it can be found at startup. The file menu-utils.lisp provides a variety of functions to make it easier to create either standard or custom menus and menu-items. Some menu-items customarily include the name of the application and by binding *app-name-for-menus* to that name we can automatically get customized menu-items when that is appropriate. We set the main menu for the application with the #/setMainMenu: call. The (set-windows-menu) call tells the application that a list of open windows should be appended to the menu named "Window". That is appropriate for the squiggle application, but may not be for others. I suggest that you study the functions available in menu-utils.lisp to see how to create your own customized main menu. In particular, the function menu-item-for-key will take any one of a large number of keywords and return a corresponding standard menu-item with the usual name, target, action, and keyboard equivalents defined. Seeing how this is done should provide a good template for creating your own custom menu items.

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 "Clozure <something>" rather than Squiggle as we defined when we created our main menu. 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. When we get to having a completely stand-alone application it will be titled in the way we would expect. 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 "Clozure ..." 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, which will always be the CCL IDE menus.

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 still not yet very good with it even after using it for many years. 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. And every once in a while you will get into a situation where the only way out is to kill the executing thread, in which case you're going to have to restart Lisp.

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 functions or method source, reevaluate it, and immediately see the effects in the changed behavior of your running application. You can modify a window-build definition, open a new instance of that window and immediately see the changed effects. 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, 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 create a delegate object in your application initialization function. To do that you would first define your delegate class as a new sub-class of ns:ns-object, create an instance of it, and make an appropriate (#/setDelegate: ...) call there. If you decide to do that, you should use the argument passed into the initialization function as the application object rather than using the #$NSApp runtime variable. In a stand-alone application, those would be the same so it would make no difference, but when running under the CCL IDE, they are not. In that case, the argument passed into the application initialization function is actually an instance of the lisp-doc-controller class and you want to make sure to set ITS delegate rather than setting a new delegate for the ns-application object that is running the CCL IDE. Doing the latter would disrupt the normal behavior of the IDE.

The other way to specify what sort of object you want to be your delegate is to designate an app delegate class in the LAD window. 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 ns:ns-application 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 an app init function 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 window build function 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 an application init function that sets up a main menu. In this case, the controller first acts something like an NSApplication object. For stand-alone applications that are constructed using these tools, an application init function is normally executed at startup passing an ns:ns-application object as the single argument. Our controller object will do much the same for an application init function that you specified for your application. If that function sets up a new main menu, that will automatically replace the current CCL IDE main menu 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 the CCL main menu items 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 setting it in your app init function, then it will be linked as the delegate of the lisp-doc-controller instead, because it passes itself as the argument to your app init function. 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 an app init function 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 (by specifying no target when the menu-item is created) or directly to the application argument provided to the app init function (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 app argument provided to the app init function 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 create and link your delegate object to the app argument 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.

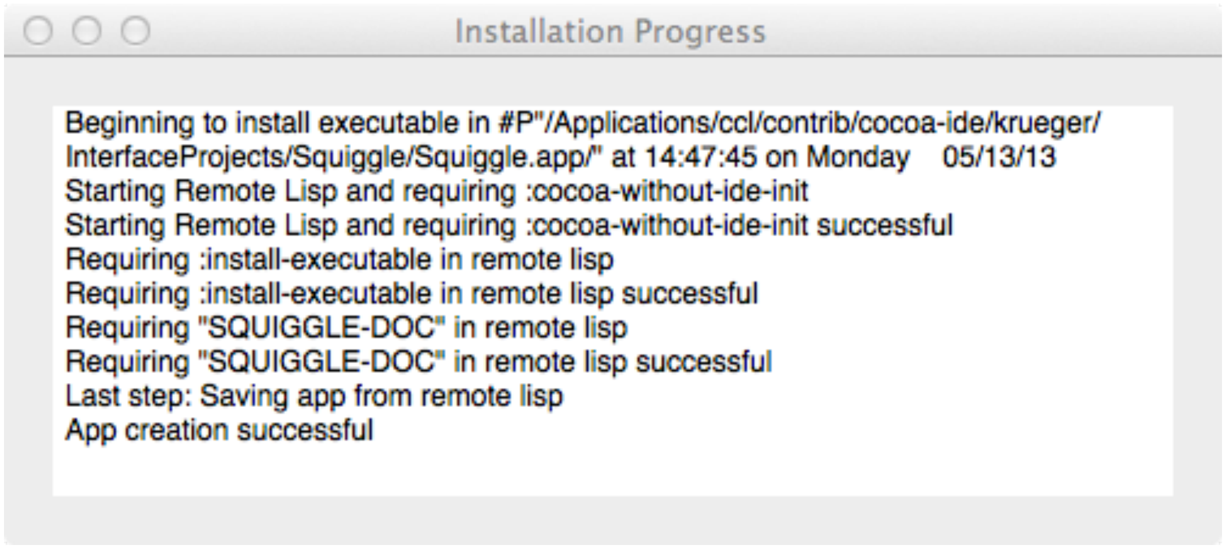
The Lisp top-level code which starts up Cocoa applications will create an instance of a subclass of ns:ns-application. This must be specified in the bundle's Info.plist file. The tools will do this for you using the class specified in the Application Class field of the LAD window. For stand-alone applications it is normally sufficient to use the basic ns:ns-application 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.

When we modified the init-cocoa-ide function as described earlier, in addition to making it aware of application initialization functions that are specified in the application's info.plist, we also changed its behavior with respect to application delegates. The CCL-provided version of this function would have created an application delegate of a default class if one wasn't explicitly provided. Our new version will only set an application delegate if it is explicitly provided. Since we don't really need one for our Squiggle app, we can leave that field blank

For those who want to create an application delegate for their own applications, I have provided a default class called app-dev::simple-lisp-app-delegate which you can use directly or inherit from if desired. The advantage of inheriting from this class is that if you use the pull-down menu next to the "App Delegate Class" field in the LAD window you will see 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.

The only thing left to do is to install an executable into the bundle. If done properly, that will result in an executable application bundle. When that application is double-clicked in the finder, the 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 done so that the CCL IDE running our tools remains up and running while we save from a separate CCL instance. We open up a pipe from the CCL running the tools 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. A separate window will pop up to show you how the installation of the executable is proceeding. If the process runs to completion, that window will look something like the following (shown full size):



```
Beginning to install executable in #P"/Applications/ccl/contrib/cocoa-ide/krueger/
InterfaceProjects/Squiggle/Squiggle.app/" at 14:47:45 on Monday    05/13/13
Starting Remote Lisp and requiring :cocoa-without-ide-init
Starting Remote Lisp and requiring :cocoa-without-ide-init successful
Requiring :install-executable in remote lisp
Requiring :install-executable in remote lisp successful
Requiring "SQUIGGLE-DOC" in remote lisp
Requiring "SQUIGGLE-DOC" in remote lisp successful
Last step: Saving app from remote lisp
App creation successful
```

It first causes the subordinate Lisp to load all the Cocoa functionality which can take a bit of time, so be patient. In addition to the message displayed in the progress window, 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 a small amount of helper code followed by your 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.

You can reinstall a new executable at any time if you find that changes are needed to your Lisp code.

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

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.

This project is a very straight-forward extension of Project #7 in the *UserInterfaceTutorial* document. The details of the application logic are described in that document and will not be reviewed here. I would strongly suggest that if you have not already done so, you work through that project (as well as those leading up to it). Here I want to emphasize how easy it is to go from a fairly full-featured document-based application that only ran under the CCL IDE to one which can run as a stand-alone application.

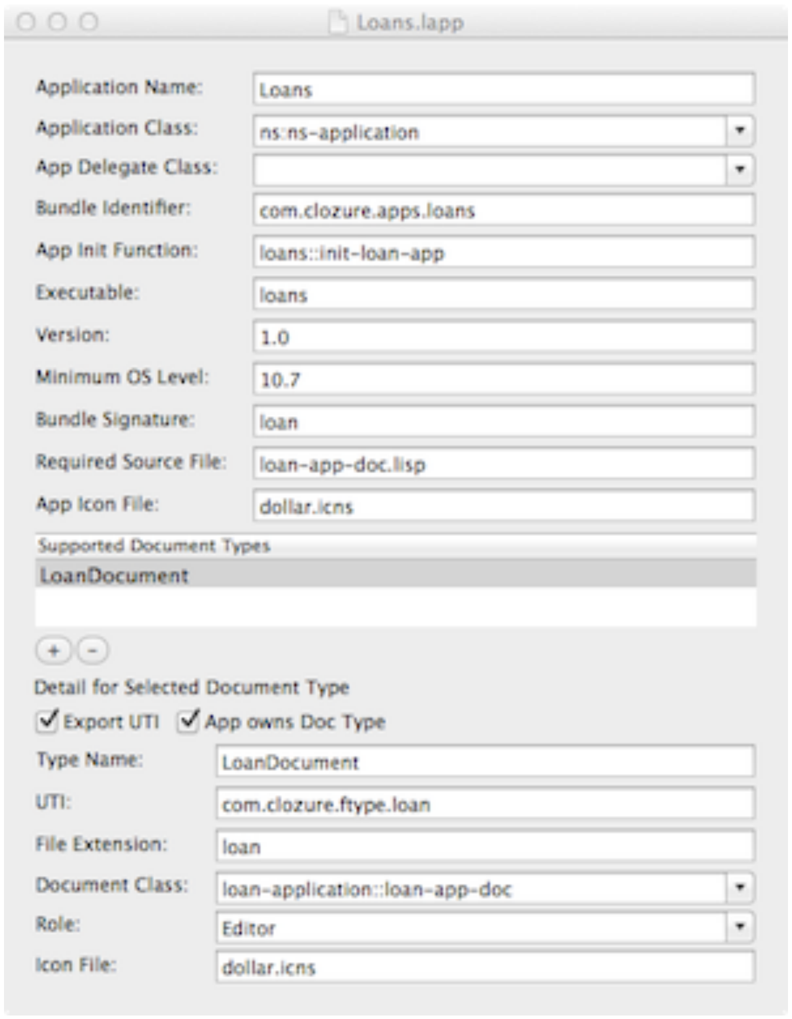
Let's start by creating a new LAD for our Loan application. All referenced files are found in:

    ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Loan Application

Select "New Lisp Application" from the File menu in the CCL IDE. Modify the window to match the figure below. Select the source file loan-app-doc.lisp by hitting the return key in the *Required Source File* field and navigating to the directory above to select it. 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 loans::loan-app-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 "Loans".

Select the icon files for the application and the LoanDocument in the same manner, by navigating (as we did previously) to:

    ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects/Cocoa Dev/Icon Templates


Loans.lapp

Application Name:	Loans
Application Class:	ns:ns-application
App Delegate Class:	
Bundle Identifier:	com.clozure.apps.loans
App Init Function:	loans::init-loan-app
Executable:	loans
Version:	1.0
Minimum OS Level:	10.7
Bundle Signature:	loan
Required Source File:	loan-app-doc.lisp
App Icon File:	dollar.icns

Supported Document Types

LoanDocument

Detail for Selected Document Type

[✓ Export UTI] [✓ App owns Doc Type]

Type Name:	LoanDocument
UTI:	com.clozure.ftype.loan
File Extension:	loan
Document Class:	loan-application::loan-app-doc
Role:	Editor
Icon File:	dollar.icns

LAD 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 Manu (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.

Let's now take a look at the source code modifications that were made to project #7 described in the *UserInterfaceTutorial* to get to this application. Here is an overview list of the modifications that I made to that project:

- Refactored code to isolate class definitions into a single file
- Changed the main loan class to inherit from lisp-document
- Eliminated the window-controller slot in the loan class
- Changed the way that loans are saved to use mechanisms provided by the lisp-document class
- Changed the way that loans are printed to use mechanisms provided by the lisp-document class
- Changed the way that window controllers are managed
- Added an application initialization function

We'll go through these one-by-one to see what was involved in making the change. It took me about an hour to make all these changes and test the stand-alone application (less than the time it took me to add this documentation of what was done).

First, in a purely esthetic change, I moved the two classes of interest into their own file: loan-app-classes.lisp. The loan-application (loans) package is also defined there. In that file the definition of the loan-app-doc class was made to inherit from the lisp-document class. That gave us access to methods which support saving, re-opening, and printing of documents. Since documents normally manage their own window controllers, we no longer needed that slot in our loan document class.

All of the remaining changes were made in the loan-app-doc.lisp file. I left the bulk of the old code in place (commented out) so that you could easily see what was eliminated. The loan-app-win-cntl.lisp file is identical (except for name changes) to the .../Loan Document/loan-win-cntrl.lisp file from project #7. We were also able to completely eliminate the loan-pr-view.lisp file which supported printing in project #7.

Since the project #7 code was perfectly adequate for saving and reloading loan data, we could have just left it intact. However, I chose to replace it with methods that made use of the mechanisms provided by the lisp-document class. That required that the following two methods be implemented:

```
(defmethod archive-slots ((self loan-app-doc))
  (declare (ignore self))
  (list 'loan-amount 'interest-rate 'loan-duration 'desired-
loan-duration
        'monthly-payment 'origination-date 'first-payment))

(defmethod document-did-open ((self loan-app-doc))
  (compute-new-loan-values self))
```

The archive-slots method should be familiar to you. It specifies which slots will be saved off. We elect to save only the slots which are not computed by the compute-new-loan-values method. All other slots will be reset when the document is reloaded.

After a document is loaded from its file representation, the lisp-document class methods arrange to call the method document-did-open. The default method does nothing. This method permits the document to do anything additional that might be needed after its slots have been restored. In our case, we want to call the compute-new-loan-values method in order to properly initialize all other slots that will be displayed in the loan window. This is exactly the same method that gets called when a user modifies some value in the loan window. What we eliminated were the Objective-C methods #/readFromData:ofType:error: and #/dataOfType:error: These were actually a pretty light-weight way of saving loan data, so this wasn't a great savings, but it could make future modifications easier to maintain.

Using the lisp-document methods also makes it possible to add or remove slots from what is saved and maintain compatibility between older saved documents and the newer versions of the application. That's because the external representation of class data that was used by the lisp-document class was intentionally designed so that future apps can simply ignore some of the slot data if they choose to do so. And as long as a future app provides reasonable default values for any new slots, it can then safely load old versions of saved data without problems.

Printing is a much simpler operation than shown for project #7. All we need to do is supply the lines to be printed and let the lisp-document print methods worry about pagination. We might have to do more if we wanted to supply page headers and/or footers for each page, but I took the easy way out here and just let the data flow from one page to the next. The method that was added was:

```
(defmethod print-lines ((self loan-app-doc) lines-per-page)
  (declare (ignore lines-per-page))
  (let ((start-lines (list (format nil
                                   "Loan ID: ~a"
                                   (coerce-obj (#/displayName
self) 'string))
                           (format nil
                                   "Amount: $~$"
                                   (/ (loan-amount self) 100))
                           (format nil
                                   "Origination Date: ~a"
                                   (date-string (origination-
date self)))
                           (format nil
                                   "Annual Interest Rate: ~7,4F
%"
                                   (* 100 (interest-rate self)))
                           (format nil
                                   "Loan Duration: ~D month~:P"
                                   (loan-duration self))
                           (format nil
                                   "Monthly Payment: $~$"
```

```
                                  (/ (monthly-payment self)
100))
                          ""))
        (pay-lines nil))
    (dolist (sched-line (pay-schedule self))
      (push (format nil
                     "~1{On ~a balance = $~$ + interest of $~$ -
payment of $~$ = ~a balance of $~$~}"
                     sched-line)
            pay-lines))
    (nconc start-lines (nreverse pay-lines))))
```

The print-lines method is called by lisp-document print methods to get a list of lines to be printed. Our implementation creates several header lines with basic loan data and then generates one line for each month of the loan's payout schedule. It returns the complete list of lines and the lisp-document method handles things from there. What was replaced was the entire ...Loan Document/loan-pr-view.lisp file as well as the Objective-C methods #/printLoan: and #/printOperationWithSettings:error: in the loan-app-doc.lisp file.

The management of the window-controller and the creation of windows was done explicitly in project #7. For this project we make it operate in a more standard fashion and take advantage of lisp-document mechanisms which help us to do that. The following two methods were added:

```
(defmethod window-build-funcs ((self loan-app-doc))
  (list #'make-loan-window))

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

These respectively specify the window build function that is needed and the class of the window controller that will be created when the document's window (singular in this case) needs to be opened. The build function is passed as an initialization argument when the window controller is created. Since the lisp-document class inherits from the ns:ns-document class, the management of the document's window controllers is handled automatically for us. This let us eliminate the window-closed and close-loan methods that were part of project #7.

The final addition for this project was the addition of an application initialization method:

```
(defun init-loan-app (app)
  ;; The function that gets called to initialize the application
  ;; It creates and installs a main menu.
  (let ((*app-name-for-menus* "Loan"))
    (#/setMainMenu: app (standard-main-menu))
    (set-windows-menu)))
```

This simply creates and installs a standard menu.

That concludes discussion of the Loan Calculation Application.

Advanced Topics

Creating applications without documents

It is quite easy to create applications that do not need documents. Simply do not add any document specifications. In the application's initialization function you should create a main menu, as you would for a document-based application, along with any other windows and supporting objects that your application needs.

Creating a custom application class

In most cases, you should not have a need to create a custom application class, using ns:ns-application will work just fine and creating an application delegate (see below) will usually provide any additional functionality that is needed. 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 standard 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. This will also be a fairly common thing to do in applications that have no documents.