

# **Cocoa Interfaces Using Clozure Common Lisp (CCL)**

## **Version 4.2 July 2017**

Copyright © 2010-2017 Paul L. Krueger All rights reserved.

Paul Krueger, Ph.D.

## **Preamble to version 4.2 July 2017**

This is a minor revision to the Cocoa user interface libraries that is consistent with the version of CCL that can be downloaded from the Apple App Store and the version of my auxiliary code that can be downloaded from github since it is no longer bundled with the CCL distribution. Documentation has been edited significantly to be consistent the names and locations of files. Various errors have been fixed.

## **Preamble to version 4.1 December 2014**

This is a minor revision to the Cocoa user interface libraries that fixes several bugs that were present in the previous version and slightly updates this documentation to be more clear about how windows and window view objects that are created are managed. The previous code would have leaked some memory. While I won't guarantee that the current code is leak-free, at least I have plugged a few holes.

## **Preamble to version 4.0 April 2013**

This version of my Cocoa user interface libraries is a major break from previous versions. If you're unfamiliar with those previous versions and don't care about the history, skip this section and go to the introduction.

Previous versions depended exclusively on the use of Apple's Interface Builder to create window designs. I won't go over all the good reasons I had for doing that, but suffice it to say that things change quickly in this industry and Apple made several "upgrades" that made it increasingly more difficult to do things in the way I recommended. They eliminated IB plugins and a big part of the Lisp interface that I provided was embedded in a LispController plugin. They also more tightly integrated IB functionality within XCode, eliminating the stand-alone IB application. IB no longer has the option of directly outputting a .nib file, using .xib format instead. Applications still load .nib files and XCode compiles these as part of their build process. While it would be possible to incorporate a similar process into Lisp using Apple converter apps, it wouldn't have been pretty.

Apple also made some very positive improvements in interface design in the form of the automatic layout functionality that was introduced with OSX 10.7. This permits users to specify constraints on how interface objects are related to each other and let the runtime system dynamically lay out the interface depending on the size of the window

and other factors. That means that a Lisp developer can specify a set of constraints and not worry too much about specific positions or sizes of things. You can explicitly size things that need that and let every other interface element adjust automatically according to relative constraints that you specify. There are also good runtime visualization functions that help you to see what your constraints are doing, so you can figure out why something isn't working quite the way that you expected. For Lisp users, being able to debug interfaces through easy REPL interaction is the most normal way to work. These ideas will make more sense as you go through the rest of this tutorial.

I also discovered along the way that avoiding IB made it possible to extend the Objective-C binding protocol to Lisp slots in a much more versatile way than is possible when bindings are specified exclusively within IB. That alone might have made the change worthwhile.

The upshot of all that is that this new approach avoids Interface Builder altogether and creates all windows dynamically at runtime.

I previously implemented a fairly substantial number of data conversion methods that went back and forth between Lisp data and Objective-C data. I have added to that set somewhat and combined all of them within a single `coerce-obj` method that functions much like the Lisp `coerce` method. So it is no longer necessary to remember the specific method names for each type of conversion that is desired.

I implemented `initialize-instance :after` methods for many common Objective-C classes. This makes it possible to use a more conventional Lisp process that initializes substantially everything for a new instance at the time it is created. That compares to the standard Objective-C process that creates an instance and then uses a number of `set...` methods to initialize it. I was careful to assure that these methods make no changes to an object's default state unless explicitly specified via an appropriate `initarg`. While I was not specifically intending to augment the `easygui` work that has been done, these methods should make life easier for `easygui` users as well. One of the example projects provides an interface to the documentation for this new functionality. It combines a class browser with information about what `initargs` are

I also added a comprehensive Lisp interface to Apple's visual constraint system. Appendix B provides complete documentation for this facility.

## Introduction

This tutorial provides a guide to creating Cocoa interfaces for Clozure Common Lisp (CCL) programs. It requires that you be running OS X 10.7 or later and CCL version 1.11 downloaded from the Apple App Store. Using previous CCL versions would require various name changes to be consistent with previous CCL class and variable names that were changed for version 1.11. I probably should have `if-def`'ed those changes, but since I didn't, version 1.11 is required. It was tested with OS X version 10.11.6.

This is not a comprehensive guide to Cocoa; that is simply too big an undertaking and

there is ample documentation from other sources. What I've tried to do is gradually introduce you to the Cocoa/Lisp development world so that you can go explore the many possibilities on your own with a good sense of how those things might be integrated into Lisp. For the sake of simplicity I have taken a fairly narrow path and not tried to explain or use every possible bell and whistle that Cocoa provides. Hopefully I will have provided enough of a foundation that each of you can individually explore the possibilities on your own.

Although I have tried to be as factual as possible, it is entirely likely that there are things here which are in error. Please notify me of any such things (email [plkrueger \(at\) comcast.net](mailto:plkrueger@comcast.net)) and I will make corrections as quickly as possible. All of this was done with the OS X version 10.11.6, and CCL version 1.11. Since all of this is a moving target, the diagrams and examples may look slightly different on your system.

Whenever I add a new project I will change the major version number of this document. Minor version number changes indicated corrections without adding a new project.

There are a number of different objectives that you might have for development of graphical user interfaces for CCL programs and it's important to understand what the goals were for this work so that you know whether to keep reading or look for an alternative.

The opinions expressed below are just that, opinions, and others will certainly disagree. So with that in mind, in order of importance to me, the goals that I had for selecting an approach to user interface development were:

1. Usable either to create stand-alone executables or interactively from within a standard

- Lisp Read/Eval/Print/Loop (REPL)

2. Looks native to the platform
3. Easy blending with Lisp code
4. Development effort commensurate with interface complexity
5. Cross-platform/OS portability

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

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

*Goal 2: Looks native to platform*

I've been developing user interfaces of one sort or another for almost five decades. One

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

### *Goal 3: Easy blending with Lisp code*

There are various ways that interfaces can be created on each platform. There are innumerable packages that exist to make this easy. For example X-windows, TCL/TK, Java Swing, and many more. None of these is particularly easy to use with Lisp although there certainly have been credible attempts to make them so (e.g. CLX and Garnet). One reason that none of these attempts has been widely adopted is that they often become obsolete rather quickly. That is because making those interface packages easily accessible to Lisp developers often entails a fair amount of bridge or translation code which becomes a burden for anyone to maintain. User interface packages tend to change rather rapidly and as a consequence the Lisp interface can quickly become out-of-date. The alternative is often to make a bridge that is NOT easy for Lisp developers and that's not much better.

### *Goal 4: Development effort matches user interface complexity*

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

### *Goal 5: Cross-platform/OS portability*

Basically I'm a Macintosh developer. This goal almost isn't on my radar, but I certainly recognize that for others it is a must. While I didn't go looking for a solution that made this a high priority I was pleased to find that there is some potential for cross-platform portability with the approach that I have taken although it is clearly not available today (July 2017). More on that later.

## **The Search for a Solution**

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

My platform of choice is the Apple Macintosh. So I started my search by trying to understand how user interfaces are constructed in that environment. That very quickly

led me to Cocoa and I decided early on that any reasonable approach must use it. I learned much of what I know about Cocoa by working my way through

"Cocoa Programming for MAC<sup>®</sup> OS X", Third Edition, by Aaron Hillegass but like all interface books, this one is now wildly out of date. There are certainly many more current books that can tell you how to build applications with various sorts of interfaces using XCode, but i'm not going to suggest that you learn how to do that.

Although I hope my examples and this accompanying documentation will give people a head-start, there just isn't any substitute for understanding how Cocoa works and there are many resources for doing so. All of Apple's current documentation uses either Objective-C or Swift to document Cocoa. I expect that many or most of the people reading this may not have any knowledge of Objective-C or Swift and will, therefore, shy away from this approach. All I can say to persuade you is that as much as I like Lisp (at last count I've developed programs in 29 different languages and Lisp is still my favorite) Objective-C is a pretty nice language if you favor C-type languages and Swift incorporates a number of lisp-like facilities. You should be able to pick up one of these reasonably easy if you have a background in almost any other language. I assure you that Objective-C is nothing like C++ if that helps any. While I like Swift better than Objective-C, CCL contains bridge code that makes making Objective-C calls easy, so you might be better off learning that until someone volunteers to write a Swift-bridge for CCL. Since both Swift and Objective-C use the Objective-C runtime, it shouldn't be all that difficult to modify the existing bridge code.

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

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

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

between them.

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

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

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

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

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

So for a couple of years I used a development paradigm that included both CCL and the

Apple InterfaceBuilder application (IB). Those of you who used previous versions of my code undoubtedly did the same. But Apple made changes to IB functionality that precluded the use of plug-ins and coupled IB with XCode much more tightly, both of which made that my process get uglier by the day.

The good news is that with OS X 10.7 Apple also added some interface layout functionality in the form of constraints that makes procedural specification of interfaces at runtime MUCH easier than it would have been previously. The ability to specify constraints between interface objects and let the runtime system decide exact positions and sizes makes it once again plausible to specify them within Lisp code and avoid IB altogether. Apple also provided functions to dynamically display constraints visually within your interface window and identify which constraints are used to influence how some element of the interface is sized and placed. Those make it fairly easy to debug interfaces directly via the Lisp REPL. Once I added some additional code to analyze all the constraints of a window I found that window design with constraints because a viable alternative.

This approach is not altogether without problems. One big thing that IB does is provide higher-level interface constructs automatically for you. To get something comparable in Lisp requires code to put together separate objects in the same fashion. The good news is that there aren't really that many of those things and I've done some of that for you already. Most of the normal interface elements that you would want to have are readily available. I hope that as a community perhaps we can turn out more and more of the more complex combinations over time.

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

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

I can clearly define interfaces via Lisp code within a standard CCL IDE and dynamically modify them in a REPL environment. I routinely redefine methods while a window is up and immediately see the modified behavior. To change the look of an interface I can also dynamically modify user-interface constraints and immediately see what effect they have. You will also see later how stand-alone applications can be generated.

*Goal 2: Looks native to platform*

Clearly Cocoa is the way to get a native look and feel on Macintosh systems and I can now access all of Cocoa's capabilities without wondering how some intermediate layer might have implemented those capabilities. Various combinations of primitive user-interface elements do require intermediate Lisp code, but this is quite comparable to the sorts of things you see in Apple's Objective-C sample applications.

*Goal 3: Easy blending with Lisp code*

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

to incorporate those changes into a Lisp layer.

*Goal 4: Development effort matches user interface complexity*

I find that developing user interfaces can be done very quickly, even without using IB, using the constraint system. I have added some additional idioms on top of basic constraints that build up a web of constraints to accomplish a particular sort of relationship. This use of constraints to define how an interface should look has a particularly lispish feel to it to me. Or perhaps it's my training in Artificial Intelligence, but this feels like a much *smarter* way to design interface functionality. Depending on the complexity of what you are trying to do this is minutes to hours in duration. Typically I require a longer amount of time to debug the behavior that I want, but this is where the Lisp REPL environment proves its worth. Rather than finding a bug and rebuilding as would be required in an Xcode world, I can very quickly examine objects, modify methods, visualize and modify constraints on display elements, and generally do the sorts of things that developers always do in Lisp.

*Goal 5: Cross-platform/OS portability*

As I said initially, portability was not my main criteria, but as I browsed through some of the CCL Objective-C code I found references to "Cocotron", so I did a bit of research to see what that was all about. It turns out that this is an open-source project that aims to provide an Objective-C compiler and runtime environment for a multitude of platforms and OS types. It also has implementations of many of the same Cocoa classes and methods provided for the Macintosh, but as Gary Byers said in an email (12/3/09) to [openmcl-devel@clozure.com](mailto:openmcl-devel@clozure.com):

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

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

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

Update: As of July 2017, it looks like the last available Cocotron download was created in April 2011, so it hasn't had much activity in a while. See <http://code.google.com/p/cocotron/downloads/list>

## **Background Reading**

CCL provides very nice documentation that describes (among other things) their Objective-C bridge code. You can find it at:

<https://ccl.clozure.com/docs/ccl.html>

Apple's developer website has an enormous wealth of information that is constantly being updated. I find it useful to get the latest documentation and have it on my own system. You can do this by running Xcode and using the help facility to subscribe and download the documentation sets. I used to recommend the AppKiDo application, but it



seems to have stopped working although there are some indications that it might come back. In the meantime I have been using the documentation that you can get within Xcode. It's not nearly as easy to use as what was available within AppKiDo, but you take what you can get. You can also find all Apple docs at

<https://developer.apple.com/library/content/navigation/>  
but you pretty much need to know what you're looking for to use that.

If you implement my developer environment within CCL you will see a new choice under the Tools menu "Class Keywords and Binding Targets". Selecting this will bring up a class browser window that will show all Lisp and objective-C classes. That can help you to understand what classes are available to be used and from there you can look for corresponding Apple documentation on Apple's web site.

## Setting up your CCL environment

Since my CocoaInterface code is no longer bundled with CCL in a contrib directory, you will now have to download it from github:

<https://github.com/plkrueger/CocoaInterface>

If you do not already have an account you can sign up for one there.

Once you have the CocoaInterface code installed on your system, you need to tell the CCL IDE how to easily find files within it. You can do that within a ccl-ide-init.lisp file which the CCL IDE will look for in your home directory when it starts up. There are various plausible ways to set this all up and if you know what you are doing, feel free to set it up however you want. You can find a basic version of that within the CocoaInterface directory that you downloaded: ".../CocoaInterface/Developer Tools/ccl-ide-init.lisp". You can either copy that to your home directory or within the Terminal app you can create a symbolic link to that file:

```
[Paul-Kruegers-Mac:~] paul% ln -s /Users/paul/.../CocoaInterface/Developer\
Tools/ccl-ide-init.lisp /Users/paul/ccl-ide-init.lisp
```

where the ... represents the appropriate path to the CocoaInterface directory.

Note that you CANNOT create an alias for the file in the distribution and move that to your home directory. CCL will not follow that in an appropriate way for some reason when loading the init file.

This init file also depends on the CocoaInterface directory being in your home directory. If it exists somewhere else, you can either modify the init file to point to the appropriate location or create a symbolic link to the proper location within your home directory via the terminal app. For example:

```
[Paul-Kruegers-Mac:~] paul% ln -s /Users/paul/.../CocoaInterface /Users/paul/
CocoaInterface
```

where once again the ... represents the appropriate path to the CocoaInterface directory.

If you choose to load the application tools (via "require :install-app-tools" line) , which I recommend but is not absolutely necessary until later projects, then there are a few

dependencies on having a logical directory called “cocoa-pk” which is created in the example ccl-ide-init.lisp file shown below.

The app-tools will provide a “Class Keyword and Binding Targets” item under the Tools menu along with a whole bunch of utility code that will be used as we go along.

I do various other things in my init file, one of them being to initialize quicklisp, which I use for some things. I commented it out of the example file, but left it there to illustrate the other sorts of things you might want to do within your version of this file.

That file contains the following code:

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

;; Create a symbolic link to this file in your home directory
;; in the terminal app type: ln -s <path toCocoaInterface> /
Users/<user>/CocoaInterface

;; Note that creating an OS X alias will not work for reasons I
don't fully understand

;; Either put the CocoaInterface directory that you got from
github into your home directory or
;; create a symbolic link to it there or modify the code below
to point to wherever you put it.
;; Here we assume that there is a CocoaInterface directory or
symbolic link to it with the same name
;; within your home directory
;; Puts pathname translations for my user interface code into
search path so that we can just require it
(setf (logical-pathname-translations "cocoa-pk")
      '(("**;*.*" "home:CocoaInterface;**;*.*")))
(push (pathname "cocoa-pk:**;") *module-search-path*)

;; Load quicklisp
;; Something useful that I do that you might want
;; You need the quicklisp directory (or an alias to it) in your
home directory
#|
(require :setup "home:quicklisp;setup.lisp")
|#

;; Load and install the Developer tools
(require :install-app-tools)

(import 'iu::coerce-obj :common-lisp-user) ;; useful to have in
the listener
```

Now when you start up the CCL IDE it will take slightly longer and you will end up with some new menu items and an entirely new Dev menu that will only be used for advanced projects where you create stand-alone applications.

## Kick the tires

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

```
Welcome to Clozure Common Lisp Version 1.11-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 :simplesum)
:SIMPLESUM
("DEMO-PACKAGES" "SIMPLESUM")
? (ss:test-sum)
#<LISP-WINDOW-CONTROLLER <LispWindowController: 0x27059ff0>
(#x27059FF0)>
? (require :speech-controller)
:SPEECH-CONTROLLER
("RADIO-BUTTON" "SPEECH-CONTROLLER")
? (spc:test-speech)
#<SPEECH-CONTROLLER <SpeechController: 0x127c44c0> (#x127C44C0)>
? (require :package-view)
:PACKAGE-VIEW
("PACKAGE-VIEW")
? (pv::test-package)
#<LISP-WINDOW-CONTROLLER <LispWindowController: 0x28bc7f30>
(#x28BC7F30)>
? (require :loan-calc)
:LOAN-CALC
("LOAN-CALC")
? (lnc:test-loan)
#<LOAN-CONTROLLER <LoanController: 0x13f9fa90> (#x13F9FA90)>
? (require :loan-doc)
:LOAN-DOC
("LOAN-WIN-CNTRL" "LOAN-PR-VIEW" "LOAN-DOC")
?
```

The loan-document project has no test function per se. It adds the menu items “New Loan”, “Open Loan”, and “Print Loan” to the File menu in the CCL IDE that (along with existing “Save” menu-items) you can use to test it. See a much longer discussion for Project 7. Feel free to play around with them; it may help you understand what is happening when we get to the development details.

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

## **Objective-C Background Knowledge**

Objective-C is an object-oriented version of C that was and perhaps still is the predominant language used by Apple application developers. Swift is rapidly overtaking Objective-C as the most-used language. Almost all Apple documentation and example code provides both Objective-C and Swift examples. Once you understand the syntax variations you should be able to read examples in either language fairly easily and translate them to corresponding Lisp syntax if needed. Both languages use the same runtime, which is what is used by the Objective-C bridge code in CCL as well.

The Objective-C paradigm is that messages are sent to objects. This is largely equivalent to calling a method in Lisp. In fact, the CCL Objective-C Bridge code treats message sending as equivalent to calling a method where the first argument of every method is the object to which the message is to be sent and the message name is the name of the function. Since Objective-C message names are case sensitive and CCL generally turns everything typed into a listener into uppercase, a reader macro, `#/`, is provided to let you specify case-sensitive method names. At this point, if you have not already done so, I urge you to read the Objective-C bridge section of the CCL documentation:

<https://ccl.closure.com/docs/ccl.html#the-objective-c-bridge>

Actually the whole document is a pretty good read and contains lots of goodies that you should know about.

Since there are no packages in Objective-C as there are in Lisp, there is a single namespace for things like classes and methods. You can implement the same method for more than one class, but it must have the same signature each time (i.e. take the same number and types of arguments). You cannot have two different classes with the same name, even if you define them in code that is ostensibly in two different packages. You will get an Objective-C runtime exception if you try to do that.

You can define new classes that inherit from Objective-C classes. In such classes you can either add Lisp slots (specified in the normal way for Lisp classes) or you can create a subclass that contains Objective-C slots (as designated by the `:foreign-type slot-specification`). That may be convenient if you want to do things like binding between those slots and interface objects. However note that I have implemented binding to

normal lisp slots as well, so in practice I rarely create classes that specify :foreign-type slots.

One of the reasons for avoiding class definitions that include :foreign-type (i.e. Objective-C) slots is that you cannot dynamically redefine the set of Objective-C slots within an existing Objective-C class. This makes debugging in Lisp a trial since the only way to redefine such a class is to quit Lisp, start it up again, and then evaluate the new class definition. If you must use :foreign-type slots for some reason, it pays to get that definition correct as soon as possible. You can, on the other hand, dynamically redefine Objective-C methods and immediately use them and you can redefine Objective-C classes as long as you do not modify the set of :foreign-type slots within it.

You will often see object slots referred to "outlets" in Apple literature. An outlet is simply a special slot that contains a pointer to another object. Typically there are methods available to set the value of such slots. The use of the word "IBOutlet" in Objective-C code compiles to nothing. It is used as a hint to Interface Builder that it can arrange to connect that slot to some other object at runtime. For our purposes it will inform us of potential connections that we might want to take advantage of as well.

### **Warnings:**

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

```
;Compiler warnings for "ip:Simple Sum;simplesum.lisp.newest" :  
; In SIMPLESUM::I-[SumWindowOwner doFirstThing]: Undefined function  
NEXTSTEP-FUNCTIONS:IdoSum:I
```

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

Let's get started!

## **Project 1: Hello World**

For this example and for all others that follow, you can either start with my code and read about what I did, try to develop your own from scratch, or modify mine.

Key Concepts: Windows, Views, Constraints, Window-Controllers, MVC, Memory Management

We must begin by talking generally about Apple interface objects. Various Objective-C classes define each of these things and the nice thing for Lisp users is that the Objective-C bridge code in CCL lets you use each of them in almost exactly the same

way you would use a comparable Lisp class. The class hierarchies are completely integrated and you can even create new classes that derive from Objective-C classes. You can define new Lisp methods for any class and new Objective-C methods for any class that derives from an Objective-C class. It all works amazingly well.

*Cocoa* is a high-level set of interface objects and methods that is commonly used to construct interactive user interfaces. As you perhaps might expect, *Windows* are displayed on screens and windows contain a hierarchical arrangement of *Views*. Many of these views are standard sorts of things provided by Cocoa like labels and text boxes and buttons and tables, and many more. Others can be custom views that you create to display things in completely new ways. These examples will show you how to use the provided objects and create your own within Lisp.

A "Hello World!" application can be done pretty easily in almost all languages. I'm going to make this example a little more complex than is absolutely necessary in order to introduce concepts that you will find useful as we progress to more complicated interfaces.

The first such concept that I want to introduce is the Model / View / Controller (MVC) development paradigm that Apple promotes. This story is about the normal division of responsibilities that Apple suggests when using Cocoa for more complex interfaces and more complex data. Models are data models, Views are user interfaces, and Controllers map back and forth between the two. For very simple applications such as the "Hello World!" example that we'll do in a moment, it is typically possible to combine the Model and Controller functionality into a single object class. As we progress from project to project you will see a more distinct division of labor between models and controllers. If we want to present multiple views for a given set of data (perhaps with multiple windows), then this division becomes almost mandatory. As the data itself becomes even more complex, there may be reasons for creating multiple ways to organize it (confusingly these are referred to "views" in the database literature), thus adding another layer to the abstraction.

One such controller that we'll use immediately is the window controller object. Apple's documentation says that window controllers are responsible for the following:

- Loading and displaying the window
- Closing the window when appropriate
- Customizing the window's title
- Storing the window's frame (size and location) in the defaults database
- Cascading the window in relation to other document windows of the

application

For typical Objective-C applications window layouts are defined using Apple's Interface Builder. This is an interactive way to define windows visually and save their definitions into a file that can be loaded at runtime. Previous versions of this document relied on IB to define interfaces. While that is almost certainly still technically possible to do, recent releases of Apple's XCode have made that process increasingly difficult for Lisp users. On the flip side, Apple has introduced runtime API enhancements that make

programmatic layout of interfaces much easier. That combination led me to conclude that runtime definition and debugging of interfaces is now a better approach for Lisp users. So this is (almost) the last time I'll mention the Interface Builder and you can forget all about it.

I have created a special subclass of `ns:ns-window-controller` (which is the Lisp name that is the translation of the case sensitive Objective-C name `NSWindowController`) called `lisp-window-controller`. Objects of this type assume that the developer will supply a Lisp function of one argument that will be called to create a window and return it to the controller to be managed. It also returns a list of any interface objects that it needs to create for the window. This list may include both Lisp and Objective-C objects. The idea is that these are objects that must exist for the duration of the window. There are some subtleties that you will need to understand here.

The memory management scheme that is used by the Objective-C runtime (as used with CCL) uses reference-counting. When an object is created it is generally given a count of one and if that count ever goes to zero, the memory for the object can be reclaimed by the runtime. In general, when `make-instance` is called to create an Objective-C object, the returned object will have a reference count of 1, which means that the caller owns the object and must assure that the count goes to 0 when it is no longer needed.

A reference count is increased by calling `#/retain` on an object and decreased by calling `#/release` on an object. Failing to `#/release` an object when it is no longer needed generates a memory leak. That is generally not harmful unless you create a great many such objects and begin to use up available memory. Inappropriately calling `#/release` on an object that is then referenced may produce many sorts of catastrophic behavior. So when in doubt, don't `#/release`; and if you find that CCL is getting some sort of Objective-C runtime error, be suspicious of inappropriate `#/release` calls.

Sometimes a function will want to return an object with a reference count of zero to a caller and let the caller decide whether to `#/retain` it or not. But if you call `#/release` before returning the object, it will immediately be reclaimed. To make this possible, Objective-C provides an `#/autorelease` call. This places an object into a pool of objects that will have their reference counts decremented at some future time when the pool is emptied. This generally happens when each new event is processed in the system. So during the duration of processing one event, objects that have had `#/autorelease` called on them will remain in existence.

As we'll see later, objects that are returned by Objective-C class methods (i.e. methods that are called on the class instance itself as opposed to methods for instances of that class) often return objects that have been autoreleased. Callers that receive such objects must increment the reference count if they want to own them for a longer period of time than the current event and then decrease the count when they are done with them.

To be consistent with other `make-instance` return values, any view objects that are

returned when creating instances of my view objects have a reference count of 1. The various make-... utility functions that I provided also return objects with a reference count of 1. The caller is responsible for them and must arrange for them to eventually be released (by calling `#/release`). The most typical way to accomplish that is just to hand them to the window controller for the window that is being built and let it release them for you when they are no longer needed.

When we create Lisp subclasses of Objective-C classes new sorts of consideration are needed. Such objects may add either new Objective-C slots or new Lisp slots as part of the subclass. We'll see later how this works. While for the most part the CCL Objective-C bridge makes this entirely intuitive, there really are distinct Lisp and Objective-C pieces to any instance. For example, when we subclass an Objective-C view to make our own custom view we'll add the resulting objects to into the view hierarchy of a window in a normal way. The parent view for our custom view will do a `#/retain` call to increase its reference count and when it is done with it (often when the window closes) it will call `#/release` to make it go away. This assures that the object remains in existence as long as necessary. Any Lisp slots that are part of this instance are also guaranteed to stay in existence until the Objective-C part has been garbage collected by creating a permanent reference to those slots in a special hash table.

If nothing was done about those permanent slot references when the Objective-C object of which it is a part is garbage-collected, then the Lisp slots would never be garbage collected. To rectify that, a special `#/dealloc` method is automatically generated for every new Objective-C subclass that contains lisp slots. This method will remove the permanent reference to the object's lisp slots so that they may be garbage collected. You may be wondering what a `#/dealloc` method is. When an Objective-C object is about to be reclaimed, it's `#/dealloc` method is called to do any cleanup that is needed for a class. It is somewhat similar to the CCL `terminate` method that you can arrange to have called a class instance is about be garbage collected. This mechanism works quite well to maintain the existence of all slots (both lisp and Objective-C) for class instances that have both sorts of slots.

There is one little caveat however. If you find that you need to create your own `#/dealloc` method for such a class, then you will need to add in an appropriate call to remove those permanent instance slot references. Don't worry about how to do that right now; you will see an example of that later.

In the course of defining a window we will, in general, create many sorts of view, data, and controller objects that must remain in existence for as long as the window does. To make this memory management easy, we will adopt a process that simply hands all objects that are created as part of window definition to a window controller which will maintain references to them as long as the window remains open. The only thing that your code must do is make sure that there is a reference to that window controller object. The lisp-window-controller will assume responsibility for calling `#/release` on Objective-C objects when the window is closed so that they are garbage-collected by the Objective-C and Lisp runtimes as necessary. You'll see many examples of this process as we go along.



Let's start by demonstrating the "Hello World!" window that we'll be creating and then we'll go back and look at the Lisp code required to do that. Start the CCL IDE and follow along with the listener dialog shown below:

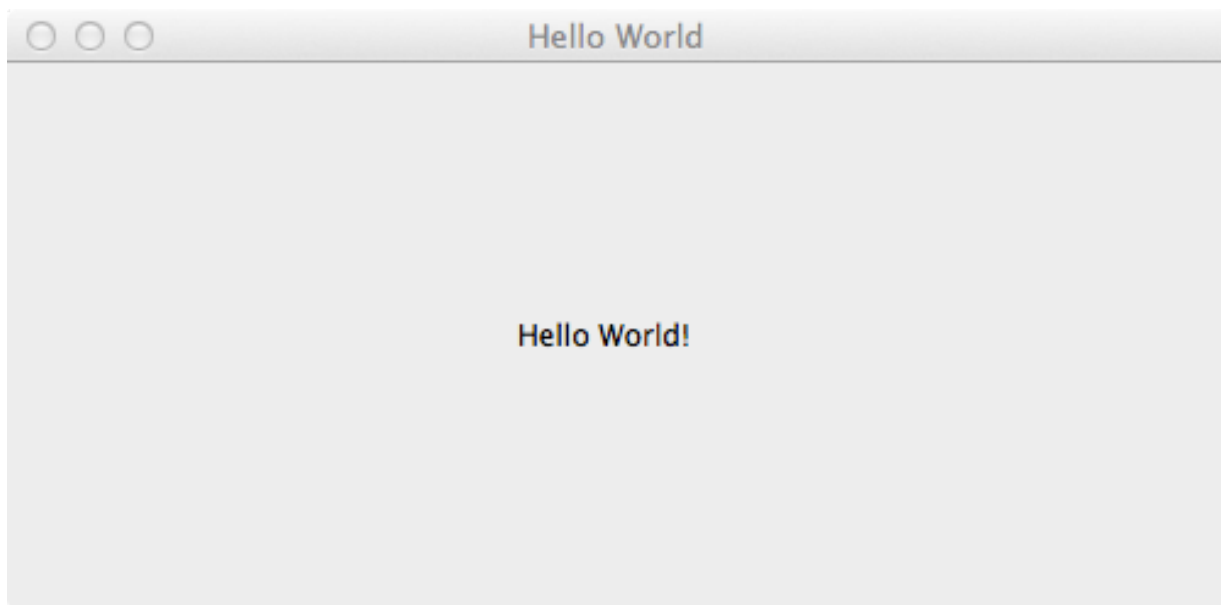
```
Welcome to Clozure Common Lisp Version 1.11-store-r16714
(DarwinX8664)!
```

```
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 :hello)
:HELLO
("DEMO-PACKAGES" "HELLO")
? (setf wc (hello::hello))
#<LISP-WINDOW-CONTROLLER <LispWindowController: 0x268a0990>
(#x268A0990)>
?
```

The output after the require may show additional packages loaded if you chose to exclude loading the app tools in your init file.

Your "Hello World" window will pop up and should look like the following:



Change the window size by the normal sort of click and drag from the lower right corner and observe the behavior. Note that the label stays in the center of the window, no matter how we change the size. That's because we constrained the view to be in that position. In a very simple way this demonstrates some of the power of Apple's constraint mechanism. It lets you define not just static view relationships, but dynamic relationships as well that automatically adjust what the user sees as various window parameters are changed. Let's see what sort of code was required for all this.

Open the file ...CocoaInterfaces/Example Projects/Hello World/hello.lisp. In this and all future examples in this document I have edited out some of the comments in the code where they don't contribute very much. Toward the top you will see the following:

```
;; hello.lisp

(eval-when (:compile-toplevel :load-toplevel :execute)
  (require :demo-packages)
  (require :window-utils)
  (require :window-controller)
  (require :text-views)
  (require :constraint-layout))

(in-package :hello)
```

Package definitions for all of the projects described here are in the demo-packages.lisp file. That file also requires the interface-packages.lisp file which defines all the packages used for all of my interface code. So if you're ever confused about what symbols are exported from my code, these two files will provide details. They are organized to make it easy to find where each symbol is defined as well.

I don't think it is necessary for you to understand all of the internals of window-controllers or any of the many other objects that are provided by my code, but it is useful to understand how to interface with them. For the most part, rather than provide a detailed API reference I will discuss new functionality as we go. Feel free to browse the lisp code to discover other functionality that might be useful to you. One exception to this is the layout constraint API that I created which is shown in Appendix B of this document.

Let's first consider the hello function because the make-hello-window function will make more sense after we do that.

```
(defun hello ()
  (let ((wc (make-instance 'lisp-window-controller
                          :build-method #'make-hello-window)))
    (show-window wc)
    wc))
```

This is just a function that we can use in the listener to create our window. The easiest

way to manage a window and assure normal behavior is by using an instance of `lisp-window-controller`, which is a subclass of `ns:ns-window-controller`. By default, these things really want to load a window from a NIB file that was created by XCode after using the InterfaceBuilder to design a window. As I indicated earlier, we aren't going to do that for any windows. Instead, we will define methods that create and return `ns:ns-window` objects. I created the `lisp-window-controller` subclass of `window-controller` specifically to invoke those window-building methods as needed. This inherits much of the default behavior of its parent class, but when it comes time to "load" a window definition, it instead calls a user-specified method to create the window. Much of what we will do throughout the remainder of this tutorial is define those window-building methods.

The `:build`-method specified for purposes of this project is called `make-hello-window`. We'll get to that method in just a bit.

The next thing called is `show-window` using the `lisp-window-controller` instance as its argument. This function is a simple `lisp` proxy for the Objective-C `#/showWindow:` method which could have been called directly. This method causes a window to be "loaded" if it is not already and then displayed. In our case, the `lisp-window-controller` will call its `build`-method rather than trying to load a NIB file. Other than that, things work pretty much the same as they do for a standard `window-controller`.

Finally our `hello` function returns the `window-controller` that was created. This may be useful in a debugging scenario because it gives you access to the window if you need to get at its views or anything else. Next let's turn to the method that builds our window.

The requirements for a window build method that is used with a `lisp-window-controller` are:

1. It must take a single argument that is the `lisp-window-controller` instance. It may not ever need this, but it is often useful to further subclass `lisp-window-controller` to make it react to other controls defined in the interface. Don't worry about that now, we'll see examples later on.
2. It must construct and return an instance of `ns:ns-window` as the first of two values returned. There are any variety of different subclasses that this might be.
3. It must return a list of objects that should be managed by the window controller. These are typically things like controllers that must respond to Objective-C messages from controls or view objects or data objects that are the source of the data shown by Objective-C views. They must exist for the duration of the window and can then be garbage-collected. Unlike `Lisp`, Objective-C (as used within `CCL`) must explicitly `#/retain` objects to keep them from being garbage-collected and `#/release` objects to allow that. So any Objective-C objects within the set of objects that is returned by the `build`-method will be released by the window controller when the window is closed. Again, we'll see examples of this in later projects.

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.

There is no precise order in which these tasks must be accomplished. Linking up objects (objective 6) is often accomplished as part of the initialization process for views. For this project, tasks 4, 5, and 6 are not required. We'll see more complicated examples later. These are virtually identical to the tasks that a window designer using Interface Builder would undertake. The difference, of course, is that IB provides a graphical interface to do much of this and in Lisp we will accomplish everything procedurally.

Here is the window build method for the "Hello World!" example:

```
(defmethod make-hello-window ((lc lisp-window-controller))
  (let* ((hello-label (make-instance 'label-view
                                     :title "Hello World!"))
         (win (make-instance 'ns:ns-window
                             :title "Hello World"
                             :resizable t
                             :content-subviews (list hello-label)))
         (cview (#/contentView win)))

    ;; constrain the size of the label to the minimum necessary
    (constrain-to-natural-size hello-label)

    ;; make sure the content view is always large enough to
    contain the label
    (constrain-size-relative-to cview hello-label :rel :>=)

    ;; align the center of the label with the center of the
    window in both x and y
    ;; directions
    (center-in-view cview hello-label)

    ;; Return the window, the objects created, and nil since we
    have added
    ;; no objects for the window-controller to manage.
    (values win (list win hello-label))))
```

The make-hello-window function creates a "Hello World!" label simply by making an instance of the label-view class. This is an ns:ns-text-field sub-class that I created which sets the attributes appropriately for a label. The class also lets you specify an orientation (:v or :h) and a background color for the label if desired. A ns:ns-window object is made using a normal make-instance with appropriate keyword arguments. We'll discuss how you know what keywords are available in just a bit.

It is never necessary to use my subclasses or front-end interfaces to create the view objects that you need. You can create your own or make direct Objective-C calls to create and initialize windows, views, and other interface objects. To make that process easier, I have added initialize-instance :after methods for many of the more common Objective-C classes. These take an extensive set of keyword arguments that permit you to initialize those objects in much the same way that you would initialize a normal lisp instance. You see an example of that in the make-instance call for the Objective-C ns:ns-window object. You can specify a title using a Lisp string and not worry about how it has to be converted into an Objective-C object or what method must be called to set that parameter.

You might be wondering at this point how you can find out which classes have such initialize-instance methods, what keywords are used for each one, and what sorts of arguments are acceptable for those keywords. Later we'll also want to know what binding targets are available for each class. I found myself asking those same questions over and over, so I made life a little easier for myself by creating a tool to display all that information. If you'd like to use it (and trust me, if you do much Cocoa development in Lisp then you will want to use it), then if you did not require :dev-tools-interface in your ccl-ide-init.lisp file, then in the listener do:

```
? (require :dev-tools-interface)
:DEV-TOOLS-INTERFACE
( "DEV-TOOLS-INTERFACE" )
```

After that you will find a new item titled "Class Keywords and Binding Targets" at the bottom of the Tools menu in the IDE. Selecting this will open up a window that presents a hierarchically arranged listing of all ns:ns-object subclasses on the left side of the window. Navigate to the class for which you want information or type any part of its name except the package into the search field at the top (make sure to hit return to engage the search) and select it. If any keywords are acceptable in make-instance calls for that class, then those keywords will be displayed in the table at the top, right. Information about what arguments are acceptable for each keyword can be found by expanding it. The table at the bottom right will display binding targets for instances of the selected class. Don't worry right now about why you want that; examples will be forthcoming. For the most part, all of this information was generated automatically by code that analyzes the source where the initialize-instance methods are defined. I have also provided methods for custom subclasses to add additional information that will be displayed for them. We'll see a few examples of that as we go along. You can use those same methods to provide documentation for custom Objective-C classes that you create.

Let's get back to the Hello World example. After we create the label-view object, we next make a `ns:ns-window` instance and specify that the label-view should be added to it. There are many other keyword options that you can provide to customize the window that you may want to check out. You can find out what those are by opening the "Class Keywords and Binding Targets" window, searching for `ns-document` and examining the keyword documentation that is available there. More details on these can of course be found in Apple documentation.

Once we have created the label and the window, we start task 3 by retrieving the window's content-view. This is the top-level view within the window. By default it is constrained to be the size of the window's main viewing area. If the user changes the size of the window, the content-view's size is changed accordingly. Generally speaking, we will add constraints to the content-view to assure that its layout is done the way we want.

The first thing we do to add constraints to the window is:

```
(constrain-to-natural-size cview hello-label)
```

This just tells the constraint system that it should constrain the size of the label to whatever it thinks is its natural minimum size and then add that constraint to the window's content view. Why is this needed? One of the quirky things about the constraint system is that Apple has worked very hard to make it backward compatible with previous layout methods. So by default it will turn various view specifications into constraints when the runtime system is trying to determine view size and position. The problem with this is that such constraints will typically conflict with the more dynamic constraints that we will create. Therefore Apple has provided a method to turn off the automatic constraint conversion for a view. It's easy to forget to do this when creating a view that you will later want to constrain. Indeed you might not even know that you want to do this when that view is created. If you watch Apple's developer videos about how to debug constraints you'll see this is a common sort of problem.

To make life easier, my constraint-creation methods will automatically turn off automatic constraints for any view that is otherwise constrained. This ability to turn off automatic constraints is an all-or-nothing thing which means that if you turn them off, then you must constrain both the size and the position of that view by one means or another. The constraint system will generally try to make views as small as possible. There is a view function called `#/fittingSize` which the constraint system will use to determine the smallest size for any object. Unfortunately I've found this to be a little hit-and-miss. For some types of views this works exactly as you would expect and for others, the minimum size is 0 in both dimensions. Unless you provide your own constraints on the size of such things, they will not be drawn in the window (i.e. they will be invisible). Specifying relative positions is often enough to also specify a size, but it may not be, in which case other size-specific constraints are required. Calling `constrain-to-natural-size` assures a minimum size for any view. If a view you end up creating disappears, then making this call on that view may help you out. If you create your own custom view classes, you can add constraints to those views which result in the `#/fittingSize` function returning a reasonable value; in which case it will not be necessary to use `constrain-to-`

natural-size. Constrain-to-natural-size tries to honor whatever size might have been specified for the view when it was created or any natural size (such as the length of the string for labels). So it goes a bit beyond what Apple's constraint system does by default. Note also that constrain-to-natural-size will do nothing if it determines that the object in question has its own internal constraints. It assumes that those constraints will completely determine the size of the object. If this happens, a console log entry is generated to inform the developer that the constrain-to-natural-size call was ignored. You can use Apple's Console application to view such messages.

The constrain-to-natural-size method takes a keyword argument, :rel, which takes one of the values := :>= or :<=. The default is :=. This results in constraining the view to be exactly equal to the natural size. If instead you used the :>= key, you would constrain the view to be at least as big as the natural size in both dimensions. The default is appropriate for labels, but :>= is sometimes a better choice to achieve a particular behavior when a window is re-sized.

Let's take a little diversion here to talk about constraints in general. Apple has defined window layout functionality that operates by using constraints that have been placed on view objects to dynamically determine both size and position of view objects within a window. If you're very curious, you can dig into Apple's documentation for this, but I suspect you'll find that the lisp API I have created to those constraint functions is easier to understand and use. See Appendix B for a complete description of that API.

What you need to understand here is that every layout constraint specifies an equality that would be described in lisp terms as:

```
(<rel> (<property> <object1>)
      (+ (* (<property> <object2>)
            <multiplier>)
          <constant>))
```

where:

```
<rel>      ::=    = | >= | <=
<property> ::=    top | left | right | bottom | leading |
trailing | width |
                                height | center-x | center-y | baseline |
none
<object1>  ::=    <NSView object>
<object2>  ::=    <NSView object> | <null object>
<multiplier> ::=  <number>
<constant> ::=    <number>
```

I found this lispish way of thinking about what a constraint can be to be so useful that I created the "constrain" macro which takes a list in the above form and turns it into an appropriate Objective-C call to make a constraint for you. For example, you could say something like:

```
(constrain cview (<= (leading cview) (leading arg1)))
where cview and arg1 are variables defined in the local environment which evaluate to
```

view objects. Note that the example does not include the + or \* clauses. The constrain method is flexible enough so that any input that can be put into the form above will automatically be converted. You can exchange the order of top level arguments or reduce the clauses included however you want and constrain will convert appropriately. You can also use + or / operations. As long as the form can be converted to the one above, the macro will do so. Note that there are many things you might want to do which cannot be converted into this form. If you try to do so, you will get various sorts of errors. You will see LOTS of examples of this as we go forward and I'm sure that will help to make this more clear. To create individual constraints, (constrain ...) is the most convenient interface method to use.

In addition to specifying layout constraints such as these, we can also specify how resistant a view is to expansion of its current size and how resistant it is to compression from its current size. This helps the layout system decide what to do. I have found that these priority constraints are not often needed, but once in a while they are useful.

Finally constraints themselves can have a priority ordering. The constrain macro will also take a keyword argument which specifies that priority. Priorities can be in the range 0 to 100, where 100 means the constraint is mandatory. By default the priority is 100. So far, I have found that the ability to set constraint priorities is not typically needed.

From this relatively limited form a surprisingly large number of relationships can be specified. On top of this basic form, I have added the methods described in detail in Appendix B to provide new constraint idioms that hopefully make the process easier. In addition to creating constraints, these methods immediately add the constraints to a parent-window that is passed in as the first argument to each method. That view can also be one of the views constrained, as when a view is constrained to have some relationship relative to its container view. You may want to browse through Appendix B to see all of the constraint idioms provided.

OK, back to Hello World again. The next thing the hello.lisp code does is:

```
(constrain-size-relative-to cview cview hello-  
label :rel :>=)
```

This makes sure that the size of the window is at least as big as the size of the label. If you make the window smaller, you will note that eventually you cannot reduce it any further. That happens as a result of this constraint. Try removing it from the function and then see how the behavior changes when the size of the window is reduced.

The final constraint added to the window is:

```
(center-in-view cview hello-label)
```

This aligns the center of the label-view with the center of the content view in both dimensions. You will see the effect of this constraint when you resize the window. As you might expect, the label stays centered in the window. Experiment by removing one or both of these constraints to see what effects you get. You may also try other forms of constraints to get other behaviors.



In the final task, the `make-hello-window` method returns the two required values: the window it created and a list containing the `hello-label` and the window itself. When the `hello-label` was added as a subview of the window's content view the content-view would have `#/retain'`ed it, so we really don't need to worry about it going away. We could have called `#/release` or `#/autorelease` on the label to reduce its reference count after it was added to the window. But there is no harm in letting the window controller do this after the window is closed, so I've made life easier for myself by just giving most every object I create for a window to the window controller and not worrying about releasing them in the window build method. But its your choice as to how you handle this memory management. Note that the window itself is an object that must be managed, just like any other view that is created. And just as for other views that you create, you can `#/autorelease` it when you create it rather than returning it in the list of values to be managed. Be certain that you don't both `#/autorelease` it AND include it in the list to be managed. If you did so, then when the window is closed, the management code would try to release it again and releasing any object that already has a reference count of zero will result in a system exception.

And that concludes our discussion of the first project. Congratulations on making your first window!

## Project 2: simplesum

Key Concepts: Objective-C subclasses within Lisp, foreign slots, `NSTextField`, `NSButton`, actions and targets, Lisp action functions

In this project we'll create a fairly simple interface that lets a user enter two numbers and click a button to add them and display the result in the window. We will create the window in much the same way that we did for the "Hello" project and then link everything up so that it functions properly. Even with this simple example there are lots of little things to learn about how this all works.

Let's begin as we did for the first project by running the example first to see what the final objective is. In the CCL IDE listener do the following:

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

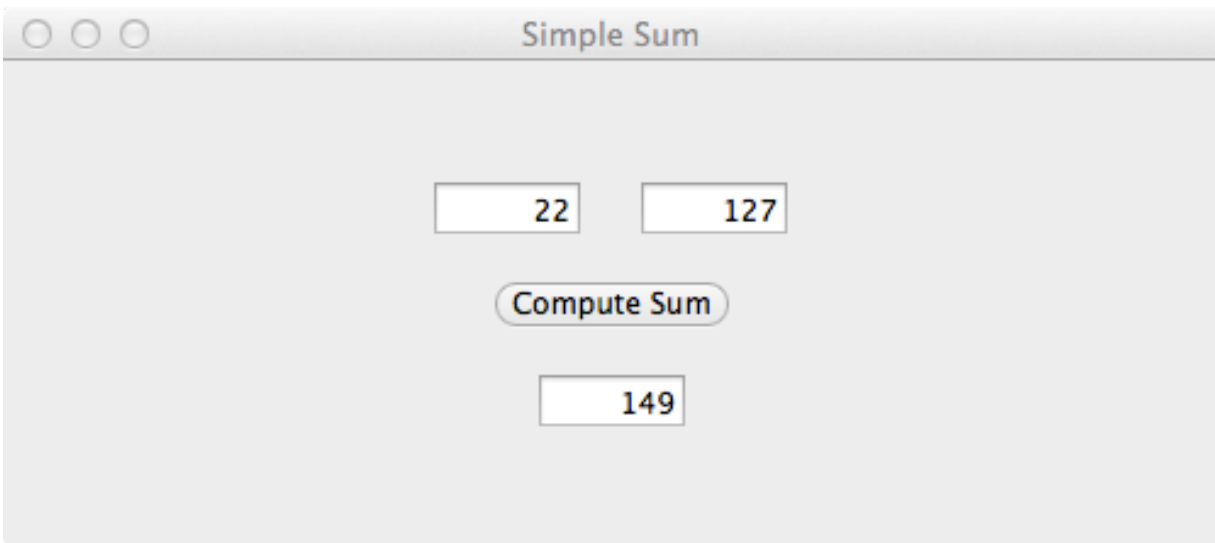
```
CCL is developed and maintained by Clozure Associates. For more
information
about CCL visit http://ccl.clozure.com. To enquire about
Clozure's Common Lisp
consulting services e-mail info@clozure.com or visit http://www.clozure.com.
```

```
? (require :simplesum)
:SIMPLESUM
```

```
( "INTERFACE-PACKAGES" "DEMO-PACKAGES" "NS-STRING-UTILS" "ASSOC-
ARRAY" "NS-BINDING-UTILS" "UNDO" "KVO-SLOT" "NSLOG-UTILS"
"TAGGED-DATES" "DATE" "ALERT" "DECIMAL" "ATTRIBUTED-STRINGS"
"NS-OBJECT-UTILS" "BUTTON-VIEW" "WINDOW-UTILS" "CONSTRAINT-
LAYOUT" "TEXT-VIEWS" "LISP-WINDOW-CONTROLLER" "WINDOW-
CONTROLLER" "SIMPLESUM")
? (setf wc (ss:test-sum))
#<LISP-WINDOW-CONTROLLER <LispWindowController: 0x2377e350>
(#x2377E350)>
```

?

This will pop up a simple-sum window. Enter integer values into the two fields at the top, then hit the "Compute Sum" button and observe the behavior. It should look much like the following:



Try resizing the window to see what happens. Look at what happens if you type non-integer numbers or alphabetic characters in the input fields. This is a pretty bad design in several ways, but serves to illustrate several new ideas. We'll see much better ways to do everything in future projects.

The source code for this project is in ...CocoaInterfaces/Example Projects/Simple Sum/simplesum.lisp. The test-sum function is, much like the hello function we saw in the first project:

```
(defun test-sum ()
  (on-main-thread
    (let ((wc (make-instance 'lisp-window-controller
                           :build-method #'make-sum-window)))
      (show-window wc)))
```

```
wc)))
```

There is one little wrinkle here worth noting. You will see the body of the function enclosed in a form beginning with `on-main-thread`. This forces everything enclosed to be executed on the main thread of the application. All event handling happens there anyway. There are many sorts of Cocoa-related operations that can only be done safely on the main thread (i.e. they are not thread-safe). Anything that alters things like lists of constraints, for example, must be done on the main thread. If you look more deeply at the code I've provided, you may see these scattered around in various places. I've also observed that sometimes problems only show up when code is executed on the main thread. So having this call in a test function may uncover problems that would not show up if it were executed on a listener thread.

We will design this window to operate as follows:

- The two fields at the top will be used to enter numbers
- The field at the bottom will display the sum of the two numbers
- The "Compute Sum" button will cause the new sum to be computed and put into that field.
- The button will stay in the center of the window and all other objects will maintain their position relative to the button.

The `make-sum-window` method is this:

```
(defmethod make-sum-window ((lc lisp-window-controller))
  (let* ((arg1 (make-instance 'ns:ns-text-field
                             :alignment :right
                             :frame-size '(60 21)))
         (arg2 (make-instance 'ns:ns-text-field
                             :alignment :right
                             :frame-size '(60 21)))
         (sum (make-instance 'ns:ns-text-field
                             :alignment :right
                             :frame-size '(60 21)
                             :selectable nil))
         (box (make-instance 'organized-box-view
                             :views (list arg1 25 arg2)))
         (data-holder (make-instance 'sum-window-data
                                     :input1 arg1
                                     :input2 arg2
                                     :sum sum))
         (sum-button (make-instance 'ns:ns-button
                                   :title "Compute Sum"
                                   :bezel-style :round-rect
                                   :button-type :momentary-push-in
                                   :target data-holder
                                   :action "doSum:"))
         (win (make-instance 'ns:ns-window
```

```

        :title "Simple Sum"
        :resizable t
        :content-subviews (list box sum sum-button)))
(cview (#/contentView win)))

;; constrain the size of the fields to the minimum necessary
;; we don't want them changing in response to changes of the
window size
;; arg1 and arg2 are already constrained when we put them in
the invisible box
(constrain-to-natural-size sum)
(constrain-to-natural-size sum-button)
;(constrain-to-natural-size box)

;; Put the button in the center of the window
(center-in-view cview sum-button)

;; Vertically align the box, the button, and the sum field
(order-views :orientation :v
              :views (list box 20 sum-button 20 sum)
              :align :center-x)

;; make sure the content view is always large enough to
contain all the objects
(constrain (>= (leading box) (+ (leading cview) 20)))
(constrain (>= (trailing cview) (+ (trailing box) 20)))
(constrain (<= (top cview) (- (top box) 20)))
(constrain (>= (bottom cview) (+ (bottom sum) 20)))

;; Return the window, the list of objects created
(values win (list arg1 arg2 sum box data-holder sum-button
win))))

```

Let's break this down to the tasks needed by window build method. For task 1 we create all the subviews needed. There are four obvious ones in the three text fields and the button. For the ns-text-field instances we specify right alignment because that looks more natural for numbers and also provide a frame size for each field. We make it impossible for the user to type something into the sum field by specifying that is not selectable.

We next come to the box variable in the let form. It would certainly be possible to constrain the layout for the two upper text views to achieve the look of the window shown above. We could, for example, constrain the top two objects by constraining their trailing and leading edges (respectively) with respect to the center-x attribute of the button. But one technique that I've found to be useful is to box up a group of objects within an invisible box and then constrain the box. This can dramatically shorten up the overall number of constraints needed. While it doesn't save us a lot of work here, it's

worth doing just to gain an understanding of the technique. So for this example, we'll box up the two input text views. By default, an organized-box-view arranges its objects horizontally and aligns their centers. If we wanted a vertical arrangement, or a different alignment, then we could have added different keyword arguments to accomplish that. The box will constrain those text views to be their natural sizes and also constrain *itself* to be the natural size required to contain those objects. This is a useful property because otherwise the constraint layout system might decide to expand the size of the box in order to meet some constraint and that might mess up our intentions. Don't worry right now about how organized-box-views work. When we get to the project 4 I'll take a look at the internals of that object and others while discussing the creation of custom views.

We'll come back to the data-holder object in just a bit.

Next we select various initialization arguments to get the sum-button to look and act as we desire. Supplying `:title`, `:bezel-style`, and `:button-type` arguments are very commonly used when creating buttons. Two other parameters that will frequently be used are the `:target` and `:action` keywords. They specify what happens when the button is pushed. All `ns-control` objects (which includes buttons and sliders and text fields and others) can be configured to send a message (as specified by the `:action` parameter here) to some other object (specified by the `:target` parameter). When the button is pushed, the action message specified will be sent to the target specified. The action message will be in the form of a call to an Objective-C method of the same name that the target must have implemented. That function must take one argument which will be the sender of the action. So in theory you could have several buttons send the same message to the same target and distinguish which button sent the message by examining that sender argument. The action-msg must be a case-sensitive string that will be converted into an appropriate Objective-C selector.

There are some simple naming and parameter ordering conventions that you should also understand. An Objective-C message always starts with a lower-case letter (as opposed to class names that always start with an upper-case letter). Where Lisp names would often add "-" for readability, Objective-C names will add an upper-case letter. Objective-C names will also indicate the parameters that can be passed by including the name of each parameter followed by a ":" as part of the name. For further information about Objective-C naming conventions and their conversion to Lisp, I'd suggest a review of both Apple and CCL documentation which discuss that process in more detail.

The Objective-C paradigm is that a message and its parameters (if any) are passed to an object. That is internally exactly the same as calling a method with the same name as the action message with the intended receiver as the first argument and the message parameters as subsequent arguments. This makes translation from Apple documentation to a corresponding Lisp call trivially easy. For example, the Apple message

`addSubview:positioned:relativeTo:`

can be sent to `NSView` objects (the name of that class as translated to Lisp would be `ns:ns-view`). You could send that message to an `ns:ns-view` object, call it `v`, by doing the

following in Lisp:

```
(#/addSubview:positioned:relativeTo: v <subview> <position> <other-subview>)
```

The ":"s in the name tell you what parameters are needed in addition to the first (target) argument. Apple's documentation provides information about the nature of each of those parameters. In many cases, the Lisp Cocoa bridge code will automatically convert Lisp data into an appropriate data object for the call. For example, if the call required a boolean value, you can pass in t or nil as an argument and it will automatically be converted for you. In other cases where objects of a particular type are required you must create the appropriate Objective-C object.

I have created a very large assortment of functions that will help you convert between Lisp and Objective-C objects (either direction), but gathered them all under the umbrella of a single "coerce-obj" method. In general you can call (coerce-obj <object> <type>) and the appropriate translation will be done. Making a call of the form:

```
(coerce-obj <objective-c object> t)
```

will convert an objective-c object to the most appropriate lisp counterpart. If there is no obvious translation possible, then the original objective-c object is returned. Similarly:

```
(coerce-obj <lisp object> 'ns:ns-object)
```

will convert any lisp object to an objective-c object. These general conversions may not always be what you want, so in general you should use the most specific type that you can.

Let's go back to the action message "doSum:" that we specified when we made the button. You now understand that it will take one argument because of the single ":" in the name. But we didn't name the message in a way that makes it obvious what that parameter might be. In fact, the action message for buttons and other forms of controls sends a pointer to the sender along with the message. So a better name might have been something like "sumForSender:". Although that would be more consistent with Apple's naming conventions, I've discovered that even Apple developers name these action messages much as I did here.

The next obvious question is where that message should be sent. The biggest limitation on that choice is that the target object must be a subclass of ns:ns-object. So the window-controller is one obvious choice. We could, for example, create a subclass of the lisp-window-controller for each project that also responds to action messages from view controls. We'll do something like that in future projects. In this case I decided to create an entirely new class to contain the data and respond to the message from the button. I've named that class sum-window-data. We'll come back to exactly what that object does in just a second. The data-holder variable is initialized in the let form to be an instance of that class and when we make the button we set up that object as the target of the "doSum:" message.

Next (task 2) we create the window, specifying that it should contain the box, the button, and the sum field. It is not necessary (in fact wrong) to include the arg1 and arg2 fields since they are already subviews of the box.

Next (task 3) we need to specify view constraints for the window. To do that we first

retrieved the window's content view because that is the parent view for everything to be added. So that will also be the view for which we will set all the constraints. Let's look at those constraints. The first three are:

```
(constrain-to-natural-size sum)
(constrain-to-natural-size sum-button)
```

These should be familiar to you. They just restrict the sum and sum-button fields to their natural sizes since we are not going to specify constraints that would cause them to be resized. We do not need to do this for the box object because it already constrains itself to be its natural size. Next comes:

```
(center-in-view cview sum-button)
```

which, as you might expect, constrains the button to be right in the middle of the content view. This is one of the constraint idiom methods that I provided. This is a fairly simple one that merely aligns the centers (both x and y) of the two views.

After that we:

```
(order-views :orientation :v
              :views (list box 20 sum-button 20 sum)
              :align :center-x)
```

This will both order the views in the vertical direction and align them as specified. The ability to align objects that you order is so common that providing an align option at the same time was the most sensible thing to do. But if you do NOT want to align objects that you order, then simply do not include an :align argument and no alignment will be done. One of the things to understand about constraint ordering is that in the vertical direction the top is always "less" than the bottom. This is true independent of the actual graphic coordinate system used to draw that view. In Cocoa, a coordinate system for any view can have the origin (0,0) be either in the lower-left corner or the upper-left corner. Constraints are always specified as if the coordinate system is in the upper-left. Any translation needed to accommodate the actual coordinate system is done automatically by the constraint layout system. This seems like something of an odd choice to me since the default coordinate system puts the origin at the lower-left, but that's the way it is.

Finally we constrain the content-view itself so that it will always be large enough to contain all of our objects. We can do that easily here because we know which objects will be at the edges.

```
(constrain (>= (leading box) (+ (leading cview) 20)))
(constrain (>= (trailing cview) (+ (trailing box) 20)))
(constrain (<= (top cview) (- (top box) 20)))
(constrain (>= (bottom cview) (+ (bottom sum) 20)))
```

These calls make use of the `constrain` macro which adds a single constraint that is specified as a Lisp form.

We have no data controllers, so we'll move on to task 5 and talk about the data-source object that we created by the `let` clause:

```
(data-holder (make-instance 'sum-window-data
                            :input1 arg1
                            :input2 arg2
                            :sum sum))
```

The `sum-window-data` class definition is:

```
(defclass sum-window-data (ns:ns-object)
  ((input1 :accessor input1
           :initarg :input1)
   (input2 :accessor input2
           :initarg :input2)
   (sum :accessor sum
        :initarg :sum))
  (:metaclass ns:+ns-object))
```

We made this a subclass of `ns-object` so that it will be able to accept Objective-C messages. The three slots will contain Objective-C objects, namely the two views that are used for input and the view that is used to display the sum. But observe that we can specify normal slot accessors and `initargs` for each slot. The Cocoa bridge code in CCL is quite happy to treat these objects just as it would any other class instance.

We also define a method that is invoked when the "doSum:" message is sent to this object (when the sum button is pushed):

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

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



No return value is required so the return value is specified as `:void`. The method is defined for the `sum-window-data` class so that the first argument will be an instance of that class. The second argument, that we called `s`, will be a pointer to the sending object. The type of this argument is defined to be a generic Objective-C pointer by using the `:id` keyword type. Since we won't be using this argument we can ignore it.

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

When the button is pushed, this method will be invoked. It calls Objective-C methods to get the integer values from the inputs and set the integer value of the sum field. Note that Lisp automatically converts the Lisp sum to an appropriate data type for passing as an Objective-C integer argument.

An astute programmer might be wondering what happens when there isn't anything in those input fields. What will the `"/intValue` calls return? Initially I put all sorts of additional code here to initialize the fields to 0 if they were blank. But after experimenting a bit I discovered that what you get in that case is 0 so it was all unnecessary. Cocoa does lots of things that make life easy and this is just one small one. Nevertheless, as I indicated earlier, this isn't an especially good design. We would really prefer to make sure that users are only able to enter acceptable arguments in the first place. In later projects you'll see how to make that happen.

So an instance of `sum-window-data` becomes the data source for this window and we have completed task 5 for our window build method. For task 6 we need to link together all the fields. We did much of that when we initialized the data source and completed the process when we initialized the button. All of the view and data objects are now appropriately linked together.

The final stage is just to return the window and a list of Objective-C objects that we want the window-controller to manage. In this case we will return a list of all the objects we created.

Project 2 is complete.

### **Project 3: menus**

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

There are times when we would like to add our own menu to the CCL IDE menubar or modify menus that already exist. If you loaded the `dev-tools-interface.lisp` file in order to

see the Class Keywords and Binding Targets window, then you've already seen one example of that. That's what we'll explore further in this project. In later projects we'll create stand-alone applications that have their own menus and explore how to switch between CCL and application menus while testing under the IDE.

You may already have noticed that the windows we created for the first two projects will automatically respond to existing menu choices whenever they are supported. For example, you can select "Close" from the File menu to close a window. How exactly does that work? All windows and view objects within them are organized as a hierarchical set of containers. Every time the application user clicks on the screen somewhere the lowest-level user interface element at that location is given the opportunity to become the first responder to that click event. If it chooses to decline the responsibility then it's superview is given the opportunity and so on up the line until some object accepts. This chain of responsibility, called the Responder Chain, is dynamically redefined every time a click is made. Why do we care about this? There are many reasons, but in this case it is because the action messages sent by menu items are often targeted at whichever object is currently the First Responder. If it has a method defined for that action, it performs it. If not, the message is passed up the responder chain until some object does respond to it.

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

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

What we're going to do here is dynamically add a menu with sub-menus to the menubar and then set one of our objects to respond to one of those menu items. We'll do that by making minor additions to the last project. I first cloned `simplesum.lisp` and renamed it `menu-test.lisp`. I then renamed the Objective-C class from `sum-window-data` to `sum-data` to avoid a collision with the class defined for the previous project. As I mentioned earlier, you cannot have two different Objective-C classes with the same name since there is only a single namespace in Objective-C.

The source code for this project is in `...CocoaInterfaces/Example Projects/Menu/menu-test.lisp`.

I modified the test function to look like the following:

```

(defun test-menu ()
  (on-main-thread
    (iu::make-and-install-menu "New App Menu"
      '("Menu Item1" "doFirstThing:")
      '("Menu Item2" "doSecondThing:"))
    (let* ((wc (make-instance 'lisp-window-controller
      :build-method #'make-sum-window)))
      (show-window wc)
      wc)))

```

When you execute this in the listener by typing (mt:test-menu) a new menu titled "New App Menu" will be added to the right end of the menubar with two sub-items. The first, "Menu Item1", sends the action message "doFirstThing:" when it is selected and the second, "Menu Item2" sends the action message "doSecondThing:" when it is selected. These are sent to whatever object is currently the first responder if, and only if, it can respond to that message. When you select any window other than our new "Simple Sum" menu, you will see that those menu items are disabled because nothing in the current responder hierarchy would know what to do with them. But if you first select the new "Simple Sum" window, you will find that the first menu item is now selectable. Type a couple of numbers into the input fields on the new window and then select "Menu Item1" from the "New App Menu". Note what happens in the window's sum field. Let's take a look at how we made that happen.

The build method is named "make-sum-window". That method will look pretty familiar to you if you worked through the last project. Here is is:

```

(defmethod make-sum-window ((lc lisp-window-controller))
  (let* ((arg1 (make-instance 'ns:ns-text-field
    :alignment :right
    :frame-size '(60 0)))
    (arg2 (make-instance 'ns:ns-text-field
    :alignment :right
    :frame-size '(60 0)))
    (sum (make-instance 'ns:ns-text-field
    :alignment :right
    :frame-size '(60 0)
    :selectable nil))
    (box (make-instance 'organized-box-view
    :views (list arg1 25 arg2)))
    (data-holder (make-instance 'sum-window-data
    :input1 arg1
    :input2 arg2
    :sum sum))
    (sum-button (make-instance 'ns:ns-button
    :title "Compute Sum"
    :bezel-style :round-rect
    :button-type :momentary-push-in

```

```

        :target data-holder
        :action "doSum:"))
(win (make-instance 'ns:ns-window
    :title "Simple Sum"
    :resizable t
    :content-subviews (list box sum sum-button)))
(cview (#/contentView win)))

;; constrain the size of the fields to the minimum necessary
;; we don't want them changing in response to changes of the
window size
;; arg1 and arg2 are already constrained when we put them in
the invisible box
(constrain-to-natural-size sum)
(constrain-to-natural-size sum-button)
(constrain-to-natural-size box)

;; Put the button in the center of the window
(center-in-view cview sum-button)

;; Vertically align the box, the button, and the sum field
(order-views :orientation :v
    :views (list box 20 sum-button 20 sum)
    :align :center-x)

;; make sure the content view is always large enough to
contain all the objects
(constrain (>= (leading box) (+ (leading cview) 20)))
(constrain (>= (trailing cview) (+ (trailing box) 20)))
(constrain (<= (top cview) (- (top box) 20)))
(constrain (>= (bottom cview) (+ (bottom sum) 20)))

;; Return the window, the list of objects created
(values win (list arg1 arg2 sum box data-holder sum-button
win))))

```

There is only one small difference between this method and its counterpart from the previous project and that is in the make-instance call for the window. We added one argument:

```

:delegate data-holder

```

As you might expect, this sets up the sum-data object as the window's delegate. What that means is that this object will now be in the responder chain for any click within our window. So let's go back to our Lisp code and add a method that will respond to the action method "doFirstThing" that we defined for the first menu item:

```

(objc:defmethod (#/doFirstThing: :void)
    ((self sum-data) (s :id))

```

```
(declare (ignore s))
;;; test function for menu tests
(#!/doSum: self (%null-ptr))
```

This simple function just calls the same action function that is invoked by clicking on the "Sum" button, namely it adds the two input values and displays the new sum. By virtue of having this method and being the window's delegate, "Menu Item1" will be enabled whenever the current responder is our new window.

If you want to remove the "New App Menu" from the menubar you can do the following in the listener window:

```
(iu::uninstall-menu "New App Menu")
```

which will remove it.

There is also a second test function defined in menu-test.lisp:

```
(defun test2-menu ()
  (on-main-thread
    (iu::make-and-install-menuitems-after "File" "New"
      ' ("New myDoc"
        "newMyDoc" ) ) ) )
```

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

Now I want to take a little closer look at the menu functions we've used here. In future projects we'll talk about several others and you can always take a look at menu-utils.lisp to see what's going on. Just by way of background, there is a bit of complexity to adding new menus and sub-menus, but thanks to Apple's sample code (Objective-C of course) it turned out to be not all that difficult. I suppose the key things to understand is that there are NSMenu objects and NSMenuItem objects. NSMenu's may only contain NSMenuItem's never subordinate NSMenu objects. NSMenuItem's may contain a single subordinate NSMenu object. So in order to create a new menu, put it into the menubar (which is just the main menu), and have subordinate menu items we must create a new NSMenuItem that we make subordinate to the main NSMenu object, then make a new NSMenu object subordinate to the new NSMenuItem we created, and finally add a series of subordinate NSMenuItem's to our new NSMenu. It might be easier to see what the code is doing than it is to parse that last sentence! So if you're curious, take a look at menu-utils.lisp.

The make-and-install-menu function has the following signature:

```
make-and-install-menu (menu-name &rest menu-item-specs)
```

A menu-item-spec is a list of the form:

```
(menu-item-name menu-item-action menu-item-key-equivalent
```

```
menu-item-target)
```

where the first three are all strings and the last must be an object that will be the target of the menuitem's action message when it is selected. Normally the target is left nil and the message is sent up the first responder chain, but for some menuitems that we will create later we want it to be sent to a specific target that we will specify.

I also created the function `make-and-install-menuitems-after` to install new menuitems at a specific

location within an existing menu. This will be used in Project #7. It's signature is:

```
make-and-install-menuitems-after (menu-name  
                                  menu-item-name  
                                  &rest menu-item-specs)
```

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

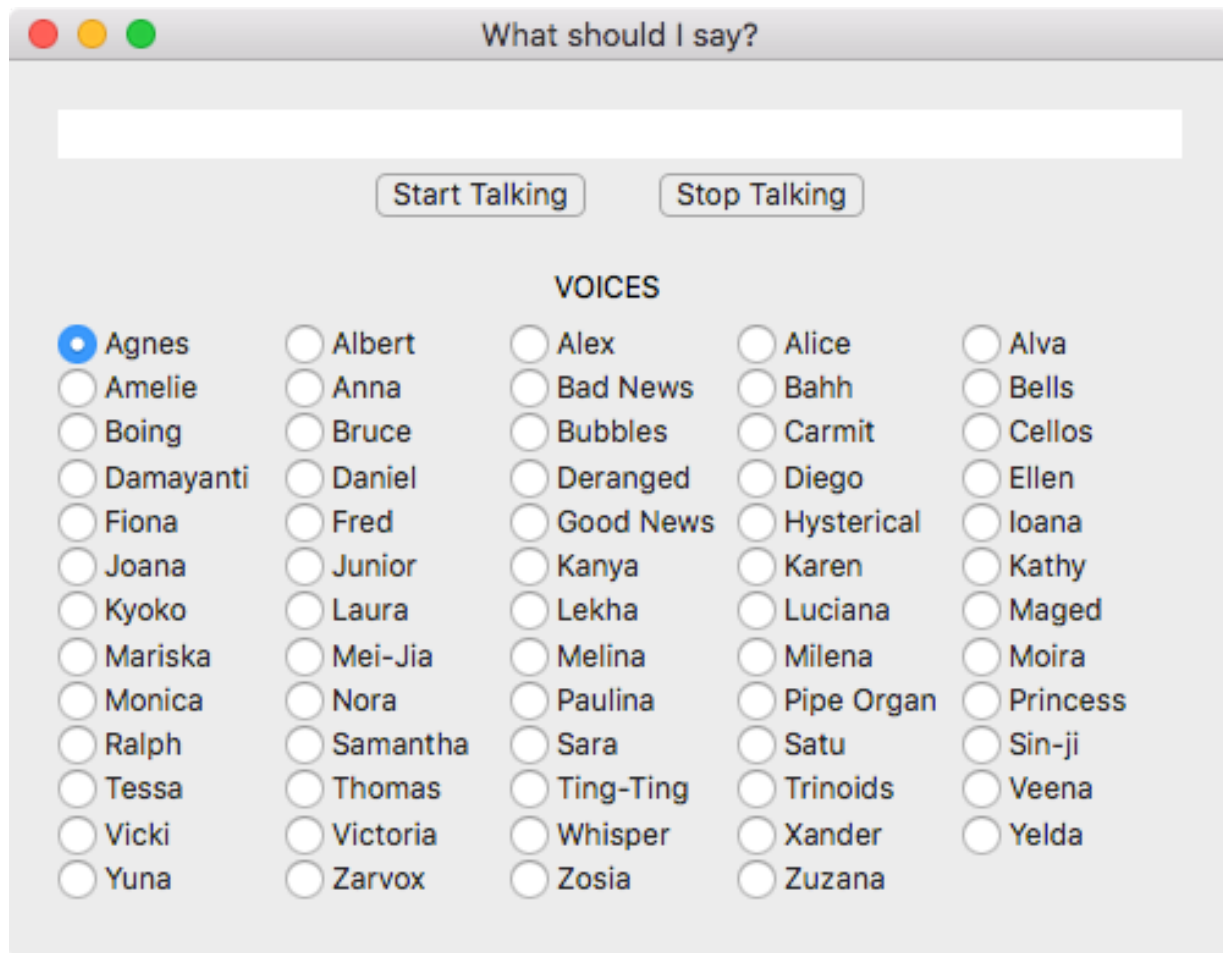
## **Project 4: Speech**

Key Concepts: Speech controller, Radio Buttons, Memory management, custom view definitions, dynamic layout constraints

At this point the runtime definition of windows should be somewhat more intuitive, so this fun little project should be a piece of cake. Basically it demonstrates how to configure and use radio buttons. It will also introduce a new class that I created called `resizable-box-view`. This is a custom view that is similar to the `organized-box-view` that we used in the `simple-sum` project except that the views within it are reorganized into an appropriately configured array as the size and relative dimensions of the box are modified at runtime (typically when the window is resized by the user).

For those of you who want to know how to implement your own custom views, you may want to take a look at the various view classes that are used for this and future projects. The emphasis in this document is on the use of existing classes rather than their implementation.

The window we're going to create will look like the following:



The user is able to type a string into the text field at the top, select a voice to use by picking one of the radio buttons at the bottom, and start and stop speaking with the two buttons in the middle. This picture of the window doesn't really do it justice because the look of the window will adjust dynamically as the window is resized. I encourage you now to do the following in the listener:

```
? (require :speech-controller)
:SPEECH-CONTROLLER
("RADIO-BUTTON" "SCROLL-VIEW" "SPEECH-CONTROLLER")
? (spc::test-speech)
#<LISP-WINDOW-CONTROLLER <LispWindowController: 0x27c09c10>
(#x27C09C10)>
```

This will pop up the window as shown above. Now resize the window and watch how the shape of things changes in response. Note that although the window does not have a fixed size, it also won't take on any arbitrary size and shape that you might ordinarily be able to get when resizing a window. Hopefully you will find that the dynamic limits for this window work to your satisfaction, but if not, then at the end of the project you should try to modify the constraints to get something more to your liking. The best way to discover what works well and what doesn't is to try different things yourself. We'll take a look at how all that window dynamics is defined as we go along. Keep both the look and

the behavior of the window in mind as the discussion continues. At a high level, what we're going to do is create an object class called speech-controller that will manage the operation of an Apple speech synthesizer object. As you might expect, it will respond to the buttons to change the voice, retrieve the text to be spoken from the text field at the top, and start and stop the synthesizer's speaking.

The source code for this project is in ...CocoaInterfaces/Example Projects/Speech/speech-controller.lisp file.

Just for a change of pace, let's start with the class definition for the speech-controller:

```
(defclass speech-data ()
  ((speech-text :accessor speech-text)
   (speech-synth :accessor speech-synth
                  :initform (make-instance 'ns:ns-speech-
synthesizer)))
  (voices :accessor voices
          :initform (coerce-obj (availableVoices ns:ns-
speech-synthesizer) 'list))))
```

You'll first note that this class is NOT a subclass of ns-object; it is pretty much an ordinary lisp class. As a result of that you will immediately realize that it cannot directly receive Objective-C messages. Instead we'll see that when buttons are pushed we will arrange for relevant lisp methods to be called which allow this object to respond appropriately. The speech-text slot will contain a reference to the text box at the top of the screen. We will need that to go get the text to be spoken. The speech-synth slot contains an ns-speech-synthesizer instance which will be used to do the speaking. The voices slot contains a list of strings that represent all the possible voices. We'll see how that is used as we go along. These strings are not just simple names of the voice. Instead, think of them as string identifiers for each voice.

We will use an initialize-instance :after method to do a little additional setup that is needed:

```
(defmethod initialize-instance :after ((self speech-data)
                                       &key &allow-other-keys)
  (setVoice (speech-synth self)
            (coerce-obj (first (voices self)) 'ns:ns-string))
  ;; we do the following so that ccl:terminate will be called
  before we are
  ;; garbage collected and we can release the speech synth
  object we created
  (ccl:terminate-when-unreachable self))
```

This first sets the voice of the speech synthesizer to the first on in the list of voices. As you'll see later, we'll assign names to radio button in the same order as this list and by default the first radio button in the collection will be set. So this just makes sure that the



voice set for the synthesizer matches the radio button that is set by default when the window first appears.

Since the speech-synth slot will contain an Objective-C object that we created, we must assume ownership for it. So we will arrange for that object to be reclaimed when the instance of speech-data is garbage collected. That's a two-part process with the first part being the `ccl:terminate-when-unreachable` call. That tells the CCL runtime to call the `ccl::terminate` method on this object just before it is garbage collected. Next we'll define that method:

```
(defmethod ccl:terminate ((self speech-data))
  (when (speech-synth self)
    (#/release (speech-synth self))))
```

As you can see, this releases the speech synthesizer object.

Next let's take a look at the operational methods that respond to button pushes. We'll see later how we arrange to have these methods called.

```
(defmethod start-stop-button-pushed ((self speech-data) title
tag)
  (declare (ignore title))
  (with-slots (speech-text speech-synth) self
    (ecase tag
      (0 ;; start talking button
        (let ((stxt (#/string speech-text)))
          (when (or (eql stxt (%null-ptr)) (zerop (#/length
stxt))))
            (setf stxt #@"I have nothing to say"))
            (#/startSpeakingString: speech-synth stxt)))
      (1 ;; stop talking button
        (#/stopSpeaking speech-synth)))))
```

The start-stop-button-pushed method is called when either the start or stop button is pushed. In effect, the tag argument tells us which of the two buttons was pushed; 0 indicating the start-talking button and 1 indicating the stop-talking button. To start talking, the method retrieves the text (or supplies its own if the text box is empty) and then calls the appropriate Objective-C method for the synthesizer to do the talking. The stop-talking button causes the synthesizer to stop operation. Pretty simple stuff.

```
(defmethod set-voice ((self speech-data) title tag)
  (declare (ignore title))
  ;; method called when a voice radio-button is pushed
  (#/setVoice: (speech-synth self)
    (coerce-obj (nth tag (voices self)) 'ns:ns-
string)))
```

The set-voice method is called when one of the voice radio-buttons is pushed. As before, the tag will tell us which button was pushed. We'll set things up so that the tag (which is always a number) corresponds to the index of the corresponding voice in the list of voice-strings in the voices slot. So it's just a matter of retrieving the right string and turning it into an Objective-C string that is passed to the speech synthesizer.

It's probably worth commenting on a few additional things here. Instead of storing a list of lisp strings in the voices slot, we could have stored a list of ns-string objects. That way we wouldn't have had to create one, each time. The tradeoff for that would have been that we would have to manage the memory for those objects as well. That's not a particularly big deal, so feel free to modify that yourself if you're offended by the extra overhead involved with turning a lisp string into an ns-string every time a radio button is pushed.

Another thing you might be wondering about is how the memory for those ns-strings that we created is managed. Don't we need to worry about that too? The answer is no, but you should understand the reason for that. The coerce-obj method is one that I created as an umbrella for a really large collection of lisp to Objective-C and Objective-C to Lisp translation methods. In particular, the methods that convert Lisp to Objective-C make sure that the resulting objects are all `autorelease`. We discussed this in the Project 1 discussion. Basically this assures that the objects have been pre-released so that unless they are subsequently `retain`'ed they will be garbage collected the next time the autorelease pool is emptied. They won't go away for the duration of processing the current event, so we can safely pass them on to the speech processor.

One final question you might have is about the reference to the text object. After all, that's an Objective-C object too, right? Don't we need to manage its memory? It is of course an object, but in this case we are only keeping a passive reference to it; we are not the owners. We didn't do a `retain` on the object when we set the slot. As developers we will assure that this object will be around for the duration of the window and after that there will be no buttons pushed so the speech-controller will have no further need to query the text object. So in this case, that's a safe thing to do.

Now let's move on to the definition of the window. Although a bit more complex than previous window creation methods, it is not all that much different.

```
(defmethod make-speech-window ((lc lisp-window-controller))
  (let* ((spc-data (make-instance 'speech-data))
        (scr-txt (make-instance 'vscrolled-text-view))
        (txt-view (#/documentView scr-txt))
        (voice-label (make-instance 'label-view
                                     :title "VOICES"))
        (button-box (make-instance 'button-box-view
                                   :titles (list "Start Talking" "Stop
Talking")
                                   :orientation :h
                                   :spacing 30
```

```

        :target spc-data
        :action-func #'start-stop-button-pushed))
(titles (mapcar #'(lambda (v)
                    (coerce-obj (objectForKey:
                                (coerce-obj v 'ns:ns-string))
                                #&NSVoiceName)
                                'string))
         (voices spc-data)))
(button-box (make-instance 'radio-button-array-view
                          :titles titles
                          :equal-width t
                          :target spc-data
                          :action-func #'set-voice))
(win (make-instance 'ns:ns-window
                   :title "What should I say?"
                   :resizable t
                   :content-subviews (list scr-txt voice-label
                                           button-box radio-box)))
(cview (cview win))

;; Link the speech-data object to the text object that
contains the string to say
(setf (speech-text spc-data) txt-view)

;; Fix the position of all the views relative to the content
and each other
(anchor cview scr-txt (list :leading :top :trailing))
(anchor cview radio-box (list :leading :bottom :trailing))
(order-views :orientation :v
             :align :center-x
             :views (list scr-txt button-box 20 voice-label
                           radio-box))

;; Set a minimum height for the scrolled text box so it
never disappears
(constrain (height scr-txt) 20))

;; We can't maintain a constant area for the radio box
because the equality that would be required
;; can't be expressed as any legal constraint. But we can
approximate that by requiring that as the
;; height decreases, the width must increase, or vice versa.
We do that by asserting that the sum of
;; the box height and width must equal 700. This starts to

```

```

fail if the box height gets
;; too small, so we'll make sure that doesn't happen either.
(constrain (= (height radio-box) (- 700 (width radio-box))))
(constrain (>= (height radio-box) 90))

;; Make sure the window is never made so narrow that the
buttons disappear
(constrain (>= (width cview) (+ (width button-box) 40)))

;; return the window and all instantiated objects
(values win (list spc-data scr-txt voice-label button-box
radio-box win)))

```

As previously, we first create the various views that we want in the window along with any controller or data source objects needed. In this case, we first make a speech-data object. Next we make the text view at the top. This is a bit different from the previous text fields that we have created (for example in the simple sum project). Those were intended to be relatively short, fixed width fields. In contrast, we'll want this field to expand in both width and height as the window is resized. So we'll make an instance of the `vscrolled-text-view` class that I created. By default, Apple provides both the `ns-text-field` class which provides those short single-line sorts of text that we used previously and `ns-text-view` class which provides views that can be both wider and taller and are intended for larger amounts of text. In most cases we would like the latter sort of text views to be scrollable. Rather than build scrolling into the class directly, Apple created a separate `ns-scroll-view` class that can be used to scroll any arbitrary sort of view. So what my `vscrolled-text-view` does is subclass the `ns-scroll-view` class and then also create an `ns-text-view` instance as the *documentView* of the scroll-view. Basically that just saves the developer the trouble of creating that combination themselves. If you are interested in how that was done you can look at the `scroll-view.lisp` file.

Since we will want to have direct access to that text view, the next line in the `let` form will extract a reference to it from the scroll view we just created:

```
(txt-view (#/documentView scr-txt))
```

After that we create a label and then a `button-box-view` that contains the start and stop talking buttons:

```

(button-box (make-instance 'button-box-view
                           :titles (list "Start Talking" "Stop
Talking")
                           :orientation :h
                           :spacing 30
                           :target spc-data
                           :action-func #'start-stop-button-pushed))

```

Basically a `button-box-view` is an `organized-box-view` (which we used previously in the simple sum example) that contains buttons. It also sets things up so that when one of those buttons is pushed a Lisp function (specified by the `:action-func` argument) will be

called with three arguments. The first is the value of the `:target` argument. The second is the title of the button that was pushed and the third is that tag of the button that was pushed. By default, buttons created will have tags that are consecutive integer numbers beginning at 0. The `:action-func` should be defined by the developer to do whatever useful thing it is you want done when the button is pushed. The same function is called for all buttons in the box, although of course such a function could be written to do completely different things depending on the title or tag argument. Earlier we looked at what the `start-stop-button-pushed` method actually does and here you see how we arranged to have it called.

The final view left to create is the box with all the radio buttons at the bottom of the window. We first generate a list of all the button names and do the following:

```
(radio-box (make-instance 'radio-button-array-view
                          :titles titles
                          :equal-width t
                          :target spc-data
                          :action-func #'set-voice))
```

This creates a `radio-button-array-view` object and populates it with a button for each title that we found. We will also look at this class more closely. Like the `button-box-view` class, you specify what buttons you want and what should happen when one of them is pushed. We looked at the `set-voice` method earlier in this discussion.

Once we've created all the objects we need, we need to make some additional connections between some of the pieces and then work on arranging the views within the window. There is a single connection to be made that is fairly straightforward:

```
(setf (speech-text spc-data) txt-view)
```

This gives the `spc-data` object a pointer to the text view at the top, so that it can retrieve the string that should be spoken.

Now we get to the fun part, i.e. the constraints needed to make everything look right and act as we desire when the window is resized.

The first thing we will do is to tie the scrolled text view at the top of the window to both sides and to the top. So as the width of the window is modified by user resizing, the width of that text object will also be modified to maintain that fixed relationship to the window's content view.

```
(anchor cview scr-txt (list :leading :top :trailing))
```

Similarly, we anchor the radio box to both sides and the bottom of the window:

```
(anchor cview radio-box (list :leading :bottom :trailing))
```

Next we will line up the views from top to bottom and space them in a way that seems good.

```
(order-views :orientation :v
              :align :center-x
              :views (list scr-txt button-box 20 voice-label
```

```
radio-box))
```

Next set a minimum height for the scrolled text box so it never disappears. I arbitrarily restricted it to about a single line. If you want to have more, then modify it accordingly.

```
(constrain (>= (height scr-txt) 20))
```

If we don't make sure that there is enough area to display all of the radio buttons, then the layout software will happily display the buttons on top of each other. Ideally we'd like to specify a fixed area so there would be enough room for them all, but this is one of those cases where the form of the constraint equation is mathematically insufficient to capture that intent. That is, we can't maintain a constant area for the radio box because the equality that is required can't be expressed as any legal constraint. But we can approximate that by requiring that as the height decreases, the width must increase, or vice versa. We do that by asserting that the sum of the box height and width must equal 700. This starts to fail if the box height gets too small, so we'll make sure that doesn't happen either.

```
(constrain (= (height radio-box) (- 700 (width radio-box))))  
(constrain (>= (height radio-box) 90))
```

The width of the text box at the top and radio-button box at the bottom can both be reduced in width pretty significantly without adversely affecting the layout, but if we let the box with the start/stop buttons shrink too much, we'd lose those buttons. So here we'll make sure the window is never made so narrow that those buttons disappear:

```
(constrain (>= (width cview) (+ (width button-box) 40)))
```

That ends our constraints. You can make your own judgment of course, but this seems like a fairly painless way to specify some fairly complex window display dynamics.

Finally we return the window and the list of objects that we want the window controller to manage.

```
(values win (list spc-data scr-txt voice-label button-box  
radio-box win)))
```

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

```
(defun test-speech ()  
  (on-main-thread  
    (let ((wc (make-instance 'lisp-window-controller  
                           :build-method #'make-speech-window)))  
      (show-window wc)  
      wc)))
```

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

Just to get some practice, you may also want to look at all the constraints that were generated. The test-speech function returns a lisp-window-controller so you could do something like the following:

```
? (setf wc (spc:test-speech))  
? (setf win (window wc))  
? (lv:analyze-constraints win)
```

### *Challenges:*

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

If you expect to develop many of your own views, then you may want to look at the internal implementation of the various view classes used for this project. I'm not going to go into them further in this document.

## **Project 5: Lisp Developer Tools**

Key Concepts: `ns-table-view` and `ns-outline-view` classes, `lisp-controller` class, binding, control enabling/disabling

So far the projects that we've done have simply used user input to do some easy things. In this project we'll be creating two different windows to display data that originates in the Lisp system. The first will show information about packages that have been created; namely what other packages either use them or are used by them. The second is the class keyword and binding target window that you may already have seen if you took my previous advice. You'll learn how easy it is to use the `lisp-controller` class along with Lisp functions provided by the developer to provide data for the `ns:ns-table-view` and `ns:ns-outline-view` objects that are the heart of these windows. We'll also begin to talk about *binding* which we will use to tie together Objective-C and Lisp slots so that a change in one of them is instantly reflected in a change in the other. Finally, we'll make both of these windows accessible from the Tools menu in the CCL IDE.

Let's start with the interface design for the package browser window. We will make it look like the one shown below:

Package	Nicknames
INSPECTOR	
INTERFACE-UTILITIES	IU
KEYWORD	
<b>LISP-CONTROLLER</b>	LC
LISP-VIEWS	LV
LOAN-CALC	LNC
LOAN-DOCUMENT	LND
MENU-TEST	MT
NEXTSTEP-FUNCTIONS	NSFIN

Package Uses	Package Used By
CCL	APP-DEV
COMMON-LISP	DEV
INTERFACE-UTILITIES	LISP-VIEWS
	PACKAGE-VIEW

At the top of the window is a two column scrollable ns:ns-table-view. These objects basically show arrays of information so the typical row/column terminology is used. In this case it is simply a 2 x N array of package information. In the first column is the package name and in the second it is a list of package nicknames for the package shown in the first column. At the bottom of the window we added two more ns-table-views. One is used to display all the classes that use the package highlighted in the upper table and one is used to display all of the packages used by the package highlighted in the upper table. Note that we could just as easily have used a single two column table at the bottom, but making them separate means that they are independently scrollable, which may be more desirable from an interface perspective.

The ns-table-view object contains ns-table-column objects. For each row that is currently visible, the ns-table-view object will request data for each column from a data source object that you will specify. We will always use a lisp-controller class instance as the specified data source because those objects know all about the protocol used between ns-table-view objects and their data sources. We'll see later how you configure the lisp-controller instance to provide the lisp data that you want to use. In the Cocoa paradigm many display objects retrieve their data automatically. It's not typically the case that a function executing somewhere tells the display how to change. Instead, if a program knows that the data displayed in some view is no longer valid it will tell the view to reload its data. This permits very fast updates. For example if you are displaying a very large table where most of it is currently invisible due to the position of the scrollbar, then the Table View object is smart enough to only reload the visible portions of its data. This also relieves the application code from having to be aware of how the scrollbar is set at each moment in time (although as we'll see in the next project it can certainly make itself aware if that's desirable).



Let's take a walk through the code for the package browser window. The source code for this project is in ...CocoaInterfaces/Example Projects/PackageView/package-view.lisp.

The first thing you will find is the definition of a Lisp class named package-data:

```
(defclass package-data ()
  ((package-ctrl :accessor package-ctrl)
   (use-ctrl :accessor use-ctrl)
   (used-by-ctrl :accessor used-by-ctrl)
   (current-package :accessor current-package :initform nil)))
```

As you will see when we do the class keyword and binding browser window, there is a technique that we can use to eliminate the need for even this simple class. But it is useful to use it so that you can we one method for specifying what is displayed with ns-table-view objects. This object contains three fields that will point to the three lisp-controller objects that provide data to the three ns-table-view objects in the window and one that keeps track of what package is currently selected in the top window.

Next we define a function that will be called when something is selected in the upper table. We'll see shortly how to specify this for the lisp-controller.

```
(defmethod pkg-selected ((self package-data)
                        (lc lisp-controller)
                        pkg-list
                        selected-indx
                        col-indx
                        selected-pkg)
  (declare (ignore pkg-list selected-indx col-indx))
  (with-slots (current-package use-ctrl used-by-ctrl) self
    (unless (eq current-package selected-pkg)
      (setf current-package selected-pkg)
      (setf (root use-ctrl) (package-use-list selected-pkg))
      (setf (root used-by-ctrl) (package-used-by-list selected-pkg))))))
```

This function is called when something is selected in the table. It is passed six parameters:

- The package-data instance
- The lisp-controller instance that is making the function call
- The complete list of packages displayed in the table
- The row index of the selected row
- The column index of the selected column
- The object (in this case a package) that is being displayed at the currently selected row and column

Most of these aren't needed for our immediate purposes and can be ignored. They are

provided by the lisp-controller to make it easy for a user method that might want them. The function is a simple one that first checks to see whether the newly-selected package is the same one that was previously selected and if so, does nothing. This can happen if you leave a window and then re-click on the same object when you come back. If that is the case, there is no need to update anything.

If something new was selected, then the only thing we need to do is change what data is displayed in the other two tables. As discussed earlier, we will use a lisp-controller object to provide data to each ns-table-view. A lisp-controller object has a *root* slot which can have any Lisp object as its value. Each row to be represented in a table will have a corresponding row object that is derived in some specified way from the root object. We will refer to rows as the *children* of that root object. This may seem like an unnecessarily awkward way to specify things at this point, but as we'll see when we talk about ns-outline-view objects, this terminology will become more reasonable. It permits us to use a single lisp-controller class as a controller for both ns-table-view and ns-outline-view objects. A root can be any sort of lisp object so long as there is a way to retrieve the sequence of row objects from it. Something like a list is a pretty good choice for a root object because the sequence of its children is just the object itself. Because this is such a common sort of root, the lisp-controller object will automatically recognize things like lists or vectors or hash-tables as root objects and know how to derive the sequence of rows from them by default. So that means that the only thing our pkg-selected method must do is set the root of the lisp-controller object used by each ns-table-view object to the list of packages we want displayed. You might be wondering how a lisp package object gets turned into something that can be displayed in each column of the table. Hang on and we'll see that in just a bit. The short answer is that lisp-controller objects know enough to convert Lisp data to Objective-C data and we give them a bit of additional help to get something displayable from a package.

The second column in the upper table lists the nicknames for a class. We will need a method to construct that list and that is what the nickname-string method does:

```
(defmethod nickname-string ((self package))
  (format nil "~{a~^,~}" (package-nicknames self)))
```

That's about all we need for now. Let's take next a look at the method that creates the package browser window. I'll insert comments as we go along

```
(defmethod make-package-window ((lc lisp-window-controller))
  (let* ((pkg-data (make-instance 'package-data))
```

First we create a pkg-data instance. It will manage the changes to the roots of the lisp-controllers for the two tables at the bottom of the window when a new package is selected in the upper table.

```
(pkg-lc (make-instance 'lisp-controller
  :func-owner pkg-data
  :select-func #'pkg-selected
```

```

:sort-key #'package-name
:sort-pred #'string<
:col-ids (list "PkgCol" "NickCol")
:col-keys (list #'package-name #'nickname-
string)

:root (list-all-packages)))

```

Next we made a lisp-controller instance for the table at the top of the window. Lisp-controller objects will call functions defined by you when certain events occur. Those events include

- The user selects an item
- The user adds an item
- The user deletes an item
- The user edits an item

In this case, we're only concerned about the first one. We configure the lisp-controller to call the pkg-selected function using the pkg-data instance as the first argument. Lisp-controllers will sort the rows displayed if you provide a sort predicate (using the :sort-pred keyword) and a sort key (using the :sort-key keyword). The sort-key is first applied to each row object to get a value for each row and then the sort-pred function is used to sort the row objects for display. The :col-ids keyword argument specifies a list of column identifiers. This is used to synch up what the lisp-controller knows about each column with what the ns-table-view knows about each column. As you'll see in just a second, we use those same identifiers when we create the column objects. The :col-keys argument specifies a list of functions to use to extract displayable information from row objects for each column. The first function in the list is used to get data from a row object for the first column listed in the :col-ids argument. For this project, row objects will be packages so the column with the "PkgCol" identifier will display the package name and the column with the "NickCol" identifier will display whatever the nickname-string function returns when it is applied to a package that represents a row.

```

(pkg-col (/autorelease (make-instance ns:ns-table-
column
                        :column-title "Package"
                        :identifier "PkgCol"
                        :min-width 80
                        :editable nil
                        :selectable t)))
(nick-col (/autorelease (make-instance ns:ns-table-
column
                        :column-title "Nicknames"
                        :identifier "NickCol"
                        :min-width 80
                        :editable nil
                        :selectable nil)))

```

Next we created the two column objects that will belong to the table at the top of the window. We give them titles that will be displayed at the top of the column and provide

an identifier that tells the lisp-controller which column it is so that it knows what data should be displayed. Columns are initially ordered in the table using the same order in which they are added to the table. But tables can be configured to allow user reordering and can also be programmatically reordered. The column identifier remains fixed so that is what the lisp-controller must use to determine what data is displayed in a column. Although we used `#/autorelease` calls here so that we wouldn't have to worry about getting rid of columns later, we could have avoided that and then passed the columns to the window controller along with everything else that we created and let them be released later when the window is closed.

```
(pkg-tab (make-instance ns:ns-table-view
  :columns (list pkg-col nick-col)
  :data-source pkg-lc
  :delegate pkg-lc
  :allows-column-resizing nil
  :column-autoresizing-style :uniform))
```

The form above makes the table at the top of the window. It specifies the two columns that were previously created and sets up the `pkg-lc` object as both the data-source and delegate for the table. We don't let the user resize the columns relative to each other and make all columns expand by the same amount when the window is resized.

```
(pkg-scroll-view (make-instance ns:ns-scroll-view
  :autohides-rollers t
  :has-horizontal-scroller nil
  :has-vertical-scroller t
  :document-view pkg-tab))
```

To make the top table scrollable, we need to enclose it in a `ns-scroll-view`. So we create one and make the `pkg-tab` its `:document-view`.

The following additional let forms do the same sorts of things for the other two tables.

```
(use-lc (make-instance 'lisp-controller
  :sort-key #'package-name
  :sort-pred #'string<
  :col-ids (list "UseCol")
  :col-keys (list #'package-name)))
column (use-col (#/autorelease (make-instance ns:ns-table-view
  :column-title "Package Uses"
  :identifier "UseCol"
  :min-width 80
  :editable nil)))
(use-tab (make-instance ns:ns-table-view
  :columns (list use-col)
  :data-source use-lc
```

```

        :delegate use-lc
        :allows-column-resizing nil
        :column-autoresizing-style :uniform))
(use-scroll-view (make-instance ns:ns-scroll-view
        :autohides-scrollers t
        :has-horizontal-scroller nil
        :has-vertical-scroller t
        :document-view use-tab))
(used-by-lc (make-instance 'lisp-controller
        :sort-key #'package-name
        :sort-pred #'string<
        :col-ids (list "UsedByCol")
        :col-keys (list #'package-name)))
(used-by-col (make-instance ns:ns-table-
column
        :column-title "Package
Used By"
        :identifier "UsedByCol"
        :min-width 80
        :editable nil)))
(used-by-tab (make-instance ns:ns-table-view
        :columns (list used-by-col)
        :data-source used-by-lc
        :delegate used-by-lc
        :allows-column-resizing nil
        :column-autoresizing-style :uniform))
(used-by-scroll-view (make-instance ns:ns-scroll-view
        :autohides-scrollers t
        :has-horizontal-scroller nil
        :has-vertical-scroller t
        :document-view used-by-tab))

```

This completes the creation of all three tables that will be displayed in the window.

```

(win (make-instance 'ns:ns-window
        :title "Package Browser"
        :resizable t
        :released-when-closed nil
        :content-subviews (list pkg-scroll-view
                                use-scroll-view
                                used-by-scroll-view)))

```

The window is a simple one, with only three tables. One of the atypical things about this window is that it will not be released when it is closed. That permits the user to close it and then reopen it from a menu. Since we didn't release it when it was previously closed, the window-controller can just show it again to make it reappear.

```
(cview (}/#{contentView win)))
```

This retrieves the content view from the newly created window so that we can constrain objects within it and relative to it. That completes the creation of the interface view objects. Next we will link everything up to make them operational. First we will set the slot values in the pkg-data object.

```
;; Set the links to the table controllers in the pkg-data  
object
```

```
(setf (package-ctrl pkg-data) pkg-lc)  
(setf (use-ctrl pkg-data) use-lc)  
(setf (used-by-ctrl pkg-data) used-by-lc)
```

Next we will set the view slot for each lisp-controller to its corresponding ns-table-view.

```
;; connect the views to the controllers
```

```
(setf (view pkg-lc) pkg-tab)  
(setf (view use-lc) use-tab)  
(setf (view used-by-lc) used-by-tab)
```

```
;; add constraints
```

```
;; Fix the position of all the views relative to the content  
view
```

```
(anchor cview pkg-scroll-view  
(list :leading :top :trailing))  
(anchor cview use-scroll-view (list :leading :bottom))  
(anchor cview used-by-scroll-view (list :bottom :trailing))
```

The next task is to add constraints so that the window displays and changes in the way we want. The code comments should be self-explanatory.

```
;; Make all tables the same height
```

```
(constrain (= (height pkg-scroll-view) (height use-scroll-  
view)))  
(constrain (= (height used-by-scroll-view) (height use-  
scroll-view)))
```

```
;; order the views vertically
```

```
(order-views :orientation :v :views (list pkg-scroll-view 15  
use-scroll-view))  
(order-views :orientation :v :views (list pkg-scroll-view 15  
used-by-scroll-view))
```

```
;; order the views horizontally
```

```
(order-views :orientation :h :views (list use-scroll-view  
used-by-scroll-view))
```

```
;; Make the two tables at the bottom the same width
(constrain (= (width used-by-scroll-view) (width use-scroll-view)))
```

```
;; Make sure the window is never too narrow so that
something disappears
(constrain (>= (width pkg-scroll-view) 300))
```

Finally we return the window and the list of objects that we want managed to the window controller.

```
;; return the window and all instantiated objects
(values win (list pkg-scroll-view use-scroll-view used-by-scroll-view
                  pkg-lc use-lc used-by-lc
                  pkg-tab use-tab used-by-tab
                  pkg-data win))))
```

The remaining code creates a global variable that will have as its value a lisp-window-controller that will manage the class browser window and then installs a new menuitem into the Tools menu that will tell that window controller to "showWindow:" when it is selected.

```
(defvar *package-window-controller*
  (make-instance 'lisp-window-controller
    :build-method #'make-package-window))

(install-menuitems-after "Tools"
  "Search Files..."
  (make-instance ns:ns-menu-item
    :title "Package Browser"
    :action "showWindow:"
    :target *package-window-controller*))
```

The class keyword and binding target window that was discussed earlier is just a slightly more advanced version of this same idea. The package-data class that we created played a pretty limited role in the whole operation. Basically it was there just so that when an object was selected in the top table, it could change the root object for the lisp-controllers that provided data to the bottom two tables. Wouldn't it be nice if there was a way for us to configure things so that this happened automatically without needing that package-data class and the pkg-selected method? As it turns out, there is. To understand how this works we must first talk about how *binding* works in Objective-C.

Bindings make use of Objective-C's *Key Value Coding (KVC)* and *Key Value Observing (KVO)* mechanisms. Briefly, binding two slots in different objects creates a dynamic relationship where a change to the value of one of the slots is immediately reflected in a change to the other. You can imagine how useful that might be to provide data to be

displayed in views. Whenever the data changes it is immediately reflected in the view. Or when the user changes the value of some control in a window, that value change is immediately reflected in a change in some data object slot. I'll provide a short introduction to KVC and KVO here, but I'd strongly suggest that you consult other resources for more information. One good source is Apple's' Cocoa Bindings Reference and you can follow other links from that document.

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

```
(id)valueForKey:(NSString *)key
```

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

If you think of this as access to object slot values by passing in the name of the slot as a parameter (i.e. as if by calling #'slot-value in Lisp) you won't be too far off the mark although there are differences in syntax and semantics as we'll see. KVC also supports access via a *key path*. A key path is basically a dot-separated list of successive keys. Often objects are connected via links between their slots and this provides a mechanism to follow those links to a final destination.

KVO provides a mechanism for objects to be notified when changes are made to other objects. One object can register itself as an observer for a specific slot of another object and when that observed slot is modified, all registered observers are notified. The observer can do anything that it wants with that data. The ns:ns-object class, from which all Objective-C object classes derive, provides default methods to make all Objective-C slots observable. The KVO protocol has additional complexities that I won't go into here.

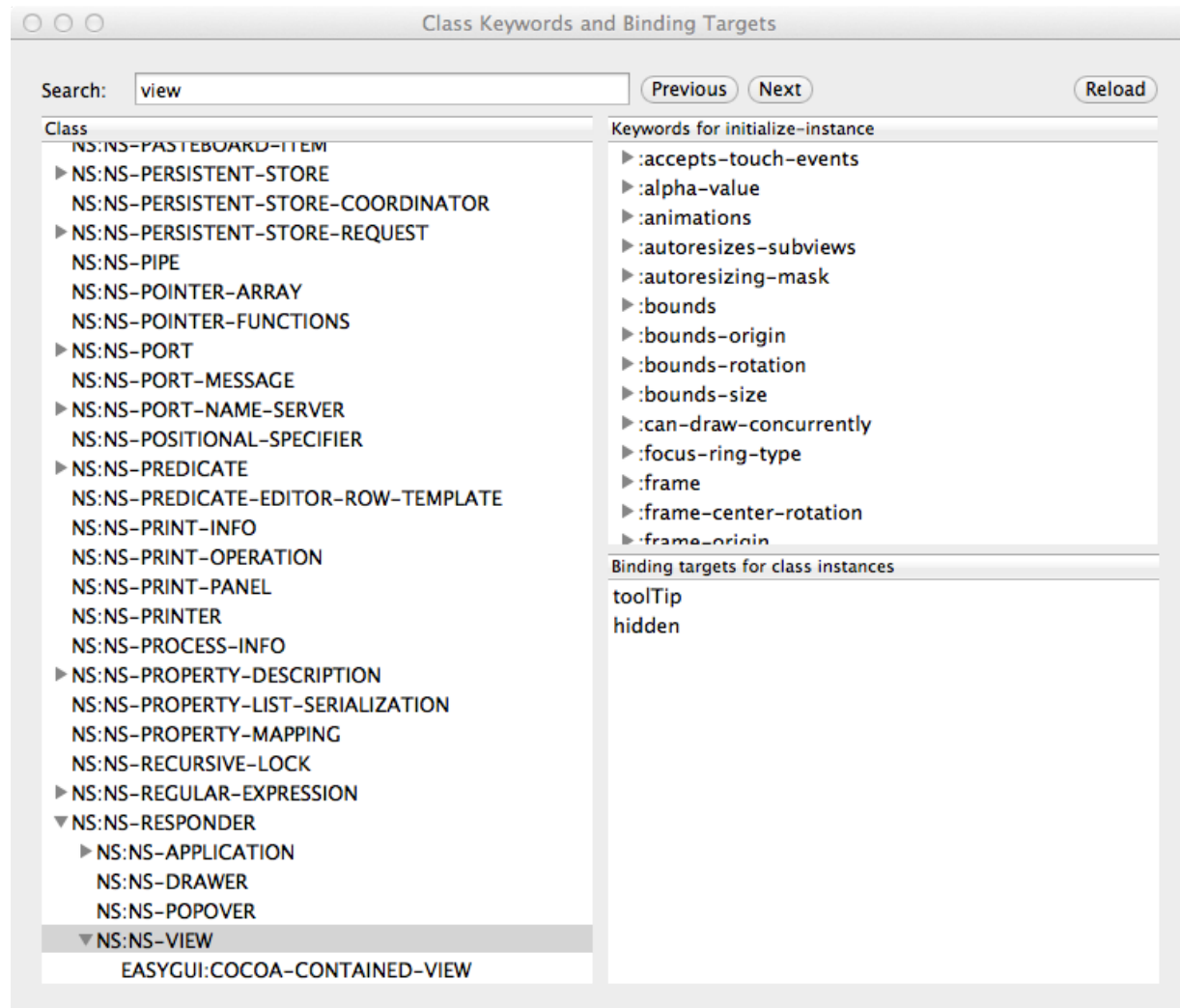
Although binding two slots sounds like a completely symmetric relationship, the implementation is not quite that simple. As you might imagine, there almost seems to be an infinite loop possible if a change to slot A results in a change to slot B and vice versa, how does the implementation avoid a loop? In actuality, the first slot specified in the call that makes a binding assumes a controlling roll. It will register as an observer for any changes to the second slot and respond to such changes by making a corresponding change to the first slot. When making a change to the first slot, it will directly change the second slot and essentially ignore the notification that it gets regarding the change that occurred.

The class ns-object (from which all of our classes so far inherit) has a default implementation of all methods needed for KVC and KVO compatibility. As we'll see in a later project, I have provided a way for you to make slots in completely normal lisp classes also be KVC and KVO compliant. This will make it possible to bind between Objective-C view slots and lisp data slots. The only thing you need to know here, is that the *root* slot in a lisp-controller object can be bound to the value of some other slot and that the *selection* slot in a lisp-controller object can also be bound. What this lets us do is bind the root in controller A to the value of the selection in controller B. So when the selection value is changed in controller B (typically by virtue of the user clicking on some row), then the value of the root in controller A will also be changed. Lisp-controller objects react to changes in their root values by changing everything displayed in the view with which they are associated. So all of that work we did with the package-data class and its methods can be replaced by creating simple bindings. We will do just that



for the class keyword and binding target window that you may have already been using. Let's have a look at that window and the code required to create it.

The window looks like this:



At the top of the window is a search field and associated *previous* and *next* buttons that can be used to quickly locate a class of interest in the taller table at the left side of the window. That table shows all sub-classes of ns-object in a hierarchical manner. Any class that has an associated arrow in front of it can be expanded (by clicking on the arrow) to display its subclasses. We'll see the code needed to make that happen shortly. The table at the top right displays all the keywords that are acceptable in a make-instance call for the class at the left. Clicking the arrow associated with a keyword will display information about acceptable values for that keyword. That may provide enough information for you to figure out what effect the keyword value has or you may have to consult other sources to really understand the real impacts. The table at the bottom right displays all of the exposed binding for a class. These are the slots to which you can

bind a value to have some effect on the operation of that object. Since you now have at least a small understanding of what bindings can do, you can probably see how useful this information might be.

The *reload* button at the top right corner of the window can be used if you discover that some class in which you are interested is either not shown or perhaps has incomplete documentation. When you then load whatever Lisp file contains that information, you need a way to specify that you want all information to be reloaded into this table. That's what the button does. Let's have a look at the code that makes all this work. You will find it in `dev-tools-interface.lisp`.

The first thing to note is what is not there. There is no class like the `package-data` class that we used for the `class-browser` window. We'll use bindings instead, as you'll see. So let's jump right into the method that makes the window:

```
(defmethod make-keyword-window ((wc lisp-window-controller))
  (let* ((search-text (make-instance 'labeled-text-field
                                     :title "Search: "
                                     :resizable t
                                     :label-pos :left)))
```

The first thing we do is make the search field, which is a simple labelled text field.

The buttons at the top will be used to tell the `lisp-controller` that manages the class hierarchy table what to do, so we'll make that `lisp-controller` first (so that we have a target that we can specify when the buttons are created).

```
(lc1 (make-instance 'lisp-controller
                   :search-key #'(lambda (cl)
                                   (symbol-name (class-name cl)))
                   :search-test #'(lambda (cl-name str)
                                   (search str cl-name :test
                                           #'string-equal)))
    :sort-key #'class-name
    :sort-pred #'string<
    :child-key #'class-direct-subclasses
    :col-ids (list "ClassCol")
    :col-keys (list #'class-name)
    :root ns:ns-object))
```

The `make-instance` call for the `lisp-controller` uses a few new keywords that we haven't previously seen. The `:search-key` and `:search-test` arguments are specifically there to support a search of all the data in the table. The `search-key` function is applied to each row and the `search-test` argument must be a function that compares each row's `search-key` result to a string from the `search-text` box. For our purposes here, the `search-key` function retrieves the name of each row object (a class) as a string. Then the `search-test` function tests whether the text field is a sub-string of the class name. If so, then it is

one of the found objects that the lisp-controller will scroll to when the next or previous button is pushed.

A hierarchical display such as we need for the class table at the left will use an ns:ns-outline-view rather than an ns:ns-table-view object. The former is a subclass of the latter. So we will need to configure the lisp-controller for that view somewhat differently than we did for those associated with simple table views. The first thing we need is to specify how to find any children of a row object. In general, this can vary depending on the level of the row being expanded within the hierarchy and it is possible to specify a different child retrieval function for each level of the hierarchy. Here, however, each level is the same; a class is represented at each level. So rather than specify how to get the children on a level-by-level basis, we can specify a single :child-key that will be used for all levels in the hierarchy. Obviously that function just retrieves the class's subclasses. The other keywords shown should be familiar from previous projects.

```
(next-button (make-instance ns:ns-button
                          :title "Next"
                          :action "searchNext:"
                          :target lcl
                          :bezel-style :round-rect
                          :button-type :momentary-light))
(prev-button (make-instance ns:ns-button
                          :title "Previous"
                          :action "searchPrev:"
                          :target lcl
                          :bezel-style :round-rect
                          :button-type :momentary-light))
```

The *next* and *previous* buttons are simple buttons that send messages to the lisp-controller associated with the class table. Those messages (searchNext: and searchPrev:) are defined for the lisp-controller class along with others (e.g. some that cause the addition or deletion of objects from a table) that we will encounter in future projects.

```
(reload-button (make-instance 'lisp-button
                          :title "Reload"
                          :action-func #'reload-doc-window
                          :bezel-style :round-rect
                          :button-type :momentary-light))
```

The *reload* button looks just like the others, but you will note that it is an instance of the class lisp-button rather than just an ordinary ns-button. The lisp-button class is used to create a button which directly call a specified Lisp function when it is pushed. If you are curious about how this was implemented, you can check out the code in button.lisp. We'll see the definition of the reload-doc-window method later.

Next we'll make the lisp-controller for the table at the top right.

```

      (lc2 (make-instance 'lisp-controller
        :col-ids (list "KeyCol")
        :col-keys-0 (list #'(lambda (class-key-list)
                              (key-name (second class-
key-list)))))
        :child-key-0 #'(lambda (class-key-list)
                          (list (apply #'key-doc-string
class-key-list))))
        :row-height-1 35
        :col-keys-1 (list #'identity)))

```

You'll note that here we used level-specific forms of the `:col-keys` and `:child-key` arguments. That's because each level shows a completely different sort of information. To understand what this code does, you will need to understand what each row object for this table is. The table displays information about a keyword for a specific class. While in most cases a single keyword is used identically for all classes where it is permitted, there are a few instances where this isn't accurate. So that means that a complete identifier for a row in this table must consist of both the keyword and the class. In fact we use a simple list of the class and the keyword as the row object. The argument `class-key-list` that you will see in the functions, represents an instance of that sort of row object. Row objects at the second level will be simple strings as returned by the function provided with the `:child-key-0` argument. The `key-doc-string` function is implemented in the `dev-tools.lisp` file. It retrieves a documentation string for a specified class keyword pair. The `:row-height-1` argument makes the size of level 1 rows large enough to display two lines of text. A more elegant solution might break documentation strings into a sequence of lines and return that sequence as the children. Note that the absence of a `:child-key-1` argument means that there are no children for any objects at that level.

```

      (lc3 (make-instance 'lisp-controller
        :col-ids (list "BindCol")
        :col-keys (list #'identity)))

```

The `lisp-controller` for the table at the bottom right will have a simple list of strings as its root. So the rows (i.e. the children of the root) will be the strings in that list and we don't have to specify anything else.

Next we create the column and the table for the class table at the left:

```

      (class-col (/#/autorelease (make-instance ns:ns-table-
column
                                :column-title "Class"
                                :identifier "ClassCol"
                                :min-width 60
                                :editable nil
                                :selectable t)))

```

```
(tv1 (make-instance 'ns:ns-outline-view
  :columns (list class-col)
  :data-source lc1
  :delegate lc1
  :allows-column-resizing nil
  :column-autoresizing-style :uniform
  :outline-table-column class-col))
```

The only additional argument needed when creating a hierarchical ns-outline-view object is the :outline-table-column which specifies which column is the one that has the hierarchical data. There is only one column in this table, so you would think it could make a decision on its own, but it turns out that you must always specify this.

```
(kw-col (#+autorelease (make-instance ns:ns-table-
column
  :column-title "Keywords for
initialize-instance"
  :identifier "KeyCol"
  :min-width 120
  :editable nil
  :wraps t
  :selectable t)))
(tv2 (make-instance 'ns:ns-outline-view
  :columns (list kw-col)
  :data-source lc2
  :delegate lc2
  :allows-column-resizing nil
  :column-autoresizing-style :uniform
  :outline-table-column kw-col))
```

Above we created the column and table for the keyword table at the upper right. Nothing very new here.

```
(bind-col (#+autorelease (make-instance ns:ns-table-
column
  :column-title "Binding
targets for class instances"
  :identifier "BindCol"
  :min-width 120
  :editable nil
  :wraps t
  :selectable t)))
(tv3 (make-instance 'ns:ns-table-view
  :columns (list bind-col)
  :data-source lc3
  :delegate lc3
  :allows-column-resizing nil
```

```
:column-autoresizing-style :uniform))
```

The binding target table is similarly simple. This is a ns-table-view so we don't need to worry about specifying which column is expanded.

Next we'll create scroll views to enclose each table:

```
(scroll-view1 (make-instance ns:ns-scroll-view
                          :autohides-scrollers t
                          :has-horizontal-scroller t
                          :has-vertical-scroller t
                          :document-view tv1))
(scroll-view2 (make-instance ns:ns-scroll-view
                          :autohides-scrollers t
                          :has-horizontal-scroller t
                          :has-vertical-scroller t
                          :document-view tv2))
(scroll-view3 (make-instance ns:ns-scroll-view
                          :autohides-scrollers t
                          :has-horizontal-scroller t
                          :has-vertical-scroller t
                          :document-view tv3))
```

Then create a window and add all the buttons and tables to it.

```
(win (make-instance 'ns:ns-window
      :title "Class Keywords and Binding Targets"
      :resizable t
      :released-when-closed nil
      :content-subviews (list search-text prev-button
next-button reload-button scroll-view1 scroll-view2 scroll-
view3)))
```

And then get the window's content view:

```
(cview (}/#{contentView win)))
```

As before, we link set the view slot for each of the lisp-controller objects:

```
;; connect the views to the controllers
(setf (view lc1) tv1)
(setf (view lc2) tv2)
(setf (view lc3) tv3)
```

Next we get to the various bindings that we'll need. The function *bind* is one that provides an interface to its Objective-C counterpart. It makes some things like binding to Lisp slots and providing Lisp functions in the binding path possible. What that means

will shortly become more clear.

```
;; bind the value of the search-text view to the controller
search-string slot
(bind search-text "value" lc1 "searchString")
```

As described above, this binds the search-text field to the lisp-controller slot search-string. Note the name conversion between the Objective-C "searchString" and the Lisp "search-string". There is some work required to make the search-string slot KVC and KVO compliant so that this works. This can be done either by adding all the necessary function calls to the code manually as the lisp-controller implementation does or by using a KVO compliance slot declaration that will be described for a later project. If you're curious about how to do this manually, have a look at the lisp-controller code. The manual process will only work for instances of Objective-C classes (which of course includes the lisp-controller class) whereas the slot declaration will work for lisp slots within pure Lisp classes.

```
;; bind enabled for the search buttons
(bind next-button "enabled" lc1 "canSearchNext")
(bind prev-button "enabled" lc1 "canSearchPrev")
```

The *next* and *previous* buttons are enabled or disabled by binding them to the "canSearchNext" and "canSearchPrev" fields in the lisp-controller respectively. These fields are set by the lisp-controller when it does its search and as a result manage whether the buttons are enabled in the interface window.

```
;; bind the root of the second table to the current
selection of the first, so that when
;; a new class is selected, its keywords are immediately
shown in the second table
(bind lc2 "root" lc1 (list "selection"
                           #'(lambda (class)
                               (when (typep class
'objc:objc-class)
                                   (mapcar #'(lambda (keys)
                                               (list class
keys))
                                   (init-keys
class :return-list t))))))
```

As described earlier we want each row object in the keyword table at the top right to be a list containing a class and a keyword for that class. We could really have accomplished this in either of two ways. An approach that might have been conceptually simpler than the one used here would have been to make the root of the table be a class object and provide a child function for the root that build a list of class-keyword lists. But to illustrate how a lisp function can be included inline in a binding path I instead chose to make the root be that list of class-keyword sublists. So what we see in the bind

call above is that the root of the keyword table is bound to a path that starts with the *selection* of the class table and then applies a Lisp function that creates the list we want. The type checking that is done in that function is necessary because we can't always be guaranteed that the object passed will be a class. That's because the selection may be nil. That's something to keep in mind when putting a Lisp function into a binding path like this. Such functions must always be prepared to accept nil as an argument.

```
;; bind the root of the third table to the current selection
of the first, so that when
;; a new class is selected, its binding targets are
immediately shown in the third table
(bind lc3 "root" lc1 (list "selection"
                          #'(lambda (class)
                              (when (typep class
'objc:objc-class)
                                (valid-bindings class))))))
```

The final binding above makes the root of the bindings target table also track the selection in the class table.

Next we will define constraints for the window. By now I expect that these should be easy for you to understand.

```
(anchor cview search-text (list :leading :top))
(anchor cview scroll-view1 (list :leading :bottom))
(anchor cview scroll-view2 (list :trailing))
(anchor cview scroll-view3 (list :trailing :bottom))
(anchor cview reload-button (list :trailing))

(constrain (>= (width search-text) 200))
(constrain (>= (width scroll-view1) 200))
(constrain (>= (height scroll-view1) 300))
(constrain (>= (width search-text) (* 0.5 (width cview))))
(constrain (= (width scroll-view1) (width scroll-view2)))
(constrain (= (width scroll-view1) (width scroll-view3)))
(constrain (= (height scroll-view2) (height scroll-view3)))
(order-views :orientation :h
              :views (list search-text prev-button next-
button)
              :align :center-y)
(order-views :orientation :v
              :views (list search-text scroll-view2 scroll-
view3))
(order-views :orientation :h
              :views (list scroll-view1 10 scroll-view2)
              :align :top)
(order-views :orientation :h
```



```

                :views (list scroll-view1 10 scroll-view3)
                :align :bottom)
    (align-views :views (list next-button reload-button)
                :align :center-y)

```

And finally we return the window and the list of objects to manage to the window-controller.

```

    (values win (list scroll-view1 scroll-view2 scroll-view3 lc1
lc2 lc3 tv1 tv2 tv3 win)))

```

Next we do pretty much the same thing that we did for the class browser. We first create a lisp-window-controller and make it the value of a global variable.

```

(defvar *dev-tools-window-controller*
  (make-instance 'lisp-window-controller
    :build-method #'make-keyword-window))

```

Next we define a function that reloads the table. This is a little ad hoc since it takes advantage of knowing something about the order of things returned to the lisp-window-controller by the make-keyword-window function. There are certainly more robust ways of doing this.

```

;; define a function that can be used to reset the documentation
window
(defun reload-doc-window ()
  (let ((lc (find-if #'(lambda (x)
                        (typep x 'lisp-controller))
                    (iu::instantiated-objects *dev-tools-
window-controller*))))
    (when lc
      (reload-documentation)
      (setf (root lc) ns:ns-object)
      (show-window *dev-tools-window-controller*)))))

```

The reload-documentation function is defined in dev-tools.lisp file. Setting the root of the lisp-controller (even though we're setting it to what it just was) causes a complete reloading of all table data. The show-window call is needed to cause the display to update on the screen.

```

(install-menuitems-after "Tools"
  "Search Files..."
  (make-instance ns:ns-menu-item
    :title "Class Keywords and Binding
Targets"
    :action "showWindow:"
    :target *dev-tools-window-

```

```
controller*))
```

Finally we add a new menuitem to the Tools menu, just as we did for the class browser window.

## Project 6: Loan Calc

Key Concepts: More about bindings, number and date formatters, slider controls, control hiding/showing, continuous updating, subclassing `lisp-window-controller`

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

We will continue to explore the division of responsibility advocated by the *model/view/controller (MVC)* paradigm. In the last project we used a window controller to keep track of window objects, but nothing else. In this project our custom window-controller will also assume responsibility for various ongoing window operations and will provide an access path to the data (i.e. the model). To do that our window-controller subclass will contain a pointer to a loan object which maintains the data.

This project will take advantage of *bindings* that we can set up between interface fields and class slots. Bindings are created using Objective-C's *Key Value Coding (KVC)* mechanism as implemented in Lisp via the *bind* method along with appropriate slot definition specifications. You can get more information about that in the *Lisp-KVO Reference and Tutorial* document.

The source code for this project is in ...CocoaInterfaces/Example Projects/Loan Calc/loan-calc.lisp.

We'll begin by looking at the data needed to compute loans:

```
(defclass loan ()
  ((loan-amount :accessor loan-amount
                :kvo "loanAmt"
                :initform 1000000)
   (interest-rate :accessor interest-rate
                  :kvo "interestRate"
                  :initform 0.04)
   (loan-duration :accessor loan-duration
                  :kvo "loanDuration"
                  :initform 12)
   (desired-loan-duration :accessor desired-loan-duration
                           :initform 12))
```

```

(monthly-payment :accessor monthly-payment
                 :kvo "monthlyPayment"
                 :initform 10000)
(origination-date :accessor origination-date
                  :kvo "originationDate"
                  :initform (now))
(first-payment :accessor first-payment
               :kvo "firstPayment"
               :initform (next-month (now)))
(pay-schedule :accessor pay-schedule
              :initform nil)
(total-interest :accessor total-interest
                :kvo "totalInterest"
                :initform 0.0)
(window-controller :accessor window-controller
                   :initform nil)
(compute-mode :accessor compute-mode
              :initform 0)
(hide-max-dur :accessor hide-max-dur
              :kvo "maxDur"
              :initform t)
(hide-min-dur :accessor hide-min-dur
              :kvo "minDur"
              :initform t)
(hide-min-pay :accessor hide-min-pay
              :kvo "minPay"
              :initform t)
(computing-new-loan :accessor computing-new-loan
                    :initform nil)))

```

Most of this should be relatively self-explanatory I think. The compute-mode slot will tell us which loan parameter is currently being computed in terms of the others. We'll let the user specify that we should compute the loan amount, the interest rate, the loan duration, or the monthly payment when the user provides the other three. The three hide-... slots will be used to control the display of warning messages if the set of parameters selected by the user results in a loan that cannot be paid off in a finite amount of time or would result in continuing payments after the loan is completely paid off. The computing-new-loan slot is something of a precaution. We want to prevent problems if the user should change a loan parameter in the window while we are already in the middle of computing new loan values. So we will keep track of the fact that a loan computation is in progress and react to value changes in a reasonable way.

Note the :kvo slot options used. These make the slot Key-Value Coding (KVC) and Key-Value Observing (KVO) compliant. See the *Lisp-KVO Reference and Tutorial* document for a more complete exposition of what this means. The upshot for this is that it makes it possible to bind interface object values to these Lisp slots. We specified all the :kvo options as strings that can then be used as part of the binding path specified in calls to

the *bind* method.

Monetary fields are notoriously difficult to represent in computer programs. Because of the decimal values the first inclination of most programmers is to use a floating point representation. But the inability to exactly represent all floats can lead to problems when numerical computations are done with those values. Some programming languages have special representations to deal with this. Objective-C has *ns-decimal-number* objects for example. Lisp has no exact equivalent, but we can easily create one just by using integers and understanding that this includes two implied decimal digits when we convert back and forth between Lisp and Objective-C representations. The conversion is handled automatically whenever a Lisp integer is translated into an *ns-decimal-number* or vice-versa, so the Lisp programmer only needs to understand what is going on when printing those values, so that the decimal point is placed appropriately. The *decimal.lisp* file provides functionality to support using integers for decimal representations and their conversion to and from various Objective-C *ns-number* objects.

In previous projects our test functions created a window-controller and then told it to construct and show the window that it would manage. In this project we will turn that around a bit. We will create the data object (a loan object) and as part of that creation the loan object will create the window controller. This is more consistent with the way that Apple document-based applications work. For example, if you open up a document that it saved on disk, the application will use the saved data to create and initialize an instance of that document and then tell the document to create windows to display the data. Here we will do essentially the same thing as part of the *initialize-instance* method for the loan object:

```
(defmethod initialize-instance :after ((self loan)
                                       &key &allow-other-keys)
  (setf (window-controller self)
        (make-instance 'loan-window-controller
                        :build-method #'make-loan-window
                        :loan self))
  (compute-new-loan-values self)
  (show-window (window-controller self)))
```

Of course in this case we don't have any saved values to use, so we start with some default values and compute everything needed. In the next project we will see how to save loan data to a disk file that can later be opened and used to initialize a new loan object.

Next we will create a custom subclass of *lisp-window-controller* that has additional slots that will contain various window views. These will be used to manage the dynamic appearance and functionality of the window as the compute mode changes:

```
(defclass loan-window-controller (lisp-window-controller)
  ((loan :accessor loan
```

```

      :initarg :loan
      :kvo "loan")
    (loan-text :accessor loan-text)
    (int-text :accessor int-text)
    (dur-text :accessor dur-text)
    (pay-text :accessor pay-text)
    (int-slider :accessor int-slider)
    (dur-slider :accessor dur-slider)
    (pay-slider :accessor pay-slider))
    (:metaclass ns:+ns-object))

```

Now we're ready to go build our interface. Our window look like the following:

We'll start by looking at the `make-loan-window` method which creates the window shown above. Then we'll examine the code that does all the loan computations.

```

(defmethod make-loan-window ((lc loan-window-controller))
  (with-slots (loan loan-text int-text dur-text pay-text int-
slider dur-slider pay-slider) lc
    (let* ((df (df (#/autorelease (make-instance ns:ns-date-

```

```

formatter
    :date-style :short
    :time-style :none)))
(mf (/#/autorelease (make-instance ns:ns-number-
formatter
    :number-style :currency
    :minimum-fraction-digits 2
    :maximum-fraction-digits 2
    :generates-decimal-numbers t
    :always-shows-decimal-separator
t)))
(intf (/#/autorelease (make-instance ns:ns-number-
formatter
    :number-style :percent
    :minimum-fraction-digits 2
    :maximum-fraction-digits 2
    :minimum-integer-digits 1
    :maximum-integer-digits 2
    :minimum 0
    :maximum .50
    :always-shows-decimal-
separator t)))

```

We start the build process by creating three ns-formatter objects. Formatters are special objects that enforce display standards in text fields. There are two subclasses that we will use here: ns-date-formatter and ns-number-formatter. Consult the documentation for those classes. If you have installed the keyword and binding browser in the Tools menu, that's a good way to see what settings are available for these formatters. Here we create one formatter for dates, one for fields that contain monetary amounts and one that shows percentages. We will specify that these should be used for text fields when we create them.

For the money formatter we specified that a currency symbol be shown, that there are always exactly two decimal digits, that a decimal point always be displayed, and that ns-decimal-number objects always be returned as the value of the text field.

For the interest rate formatter we specified a maximum value of .50 (i.e. 50%).

```

(orig-dt-tf (make-instance 'labeled-text-field
    :title "Origination Date"
    :label-pos :top
    :width 75
    :alignment :center
    :formatter df))
(first-payment-tf (make-instance 'labeled-text-field
    :title "First Payment"
    :editable nil

```

```
:selectable nil
:label-pos :top
:width 75
:alignment :center
:formatter df))
```

The first two fields we create are used to display the origination date and first payment date for the loan. We use the `labeled-text-field` class that was discussed for previous projects and put the label at the top center of the text field. The same formatter object is used for both of the fields. Initially, the first payment date is set to be one month after the origination date, but we allow the user to edit it. The date formatter object will enforce that only valid dates can be input.

```
(loan-amount-tf (make-instance 'labeled-text-field
                                :title "Loan Amount"
                                :label-pos :top
                                :width 90
                                :alignment :right
                                :formatter mf
                                :enabled nil))

(tot-int-tf (make-instance 'labeled-text-field
                            :title "Total Interest Paid:"
                            :editable nil
                            :selectable nil
                            :label-pos :left
                            :width 90
                            :alignment :right
                            :formatter mf))
```

Next we create monetary text fields for the loan amount and total interest paid. By default, text fields are both selectable and editable, so for the loan amount field we don't need to specify arguments for either of those. We do specify that initially the loan field will not be enabled. As mentioned previously, we will always be computing one loan variable given the others and initially the interface will be computing the loan value. Given that, we make sure that the user is not initially allowed to enter a value into that field. Later we'll see how text fields become enabled and disabled as we change the computation mode.

The total interest field only provides information and is not a source of input, so we make sure it is not selectable or editable.

[illegible]

We permit the annual interest rate to be entered using either a text field or a horizontal slider. We'll see below how to enforce that they have the same value. Here we just have to create the text field and a label for it. Note that we use the interest rate formatter for the text field.

```
(loan-dur-tf (make-instance ns:ns-text-field
                          :width 90
                          :alignment :right))
(loan-dur-label (make-instance 'label-view
                              :title "Loan Duration (months)"))
(month-pmt-tf (make-instance ns:ns-text-field
                          :width 90
                          :alignment :right
                          :formatter mf))
(month-pmt-label (make-instance 'label-view
                              :title "Monthly Payment"))
```

In a similar manner we make a text fields and labels to represent the loan duration and monthly payment.

```
(int-sl (make-instance ns:ns-slider
                      :continuous t
                      :enabled t
                      :max-value 0.5
                      :min-value 0.0))
(dur-sl (make-instance ns:ns-slider
                      :continuous t
                      :enabled t
                      :max-value 600.0
                      :min-vlaue 0.0))
(pay-sl (make-instance ns:ns-slider
                      :continuous t
                      :enabled t
                      :max-value 20000.0
                      :min-vlaue 0.0))
```

And also in a similar manner we make sliders to represent the three input fields. While it is technically possible to attach a formatter to a slider in the same way that we did for the text fields, it won't actually do anything. But we can specify minimum and maximum values for the slider, just as we did for the interest rate formatter. The `:continuous` argument is worth some discussion. This specifies that the value is continuously updated and not done only when the control loses focus. By doing that, we will see that the text field will instantly reflect the value of the slider.

We could have also specified `:continuous` for the text fields if desired. There are pros and cons to doing so. If you do not do this, then the value is not actually updated until



the field is exited (or the tab/enter/return key is pressed). So you may see a value entered in the field that is not the same as the value in the corresponding data object and is not therefore reflected in the computation of other fields. If you do make the update continuous, then when you erase a value to enter a new one, the updated value will cause all the values for other loan parameters to be instantly computed and displayed. This can make the values jump around quite a bit and be fairly annoying. Welcome to the world of interface design. You may make your own choice about this.

```
(radio-box (make-instance radio-button-box-view
                        :titles '("Loan Amount" "Interest Rate"
"Loan Duration" "Monthly Payment")
                        :target lc
                        :action-func #'set-compute-mode))
(rb-box (make-instance ns:ns-box
                      :title "Compute"
                      :title-position :at-top
                      :content-view radio-box
                      :content-view-margins '(4 4)))
```

The collection of radio buttons at the lower left corner of the window permit the user to specify which loan parameter is computed from the others. Changing this will result in some of the input fields being enabled while others are disabled and obviously what is computed is also affected. We create a radio-button-box-view which is just an invisible view that creates a button for each title specified and then organizes those buttons within it (vertically by default). When a button is pushed, a button-controller object that is contained within the radio-button-box-view will arrange for the specified :action-func function to be called with the :target as its argument. In this case the target is a custom window controller object that knows what to do when a button is pushed.

I could have just directly displayed the radio-box and created some kind of label to put close to it, but for demonstration purposes I chose to enclose it within an ns-box object. These are titled boxes that have a variety of options to customize the display (although here we aren't using many of them). We specify an inset margin size which results in the appearance you see in the window.

```
(small-font (smallSystemFontSize ns:ns-font))
(warn1 (make-instance label-view
                    :title "Above max duration needed to pay
loan"
                    :font small-font))
(warn2 (make-instance label-view
                    :title "Below min duration needed to pay
loan"
                    :font small-font))
(warn3 (make-instance label-view
                    :title "Below min payment needed for first
```

```
month's interest"
      :font small-font))
```

There are times when the user may try to set a value for some loan parameter that makes it impossible to pay off the loan in any finite amount of time or otherwise makes little sense. The loan code will detect these situations and fix the value at whatever extreme results in a legal state. It will then indicate the situation to the user by making a warning message visible. There are three such warning messages that we create as labels. As you will see later, we make them visible or invisible as needed.

```
(win (make-instance ns:ns-window
  :title "Loan Calculator"
  :resizable t
  :content-subviews (list orig-dt-tf first-
payment-tf loan-amount-tf tot-int-tf
                                ann-int-rt-tf month-
pmt-tf loan-dur-tf
                                int-sl dur-sl pay-sl
                                rb-box
                                ann-int-rt-label loan-
dur-label month-pmt-label
                                warn1 warn2 warn3)))
  (cv (#/contentView win)))
```

We create the "Loan Calculator" window and add all the subviews to it and then retrieve the window's content view for use in setting constraints.

Since we want the window controller to manage the various controls in the window we will next set those fields in the window controller object. Because of the (with-slots ...) form at the beginning of the make-loan-window method we can just directly do the setfs using the slot names.

```
;; we want to enable and disable the four text fields that
depend on the current loan
;; computation mode, so make pointers to these views
available to the loan-window-controller
(setf loan-text (text-field loan-amount-tf))
(setf int-text ann-int-rt-tf)
(setf dur-text loan-dur-tf)
(setf pay-text month-pmt-tf)

;; set slider fields
(setf int-slider int-sl)
(setf dur-slider dur-sl)
(setf pay-slider pay-sl)
```

All ns-view objects have a "hidden" property which can be set or bound. It controls

whether or not the object is currently displayed. We will take advantage of that to hide or show our three warning messages. We can easily bind the hidden property to a slot in the loan object that will indicate whether or not the warning should be displayed. When the value of the Lisp slot is modified as part of the loan computation, then the corresponding warning message in the window will either be displayed or hidden, as appropriate.

```
;; To make the warnings show up when we want, bind their
"hidden" property
(bind warn1 "hidden" loan "maxDur")
(bind warn2 "hidden" loan "minDur")
(bind warn3 "hidden" loan "minPay")
```

Next we bind the *value* slot of the various window controls to corresponding slots in the loan object. You will note that for the interest rate, loan duration, and monthly payment both a text field and a slider are bound to the same slot in the loan object. That means that the slider and text fields will always be synchronized. If the user moves a slider, for example, that will cause the corresponding value in the loan slot to be modified, which will in turn cause the value in the text field to be modified. If the loan slot is modified as a result of a loan computation, then both the text field and slider values are immediately set to that new value.

```
;; bind all the relevant value fields in controls
(bind first-payment-tf "value" loan "firstPayment")
(bind loan-amount-tf "value" loan "loanAmt")
(bind ann-int-rt-tf "value" loan "interestRate")
(bind loan-dur-tf "value" loan "loanDuration")
(bind month-pmt-tf "value" loan "monthlyPayment")
(bind orig-dt-tf "value" loan "originationDate")
(bind tot-int-tf "value" loan "totalInterest")
(bind int-sl "value" loan "interestRate")
(bind dur-sl "value" loan "loanDuration")
(bind pay-sl "value" loan "monthlyPayment")
```

Next we establish the constraints that affect where views are displayed

```
;; constrain sizes
(constrain (= (width rb-box) (+ 8 (width radio-box))))
(constrain (= (height rb-box) (+ 18 (height radio-box))))
(constrain (= (leading radio-box) (+ 4 (leading rb-box))))
```

We need to constrain the rb-box (ns-box-view) width and height to be greater than that of its contents and to make sure the leading edge is inset properly. There are other mechanisms that are supposed to make this happen, but I suspect that I invalidated them when positional constraints were specified for the rb-box. In any case, we can get the desired appearance by specifying appropriate margins and a constraint on the leading edge of the embedded radio-box.

```
(constrain-to-natural-size loan-dur-tf)
(constrain-to-natural-size month-pmt-tf)
(constrain-to-natural-size ann-int-rt-tf)
```

Above we constrain the text fields to a fixed natural size so that they do not change size as the window size is changed.

```
(constrain-to-natural-height int-sl)
(constrain-to-natural-height dur-sl)
(constrain-to-natural-height pay-sl)
```

We want the width of the sliders to change as the width of the window is modified, so we only constrain their height and let the width float.

```
;; top row of objects
(anchor cv orig-dt-tf '(:top :leading))
(anchor cv first-payment-tf '(:top :center-x))
(anchor cv loan-amount-tf '(:top :trailing))
```

We start by fixing the position of the objects at the top of the window. As the window width changes, those objects will just spread out without changing the size of the field (since we already constrained that to a fixed width).

```
;; sliders and related text fields
(order-views :views (list :border int-sl ann-int-rt-
tf :border)
              :orientation :h
              :align :center-y)
(order-views :views (list :border dur-sl loan-dur-
tf :border)
              :orientation :h
              :align :center-y)
(order-views :views (list :border pay-sl month-pmt-
tf :border)
              :orientation :h
              :align :center-y)
```

Above we established the horizontal ordering between the side borders of the window, the sliders, and their related text fields. The ordering implies a fixed-size space constraint between the borders and the objects and between the slider and the text field. The alignment assures that they are vertically aligned as well.

```
;; align centers of radio box and total-interest text
field
  (align-views :views (list rb-box tot-int-tf)
               :align :center-y)
```

```
(constrain (>= (leading tot-int-tf) (+ 6 (trailing rb-
box))))
```

Above we specify an alignment between the radio box and the total interest text field. We don't order them because we don't want to constrain the space between them to a fixed distance, but we do add a constraint that keeps them separated by a fixed amount. This has the side-effect of constraining the width of the window to be at least wide enough to avoid violating this constraint.

```
;; Then vertically
(order-views :views (list orig-dt-tf 30 int-sl 30 dur-sl
30 pay-sl rb-box :border)
              :align :leading
              :orientation :v)
```

Next we order the views going down the left side of the window. Since we already aligned them horizontally we don't need to do a similar ordering down the middle or right side of the window.

```
(align-views :views (list loan-amount-tf ann-int-rt-tf
loan-dur-tf month-pmt-tf tot-int-tf)
              :align :trailing)
```

But we do need to align the views down the right side to make sure that as the window expands in width, the objects stay together at the right edge. These constraints, in combination with the horizontal constraints specified earlier, can only be satisfied by modifying the width of the sliders, so that's what happens when the width of the window is modified.

```
;; constrain label positions
(constrain (= (top int-sl) (+ 3 (bottom ann-int-rt-
label)))))
(align-views :views (list int-sl ann-int-rt-label)
              :align :center-x)
(constrain (= (top dur-sl) (+ 3 (bottom loan-dur-label)))))
(align-views :views (list dur-sl loan-dur-label)
              :align :center-x)
(constrain (= (top pay-sl) (+ 3 (bottom month-pmt-
label)))))
(align-views :views (list pay-sl month-pmt-label)
              :align :center-x)
```

Above we set the position of the labels over the sliders. Since we're only aligning two views rather than several, it probably would have been simpler to use simple constrain calls rather than align-view calls (e.g. something like "(constrain (= (center-x

ins-sl) (center-x ann-int-rt-label)))"), but the constraints created would have been identical.

```
;; constrain warning label positions
(constrain (= (top warn1) (+ *no-space* (bottom loan-dur-
tf))))
  (align-views :views (list warn1 loan-dur-tf)
               :align :trailing)
  (constrain (= (top warn2) (+ *no-space* (bottom loan-dur-
tf))))
  (align-views :views (list warn2 loan-dur-tf)
               :align :trailing)
  (constrain (= (top warn3) (+ *no-space* (bottom month-pmt-
tf))))
  (align-views :views (list warn3 month-pmt-tf)
               :align :trailing)
```

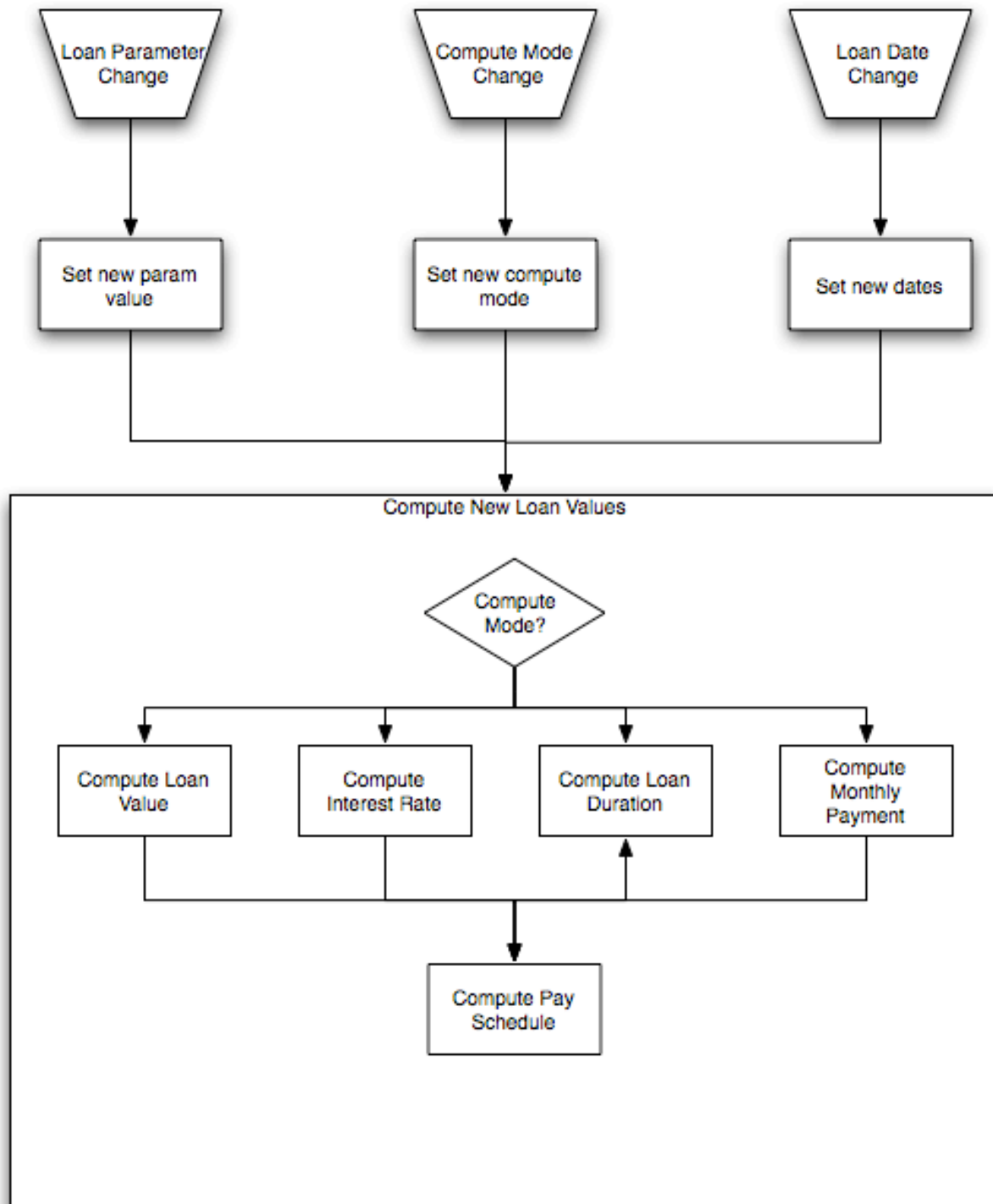
Similarly, we set the positions of the warning labels. *\*no-space\** is actually a non-zero constant that makes the objects appear to be right next to each other. The align-views calls here could also have been done with simpler constrain calls.

```
(values win (list orig-dt-tf first-payment-tf loan-amount-
tf tot-int-tf
               ann-int-rt-tf month-pmt-tf int-sl dur-sl
pay-sl radio-box rb-box
               ann-int-rt-label loan-dur-tf loan-dur-
label month-pmt-label
               warn1 warn2 warn3 win))
```

Finally, as we always do, we return the window and the list of objects to be managed to the window controller.

The following will discuss the implementation of the loan calculation application. If you don't really care about this, feel free to skip it.

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



Loan Program flow

When users make changes in the loan window the binding that we specified results in them being converted and saved as Lisp values in the appropriate slot of the loan object. As you will see, we set things up so that such changes result in a notification function being called. Then, depending on the current compute mode, the variable that is free to move based on the combination of other values is recomputed. Finally the

whole pay schedule is recomputed.

There are various places in this process where we may change a value displayed in the loan window. It should be obvious that anytime we compute the new value of a loan parameter that will cause the display to be updated. What is less obvious is that there are various combinations of parameters that may cause us to change a loan parameter other than the one we were explicitly directed to compute. For example, suppose we are told to compute the loan duration while the user manipulates the loan amount, interest rate, and monthly payment. Suppose the user increases the interest rate to a point where the monthly payment is no longer even as large as the first month's interest? Clearly the user is adjusting the interest rate, so they must want it higher. But if we left the monthly payment alone then the loan could never be paid off. Not the least of our problems would be an infinite loop when we tried to compute a payment schedule. So in this case we choose to arbitrarily adjust the monthly payment so that it minimally covers the first month's interest plus \$1 to start paying off the loan principle. There are other cases where we must adjust the loan duration either up (to pay off a loan) or down (because the loan is paid off earlier than the duration the user requested). So we change these values dynamically as necessary to maintain a consistent set of loan parameters. In these cases where we adjust a value that wasn't dictated directly by the user, we will display a small banner explaining why we did so. These are the banners that we created that are conditionally hidden in the loan window.

### *Loan-Controller Functionality*

The next section of loan-calc.lisp concerns the loan-controller class and its functionality. We already showed the class definition.

When a radio button is pushed we will change which of the loan variables is computed from the others. You may recall that when we created the radio-button-box we specified:

```
:target lc
:action-func #'set-compute-mode
```

which causes the set-compute-mode method to be called with the loan-window-controller as the first argument. The second and third arguments passed are the title of the button that was pushed and its 0-relative index in the list of buttons. So we define that method for the loan-window-controller as follows:

```
(defmethod set-compute-mode ((self loan-window-controller) title
cm)
  (declare (ignore title))
  ;; called when one of the loan mode radio buttons is pushed
  ;; We use the button tag as the compute mode
  (with-slots (loan loan-text int-text dur-text pay-text int-
slider
              dur-slider pay-slider) self
    (unless (eql cm (compute-mode loan))
      (case (compute-mode loan)
        ;; enable things disabled for the previous compute mode
```



```

        (0 (/#setEnabled: loan-text #$YES))
        (1 (/#setEnabled: int-text #$YES)
          (/#setEnabled: int-slider #$YES))
        (2 (/#setEnabled: dur-text #$YES)
          (/#setEnabled: dur-slider #$YES))
        (3 (/#setEnabled: pay-text #$YES)
          (/#setEnabled: pay-slider #$YES)))
(setf (compute-mode loan) cm)
(case cm
  ;; disable things as appropriate for the new compute
mode
  (0 (/#setEnabled: loan-text #$NO))
  (1 (/#setEnabled: int-text #$NO)
    (/#setEnabled: int-slider #$NO))
  (2 (/#setEnabled: dur-text #$NO)
    (/#setEnabled: dur-slider #$NO))
  (3 (/#setEnabled: pay-text #$NO)
    (/#setEnabled: pay-slider #$NO)))
(compute-new-loan-values loan)))

```

This does a few simple things. We first determine which user interface objects are currently disabled and re-enable them. Then we disable those that correspond to the value that we are now being asked to compute. And of course we set the slot value that contains the compute mode in the attached loan object. Note that the Loan object (which controls data) does not have to be aware of which radio button is pushed or even the fact that we're using radio buttons to select the compute mode.

```

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

```

We also provide a method that will notify the loan object when the window is closed. As we will see, the loan will then release the window-controller, allowing its memory to be garbage collected.

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

### *Loan Functionality*

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

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

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

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

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

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

The loan class and its initialize-instance :after method were shown and discussed previously.

When the user modifies one of the loan parameters in the window, we know that the corresponding slot value in the loan object will be modified, but how do we know that happened so that we can go recompute the other loan parameters? Part of the Lisp extension to the binding protocol will call the method *bound-slot-modified* whenever a slot value is changed as the result of a binding action. We need only define a specific version of that method for the loan class:

```
(defmethod bound-slot-modified ((self loan) slot-name)
  ;; will only be called if a slot is modified as the result of
  a KVO binding
  ;; any other change will not result in calling this method
  (when (eq slot-name 'origination-date)
    ;; also set the first-payment date
    (setf (first-payment self) (next-month (origination-date
self))))
  (when (eq slot-name 'loan-duration)
    ;; also set the desired-loan-duration
    (setf (desired-loan-duration self) (loan-duration self)))
  (setf (pay-schedule self) nil)
  (compute-new-loan-values self))
```

This method will do a few additional things that we want when certain slots are modified. When the origination date is set, we will also set the first payment date. The next-month function is defined along with other date functions in date.lisp. The loan duration slot is one that we sometimes have to modify as part of a loan calculation process. But we want to stay as close to what the user explicitly requested as possible. So we also keep track of that requested value and return to it for each calculation. If instead we used the current value of the loan duration slot then each time we recomputed the loan it might

get farther and farther from the value requested. We also reset the pay schedule when anything changes and then compute new loan values.

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

```
(defmethod compute-new-loan-values ((self loan))
  ;; For the sake of expediency we assume monthly compounding

  ;; First check to see if we're already running this method. If
  so
  ;; just get out.
  (when (computing-new-loan self)
    (return-from compute-new-loan-values))
  (setf (computing-new-loan self) t)
  (with-slots (compute-mode interest-rate loan-duration desired-
loan-duration monthly-payment
loan-amount pay-schedule) self
    (case compute-mode
      (0
       ;; compute the loan amount
       (unless (or (zerop interest-rate)
                   (zerop desired-loan-duration)
                   (zerop monthly-payment))
         (setf loan-amount
                 (round (/ monthly-payment
                           (pay-to-loan-ratio (/ interest-rate 12)
                                                desired-loan-
duration))))))
       (set-pay-schedule self)))
      (1
       ;; compute the interest rate
       (unless (or (zerop loan-amount)
                   (zerop desired-loan-duration)
                   (zerop monthly-payment))
         (setf interest-rate
                 (* 12.0 (/ (floor (* 1000000 (compute-int-rate
self)))
                           1000000))))
       (set-pay-schedule self)))
      (2
       ;; compute the loan duration
       (unless (or (zerop interest-rate)
                   (zerop loan-amount)
                   (zerop monthly-payment))
         (set-pay-schedule self)
         (setf loan-duration
```

```

                (list-length pay-schedule))
        (setf desired-loan-duration
              (list-length pay-schedule))))
    (3
     ;; compute the monthly payment
     (unless (or (zerop interest-rate)
                 (zerop loan-amount)
                 (zerop desired-loan-duration))
              (setf monthly-payment
                    (round (* loan-amount
                              (pay-to-loan-ratio (/ interest-rate 12)
                                                    desired-loan-
duration))))))
        (set-pay-schedule self))))
    (setf (computing-new-loan self) nil))

```

This method obviously does things a bit differently for each compute mode. The computations for loan amount (mode 0) or monthly payment (mode 3) have easy closed-form solutions that are derived from the basic loan equation. The loan duration (mode 2) can be directly derived from the length of the payment schedule so we just call that first and then set the value of the loan-duration slot. We use a search algorithm to compute the interest rate (mode 1) and round the value returned to the number of digits we want to display. There is a separate method to do the search (compute-int-rate) which is shown below.

```

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

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

    ;; First we'll check to see whether the monthly payment is
    greater than the loan amount.
    ;; If so we'll set the interest rate directly so that the
    loan is paid off in one month.
    ;; This avoids some ugly arithmetic overflow things that can
    happen when interest rates
    ;; go off the charts
    ;; The probability is very high that the interest rate
    computed here will result in a
    ;; loan duration that is one month more than that desired.

```

So we'll just reduce the

```
;; desired duration by one month before beginning.
(let ((max-monthly-rate (/ $max-interest-rate$ 12))
      (effective-duration (1- desired-loan-duration)))
  (if (>= monthly-payment loan-amount)
      (min max-monthly-rate (1- (/ monthly-payment loan-
amount)))
      (let ((imin (max 0 (min max-monthly-rate
                              (/ (- (* monthly-payment
effective-duration) loan-amount)
                                  (* effective-duration loan-
amount))))))
        ;; imin is basically a rate that would result in
the first month's interest as
        ;; the average interest paid for all months. Since
we know it must be greater
        ;; than this, we have a guaranteed lower bound.
But we cap it at our allowed
        ;; monthly maximum interest.
(imax (min max-monthly-rate
            (- (/ monthly-payment loan-amount) .
000008333)))
        ;; imax is a rate that would result in the first
month's interest being
        ;; minimally smaller than the payment. Since we
must pay off in a finite
        ;; duration, this is a guaranteed maximum. We cap
it the allowed maximum
        ;; monthly rate.
(target-p-l-ratio (/ monthly-payment loan-
amount)))
      (unless (>= imax imin)
        (error "Max int = ~8,4f, Min int = ~8,4f" imax
imin))
      (do* ((i (/ (+ imin imax) 2)
                  (/ (+ imin imax) 2))
            (p-l-ratio (pay-to-loan-ratio i effective-
duration)
                        (pay-to-loan-ratio i effective-
duration)))
        ((<= (- imax imin) .000001) imax)
        (if (>= target-p-l-ratio p-l-ratio)
            (setf imin i)
            (setf imax i))))))
```

This method uses a binary search to find the interest rate that produces something very close to the target payment to loan ratio. We can compute that target from the user-

specified payment and loan value parameters. The initial bounds for the interest rate are computed as discussed in the code comments.

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

The final method we will discuss computes a detailed payment schedule given all the loan parameters. Since we never display this anywhere you may wonder why it exists. There are four reasons. First it makes it very easy to determine the loan duration when we need to compute that from the other values. Second, I found it to be a useful debugging tool. I could print out the payment schedule in the listener just to make sure that everything looked good. I uncovered several bugs that way. Third, I wanted to provide a challenge for you to add functionality to this application. And fourth, I wanted to have something more substantial to print out when we get to the next project.

```
(defmethod set-pay-schedule ((self loan))
  ;; create a detailed payment schedule for the loan using daily
  compounding of interest
  ;; Payments are on the same date of each month, but the number
  of days between payments
  ;; varies because the number of days in each month varies.
  ;; We compute accurate interest compounded daily for the
  actual number of days.
  (let ((monthly-interest (/ (interest-rate self) 12))
        (payment (monthly-payment self))
        (sched nil)
        (display-min-pay-banner nil))
    (progl
      (do* ((begin (loan-amount self) end)
            (begin-date (first-payment self) end-date)
            (end-date (next-month begin-date) (next-month
begin-date)))
          (int (round (* begin monthly-interest))
              (round (* begin monthly-interest)))
          (end (- (+ begin int) payment) (- (+ begin int)
payment))))
      ((not (plusp end))
       (progn
        (push (list (short-date-string begin-date)
                    (/ begin 100)
                    (/ int 100)
                    (/ payment 100)
                    (short-date-string end-date)
```

```

                                (/ end 100)
                                int)
                                sched)
                                (setf (pay-schedule self) (nreverse sched))))
(when (>= end begin)
  ;; oops, with this combination of values the loan
will never
  ;; be paid off, so let's set a minimum payment
required
  ;; Display a field that tells user the minimum
payment was reached
  (setf display-min-pay-banner t)
  (setf (monthly-payment self) (1+ int))
  ;; now patch up our loop variables and keep going
  (setf payment (monthly-payment self))
  (setf end (1- begin)))
;; put the last payment into the list
(push (list (short-date-string begin-date)
            (/ begin 100)
            (/ int 100)
            (/ payment 100)
            (short-date-string end-date)
            (/ end 100)
            int)
      sched))
(setf (total-interest self) (compute-total-interest self))
(if display-min-pay-banner
  (progn
    ;; Set a condition that says the minimum payment was
reached
    (setf display-min-pay-banner t)
    (setf (hide-min-pay self) nil))
  ;; otherwise reset that condition
  (setf (hide-min-pay self) t))
;; If we happen to be computing the interest rate, then
;; the combination of loan-amount and monthly payment will
;; determine a maximum interest rate. This, in turn,
;; determines a maximum loan duration. If the duration was
set
  ;; longer than this by the user, we will reset the
  ;; lone duration value to the maximum needed.
  ;; If, on the other hand, the monthly payment is set so
low that
  ;; the interest rate approaches 0, then we may have to
adjust the
  ;; loan duration up to the minimum needed to pay the loan.
  ;; Let's start by resetting our two "duration" conditions

```

```

and then we'll
    ;; set them if conditions dictate.
    ;; Reset a condition that indicates the max duration was
reached
    (setf (hide-max-dur self) t)
    ;; Reset a condition that indicates the min duration was
reached
    (setf (hide-min-dur self) t)
    (let ((duration-diff (- (desired-loan-duration self)
(list-length (pay-schedule self)))))
        (unless (or (eql (compute-mode self) 2) (zerop duration-
diff))
            ;; i.e. we're not calling this function just to
determine the loan duration
            ;; and we have to adjust the loan duration
            (if (plusp duration-diff)
                (progn
                    ;; change the loan-duration value to what it must
be
                    (setf (loan-duration self) (list-length (pay-
schedule self)))
                    (when (> duration-diff 2)
                        ;; If we're one-off just fix it and don't post a
message
                        ;; This can occur almost anytime because of
numerical issues
                        ;; Display a field that tells user the max
duration was reached
                        (setf (hide-max-dur self) nil)))
                    (progn
                        ;; change the loan-duration value to what it must
be
                        (setf (loan-duration self) (list-length (pay-
schedule self)))
                        (when (< duration-diff -2)
                            ;; If we're one-off just fix it and don't post a
message
                            ;; This can occur almost anytime because of
numerical issues
                            ;; Display a field that tells user the min
duration was reached
                            (setf (hide-min-dur self) nil))))))))))

```

This is a pretty simple function that starts with the initial loan value, calculates and adds the interest on that amount for one month, subtracts the payment to derive a new loan value and iterates until the loan value drops to zero or below. As it proceeds it keeps a record of each of the payments. You may note that we put the monthly interest paid into



each record twice; once as a float value that we can print directly and once as internally represented with a fixnum. The former is used in our payment schedule print function and the latter is used to compute the total interest for the interface.

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

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

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

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

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

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

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

Give it a try.

### *Challenges:*

One of the things that might be useful for users is to see the whole loan payout schedule. Add a button to the window (or a menu item or both) that the user can select to open up a whole new window in which to display the loan-payout schedule. Create a `make-sched-window` method for this window and create a window controller, specifying this build-method when the user requests it.

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

## **Project 7: Loan Document**

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

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

It's first necessary to understand the relationship between various classes in a normal application that supports documents. I will provide a brief introduction here, but if you want the whole story I suggest that you read Apple's "Document-Based Applications Overview". You can find this on Apple's web-site at:

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

Every document-based application will create a single instance of an `ns-document-controller`. The first instance created will be used as the shared instance for any later reference. Most document-based applications (including the CCL IDE) make a subclass of `ns-document-controller` that is unique to the application. They arrange to create an instance of it as one of the first things that the application does when it starts up. Since we do not want to interfere in any way with the functionality of that class while operating within the IDE, we will leave that class alone. When a user requests that a file be opened or that a new file be created, it is functionality within the `ns-document-controller` that handles that request. For the most part it knows what to do because of a resource

file that is put into the application's bundle. Once again, we don't want to modify that file, so we can't directly make use of that functionality to open or create a new type of document. I instead created a sort of pseudo document controller which does many of the same things that a normal document controller would do, but only for our new class. I tried to do that in such a way that if we decide to create a stand-alone application, we can replace our custom document controller with the more standard one and everything will work just as one would normally expect. We use our pseudo document controller to do a few things that we can't otherwise do and then register our documents with the standard ns-document-controller and let it do the rest of the things that it can do just fine (like saving and closing files).

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

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

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

I'm not going to document the internal workings of the lisp-doc-controller class in this document. It isn't necessary for a developer to know how it functions in order to use it. The interested reader should feel free to look at lisp-doc-controller.lisp to see what's going on. I will say a couple of general things about what it does. One design goal for this class was to make it as easy as possible to later migrate from "within-IDE" document functionality to "stand-alone" document functionality. Another was to be able to support multiple different types of documents with this one single class. To address the second goal, this class is "document-class agnostic". It does not know or care what class of document is being created. If you wanted to support two or more different types of document you could do so with this class. In a standard stand-alone document application, only a single instance of ns-document-controller is used. For our purposes we will use a single instance of the lisp-doc-controller for each *class* of document that is to be supported. For this project we will need only one instance that will be created

when the loan-doc class is loaded.

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

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

```
(orig-date-text :accessor orig-date-text)
```

This will keep a reference to the origination date text field.

The only changes to the make-loan-window function are that when we create the text fields we disable the handling of undo by adding the following initialization argument to each of them:

```
:allows-undo nil
```

By default, all text fields do a rudimentary amount of undo and redo handling. They will allow you to undo and redo typing. For our purposes, we want a more robust form of undo. In general when one field is changed, any number of other fields may also change as a consequence. Just reversing that change will not guarantee that all the other values also return to their previous state. You will shortly see how a more general form of undo and redo is implemented. But for now just recognize that we want to stop the text fields from interfering with our undo mechanism.

Those are all of the changes needed for our loan-win-controller, so we will next discuss the loan-doc class.

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

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

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

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

The first method we will discuss is `#/makeWindowControllers`.

```
(objc:defmethod (#/makeWindowControllers :void)
  ((self loan-doc))
  (let ((lwc (make-instance 'loan-win-controller
    :build-method #'make-loan-window
    :loan self
    :should-close-document t)))
    (#/addWindowController: self lwc)))
```

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

Note that the window controller will now have two slots that point to our `loan-doc` object: `document` and `loan`. We do not necessarily need both. The `document` slot is necessary because it is used for standard `ns-window-controller` functionality, so we could change all of the KVC paths used to bind user interface objects to use the `document` slot rather than the `loan` slot and do away with the latter altogether. Part of the reason I didn't do this is just laziness. I didn't want to go back and change all of the binding paths. But as I considered doing this I also realized that the `document` slot in the `lisp-window-controller` would not be set until sometime after the window was created whereas the `loan` slot was set prior to this event. This could have created problems when the newly created interface objects began to ask for values via a slot that was not yet defined. I would have had to provide additional information for every binding regarding default values to use if the slot was not bound. Rather than walk down that path I chose to take the easy route and have two different slots.

### *Undo/Redo Functionality*

Before examining the next `loan-doc` methods we need to discuss how "undo" and "redo" normally work and how we want to implement them for our loan documents. Apples' functionality is really quite clever. Each document has its own `ns-undo-manager` instance that coordinates undo and redo operations whenever a window owned by that document has the current focus. The way this works is that whenever an operation that we want to undo is done, it creates the invocation that would be needed to reverse the action being taken and registers that action with the undo manager. That action should itself be "undoable". Assuming that the reverse action is in fact undoable, then when it is

invoked it will create and register its own reverse action. The undo manager knows that it is in the process of doing an undo when this happens and puts the reverse action on the "redo" stack instead of on the "undo" stack. All of this seems fairly straight-forward, but we need to think about how it applies to our loan application.

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

Another problem that we used to have to worry about, but no longer do, is managing the size of the undo stack. Recall that we made all the slider updates continuous to provide instantaneous feedback to the user about the effects that are generated by the change. If we regarded each of these individual changes as an action that could be undone, then there would be a huge number of actions on the undo stack and that would make the undo functionality virtually unusable. To address this problem Apple automatically groups undo actions that occur in situations like this. So we can effectively ignore the problem and everything works the way we would hope.

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

```
(defmethod create-undo ((self loan-doc) slot-name)
  (let ((st (get-loan-state self)))
    (set-undo self
      #'(lambda ()
          (create-undo self nil)
          (set-loan-state self st))
      (when slot-name
        (format nil "set ~a" (string-for-slot-name
                              slot-name))))))
```

The create-undo method is called as the first thing from every action that changes a loan parameter value. The file undo.lisp defines the set-undo macro which shields the Lisp programmer from some of the ugly details of interfacing with the undo manager. The Lisp developer can think of the undo facility as accepting a zero-argument closure that will be executed to accomplish an undo action and a string that will be used in the

undo menu to identify what exactly is being undone. In general, the undo actions effected by the closure will themselves be "undoable" and therefore will set up their own undo.

So the way to think about setting up undo actions for this project is that just before we make any change to the loan-doc (such as changing a loan parameter or setting the complete loan state), we will capture the current loan state and set up an undo action (closure) that will return the loan-doc object to that state.

Note that the reverse of the set-loan-state action that is done to accomplish an undo is just another set-loan-state call to put it back the way it was. That's why the closure created by create-undo when we change a slot value also works to "undo the undo". That's why there is a create-undo call as the first thing done by the closure.

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

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

```
(defmethod set-loan-state ((self loan-doc) state-list)
  (setf (loan-amount self) (pop state-list))
  (setf (interest-rate self) (pop state-list))
  (setf (loan-duration self) (pop state-list))
  (setf (desired-loan-duration self) (pop state-list))
  (setf (monthly-payment self) (pop state-list))
  (setf (origination-date self) (pop state-list))
  (setf (first-payment self) (pop state-list))
  (setf (hide-max-dur self) (pop state-list))
  (setf (hide-min-dur self) (pop state-list))
  (setf (hide-min-pay self) (pop state-list))
  (setf (pay-schedule self) nil)
  (compute-new-loan-values self))
```

The set-loan-state method takes a list of loan parameters and uses them to set the corresponding loan values. After that it calls the compute-new-loan-value method to

compute the new loan-schedule slot. Note that because all the slots were declared with :kvo options so any user interface objects that are bound to them will be instantly changed when the slot values are reset. Also note that any argument given to set-loan-state is used exactly once so there is no problem with destroying it using the pop calls.

To complete the undo functionality we must assure that every time a loan variable is modified by the interface user we create an undo. Fortunately for us the Lisp extension to the binding protocol that I create provides an easy way to do this. The *bound-slot-will-be-modified* method is called just before a slot value is set via a bind connection. We can use that method to create an undo for the change that is about to occur:

```
(defmethod bound-slot-will-be-modified ((self loan-doc) slot-name)
  ;; Create an undo to restore the current state before changing
  the slot value.
  (create-undo self slot-name))
```

### *Open/Save Functionality*

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

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

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

When an application is saved we want to force it to be saved with a ".loan" extension. To



do that we define the method below.

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

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

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

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

This method is called when a user elects to save a file. Having implemented the undo functionality, this is now a pretty easy thing to do. We get the loan state and package it up as an ns-data object. If you are curious about how Lisp lists are transformed into ns-data objects and then back to lists again, feel free to work your way through all the code. As long as we are given an equal object back we will be just fine. But in fact it turns out that for ns-data objects such as this the output file will simply contain a text representation of the list that we gave it. It is possible to use TextEdit or most any other text editing application (including CCL itself) to open it up and look at it.

If saving and restoring our document required more complex and/or interconnected lisp objects, then we would certainly want to do something other than create a simple list. A general mechanism to do this for arbitrary Lisp data types including class instances exists and is demonstrated in the examples shown in the *CCL Cocoa Developer Tools Tutorial*.

```
(objc:defmethod (/#/readFromData:ofType:error: #>BOOL)
  ((self loan-doc) (data :id) (dtype :id) (err
  (:* :id)))
  (declare (ignore err dtype))
  (set-loan-state self (coerce-obj data 'list))
  #$YES)
```

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

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

### *Print Functionality*

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

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

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

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

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

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

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

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

Our loan-print-view class is entirely new, so we will examine it and its methods in some detail.

```
(defclass loan-print-view (ns:ns-view)
  ((loan :accessor loan
         :initarg :loan)
   (attributes :accessor attributes
```

```

        :initform (make-instance ns:ns-mutable-
dictionary))
    (page-line-count :accessor page-line-count
        :initform 0)
    (line-height :accessor line-height)
    (page-num :accessor page-num
        :initform 0)
    (page-rect :accessor page-rect))
    (:metaclass ns:+ns-object))

```

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

```

(defmethod initialize-instance :after ((self loan-print-view)
                                       &key &allow-other-keys)
  ;; assure loan still exists if user closes the window while we
  are printing
  (retain (loan self))
  (let* ((font (font (fontWithName:size: ns:ns-font @"Courier"
8.0)))
        (setf (line-height self) (* (+ (ascender font) (abs (descender font))) 1.5))
        (setf (font self) font))
    (setf (font self) font))

```

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

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

```

(objc:defmethod (dealloc :void)
  ((self loan-print-view))

```

```

(#!/release (loan self))
(#!/release (attributes self))
(call-next-method)
(objc:remove-lisp-slots self))

```

As previously noted, the `#!/dealloc` method will release objects that it owns. The `call-next-method` and `objc:remove-lisp-slots` are required (in this order) as noted in the first discussion of `#!/dealloc` methods in this document.

```

(objc:defmethod (#!/knowsPageRange: :<BOOL>)
  ((self loan-print-view) (range (:* #>NSRange)))
  ;; compute printing parameters and set the range
  (let* ((pr-op (#!/currentOperation ns:ns-print-operation))
        (pr-info (#!/printInfo pr-op))
        (image-rect (#!/imageablePageBounds pr-info))
        (pg-rect (ns:make-ns-rect 0
                                   0
                                   (ns:ns-rect-width image-rect)
                                   (ns:ns-rect-height image-rect))))
    (setf (page-rect self) pg-rect)
    (#!/ setFrame: self pg-rect)
    (setf (page-line-count self) (floor (ns:ns-rect-height pg-rect)
                                         (line-height self)))

    ;; start on page 1
    (setf (ns:ns-range-location range) 1)
    ;; compute the number of pages for 9 header lines on page 1
    and 2 header
    ;; lines on subsequent pages plus a line per payment
    (let* ((pay-lines-on-p-1 (- (page-line-count self) 7))
          (other-pages-needed (ceiling (max 0 (- (list-length
                                                    (pay-schedule (loan self))
                                                    pay-lines-on-p-1))
                                         (page-line-count self)))))
      (setf (ns:ns-range-length range)
            (1+ other-pages-needed))))
  #YES)

```

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

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

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

The `#/imageablePageBounds` call resulted in

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

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

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

```
(objc:defmethod (#/rectForPage: #>NSRect)
  ((self loan-print-view) (pg #>NSInteger)))
```



```

loan) 'string)))
      (draw-next-line (format nil
                              "Amount: $~$"
                              (/ (loan-amount loan) 100)))
      (draw-next-line (format nil
                              "Origination Date: ~a"
                              (date-string (origination-date
loan))))))
      (draw-next-line (format nil
                              "Annual Interest Rate: ~7,4F%"
                              (* 100 (interest-rate loan))))
      (draw-next-line (format nil
                              "Loan Duration: ~D month~:P"
                              (loan-duration loan)))
      (draw-next-line (format nil
                              "Monthly Payment: $~$"
                              (/ (monthly-payment loan)
100)))
      ;; draw spacer line
      (incf (ns:ns-rect-y line-rect) line-height))
      ;; print the appropriate schedule lines for this page
      (let* ((lines-per-page (- page-line-count (if (zerop
page-num) 7 0)))
              (start-indx (if (zerop page-num) 0 (+ (- page-
line-count 7)
                                                         (* lines-
per-page (1- page-num)))))
              (end-indx (min (length (pay-schedule loan))
                             (+ start-indx lines-per-page 1))))
              (dolist (sched-line (subseq (pay-schedule loan) start-
indx end-indx))
                (draw-next-payment sched-line))))))

```

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

This method makes use of the ability of an `ns-string` to draw itself within a specified rectangle using specified attributes. Basically what we do here is create such a rectangle at the top of the page that spans the width available and as each line is



printed we move it down the page. We move the rectangle down prior to printing, so it is initialized above the actual position desired for the first line.

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

The rest of the `#/drawRect:` method is straight-forward. Header lines are first formatted and printed if we are printing page zero. Then we extract the appropriate sub-sequence of loan payment lines from the loan-doc's pay-schedule and print each of them on a subsequent line. Each payment-line is a list of values, so it is easy to just pass that list to the `draw-next-payment` function as discussed above. Arguably we should grab the pay-schedule and copy it once at the beginning of printing so that if the user changes some parameter in the window while printing is being done it won't affect what is printed. In practice it all seemed to happen pretty quickly, but if you are concerned about this, go ahead and make this change as well.

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

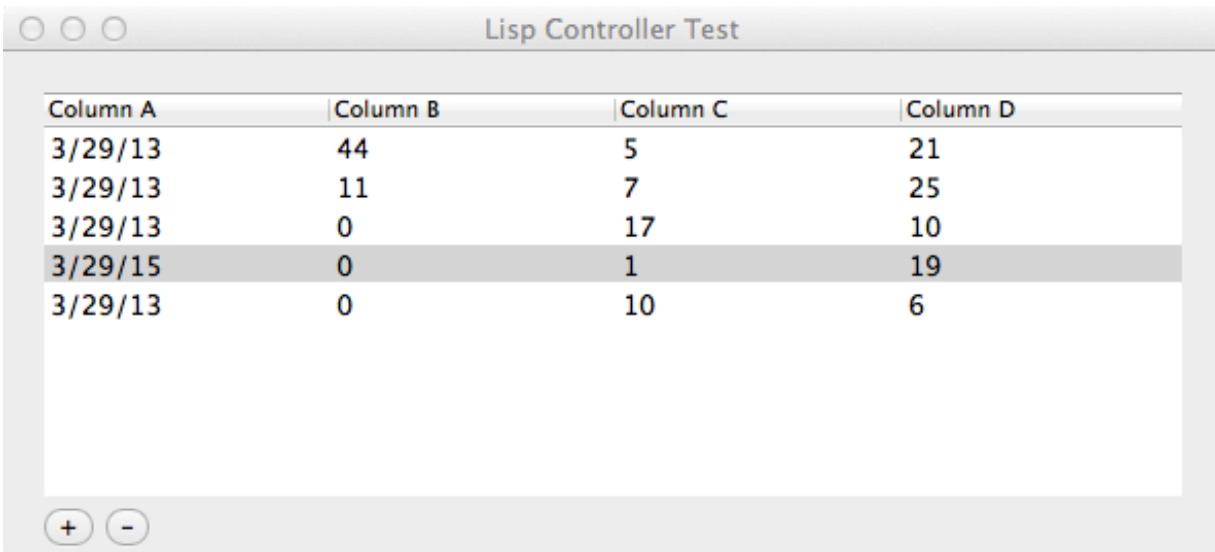
## **Project 8: Other functions of lisp-controllers**

Key Concepts: lisp-controller keyword arguments `:func-owner` `:select-func` `:edited-func` `:added-func` and `:removed-func`, adding and removing objects from displayed tables, formatters for table columns

This example shows how to configure a lisp-controller to generate new objects that are displayed in an `ns:ns-table-view`. It shows how to configure interface buttons to add and remove objects from a table. It shows how to use formatters for table columns both to control how data is displayed and to provide hints that tell the lisp-controller how data conversion should be done. Finally, it contains example notification functions for all types of notification. This example doesn't do anything too useful, but it nicely illustrates a number of different lisp-controller features.

The source code for this project is in ...CocoaInterfaces/Example Projects/Controller

Test/controller-test.lisp. The window that we will create will look as shown below:



Column A	Column B	Column C	Column D
3/29/13	44	5	21
3/29/13	11	7	25
3/29/13	0	17	10
3/29/15	0	1	19
3/29/13	0	10	6

There is a four-column `ns:ns-table-view` with two buttons below it. The buttons add and delete rows from the table. The table will be sorted by descending order of the value in Column B. Column A will display a date, Column B will be configured to always display an integer number. Columns C and D can take any type of value, but note that the `lisp-controller` always tries to preserve the type of data that is already in each cell. Since we will initialize them with integers, the `lisp controller` will try to convert anything typed into those columns by the user to an integer. If it cannot do that type coercion, then it will accept whatever is typed as a string. From that point forward, anything typed into that cell will be converted to a string. New values typed into other cells in the same column will continue to be converted to integers if possible. Generally, if you want to enforce that the value in any particular column is of a particular type, then you should attach a formatter to that column. We'll show examples of that for Column A and Column B.

As with all examples, this one is put into its own package (`:ct`) so you can safely experiment with it without worrying about interfering with your own code. Let's walk through that code.

```
(defclass lisp-controller-test (lisp-window-controller)
  ((lisp-ctrl :accessor lisp-ctrl))
  (:metaclass ns:+ns-object))
```

We will subclass the `lisp-window-controller` just to have an easy way to access the `lisp-controller`. Since we let the user add rows to the root object (which will be `nil` initially), we might want to have a way to access it later to get the new root object.

```
(defun make-dated-list ()
  (list (now) 0 (random 20) (random 30)))
```

The `make-dated-list` function will be used to create new row objects when the user clicks the + button in the window. The `now` function is defined in `date.lisp` and returns a lisp date. Since this is just an integer value, you might wonder how the code knows enough to turn it into a date format when it gets displayed in the table. The lisp-controller looks for formatter objects in the views that it supplies data for and when it sees a date formatter for a column it will store that information so that when it comes time to supply data for the column it will assume that the Lisp integer it is given is in date format and convert it to an `ns:ns-date` object that is passed back as the value of the cell being displayed. Similarly, if the user modifies that date, then the updated `ns-date` that the Lisp controller is given will be converted back to a Lisp date.

```
(defmethod selected-cell ((self lisp-controller-test) controller
root row-num col-num obj)
  (declare (ignore controller root))
  (cond ((and (minusp row-num) (minusp col-num))
        (ns-log "Nothing selected"))
        ((minusp row-num)
         (ns-log (format nil "Selected column ~s with title ~s"
col-num obj)))
        ((minusp col-num)
         (ns-log (format nil "Selected row ~s: ~s" row-num
obj))))
        (t
         (ns-log (format nil "Selected ~s in row ~s, col ~s"
obj row-num col-num))))))
```

The `selected-cell` method will be called when a new cell is selected in the table. Just for demonstration purposes, this function will log information about what was selected. You will find these messages in the console log on your system. The arguments passed are defined by the `lisp-controller` class.

```
(defmethod edited-cell ((self lisp-controller-test) controller
root row-num col-title obj old-val new-val)
  (declare (ignore controller root obj))
  (ns-log-format "Changed ~s in row ~s, ~a to ~s"
old-val row-num col-title new-val))
```

The `edited-cell` method is called when a table cell is modified by the user.

```
(defmethod added-row ((self lisp-controller-test) controller
root parent new-row)
  (declare (ignore controller root))
  (ns-log-format "Added ~s to ~s" new-row parent))

(defmethod removed-row ((self lisp-controller-test) controller
root parent old-row)
  (declare (ignore controller root))
```

```

(ns-log-format "Removed row ~s from ~s " old-row parent))

(defmethod make-ctest-window ((wc lisp-controller-test))
  (let* ((lc (make-instance 'lisp-controller
    :col-ids (list "ColA" "ColB" "ColC" "ColD")
    :root nil
    :initform '(make-dated-list)
    :sort-key #'second
    :sort-pred #'>
    :col-keys (list #'first #'second #'third
#'fourth)
    :func-owner wc
    :select-func #'selected-cell
    :edited-func #'edited-cell
    :added-func #'added-row
    :removed-func #'removed-row)))
    (add-button (make-instance ns:ns-button
      :title "+"
      :action "insert:"
      :target lc
      :bezel-style :round-rect
      :button-type :momentary-light))
    (del-button (make-instance ns:ns-button
      :title "-"
      :action "remove:"
      :target lc
      :bezel-style :round-rect
      :button-type :momentary-light))
    (df (*/autorelease (make-instance ns:ns-date-formatter
      :date-style :short
      :time-style :none)))
    (nf (*/autorelease (make-instance ns:ns-number-
formatter)))
    (col-a (*/autorelease (make-instance ns:ns-table-column
      :column-title "Column A"
      :identifier "ColA"
      :min-width 60
      :editable t
      :formatter df)))
    (col-b (*/autorelease (make-instance ns:ns-table-column
      :column-title "Column B"
      :identifier "ColB"
      :min-width 60
      :editable t
      :formatter nf)))
    (col-c (*/autorelease (make-instance ns:ns-table-column
      :column-title "Column C"

```

```

                                :identifier "ColC"
                                :min-width 60
                                :editable t)))
(col-d (/autorelease (make-instance ns:ns-table-column
                                :column-title "Column D"
                                :identifier "ColD"
                                :min-width 60
                                :editable t)))
(tv (make-instance 'ns:ns-table-view
    :columns (list col-a col-b col-c col-d)
    :data-source lc
    :delegate lc
    :allows-column-resizing t
    :column-autoresizing-style :uniform))
(sv (make-instance ns:ns-scroll-view
    :autohides-scrollers t
    :has-horizontal-scroller t
    :has-vertical-scroller t
    :document-view tv))
(win (make-instance 'ns:ns-window
    :title "Lisp Controller Test"
    :resizable t
    :content-subviews (list sv add-button del-
button)))
(cview (/contentView win)))

;; set the lisp-ctrl field in the window controller
(setf (lisp-ctrl wc) lc)

;; set the view for the lisp-controller
(setf (view lc) tv)

(bind add-button "enabled" lc "canInsert")
(bind del-button "enabled" lc "canRemove")

(anchor cview sv (list :leading :top :trailing))
(order-views :orientation :v
    :views (list sv add-button :margin)
    :align :leading)
(order-views :orientation :h
    :views (list add-button del-button)
    :align :center-y)

;; Return the window, the list of objects created
(values win (list lc add-button del-button tv sv win)))

(defun test-controller ())

```

```
(on-main-thread
  (let ((wc (make-instance 'lisp-controller-test
                          :build-method #'make-ctest-window)))
    (show-window wc)
    wc)))
```

## **Project 9: Card Dealer**

Key Concepts: hash-tables as roots of lisp-controllers

This example is intended to show how the LispController works in conjunction with hash-tables. In fact, not only is the root object a type of hash-table, so are its children and grandchildren. The application deals out four hands of cards (North, South, East, and West) as in a bridge game from a conventional 52-card deck. The display shows each hand and each hand can be expanded to show four suits. A button is defined to deal out a new hand. This isn't a particularly useful application, but does nicely demonstrate the use of hash-tables. And I admit that I have spent some time dealing out hands, showing just the North/South cards, and imagine how I would bid and play them in a bridge match. Then I reveal the East/West cards to figure out what I should have done instead. If you are a bridge player, you might find this amusing.

At runtime after complete expansion of displayed rows the window will look something like this:

Deal It!	
Position and Suit	Cards
▼ North	
Spades	Q, 4, 3
Hearts	Q, 7
Diamonds	8, 4, 2
Clubs	K, Q, J, 5, 3
▼ East	
Spades	10, 8, 5, 2
Hearts	9
Diamonds	J, 10, 7, 5, 3
Clubs	A, 10, 9
▼ South	
Spades	A, K, J, 9, 7
Hearts	10, 8, 4, 3
Diamonds	K, Q
Clubs	6, 2
▼ West	
Spades	6
Hearts	A, K, J, 6, 5, 2
Diamonds	A, 9, 6
Clubs	8, 7, 4

New Deal

The positions North, East, South, and West are always shown in this order. And the suits are always displayed in the order shown (which will be familiar to bridge players). For this project it may be easier to look at the Lisp data side of the design first and then design the interface. I am a strong proponent of a data-first design methodology, but my experience is that a sort of iterative approach is probably best: first design the core data structures and data manipulation functionality, next design the interface, and finally design any control/integration functionality needed to bring the two together. So let's see if we can follow that path for this project.

The source code for this project is in ...CocoaInterfaces/Example Projects/Cards/cards.lisp. For purposes of this project we want to represent a complete deal of the cards; i.e. four hands, each with four suits, and some number of cards in each suit. There are, of course, many possible ways that we might choose to implement this and I'll admit that my choice was perhaps a bit artificial and made primarily because it

demonstrates how hash-tables are handled by the lisp-controller. I chose to represent a whole deal as an instance of an assoc-array. What, you may well ask, is that?

An assoc-array is a class that I created that lets me make a multi-dimensional sparse array that can be indexed by arbitrary lisp objects (anything that could be used as a key in a hash table). I have found this to be a useful way of representing lots of miscellaneous data and in fact these are used in several places in the implementation of the lisp-controller class itself.

Assoc-arrays are defined in the file ...CocoaInterfaces/Utilities/assoc-array.lisp. An assoc-array is basically a hierarchical set of hash-tables. An assoc-array object contains a root hash-table. Each key in that table is some Lisp value that has been used as the first index when storing into the assoc-array. The value corresponding to that key will either be another hash-table if this is not the last dimension of the assoc-array or the value being stored. In an N-dimensional assoc-array then, there would be N-1 levels of embedded hash-tables and the last index would be the key into the most deeply embedded table. Hash tables are only created as needed to store new values.

OK, so now that you know a little bit more about what an assoc-array is, we'll show how to use one to represent four bridge hands. Later we will display it in an NSOutlineView using our lisp-controller. All the following code is defined in package :cards as shown in cards.lisp. At this point we will only look at the parts of that file that define the data structures (i.e. the "model" from Apple's Model/View/Controller paradigm). After that we will define the interface much as we did for previous projects.

We will represent any combination of cards from a deck as a bit vector that is 52 bits long. Cards included have a corresponding bit value of 1. Cards not included have a corresponding bit of 0. This representation can be used to represent a single hand, a complete deck, a single suit, or any other card combination. For example, we define some useful constants as follows:

```
(defconstant *aces*
#*100000000000010000000000001000000000001000000000000)
(defconstant *kings*
#*010000000000001000000000000100000000000100000000000)
(defconstant *queens*
#*001000000000001000000000000100000000000100000000000)
(defconstant *jacks*
#*00010000000000010000000000010000000000010000000000)
(defconstant *spades*
#*111111111111100000000000000000000000000000000000000)
(defconstant *hearts*
#*000000000000011111111111100000000000000000000000000)
(defconstant *diamonds*
#*00000000000000000000000001111111111110000000000000)
(defconstant *clubs*
#*00000000000000000000000000000000000000000111111111111)
```



In addition, we want to define some constants to represent card ranks and suits:

```
(defconstant *card-ranks* '("A" "K" "Q" "J" "10" "9" "8" "7" "6"
"5" "4" "3" "2"))
(defconstant *card-suits* '("Spades" "Hearts" "Diamonds"
"Clubs"))
(defconstant *hand-suits* '("Spades" "Hearts" "Diamonds" "Clubs"
"North" "East" "South" "West"))
```

Using all of these constants we can easily create two utility functions that tell us about any particular card.

```
(defun card-rank (card)
  ;; card is a bit index
  (nth (mod card 13) *card-ranks*))

(defun card-suit (card)
  ;; card is a bit index
  (nth (floor card 13) *card-suits*))
```

To deal cards we will start with a full deck and randomly remove cards from it one by one.

```
(defun deal-cards ()
  ;; randomizes and returns four unique hands
  (let ((deck (full-deck))
        (deal (make-instance 'assoc-array :rank 2))
        (card nil))
    (dotimes (i 13)
      (setf card (pick-random-card deck))
      (add-card deal "West" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "North" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "East" card)
      (remove-card card deck)
      (setf card (pick-random-card deck))
      (add-card deal "South" card)
      (remove-card card deck))
    deal))
```

The deal parameter defined within the let form just creates a two-dimensional assoc-array. The indices of this assoc-array will be the position (i.e. North, East, South, or West) and the suit of the card. The value indexed for any position and suit combination

will be a list of the ranks of the cards of the specified suit dealt to the specified position. The function was deliberately designed to represent the way that a dealer would hand out cards; viz. one at a time in rotation. In the following we'll look at the functions called in a little more detail.

```
(defun full-deck ()
  (make-array '(52) :element-type 'bit :initial-element 1))
```

The full-deck function simply makes a bit-vector with all bits set.

```
(defun pick-random-card (deck)
  ;; returns a card index
  (let* ((cnt (count 1 deck))
        (card (1+ (random cnt))))
    (position-if #'(lambda (bit)
                     (when (plusp bit)
                         (decf card)
                         (zerop card)
                         deck)))
```

There are almost certainly more efficient mechanisms for selecting a random card from the set, but in practice the previous function seems to work fast enough. It counts the number of cards left and then picks one of them at random, returning the bit index that refers to that card.

```
(defun add-card (deal hand card)
  (setf (assoc-aref deal hand (card-suit card))
        (cons (card-rank card) (assoc-aref deal hand (card-suit
card)))))
```

The add-card function adds the rank of the card to the list of ranks that is the value of the assoc-array for the specified position (as given by the hand parameter) and suit of the card.

```
(defun remove-card (card deck)
  ;; card is a bit index
  (setf (aref deck card) 0))
```

This function removes the card from the deck by setting the corresponding bit to 0.

This completes the definition of the basic data structures. Next we will define an interface to display deals. We want to have an object that responds to a click of the *New Deal* button. For a normal button that must be an Objective-C object, so we could create a special class of object just for that, but it's just as easy to subclass the lisp-window-controller and let it do this for us. We'll call that subclass hand-of-cards:

```
(defclass hand-of-cards (lisp-window-controller)
```

```
((lisp-ctrl :accessor lisp-ctrl))
(:metaclass ns:+ns-object))
```

The `lisp-ctrl` slot exists so that when a new deal is requested we can set a new root object for the lisp-controller object that provides data to the table.

```
(Objc:defmethod (#/deal: :void)
  ((self hand-of-cards) (sender :id))
  (declare (ignore sender))
  (unless (eql (lisp-ctrl self) (%null-ptr))
    (setf (root (lisp-ctrl self)) (deal-cards))))
```

The Objective-C method *deal* is called when the *New Deal* button is pushed. It simply sets the root of the lisp-controller.

```
(defun first-ht (aa)
  ;; hand will be an assoc-array
  (iu::index1-ht aa))
```

The `first-ht` method takes advantage of knowledge that we have about how `assoc-array` objects are implemented to retrieve the highest-level hash-table. We'll see how this is used shortly. Next we define several functions that will be useful when specifying how to display data in the `ns-outline-view`

```
(defun first-ht (aa)
  ;; hand will be an assoc-array
  (iu::index1-ht aa))
```

The `first-ht` function will be specified as the `:root-child-key` for the lisp-controller we'll create. This just returns the top-level hash-table from the `assoc-array` that will be the root of the lisp-controller. When a hash-table is returned as a representation of an object's children, the lisp-controller will convert it into a list of key-value pairs. In this case, there will be one key for each of the four positions (North East South and West) and the value of each key will be another hash-table.

```
(defun hand-suit-order (a b)
  (< (position a *hand-suits* :test #'string=)
     (position b *hand-suits* :test #'string=)))
```

The `hand-suit-order` function will be specified as the `sort-predicate` for all row objects. Since those objects might have either positions as keys or suits as keys (depending on how deeply nested we are), the sort predicate must be able to handle either case. So we made a single ordering that works for either that we called `*hand-suits*`. This is a bit of a kludge and it might have been somewhat clearer to define separate predicates for each level of the table. If you're ambitious, feel free to modify the code to make this happen.

For the second column at the second level, we will specify a column key that uses a function called `sorted-by-rank`. We do a type check, just to assure that the object is what we expect.

```
(deftype rank-list ()
  '(satisfies all-ranks))

(defun all-ranks (rank-list)
  (and (listp rank-list)
       (null (set-difference rank-list *card-ranks* :test
                              #'string=))))

(defun higher-rank (r1 r2)
  (< (position r1 *card-ranks* :test #'string=) (position r2
                                                           *card-ranks* :test #'string=)))

(defun sorted-by-rank (rlist)
  (when (typep rlist 'rank-list)
    (format nil "~{~a~^, ~}" (sort-list-in-place rlist #'higher-rank))))
```

Let's next look at the `make-deal-window` method which creates the window.

```
(defmethod make-deal-window ((hand hand-of-cards))
  (let* ((lc (make-instance 'lisp-controller
                           :col-ids (list "PosSuit" "Cards")
                           :root-child-key #'first-ht
                           :child-key-0 #'ht-value
                           :sort-key #'ht-key
                           :sort-pred #'hand-suit-order
                           :col-keys-0 (list #'ht-key)
                           :col-keys-1 (list #'ht-key #'(lambda (hte)
                                                            (sorted-by-rank
                                                             (ht-value hte)))))))
```

We first create the `lisp-controller` object that will provide data to the `ns-outline-view` that will form the main part of the window. This is a good example of how we can set parameters when each level of the hierarchy being displayed must display a different sort of object. By the way, this is something that is very difficult to do with Apple's Objective-C controllers. We first specified that there would be two columns, identified as "PosSuit" and "Cards".

The root object will be an `assoc-array` and `lisp-controllers` don't know anything about how to find children objects of such a critter, so we provide a `:root-child-key` argument to do that. This used the `first-ht` method that was discussed above to retrieve the top-level hash-table of the `assoc-array`. Now hash-tables are something that `lisp-controllers` do understand. When it gets back a hash-table as the "children" of the root, it will treat

each of its key-value pairs as a child object. It gets a bit more complicated than that because we want to permit the user to edit those objects and have those changes reflected within the original root object, but for now you can think of the children of a hash-table as a list key-value-pair objects. Each child row is represented by one of those. As you might expect, what we want to display will generally be some function of either the key or value from that pair. Similarly, we might expect that finding the children of such a key-value pair would also be a function of either the key or the value. To facilitate these data accesses, two accessor functions `ht-key` and `ht-value` are provided to retrieve the key and value, respectively, from the row object. In this case we know that the value of the key-value pair within a two-dimensional assoc-array is another hash-table that will effectively provide the children row objects at the next level. So we tell the lisp-controller that by using the `:child-key-0` argument and telling it to just use the value of the key-value pair that represents the row.

Next we provide functions for sorting the display. We could have provided separate `:sort-key` and `:sort-pred` arguments for each level, but since we cleverly arranged to create a single sort predicate that works at either level, we can just use the `:sort-key` and `:sort-pred` arguments that will apply to all levels of the hierarchy.

Next we provide functions for displaying the columns for the highest level rows using the `:col-keys-0` keyword argument. But note that the list of functions only contains a single function, rather than one for each column. That's because in this case we do not want to display anything in the second column at the highest level. We effectively provided `nil` as the function, which causes nothing to be displayed. For level 1 rows we provide the `:col-keys-1` argument which specifies that the first column should display the key of the key-value pair and the second column should display the result of applying the sorted-by-rank function to the value of the key-value pair.

```
(deal-button (make-instance ns:ns-button
                           :title "New Deal"
                           :action "deal:"
                           :target hand
                           :bezel-style :round-rect
                           :button-type :momentary-light))
```

The *New Deal* button is straight-forward. When pushed, it will send the "deal:" message to the window-controller.

```
(hand-col (/#autorelease (make-instance ns:ns-table-
column
                           :column-title "Position and
Suit"
                           :identifier "PosSuit"
                           :min-width 60
                           :editable nil
                           :selectable nil)))
(suit-col (/#autorelease (make-instance ns:ns-table-
```

column

```
:column-title "Cards"  
:identifier "Cards"  
:min-width 60  
:editable nil  
:selectable nil)))
```

The two column objects are straight-forward.

```
(tv (make-instance 'ns:ns-outline-view  
  :columns (list hand-col suit-col)  
  :data-source lc  
  :delegate lc  
  :allows-column-resizing nil  
  :column-autoresizing-style :uniform  
  :outline-table-column hand-col))
```

The ns-outline-view is also simple to create.

```
(sv (make-instance ns:ns-scroll-view  
  :autohides-scrollers t  
  :has-horizontal-scroller t  
  :has-vertical-scroller t  
  :document-view tv))
```

The outline view is then put into an ns-scroll-view.

```
(win (make-instance 'ns:ns-window  
  :title "Deal It!"  
  :resizable t  
  :content-subviews (list sv deal-button)))
```

The window is created to contain the scroll view and the button.

```
(cview (#/contentView win)))
```

We get the window's content-view to use when creating window constraints. First set the lisp-ctrl field in the window-controller and the view field in the lisp-controller.

```
;; set the lisp-ctrl field in the window controller  
(setf (lisp-ctrl hand) lc)  
  
;; set the view for the lisp-controller  
(setf (view lc) tv)
```

Constraints for this window are extremely easy:

```
(anchor cview sv (list :leading :top :trailing))
(order-views :orientation :v
              :views (list sv deal-button :margin)
              :align :leading)
```

Finally we return the window and a list of object to be managed to the window controller.

```
;; Return the window, the list of objects created
(values win (list lc deal-button tv sv win)))
```

The final bit of Lisp code needed is a test function that we can use in the listener to start things off:

```
(defun test-deal ()
  (on-main-thread
   (let ((wc (make-instance 'hand-of-cards
                           :build-method #'make-deal-window)))
     (show-window wc)
     wc)))
```

In the listener type (cards:test-deal) to open up the window.

## Appendix A: Debugging Hints

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

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

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

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

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

```
> Error: value #1=((2 3 2 2) . #1#) is not of the expected type (SATISFIES
CCL::PROPER-LIST-P).
```

```
> While executing: (:INTERNAL LISP-CONTROLLER::I-[LispController
numberOfRowsInTableView:|I], in process Initial(0).
```

```
> Type :POP to abort, :R for a list of available restarts.
```

```
> Type :? for other options.
```

```
1 > :r
```

```
> Type (:C <n>) to invoke one of the following restarts:
```

```
0. Return to break level 1.
```

```
1. #<RESTART ABORT-BREAK #x493A1D>
```

```
2. Process the next event
```

```
1 > (:C 2)
```

```
Invoking restart: Process the next event
```

## Appendix B: Layout Constraint API

Constraints specify an equality that would be described in lisp terms as:

```
(<rel> (<property> <object1>)
      (+ (* (<property> <object2>)
           <multiplier>)
         <constant>))
```

where:

```
<rel>      ::= = | >= | <=
```

```
<property> ::= top | left | right | bottom | leading |
```

```
trailing | width | height | center-x | center-y | baseline |
```



```
none
<object1>      ::=    <NSView object>
<object2>      ::=    <NSView object> | <null object>
<multiplier>   ::=    <number>
<constant>     ::=    <number>
```

I found this lispish way of thinking about what a constraint can be to be so useful that I created the "constrain" method which takes a list in the above form and turns it into an appropriate Objective-C call to make a constraint for you. For example, you could say something like:

```
(constrain cview (<= (leading cview) (leading arg1)))
```

where cview and arg1 are variables defined in the local environment which contain views. Note that the example does not include the + or \* clauses. The constrain method is flexible enough so that any input that can be put into the form above will automatically be converted. You can even exchange the order of top level arguments and constrain will convert appropriately. To create individual constraints, this is the most convenient form.

In addition, views have a priority for how resistant they are to expansion of their current size and how resistant they are to compression from their current size. Priorities range from 0 to 100. This helps the layout system decide what to do.

Finally constraints themselves can have a priority ordering. These also range from 0 to 100, where 100 designates a mandatory constraint. So far, I have found that the ability to set constraint priorities is only rarely needed.

From this relatively constrained form a surprisingly large number of relationships can be specified. On top of this basic form, I have added a variety of methods that provide new constraint idioms that hopefully make the process easier. In addition to creating constraints, these methods immediately add the constraints to a parent-window that is passed in as the first argument to each method. That view can also be one of the views constrained, as when a view is constrained to have some relationship relative to its container view.

API method descriptions are listed below. I have tried to export all functions that might be useful to people who want to create their own constraint idioms. If you come up with one that you are especially proud of and think it would be useful to others, let me know and we can perhaps include it with mine.

All of these are defined in the constraint-layout.lisp file.

### ***Methods for creating constraints and adding them to a parent-view***

```
align-views (&key
             (install-view nil install-view-provided)
             (priority nil)
             (align :center-x))
```

**(views nil))**

Aligns the views specified using the alignment property specified (e.g. :left :right :top ...).

**anchor ((parent-view ns:ns-view) (anchored-view ns:ns-view) key-list  
&key  
(priority nil)  
(install-view parent-view)  
(margin \*border-space\*)  
(rel :=))**

Keep a view within a specified margin of one of the parent-view borders. The key-list is a combination of the keys (:top :bottom :left :right :leading :trailing). If the normal viewing order of the system is left-to-right, then :left and :leading are synonymous and :right and :trailing are synonymous. If the normal viewing order is right to left, then :leading and :right are synonymous and :trailing and :left are synonymous. For portability, :leading and :trailing are preferred to :right and :left.

**anchor-window ((win ns:ns-window) orientation fixed-point)**

When the window is resized, this assures that the designated fixed-point attribute remains fixed on the screen. The orientation is either :h or :v and the fixed-point is any of the standard view attribute

keywords: :top, :bottom, :left, :right, :leading, :trailing, :height, :width.

**center-in-view ((parent-view ns:ns-view) (centered-view ns:ns-view)  
&key  
(install-view parent-view)  
(priority nil))**

Center the centered-view in the middle of the parent-view. This can be useful when you want to design your window from the inside out, with some view always in the center.

**center-relative-to ((centered-on-view ns:ns-view)  
&key  
(install-view (superview centered-on-view))  
(priority nil)  
(pos :below)  
(min-dist \*default-space\*)  
(dist nil)  
(views nil)  
(align nil))**

Center a set of ordered views relative to some other view. For example, imagine wanting to center a group of three buttons underneath a table. The views argument would contain the buttons which would be ordered left to right and the entire group would be centered under the table view. The pos argument specifies the relative direction from the centered-on-view and can be any of: {:left :before :leading}

{:right :after :trailing} {:above :top} {:below :bottom} where synonyms are shown together in the braces. The min-dist argument specifies that the views must be at least that far from the centered-on-view and the dist argument specifies that they must be *exactly* that far from the centered-on-view. The align argument specifies the view attribute that should be aligned on all views.

### **constrain (lisp-constraint-form &rest other-keys)**

This is a macro that transforms the lisp-constraint-form to create a single constraint. The lisp-constraint-form argument should have the lisp syntax specified at the beginning of this appendix. Other allowed keys include :priority and :install-view. Examples of valid constraints might be:

```
(constrain cview (>= (leading arg1) (trailing arg2)))
```

to cause the arg2 view to be to the right of arg1

```
(constrain cview (>= (* 3 (width arg1)) (height arg1)) :priority 90 :install-view cview)
```

to cause the width of arg1 to always be at least 1/3 of its height. Note in this case that the first argument contains the multiplier rather than the second, as shown in the lisp syntax above.

It is also possible to specify a constrain form that uses a variable in place of the <rel> operator or an <attribute> operator, but that variable must evaluate to an appropriate keyword. For example, you could add something like: (constrain (rel (att1 view1) (att2 view2))) and the local variables rel, att1, and att2 must evaluate to appropriate keywords. The variable rel should evaluate to one of :=, :>=, or :<= and the variable att1 should evaluate to one of :top, :bottom, ...

The constrain method will automatically convert any valid expression that it can into a set of suitable keyword arguments which are passed to the make-constraint method which creates the constraint. You can find many other examples of such calls within this tutorial.

**constrain-size ((sized-view ns:ns-view)**  
    **&key**  
    **(install-view sized-view)**  
    **(priority nil)**  
    **(min-width nil)**  
    **(max-width nil)**  
    **(min-height nil)**  
    **(max-height nil)**  
    **(width nil)**  
    **(height nil))**

Constrain the min, max, or exact size to specific point values. With a single call you can specify both min and max sizes if desired.

**constrain-size-relative-to ((sized-view ns:ns-view) (relative-view ns:ns-view)  
    &key  
    (priority nil)  
    (install-view (common-superview sized-view relative-view))  
    (rel :=)  
    (width t)  
    (height t))**

Constrain the width and/or height of one view to be =, >=, or <= to some other view. This can easily be done with a simple constrain call as well.

**constrain-to-array (col-count  
    &key  
    (install-view nil install-view-provided)  
    (h-space \*default-space\*)  
    (v-space \*default-space\*)  
    (h-align :top)  
    (v-align :leading)  
    (priority nil)  
    (views nil))**

Layout the views within the parent in the number of rows required for the number of columns specified. The :views argument should be a list of views with no embedded numbers for spacing. If present, such numbers will be ignored. This could be done trivially if we could specify positional constraints relative to the size of the parent, but unfortunately that is not permitted. Constraints must relate sizes to other sizes or positions to other positions. So ... The largest width of any view and the largest height of any view are computed and assumed to specify the size of each array element. The natural-size of each view, which will always be non-zero, is used to make that determination although it may not be correct. When in doubt, make sure that the views have reasonable non-zero frame rectangle values.

**constrain-to-natural-size ((sized-view ns:ns-view)  
    &key  
    (priority nil)  
    (install-view sized-view)  
    (rel :=))**

Constrain the size to be =, >=, or <= to its natural size. The natural size is determined in a number of different ways, depending on the type of view and other factors. If the user set the size explicitly (by setting the object's frame) when the object was created, then that size is used as the normal size. If no normal size can be determined the view will be arbitrarily constrained to a size of 60 x 20 points and the fact that this happened will be logged into the system log. You can look there if you suspect this might be happening.

**constrain-to-natural-height ((sized-view ns:ns-view)**

```
&key  
(priority nil)  
(install-view sized-view)  
(rel :=)
```

Constrains the sized-view height only to be its natural height.

```
constrain-to-natural-width ((sized-view ns:ns-view)  
  &key  
  (priority nil)  
  (install-view sized-view)  
  (rel :=))
```

Constrains the sized-view width only to be its natural width.

```
distribute-relative-to ((centered-on-view ns:ns-view)  
  &key  
  (install-view (superview centered-on-view))  
  (priority nil)  
  (pos :below)  
  (min-dist *default-space*)  
  (views nil)  
  (align nil))
```

Distribute an ordered set of views so that they fill a space relative to the centered-on-view. For example, if you distribute a set of buttons to the left of a table, the top of the first button would align with the top of the table and the bottom of the last button would align with the bottom of the table and the other buttons would be evenly distributed between them. The align argument specifies how the buttons would be aligned. The pos argument specifies the relative direction from the centered-on-view and can be any of: `{:left :before :leading} {:right :after :trailing} {:above :top} {:below :bottom}` where synonyms are shown together in the braces. The min-dist argument specifies a minimum distance between the centered-on-object and the views.

```
distribute-views (&key  
  (install-view nil install-view-provided)  
  (priority nil)  
  (orientation :h)  
  (min-space 2.0)  
  (views nil))
```

Distribute a set of views. The space over which the views are distributed is determined from the extreme positions of the first and last objects. Those positions can, of course, vary at runtime as their parent view expands or contracts. The orientation argument must be `:h` or `:v`.

```
make-constraint (&key  
  (install-view nil install-view-provided)  
  (priority nil)
```

```

item1
(att1 :width)
(relation :=)
(item2 nil)
(att2 nil) ;; defaults to att1 if item2 is not null
(mult 1)
(const 0))

```

Basic function for creating a single constraint and adding it to the parent-view. The `item1` argument must be a `NSView` instance. The `att1` argument must be one of the allowed relations as described above. The `relation` argument must be `:=`, `>=` or `<=`. The `item2` argument may be a `NSView` instance or `nil`. If `nil`, then the `att2` argument must be `:none`. The `mult` and `const` arguments must be numbers. Typically you would call `constrain` and let it translate into a `make-constraint` call.

```

make-constraints-for (subview-assoc-list constraint-string
&key
(install-view nil)
(options 0)
(metrics nil))

```

This is the programmatic interface to Apple's graphical layout language. This method allows you to specify layout relationships using text strings. You must supply an association-list that maps string names for views with views. Those names are used in the `constraint-string` argument to refer to views. See Apple's documentation for more information about what the `constraint-string` argument can contain. In terms of functionality provided, it is quite similar to the `order-views` method that I provide. I believe you will find that `order-views` is much easier for lisp developers to use. Note that it is perfectly ok to specify some constraints using this method and others using other methods.

```

order-views (&key
(install-view nil install-view-provided)
(priority nil)
(orientation :h)
(spacing nil)
(from nil)
(to nil)
(rel :=)
(align nil)
(views nil))

```

Constrain a set of views to follow each other in some direction. The `views` argument is a bit of a misnomer because this list can contain numbers, in addition to views. If included, a number in the list of views specifies the distance that should be kept between successive objects. If a number is not specified between views then a default spacing value is used. It is never legal to have two numbers in a succession in the list. If you do that, then constraints will be added to reflect processing of the views up to that

point, but no other constraints are created and a message is logged to the system console explaining that the list is invalid. A number at the beginning (end) of the views list indicates a constraint should be added between the relevant edge of the parent (depends on the orientation) and the first (last) view in the views list. This is something of a lisp analog to the string layout language that is used as input to the make-constraints-for method.

## ***Methods for prioritizing view expansion and compression***

### **prioritize-compression-resistance ((parent-view ns:ns-view) orientation &rest subviews)**

This allows you to specify the relative priority for deciding what view dimensions should be compressed. Apple's layout functionality takes this into account when computing sizes of objects whose dimensions are flexible. The orientation argument must be :h or :v. Priorities are set relative to the existing priority of the first view argument. The first view specified will be compressed after later views. That is, it has the highest resistance to being compressed. If multiple views should have the same priority, simply enclose them in a list. Be careful here because it is possible to specify a view which the layout system may compress to the point of invisibility. If something doesn't appear at all, it's likely that you didn't specify a frame rect for it and didn't provide any other constraints on it that would guarantee a non-zero size in both dimensions. If that happens, try adding a constraint on the object such as:

```
(constrain-to-natural-size <parent-view> <view> :rel :>=)
```

which will guarantee a non-zero size for <view>.

### **prioritize-expansion-resistance ((parent-view ns:ns-view) orientation &rest subviews)**

This allows you to specify the relative priority for deciding what view dimensions should be expanded. Apple's layout functionality takes this into account when computing sizes of objects whose dimensions are flexible. The orientation argument must be :h or :v. Priorities are set relative to the existing priority of the first view argument. The first view specified will be expanded after later views. That is, it will have a higher resistance to expansion than later items. If multiple views should have the same priority, simply enclose them in a list.

## ***Methods for debugging window layout behavior***

### **analyze-constraints ((win ns:ns-window) &optional indent)**

This method walks through the views in the window, displaying the view hierarchy and frame rectangles for each view. It also looks for a few types of anomalies. It looks for views that have a 0 width or height or which have an ambiguous layout (as determined

by Apple layout software). It then lists all the constraints that are used to layout those views. So if your window looks incorrect, running this method in the listener can help to pinpoint problems.

#### **constraints ((win ns:ns-window))**

##### **constraints ((v ns:ns-view))**

Return a list of constraints contained by the argument specified. This list can be useful to see when debugging window behavior. You may decide to remove selected constraints to see how the window reacts. If a window is specified the constraints returned are those owned by its content view.

##### **constraints-affecting-layout ((v ns:ns-view) &key (orientation :h))**

Return a list of constraints that are used to determine the current layout of the specified view. This is useful when debugging window behavior. You may be surprised to find that some constraint you thought should influence a views layout does not do so. This list may also be used as an argument to the visualize-constraints method, permitting you to see them graphically.

##### **log-constraints (&optional (on-off t))**

Turn on/off logging of constraint information to the system console log as constraints are created. Information includes which method is being invoked and detailed information about each constraint created by it. View these using the Apple "Console" application. Logging constraints in this way may slow down the creation of more complicated windows. That happens because there is a maximum number of log messages that an application can create in any given second. Any more than that are discarded. So the constraint software tries to detect when that might happen and sleep for 1 second to permit additional log messages to be issued. In that way, no information about constraint creation is lost.

##### **remove-constraints ((parent-view ns:ns-view) constraint-list)**

Remove constraints from a view. All constraint functions return a list of the constraints that they created. It may be useful to keep track of them for debugging purposes, so that you can remove them if things don't work out the way you expect. Note that you can always find window views and constraints using listener commands. Let's assume you start with a window-controller object contained in the variable wc. Then you can do something like the following:

```
(setf win (#/window wc))
```

```
(setf cv (#/contentView win))
```

```
(setf subviews (coerce-obj (#/subviews cv) 'list))
```

```
(setf constraints (coerce-obj (#/constraints cv) 'list))
```

and go on from there looking at subviews and their constraints as deeply as needed.

##### **visualize-constraints ((win ns:ns-window) &optional (constraints nil))**



**visualize-constraints ((v ns:ns-view) &optional (constraints nil))**

Exercise Apple's constraint visualization. This can be a very valuable tool to use when debugging window behavior. It adds colored lines to your view to represent each constraint you have requested to see. If you do not specify a list of constraints, then all constraints for the specified window or view are visualized. In a complex window that can be very confusing. If you click on any line (constraint) that looks odd to you, a message will be logged to the system console telling you what constraint that is. If the layout system detects any ambiguity in the way that constraints can be interpreted, it will display a button that lets you exercise that ambiguity to see alternative ways that the view might have been displayed. That can permit you to add additional constraints that remove that ambiguity and assure that the layout always matches your intent.

***Utility methods primarily for those who want to create their own constraint idioms*****attribute-convert (att-key)**

Convert a keyword { :top :bottom :left :right :leading :trailing :center-x :center-y :baseline :none } to the appropriate form for an Objective-C call.

**make-invisible-spacer ((parent-view ns:ns-view))**

Create an invisible view with nothing in it. This is useful at times because you cannot directly constrain the space between different sets of objects in any other way. The distribute-relative-view makes use of invisible spacers to represent white space that can be constrained to be equal to each other (either width or height, whichever is appropriate).

**natural-size ((v ns:ns-view))**

This returns whatever the view considers to be its minimal natural size. This is computed in different ways, depending on the type of view and also the way it was initialized by the user. The first priority is to honor any size specified when the object was initialized by specifying its frame. Other methods are used after that to make a determination.

**orientation-convert (o-key)**

Convert a keyword { :h :v } to the appropriate form for an Objective-C call.

**priority-convert (p-key)**

Convert a keyword { :required :high :can-resize :window-stay-put :cannot-resize :low :fitting-size-compression } to the appropriate form for an Objective-C call.

**relation-convert (rel-key)**

Convert a keyword {:= :>= :<=} to the appropriate form for an Objective-C call.

**view-p (thing)**

Returns t if thing is an ns:ns-view type.

**zero-size-p (ns-size)**

Returns t if the size has a zero value in either dimension.

## **Appendix C: List of Lisp View Classes in the CocoaInterface distribution**

Here is a list of new Lisp view classes that are included in the distribution:

***Container Views***

organized-box-view

resizable-box-view

This view will layout a set of objects as an array that is dynamically modified as the enclosing view's dimensions change.

***Text views***

rotated-text-view

Similar to an ns-text-view but allows text to be slanted at an arbitrary angle

form-view

A box containing right-aligned ns-text-fields of the same width and left-aligned labels for each text field. Size of text fields change as width of the box changes.

label-view

A simple fixed label

labeled-text-field

A box containing a label and a text-field

vscrolled-text-view

A text-view inside a vertical scroll-view which also has a stream acceptable to lisp format calls

***Printing views***

`lisp-app-doc-print-view`

View that provides support for printing a document

### ***Buttons***

`lisp-button`

A button that can be configured to call a Lisp function when pushed.

`button-controller`

Controls a set of buttons that can be configured to call a specified method for a specified instance when any of the button are depressed.

`radio-button-controller`

A button controller specific to radio buttons.

`button-box-view`

Buttons are arranged in a fixed size row or column (default). Set the spacing keyword argument to specify the space between the buttons.

`radio-button-box-view`

A button-box-view specific to radio buttons

`radio-button-array-view`

A resizable-box-view specific to radio buttons