Reference Manual and Tutorial for KVO-Slot Functionality

Version 2.0 April 2013

Paul Krueger, Ph.D.

Declaration of Slot KVO Compliance

Making slots within Lisp classes Key-Value Observing (KVO) compliant enables them to be used as binding targets for user interface fields. The functionality included in kvo-slot.lisp permits several forms of slot binding for Lisp developers. Any use of the :kvo declaration within a slot will make it possible for a key-value observer object to bind to it and be notified when a change is made to the slot's value. The way in which that value may be accessed depends both on the class for which the slot is defined (whether it is an Objective-C class or standard Lisp class) and the type of slot (whether it is a :foreign-type slot or not).

The Apple binding protocol is implemented between the slots of two Objective-C objects. The methods described here extend that protocol to make such bindings possible to Lisp slots within normal Lisp objects. We will use a little Common Lisp Meta-Object Protocol (MOP) magic to make slots KVO compliant when accessed via lisp functions just by adding the :kvo keyword to the slot specification within a normal defclass form. The value of the :kvo slot-option keyword can be either the name of a lisp accessor function or a case-sensitive string that provides a binding pathname that can be used to bind to that slot.

The first form of the :kvo specification might look something like the following:

```
(defclass my-class ()
  ((slot-a :accessor slot-a :initform 1 :kvo slot-a)))
```

Here is another example demonstrating that it is possible to make Objective-C slots within user-defined classes KVO compliant by using the same mechanism:

```
(defclass my-class (ns:ns-object)
  ((slot-a :accessor slot-a :foreign-type :id :kvo slot-a)
   (slot-b :accessor slot-b :kvo slot-b))
  (:metaclass ns:+ns-object))
```

When binding to a slot with such a specification the pathname used should be translated from the Lisp name specified in the normal way that CCL converts between Lisp and Objective-C names (see the CCL documentation for a description of this). So in this case the symbol slot-a would be translated to the case-sensitive path "slotA". When accessing the slot via a binding path, the accessor provided as the :kvo value will

be used.

The second form of :kvo specification might look something like:

```
(defclass my-class (ns:ns-object)
  ((slot-a :accessor slot-a :foreign-type :id :kvo "slA")
   (slot-b :accessor slot-b :kvo "slB"))
  (:metaclass ns:+ns-object))
```

In this form the Objective-C pathname is explicitly provided as a case-sensitive string. The use of a string rather than a symbol in the :kvo slot specification, means that no translation will be done.

Using either of these forms makes it possible to bind to slot-a without adding any additional lisp code. The bottom line for a developer is that they can specify KVO compliance for any slot (foreign or otherwise) in any class defined within lisp (i.e. standard or subclass of ns-object). Any access to that slot using Lisp methods either directly or indirectly will make the correct function calls to support KVO compliance. Note that if you define your own Objective-C methods which access the slot using any Lisp method (i.e. one of the defined slot accessors or via the slot-value method), then they will also be KVO-compliant.

To extend the binding protocol to slots in standard Lisp classes, the implementation transparently creates a proxy Objective-C object for each object that has a slot to which we want to bind. This proxy knows what Lisp object is being referenced and retrieves or sets values as needed. If the first form of :kvo slot specification was used, then the binding pathname can be directly transformed into a slot accessor. If the second form was used, then the proxy object will search for a slot within the referenced object that is bound to the specified path string and return its value. In this case, none of the accessors specified for the slot in its slot definition are called to set or return its value. This can be useful if you want to differentiate between a slot change that was made as a result of a change in an interface object slot and a change that was made as the result of some Lisp computation. We will see later how to define methods that will be invoked only when a change is made as a result of some binding.

In some cases it may be desirable to force binding accesses to be made using a specified Lisp accessor function. In such cases the :kvo option should always be of the first forr (i.e. a symbol naming the accessor that should be used and not a string). As mentioned earlier, this just requires that appropriate name translation be done when the binding is specified in Interface Builder.

Lisp developers who wish to bind to lisp slots should use the Lisp *bind* method (defined in binding-utils.lisp) rather than any Objective-C method. Not only does this extend the binding protocol to Lisp slots, it also makes it possible to include Lisp functions in the specified binding path that can transform objects along the way. Various examples of using the bind method are provided in the *UserInterfaceTutorial* document.

Notification of Binding Changes

There are three sorts of notification methods that can be defined for objects that have slots with :kvo specification.

Whenever a :kvo slot is about to be modified as the result of a binding, the method *bound-slot-will-be-modified* will be called with two arguments: the object that contains the slot and the name of the slot that will be modified. The default method does nothing. Developers may define more specific methods for their object that respond to one or more slot names in the object. One example of how this might be used is to capture current slot information before the change occurs in order to support undo functionality. An example of this is show for the loan document project in the *UserInterfaceTutorial* document.

Whenever a :kvo slot is modified as the result of a binding, the method *bound-slot-modified* will be called with two arguments: the object that contains the slot and the name of the slot modified. The default method does nothing. Developers may define more specific methods for their object that respond to specific slot names or make one generic method that handles all :kvo slots for the class. Instance-specific methods could also be defined.

The third method is to use the *defnotification* macro provided in kvo-slot.lisp. This is defined as follows:

```
(defmacro defnotification ((instance-arg slot-arg new-value-arg) &rest function-forms)
```

The method defined using defnotification will be called for every type of change made to the slot, both those made as a result of a binding and those made using any form of Lisp accessor method, including (setf (slot-value ...)) forms.

For those with a deeper interest in how all this is implemented, the remainder of this document provides that information.

Introduction to KVC and KVO

For those who want more low-level information I'll provide a short introduction to KVC and KVO compliance here, but I'd strongly suggest that you consult other resources for more information. A good source is Apple's' Cocoa Bindings Reference and you can follow other links from that document. The interested reader may also find Apple's reference manual for Key Value Observing useful. You can download it from:
http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/KeyValueObserving/KeyValueObserving.pdf

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

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

If you think of this as access to object slot values by passing in the name of the slot as a parameter (i.e. as if by calling #'slot-value in Lisp) you won't be too far off the mark

although there are differences in syntax and semantics as we'll see.

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

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

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

To be KVO-compliant, slots must be key-value coding compliant and proper observer notification calls must be made (either manually or automatically). The functionality implemented via the :kvo slot specification ensures that both of those conditions are met. Access to slots for non-Objective-C objects is done via the use of a lisp-ptr-wrapper object (defined in ns-object-utils.lisp). The KVO code described below works in conjunction with the lisp-ptr-wrapper code to provide complete KVO compliance. The easiest way for a developer to make use of this functionality is to use the *bind* method provided in bind-utils.lisp. This assures that all necessary lisp-ptr-wrapper proxies are created whenever needed.

MOP Basics

The Lisp Meta-Object Protocol (MOP) is designed to allow developers to define new classes of classes or class slots that can then be used to define new user classes with different behavior than standard classes. Developing with the MOP can be a daunting task. The necessary background can be found in the MOP specification itself: Concepts (http://dreamsongs.com/Files/concepts.pdf) and Functions (http://dreamsongs.com/Files/Functions.pdf). Additional very useful discussion can be found at http://

www.alu.org/mop/index.html. And of course the book "The Art of the Metaobject Protocol" by Kiczales, des Rivieres, and Bobrow is still the best introduction available.

I won't undertake to create a new MOP tutorial here; there are other sources for such information (although not too many). Instead I will outline just enough to aid in understanding the approach used to provide KVO functionality.

There are two sorts of metaclasses that provide slot information. A *Direct Slot* metaclass contains information about how a slot was defined within a defclass form. But class slots in an instantiated instance may also inherit options defined for that slot within superclasses. Instances of *Effective Slot* metaclasses merge information defined for the same slot in different places within the class hierarchy and therefore contain complete information about the actual behavior of every slot (either directly defined within the class or inherited or merged from several places) that is accessible within any instance of a particular class.

It is possible to define new Direct Slot and Effective Slot metaclasses that result in different runtime behavior for the slots that they represent and that is exactly what we will do to implement KVO functionality. We will define a new Direct Slot class (a couple actually, but we will get to that later) that accepts the :kvo keyword option. We will define a new Effective Slot class that permits us to specialize the method defined for all slot accesses. That specialization will make the additional calls needed to support KVO compliance for these slots.

The MOP specifies methods that will be called to determine what direct-slot metaclass should be used and what effective-slot metaclass should be used for each slot. We will add around methods for those that will return our new classes when appropriate. More on that as those functions are discussed below.

Finally the MOP specifies a generic function that will be called to update a slot within a particular class of effective-slot. We will create special methods for that function that will be invoked when the effective-slot class is one of ours and they will make the necessary calls to ensure KVO-compliance.

Sounds pretty simple right? There are a few details to come, but thanks to the design of the MOP it really isn't very difficult to make this all work.

Implementation

The following code from kvo-slot.lisp implements the needed functionality. A number of utility functions and a new type are defined first.

To decide whether an object has KVO slots we provide a special type that is used by the lisp-ptr-wrapper class (defined in ns-object-utils.lisp). It only tries to access the slot using special KVO methods when (typep <object> 'kvo-object) is true. The "kvo-object" type is defined in terms of the kvo-slots method which is, in turn, defined in terms of the class-kvo-slots method. That method caches a list of all KVO slots for each class in a

hash table to avoid repeated type-checking of all the slots in the class.

```
(let ((kvo-hash (make-hash-table)))

  (defmethod invalidate-kvo-hash ((self standard-class))
    (setf (gethash self kvo-hash) nil))

  (defmethod class-kvo-slots (class)
    (or (gethash class kvo-hash)
        (setf (gethash class kvo-hash)
              (remove-if-not #'(lambda (slot)
                                 (or (typep slot
                                            'kvo-foreign-
effective-slot-definition)
                                     (typep slot
                                            'kvo-effective-slot-
definition)))
                             (class-slots class)))))

  (defmethod kvo-slots ((self standard-object))
    (class-kvo-slots (class-of self)))

)
```

The kvo-hash simply keeps track of which slots in a class have a KVO declaration. We discover this by looking at the type of each of the effective slots in the class to see if it matches one of the two new types that we will discuss in more detail below. The invalidate-kvo-hash method is provided to invalidate the hash table entry. This is called when a class is redefined. The class-kvo-slots method returns the value from the hash-table if it exists or sets and returns the appropriate value otherwise.

The kvo-slots method is used for convenience since typically we have an object for which we want a list of slots and not that object's class.

Finally we define the kvo-object type as any object that has slots including a :kvo declaration:

```
(deftype kvo-object ()
  '(satisfies kvo-slots))
```

The kvo-slot-for method searches for an effective slot that has as one of its :kvo keys a value that matches the given key-string parameter.

```
(defmethod kvo-slot-for ((self standard-object) key-string)
  (find key-string
        (kvo-slots self)
        :key #'kvo-slot-definition-kvo
```

```
      :test #'(lambda (key-string slot-keys)
                (member key-string slot-keys :test #'string=))))
```

Next we provide methods for retrieving and setting the value of a KVO slot when given a string key.
The method value-for-kvo-key will retrieve a value for a specified key-string within an object. This is called by a lisp-ptr-wrapper to get an actual slot value. That value will be supplied as the return value when #/valueForKey: is called on the lisp-ptr-wrapper by some interface object. In effect, the lisp-ptr-wrapper acts as a proxy for the lisp instance.

```
(defmethod value-for-kvo-key ((self standard-object) key-string)
  ;; find a slot where the key-string is a member of the list that is the
  ;; value of its kvo slot and return the value of that slot
  (let ((slot (kvo-slot-for self key-string)))
    (when slot
      (slot-value self (slot-definition-name slot)))))
```

Similarly, the (setf value-for-kvo-key) method will set the slot for which the specified key-string is one of the declared :kvo strings.

```
(defmethod (setf value-for-kvo-key) (new-value (self standard-object) key-string)
  (let ((slot (kvo-slot-for self key-string)))
    (if slot
      (values (setf (slot-value self (slot-definition-name slot)) new-value) t)
      (values nil nil))))
```

That explains how kvo slots are used, but we need to dig a bit deeper to see how the slots themselves are defined and used.

So let's look at how the MOP can be used to make all this happen. We start with two new direct-slot classes and two new effective-slot classes that will be used when a slot has a :kvo declaration. One of each is for slots that also include the :foreign-type declaration and the other is for slots that are otherwise standard. We need to discriminate between the two so that we can do somewhat different things when those slots are accessed.

```
(defclass kvo-direct-slot-definition (standard-direct-slot-
definition)
  ((kvo :initarg :kvo :initform nil :accessor kvo-slot-
definition-kvo)))

(defclass kvo-foreign-direct-slot-definition (foreign-direct-
slot-definition)
  ((kvo :initarg :kvo :initform nil :accessor kvo-slot-
definition-kvo)))
```

These two classes are subclasses of standard-direct-slot-definition and foreign-direct-

slot-definition respectively. As you might expect, a foreign-direct-slot-definition instance is specified for use by the Objective-C bridge code whenever a :foreign-type declaration is made for a slot. In our case, we'll use a kvo-direct-slot-definition whenever a slot makes a :kvo declaration without a :foreign-type declaration and a kvo-foreign-direct-slot-definition if it has both declarations. We'll see how that choice is made shortly.You will note that the direct slot classes have a kvo slot that allows us to accept the :kvo keyword in a slot definition. They will contain the name of the binding that was specified by the :kvo <name> declaration within the defclass form for that slot.

The following two direct-slot-definition-class methods provide an answer to the question "What direct slot metaclass should be used for this slot?". This method is specified as part of the MOP. The arguments to it are the class object and the arguments specified by the user to initialize the slot.

```
(defmethod direct-slot-definition-class :around ((class
objc:objc-class-object)
                                                 &rest initargs)
  (let ((base-class (call-next-method)))
    (if (getf initargs :kvo)
      (if (eq base-class (find-class 'foreign-direct-slot-
definition))
        (find-class 'kvo-foreign-direct-slot-definition)
        (find-class 'kvo-direct-slot-definition))
      base-class)))
```

For Objective-C classes we first let the Objective-C bridge code specify what class it would have picked for the direct slot by invoking (call-next-method). We then look to see whether the :kvo option was also specified for that slot. If it was not, then we use the class returned by (call-next-method). If it was, then we use the class picked by the Objective-C bridge code to decide which of our classes to use. We pick the one that is a subclass of the class picked by the Objective-C bridge code.

```
(defmethod direct-slot-definition-class :around ((class
standard-class)
                                                 &rest initargs)
  (if (getf initargs :kvo)
    (find-class 'kvo-direct-slot-definition)
    (call-next-method)))
```

For standard-class definitions we simply look to see whether the :kvo option was specified and use the kvo-direct-slot-definition if it was. Otherwise we just use whatever is returned by (call-next-method).

The MOP dictates that classes be finalized by creating a set of effective slot objects, each of which represents one of the slots that will be created for instantiated instances of the class. We won't go into all of the detail of how and when various methods are invoked. We refer the interested reader to the Kiczales, des Rivieres, Bobrow book

mentioned earlier. Only the relevant classes and methods that we need to override are described here. In essence what happens is that all of the direct slot instances from all classes in the hierarchy of the new class being defined are combined to create a set of effective slots for the class. When this happens for slots with the :kvo option specified, we will want to use a new effective-slot subclass to assure proper KVO functionality.

First let's define two new effective-slot classes that directly parallel the direct-slot classes that we previously defined.

```
(defclass kvo-effective-slot-definition (standard-effective-
slot-definition)
  ((kvo :accessor kvo-slot-definition-kvo)))
```

This class is used for slots that specify the :kvo option in otherwise normal (i.e. non-Objective-C) slots. Before we show the initialization method for this class we introduce a useful utility method that returns a list of all the direct-slots defined anywhere within the class inheritance hierarchy with a specified slot name:

```
(defmethod corresponding-direct-slots ((class slots-class) slot-
name)
  ;; Find an ordered list of any direct slots with slot-name in
class
  ;; hierarchy of class
  (do* ((top-class (find-class t))
        (dslots nil)
        (classes (list class) new-classes)
        (new-classes nil))
       ((null classes) (nreverse dslots))
    (setf new-classes nil)
    (dolist (cl classes)
      (unless (eq cl top-class)
        (setf new-classes (append (class-direct-superclasses cl)
new-classes))
        (let ((ds (find slot-name
                        (%class-direct-slots cl)
                        :key #'slot-definition-name)))
          (when ds
            (push ds dslots)))))))
```

Recall that an effective-slot-definition is created from possibly many applicable direct-slot definitions if that slot name was declared in more than one of the classes in the class hierarchy. Our initialize-instance :after method will merge multiple :kvo options if they exist.

```
(defmethod initialize-instance :after ((self kvo-effective-slot-
definition)
                                       &key &allow-other-keys)
```

```
  (let* ((class (standard-effective-slot-definition.class self))
         (slot-name (standard-effective-slot-definition.name
self))
         (dslots (corresponding-direct-slots class slot-name))
         (keys (delete-duplicates (mapcar #'canonical-kvo-key
dslots)
                                  :test #'string=)))
    (invalidate-kvo-hash class)
    (setf (kvo-slot-definition-kvo self) keys)))
```

We first invalidate the kvo-hash for this class. This will cause the list of KVO slots for the class to be updated the next time that any object is bound to an instance of the class. Next we initialize these effective-slot instances by setting the kvo-slot-definition-kvo slot to a list of strings. Each string corresponds to one of the :kvo options specified anywhere within the class hierarchy. This means that it is possible to specify a subclass of a class that has a slot with a :kvo specification and in that subclass specify the same slot name with a different :kvo specification. The slot will be bindable by BOTH of the specified :kvo names. For example, a developer might do this to specify a symbol that names a new slot accessor so that some side-effect can be done whenever a slot is changed or accessed by a user-interface object.

```
(defclass kvo-foreign-effective-slot-definition (foreign-
effective-slot-definition)
  ((kvo :accessor kvo-slot-definition-kvo)))
```

This class represents effective-slots that have both a :foreign-type and a :kvo specification.

```
(defmethod initialize-instance :after ((self kvo-foreign-
effective-slot-definition)
                                       &key &allow-other-keys)
  (let* ((class (standard-effective-slot-definition.class self))
         (slot-name (standard-effective-slot-definition.name
self))
         (dslots (corresponding-direct-slots class slot-name)))
    (invalidate-kvo-hash class)
    (setf (kvo-slot-definition-kvo self)
          (delete-duplicates (mapcar #'canonical-kvo-key
dslots) :test #'string=))))
```

The additional initialization of these effective-slots to handle the :kvo option is identical to that for the kvo-effective-slot-definition class.

Next let's talk about how we decide which effective-slot class to use. In a manner that is analogous to how the direct slot class was selected, the MOP dictates that the generic function effective-slot-definition-class will be called. When deciding which effective-slot class to use we want to look at all of the corresponding direct-slots to see whether any

of them is an instance of one of the KVO direct slot classes. If so, then we will use the corresponding effective-slot class. We want our :around method to apply to both Objective-C and standard classes, so we specialize the method on the most specific common superclass, which in CCL happens to be the class "slots-class".

```
(defmethod effective-slot-definition-class :around ((class
slots-class)
                                &rest initargs)
  ;; basically we want to select the effective-slot-definition-
class based on the
  ;; class of the corresponding direct-slot. So we find a direct
slot of the same
  ;; name if it exists and go from there. If there isn't one
just call-next-method.
  (let* ((slot-name (getf initargs :name))
         (dslots (corresponding-direct-slots class slot-name))
         (dslot-classes (mapcar #'class-of dslots))
         (base-class (call-next-method)))
    (if dslots
      (cond ((find (find-class 'kvo-foreign-direct-slot-
definition) dslot-classes)
             (find-class 'kvo-foreign-effective-slot-
definition))
            ((find (find-class 'kvo-direct-slot-definition)
dslot-classes)
             (find-class 'kvo-effective-slot-definition))
            (t
             base-class))
      base-class)))
```

Once the effective-slot class has been determined, standard MOP methods will create an instance of that class. The value of the KVO slot in the effective slot instance will be a list of names to which Objective-C objects can be bound. Although each :kvo declaration is only allowed a single name, there might be several such declarations in the class hierarchy and each of them defines a name that can be used. Those names are specified in Lisp syntax and converted to a corresponding Objective-C name in much the same way that all slot accessor names are converted.

This completes the discussion about the basic data structures needed and how they are created. Let's go back and look at how we use them to ensure KVO compliance.

The first criteria for KVO compliance is Key-Value Coding (KVC) compliance.
For :foreign-type slots that are bound to directly it is the developer's responsibility to provide the necessary accessor functions. There are several examples of this provided in the InterfaceBuilderWithCCLTutorial3.0 document. For example, if the slot is named "slot-one", then the methods #/slotOne and #/setSlotOne: must be defined.

For slots that are bound to using the lisp-ptr-wrapper functionality (as implemented by the lisp-controller class) we will provide automatic KVC compliance without requiring the developer to provide additional accessor functions. We saw above how the lisp-ptr-wrapper will access KVO slots when either of the Objective-C functions #/value:forKey: or #/setValue:forKey: are called on it. It will decide what method should be used to retrieve or set the specified key (let's call it the input-key) in the specified target object. If the :kvo option in the slot was a symbol, then the input-key should correspond to a lisp-accessor function and the lisp-ptr-wrapper will find that function and invoke it. If the :kvo option in the slot was a string, then the input-key should be a matching string. The lisp-ptr-wrapper will find the slot in the designated object for which a string that matches the input-string was specified and invoke either value-for-kvo-key or (setf value-for-kvo-key) as discussed above. This satisfies the first criteria for KVO compliance.

The second criteria for KVO compliance is that appropriate notification function calls be made whenever a slot is updated. For slots in Objective-C subclasses that are bound to directly, this means making calls to #/willChangeValueForKey: and #/didChangeValueForKey:. For slots in classes that do not derive from an Objective-C class this means calling the corresponding methods iu:will-change-value-for-key and iu:did-change-value-for-key which are defined as part of the lisp-ptr-wrapper functionality described earlier. An interface object can bind itself to a foreign-type slot either directly (if appropriate Objective-C access methods have been defined) or indirectly via a lisp-ptr-wrapper object or perhaps both can be used by different interface objects. The key used will be different for each method, so for such slots we must assure that both types of update call are made.

The MOP dictates that the generic function (setf slot-value-using-class) will be called when setting any slot-value. So to add the functionality just described we can simply create an :around method for that generic function:

```
(defmethod (setf slot-value-using-class)
           :around (value
                    (class objc:objc-class-object)
                    instance
                    (slotd kvo-foreign-effective-slot-
definition))
  (declare (ignore value))
  ;; This function assures that all possible accessors from
inherited versions
  ;; of the same slot are kept KVO compliant
  ;; Calls to #/willChange... and #/didChange... must be nested
so the order
  ;; of calls to these two is reversed
  (flet ((objc-key (key)
           (iu:lisp-to-temp-nsstring key))
         (will-change (key objc-key)
           (iu:will-change-value-for-key instance key)
           (#/willChangeValueForKey: instance objc-key))
```

```
          (did-change (key objc-key)
            (#/didChangeValueForKey: instance objc-key)
            (iu:did-change-value-for-key instance key)))
      (let ((objc-keys (mapcar #'objc-key (kvo-slot-definition-kvo
slotd)))
            (lisp-keys (kvo-slot-definition-kvo slotd)))
        (mapcar #'will-change lisp-keys objc-keys)
        (prog1
            (call-next-method)
          (mapcar #'did-change (reverse lisp-keys) (reverse objc-
keys))))))
```

One of the things to observe in this function is that it assures that all "will-change" and "did-change" calls are strictly nested. This is a requirement of KVO.

For non :foreign-type slots that have a :kvo option the only binding possible is via a lisp-ptr-wrapper so we can omit the calls that would be required for direct bindings. Otherwise the definition is very similar to that for :foreign-type :kvo slots:

```
(defmethod (setf slot-value-using-class)
           :around (value
                    class
                    instance
                    (slotd kvo-effective-slot-definition))
  (declare (ignore value class))
  ;; Calls to #/willChange... and #/didChange... must be nested
so the order
  ;; of calls to these two is reversed
  (flet ((will-change (key)
            (iu:will-change-value-for-key instance key))
          (did-change (key)
            (iu:did-change-value-for-key instance key)))
    (mapcar #'will-change (kvo-slot-definition-kvo slotd))
    (prog1
        (call-next-method)
      (mapcar #'did-change (reverse (kvo-slot-definition-kvo
slotd))))))
```

This provides all the functionality needed for KVO compliance.

Examples

All of the example code can be found in ...ccl/contrib/cocoa-ide/krueger/InterfaceProjects and each is described in the *UserInterfaceTutorial*.