Categories and Posers

by Theresa Ray of Tensor Information Systems, Inc.

Sponsored by Apple Computer, Inc.
Apple Developer Connection

Categories and Posers

by Theresa Ray of Tensor Information Systems, Inc.

Categories and posers are powerful mechanisms for extending and/or replacing the behavior of classes for which you cannot update the source. These techniques are essential tools that every WebObjects programmer should understand.

Why do you need categories and posers?

Object-oriented programming techniques have proven to be enormously beneficial for both programmers and those financing the programs. By being able to create and easily distribute common logic into reusable classes, applications are more robust, and are more quickly developed and deployed. Inherent to object-oriented programming is a concept known as encapsulation. Encapsulation is best described as "providing a well-defined interface to the users of a class, while hiding the implementation and behavior of that class from the user". In other words, a header is provided which defines the method syntax and the order and class of the arguments for that method (as well as useful documentation regarding the purpose of the method). But exactly how the developer implements that method is NOT provided, nor is it supposed to be important to the user.

In the real world of programming, however, you may find that you need some extensions or enhancements made to the provided classes. But without source code, what can you do to alter the behavior of the class? Subclasses are the first, and preferred, option for extending or modifying the behavior of a class. For a subclass to extend or modify the behavior of a class, you must message the subclass that implements the new behavior in place of the parent class. For example, if you needed to extend the behavior of an NSArray, you could create a subclass called TISNSArray that implements the new features. All applications requiring the extended behavior would have to message the TISNSArray subclass.

There are times where this requirement is not feasible. For example, when an instance of an EOEditingContext fetches, it returns an NSArray with the results. If you wanted, by default, to rely on behavior implemented in your subclass (TISNSArray), there is no EOEditingContext method you can call which would put the results directly into a TISNSArray instead of an NSArray. How can you overcome this obstacle?

How categories and posers work

Categories and posers are two options that allow you to extend or modify the behavior of any class in a more direct manner than subclassing, but may only be implemented in Objective-C. Categories and posers are special kinds of subclasses that merge themselves more seamlessly with the parent class. The parent class will directly exhibit the extended or overridden behavior that is actually defined in the category or poser subclass.

For example, you could create a framework containing a category on NSString to add a method called *stringByRemovingWhitespace*. Applications that link in the framework containing this category would be able to successfully send the message *stringByRemovingWhitespace* to any instance of NSString. Similarly, if that same framework included a class that posed as EOEditingContext and altered the behavior of the *initWithParentObjectStore:* method, any *initWithParentObjectStore:* message would behave in the altered manner.

Both categories and posers should be well understood before an attempt is made to use them in a production environment - particularly posers. Because you are altering the behavior of a common class, categories and posers are powerful mechanisms which can be essential tools for the programmer, but are inherently dangerous to the unassuming.

Categories and posers are related in their capabilities. Both alter the behavior of the parent class, but categories and posers provide different levels of control in their implementation. Categories are typically used to extend the behavior of a class, and cannot message super. Posers can message super, and therefore are more powerful in their abilities (particularly in extending or overriding a parent class's default behavior for a given method). Categories are relatively safe to use as long as the technique and limitations are well understood. Posers can be much more dangerous to implement due to their inherent wide-ranging effects and abilities.

Category details

Categories are defined explicitly on their parent class, and "just happen" when loaded (that is, you do not need to make a special call in order to "activate" a category). There is no limit to the number of categories you may add to a parent class, but each category class and method must have a different name. If you attempt to override the same method or define methods with the same name in different categories on the same class, you will experience runtime issues related to the order in which the conflicting classes or methods were loaded.

Categories can add new methods and override methods defined in the parent class they act on. Categories also have access to ALL instance variables for the class (including private instance variables). Categories are indistinguishable at runtime from their parent class, and any changes to the parent class methods are inherited by all subclasses of the parent class.

Category methods can not message super (posers can), nor can they add any instance variables (the compiler will catch any category attempt to add an ivar). If you need to add an instance variable, you need to use a regular subclass instead of a category.

Because of their more restrictive nature (not being able to call super or add instance variables), categories are relatively safe to implement, and are routinely used to add functionality to a class. However, it is dangerous to attempt to add a category to a Root object, since class objects can perform instance methods defined in the root class and Root objects have no superclass.

A category example

To illustrate an example of a implementing a category, if you wanted to create an NSString category you would create a new class with a new name. This class is defined as a category on the parent class by using the syntax:

```
@interface NSString (FoundationExtensions)
```

In this example, the framework FoundationExtensions contains the new class (named NSStringCategories) which implements the category methods. Any application which links in the FoundationExtensions framework will use the category methods for NSString defined in NSStringCategories. Appendix I contains source code for an example of a category. These categories were developed to provide convenience extensions to the Yellow Box NSString class.

Poser details

Posers, too, must be a subclass of the class they're posing as. These posing subclasses stick themselves in the class hierarchy between the parent "posed-as" class and all of its subclasses (like so many spliced genes). The poser receives any sent messages first, and may or may not message super (the beauty and danger of posing). Any subclasses inherit the new behavior implemented by the poser. Posers cannot add instance variables. If you try to add an instance variable, the compiler will not catch the attempt. You will get a runtime error, and the message is not always clear that your poser's attempt to add an instance variable is the cause. Posers can add new methods, as well as override or extend methods defined in the classes they are posing as.

In order to make your new subclass actually pose as the original class, you must use send the *poseAsClass:* or the *class_poseAs:* method to the original class before any other messages are sent to the original class, and before any instances of the original class are created. This may be accomplished in two ways. The first way is for your subclass to implement the class method named *load* (the *load* message is sent to classes when they are added to the Objective-C runtime). In your subclass's implementation of *load*, you would send the *poseAsClass:* message to self. The second way is for all applications that want to implement the poser to send the *poseAsClass:* message to the posing subclass from the application's main. For example the code:

```
[ EOEditingContextPoser poseAsClass: [ EOEditingContext class ] ] ;
```

would cause the EOEditingContextPoser subclass to begin posing as EOEditingContext for all messages sent to the EOEditingContext class and all instances of the EOEditingContext class for the entire application.

The second way - having each application explicitly specify the poser - is the recommended method, as it is the most conservative and consequently the safest. If you use the first way - having the subclass pose in the *load* method - there are many potential pitfalls. First, you cannot know for sure exactly when the *load* message will be received. It is generally received before the *initialize* message, but the order in which *load* messages are sent to classes is unspecified (and specifically, a class's superclass is not guaranteed to have its *load* method called before the subclass receives the message.)

Second, the Foundation framework has classes and subsytems that depend on *load* being called before they are fully functional. You may run into inter-dependency issues if you try to specify your poser in the *load* method. Third, if you use the pose-in-load solution, you don't know for sure whether users of your framework are aware that a poser is invoked. The *load* method was never intended to be a general initialization method. If you must have your subclass pose "automatically", you should investigate sending the *poseAsClass:* message in the subclass's *initialize* method instead of using the *load* method. There are cases, however, where no other option is available. Pose-in-load should be an absolute, final resort if no other mechanism is possible.

A poser example

To create an EOEditingContext poser, create a new class with a new name. This class is defined as any usual subclass of EOEditingContext. If you want to override or extend the posed-as class's methods, just implement them in the new subclass. To override behavior completely, the new subclass would not message super. To extend behavior, call the super's implementation of the same method and then execute your new functionality. Make sure your application's main sends your subclass the *poseAsClass:* message. Appendix II contains source code for an example of a poser. This poser was developed as a solution to the problem of snapshot retention after release of the EOEditingContext instance. A thorough description of the problem is included in the Appendix.

Conclusion

Use of categories and posers can be useful to extend, enhance and improve the basic behavior of a class. The casual use of both categories and posers is not recommended, as both have wide-ranging effects. Creation of a subclass to add or override behavior of a parent class should always be your first choice. There are purists who consider the entire topic of categories and posers as "evil". However, both are necessary tools for a real-world programming environment, and category methods in particular often make sense; they allow you to put methods "where they ought to be". I strongly recommend that posers be used mainly as a debugging tool, and only as a last resort for production use, while category methods can be used more openly but with caution.

Phone: 817-335-7770
E-mail: theresa@tensor.com
URL: http://www.tensor.com

Resources…

http://gemma.apple.com/techinfo/techdocs/enterprise/WebObjects
WebObjects Developer's Guide
Enterprise Objects Framework Developer's Guide
http://www.omnigroup.com/MailArchive/WebObjects
http://www.omnigroup.com/MailArchive/eof
http://www2.stepwise.com/cgi-bin/WebObjects/Stepwise/Sites
ftp://dev.apple.com/devworld/Interactive_Media_Resources
http://www.apple.com/developer
http://developer.apple.com/media
http://til.info.apple.com/

About the Author…

Theresa Ray is a Senior Software Consultant for Tensor Information Systems in Fort Worth, TX (http://www.tensor.com). She has programmed in OPENSTEP/Yellow Box (both WebObjects and AppKit interfaces) on projects for a wide variety of clients including the U.S. Navy, the United States Postal Service, America Online, and Lockheed-Martin. Her experience spans all versions of WebObjects from 1.0 to 4.0, EOF 1.1 to 3.0, NEXTSTEP 3.1 to OPENSTEP 4.2, Rhapsody for Power Macintosh, and yellow-box for NT. In addition, she is an Apple-certified instructor for WebObjects courses.

Tensor Information Systems is an Apple partner providing systems integration and enterprise solutions to its customers. Tensor's employees are experienced in all Apple technologies including OPENSTEP/Yellow Box, NEXTSTEP, Rhapsody, EOF and WebObjects. Tensor also provides Apple-certified training in WebObjects, Oracle consulting and training, as well as systems integration consulting on HP-UX. And Oracle.

You may reach Theresa by e-mail: *theresa@tensor.com* or by phone at (817) 335-7770.

Tips…

- Both categories and posers can access private instance variables defined in the parent or original class.

- Use of public instance variables in categories is not the cleanest programming technique. If you can get what you need via some public method, do that instead.

- Choose category method names carefully. If a new version of WebObjects implements a new method with the same name as a method you defined, you will be overriding the new feature unwittingly. My method names tend to be quite lengthy and nearly self-documenting. While this can be annoying to use in an application, it adds some measure of assurance that the same name will not be chosen by someone else in the future.

- Make sure the implementation for your categories and posers are robust, well tested, and not likely to cause conflict with other classes or methods (either present or future).

- Remember that Java does not support categories or posers, so don't get married to this technique.

- Unless you are either VERY, VERY experienced in the topic of categories and posers or you are very reckless, do NOT pose as any Root object (especially NSObject) or create categories on any Root object.

- Several calls to poseAsClass: or class_poseAs: may cause a segmentation fault on the HP-UX platform. Refer to Apple Bug reference #2176625 for more details (http://til.info.apple.com/).

- If you use an Interface Builder palette which poses classes, you need to be careful if you save a nib file that includes instances of those classes. Opening these nib files in an application which doesn't do the same poses (or trying to open them in Interface Builder without that palette loaded) will cause an unarchiving error. The error will say that the %PosedClassName was not loaded. The Yellow Box runtime uses the preceding % for posed classes.

Appendix I

This appendix contains an example of a category used to provide convenience extensions to the Yellow Box NSString class.

Here is the interface file for our category.

```
#import <Foundation/NSString.h>

@interface NSString (FoundationExtensions)
/** The (FoundationExtensions) category on NSString provides
convenience extensions to the Yellow Box NSString class.*/

·  (NSString *) stringByRemovingWhitespace ;
/** Returns a copy of the receiver with all whitespace removed. */

@end

@interface NSMutableString (FoundationExtensions)
/** The (FoundationExtensions) category on NSMutableString provides
convenience extensions to the Yellow Box NSMutableString class.*/

·  (void) replaceSubstring: (NSString *) aSubstring withString:
      (NSString *) aString ;
/** Replaces all instances of aSubstring within self with aString.*/

@end
```

Here is the implementation file for our category.

```
#import <Foundation/NSCharacterSet.h>
#import "TISNSStringCategories.h"

@implementation NSString (FoundationExtensions)

-  (NSString *) stringByRemovingWhitespace
{
   NSMutableString *copy = [ self mutableCopyWithZone: [ self zone ] ]
;
   NSCharacterSet *wsCharSet = [ NSCharacterSet
      whitespaceAndNewlineCharacterSet ] ;
   NSRange r ;

   while ( ( r = [ copy rangeOfCharacterFromSet: wsCharSet ] ).length >
      0 )
   {
      [ copy replaceCharactersInRange: r withString: @"" ] ;
   }

   return [ copy autorelease ] ;
}
@end


@implementation NSMutableString (FoundationExtenstions)

-  (void) replaceSubstring: (NSString *) aSubstring withString:
      (NSString *) aString
{
   NSRange theRange = [ self rangeOfString: aSubstring ] ;

   while ( theRange.length != 0 )
   {
      [ self replaceCharactersInRange: theRange withString: aString ] ;
      theRange = [ self rangeOfString: aSubstring ] ;
   }
}
@end
```

Appendix II

This appendix contains an example of a subclass used to pose as an EOEditingContext. This example is extensive, but provides a solution to a real-world problem. Posers should not be invoked lightly, and so their implementation tends to be extensive. This appendix contains all the pieces required to solve the problem at hand, and includes a poser, a category, and an example of the code used to call the extensions built by the poser and the category. The problem being solved is relevant for WebObjects versions 3.0 through 4.0.1, and is as follows:

Most applications use an optimistic locking mechanism when fetching information from a database. In optimistic locking, when a row is fetched from a database for update, EOF makes a snapshot of that row. When saving updates to that row, the attributes in the snapshot marked by EOModeler as "used for locking" are compared with the values stored in the database at the time of the update. If any of the attributes have changed, EOF raises an exception (because some other application has altered the row in the database between the time the row was fetched for update and the time the update is being committed to the database). This is obviously an essential and beneficial feature of EOF.

The problem arises when the instance of the EOEditingContext that fetched the rows is released - the snapshot is NOT released. The EOF class responsible for maintaining snapshots does so across potentially many EOEditingContexts. Just because one of the instances is released doesn't mean that all instances are finished with the snapshot. The only way one can know when to garbage collect the snapshot is to have every instance of an EOEditingContext register itself in a non-retaining global array in the userInfo of the EOObjectStoreCoordinator. You can subsequently message every instance of an EOEditingContext to determine if a particular snapshot should be released. If you did not periodically garbage collect snapshots, an application which runs in a 24x7 environment for long periods of time without restarting could conceivably suck the entire database into memory.

This problem must be solved for WebObjects applications by use of a poser, since each session instance automatically instantiates an EOEditingContext for use as the defaultEditingContext upon initialization. There is no way to indicate that the Session should instantiate a subclass of EOEditingContext instead of the usual EOEditingContext. The poser registers each editing context with the EOObjectStoreCoordinator. A category method on the EOObjectStoreCoordinator completes the solution by traversing the list of editingContexts and cleaning up any unused snapshots.

WebObjects applications should periodically call this category method to enforce deallocation of unused snapshots. How often is periodically? The answer depends on how extensively your application fetches information at the session level. In applications that have extensive database access, calling pruneObjectGraph after every ten or twenty new requests to the site is prudent. For applications with more minimal database access, calling pruneObjectGraph after every fifty sessions or so may be acceptable. You can determine the appropriate frequency for your application by monitoring the memory used by your application over time, and analyzing how those memory requirements are reduced after calling the pruneObjectGraph method. (Special thanks to Craig Federighi for helping to architect this solution).

Here is the interface file for our poser. We define the poser the same way as any other subclass of EOEditingContext.

```
#import <Foundation/NSObject.h>
#import <Foundation/NSValue.h>
#import <EOControl/EOEditingContext.h>
@interface EOEditingContextPoser: EOEditingContext
{      }
@end
```

Here is the implementation file for our poser.

```
#import <Foundation/NSArray.h>
#import <Foundation/NSDictionary.h>
#import <EOControl/EOEditingContext.h>
#import <EOControl/EOObjectStore.h>
#import <EOControl/EOObjectStoreCoordinator.h>

// Import the interface file for our poser - contained in a
// framework called EOControlExtensions

#import <EOControlExtensions/EOEditingContextPoser.h>
@implementation EOEditingContextPoser

// Enhance the behavior of the initWithParentObjectStore: method
// Get the default EOObjectStoreCoordinator, pull our list of editing
// contexts out of its userInfo (creating a list if none is present),
// then add ourself to the list hidden inside a non-retaining NSValue
// instance.  When the NSValue object is added to the array, its retain
// count will be incremented, but self's will not.

- initWithParentObjectStore: (EOObjectStore *) anObjectStore
{
   if ( self = [ super initWithParentObjectStore: anObjectStore ] )
   {
      EOObjectStoreCoordinator *defaultCoordinator =
         [ EOObjectStoreCoordinator defaultCoordinator ] ;
      NSMutableDictionary *userInfo =
         (NSMutableDictionary *)[ defaultCoordinator userInfo ] ;
      NSMutableArray *editingContextList = [ userInfo objectForKey:
         @"editingContextList" ] ;
      NSValue *nonretainedSelf = [ NSValue valueWithNonretainedObject:
         self ] ;
      if ( userInfo == nil )
      {
         userInfo = [ NSMutableDictionary dictionaryWithCapacity: 1 ] ;
         [ defaultCoordinator setUserInfo: userInfo ] ;
      }
      if ( editingContextList == nil )
      {
         editingContextList = [ NSMutableArray array ] ;
         [ userInfo setObject: editingContextList forKey:
            @"editingContextList" ] ;
      }
      [ editingContextList addObject: nonretainedSelf ] ;
   }
   return self ;
}
@end
```

As stated above, the poser itself is not a complete solution for this problem. The poser registers each editing context with the EOObjectStoreCoordinator. A category method on EOObjectStoreCoordinator completes the solution by traversing the list of editing contexts and database contexts, and releasing any unused snapshots.

The interface file is:

```
#import <EOControl/EOObjectStoreCoordinator.h>

@interface EOObjectStoreCoordinator (EOControlExtensions)

-  (void) pruneObjectGraph ;

@end
```

The implementation file is:

```
#import <Foundation/NSSet.h>
#import <Foundation/NSValue.h>
#import <EOAccess/EODatabase.h>
#import <EOAccess/EODatabaseContext.h>
#import <EOControl/EOEditingContext.h>
#import <EOControl/EOFault.h>
#import "EOObjectStoreCoordinatorCategories.h"

@implementation EOObjectStoreCoordinator (EOControlExtensions)

// Traverse the contents of the list of editing contexts built
// by the poser to build an NSSet of unused snapshots ( use [ ec
// registeredObjects ] in combination with [ EOFault isFault: ] and
// [ ec globalIDForObject: ]). Now that you have the set, walk through
// theEODatabase instances and get the list of all globalIDs (call
// [ [ db snapshots ] allKeys ] to get the list).  Subtract the sets
// to get the list of snapshots to be invalidated, and call
// [ objectStoreCoordinator invalidateObjectsWithGlobalIDs: ].

-  (void) pruneObjectGraph
{
   NSMutableArray *editingContextList = [ [ self userInfo ]
      objectForKey:  @"editingContextList" ] ;

   if ( [ editingContextList count ] > 0 )
   {
      NSMutableSet *ecGIDs = [ NSMutableSet set ] ,
                   *dbGIDs = [ NSMutableSet set ] ;
      NSArray *coopObStores = [ self cooperatingObjectStores ] ;
      NSArray *allObjects ;
      int i,
      numEC = [ editingContextList count ],
      numCOS = [ [ self cooperatingObjectStores ] count ],
      j, numObjects ;
      EOGlobalID *gid;
```

```
// We first must make an NSSet containing all the GIDs
// referenced by all the non-EOFault objects in all
// the editingContexts.

for ( i=0; i<numEC; i++)
{
   NSValue *aValue ;
   EOEditingContext *ec ;
   id anObject ;

   aValue = [ editingContextList objectAtIndex: I ] ;
   ec = (EOEditingContext *)[ aValue nonretainedObjectValue ] ;
   allObjects = [ ec registeredObjects ] ;
   numObjects = [ allObjects count ] ;

   for ( j=0; j<numObjects; j++ )
   {
       anObject = [ allObjects objectAtIndex: j ] ;
       if ( ![ EOFault isFault: anObject ] )
       {
          gid = [ ec globalIDForObject: anObject ] ;
          if ( ![ ecGIDs containsObject: gid ] )
             [ ecGIDs addObject: gid ] ;
       }
   }
}

// At this point, the ecGIDs array contains all the global
// Ids referenced by all the non-EOFault objects in all the
// editing contexts. The next step is to build a similar list
// for all the GIDs of all the snapshots in all the EODatabases.

for ( i=0; i<numCOS; i++ )
{
   EOCooperatingObjectStore *aCOS = [ coopObStores
      objectAtIndex: I ] ;
   if ( [ aCOS respondsToSelector: @selector( database ) ] )
   {
      EODatabase *db = [ (EODatabaseContext *)aCOS database ] ;

      allObjects = [ [ db snapshots ] allKeys ] ;
      numObjects = [ allObjects count ] ;

      for ( j=0; j<numObjects; j++ )
      {
         gid = [ allObjects objectAtIndex: j ] ;
         if ( ![ dbGIDs containsObject: gid ] )
            [ dbGIDs addObject: gid ] ;
      }
   }
}

// Now we have an NSSet with all the GIDs referenced by all the
// editingContexts (ecGIDs) and an NSSet with all the GIDs
// referenced by all the databases (dbGIDs). Subtract the ec list
// from the db list and we have a list of GIDs that are not
// "needed" by any object in any ec.

[ dbGIDs minusSet: ecGIDs ] ;
```

```
        // Use the invalidateObjectsWithGlobalIDs: method to
        // dump all the referenced snapshots and refault any
        // EOs that depended on the snapshots.  Since there
        // are no EOs depending on the snapshots, this has
        // the effect of purging all the unneeded snapshots
        // from memory.

         [ self invalidateObjectsWithGlobalIDs: [ dbGIDs allObjects ] ] ;

        }
    }

@end
```

Interactive Media Resources Include:

- **Interactive Media Guidebooks**
- **Market Research Reports**
- **Survival Guides—Technical "How To" Guides**
- **Comarketing Opportunities**
- **Special Discounts**

Apple Developer Connection

As Apple technologies such as QuickTime, ColorSync, and AppleScript continue to expand Macintosh as the tool of choice for content creators and interactive media authors, the Apple Developer Connection continues its commitment to provide creative professionals with the latest technical and marketing information and tools.

Interactive Media Resources

Whether looking for technical guides from industry experts or for market and industry research reports to help make critical business decisions, you'll find them on the Interactive Media Resources page.

Apple Developer Connection Programs and Products

ADC programs and products offer easy access to technical and business resources for anyone interested in developing for Apple platforms worldwide. Apple offers three levels of program participation serving developer needs .

Membership Programs

Online Program—Developers gain access to the latest technical documentation for Apple technologies as well as business resources and information through the Apple Developer Connection web site.

Select Program—Offers developers the convenience of technical and business information and resources on monthly CDs, provides access to prerelease software, and bundles two technical support incidents.

Premier Program—Meets the needs of developers who desire the most complete suite of products and services from Apple, including eight technical support incidents and discounts on Apple hardware.

Standalone Products

Apple offers many standalone products that allow developers to choose their own level of support from Apple or enhance their Select or Premier Program membership. Choose from the following products and begin enjoying the benefits today.

Developer Connection Mailing—Subscribe to the Apple Developer Connection Mailing for the latest in development tools, system software, and more.

Technical Support—Purchase technical support and work directly with Apple's Worldwide Developer Technical Support engineers.

Apple Developer Connection News—Stay connected to Apple and developer-specific news by subscribing to our free weekly e-mail newsletter, *Apple Developer Connection News*. Each newsletter contains up-to-date information on topics such as Mac OS, Interactive Media, Hardware, Apple News and Comarketing Opportunities.

Macintosh Products Guide

The most complete guide for Macintosh products! Be sure to list your hardware and software products in our *free* online database!

http://developer.apple.com/media

The ultimate source for creative professionals.