TheModel-View-ControllerParadigm;¬The Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). MVC derives from Smalltalk-80; it proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.

Model Objects

This type of object represents special knowledge and expertise. Model objects hold a company's data and define the logic that manipulates that data. For example, a Customer object, common in business applications, is a Model object. It holds data describing the salient facts of a customer and has access to algorithms that access and calculate new data from those facts. A more specialized Model class might be one in a meteorological system called Front; objects of this class would contain the data and intelligence to represent weather fronts. Model objects are not displayable. They often are reusable, distributed, and portable to a variety of platforms.

View Objects

A View object in the paradigm represents something visible on the user interface (a window, for example, or a button). A View object is ªignorantº of the data it displays. The Application Kit usually provides all the View objects you need: windows, text fields, scroll views, buttons, browsers, and so on. But you might want to create your own View objects to show or represent your data in a novel way (for example, a graph view). View objects, especially those in kits, tend to be very reusable and so provide consistency between applications.

Controller Object

Acting as a mediator between Model objects and View objects in an application is a Controller object. There is usually one per application or window. A Controller object communicates data back and forth between the Model objects and the View objects. It also performs all the application-specific chores, such as loading nib files and acting as window and application delegate. Since what a Controller does is very specific to an

application, it is generally not reusable even though it often comprises much of an application's code.

Because of the Controller's central, mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of the Model objects. You can make your View and Model objects available to others from a palette in Interface Builder.

Hybrid Models

MVC, strictly observed, is not advisable in all circumstances. Sometimes its best to combine roles. For instance, in a graphics-intensive application, such as an arcade game, you might have several View objects that merge the roles of View and Model. In some applications, especially simple ones, you can combine the roles of Controller and Model; these objects join the special data structures and logic of Model objects with the Controller's hooks to the interface.

225866_TableRule.eps ¬

APerspectiveontheClassHierarchy;¬A Perspective on the Class Hierarchy

The Classes display of the nib file window shows the classes that the current nib file is aware of. The display lets you browse through both OpenStep classes and custom classes. The Classes display also depicts (by indentation) class-inheritance relationships and reveals the names of each class's outlets and actions.

Keyboard Navigation Move up and down in the list of classes pressing the up arrow and the down arrow. When a class is highlighted, show its subclasses by pressing the right arrow; collapse an indented list by selecting the superclass and pressing the left arrow. If the nib file window is active, incremental search is active: just type the first few letters of a class until its name is highlighted.

_PerspectiveClassHierarchy.eps ¬

225866_TableRule.eps ¬

AShortPracticalGuidetoSubclassing;¬A Short Practical Guide to Subclassing

Subclassing is not an esoteric art but one of the most common and essential tasks in object-oriented programming. But it doesn't need to be a difficult chore, especially if you take the time to learn what's in the class hierarchy.

What is Subclassing?

The principal notion behind subclassing is inheritance. Classes stand in relation to other classes as child to parent or parent to child. A class might have many child classes (or subclasses), but always has only one parent class (superclass). At the head of this class hierarchy is the root class.

Subclassing.eps ¬

The attributes (instance variables) and behavior (methods) defined by a class are shared by all descendents of that class. To put it another way, each new class is the accumulation of all class definitions in its inheritance chain.

For example, the NSView class defines two instance variables for location and size (**frame** for the superview orientation, and **bounds** for within the view) from which all instances of its numerous subclasses derive their own basic position and dimensions. The NSView class also defines several methods for setting and getting these instance variables; again, all subclasses of NSView inherit the behavior defined by these methods. You can send the same messages to any instance of an NSView subclass to have it resize itself.

So subclassing is usually the extension and specialization of the inheritance chain. When you define a class that inherits from another class, you are specifying how it differs from that superclass.

But there are reasons for creating a subclassÐor a ªbranchº of subclassesÐother than getting different behavior. You may want to define a class that dispenses generic functionality to its subclasses, such as an Output class that performs tasks common to both a Printer class and a Fax class. You might want a class to declare methods (perhaps unimplemented) that set up a protocol that future subclasses can implement. Code reusability is an additional motive: the behavioral elements shared among classes can go into a single superclass for those classes.

Analyzing the Inheritance Chain

As the first step in subclassing you should analyze the inheritance chain. This point may seem obvious, but it is important enough to emphasize. You should do more than just identify the most suitable superclass; you

want to understand exactly what it does and how it interacts with other classes.

Carefully read the specifications. Note which methods are available. Determine what the methods do and how they are related to each other; identify the accessor methods, those that get and set the instance variables; identify the interfaces to instances of other classes (such as outlets).

If the behavior you want for your class is targeted at a special problem, even if that problem is managing an application or window, it might make the most sense to subclass NSObject. These kind of subclasses, often called controller or model classes, are common in OpenStep applications. See ªImplementing a subclass of NSObjectº for details on creating typical controller classes. Also, see ªThe Model-View-Controller Paradigmº in this chapter for a description of the distinguishing characteristics of controller and model types of classes.

Instance Variables: To Add or Not to Add

Instance variables represent an object's attributes and hold pointers to other objects (outlets). If instances of your class require special attributes or outlets, add them.

But, as a general rule, avoid adding instance variables unless they are absolutely necessary. Instance variables add weight to objects. You sometimes generate certain objects (for example, cells in a file-system browser) in large numbers. The more data these objects carry, the more memory gets consumed.

Often you can compute values from other values. Sometimes you can get pointers to other objects without having to specify outlets. Or you can represent attributes in lightweight fashion, especially if they are Boolean in nature, by encoding them as bits in an integer.

If you do not want to give subclasses of your class access to its instance variables, put the @**private** directive before the declarations of the instance variables you want to conceal. (Many instance variables are private in OpenStep classes.)

Determining Your Class's Methods

Look at your class from the perspective of potential clients. What will they want it to do? What information will they expect back? The answers to these and similar questions will lead to the set of methods for your class.

Based on relation to superclass, methods generally come in three types:

- **Added methods** These new methods extend the class definition. They include accessor methods for new instance variables.
- **Replacement methods** These types of methods completely override the superclass method of the same name. They can also, by being a ªnullº implementation, block the invocation of the superclass method.
- **Extended methods** These methods also override a superclass method, but then in the implementation invoke the superclass method by calling **super**. This is a common technique for adding behavior or getting cumulative behavior (such as archiving) across the inheritance chain in response to a single message (such as **encodeWithCoder:**).

What is Public, What is Private?

When designing your subclass, also identify the code that is part of the interface and code that is private to the class.

- **Public methods** These implement your class's interface. External objects invoke these methods by sending messages to instances of your class. Among these types of methods are accessor methods, which mediate client access to instance variables. You declare public methods in the header file for your class.
- **Private methods** These methods can be invoked by objects within a project but are invisible to external objects. You usually declare them in a private header file and prefix the method name with an underscore character.
- **Functions** Non-library static C functions are also private to your class. They are marginally faster than methods because they don't involve the overhead of the run-time object system.

Use a method if you're accessing instance variables, and use a public method if that method is part of your public interface.

This example illustrates the effects of polymorphism and inheritance in a hypothetical class hierarchy. The Shape class provides

basic functionality and a single instance variable. The Circle class, a subclass of Shape, adds more instance data and actually implements drawing. The Crescent class supplements its superclass (Circle) with more specialized behavior and data.

InstanceVariable.eps ¬

Alternatives to Subclassing

Sometimes you can get particular behavior without additional subclassing. OpenStep and the Objective-C language give you many ways to merge and synchronize your class's behavior with the behavior of OpenStep classes and even other custom classes.

- **Delegation** An object can send, on specific occasions, messages to another object registered as its delegate. If the delegate implements the methods so invoked, it can participate in the work of the object. For example, an NSBrowser object sends messages to its delegate requesting cells to insert into a column. Other major Application Kit classes with delegation protocols are NSApplication, NSWindow, and NSText.
- **Notifications** Many objects post notifications to all interested observers when a particular event takes place or is about to take place. Notifications allow observing objects to coordinate related activities and sometimes give them a chance to veto the event. This can be better than delegation because an object can have many observers but only one delegate. See the specification for NSNotificationCenter (a Foundation Framework class) for details on adding an observer object and on responding to notifications.
- **Protocols** A protocol is a list of method declarations associated with a particular purpose but unattached to a class definition. By adopting the protocol and implementing the methods, your class can interact with OpenStep classes and accomplish that purpose. OpenStep publishes many protocols, including those for copying objects and encoding objects for archiving.
- **Categories** These are Objective-C constructs that enable you to add methods to a class without having to subclass it. The methods become part of the class, inherited by all of its subclasses. The only major drawback is that you cannot declare new instance variables (however, you can access all existing instance variables). Besides extending a class definition, you use categories to group, manage, and configure methods in large classes.

225866_TableRule.eps ¬

OtherNSObjectMethodsYouCouldOverride;¬Other NSObject Methods You Could Override

There are several other NSObject methods that you might want to implement:

description Implement this method to return a descriptive debugging message as an NSString object. When you're debugging, **gdb** displays your message when you use the **po** command.

awakeAfterUsingCoder: Implement this method to re-initialize the object, providing it one last chance to propose another object in its place.

replacementObjectForCoder: Implement this method to substitute another object for your object during encoding.

initialize Implement this class method if you want to initialize your class before it receives its first message. This is a good place to set the version of your class (**setVersion:**).

forwardInvocation: Implement this method if you want to forward messages with unrecognized selectors to another object that can handle the message.

225866_TableRule.eps ¬

TheStructureofHeaderFilesandImplementationFiles;¬The Structure of Header Files and Implementation Files

Header File

```
#import <AppKit/AppKit.h>

@interface TAController:NSObject
{
```

```
    id  tableView;
    ...
    BOOL recordNeedsSaving;
}

/* target/action */
- (void)addRecord:(id)sender;
- (void)deleteRecord:(id)sender;
/* housekeeping methods */
- (id)init;
- (void)awakeFromNib;
- (void)dealloc;
...
@end
```

- Begin by importing header files for declaration types (**#import**).
- **@interface** begins class interface declaration. Class name precedes superclass, separated by a colon.
- Put the declarations of instance variables within curly braces.
- After right curly brace declare your methods.
- Action methods take the argument **sender**.
- End class interface declaration with **@end**.

Implementation File

```
#import "TAController.h"

@implementation TAController
```

```
- (void)addRecord:(id)sender
{
     /* some code here */
     return;
}

- (id)init
{
     /* some code here */
     return self;
}

/* ... */
@end
```

- Begin by importing relevant header files, especially the class header file.
- @**implementation** followed by class name begins implementation section.
- Implement all methods.
- End implementation section with @**end**.

225866_TableRule.eps ¬

CreatingandDeallocatingDifferentTypesofObjects;¬Creating and Deallocating Different Types of Objects

As you create objects, you need to make sure that they are going to be deallocated eventually, and you also need to make this doesn't happen until you don't need the object anymore. You do this by sending messages that increment and decrement the object's reference count, a count of how many objects refer to it. When and how you should do this depends on when and how you create the object.

The Autorelease Pool

OpenStep uses an autorelease pool to automatically deallocate objects. When you send an **autorelease** message to an object, it adds the object to the autorelease pool. At the top of the event loop, the pool sends every object in it the **release** message. **release** decrements the reference count. If the reference count becomes 0, it deallocates the object (by sending **dealloc**).

Application projects automatically have an autorelease pool, just as they automatically have an event loop. If you're working on a non-Application project, you can create an autorelease pool by creating an instance of the Foundation NSAutoreleasePool class. (See its specification in the *Foundation Framework Reference*.)

Temporary Objects

If you create an object inside a method and you want that object to go away after the method has finished executing, use a ***+classname*** method (so called because their names begin with the name of the class minus the NS prefix) to create the object. These methods allocate the object (which increments the reference count), initialize it, and send it an **autorelease** message so that it is deallocated at the top of the event loop. For example, this NSNumber object will exist only for one event cycle:

```
NSNumber *intObject = [NSNumber numberWithInt:anInt];
```

The methods **alloc**, **copy**, and **mutableCopy** increment an object's reference count, so if you use one of these to create a temporary object, be sure to send that same object an **autorelease** message.

Instance Variables

Objects that are instance variables should be created when an object is initialized and not go away until that object is deallocated. If you use a ***+classname*** method to create an instance variable, it will be deallocated at the top of the event loop. To prevent this, send **retain** to the object immediately after you create it. **retain** increments the reference count. Another way to make sure that an instance variable is not deallocated is to use the **alloc** method directly (or **copy** or **mutableCopy**) to create it.

No matter which method you use to create the instance variable, send it a **release** message in your object's **dealloc** method to indicate that you're done with it.

Sometimes you have two objects with instance variables that refer to each other. In this case, only one of the objects should retain the other. For example, an NSView object has a superview and one or more subviews, each pointing to other NSView objects. If an NSView object retained both its superview and its subviews, no NSView would ever be deallocated. The superview won't release its subview instance variables until it is deallocated, and it can't be deallocated because the subviews don't release the superview until they are deallocated. For this reason, NSView objects retain their subviews, but not their superviews.

As a rule of thumb, if your application has a similar object hierarchy, the ªparentº object should retain its ªchildren,º but the children should not retain their parents.

Custom Objects Created in Interface Builder

If you create a custom object that does not inherit from NSView or NSWindow in Interface Builder, send it a **release** message in your object's **dealloc** method. Custom objects have a retain count of 1 when they're unarchived from the nib file.

NSView Objects Created in Interface Builder

Views created in Interface Builder are retained and released automatically. Superviews retain all subviews as they are added to the hierarchy and release them as they are removed. If you swap views in and out of the hierarchy or move views from one window to another, you should **retain** the views that are not in the hierarchy (and **release** them either after you add them to the hierarchy or in **dealloc**).

NSWindow Objects Created in Interface Builder

Windows created in Interface Builder are not released until the user quits the application. If you want a window to be released when the user closes it, set the ªRelease when closedº attribute in Interface Builder.

For more on this topic, see the introduction to the *Foundation Framework Reference*.