Chapter 4 Encapsulation and Inheritance

We have just taken you on a whirl-wind tour of creating your first object. We will now take a step back and take a look at the details of object creation.

The late 1980's saw the migration of programs traditionally found on large mainframe systems to personal computer platforms - word processors, data bases and spread sheets being the most popular. As these programs became more popular, new software companies began entering the PC software market with less expensive and more powerful versions of the standard programs. To be competitive software had to be feature-rich and still run on systems with severe memory limitations. As a result, the marketing divisions of software firms started to promise new versions of products with a large number of new features but which would still run on computer systems with very little memory. Software developers found that they couldn't deliver the programs on time. The software had so many features that it was called "bigware" and it took so long to develop that it was later known as "lateware".[1]

What software developers were finding was that as they tried to add new features they would introduce new bugs. And trying to fix those bugs introduced additional bugs. The result was a drastic decline in software productivity as the the size of a project grew (see Figure 4-1)

[1]Newsweek April 1989

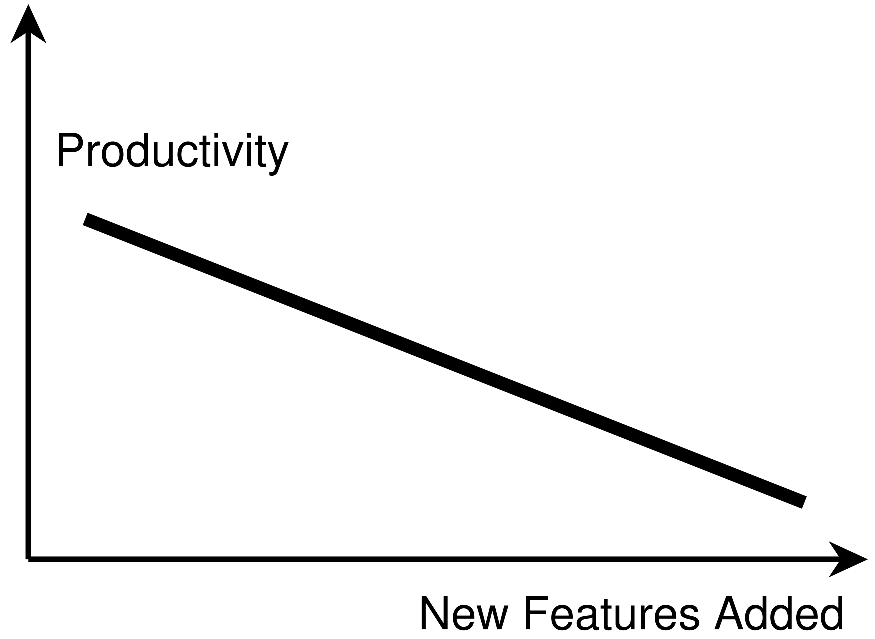

Figure 4-1 Dropping Productivity as Project Grows

To solve this problem we must look at the ways we create programs and integrate them together to create a software system. We must not look at the superficial features of software productivity tools but at the very foundations of the way we create software.

Researchers have been studying software design techniques for many years. Much of the pioneering research done at Xerox Palo Alto Research Center with totally new programming environments such as SmallTalk has shown that there are radically different approaches to creating software that challenge traditional methods currently being taught. One family of techniques gained a great deal of popularity in the mid-eighties are now currently referred to as the principals of object oriented programming. We will introduce these techniques to you in this and later chapters. Our first techniques will cover the creation of new objects.

Encapsulation

Encapsulation has various names. Computer scientists often refer to it as "information hiding" or "data abstraction". In general, these terms imply grouping data and the procedures to access that data together in the same unit. We call this unit an object. We also set down some rules about how you can access the data in the object. One rule is that if we develop and object, you **only** let people see or change your internal data using the procedures we provide with that object. This means the creator of the object has the ultimate control of how users access this object and change the internal states of the object. The creator of an object is responsible for creating and testing all of the ways a user will read the state of on object as well as checking that inputs to change the state of an object are valid. The creator of an object (rather then the user of an object) has complete responsibility of completeness and correctness of all access methods.

When we try to create a mental image of the objects, we might find it helpful to imagine a box with a thick brick wall around it as in Figure 4-2.

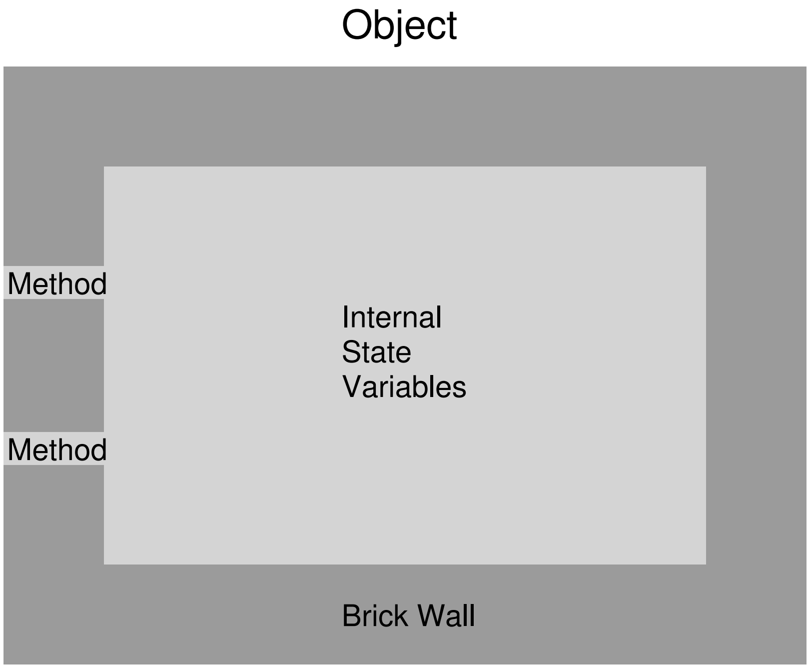

Figure 4-2: A suggested mental image of an object.

Inside the box we have the data structures such as integers, floating point numbers, strings, and other more complicated structures such as linked lists or directed graphs. We call these the **instance variables**. They hold the state of the object. We use the word instance because there is a different group of these variables associated with each instance of the object. The only way to read or write the values of these variables is to use one of the **access methods** provided with each object. We often just abbreviate this by just using the word "method" to refer to the way a program gets access to the structures inside an object.

Benefits of Encapsulation

Once we start encapsulating your data you will find that you can quickly control the data types that are passed to your objects. Since the NextStep Objective C compiler has type checking built into its messaging you will always be able to catch data type mismatches at early in the design process when it is much easier to isolate. This will dramatically cut down time spent with the debugger and greatly enhance the programs final reliability.

Once we define the set of messages that an object can receive we are then fixing the interface to that object. If, at a later time, you find another more efficient data structure we would like to use inside the object we can change it internally and not effect the interface. This means that we can make updates without affecting the other components of our system.

After we have an object that performs some specific function, we can then create a symbolic abstraction of that object using a "view" of it on the screen. The connections to the object can then be done with NextStep's connection based programming tools.

Views of Subroutine Libraries

By creating these views, a user can now graphically manipulate the object and integrate it with other objects. When users make a connection to one of our objects, NextStep will ask them which of the access methods they would like to use. This means that non-programmers can start using tools that were previously only accessible to a very small group of experienced people. And since it is up to the creators of the objects to validate the correctness of the access methods, a much larger group of people will be able to use the objects without the traditional debugging efforts.

Imagine what the world would be like if the only people who could drive a car were the people who could assemble an internal combustion engine. We certainly wouldn't have the traffic problems we have today. But what we have in cars is a simplified user interface: a steering wheel, a break and a gas peddle. object based computing platforms give us these same advantages: easy to use interfaces to traditional subroutine libraries. This helps both the creators and users of a sub-routine library system. It helps the creators because their potential user base goes up dramatically. This help users because the amount of training they have to go through drops dramatically. We will see that the number of people who are creating applications with these graphic subroutine libraries will grow exponentially for the next several years.

Before object based computing we had to use a manual to find out all the arguments to a subroutine, declare all the arguments with the correct data types, pass these in the correct order to subroutines and then if you get any of them wrong, start a learning how to use the debugger. Now we will just point to a source object, drag a line to a destination object and click on the message to be sent. Correctness is enforced by the user interface.

Inheritance

Before we discuss inheritance, I want to make the distinction between a instance of an object and a **class** of objects. The characteristics of class of objects, like a Ford Truck, is determined by the factory which creates the trucks. If I had a Ford Truck, I would have an instance of the truck. Similarly we have class of objects which create new instances of objects. And these are naturally called **factory objects**.

Our second technique is Inheritance. Whenever we create a new **class** of objects, we always create it relative to other objects classes. These classes then fit together into a "tree" of object classes.

Sample Inheritance Tree

The structure is very similar to an evolutionary tree (see figure 4-3).

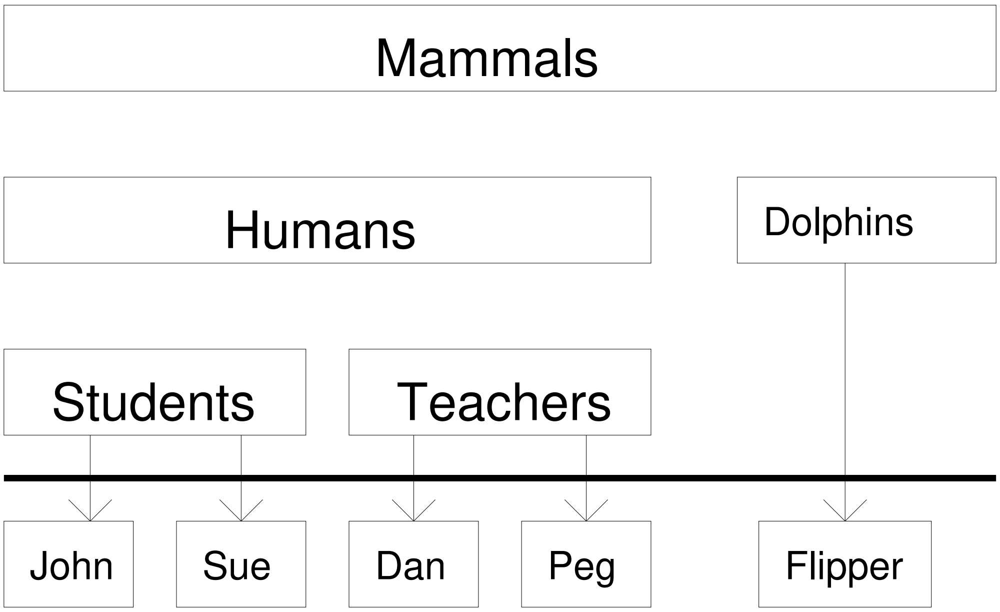

Figure 4-3: A sample inheritance tree

The most general class is at the top, and each class that has a group of common characteristics would be a lower class. An important point here is that you can create an instance from any level in the tree. And when you think of the difference between an **instance** of an object and a **class** of object, remember that are as different as a car and a factory that produces cars. This can be difficult for beginners using Interface Builder because both classes and instances are represented by small windows that are very close together and look similar. Both classes and instances of objects can be changed. Changing the factory that produces cars could change some aspect of every car produced by that factory. But changing one single car would not have a direct effect on all other cars of its class.

We will also use some new terminology to describe the relationship between classes. In the example above, we would say that Humans are a **super-class** of Students and that Humans are a **sub-class** of mammals.

When we create a new class, we inherit all the instance variables as well as methods defined in its super-class. It in turn inherits all the instances and methods of all the super-classes above it. This is one of the principal ways that we re-use programs in object oriented programming. We find the object in an inheritance tree that most closely matches our problem and make extensions to it from there. There are ways to create new structures as well as change existing structures to meet our needs.

Let us now take a close look at the inheritance tree for the NextStep Application Kit. This is a tool-kit of objects that we can use to build applications. The structure for this tree is given is figure 4-4.

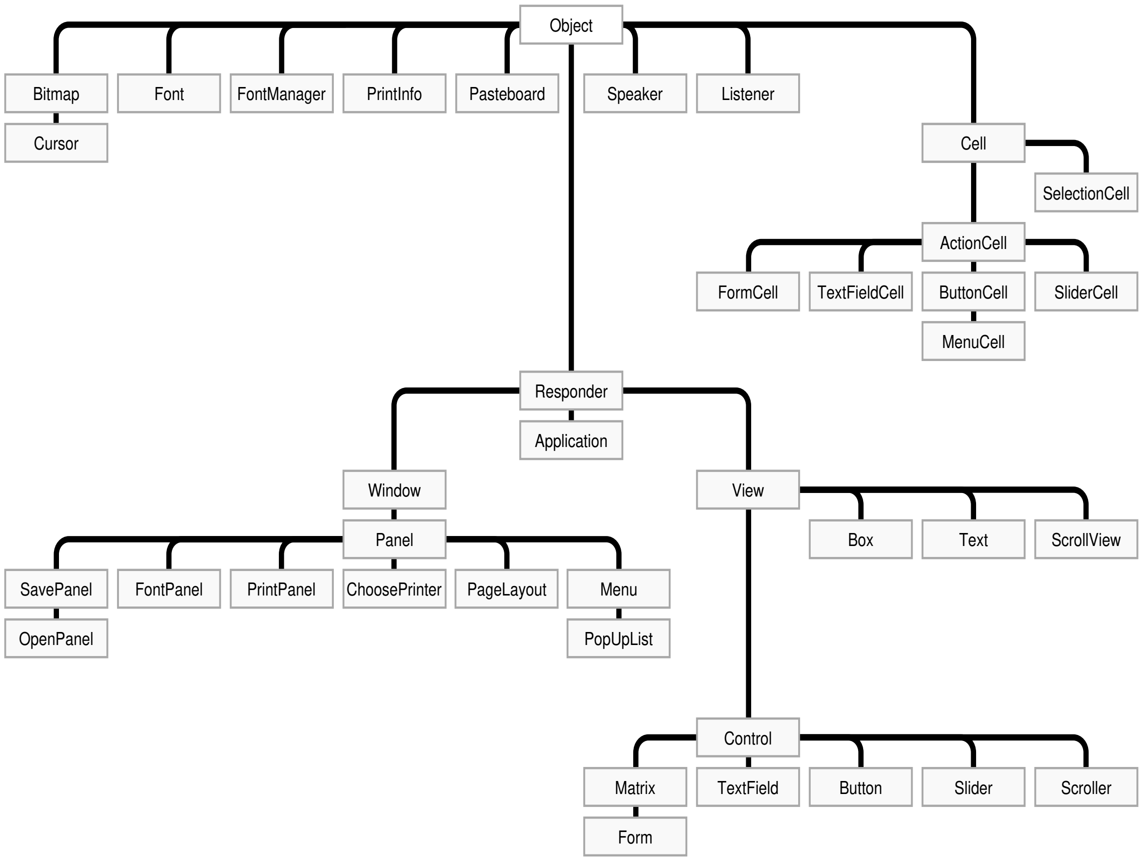

Figure 4-4: Application Kit Inheritance Hierarchy

Understanding the structure of the application kit is necessary if we are to be able to use the application kit and extend the user interface objects to meet our needs.

At the top of the structure we see a box titled "Object". This is the most generic object in the tree. It has the fewest specialized characteristics of any of the Application Kit objects. Any characteristic of the object class will be shared with all other appkit objects. Directly below the object class is the Responder class. This consists of all objects that can respond to user generated events such as pressing the mouse and typing on the keyboard. To the lower right of the Responder is the View class. All objects that are on the screen are a subclass of the View class. Below the View class is the Control class. All classes which are sub-classes of the Control will respond to events by sending message directly to other objects. They can serve as controller inputs to our custom objects. One example of a control is the Button class. If we take a closer look at the button class we will see that most of the characteristics of the Button class are not created in the object itself but arise from its location in the inheritance tree. Lets take a closer look at the documentation NeXT provides about the Button class. It can be found in the following path of the NeXT on-line documentation (see figure 5)

(from /NextLibrary/Documentation/NeXT/SysRefMan/21ClassSpecs/Appkit/Button.wn)

Button

INHERITS FROM	Control : View : Responder : Object

INSTANCE VARIABLES

Inherited from Object	struct _SHARED
*isa;	
Inherited from Responder	id
nextResponder;	
Inherited from View	NXRect
frame;	NXRect
bounds;	id
superview;	id
subviews;	id
window;	struct __vFlags
vFlags;	
Inherited from Control	int
tag;	id
cell;	struct _conFlags
conFlags;	
Declared in Button	(none)

Figure 4-5: Class specification for the Button Class

The first line of the specification sheet reads:

INHERITS FROM	Control : View : Responder : Object

This shows you the direct ancestors of the Button class. Below that we have the instance variables for each of these levels. The mental picture we create for the Button class might look something like figure 6.

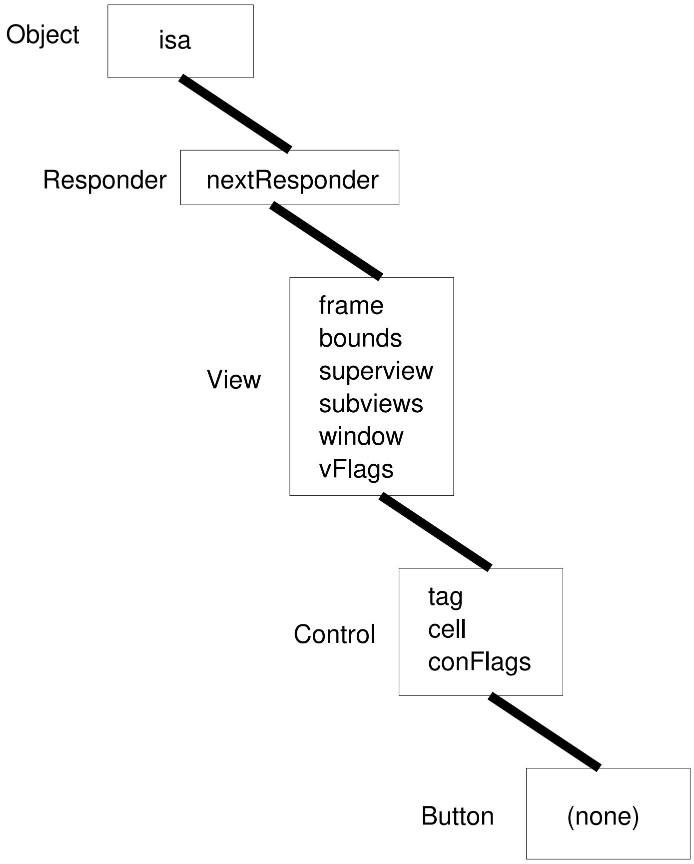

Figure 4-6: Button Inheritance Hierarchy

If we create a button of our own that needed to know the bounds of the button, we would access the bounds in our own button as the variable "bounds". The bounds would actually come from the state variable of the View class. Information such as bounds is used over and over again every time we manipulate any on-screen objects. By re-using this code we increase our productivity and encourage re-use rather then re-invention.

Exercises

1. Bring up Interface Builder and double click the Class Editor window by double-clicking on the "Classes" tool-kit icon in the lower left window. Compare this with the Inheritance structure in Figure 4-5. What happens when you try to use the "Instantiate" selection under the pop-up list labeled "Operations". Try instantiating an Object. Why can't you instantiate a View? Try sub-classing a View and call it MyView. If you drag a customView from the palette and then inspect its attributes what do you see? Can you make the custom view an instance of the MyView class.

2. What are some sample inheritance structures you might create? What would an inheritance structure for banking objects such as saving accounts, checking accounts, and banking transactions look like? What fields would they have in common? What fields would be unique? What would an inheritance structure for electrical components like wires, resistors, capacitors and transistors look like?